DS 3 d'option informatique - MP* - Sujet Mines-Ponts 2020 - Correction

Question 1 –

⋆ On a `fonction1: texte -> int array`. En effet, le type du paramètre `t` a été forcé et le type de retour est `int array` à cause de l'appel à `array_make` (ligne 3).

⋆ Montrons par récurrence sur `|t|` que « `fonction1 t` » s'évalue en un tableau `theta` de taille `lambda`, tel que pour tout $u \in [\![0; \texttt{lambda} - 1]\!]$, `theta.(u)` est le nombre d'occurrences de `u` dans `t`.

- Initialisation. Si `|t| = 0`, alors pour tout $u \in [\![0; \texttt{lambda} - 1]\!]$, l'entier `u` apparaît 0 fois dans `t`. Dans ce cas, `fonction1` renvoie un tableau de taille `lambda` ne contenant que des 0 (voir le premier cas du filtrage), ce qui est correct.
- Hérédité. Soit $n \in \mathbb{N}$. On suppose la propriété vraie au rang `n` et on la montre dans le cas où `|t| = n + 1`. Comme `t` n'est pas la liste vide, si on décompose `t = u :: t'` (comme dans `fonction1`), et qu'on note `theta'` le tableau renvoyé par « `fonction1 t'` », alors d'après le code de la fonction :

$$\begin{cases} \texttt{theta.(u)} = \texttt{theta}'\texttt{.(u)} + 1, \\ \texttt{theta.(u')} = \texttt{theta}'\texttt{.(u')}, \forall \texttt{u}' \neq \texttt{u}. \end{cases}$$

  Or, le nombre d'occurrences de `u` dans `t` est effectivement égal à $\texttt{theta}'\texttt{.(u)} + 1$ (une de plus que dans `t'`) et pour tout $\texttt{u}' \neq \texttt{u}$, le nombre d'occurrences de `u'` dans `t` est effectivement égal $\texttt{theta}'\texttt{.(u')}$ (le même nombre que dans `t'`).

Ainsi, le résultat est vrai pour tout `t`.

Question 2 –

```
let cree_repartition (t: texte): repartition =
  let theta = fonction1 t in
  let rec to_list (u: int) =
    if u = array_length theta then []
    else if theta.(u) = 0 then to_list (u+1)
    else (u, theta.(u)) :: to_list (u+1) in
  to_list 0;;
```

Question 3 –

⋆ Dans un premier temps, il faut déterminer la complexité en temps de `fonction1`. On remarque que la taille de la liste en entrée diminue de 1 à chaque appel récursif. L'appel à la fonction `array_make` s'exécute en temps $\mathcal{O}(\lambda)$ et les autres opérations en $\mathcal{O}(1)$.

En résumé, si on note `|t|` la taille de `t`, il y a $\mathcal{O}(|t|)$ appels à `fonction1`, chaque appel se fait en temps $\mathcal{O}(1)$ (plus l'appel récursif) sauf le dernier appel qui se fait en temps $\mathcal{O}(\lambda)$. Donc `fonction1` a une complexité en $\mathcal{O}(\lambda + |t|)$.

⋆ Pour la fonction `cree_partition`, l'appel à `fonction1` se fait en temps $\mathcal{O}(\lambda + |t|)$. Pour la fonction `to_list`, il y a $\mathcal{O}(\lambda)$ appels récursifs et toutes les opérations s'exécutent en temps constant (sauf l'appel récursif). Ainsi, la complexité en temps de `to_list` est en $\mathcal{O}(\lambda)$ et donc :

La complexité en temps de `cree_repartition` est en $\mathcal{O}(\lambda + \|t\|)$

Question 4 – La fonction `cree_repartition` créé deux objets : le tableau `theta` de taille λ (renvoyé par `fonction1`) et la liste renvoyée par `to_list` qui est de taille au plus λ. Ainsi :

La complexité en espace de `cree_repartition` est en $\mathcal{O}(\lambda)$.

Remarque. À priori, la complexité en espace de `fonction1` devrait être $\mathcal{O}(\lambda + |t|)$ à cause de la pile d'exécution. L'énoncé demande une complexité en fonction de λ seulement. J'imagine qu'il ne faut pas compter la complexité en espace due à la pile d'exécution.
Le rapport du jury indique que les compilateurs récents d'OCaml transforment `fonction1` en fonction récursive terminale. Il me semble difficile d'utiliser ce genre d'argument le jour d'un concours ; d'autant plus que comme l'indique le rapport du jury, la récursivité terminale n'est plus au programme.

Question 5 – La valeur de retour est une liste contenant autant de couples qu'il y a de lettres différentes dans le texte donné en entrée. Ainsi :

La taille de « `cree_repartition t` » est en $\mathcal{O}([w])$ où $[w]$ est la valence du mot w associé au texte `t`

Question 6 – Pour un courriel de la vie courante, on a :

- $\lambda = 1\,114\,112$: taille de l'alphabet unicode.
- $|t| \approx 1000$: ordre de grandeur du nombre de caractères dans le courriel.
- $[w] \approx 100$: ordre de grandeur du nombre de caractères différents dans le courriel (en comptant les lettres minuscules, majuscules, accentuées, la ponctuation ...).

Ainsi, les complexités en temps et en espace de `cree_partition` sont approximativement 10^3 fois plus grandes que la taille de la liste renvoyée par cette fonction. En conclusion :

La complexité de `cree_partition` ne semble pas optimale si on la compare à la taille de l'objet obtenu en sortie.

Question 7 –

```
let rec incremente_repartition (rep: repartition) (u: unicode): repartition =
  match rep with
  | [] -> [(u,1)]
  | (v,k) :: q when v <> u -> (v,k) :: incremente_repartition q u
  | (v,k) :: q -> (v,k+1) :: q;;
```

Question 8 –

```
let rec cree_modulaire (t: texte) (m: int): repartition array = match t with
  | [] -> array_make m []
  | u :: tprime -> let tab_mod = cree_modulaire tprime m in
                   tab_mod.(u mod m) <- incremente_repartition tab_mod.(u mod m) u;
                   tab_mod;;
```

Remarque. Extrait du rapport du jury : "il n'est pas pertinent d'implanter des fonctions auxiliaires récursives terminales".

