Compression entropique - X ENS A 2023

Pour toutes les remarques ou corrections, vous pouvez m'envoyer un mail à galatee.hemery@gmail.com

1 Partie I. Comptable d'occurrences

Question 1.1. On utilise une fonction récursive qui parcours le mot et complète un tableau de 256 cases remplies de 0.

```
let occurences (l : int list) : int array =
    let q = Array.make 256 0 in
    let rec parcours w = match w with
        |[] -> q
        |t::suite -> q.(t)<-q.(t) +1 ; parcours suite
    in parcours l ;;
```

Question 1.2. On parcours la le tableau depuis 0 à l'aide d'une fonction récursive qui renvoie la liste des couples attendues. Ensuite, on utilise la fonction donnée en rappel pour transformer le résultat en tableau.

```
let nonzero_occurrences (tab : int array) : (int * int) array =
    let n = Array.length tab in
    let rec test_zero i = match i with
        |i when i = n -> []
        |i when tab.(i) = 0 -> test_zero (i+1)
        |i -> (i,tab.(i))::(test_zero (i+1))
    in Array.of_list (test_zero 0) ;;
```

2 Partie II. Arbres binaires

Question 2.1. Pour tout ensemble de 1 élément, on a un unique arbre : l'arbre feuille.
Pour un ensemble à 2 éléments, on a deux arbres possibles correspondant à l'unique partition en deux ensembles non vide : $N(F(x_1), F(x_2))$ et $N(F(x_2), F(x_1))$.
Pour un ensemble à 3 éléments, on a 3 partitions possibles en deux sous-ensembles non vides. Pour chaque partition, on a un singleton et un ensemble à deux éléments soit 1 et 2 arbres possibles et deux ordre possibles (gauche droite ou droite gauche) donc 4 possibilités. En tout, on a 12 possibilités.
Enfin, pour un ensemble à 4 éléments, on a 4 partitions mobilisant un singleton et un ensemble à 3 éléments, qui correspondent chacunes à $2 \times 12 = 24$ arbres, soit $4 \times 24 = 96$ abres.
On a également $\frac{1}{2}\binom{4}{2} = 3$ partitions avec deux ensembles à deux éléments, correspondant à $2 \times 2 \times 2 = 8$ abres, soit $8 \times 3 = 24$ arbres.
On en compte bien 120 au total.

Question 2.2. On utilise une fonction `parcours` récursive qui explore les deux partie l'arbre et additionne les valeurs obtenue en conservant dans un argument la profondeur courante. Chaque noeud est exploré une unique fois, il y a $|S|$ feuilles et donc $2|S| - 1$ noeuds dans l'arbre, donc la complexité est en $O(|\mathcal{S}|)$.

```
let cq (t:tree) (q: int array) : int =
    let rec parcours profondeur t = match t with
        |F x -> profondeur * q.(x)
```

```
        |N (g,d) -> parcours (profondeur+1) g + parcours (profondeur+1) d
    in parcours 0 t ;;
```

Question 2.3. On réalise un parcours en profondeur jusqu'à rencontrer la feuille correspondant à σ. La complexité d'un tel parcours est en $O(|\mathcal{S}|)$.

```
exception WrongPath ;;

let get_path (s : int) (t : tree) : int list =
    let rec parcours a = match a with
        |F x when x = s -> []
        |F x -> raise WrongPath
        |N (g,d) -> try
                        0::(parcours g)
                    with
                        WrongPath -> 1::(parcours d)
    in parcours t ;;
```

Question 2.4. La fonction `arbre_complet` permet de construit un arbre dont les chemins vers les feuilles sont exactement toutes les séquences de longueur k.
La fonction `construit` construit récursive l'arbre en traitant chaque valeur de k possible.

```
let integers (l : int) : tree =
    let compteur = ref 0 in
    let rec arbre_complet k =
        if k = 0 then (incr compteur; F (!compteur-1))
        else begin
            let g = arbre_complet (k-1) in
            let d = arbre_complet (k-1) in
            N(g,d)
        end in
    let rec construit k =
        if k = -1 then arbre_complet 0
        else begin
            let g = arbre_complet (l-k) in
            let d = construit (k-1) in
            N (g,d)
        end
    in construit l;;
```

