Devoir surveillé #6 (avec solutions)

Lundi 2026-06-01 ; durée : quatre heures

Compression entropique

Sujet repris de l'épreuve d'informatique A du concours X-ENS 2023. Les principales modifications au sujet original sont l'ajout de quelques fonctions à l'aide à la programmation OCaml ainsi que de quelques indications dans les quatre premières parties.
Merci à Galatée Hémery pour des versions TikZ des figures du sujet original.

Préliminaires

Ce sujet s'intéresse à la compression et à la décompression de mots sur un alphabet fini $\mathcal{S}$ de cardinal $|\mathcal{S}| \geq 2$. Plus précisément, on se donne une fonction q de $\mathcal{S}$ dans $\mathbb{N}$ et on s'intéresse aux mots (a priori *non vides*) de $\mathcal{S}^*$ tels que chaque lettre $\sigma \in \mathcal{S}$ a exactement $q(\sigma)$ occurrences dans ces mots. On note $\mathcal{S}^q$ l'ensemble de ces mots. Remarque : les mots de $\mathcal{S}^q$ ont pour longueur $N = \sum_{\sigma\in\mathcal{S}} q(\sigma)$.
Une lettre de $\mathcal{S}$ peut être représentée avec $\lceil \log_2 |\mathcal{S}| \rceil$ bits, où $\lceil x \rceil$ désigne la partie entière supérieure de x, c'est-à-dire l'entier tel que $\lceil x \rceil - 1 < x \leq \lceil x \rceil$. Si l'on concatène les bits représentant chacune des lettres d'un mot de $\mathcal{S}^q$, on peut donc représenter ce mot de façon non ambiguë à l'aide d'une séquence de bits de taille :

$$\sum_{\sigma\in\mathcal{S}} q(\sigma) \times \lceil \log_2 |\mathcal{S}| \rceil = \lceil \log_2 |\mathcal{S}| \rceil \times N.$$

Il s'agit d'une représentation non compressée. La théorie de l'information affirme qu'il faut au moins $\sum_{\sigma\in\mathcal{S}} q(\sigma) \times \log_2(N/q(\sigma))$ bits pour représenter tous les mots de $\mathcal{S}^q$ de façon non ambiguë. Ce sujet explore quelques façons de compresser les mots pour s'approcher de cette borne théorique.

La partie I s'intéresse à la fonction q, c'est-à-dire au comptage des lettres d'un mot. La partie II étudie les arbres binaires dont les feuilles sont étiquetées par des lettres de $\mathcal{S}$. La partie III utilise ces arbres pour (dé)compresser des mots de $\mathcal{S}^*$. La partie IV s'intéresse aux arbres qui donnent les meilleurs taux de compression pour les mots de $\mathcal{S}^q$. Certains de ces arbres ont une représentation compacte ; c'est l'objet de la partie V. D'autres arbres sont encore plus compacts, mais au prix d'un moindre taux de compression, comme le montre la partie VI. Finalement, la partie VII attaque le problème d'une façon complètement différente afin de s'approcher un peu plus du taux de compression théorique optimal. Les différentes parties sont indépendantes : il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie ; cependant les notions abordées dans une partie sont souvent utiles aux parties suivantes.

Vous avez deux possibilités pour traiter ce sujet (chacune notée sur 20) :

— soit vous traitez **uniquement** les parties I à IV
— soit vous traitez l'intégralité du sujet

Dans les deux cas, vous devez indiquer clairement votre choix au début de votre copie.

OCaml

L'intégralité de ce sujet est à traiter en langage OCaml. On rappelle ici quelques fonctions de la bibliothèque standard :

— `Array.length t` renvoie la longueur du tableau `t`.
— `Array.make n v` crée un tableau de `n` cases qui sont toutes initialisées avec `v`.
— `Array.make_matrix n m v` crée un tableau à deux dimensions de `n` lignes et `m` colonnes dont toutes les cases sont initialisées avec `v`.

— `Array.of_list l` renvoie un tableau initialisé avec les valeurs de la liste `l`, tandis que `Array.to_list` réalise l'opération inverse.
— `Array.copy a` renvoie une copie du tableau `a`.
— La case numéro `i` du tableau `t` peut être accédée avec `t.(i)`. Les cases sont numérotées à partir de zéro.
— `List.rev l` renvoie le miroir de la liste `l` en argument, en temps linéaire en sa longueur.
— `List.map f x` renvoie la liste `[f x0; f x1 ; ...]` obtenue en appliquant la fonction `f` à chacun des éléments de la liste `x = [x0; x1; ...]`
— `failwith s` avec `s` une chaîne de caractères lève une exception `Failure s`.

I. Comptage d'occurrences

Un texte non compressé est un mot de $\mathcal{S}^*$. Dans la suite, à chaque fois qu'il s'agira de définir une fonction en OCaml, les lettres de $\mathcal{S}$ seront représentées par des entiers `int` et un mot de $\mathcal{S}^*$ sera représenté par un tableau d'entiers `int array` ou une liste d'entiers `int list` en fonction des questions. La première étape consiste à calculer le nombre d'occurrences $q(\sigma)$ de chaque lettre σ dans un mot.

1. Supposons $\mathcal{S} = [0, 255]$. Définissez une fonction OCaml `occurrences : int list -> int array` qui reçoit en argument un mot représenté par une liste d'entiers et renvoie le nombre d'occurrences $q(\sigma)$ de chaque lettre $\sigma \in \mathcal{S}$ sous forme d'un tableau $[\![q(0); q(1); \ldots; q(255)]\!]$.

On propose :
```
let occurrences w =
    let q = Array.make 256 0 in
    let rec loop
        = function
        | [] -> q
        | x::xs -> q.(x) <- q.(x) + 1 ; loop xs
    in loop w
```

Il peut être intéressant de ne considérer que le sous-ensemble des lettres qui ont au moins une occurrence. Dans la question suivante, on ne représentera pas q par $[\![q(0); q(1); \ldots; q(|\mathcal{S}|-1)]\!]$ mais par un tableau de paires $[\![(\sigma_1, q(\sigma_1)); (\sigma_2, q(\sigma_2)); \ldots]\!]$ avec $\{\sigma_1, \sigma_2, \ldots\} = \{\sigma \in \mathcal{S} \mid q(\sigma) \neq 0\}$ et $\sigma_1 < \sigma_2 < \ldots$

2. Définissez une fonction OCaml `nonzero_occurrences : int array -> (int * int) array` qui passe de la représentation de q de la question 1 (tableau $[\![q(0); q(1); \ldots; q(|\mathcal{S}|-1)]\!]$) à la représentation sous forme d'un tableau de paires. Cette fonction devra avoir une complexité temporelle en $O(|\mathcal{S}|)$.

On propose deux versions (parmi plein de possibilités). La seconde est sans doute la plus dans l'esprit de ce qui était attendu, étant donnés les rappels OCaml donnés en début du sujet :
```
let nonzero_occurrences q =
    let j = ref 0 in
    let q' = Array.make 256 (0, 0) in
    let () = Array.iteri (fun i x ->
                 if x > 0 then (
                     q'.(!j) <- (i, x) ;
                     incr j )
             )
             q in
    Array.init !j (fun i -> q'.(i))

let nonzero_occurrences' q =
    let q' = ref [] in
    let rec loop i =
        if i = 256 then !q' else (
        if q.(i) > 0 then
            q' := (i,q.(i))::!q'
        ; loop (i + 1))
```

```
    in
    loop 0 |> List.rev |> Array.of_list
```

II. Arbres binaires

Soit $\mathcal{T}_\mathcal{S}$ l'ensemble des arbres binaires dont les feuilles sont en bijection avec un ensemble fini $\mathcal{S}$. Autrement dit, un arbre de $\mathcal{T}_\mathcal{S}$ a exactement $|\mathcal{S}|$ feuilles ; chacune de ses feuilles est étiquetée par un élément de $\mathcal{S}$; toutes les feuilles sont étiquetées par des éléments différents de $\mathcal{S}$. Ces arbres peuvent être construits comme suit :

— Une feuille étiquetée par x est notée $F(x)$. C'est un arbre de $\mathcal{T}_{\{x\}}$.
— Si g et d sont des arbres appartenant respectivement à $\mathcal{T}_{\mathcal{S}_1}$ et $\mathcal{T}_{\mathcal{S}_2}$, alors l'arbre dont le sous-arbre gauche est g et le sous-arbre droit est d, noté $N(g,d)$, est un arbre de $\mathcal{T}_{\mathcal{S}_1\cup\mathcal{S}_2}$ à condition que $\mathcal{S}_1\cap\mathcal{S}_2=\emptyset$.