Question 9 –

```
let valence (tab_mod: repartition array): int =
  let s = ref 0 in
  for i = 0 to array_length tab_mod - 1 do
    s := !s + list_length tab_mod.(i)
  done;
  !s;;
```

Question 10 –

Remarque : la notion d'espérance conditionnelle est hors-programme, cette question n'aurait pas dû être posée le jour du concours. Extrait du rapport du jury : "les correcteurs ont décidé de ne pas sanctionner l'ensemble des candidats sur cette question".

On fixe $\ell \in [\![0, m-1]\!]$. Pour tout $i \in [\![1, v]\!]$, on note Y_i la variable aléatoire qui vaut 1 si $X_i = \ell$ et 0 sinon. Alors :

$$Z_\ell = \sum_{i=1}^{v} Y_i.$$

Par linéarité de l'espérance :

$$\begin{aligned}\mathbb{E}[Z_\ell | X_{i_0} = \ell_0] &= \sum_{i=1}^{v} \mathbb{E}[Y_i | X_{i_0} = \ell_0] \\ &= \mathbb{E}[Y_{i_0} | X_{i_0} = \ell_0] + \sum_{\substack{i \in [\![1;v]\!] \\ i \neq i_0}} \mathbb{E}[Y_i | X_{i_0} = \ell_0]\end{aligned}$$

Pour tout i :

$$\begin{aligned}\mathbb{E}[Y_i | X_{i_0} = \ell_0] &= 0 \times \mathbb{P}[Y_i = 0 | X_{i_0} = \ell_0] + 1 \times \mathbb{P}[Y_i = 1 | X_{i_0} = \ell_0] \\ &= \mathbb{P}[X_i = \ell | X_{i_0} = \ell_0]\end{aligned}$$

Comme les X_i sont mutuellement indépendantes et suivent la loi de distribution uniforme, pour tout $i \neq i_0$:

$$\mathbb{P}[X_i = \ell | X_{i_0} = \ell_0] = \mathbb{P}[X_i = \ell] = \frac{1}{m}$$

Pour $i = i_0$, on a deux cas :

$$\begin{cases} \text{Si } \ell = \ell_0 : & \mathbb{P}[X_i = \ell | X_{i_0} = \ell_0] = 1 \\ \text{Si } \ell \neq \ell_0 : & \mathbb{P}[X_i = \ell | X_{i_0} = \ell_0] = 0 \end{cases}$$

Finalement :

$$\boxed{\begin{cases} \text{Si } \ell = \ell_0 : & \mathbb{E}[Z_\ell | X_{i_0} = \ell_0] = 1 + \dfrac{v-1}{m} \\ \text{Si } \ell \neq \ell_0 : & \mathbb{E}[Z_\ell | X_{i_0} = \ell_0] = \dfrac{v-1}{m} \end{cases}}$$

Question 11 –

Remarque : la notion de complexité moyenne est hors-programme, cette question n'aurait pas dû être posée le jour du concours. Extrait du rapport du jury : "les correcteurs ont décidé de ne pas sanctionner l'ensemble des candidats sur cette question".

Le temps d'exécution de la fonction `incremente_repartition` est en $\mathcal{O}(k)$ avec k la taille de la liste donnée en entrée. Il s'agit donc de déterminer la taille moyenne des listes données à `incremente_repartition` lors de l'exécution de `cree_modulaire`.

Soient $U_1, \ldots, U_v$ les variables aléatoires dont les valeurs sont les v entiers distincts présents dans le texte `t` (pour fixer les idées, disons que $U_1 < U_2 < \ldots < U_v$). Pour tout i, on définit $X_i = U_i \mod m$. En suivant l'hypothèse de l'énoncé, on suppose que les X_i sont à valeurs dans $[\![0, m-1]\!]$, qu'elles sont mutuellement indépendantes et suivent la loi de distribution uniforme. À partir des X_i, on définit les variables aléatoires Z_ℓ comme dans l'énoncé de la question 10.

Soit u_0 un élément de `t`, soit $\ell_0 = u_0 \mod m$ et i_0 l'unique entier tel que $U_{i_0} = u_0$ (alors $X_{i_0} = \ell_0$). Lorsqu'on considère u_0 dans l'exécution de `cree_modulaire`, on a un appel à la fonction `incremente_repartition` sur une liste de taille au plus Z_{ℓ_0}. D'après la question précédente, en moyenne, Z_{ℓ_0} vaut :

$$\mathbb{E}[Z_{\ell_0} | X_{i_0} = \ell_0] = 1 + \frac{v-1}{m}.$$

Ainsi, pour chaque élément u_0 de `t`, le temps d'exécution moyen de l'appel à `incremente_repartition` est en $\mathcal{O}(1 + \frac{v-1}{m})$. Si on considère tous les éléments de `t`, on obtient une complexité moyenne en $\mathcal{O}(|t| + \frac{v-1}{m}|t|)$. Si on ajoute l'appel à `array_make` au début de la fonction :

La complexité moyenne en temps de `cree_modulaire t m` est $\mathcal{O}\left(m + |t| + \dfrac{(v-1)|t|}{m}\right)$

Question 12 – Si on note v la valence de t :

La complexité en espace de `cree_modulaire t m` est $\mathcal{O}(m+v)$

Remarque. Le rapport du jury indique que les compilateurs modernes savent gérer les fonctions récursives non terminales. Mon interprétation de cette remarque est qu'il ne faut pas compter la taille de la pile d'exécution dans la complexité en espace.

Question 13 – Pour la complexité en espace, on remarque que pour tout m : $m+v \geqslant v$. Pour la complexité en temps, on remarque que :

$$\begin{cases} \text{Si } m \geqslant v : m + |t| + \dfrac{(v-1)|t|}{m} \geqslant v + |t| \\ \text{Si } m \leqslant v : m + |t| + \dfrac{(v-1)|t|}{m} \geqslant m + |t| + \dfrac{(v-1)v}{v} \geqslant |t| + v \end{cases}$$

Ce qui précède suggère que la complexité en espace optimale est $\mathcal{O}(v)$ et que la complexité en temps optimale est $\mathcal{O}(v+|t|)$. Avec $m = v$, on atteint ces deux complexités. En réutilisant les ordres de grandeur donnés dans la question 6 :

$m \approx 100$ semble judicieux.