3 Partie III. Codes préfixes

Question 3.1. Soit t un arbre de $\mathcal{T}_\mathcal{S}$ Supposons par l'absurde que pour une séquence $s = s_1...s_k$ de bits il existe deux mots $u = u_1...u_n$ et $v = v_1...v_m$ représentés.
On remarque que u et v ne peuvent pas être préfixe l'un de l'autre sinon le préfixe serait représenté par une séquence plus courte.
Ainsi, on peut considèrer i le premier indice tel que $u_i \neq v_i$ et on considère l'indice j tel que $s_1...s_{j-1}$ représente $u_1...u_{i-1} = v_1....v_{i-1}$.
Ainsi un préfixe s' de $s_j...s_k$ représente u_i et un autre préfixe s'' de $s_j...s_k$ représente v_i.
Quitte à échanger les rôles de u et v, on peut supposer que s' est préfixe stricte de s''.
Ainsi, dans l'arbre t, s' mène nécessairement à un noeud interne et non une feuille car la branche correspondant à s'' existe. s' ne correspond donc pas à un élément de $\mathcal{S}$.
CONTRADICTION

Question 3.2. On parcourt la séquence et l'arbre simultanément. A chaque fois qu'une feuille est atteinte, on récupère le symbole correspondant et on continue en reprenant la racine de l'arbre. La complexité est en $O(|s|)$ où $|s|$ est le nombre de bits de la séquence.

```
let decomp1 s t =
    let rec parcours seq arbre = match (seq,arbre) with
        |([], F sigma) -> [sigma]
        |(_,F sigma) -> sigma :: (parcours seq t)
        |(0::suite , N (g,d)) -> parcours suite g
        |(1::suite , N (g,d)) -> parcours suite d
        |_ -> failwith "Sequence invalide"
    in parcours s t ;;
```

4 Partie IV. Arbres optimaux

Question 4.1. Il est clair que g est un arbre de $\mathcal{T}_{\mathcal{S}_g}$ car tous les éléments de $\mathcal{S}_g$ sont représentées sur une unique feuille et g a la même structure que t.
Supposons par l'absurde que g ne soit pas optimal pour $(q, \mathcal{S}_g)$.
Alors il existe g' un arbre optimal tel que $c_q(t') < c_q(t)$.
On considère $t' = N(g', d)$ un arbre de $\mathcal{T}_{\mathcal{S}}$.
On note $\mathcal{S}_d$ 'ensemble des symboles apparaissant dans d.
On a $c_q(t') = \sum_{\sigma\in\mathcal{S}} q(\sigma)\ell_t(\sigma) = \sum_{\sigma\in\mathcal{S}_g} q(\sigma)\ell_t(\sigma) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)\ell_t(\sigma)$
Puis $c_q(t') = \sum_{\sigma\in\mathcal{S}_q} q(\sigma)(1 + \ell_{g'}(\sigma)) + \sum_{\sigma\in\mathcal{S}_d} q(\sigma)(1 + \ell_d(\sigma))$
Ainsi, $c_g(t') = c_q(g') + c_q(d) + \sum_{\sigma\in\mathcal{S}} q(\sigma) < c_q(g) + c_q(d) + \sum_{\sigma\in\mathcal{S}} q(\sigma) = c_q(t)$ de même.
Donc t n'est pas optimal, CONTRADICTION.

Question 4.2. Remarque préliminaire : Soit t_0 un arbre optimal pour $(q, \mathcal{S} \setminus \{\sigma_0\})$.
$t_1 = N(F(\sigma_0), t_0)$ est arbre de $\mathcal{T}_{\mathcal{S}}$ tel que
$c_q(t_1) = q(\sigma_0) + \sum_{\sigma\in\mathcal{S}\setminus\{\sigma_0\}} q(\sigma) + c_q(t_0) < 2q(\sigma_0) + c_q(t_0)$.