Remarque : un tel arbre binaire possède exactement $|\mathcal{S}|-1$ nœuds internes. Par ailleurs, on étiquettera implicitement les arêtes par 0 ou 1 suivant qu'elles mènent à un sous-arbre gauche (0) ou droit (1).

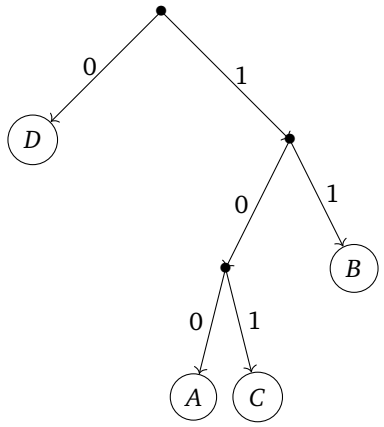

FIGURE 1 – Exemple d'arbre de $\mathcal{T}_{\{A,B,C,D\}}$

3. Montrez que l'ensemble $\mathcal{T}_{\{A,B,C,D\}}$ contient 120 arbres.

Un arbre de $\mathcal{T}_{\{A,B,C,D\}}$ est de la forme $N(g,d)$ avec g,d arbres de $\mathcal{S}_1\neq\emptyset\subset\{A,B,C,D\}$ et $\mathcal{S}_2=\{A,B,C,D\}\setminus\mathcal{S}_1$.
Soit C_i le nombre d'arbres de $\mathcal{T}_\mathcal{S}$ sur un ensemble $\mathcal{S}$ de taille n, on a alors que pour $\mathcal{S}_1$ comme ci-dessus de taille fixée à $1\leq k<n$ il y a C_kC_{n-k} tels arbres, et puisqu'il y a $\binom{n}{k}$ façons de choisir un tel ensemble $\mathcal{S}_1$, on obtient la récurrence :

$$C_n=\sum_{k=1}^{n-1}\binom{n}{k}C_kC_{n-k}$$

et $C_1=1$.
On peut alors calculer :

— $C_2=2$
— $C_3=\binom{3}{1}\times C_1C_2+\binom{3}{2}\times C_2C_1=12$
— $C_4=\binom{4}{1}\times C_1C_3+\binom{4}{2}C_2C_2+\binom{4}{3}C_3C_1=4\times12+6\times4+4\times12=120$

Remarque : on pouvait procéder autrement, le rapport du jury suggérant par exemple l'approche relativement informelle consistant à dessiner les cinq arbres de $3=4-1$ nœuds internes et de conclure avec le fait que chacun peut s'étiqueter de 4! façons.

Le type OCaml utilisé pour représenter les arbres est le suivant :
```
type tree = F of int | N of tree * tree
```

Étant donnés un arbre t de $\mathcal{T}_\mathcal{S}$ et un élément σ de $\mathcal{S}$, on note $\ell_t(\sigma)$ la profondeur de la feuille étiquetée par σ, c'est à dire le nombre d'arêtes entre la racine de t et cette feuille. Par exemple, avec l'arbre de la Figure 1, on a $\ell_t(A)=3$.

Comme précédemment, soit q une fonction de $\mathcal{S}$ dans $\mathbb{N}$. Pour tout arbre de $t \in \mathcal{T}_{\mathcal{S}'}$ avec $\mathcal{S}' \subseteq \mathcal{S}$, on note $c_q(t)$ la somme pondérée suivante :

$$c_q(t) = \sum_{\sigma \in \mathcal{S}'} q(\sigma)\ell_t(\sigma).$$

4. Supposons $\mathcal{S} = [0, n-1]$. Définissez une fonction OCaml `cq : tree -> int array -> int` qui reçoit deux arguments, un arbre $t \in \mathcal{T}_{\mathcal{S}}$ et un tableau $[\![q(0); q(1); \ldots; q(n-1)]\!]$ représentant q, et qui renvoie la valeur de $c_q(t)$. On cherchera à écrire une fonction efficace. Donnez et justifiez sa complexité temporelle.

Une référence mise à jour lors d'un parcours en profondeur suffit. On propose :

```
let cq t q =
    let r = ref 0 in
    let rec dfs k (* k : niveau de profondeur de l'appel *)
        = function
        | F i -> r := !r + (k * q.(i))
        | N (t1, t2)
              -> dfs (k + 1) t1 ; dfs (k + 1) t2
    in
    dfs 0 t ; !r
```

Cette fonction est de coût linéaire en la taille de l'arbre en argument `t`, qui est lui-même de taille $O(|\mathcal{S}|)$ (il possède $|\mathcal{S}|$ feuilles, et $|\mathcal{S}|-1$ nœuds internes d'après le sujet). En effet, le coût marginal de chaque appel à `dfs` est constant, et cette fonction visite exactement une fois chaque nœud de l'arbre.

Étant donné un arbre t de $\mathcal{T}_{\mathcal{S}}$, on représente une lettre $\sigma \in \mathcal{S}$ par la séquence des bits obtenus en parcourant t de la racine vers la feuille $F(\sigma)$. Par exemple, dans l'arbre de $\mathcal{T}_{\{A,B,C,D\}}$ de la Figure 1, D est représenté par 0 tandis que A est représenté par 100.

5. Définissez une fonction OCaml `get_path : int -> tree -> int list` qui reçoit en argument un entier $\sigma \in \mathcal{S}$ et un arbre $t \in \mathcal{T}_{\mathcal{S}}$ et qui renvoie la liste de 0 et 1 représentant σ dans t. Donnez et justifiez la complexité temporelle de cette fonction.

On effectue un parcours qui teste les deux sous-arbre à chaque nœud interne et complète le chemin en fonction de celui (unique) où se trouve (dans une feuille) l'élément recherché. Il faut cependant prendre garde à ne pas effectuer d'opérations trop coûteuses lors de la construction du chemin. On propose deux versions :

```
let get_path s t =
    let rec gp x
        = function
        | F i -> if i = s then Some x else None
        | N (t1, t2)
              ->(match gp (0::x) t1 with
                 | Some p -> Some p
                 | None -> gp (1::x) t2)
    in
    match gp [] t with Some p -> List.rev p | _ -> assert false

let get_path' s t =
    let rec gp
        = function
        | F i -> if i = s then Some [] else None
        | N (t1, t2)
              ->(match (gp t1, gp t2) with
                 | Some p, _ -> Some (0::p)
                 | _, Some p -> Some (1::p)
                 | _         -> None
              )
    in
    match gp t with Some p -> p | _ -> assert false
```

La fonction gp effectue un parcours de `t` dont toutes les opérations marginales sont de coût constant ; son appel coûte donc un $O(|\mathcal{S}|)$. Dans la première version, l'appel à `List.rev` est de coût linéaire en la longueur de son

argument qui est un $O(|\mathcal{S}|)$ (elle est majorée par la profondeur de l'arbre, elle même majorée par sa taille). Dans tous les cas, le coût total est un $O(|\mathcal{S}|)$.

Considérons les séquences de bits définies de la façon suivante. Elles commencent par $k \geq 0$ bits valant 1, suivis d'un bit à 0, suivis de k bits arbitraires, ce que l'on notera $1^k0(0|1)^k$. Les dix premières séquences par ordre lexicographique sont donc 0, 100, 101, 11000, 11001, 11010, 11011, 1110000, 1110001, 1110010. L'arbre dont les branches constituent ces séquences et dont les feuilles sont étiquetées de gauche à droite par des entiers croissants commence comme illustré sur la Figure 2.