Question 14 – Soit $\Sigma' = \{\sigma_0, \sigma_1, \ldots, \sigma_{v-1}\}$ et Σ_w l'ensemble des lettres présentes dans le mot w. Par le point (ii), $\Sigma_w \subset I(\Sigma')$. De plus : $|I(\Sigma')| \leqslant |\Sigma'| = v = |\Sigma_w|$. Ainsi, on a nécessairement $I(\Sigma') = \Sigma_w$. En d'autres termes, si $\tau \leqslant \sigma_{v-1}$, alors $I(\tau)$ est une lettre présente dans le mot w.

Question 15 – On utilise les notations de la questions précédente pour montrer la propriété par double implication.

($\Rightarrow$) Soit $\sigma \in \Sigma_w$.
Par le point (ii) : il existe $\tau \in \Sigma'$ telle que $I(\tau) = \sigma$.
Par le point (iii) : $A(I(\tau)) = \tau$.
Ainsi : $A(\sigma) = A(I(\tau)) = \tau \in \Sigma'$ et $I(A(\sigma)) = I(\tau) = \sigma$.

($\Leftarrow$) Soit $\sigma \in \Sigma$ tel que $A(\sigma) \in \Sigma'$ et $I(A(\sigma)) = \sigma$. D'après la question précédente, on a $I(A(\sigma)) \in \Sigma_w$ et donc $\sigma \in \Sigma_w$.

Question 16 –

```
(* Suppose que u est un indice valide pour tabA et que tabA.(u) est un indice
   valide pour tabI *)
let est_present (theta: creux) (u: unicode) =
  let v, _, tabI, tabA = theta in
  tabA.(u) < v && tabI.(tabA.(u)) = u;;
```

Pour la terminaison, on remarque que toutes les opérations utilisées sont élémentaires et terminent. Donc l'appel à `est_present` termine pour toute entrée.

Pour la correction, on note (v, F, I, A) le tableau creux de comptage donné en entrée de la fonction et w le mot associé. On remarque que `est_present` renvoie `true` si et seulement si $A(\sigma_u) \leqslant \sigma_{v-1}$ et $I(A(\sigma_u)) = \sigma_u$. D'après la question précédente, cette propriété est équivalente à dire que σ_u est présente dans w.

Question 17 –

```
(* Attention, cette fonction modifie les tableaux donnés en entrée *)
let incremente_tableaucreux (theta: creux) (u: unicode): creux =
  let v, tabF, tabI, tabA = theta in
  if est_present theta u then begin
      tabF.(u) <- tabF.(u) + 1;
      (v, tabF, tabI, tabA)
    end else begin
      tabF.(u) <- 1;
      tabI.(v) <- u;
      tabA.(u) <- v;
      (v+1, tabF, tabI, tabA);
    end;;
```

Question 18 –

```
let rec cree_tableaucreux (t: texte): creux = match t with
  | [] -> make_creux lambda (* lambda = variable globale de l'énoncé *)
  | u :: q -> incremente_tableaucreux (cree_tableaucreux q) u;;
```

Remarque. Extrait du rapport du jury : "il n'est pas pertinent d'implanter des fonctions auxiliaires récursives terminales".

Question 19 – On remarque que les fonctions `est_present` et `incremente_tableaucreux` s'exécutent en temps constant. Par hypothèse de l'énoncé, la fonction `make_creux` s'exécute aussi en temps constant. Ainsi, dans la fonction `cree_tableaucreux`, on a $\mathcal{O}(|t|)$ appels récursifs et toutes les opérations s'effectuent en temps constant. En conclusion :

La complexité en temps de `cree_tableaucreux` est $\mathcal{O}(|t|)$

Question 20 – La complexité en espace de `cree_tableaucreux` est $\mathcal{O}(\lambda)$

Remarque. Le rapport du jury indique que les compilateurs modernes savent gérer les fonctions récursives non terminales. Mon interprétation de cette remarque est qu'il ne faut pas compter la taille de la pile d'exécution dans la complexité en espace.

Question 21 – La complexité temporelle semble raisonnable. En effet, si on estime que $|t| \approx 1000$, sachant qu'un ordinateur actuel effectue plusieurs milliards d'opérations par seconde, le fonction sera exécutée instantanément. De plus, il semble difficile d'obtenir une meilleure complexité temporelle, en effet l'opération consistant simplement à lire chaque lettre du courriel est déjà en temps linéaire en $|t|$.

Pour la complexité en espace, on a $\lambda \approx 10^6$. Ainsi, la quantité de mémoire pour stocker le courriel sera de l'ordre de quelques mega-octets. Même si ça ne semble pas optimal (espace élevé pour stocker un simple texte), cela semble réaliste avec un ordinateur ou un téléphone actuel.

Utiliser cette fonction semble réaliste

Question 22 –

```
(* Renvoie le codage du caractère u *)
let rec encodeur_unicode (u: unicode) (c: code): bool list = match c with
  | Nil -> failwith "Le caractère n'apparait pas dans l'arbre de code"
  | Noeud (v, b, _, _)  when u = v -> b
  | Noeud (v, _, cg, _) when u < v -> encodeur_unicode u cg
  | Noeud (v, _, _, cd) (* when u > v *) -> encodeur_unicode u cd;;
```

```
let rec encodeur (t: texte) (c: code): bool list = match t with
  | [] -> []
  | u :: s -> list_append (encodeur_unicode u c) (encodeur s c);;
```

Question 23 – On s'intéresse d'abord à la complexité d'évaluation de « `encodeur_unicode u c` ». Soit σ la lettre associée à `u` et $\mathcal{A}$ l'arbre de code associé à `c`. La fonction parcourt l'unique chemin entre le sommet associé à `u` et la racine de $\mathcal{A}$. Ainsi la complexité est $\mathcal{O}(\text{prof}_{\mathcal{A}}(\sigma))$.