Raisonnons par récurrence forte sur $|\mathcal{S}|$.
Initialisation :
Pour $|\mathcal{S}| = 2$, on a deux arbres possibles et optimaux $N(F(\sigma_0), F(\sigma_1))$ et $N(F(\sigma_1), F(\sigma_0))$ tel que $\ell_t(\sigma_0) = 1$.
Hérédité :
On suppose que le résultat vrai pour $|\mathcal{S}| \leq n$.
Soit $\mathcal{S}$ un ensemble et q une fonction tel que $\sigma_0 \in \mathcal{S}$ vérifie $q(\sigma_0) > \sum_{\sigma\in\mathcal{S}\setminus\{\sigma_0\}} q(\sigma)$.
Soit t un arbre optimal pour $(q, \mathcal{S})$.
On a $t = N(g, d)$. Avec g, d arbres optimaux pour $(q, \mathcal{S}_g)$ et $(q, \mathcal{S}_d)$ en utilisant les notation et le résultat de la question précédente.
Sans perte de généralité, quitte à considérer $N(d, g)$, on peut supposer que $\sigma_0 \in \mathcal{S}_g$.
Si $\mathcal{S}_g = \{\sigma_0\}$, on a $g = F(\sigma_0)$ et $\ell_t(\sigma_0) = 1$.
Sinon, par hypothèse de récurrence, $\ell_g(\sigma_0) = 1$ donc $\ell_t(\sigma_0) = 2$.
Donc $g = N(F(\sigma_0), d')$ et d' optimal pour $(q, \mathcal{S}_g \setminus \{\sigma_0\})$.
Ainsi $c_q(t) = 2q(\sigma_0) + 2\sum_{\sigma\in\mathcal{S}_g\setminus\{\sigma_0\}} q(\sigma) + c_q(d') + \sum_{\sigma\in\mathcal{S}_d} q(\sigma) + c_q(d)$.
On a $c_q(t_0) \leq c_q(N(d', d)) = c_q(d') + c_q(d) + \sum_{\sigma\in\mathcal{S}\setminus\{\sigma_0\}} q(\sigma)$.
Ainsi $c_q(t) \geq 2q(\sigma_0) + c_q(t_0) > c_q(t_1)$ donc t n'est pas optimal.
On est donc nécessairement dans le premier cas, ce qui permet de conclure.

Question 4.3.

1. Montrons d'abord que σ_1 et σ_2 peuvent être de profondeur maximale dans t optimale. Supposons par l'absurde que ce n'est pas le cas, alors pour tout arbre optimal t, il existe $\sigma \notin \{\sigma_1, \sigma_2\}$ tel que $\ell_t(\sigma_1) < \ell(\sigma)$ (quitte à échanger les rôles de σ_1 et σ_2).
   Alors en intervertissant $F(\sigma_1)$ et $F(\sigma)$ on obtient un arbre t' tel que
   $c_q(t') = c_q(t) - q(\sigma_1)\ell_t(\sigma_1) - q(\sigma)\ell_t(\sigma) + q(\sigma)\ell_t(\sigma_1) + q(\sigma_1)\ell_t(\sigma)$
   $c_q(t') = c_q(t) - (q(\sigma_1) - q(\sigma))(\ell_t(\sigma_1) - \ell_t(\sigma)) \leq c_q(t)$ CONTRADICTION
   Ainsi, comme l'arbre est binaire stricte, on a au moins deux noeuds de profondeur maximal et chaque feuille a une feuille frère.
   Quitte à intervertir les feuilles de même profondeur, ce qui ne change pas la valeur de $c_q(t)$, on peut considérer l'arbre où $F(\sigma_1)$ et $F(\sigma_2)$ ont le même père.
   On a trouvé l'arbre optimal attendu.
2. On a $c_{q'}(t') = c_q(t) - q(\sigma_1)\ell_t(\sigma_1) - q(\sigma_2)\ell_t(\sigma_2) + q'(\sigma_3)(\ell_t(\sigma_1) - 1)$ car $\ell_{t'}(\sigma_3) = \ell_t(\sigma_1) - 1 = \ell_t(\sigma_2) - 1$.
   Ainsi, $c_q(t) = c_{q'}(t') + q(\sigma_1) + q(\sigma_2)$.
   A partir de l'arbre optimal de la question précédente, que l'on renomme ici t_2, on peut remplacer $N(F(\sigma_1), F(\sigma_2))$ par $F(\sigma_3)$ pour obtenir un arbre t_2' de $\mathcal{T}_{\mathcal{S}'}$ tel que $c_q(t_2) = c_{q'}(t_2') + q(\sigma_1) + q(\sigma_2)$.
   Par optimalité de t' on a $c_{q'}(t_2') \geq c_{q'}(t')$ et par optimalité de t_2, on a $c_q(t_2) \leq c_q(t)$.
   Ainsi $c_q(t) = c_q(t_2)$ et t est optimal.