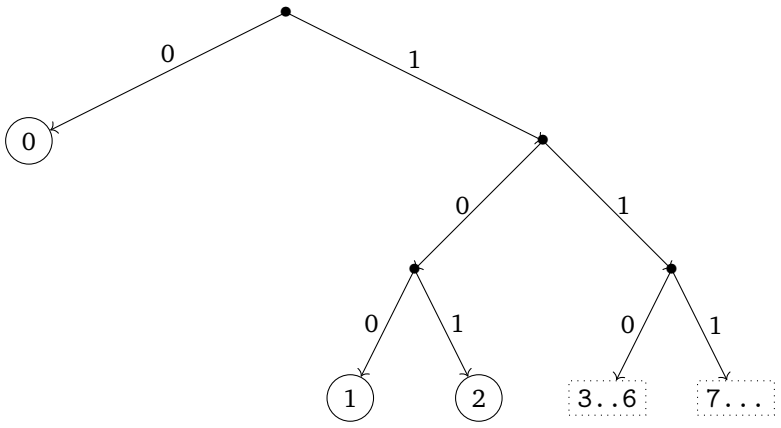

FIGURE 2 – Arbre des séquences $1^k0(0|1)^k$

Remarque : l'étiquette de chaque feuille correspond à l'indice de la séquence quand elles sont triées par ordre lexicographique. Les séquences croissant indéfiniment, l'arbre est a priori infini. C'est pourquoi la question suivante se limite aux séquences de bits $1^k0(0|1)^k$ pour lesquelles k n'excède pas une certaine borne ℓ, afin d'obtenir un arbre de profondeur bornée $2\ell + 1$.

6. Définissez une fonction OCaml `integers : int -> tree` qui prend un entier $\ell \geq 0$ en argument et renvoie l'arbre dont les branches sont les séquences de la forme $1^k0(0|1)^k$ avec $k \leq \ell$, ainsi que la séquence $1^{\ell+1}$. Les feuilles seront étiquetées de gauche à droite par les entiers 0, 1, 2, etc.

On remarque que l'arbre infini est tel que le nœud de numéro 1^k0 est un sous-arbre complet de taille k. Il s'agit alors (pour répondre à la question) de construire cet arbre tronqué en son nœud $1^{\ell+1}$ (remplacé par une feuille). Le plus commode est sans-doute de définir une fonction séparée qui construit un arbre complet d'une certaine profondeur et d'utiliser une référence pour garder le compte des feuilles à étiqueter :

```
let rec complete_tree d s =
    if d = 0 then
        let f = F !s in incr s ; f
    else
        let t1 = complete_tree (d - 1) s in
        let t2 = complete_tree (d - 1) s in
        N (t1, t2)

let integers ell =
    let s = ref 0 in
    let rec _integers d =
        if d = ell + 1 then F !s
        else
            let t1 = complete_tree d s in
            let t2 = _integers (d + 1) in
            N (t1, t2)
    in
    _integers 0
```

Une version purement fonctionnelle est également possible, mais nécessite (dans notre implémentation) une façon

efficace de calculer le nombre de nœuds d'un arbre binaire complet d'une certaine profondeur, ce que l'on peut faire avec la fonction `lsl` :

```
let rec complete_tree' k s n =
    if k = 0 then
        F s
    else
        let t1 = complete_tree' (k - 1) s (n/2) in
        let t2 = complete_tree' (k - 1) (s + (n/2)) (n/2) in
        N (t1, t2)

let integers' ell =
    let rec _integers d s =
        if d = ell + 1 then F s
        else
            let t1 = complete_tree' d s (1 lsl d) in
            let t2 = _integers (d + 1) (s + (1 lsl d)) in
            N (t1, t2)
    in
    _integers 0 0
```

III. Codes préfixes

Étant donnés un alphabet $\mathcal{S}$ et un arbre t de $\mathcal{T}_\mathcal{S}$, un mot de $\mathcal{S}^*$ peut être représenté par la concaténation des séquences de bits représentant chacune de ses lettres dans t, de la gauche vers la droite. Supposons que l'arbre t est l'exemple de la Figure 1. Le mot « ADBDCD » est alors représenté par la séquence de bits 100|0|11|0|101|0. Remarque : les barres verticales ne servent qu'à rendre l'exemple plus lisible ; elles ne font pas partie de la séquence de bits et ne sont pas nécessaires pour retrouver le mot initial.

7. Soit t un arbre de $\mathcal{T}_\mathcal{S}$. Étant donnée une séquence de bits, montrez qu'il existe au plus un mot de $\mathcal{S}^*$ qui est représenté par cette séquence. On justifiera *soigneusement*.

On appelle $\mathcal{A}$ l'ensemble des séquences de bits $t(\sigma)$ représentant les lettres de $\mathcal{S}$ dans t. Par construction, pour tous $\sigma, \sigma' \neq \sigma \in \mathcal{S}$ l'on a que $t(\sigma)$ n'est pas préfixe de $t(\sigma')$. En effet, $t(\sigma)$ (resp. $t(\sigma')$) est défini comme la séquence des bits obtenus en parcourant t de la racine vers la feuille $F(\sigma)$ (resp. $F(\sigma')$), et le fait que $t(\sigma)$ soit préfixe de $t(\sigma')$ impliquerait alors que le chemin de la racine à $F(\sigma')$ passe par $F(\sigma)$, ce qui est absurde puisque ce dernier nœud est une feuille.
Supposons maintenant par l'absurde l'existence d'une séquence représentant (au moins) deux mots distincts de $\mathcal{S}^*$. Soit s une telle séquence de longueur minimale k et m_1 et $m_2 \neq m_1$ deux mots qu'elle représente. Par la propriété ci-dessus, on a nécessairement $m_1 = \sigma \cdot q_1$ et $m_2 = \sigma \cdot q_2$ pour certains suffixes $q_1 \neq q_2$ (le contraire $m_1 = \sigma \cdots$ et $m_2 = \sigma' \cdots$ avec $\sigma \neq \sigma'$ impliquerait que les premiers bits de la séquence représentent (sans perte de généralité) $t(\sigma)$ et un préfixe de $t(\sigma')$, ce qui est impossible). On a donc que la séquence s dont on a retiré les $|t(\sigma)| > 1$ premiers bits représente deux mots distincts q_1 et q_2, ce qui est une contradiction par minimalité de k. Une telle séquence s ne peut donc pas exister.

8. Supposons $\mathcal{S} \subseteq \mathbb{N}$. Définissez une fonction OCaml `decomp1 : int list -> tree -> int list` qui reçoit en argument une liste non vide d'entiers 0 ou 1 et un arbre `t` de $\mathcal{T}_\mathcal{S}$ et qui renvoie la liste des éléments de $\mathcal{S}$ représentée par cette liste de bits. On sera attentif au cas où la liste en argument ne représente aucun mot (non vide) de $\mathcal{S}^*$. On a aussi toujours que $|\mathcal{S}| \geq 2$, et donc que l'argument `t` est de la forme `N (t1, t2)` pour certains arbres `t1`, `t2`. Donnez et justifiez la complexité temporelle de cette fonction.

Pour décompresser une lettre de $\mathcal{S}$, on suit le chemin dans l'arbre `t` donné par la suite d'entiers de l'argument `x`. Chaque fois que l'on atteint une feuille, on ajoute l'étiquette de celle-ci à la liste (construite à l'envers) des lettres lues et l'on recommence. La fonction échoue si l'on a épuisé la liste d'entiers avant d'atteindre une feuille. Ceci donne par exemple :

```
let decomp1 x t =
    let rec decomp1' x' t' acc =
        match t' with
```

```
        | F s -> if x' = [] then (s::acc) else decomp1' x' t (s::acc)
        | N (t1, t2)
              -> (match x' with
                  | 0::xs' -> decomp1' xs' t1 acc
                  | 1::xs' -> decomp1' xs' t2 acc
                  | _ -> failwith "invalid sequence")
    in
    decomp1' x t [] |> List.rev
```

Chaque appel à `decomp1'` est de coût marginal constant, et la complexité temporelle de `decomp1` est donc donnée par le nombre d'appels à `decomp1'` plus celui de `List.rev` sur le résultat. Or, un appel à `decomp1'` se fait ou bien sur un argument `x'` strictement plus petit (cas `N (t1, t2)`) ou bien termine ou effectue un appel récursif sur `t`, qui est un arbre de la forme `N (t1, t2)` (cas `F s`). Il s'ensuit que le nombre d'appels récursifs de `decomp1'` est majoré par deux fois la longueur de `x`. Enfin, la longueur du résultat renvoyé augmente également au plus de 1 à chaque appel récursif (cas `F s`), ce qui permet de conclure quand au coût temporel total linéaire en la longueur de l'argument `x`.