Pour la fonction `encodeur`, on note w le mot associé à `t`. Il y a un appel à `encodeur_unicode` pour chaque caractère `u` de `t`, soit une complexité en :

$$\mathcal{O}\left(\sum_{\sigma\in\Sigma} |w|_\sigma \text{prof}_{\mathcal{A}}(\sigma)\right)$$

De plus, la fonction `list_append` est appelée $|w|$ fois, une fois pour chaque lettre σ de w avec pour premier argument une liste de taille $|c(\sigma)| \leqslant L$. Ainsi, le temps d'exécution total est :

$$\boxed{\mathcal{O}\left(|w|L + \sum_{\sigma\in\Sigma} |w|_\sigma \text{prof}_{\mathcal{A}}(\sigma)\right)}$$

Question 24 – Soit $\rho \in \Sigma$ la racine de $\mathcal{A}$, Σ_g l'ensemble des lettres présentes dans $\mathcal{A}_g$ et Σ_d l'ensemble des lettres présentes dans $\mathcal{A}_d$. Alors :

$$\begin{aligned}
\text{Prof}(\mathcal{A}) &= \sum_{\sigma\in\Sigma} f(\sigma)\text{prof}_{\mathcal{A}}(\sigma) \\
&= f(\rho)\text{prof}_{\mathcal{A}}(\rho) + \sum_{\sigma\in\Sigma_g} f(\sigma)\text{prof}_{\mathcal{A}}(\sigma) + \sum_{\sigma\in\Sigma_d} f(\sigma)\text{prof}_{\mathcal{A}}(\sigma) \\
&= f(\rho) + \sum_{\sigma\in\Sigma_g} f(\sigma)(1+\text{prof}_{\mathcal{A}_g}(\sigma)) + \sum_{\sigma\in\Sigma_d} f(\sigma)(1+\text{prof}_{\mathcal{A}_d}(\sigma)) \\
&= \sum_{\sigma\in\Sigma} f(\sigma) + \sum_{\sigma\in\Sigma_g} f(\sigma)\text{prof}_{\mathcal{A}_g}(\sigma) + \sum_{\sigma\in\Sigma_d} f(\sigma)\text{prof}_{\mathcal{A}_d}(\sigma)
\end{aligned}$$

$$\boxed{\text{Prof}(\mathcal{A}) = \sum_{\sigma\in\Sigma} f(\sigma) + \text{Prof}(\mathcal{A}_g) + \text{Prof}(\mathcal{A}_d)}$$

Remarque. Attention, pour que cette formule soit valide, il faut que Σ soit l'ensemble des lettres qui apparaissent dans $\mathcal{A}$ (ce n'est pas nécessairement tout l'alphabet unicode).

Question 25 –

⋆ Soit u un entier compris entre 0 et $\lambda - 1$. Le seul arbre de code $\mathcal{A}_u$ associé à $c_{u,u}$ est un arbre réduit à la feuille σ_u. Ainsi, $\Pi_{u,u} = \text{Prof}(\mathcal{A}_u)$:

$$\text{Prof}(\mathcal{A}_u) = f(\sigma_u)\text{prof}_{\mathcal{A}_u}(\sigma_u) = f(\sigma_u)$$

Donc $\boxed{\Pi_{u,u} = f(\sigma_u)}$

⋆ Soient u et v deux entiers compris entre 0 et $\lambda - 1$ avec $u < v$. On a :

$$\boxed{\Pi_{u,v} = \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \min_{r \in [\![u,v]\!]} \Big[\Pi_{u,r-1} + \Pi_{r+1,v}\Big]}$$

Dans cette formule, pour $r = u$, on a posé $\Pi_{u,r-1} = 0$ et pour $r = v$, on a posé $\Pi_{v+1,v} = 0$. Montrons cette formule par double inégalité :

($\geqslant$) Soit $\mathcal{A}$ un arbre de code optimal représentant le code $c_{u,v}$, soit $t \in [\![u, v]\!]$ tel que la racine de $\mathcal{A}$ est σ_t, soit $\mathcal{A}_g$ le sous-arbre gauche de $\mathcal{A}$ et $\mathcal{A}_d$ son sous-arbre droit. Par définition d'un arbre de code, l'ensemble des noeuds de $\mathcal{A}_g$ est $\Sigma_{u,t-1}$ et l'ensemble des noeuds de $\mathcal{A}_d$ est $\Sigma_{t+1,v}$. Avec la formule obtenue à la question 24, on a :

$$\begin{aligned}
\Pi_{u,v} &= \text{Prof}(\mathcal{A}) \\
&= \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \text{Prof}(\mathcal{A}_g) + \text{Prof}(\mathcal{A}_d) \\
&\geqslant \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \Pi_{u,t-1} + \Pi_{t+1,v} \\
&\geqslant \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \min_{r \in [\![u,v]\!]} \Big[\Pi_{u,r-1} + \Pi_{r+1,v}\Big]
\end{aligned}$$

($\leqslant$) Soit $t \in [\![u, v]\!]$ l'entier qui minimise la quantité $\Pi_{u,t-1} + \Pi_{t+1,v}$. Soit $\mathcal{A}_g$ (resp. $\mathcal{A}_d$) un arbre de code optimal pour $c_{u,t-1}$ (resp. $c_{t+1,v}$). Soit $\mathcal{A}$ l'arbre dont la racine est σ_t étiquetée par $f(\sigma_t)$ et dont les deux sous-arbres sont $\mathcal{A}_g$ et $\mathcal{A}_d$. Alors $\mathcal{A}$ est un arbre de code pour $c_{u,v}$. Avec la formule établie à la question 24 :

$$\begin{aligned}
\Pi_{u,v} &\leqslant \text{Prof}(\mathcal{A}) \\
&= \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \text{Prof}(\mathcal{A}_g) + \text{Prof}(\mathcal{A}_d) \\
&= \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \Pi_{u,t-1} + \Pi_{t+1,v} \\
&= \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \min_{r \in [\![u,v]\!]} \Big[\Pi_{u,r-1} + \Pi_{r+1,v}\Big]
\end{aligned}$$

Question 26 – On va construire trois tableaux de tableaux S, P, A où pour tout $(u, v) \in [\![0, \lambda - 1]\!]^2$:

$$\begin{cases}
S.(u,v) = \displaystyle\sum_{t=u}^{v} f(\sigma_t) \\
P.(u,v) = \Pi_{u,v} \\
A.(u,v) \text{ est un arbre de code optimal pour } c_{u,v}.
\end{cases}$$

Commençons par construire S :

$S \leftarrow$ une matrice de taille $\lambda \times \lambda$ ne contenant que des 0.
pour u variant de 0 à $\lambda - 1$ **faire**
  $S.(u,u) \leftarrow f(\sigma_u)$
  pour v variant de $u+1$ à $\lambda - 1$ **faire**
    $S.(u,v) \leftarrow S.(u,v-1) + f(\sigma_v)$
  fin pour
fin pour
Renvoyer S

Pour les matrices P et A, on complète les cases de coordonnées (u,v) avec $u > v$, puis celles avec $u = v$, puis $u = v-1$, puis $u = v-2$, ..., puis $u = v-\lambda+1$. On utilise les formules établies à la question précédente.