Question 4.4.

1. ```
   let rec insert (c : int * tree) (l : (int * tree) list) : (int * tree) list =
       let i,t = c in
       match l with
           |[]-> [c]
           |(j,a)::suite when i<j -> c::l
           |(j,a)::suite -> (j,a)::(insert c suite) ;;
   ```
2. ```
   let optimal (l : (int * int) list) : tree =
       let rec cree_feuille lc = match lc with
           |[] -> []
           |(sigma,qs)::q -> insert (qs, F sigma) (cree_feuille q)
       in
       let rec fusion lt = match lt with
           |[] -> failwith "Alphabet vide"
           |[(qt,t)] -> t
           |(qt1,t1)::(qt2,t2)::suite -> fusion (insert (qt1+qt2,N(t1,t2)) suite)
       in fusion (cree_feuille l) ;;
   ```
3. La fonction `optimal` crée et trie d'abord tous les arbres feuilles. L'insertion se fait en $O(|l|)$ où $|l|$ est le nombre d'élément dans la liste.
   Ici, on a appelle `insert` $|\mathcal{S}|$ fois et la liste est de taille croissante jusqu'à $|\mathcal{S}|$, on a donc une complexité temporelle en $O(|\mathcal{S}|^2)$ pour `cree_feuille`.
   On fusionne ensuite les arbres deux par deux, on doit le faire $|\mathcal{S}|-1$ fois pour obtenir un unique arbre. On fait appel à `insert` à chaque étape pour conserver une liste triée. Ainsi, `fusion` a une complexité temporelle en $O(|\mathcal{S}|^2)$.
   Finalement, la complexité de `optimal` est en $O(|\mathcal{S}|^2)$.

Question 4.5. Soit $t = N(t_1, t_2)$ et $t' = N(t_1', t_2')$ deux arbres ajoutés à E tel que t a été ajouté avant t'.
On a alors $\bar{q}(t_1) \leq \bar{q}(t_1')$, $\bar{q}(t_1) \leq \bar{q}(t_2')$, $\bar{q}(t_2) \leq \bar{q}(t_1')$ et $\bar{q}(t_2) \leq \bar{q}(t_2')$ car t_1 et t_2 sont choisis minimisant $\bar{q}(t)$ dans E et les arbres ajoutés après ont tous de plus grandes valeurs

de $\bar{q}(t)$ car ils sont issus de deux arbres de plus grande valeur.
Ainsi, $\bar{q}(t) = \bar{q}(t_1) + \bar{q}(t_2) \leq \bar{q}(t') = \bar{q}(t'_1) + \bar{q}(t'_2)$.
Ils sont bien ajoutés par valeur croissante.

Question 4.6. L'appel à `insert` n'est pas nécessaire à chaque étape de fusion.
On conserve dans une liste les feuilles non fusionnées qui peuvent être généré en temps linéaire à partir de la liste triée et on conserve les arbres non feuilles que l'on ajoute dans une nouvelle file initialement vide.
Ainsi, les nouveaux arbres sont ajoutés en fin de file et on peut consulter les premiers éléments de la file et de la liste pour savoir quels arbres fusionnés.
Ainsi, la complexité sera linéaire car chaque fusion prend un temps constant et non linéaire.

5 Partie V. Arbres canoniques

Question 5.1. La complexité temporelle est en $O(|S|)$ car le nombre de noeud est $2|\mathcal{S}|-1$ et que l'on appelle la fonction auxiliaire une fois pour chaque noeud et que les vérifications et calculs qu'elle fait à chaque appel se font en temps constant.
Il s'agit de construire l'arbre de gauche à droite. On connaît les profondeurs de chaque élément, on garde en mémoire la profondeur cible courante traitée, un curseur indiquant la place dans l'ordre, le nombre de feuilles de la profondeur cible courante qu'il reste à créer, ainsi que la profondeur courante du noeud actuel.

```
let canonical (nmb_p : int array) (ordre : int array) : tree =
    let n = Array.length ordre in
    let rec construit p_cur p_cible curseur np =
        if curseur = n-1 then
            (F ordre.(curseur),curseur +1,p_cible, np-1)
        else begin
            if np = 0 then
                construit p_cur (p_cible+1) curseur (nmb_p.(p_cible+1))
            else begin
                if p_cur = p_cible then
                    (F (ordre.(curseur)),curseur +1,p_cible,np-1)
                else begin
                    let (t1,c1,pc,npc) =
                        construit (p_cur +1) p_cible curseur np in
                    let (t2,c2,pc2,npc2) =
                        construit (p_cur +1) pc c1 npc in
                    (N(t1,t2),c2,pc2,npc2)
                end
            end
        end
    in let t,c,pc,npc = construit 0 0 0 (nmb_p.(0))
    in t;;
```