Cette fonction est dite de *décompression*. Supposons que $\mathcal{S}$ est $\{A, B, C, D\}$ et que la fonction q donne le nombre d'occurrences de chaque lettre dans le mot « ADBDCD », par exemple $q(D) = 3$. Dans ce cas, $c_q(t)$ vaut 11 pour l'arbre exemple de la Figure 1. Il s'agit de la longueur de la séquence 10001101010 représentant le mot « ADBDCD ». Plus généralement, $c_q(t)$ est la longueur de n'importe quelle séquence représentant un mot de $\mathcal{S}^q$ compressé avec l'arbre t. Il s'avère qu'il n'existe aucun arbre $t' \in \mathcal{T}_\mathcal{S}$ tel que $c_q(t') < 11$; t est donc optimal.
Étant donnée une fonction q, l'objectif de la partie suivante sera de construire un des arbres de $\mathcal{T}_\mathcal{S}$ offrant la meilleure compression des mots de $\mathcal{S}^q$, c'est-à-dire un arbre qui minimise c_q.

IV. Arbres optimaux

Soit q une fonction dont le domaine inclut $\mathcal{S}$. Un arbre de $\mathcal{T}_\mathcal{S}$ est dit optimal pour $(q, \mathcal{S})$ s'il minimise c_q parmi tous les arbres de $\mathcal{T}_\mathcal{S}$. Autrement dit, un arbre optimal $t \in \mathcal{T}_\mathcal{S}$ vérifie $c_q(t) = \min_{t' \in \mathcal{T}_\mathcal{S}} c_q(t')$. De tels arbres optimaux existent et, dès lors que $|\mathcal{S}| \geq 2$, ils ne sont pas uniques.

9. Soit $t = N(g, d) \in \mathcal{T}_\mathcal{S}$ un arbre optimal pour $(q, \mathcal{S})$. Soit $\mathcal{S}_g$ l'ensemble des lettres qui étiquettent les feuilles du sous-arbre g. Montrez que g est un arbre de $\mathcal{T}_{\mathcal{S}_g}$ optimal pour $(q, \mathcal{S}_g)$.

On remarque que les profondeurs des feuilles de $\mathcal{S}_g$ (resp. $\mathcal{S} \setminus \mathcal{S}_g$) dans $N(g, d)$ valent un de plus que leur profondeur dans g (resp. d). Ainsi l'on a $c_q(t) = c_q(g) + c_q(d) + \sum_{\sigma \in \mathcal{S}} q(\sigma)$ (où le terme $\sum_{\sigma \in \mathcal{S}} q(\sigma)$ compte simplement le nombre total d'occurrences des lettres).
Supposons maintenant par l'absurde l'existence d'un arbre $g' \in \mathcal{T}_{\mathcal{S}_g}$ tel que $c_q(g') < c_q(g)$. Alors l'arbre $t' := N(g', d)$ est bien un arbre de $\mathcal{T}_\mathcal{S}$ et a un coût $c_q(t') = c_q(g') + c_q(d) + \sum_{\sigma \in \mathcal{S}} q(\sigma) < c_q(t)$, ce qui contredit l'optimalité de t. Un tel arbre g' ne peut donc pas exister.

Malheureusement, cette propriété ne permet pas d'en déduire un algorithme de type « diviser pour régner » pour trouver l'arbre optimal. En effet, il faudrait que l'algorithme puisse efficacement deviner comment partitionner $\mathcal{S}$ entre les feuilles du sous-arbre gauche et celles du sous-arbre droit.

10. Supposons qu'il existe une lettre $\sigma_0 \in \mathcal{S}$ telle que $q(\sigma_0) > \sum_{\sigma \in \mathcal{S} \setminus \{\sigma_0\}} q(\sigma)$. Montrez que, pour tout arbre de $\mathcal{T}_\mathcal{S}$ optimal pour $(q, \mathcal{S})$, la feuille $F(\sigma_0)$ est à profondeur 1, c'est-à-dire directement attachée à la racine.

On va considérer un arbre t quelconque et montrer que s'il n'est pas de la forme ci-dessus, on peut en déduire un arbre t' de la forme voulue de coût strictement inférieur (ce qui est suffisant pour conclure). Soit $t = N(g, d)$ un tel arbre, on suppose sans perte de généralité que σ_0 apparaît comme feuille de d à une profondeur $\ell_t(\sigma_0) > 1$, et l'on construit t' comme $N(F(\sigma_0), d')$ avec d' l'arbre obtenu en substituant g à $F(\sigma_0)$.
En notant $\mathcal{S}_g$ l'ensemble des lettres apparaissant comme feuilles dans g, la différence de coût $c_q(t') - c_q(t)$ est alors donnée par :

$$(1-\ell_t(\sigma_0)) \times q(\sigma_0) + (\ell_t(\sigma_0)-1) \times \sum_{\sigma \in \mathcal{S}_g} q(\sigma) <$$
$$(1-\ell_t(\sigma_0)) \times \sum_{\sigma \in \mathcal{S}'} q(\sigma) + (\ell_t(\sigma_0)-1) \times \sum_{\sigma \in \mathcal{S}_g} q(\sigma) =$$
$$(1-\ell_t(\sigma_0)) \times \sum_{\sigma \in \mathcal{S}' \setminus \mathcal{S}_g} q(\sigma) \leq 0$$

où le caractère strict de la première inégalité vient du fait que $1-\ell_t(\sigma_0) < 0$, et la dernière inégalité de ce même fait et de $q(\sigma) \geq 0$ pour tout σ.

11. Soient σ_1 et σ_2 deux éléments différents de $\mathcal{S}$ tels que, pour tout $\sigma \in \mathcal{S} \setminus \{\sigma_1, \sigma_2\}$, on a $q(\sigma) \geq q(\sigma_1)$ et $q(\sigma) \geq q(\sigma_2)$. Soit $\mathcal{S}' = \mathcal{S} \setminus \{\sigma_1, \sigma_2\} \cup \{\sigma_3\}$ avec σ_3 un tout nouvel élément. On définit q' de telle sorte que $q'(\sigma_3) = q(\sigma_1) + q(\sigma_2)$ et $\forall \sigma \in S \setminus \{\sigma_1, \sigma_2\}$, $q'(\sigma) = q(\sigma)$.

i. Montrez qu'il existe un arbre t de $\mathcal{T}_\mathcal{S}$ optimal pour $(q, \mathcal{S})$ ayant un nœud $N(F(\sigma_1), F(\sigma_2))$.

ii. Soit un arbre t' de $\mathcal{T}'_\mathcal{S}$ optimal pour $(q', \mathcal{S}')$. Montrez que l'arbre $t \in \mathcal{T}_\mathcal{S}$ obtenu en remplaçant dans t' la feuille $F(\sigma_3)$ par $N(F(\sigma_1), F(\sigma_2))$ est optimal pour $(q, \mathcal{S})$.

i. Soit $t' \in \mathcal{T}_\mathcal{S}$ optimal, on va construire t optimal de la forme voulue. Si t' n'est pas déjà de cette forme, on choisit arbitrairement deux feuilles de profondeur maximum, attachées à un nœud $N(F(\sigma_3), F(\sigma_4))$ et l'on échange $F(\sigma_3)$ (resp. $F(\sigma_4)$) avec $F(\sigma_1)$ (resp. $F(\sigma_2)$). La différence entre $c_q(t)$ et $c_q(t')$ est alors $(q(\sigma_1) - q(\sigma_3)) \times (\ell_{t'}(\sigma_3) - \ell_{t'}(\sigma_1)) + (q(\sigma_2) - q(\sigma_4)) \times (\ell_{t'}(\sigma_4) - \ell_{t'}(\sigma_2))$ qui est ≤ 0 par le fait que $q(\sigma_1) < q(\sigma_3)$, $q(\sigma_2) \leq q(\sigma_4)$, et que $F(\sigma_3)$ et $F(\sigma_4)$ sont à profondeur maximale.