Dans l'algorithme qui suit, lorsqu'on essaye d'accéder à $P.(u,v)$ ou $A.(u,v)$ avec $u < 0$ ou $v > \lambda - 1$, on considérera que $P.(u,v) = 0$ et $A.(u,v)$ est l'arbre vide. Dans l'implémentation, on pourra écrire une fonction intermédiaire qui prend en entrée u, v et P (resp. A) et qui renvoie $P.(u,v)$ ou 0 (resp. $A.(u,v)$ ou l'arbre vide).

$P \leftarrow$ une matrice de taille $\lambda \times \lambda$ ne contenant que des 0.
$A \leftarrow$ une matrice de taille $\lambda \times \lambda$ ne contenant que des arbres vides.
pour u variant de 0 à $\lambda - 1$ **faire**
  $P.(u,u) \leftarrow f(\sigma_u)$
  $A.(u,u) \leftarrow$ l'arbre réduit à la feuille σ_u.
fin pour
pour k variant de 1 à $\lambda - 1$ **faire**
  pour u variant de 0 à $\lambda - 1 - k$ **faire**
    Soit $v = u + k$.
    À l'aide d'une boucle for, on calcule l'entier $r \in [\![u, v]\!]$ qui minimise $P.(u,r-1) + P.(r+1,v)$.
    $P.(u,v) \leftarrow S.(u,v) + P.(u,r-1) + P.(r+1,v)$
    $A.(u,v) \leftarrow$ l'arbre dont $\begin{cases} \text{la racine est } \sigma_r \text{ et est étiquetée par } c(\sigma_u), \\ \text{le sous-arbre gauche est } A.(u,r-1), \\ \text{le sous-arbre droit est } A.(r+1,v). \end{cases}$
  fin pour
fin pour

Pour la complexité, on remarque que la construction de S se fait en temps $\mathcal{O}(\lambda^2)$ puisqu'il y a deux boucles imbriquées avec au plus λ tours de boucle. Pour la construction de P et A, on a trois boucles imbriquées avec au plus λ tours de boucle, soit une complexité en $\mathcal{O}(\lambda^3)$.

Question 27 – En plus des matrices S, P et A évoquées dans la question précédente, on construit une matrice R de taille $\lambda \times \lambda$ telle que pour tout $(u,v) \in [\![0, \lambda-1]\!]^2$, $R.(u,v) = r_{u,v}$. L'intérêt est qu'on va pouvoir diminuer le nombre de tours de la boucle `for` qui calcule le minimum des $P.(u,r-1) + P.(r+1,v)$. Dans la première boucle, on ajoute :

$R.(u,u) \leftarrow u$

Dans les boucles imbriquées, il suffit de faire varier r de $R.(u,v-1)$ à $R.(u+1,v)$ (l'algorithme a déjà calculé ces valeurs dans les tours précédents). Ainsi, on remplace la ligne « À l'aide d'une boucle for ... » par :

À l'aide d'une boucle for, on calcule le plus grand entier $r \in [\![R.(u,v-1); R.(u+1,v)]\!]$ qui minimise $P.(u,r-1) + P.(r+1,v)$.
$R.(u,v) \leftarrow r.$

Montrons que la complexité temporelle est en $\mathcal{O}(\lambda^2)$. Soit N le nombre de tours total des trois boucles

imbriquées, alors :

$$\begin{aligned} N &= \sum_{k=1}^{\lambda-1} \sum_{u=0}^{\lambda-1-k} (1 + r_{u+1,u+k} - r_{u,u+k-1}) \\ &= \mathcal{O}(\lambda^2) + \sum_{k=1}^{\lambda-1} \sum_{u=0}^{\lambda-1-k} r_{u+1,u+k} - \sum_{k=1}^{\lambda-1} \sum_{u=0}^{\lambda-1-k} r_{u,u+k-1} \\ &= \mathcal{O}(\lambda^2) + \sum_{k=1}^{\lambda-1} r_{\lambda-k,\lambda-1} - \sum_{k=1}^{\lambda-1} r_{0,k-1} \end{aligned}$$

Or, pour tout (u, v), $r_{u,v} \leqslant \lambda - 1$ et donc le nombre de tours de boucle est bien en $\mathcal{O}(\lambda^2)$.

Le temps d'exécution de ce nouvel algorithme est $\mathcal{O}(\lambda^2)$

Question 28 – Par définition d'un arbre de code, pour tout τ dans le sous-arbre droit de σ_u, on a $\tau > \sigma_u$. Si un arbre de code représente $c_{u,v}$, alors aucun noeud τ ne vérifie $\tau > \sigma_u$. D'où le résultat.

Question 29 – Soient $\mathcal{A}$ et $\mathcal{A}'$ les arbres décrits dans l'énoncé. Pour tout $\sigma \in \Sigma_{u,v}$, $\text{prof}_{\mathcal{A}}(\sigma) = \text{prof}_{\mathcal{A}'}(\sigma)$. En appliquant la définition de la profondeur pondérée, on obtient $\text{Prof}(\mathcal{A}) = \text{Prof}(\mathcal{A}')$.