6 Partie VI. Arbres alphabétiques

Question 6.1. On a $m_{i,j} = \min_{k\in[i+1,j]}(m_{i,k-1} + m_{k,j} + \sum_{\sigma\in[i,j]} q(\sigma))$.
On peut ainsi écrire un algorithme de programmation dynamique qui précalcule les $m_{i,j}$ puis reconstruit l'arbre à partir du tableau en respectant les découpages obtenus.

```
let min a b = if a < b then a else b ;;
```

```
let q_bar (q : int array) : int array array =
    let n = Array.length q in
    let qbar = Array.make n [||] in
    for i=0 to (n-1) do
        qbar.(i)<- Array.make n 0 ;
        for j = i to (n-1) do
            for k = i to j do
                qbar.(i).(j)<- qbar.(i).(j) + q.(k)
            done;
        done;
    done;
    qbar;;

let alpha_optimal (q: int array) : tree =
    let n = Array.length q in
    let qbar = q_bar q in
    let m = Array.make n [||] in
    for i=0 to (n-1) do
        m.(i)<- Array.make n 0
    done;
    for e = 1 to (n-1) do
        for i = 0 to (n-1-e) do
            m.(i).(i+e) <- m.(i).(i)+m.(i+1).(i+e) + qbar.(i).(i+e) ;
            for k = i+2 to i+e do
                m.(i).(i+e) <-
                    min (m.(i).(i+e))
                    (m.(i).(k-1)+m.(k).(i+e) + qbar.(i).(i+e))
            done;
        done;
    done;
    let rec construit i j k =
        if i = j then F i else begin
            if  m.(i).(j) = m.(i).(k-1)+m.(k).(j) + qbar.(i).(j) then begin
                let t1 = construit i (k-1) (i+1) in
                let t2 = construit k j (k+1)
                in N(t1,t2)
            end
            else construit i j (k+1)
        end
    in construit 0 (n-1) 1 ;;
```

Question 6.2.

```
let alpha (s : int array) : tree =
    let l = Array.length s in
    let rec construit curseur nmb =
        if curseur > l -1 then failwith "Echec" ;
        if s.(curseur) = 1 then (F nmb,curseur +1, nmb +1)
        else begin
            let g,c,n = construit (curseur +1) nmb in
            let d,c2,n2 = construit c n in
            (N (g,d), c2,n2)
```

```
        end
    in let t,c,n = construit 0 0
    in if c!= l then failwith "Echec" else t;;
```

7 Partie VII. Codes arithmétiques

Question 7.1.

1. Selon la question 4.2, A est à profondeur 1 dans un arbre optimal. On en déduire que $t = N(F(A), N(F(B), F(C))$ est un arbre optimal.
   Le nombre de bits nécessaire est donc $c_q(t) = 15 \times 1 + 4 \times 2 + 1 \times 2 = 25$.
2. Le mot est de longueur $15 + 4 + 1 = 20$.
   — On choisit d'abord la place du seul C : $20 = 5 \times 2^2$ possibilités.
   — Il reste 19, on place ensuite les 4 B : $\binom{19}{4} = \frac{19\times18\times17\times16}{2\times3\times4} = 19 \times 17 \times 3 \times 2^2$.
   — Finalement, on place les A sur les 15 places restantes.
   On a donc $|\mathcal{S}^q| = 2^4 \times 3 \times 5 \times 17 \times 19$.
3. En utilisant la remarque et $19 < 32$, on a $|\mathcal{S}^q| - 1 = 2^4 \times (2^8 - 1) \times 19 - 1 < 32 \times 2^{12} = 2^{17}$.
   Ainsi, 17 bits suffisent effectivement.