ii. On a $c_q(t) = c_{q'}(t') + q(\sigma_1) + q(\sigma_2)$ (les feuilles $F(\sigma_1)$ & $F(\sigma_2)$ sont à une profondeur $\ell_{t'}(\sigma_3) + 1$, $q'(\sigma_3) = q(\sigma_1) + q(\sigma_2)$, et rien d'autre n'a changé dans l'arbre). Supposons par l'absurde qu'il existe un arbre t'' avec $c_q(t'') < c_q(t)$. Par la sous-questions précédente, on peut supposer sans perte de généralité que t'' possède un nœud $N(F(\sigma_1), F(\sigma_2))$. Soit alors l'arbre t''' obtenu en remplaçant ce nœud par une feuille $F(\sigma_3)$, l'on a $c_{q'}(t''') = c_q(t'') - q(\sigma_1) - q(\sigma_2)$ (cette feuille est à une profondeur $\ell_{t''}(\sigma_1) - 1 = \ell_{t''}(\sigma_2) - 1$), et donc $c_{q'}(t''') < c_q(t) - q(\sigma_1) - q(\sigma_2) = c_{q'}(t')$, ce qui est une contradiction. Un tel arbre t'' ne peut donc pas exister.

On définit la fonction $\bar{q}$ en étendant la fonction q aux arbres de la façon suivante. Pour une feuille, $\bar{q}(F(\sigma))$ vaut $q(\sigma)$. Pour un nœud interne, $\bar{q}(N(g,d))$ vaut récursivement $\bar{q}(g) + \bar{q}(d)$. Remarque : en général, $c_q(t) \neq \bar{q}(t)$.

La question précédente montre qu'il est possible de construire un arbre optimal pour $(q, \mathcal{S})$ à l'aide de l'algorithme suivant. On initialise un ensemble d'arbres E avec toutes les feuilles $F(\sigma)$ avec $\sigma \in \mathcal{S}$. À chaque étape, on retire de E deux arbres t_1 et t_2 qui minimisent $\bar{q}$; puis on ajoute à E l'arbre $N(t_1, t_2)$. On répète cette procédure jusqu'à ce qu'il ne reste qu'un seul arbre dans E. Il s'agit alors d'un arbre optimal pour $(q, \mathcal{S})$.

12. Implantez l'algorithme décrit ci-dessus en définissant les deux fonctions OCaml suivantes : `insert` et `optimal`.

i. La fonction `insert : (int * tree) -> (int * tree) list -> (int * tree) list` insère dans une liste son premier argument. La liste en argument est supposée triée par rapport à la première composante de chacune de ses paires. La liste renvoyée doit l'être aussi.

ii. La fonction `optimal : (int * int) list -> tree` renvoie un arbre optimal. La liste passée en argument est de taille $|\mathcal{S}|$; elle contient toutes les paires $(\sigma, q(\sigma))$ pour $\sigma \in \mathcal{S}$; ces paires sont triées par valeur croissante de $q(\sigma)$.

iii. Donnez et justifiez la complexité temporelle de la fonction `optimal`.

i. On propose :

```
let rec insert (x, t) y
    = match y with
    | [] -> [(x, t)]
    | (y', t')::ys
        -> if y' < x then
               (y', t')::(insert (x, t) ys)
           else
               (x, t)::y
```

ii. Attention à l'ordre des éléments dans les paires de la liste en argument. On propose :

```
let optimal ep =
    let e = List.map (fun (s, q) -> (q, F s)) ep in
    let rec loop
        = function
        | (_, t)::[] -> t
        | (q1, t1)::(q2, t2)::xs
                    -> insert (q1 + q2, N (t1, t2)) xs |> loop
        | [] -> assert false
    in
    loop e
```

iii. Le fonction `insert` est de coût temporel (pire cas) linéaire en la longueur de son second argument. Dans la fonction `optimal`, chaque appel à `loop` a donc un coût temporel en $O(\ell)$ avec ℓ la longueur de son argument. L'appel initial à `loop` se fait sur `e`, de longueur $|\mathcal{S}|$, et chaque appel décroît la longueur de l'argument de un (on remplace deux éléments par un seul) jusqu'à atteindre une longueur un. On fait donc $O(|\mathcal{S}|)$ appels de coût marginal $O(|\mathcal{S}|)$, pour un coût total $O(|\mathcal{S}|^2)$.

Les deux questions suivantes montrent que, quand la liste initiale des $(\sigma, q(\sigma))$ est triée par valeur croissante de $q(\sigma)$, de simples listes et/ou tableaux suffisent pour écrire une fonction `optimal` dont la complexité temporelle est en $O(|\mathcal{S}|)$.

13. Montrez que, lors de l'exécution de l'algorithme décrit ci-dessus, les arbres $t = N(t_1, t_2)$ sont ajoutés à l'ensemble E par valeur croissante de $\bar{q}(t)$.

On prouve la propriété par récurrence sur le rang n d'ajout. En notant, t^i le $i^{\text{ème}}$ arbre à être ajouté, on pose $\mathcal{P}(n)$: $\bar{q}(t^n) \geq \bar{q}(t^i)$ pour tout $i < n$.
Cas de base ($n = 1$) : il n'y a rien à montrer car $\{t^i \mid i < 1\} = \varnothing$.
Conservation : soit t^n le $n^{\text{ième}}$ arbre à être ajouté à E, on considère deux cas :

— $t^n = N(t^{n-1}, \tau)$ pour un certain arbre τ quelconque ; alors $\bar{q}(t^n) = \bar{q}(t^{n-1}) + \bar{q}(\tau) \geq \bar{q}(t^{n-1})$, et par hypothèse de récurrence $\bar{q}(t^n) \geq \bar{q}(t^i)$ pour $i < n$.
— $t^n = N(\tau_1, \tau_2)$ pour deux arbres τ_1, τ_2 quelconques distincts de t^{n-1}. Alors ces arbres étaient présents dans E au moment de la construction de $t^{n-1} =: N(\tau_3, \tau_4)$, et donc $\bar{q}(\tau_3) + \bar{q}(\tau_4) \leq \bar{q}(\tau_1) + \bar{q}(t_2)$. On a à nouveau $\bar{q}(t^n) \geq \bar{q}(t^{n-1})$, et par hypothèse de récurrence $\bar{q}(t^n) \geq \bar{q}(t^i)$ pour $i < n$.

Dans les deux cas la propriété est conservée.

14. Expliquez comment écrire la fonction `optimal` pour que sa complexité temporelle soit linéaire. Le code OCaml n'est pas demandé.

On peut utiliser deux structures de file (qui implémentent une opération *peek* permettant de consulter l'élément s'apprêtant à être défiler, sans l'extraire) : l'une f_1 remplie avec les éléments initiaux de E ajoutés par valeur croissante de q, et l'autre f_2 où l'on enfilera chaque arbre ajouté à E. On ne fait qu'extraire des éléments de f_1, et par la question précédente les éléments ajoutés à f_2 le sont par valeur de $\bar{q}$ croissante. On va donc pouvoir maintenir un invariant qui est que les deux files contiennent des arbres par valeur de $\bar{q}$ croissante, et un arbre de valeur $\bar{q}$ minimum est donc forcément l'un en attente d'être défilé dans f_1 ou f_2. Chaque itération de l'algorithme demande donc seulement quatre opérations *peek* (pour déterminer les deux arbres de valeur $\bar{q}$ minimale), deux *pop* (pour les extraire), un *push* (pour ajouter le résultat à f_2). Toutes ces opérations peuvent être implémentées en temps constant, ce qui donne un coût temporel $O(|E|) = O(|\mathcal{S}|)$.