Montrons que $\Pi_{u,v+1} \geqslant \Pi_{u,v}$. Soit $\mathcal{B}$ un arbre de code optimal pour $c_{u,v+1}$. Par la question précédente, la sommet σ_{v+1} n'a pas de fils droit. On note $\mathcal{B}'$ l'arbre $\mathcal{B}$ dans lequel on a remplacé σ_{v+1} par son sous-arbre gauche (éventuellement vide). Alors $\mathcal{B}'$ est un arbre de code pour $c_{u,v}$. De plus, pour tout $\sigma \in \Sigma_{u,v}$, $\text{prof}_{\mathcal{B}'}(\sigma) \leqslant \text{prof}_{\mathcal{B}}(\sigma)$ et donc $\text{Prof}(\mathcal{B}') \leqslant \text{Prof}(\mathcal{B})$. Finalement :

$$\Pi_{u,v+1} = \text{Prof}(\mathcal{B}) \geqslant \text{Prof}(\mathcal{B}') \geqslant \Pi_{u,v}.$$

Ainsi :

$$\text{Prof}(\mathcal{A}') = \text{Prof}(\mathcal{A}) = \Pi_{u,v} \leqslant \Pi_{u,v+1}$$

Étant donné que $\mathcal{A}'$ est un arbre de code pour $c_{u,v+1}$, on en déduit que $\text{Prof}(\mathcal{A}') = \Pi_{u,v+1}$, c'est à dire que $\mathcal{A}'$ est optimal.

Question 30 – On note $\mathbb{A}$ l'ensemble de tous les arbres de codes représentant $c_{u,v+1}$. Alors $\mathbb{A}$ est fini. On fixe des réels $(f(\sigma_x))_{u \leqslant x \leqslant v}$ et pour tout $\mathcal{A} \in \mathbb{A}$, note $P_{\mathcal{A}}$ la fonction :

$$P_{\mathcal{A}} : \begin{cases} \mathbb{R}_+ & \to \quad \mathbb{R}_+ \\ x & \mapsto \quad x \times \text{prof}_{\mathcal{A}}(\sigma_{v+1}) + \displaystyle\sum_{\sigma \in \Sigma_{u,v}} f(\sigma)\text{prof}_{\mathcal{A}}(\sigma) \end{cases}$$

Alors, pour tout réel $f(\sigma_{v+1})$, on a $P_{\mathcal{A}}(f(\sigma_{v+1})) = \text{Prof}(\mathcal{A})$. De plus, la fonction $P_{\mathcal{A}}$ est affine de pente $\text{prof}_{\mathcal{A}}(\sigma_{v+1}) \in [\![1; v - u + 2]\!]$. L'application $\pi_{u,v+1}$ vérifie :

$$\forall x \in \mathbb{R}_+ : \pi_{u,v+1} = \min_{\mathcal{A} \in \mathbb{A}} P_{\mathcal{A}}(x).$$

$\pi_{u,v+1}$ est donc continue et affine par morceaux sur $\mathbb{R}_+$.

Montrons que la pente de $\pi_{u,v+1}$ diminue lorsque $f(\sigma_{v+1})$ augmente. Notons $I_1, I_2, \ldots, I_r$ les intervalles sur lesquels $\pi_{u,v+1}$ est affine et $\mathcal{A}_1, \ldots, \mathcal{A}_r$ les arbres de code optimaux pour chacun de ces intervalles. Alors $\pi_{u,v+1}$ est égale au minimum des $P_{\mathcal{A}_k}$.

Soit $k < r$, soient a, b les réels tels que $I_k = [\, a\,;b\,]$ et $\alpha, \beta, \gamma, \delta$ tels que :

$$\begin{cases} \forall x \in I_k : & \pi_{u,v+1}(x) = P_{\mathcal{A}_k}(x) = \alpha x + \beta \\ \forall x \in I_{k+1} : & \pi_{u,v+1}(x) = P_{\mathcal{A}_{k+1}}(x) = \gamma x + \delta \end{cases}$$

On suppose par l'absurde que $\alpha < \gamma$. Comme $\pi_{u,v+1}$ est continue en b :

$$\alpha b + \beta = \gamma b + \delta \qquad \text{donc} \qquad \beta - \delta = b(\gamma - \alpha)$$

En $x = a$, on obtient :

$$P_{\mathcal{A}_k}(a) - P_{\mathcal{A}_{k+1}}(a) = (\alpha a + \beta) - (\gamma a + \delta) = a(\alpha - \gamma) + b(\gamma - \alpha) = (\gamma - \alpha)(b - a) > 0$$

Ce qui contredit le fait que $\pi_{u,v+1}(a)$ est le minimum des $P_{\mathcal{A}_k}(a)$. En résumé :

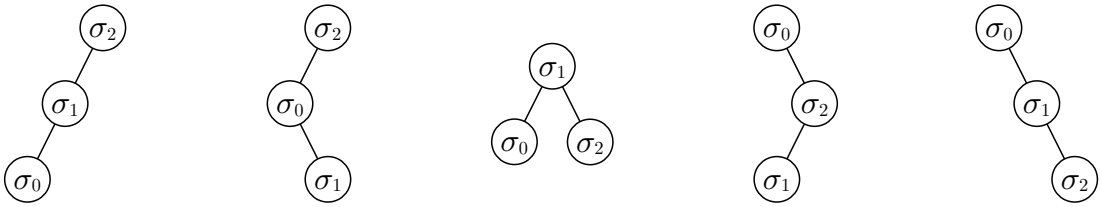
Sur chaque morceau I_k, la pente est un entier $\alpha_k \in [\![1; v-u+2]\!]$ avec $\alpha_1 > \alpha_2 > \ldots > \alpha_r$.
De plus, pour tout $k, \alpha_k = \text{prof}_{\mathcal{A}_k}(\sigma_{v+1})$ où $\mathcal{A}_k$ est l'arbre de code optimale lorsque $f(\sigma_{v+1}) \in I_k$.

Pour tout $u \leqslant u' \leqslant v' < v+1$, comme $r_{u',v'}$ est défini indépendamment de $f(\sigma_{v+1})$:

Les indices $(r_{u',v'})_{u \leqslant u' \leqslant v' < v+1}$ sont constants lorsque $f(\sigma_{v+1})$ varie.

Lorsque $v' = v+1$, le résultat proposé par l'énoncé me semble faux.