Question 7.2. Soit y un entier, pour trouver le couple X, σ tel que $E_q(X, \sigma) = y$, on utilise N et les sommes de $q(\sigma)$ cumulées.
En effet

$$y = E(X, \sigma) = \left\lfloor \frac{X}{q(\sigma)} \right\rfloor \cdot N + (X \mod q(\sigma)) + \sum_{0 \le k < \sigma} q(k)$$

Ainsi $\left\lfloor \frac{X}{q(\sigma)} \right\rfloor = \lfloor \frac{y}{N} \rfloor$ car $0 \le (X \mod q(\sigma)) + \sum_{0 \le k < \sigma} q(k) < \sum_{0 \le k \le \sigma} q(k) \le N$.
Le reste de la division euclidienne est

$$\sum_{0 \le k < \sigma} q(k) \le (X \mod q(\sigma)) + \sum_{0 \le k < \sigma} q(k) < \sum_{0 \le k \le \sigma} q(k).$$

On peut donc en déduire σ en comparant les sommes cumulées.
Puis $X = \left\lfloor \frac{X}{q(\sigma)} \right\rfloor \times q(\sigma) + (X \mod q(\sigma))$.

```
let q_sum (q : int array) : int array =
    (* calcule les sommes cumulées utilisées dans Eq *)
    let n = Array.length q in
    let qs = Array.make n 0 in
    qs.(0) <- q.(0) ;
    for i = 1 to n-1 do
        qs.(i) <- q.(i) + qs.(i-1)
    done;
    qs;;

let cherche_sigma (qs : int array) (reste : int) : int =
    let n = Array.length qs in
    let rec test i =
        if (qs.(i) > reste) || (i = n-1) then i
        else test (i+1)
    in test 0 ;;
```

```
let decomp2 (q : int array) (y : int) (k : int) : int * int array =
    let sigma = Array.make k (-1) in
    let qs = q_sum q in
    let n = Array.length q in
    let grand_n = qs.(n-1) in
    let rec complete y_cur indice =
        let partie_entiere = y_cur / grand_n in
        let reste = y_cur mod grand_n in
        let s = cherche_sigma qs reste in
        sigma.(indice) <- s ;
        if s = 0 then begin
            let x = partie_entiere * q.(s) + reste in
            if indice = 0 then x
            else complete x (indice-1)
        end
        else begin
            let x = partie_entiere * q.(s) + reste - qs.(s-1) in
            if indice = 0 then x
            else complete x (indice-1)
        end
    in (complete y (k-1),sigma) ;;
```

Question 7.3. A chaque étape, on veut que

$$x_{i+1} = E_q(y_i, \sigma_i) = \left\lfloor \frac{y_i}{q(\sigma_i)} \right\rfloor \cdot 2^K + (y_i \mod q(\sigma_i)) + \sum_{0 \leq k < \sigma_i} q(k) < 2^B$$

Pour cela, il suffit que $\left\lfloor \frac{y_i}{q(\sigma_i)} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{x_i}{2^{k_i}} \right\rfloor}{q(\sigma_i)} \right\rfloor < 2^{B-K}$.

Il suffit que $\frac{x_i}{2^{k_i} q(\sigma_i)} < 2^{B-K}$.

Toute valeur $k_i \geq \max\left(\log_2\left(\frac{x_i}{q(\sigma_i)}\right) + K - B, 0\right)$ convient.

Néanmoins, le choix de k_i doit permettre de retrouver k_i et x_i à partir de $y_i = \left\lfloor \frac{x_i}{2^{k_i}} \right\rfloor$ et la séquence.

On prend idéalement $k_i = \max\left(\left\lfloor \log_2\left(\frac{x_i}{q(\sigma_i)}\right)\right\rfloor + 1 + K - B, 0\right)$.

Or on sait que $x_i \leq 2^B$, on peut prendre $k_i = \max\left(\left\lfloor \log_2\left(\frac{2^B}{q(\sigma_i)}\right)\right\rfloor + 1 + K - B, 0\right)$ On retrouve alors k_i simplement à l'aide de σ_i que l'on peut déduire de $y_i = E_q(x_i, \sigma_i)$.