Cette partie a permis de calculer un arbre qui minimise la longueur c_q de la séquence de bits représentant un mot de $\mathcal{S}^q$. Mais pour décompresser ce mot, il faut disposer de l'arbre et donc l'avoir stocké quelque part. Parmi tous les arbres optimaux, il est donc important d'en choisir un qui nécessitera le moins de bits pour être représenté ; c'est l'objet de la partie suivante.

V. Arbres canoniques

On suppose maintenant qu'il existe un ordre alphabétique noté $<$ sur les éléments de $\mathcal{S}$. Pour $\mathcal{S} \subseteq \mathbb{N}$, il s'agira de l'ordre usuel sur $\mathbb{N}$. Soit t un arbre de $\mathcal{T}_{\mathcal{S}}$. On définit la relation $<_t$ entre éléments de $\mathcal{S}$ de la façon suivante : pour toute paire $(\sigma_1, \sigma_2) \in \mathcal{S}^2$,

$$\sigma_1 <_t \sigma_2 \Leftrightarrow \ell_t(\sigma_1) < \ell_t(\sigma_2) \text{ ou } (\ell_t(\sigma_1) = \ell_t(\sigma_2) \text{ et } \sigma_1 < \sigma_2).$$

L'arbre t est dit canonique si, en parcourant les feuilles de la gauche vers la droite, leurs étiquettes respectent la relation $<_t$. Autrement dit, plus une feuille est à gauche, plus elle est proche de la racine ; et les étiquettes des feuilles à une même profondeur sont triées de gauche à droite par ordre alphabétique.
L'arbre de la Figure 1 vérifie bien la deuxième partie de la propriété : les feuilles à une même profondeur sont triées de gauche à droite par ordre alphabétique. Mais il ne respecte pas la première partie : la feuille étiquetée par A est plus profonde que celle étiquetée par B alors qu'elle est plus à gauche. Cet arbre n'est donc pas canonique. L'arbre de la Figure 3 suivante est par contre canonique :

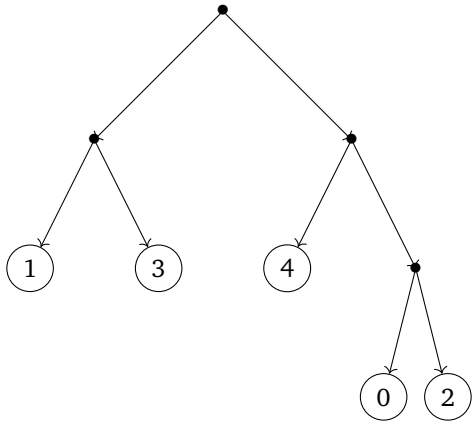

FIGURE 3 – Exemple d'arbre canonique

Un arbre canonique peut être représenté par deux tableaux d'entiers. Le premier tableau associe à chaque indice i le nombre de lettres $\sigma \in \mathcal{S}$ telles que $\ell_t(\sigma) = i$. Le deuxième tableau contient les éléments de $\mathcal{S}$ obtenus en parcourant les feuilles de l'arbre de gauche à droite. Ainsi, l'arbre de $\mathcal{T}_{[0,4]}$ de la Figure 3 est représenté par les deux tableaux $[\![0;0;3;2]\!]$ et $[\![1;3;4;0;2]\!]$.

15. Définissez une fonction OCaml `canonical : int array -> int array -> tree` qui, étant donnés deux tels tableaux, renvoie l'arbre canonique qu'ils représentent, s'il existe. Donnez et justifiez la complexité temporelle de cette fonction.

Il existe une solution « simple » en style impératif : on mémorise le nombre de feuilles restantes pour un certain niveau de profondeur et (l'indice permettant de retrouver) la valeur de la prochaine feuille à créer, et l'on construit l'arbre par un parcours en profondeur de gauche à droite. En mémorisant la profondeur dans l'arbre, on sait s'il faut créer une feuille ou des sous-arbres par appel récursif.
Ceci donne par exemple :

```
let canonical a1 a2 =
    let a1' = Array.copy a1 in
    let i = ref 0 in
    let rec make d =
        if a1'.(d) >= 1 then
            let t = F a2.(!i) in
            (incr i ; a1'.(d) <- a1'.(d) - 1 ; t)
        else
            let t1 = make (d + 1) in
            let t2 = make (d + 1) in
            N (t1, t2)
    in
    make 0
```

La création de `a1'` est de coût linéaire en la profondeur de l'arbre construit, qui est au plus linéaire en son nombre de sommets, et le reste des opérations sont toutes de coût constant. Le coût total est donc dominé par la taille du résultat construit, qui est un $O(|\mathcal{S}|)$.

VI. Arbres alphabétiques

L'arbre de la Figure 1 était optimal pour le mot « ADBDCD ». Un autre arbre optimal pour ce mot est celui de la Figure 4.

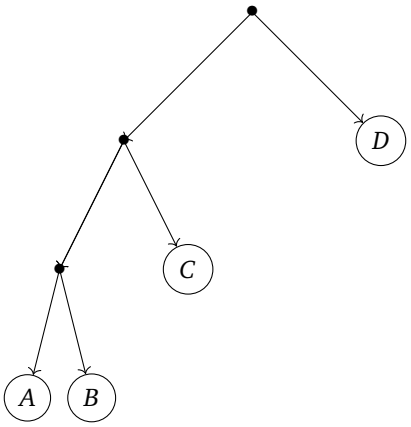

FIGURE 4 – Arbre à la fois optimal pour le mot « ADBDCD » et alphabétique

Il n'est pas canonique mais il a une propriété très intéressante : ses feuilles sont dans l'ordre alphabétique. Cela signifie que lors du stockage de l'arbre, il n'est pas nécessaire de stocker les étiquettes des feuilles, il suffit de stocker la forme de l'arbre. Un tel arbre est dit *alphabétique*.
Plus généralement, un arbre de $\mathcal{T}_\mathcal{S}$ est *alphabétique* si, en parcourant ses feuilles de gauche à droite, les étiquettes apparaissent dans l'ordre alphabétique. On note $\mathcal{A}_\mathcal{S}$ l'ensemble de ces arbres. On dit d'un arbre qu'il est *alphabétique-optimal* pour $(q, \mathcal{S})$ s'il minimise c_q parmi tous les arbres de $\mathcal{A}_\mathcal{S}$. Remarque : un arbre alphabétique-optimal pour $(q, \mathcal{S})$ n'est pas nécessairement optimal pour $(q, \mathcal{S})$.
Soit $n \in \mathbb{N}$ et $\mathcal{S} = [0, n-1]$. On se donne une fonction $q : \mathcal{S} \to \mathbb{N}$. Étant donnés deux entiers i et j tels que $0 \leq i \leq j < n$, on note $m_{i,j}$ la valeur de $c_q(t)$ pour n'importe quel arbre t de $\mathcal{A}_{[i,j]}$ alphabétique-optimal pour $(q, [i, j])$. En particulier, si t est un arbre de $\mathcal{A}_\mathcal{S}$. alphabétique-optimal pour $(q, \mathcal{S})$, il vérifie $c_q(t) = m_{0,n-1}$.

16. Exprimez la valeur de $m_{i,j}$ en fonction des valeurs de $m_{i',j'}$ avec $[i', j'] \subset [i, j]$. Déduisez en une fonction OCaml `alpha_optimal : int array -> tree` qui calcule un arbre de $\mathcal{A}_{[0,n-1]}$ alphabétique-optimal pour $(q, [0, n-1])$. La fonction q est donnée par le tableau de taille n passé en argument. La complexité temporelle de la fonction devra être en $O(n^3)$.