Par exemple, avec $u = 0$, $v = 1$, $f(\sigma_0) = f(\sigma_1) = 1$ et $f(\sigma_2) = x \in \mathbb{R}_+$, l'ensemble $\mathbb{A}$ contient cinq arbres :



Les profondeurs pondérées associées sont :

$$P_1(x) = x+5 \qquad P_2(x) = x+5 \qquad P_3(x) = 2x+3 \qquad P_4(x) = 2x+4 \qquad P_5(x) = 3x+3$$

Pour tout $x \in \mathbb{R}_+$:

$$P_1(x) = P_2(x), \qquad P_3(x) \leqslant P_4(x), \qquad P_3(x) \leqslant P_5(x).$$

On obtient :

$$\begin{cases} \pi_{u,v+1}(x) = 2x+3 & \text{si } x \leqslant 2 \\ \pi_{u,v+1}(x) = x+5 & \text{si } x \geqslant 2 \end{cases}$$

Si on s'intéresse à $r_{1,2}$, on ne considère que les lettres σ_1 et σ_2, il y a deux arbres de codes possibles :

Les profondeurs pondérées associées sont :

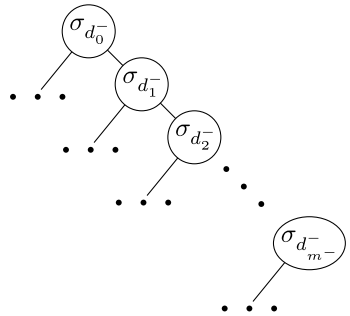
$$Q_1(x) = x+2 \qquad Q_2(x) = 2x+1$$

Pour $x \in]0\,;1[: Q_1(x) > Q_2(x)$, c'est à dire $r_{1,2} = \sigma_1$.
Pour $x \in]1\,;2[: Q_1(x) < Q_2(x)$, c'est à dire $r_{1,2} = \sigma_2$.
Donc $r_{1,2}$ n'est pas constant sur l'intervalle ouvert $]0\,;2[$ sur lequel $\pi_{u,v+1}$ est affine.

Question 31 – Comme $r_{u,v} \in [\![u;v]\!]$, les suites (d_k^+) et (d_k^-) sont strictement croissantes, prennent des valeurs entières positives et s'arrêtent lorsqu'elles atteignent $v+1$.

$$\boxed{\text{Les deux suites sont donc finies}}$$

On fixe $f(\sigma_{v+1})$ dans l'intervalle I^-. Montrons qu'il existe un arbre de code optimal pour $c_{u,v+1}$ noté $\mathcal{A}^-$ dont la racine est $\sigma_{d_0^-}$ et dans lequel, pour tout k tel que d_{k+1}^- est défini, le fils droit de $\sigma_{d_k^-}$ est $\sigma_{d_{k+1}^-}$:



Pour cela, on note $\mathcal{A}_0$ un arbre de code optimal pour $c_{u,v+1}$ dont la racine est $\sigma_{d_0^-}$. De même, pour chaque k tel que d_{k+1}^- est défini, on note $\mathcal{A}_{k+1}$ un arbre de code optimal pour $c_{d_k^++1,v+1}$ dont la racine est $\sigma_{d_{k+1}^-}$. Maintenant que $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_{m^--1}, \mathcal{A}_{m^-}$ sont définis, on définit $\mathcal{B}_{m^-}, \mathcal{B}_{m^--1}, \ldots, \mathcal{B}_1, \mathcal{B}_0$ de la manière suivante :

$$\begin{cases} \mathcal{B}_{m^-} = \mathcal{A}_{m^-} \\ \text{Pour tout } k < m^-,\ \mathcal{B}_k \text{ est l'arbre } \mathcal{A}_k \text{ dans lequel le sous-arbre droit est remplacé par } \mathcal{B}_{k+1}. \end{cases}$$

Avec cette définition, les $\mathcal{B}_k$ sont des arbres de code optimaux et l'arbre $\mathcal{A}^- = \mathcal{B}_0$ est de la forme évoquée ci-dessus. Le point important est de noter que $\sigma_{d_{m^-}^-}$ est à profondeur m^-. Par définition de la profondeur pondérée, m^- est donc la pente de la fonction $\pi_{u,v+1}$ sur l'intervalle I^-.

Avec le même raisonnement, on montre que m^+ est la pente de la fonction $\pi_{u,v+1}$ sur l'intervalle I^+. Comme la pente de $\pi_{u,v+1}$ diminue entre I^- et I^+, d'après le résultat de la question précédente :

$$\boxed{m^- > m^+}$$

Question 32 – On suppose $\mathcal{E}(n)$ pour un certain $n \in [\![0, \lambda-3]\!]$ et que les entiers u, v vérifient $v-u \leqslant n+1$. En utilisant plusieurs fois l'inégalité de droite de $\mathcal{E}(n)$, on remarque que :

$$r_{u+1,v+1} \leqslant r_{u+2,v+1} \leqslant r_{u+3,v+1} \leqslant \ldots \leqslant r_{v,v+1} \leqslant r_{v+1,v+1}.$$

Supposons que $d_0^+ < d_0^-$. Comme $m^- > m^+$ (voir la question précédente), la suite (d_k^-) ne s'est pas encore arrêtée à l'indice $k = m^+$, c'est à dire que :

$$d_{m^+}^+ = v+1 > d_{m^+}^-$$

On peut donc définir $s \leqslant m^+$ le plus petit entier tel qu'on ait $d_s^+ \geqslant d_s^-$ et pour tout ℓ compris entre 0 et $s-1$, on ait $d_\ell^+ < d_\ell^-$. Si on s'intéresse à d_{s-1}^+ et d_{s-1}^-, on a $d_{s-1}^+ + 1 \geqslant u+1$ et donc :

$$r_{d_{s-1}^++1,v+1} \leqslant r_{d_{s-1}^++2,v+1} \leqslant r_{d_{s-1}^++3,v+1} \leqslant \ldots \leqslant r_{v,v+1} \leqslant r_{v+1,v+1}.$$

Comme $d_{s-1}^+ < d_{s-1}^- \leqslant v+1$, on obtient :

$$r_{d_{s-1}^++1,v+1} \leqslant r_{d_{s-1}^-+1,v+1} \qquad \text{c'est à dire} \qquad d_s^+ \leqslant d_s^-$$

Comme $d_{m^+}^+ > d_{m^+}^-$, on a nécessairement $s < m^+$. Donc :