Un arbre $t \in \mathcal{A}_{[i,j]}$ est nécessairement de la forme $N(g, d)$ avec $g \in \mathcal{A}_{[i,k]}$, $d \in \mathcal{A}_{[k+1,j]}$. S'il est alphabétique-optimal alors g et d le sont aussi, et k doit être la meilleure position pour séparer $[i, j]$ en deux. En clair :

$$m_{i,j} = \min_{i \leq k < j} m_{i,k} + m_{k+1,j} + \sum_{i \leq \sigma \leq j} q(\sigma),$$

où le terme $\sum_{i \leq \sigma \leq j} q(\sigma)$ compte l'augmentation de hauteur de un des feuilles de g et d dans $N(g, d)$.
On peut alors calculer cette récursion de type « programmation dynamique » par un algorithme similaire à celui « de Floyd-Warshall », et comme ce dernier il faut prendre garde à écrire les boucles dans le bon ordre : ici, le calcul des $m_{i,j}$ doit se faire *par écart croissant* entre i et j (correspondant à la variable de boucle `d` ci-dessous). Pour éviter d'avoir à également implémenter une fonction de reconstruction à partir de la seule matrice des coûts, on peut aussi simplement stocker des arbres optimaux conjointement à ceux-ci (grâce au partage, le coût de construction d'un tel arbre est constant à chaque itération).
On se donne une fonction :

```
let sumrange q i j =
    let r = ref 0 in
    for k = i to j do
        r := !r + q.(k)
    done ;
    !r
```

On obtient alors (par exemple) :

```
let alpha_optimal q =
    let n = Array.length q in
    let m = Array.make_matrix n n (0, (F 0)) in
    for i = 0 to (n - 1) do
        m.(i).(i) <- (0, F i)
    done ;
    for d = 1 to (n - 1) do
        for i = 0 to (n - 1) do
            let j = i + d in
            if j < n then (
                let q1, t1 = m.(i).(i) in
                let q2, t2 = m.(i + 1).(j) in (* calculé à l'itération externe *)
                let minc = ref (q1 + q2) in   (* précédente *)
                let mint = ref (N (t1, t2)) in
                for k = i + 1 to j - 1 do
                    let q1, t1 = m.(i).(k) in   (* calculés à des itérations ext. *)
                    let q2, t2 = m.(k+1).(j) in (* précédentes *)
                    if q1 + q2 < !minc then (
                        minc := q1 + q2 ;
                        mint := N (t1, t2)
                    )
                done ;
                m.(i).(j) <- (!minc + sumrange q i j), !mint
            )
        done ;
    done ;
    snd m.(0).(n - 1)
```

Un arbre alphabétique de $\mathcal{A}_{[0,n-1]}$ peut être représenté par la séquence de $2n-1$ bits obtenue en effectuant un parcours en profondeur préfixe, c'est-à-dire que la racine d'un sous-arbre est visité avant ses feuilles et qu'un sous-arbre gauche est parcouru avant un sous-arbre droit. Pour chaque nœud, les chiffres 0 et 1 indiquent respectivement s'il s'agit d'un nœud interne ou d'une feuille. Par exemple, l'arbre de la Figure 4 est représenté par la séquence 0001111.

17. Définissez une fonction OCaml `alpha : int array -> tree` qui reçoit en argument un tableau de 0 et 1 et renvoie l'arbre alphabétique correspondant. On sera attentif au cas où le tableau en argument ne représente aucun arbre. Cette fonction devra avoir une complexité temporelle en $O(n)$ avec n la taille du tableau.

Il s'agit de reproduire le parcours en profondeur ayant mené au tableau en argument. Les valeurs 0 ou 1 des éléments de celui-ci indiquent si l'on se trouve sur un nœud interne ou une feuille. On propose :

```
let alpha a =
    let n' = Array.length a in
    let cnt = ref 0 in
    let rec _alpha i =
        if a.(i) = 1 then
            let t = F !cnt in
            incr cnt ;
            (t, i)
        else (* a.(i) = 0 *)
            let t1, j = _alpha (i + 1) in
            if j >= n'- 1 then failwith  "pas un arbre" else
            let t2, k = _alpha (j + 1) in
            N (t1, t2), k
    in
    _alpha 0 |> fst
```

La fonction auxiliaire `_alpha` prend en argument l'indice de `a` donnant le nœud à considérer, et renvoie l'arbre enraciné en ce nœud ainsi que l'indice de sa dernière feuille.
On utilise à nouveau une référence pour garder trace de la prochaine valeur d'étiquette à utiliser.

VII. Codes arithmétiques

Soient un alphabet $\mathcal{S}$ et une fonction $q : \mathcal{S} \to \mathbb{N}$. Comme précédemment, $\mathcal{S}^q$ est le sous-ensemble des mots de $\mathcal{S}^*$ qui contiennent exactement $q(\sigma)$ occurrences de σ pour chaque $\sigma \in S$.

18. Supposons que $\mathcal{S}$ est $\{A, B, C\}$ et que q satisfait $q(A) = 15$, $q(B) = 4$, $q(C) = 1$.

i. Combien de bits faut-il pour représenter un mot de $\mathcal{S}^q$ en utilisant un arbre optimal pour $(q, \mathcal{S})$?
ii. Combien y a-t-il de mots dans $\mathcal{S}^q$? On pourra exprimer ce nombre sous forme d'un produit de nombres premiers.
iii. Montrez que 17 bits suffisent pour représenter chaque entier entre 0 et $|\mathcal{S}^q| - 1$ (et donc n'importe quel mot de $\mathcal{S}^q$). Remarque : $15 \times 17 = 2^8 - 1$.

i. Un arbre optimal donne par exemple les codes $A \mapsto 0$, $B \mapsto 10$, $C \mapsto 11$, pour une longueur totale de $15 + 2 \times 5 = 25$ bits.
ii. Il suffit de compter le nombre de positions totales de A dans le mot multiplié par le nombre de positions totales de C dans les positions restantes, soit $\binom{20}{15} \times \binom{5}{1}$. On peut alors exprimer ce nombre comme :

$$\frac{20 \times 19 \times 18 \times 17 \times 16 \times 5}{120} = \frac{19 \times 18 \times 17 \times 16 \times 5}{6} = 19 \times 17 \times 5 \times 3 \times 2^4$$

iii. On majore 19 par 2^5, $17 \times 5 \times 3$ par 2^8 (en utilisant la remarque), ce qui permet bien de majorer $|\mathcal{S}^q|$ par $2^5 \times 2^8 \times 2^4 = 2^{17}$.

L'exemple de la question précédente montre que, dans certains cas, par exemple quand une lettre est bien plus fréquente que les autres, l'utilisation d'un code préfixe n'est pas l'approche optimale. Il faut alors se tourner vers d'autres mécanismes de compression.
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 $q(\sigma) \neq 0$:

$$E_q(x, \sigma) = \lfloor x / q(\sigma) \rfloor \times N + (x \mod q(\sigma)) + \sum_{0 \leq k < \sigma} q(k),$$

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 :

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

Ainsi, après avoir choisi x arbitrairement, un mot $\sigma_0 \cdots \sigma_{N-1} \in \mathcal{S}^q$ peut être compressé en un entier $y = C_q(x, \sigma_0 \cdots \sigma_{N-1})$. Il est alors possible de retrouver le mot original à partir de y en calculant progressivement σ_{N-1}, σ_{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 $E_q(x, \sigma) = y$.

19. Définissez une fonction OCaml `decomp2 : int array -> int -> int -> int * int array` qui reçoit en argument un tableau représentant q et deux entiers y et k, et renvoie un entier x et un tableau $[\![\sigma_0, \sigma_1, \ldots, \sigma_{k-1}]\!]$ tels que $C_q(x, \sigma_0 \sigma_1 \cdots \sigma_{k-1}) = y$

Pour renvoyer x et le tableau demandé, il faut et il suffit d'être capable d'inverser la fonction $E_q(x, \sigma)$, c'est à dire retrouver x et σ étant donné $y = E_q(x, \sigma)$.
On remarque tout d'abord que si l'on connaît σ, on peut retrouver x de la façon suivante :

— on calcule $x_q := \lfloor x / q(\sigma) \rfloor = \lfloor y / N \rfloor$ (puisque $(x \mod q(\sigma)) + \sum_{0 \leq k < \sigma} q(k) < \sum_{0 \leq k \leq \sigma} q(k) = N$) ;
— on calcule $z(\sigma) := y - \sum_{0 \leq k < \sigma} q(k)$;
— on calcule $x_r(\sigma) := x \mod q(\sigma)$ comme $(y \mod N) - z(\sigma)$ (même remarque que ci-dessus) ;
— on calcule $x = x_q \times q(\sigma) + x_r(\sigma)$.

Comme on ne connaît pas σ, on va simplement tester toutes les valeurs possibles σ' dans l'ordre et montrer que la bonne valeur est la première pour laquelle on a $x_r(\sigma') < q(\sigma')$ lors du calcul. C'est évidemment vrai lorsque

$\sigma' = \sigma$, et il est tout aussi évident que c'est une condition nécessaire pour que le résultat soit correct (puisque $x_r(\sigma')$ est censé être égal à $x \mod q(\sigma')$).
Soit $\sigma' < \sigma$, on a alors :

$$x_r(\sigma') = (y \mod N) - z(\sigma') = (x \mod q(\sigma)) + z(\sigma) - z(\sigma') = (x \mod q(\sigma)) + \sum_{\sigma' \leq k < \sigma} q(k) \geq q(\sigma')$$

comme voulu.
Ceci donne par exemple la fonction :

```
let decomp2 q y k =
    let n = Array.length q in
    let bign = k in
    let r = Array.make k 0 in
    let y' = ref y in
    for i = k - 1 downto 0 do
        let xq = !y' / bign in
        let xr_pre = !y' mod bign in
        let rec loop j z =
            if j = n then failwith "erreur de décodage" else
            let xr = xr_pre - z in
            if (xr_pre - z) < q.(j) then
                let x = xq * q.(j) + xr in
                (y' := x ;
                 r.(i) <- j)
            else
                loop (j + 1) (z + q.(j))
        in
        loop 0 0
    done ;
    !y', r
```

Remarque : on pourrait sans doute améliorer la boucle interne `loop` par une recherche en temps logarithmique en `k`, mais cela ne semble pas pertinent d'essayer de le faire ici.

Le nombre de bits de l'entier $C_q(x, \sigma_0 \dots \sigma N - 1)$ est proche de l'optimum théorique $\sum_{\sigma \in S} q(\sigma) \times \log_2(N/q(\sigma))$ mais un problème se pose en pratique. Sauf pour de tout petits mots sur de tout petits alphabets, le calcul de $C_q(x, \sigma_0 \cdots \sigma_{N-1})$ va nécessiter de manipuler des entiers très grands. Pour éviter ce problème, une solution consiste, avant chaque appel à E_q, à se débarrasser d'un certain nombre k_i de bits de poids faible. Plus précisément, on construit les suites (x_n) et (y_n) suivantes :

$$\begin{cases} y_i = & \lfloor x_i / 2^{k_i} \rfloor \\ x_{i+1} = & E_q(y_i, \sigma_i) \end{cases}$$

Comme précédemment, la valeur de x_0 est fixée arbitrairement. Le mot $\sigma_0 \cdots \sigma_{N-1}$ peut alors être représenté par d'une part x_N et d'autre part la concaténation des séquences de bits suivantes, les bits de poids forts apparaissant en premier :

$$\underbrace{x_{N-1} \mod 2^{k_{N-1}}}_{k_{N-1}\text{bits}}; \ \dots; \ \underbrace{x_2 \mod 2^{k_2}}_{k_2\text{bits}}; \ \underbrace{x_1 \mod 2^{k_1}}_{k_1\text{bits}}.$$

Pour chaque i en ordre décroissant, le décompresseur déduit de x_{i+1} les valeurs de y_i et σ_i (par exemple avec `decomp2`, question 19, avec $k = 1$), puis il lit k_i bits dans le mot compressé et les concatène à y_i pour obtenir x_i, et ainsi de suite jusqu'à avoir décompressé tout le mot.
Dans la suite, on supposera que N est une puissance de deux : $N = 2^K$. On supposera par ailleurs que le processeur n'est efficace que pour des entiers ne dépassant pas 2^B pour un certain B bien plus grand que K. En particulier, $x_{i+1} = E_q(y_i, \sigma_i)$ ne doit jamais atteindre cette borne 2^B lors de la compression.
Pour que le taux de compression se rapproche de l'optimum théorique, il faut que chaque k_i soit le plus petit possible (idéalement 0) et donc que chaque y_i soit le plus grand possible. Par ailleurs, les différents k_i ne font pas partie du mot compressé. Autrement dit, en ne connaissant que y_i, le décompresseur doit pouvoir deviner le nombre k_i de bits qui doivent être lus dans le mot compressé pour reconstruire x_i. Une solution

correcte mais mauvaise serait, par exemple, de fixer tous les k_i à la constante K ; le décompresseur pourrait alors deviner trivialement les k_i mais le mot résultant ne serait absolument pas compressé.

20. Proposez une façon pour le compresseur de choisir k_i en fonction de x_i et σ_i. Expliquez comment le décompresseur choisit k_i en fonction de y_i et prouvez que ce choix correspond bien à celui du compresseur. Si cela a une importance, expliquez comment le compresseur doit choisir x_0.

On a la contrainte que l'on doit toujours avoir $x_i < 2^B$. Une idée possible est alors de chercher à garantir que l'on a aussi toujours $x_i \geq 2^{B-1}$: ceci minimise le nombre de bits k_i (ce qui est un objectif de la compression) et rend la tâche aisée à la décompression : étant donné y_i, il suffit de choisir k_i (nécessairement unique) tel que la valeur x_i reconstruite soit dans le bon intervalle. Il reste cependant à montrer qu'à la *compression*, il existe toujours pour tout $2^{B-1} \leq x_i < 2^B$ une valeur k_i telle que $x_{i+1} = E_q(\lfloor x_i/2^{k_i} \rfloor, \sigma_i)$ soit dans le même intervalle.
Dans tout ce qui suit, les calculs ne dépendent pas (fondamentalement) de σ, et l'on notera donc simplement q pour $q(\sigma)$ et z la $\sum_{0 \leq j < \sigma} q(k)$ à considérer.

On commence par montrer comment choisir une valeur x_0 telle que x_1 soit comme voulu. On a $x_1 = \lfloor x_0/q \rfloor \times 2^K + (x_0 \mod q) + z$. On peut donc choisir $x_0 = (2^{B-1}/2^K) \times q$, ce qui donne $x_1 = 2^{B-1} + (x_0 \mod q) + z = 2^{B-1} + z < 2^B$ (par $z < 2^K < 2^{B-1}$, puisque B est « bien plus grand que K). (Dans le cas où cette dernière condition n'était en fait pas satisfaite, il suffirait alors de retrancher une petite quantité (par exemple de l'ordre de z) à x_0 avant sa division par 2^K.)

Dans le cas général x_i, on note $x_q = \lfloor \lfloor x_i/2^{k_i} \rfloor / q \rfloor$ et l'on commence par montrer que si $x_q \times 2^K < 2^B$, alors on a aussi $x_q \times 2^K \leq 2^B - 2^K$. Ceci vient directement du fait que $x_q \times 2^K < 2^B$ implique $x_q < 2^{B-K}$, donc $x_q \leq 2^{B-K} - 1$, d'où $x_q \times 2^K \leq 2^B - 2^K$.
De par $x_i \mod q + z < 2^K$, ceci implique que pour un tel x_q l'on a $E_q(y_i, \sigma_i) < 2^B$ comme voulu. Il suffit alors de choisir k_i de façon à ce que $2^{B-1} \leq x_q \times 2^K < 2^B$ pour conclure, et l'on va montrer que $k_i = -\lceil \log q \rceil + K$ convient.
Dans ce cas, par hypothèse que $2^{B-1} \leq x_i < 2^B$ l'on a $\lfloor x_i/2^{k_i} \rfloor = 2^{B-1+\lceil \log q \rceil - K} + r_1$ pour un certain $r_1 < 2^{B-1+\lceil \log q \rceil - K}$, d'où $x_q = 2^{B-1-K} + r_2$ avec $r_2 < 2^{B-1-K}$, et $x_q \times 2^K = 2^{B-1} + r_3$ avec $r_3 < 2^{B-1}$, et donc $2^{B-1} \leq x_q \times 2^K < 2^B$.

Remarque : la contrainte imposant que N soit une puissance de deux ne pose aucune difficulté en pratique. En effet, en matière de compression entropique, la seule chose qui importe est que $q(\sigma)/N$ soit proche de la proportion de la lettre σ dans le mot à compresser.

Fin du sujet