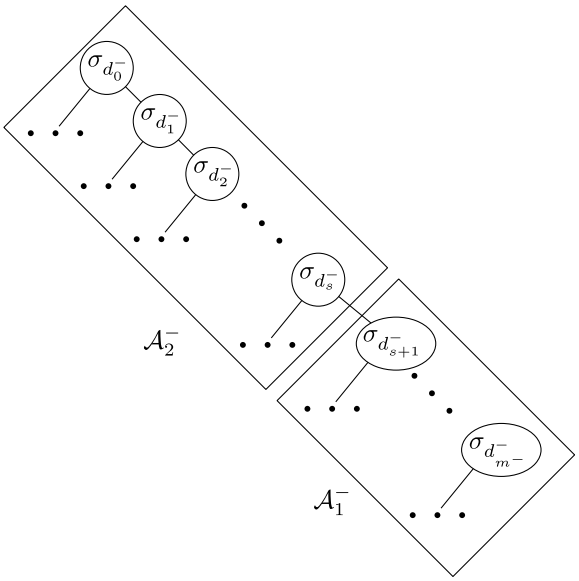
$$\boxed{d_s^+ = d_s^- \text{ et } \forall \ell \in [\![0; s-1]\!] : d_\ell^+ < d_\ell^-}$$

On note $\mathcal{A}_-$ l'arbre de code optimal pour $c_{u,v+1}$ définit dans la question précédente et $\mathcal{A}_+$ l'arbre de code optimal pour $c_{u,v+1}$ obtenu par la même procédure en utilisant (d_k^+) à la place de (d_k^-).

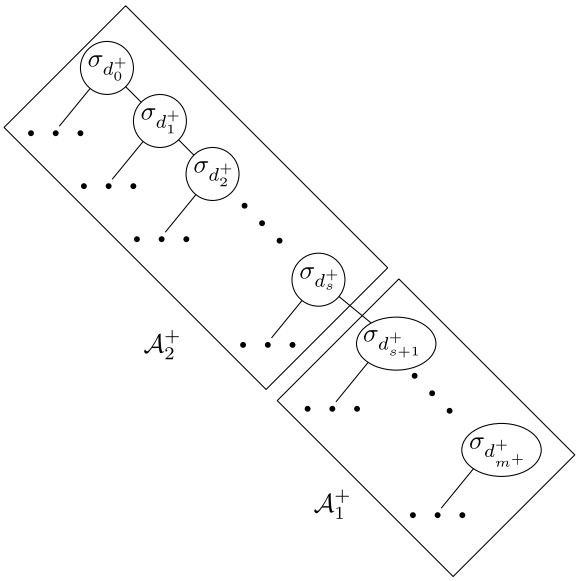
Soit $\mathcal{A}_1^-$ le sous-arbre de $\mathcal{A}^-$ enraciné en $\sigma_{d_{s+1}^-}$ et $\mathcal{A}_2^-$ l'arbre $\mathcal{A}^-$ privé de $\mathcal{A}_1^-$.

Soit $\mathcal{A}_1^+$ le sous-arbre de $\mathcal{A}^+$ enraciné en $\sigma_{d_{s+1}^+}$ et $\mathcal{A}_2^+$ l'arbre $\mathcal{A}^+$ privé de $\mathcal{A}_1^+$.

$\mathcal{A}^-$:



$\mathcal{A}^+$:



Les arbres $\mathcal{A}_2^-$ et $\mathcal{A}_2^+$ sont tous les deux des arbres de codes optimaux pour c_{u,d_s^+} (notez que $d_s^+ = d_s^-$). Leurs profondeurs pondérées, qui ne dépendent pas de la variable $f(\sigma_{v+1})$, sont donc égales.

On fixe maintenant $f(\sigma_{v+1})$ dans I^+. Si on note $\mathcal{B}$ l'arbre $\mathcal{A}^+$ dans lequel on a remplacé $\mathcal{A}_2^+$ par $\mathcal{A}_2^-$, alors $\mathcal{B}$ est un arbre de code ayant la même profondeur pondérée que $\mathcal{A}^+$. C'est à dire que $\mathcal{B}$ est un arbre de code optimal pour $c_{u,v+1}$. Pourtant la racine de $\mathcal{B}$ est $\sigma_{d_0^-}$ et $d_0^- > d_0^+ = r_{u,v+1}$. Cette inégalité contredit la définition de $r_{u,v+1}$.

On a une contradiction, c'est à dire que $d_0^+ \geqslant d_0^-$

Question 33 – On le montre par récurrence sur n.

⋆ Initialisation. Pour $n = 0$, on doit montrer que si $u = v$, alors $r_{u,v} \leqslant r_{u,v+1} \leqslant r_{u+1,v+1}$. On a nécessairement $r_{u,u} = u$, $r_{u+1,u+1} = u + 1$ et $r_{u,u+1} \in \{u, u + 1\}$. D'où le résultat.

⋆ Hérédité. Soit $n \leqslant \lambda - 3$. On suppose $\mathcal{E}(n)$ et on montre $\mathcal{E}(n + 1)$. Soient u et v deux entiers tels que $v - u \leqslant n+1$. Soit $R : \mathbb{R}_+ \to [\![0, \lambda - 1]\!]$ la fonction qui à un poids $f(\sigma_{v+1})$ associe $r_{u,v+1}$. D'après la question 32, la fonction R est croissante.

On s'intéresse à $R(0)$, c'est à dire au cas où $f(\sigma_{v+1}) = 0$. En appliquant la question 29 avec $\mathcal{A}$ un arbre de code optimal pour $c_{u,v}$ dont la racine est $\sigma_{r_{u,v}}$, on obtient un arbre de code optimal pour $c_{u,v+1}$ dont la racine est $\sigma_{r_{u,v}}$. Ainsi :

$$R(0) \geqslant r_{u,v}.$$

Si $f(\sigma_{v+1})$ a une valeur quelconque, on a $r_{u,v+1} = R(f(\sigma_{v+1})) \geqslant R(0) \geqslant r_{u,v}$ qui est la première inégalité de $\mathcal{E}(n + 1)$.

Dans la preuve de la première inégalité (qui a commencé à la question 28), on a étudié ce qu'il se passe lorsqu'on passe de l'alphabet $\Sigma_{u,v}$ à $\Sigma_{u,v+1}$. Pour la deuxième inégalité, on utilise un raisonnement analogue en passant de $\Sigma_{u+1,v+1}$ à $\Sigma_{u,v+1}$.