La dernière partie du problème semble inspirée d'un article de D. Knuth, consultable en ligne à l'adresse suivante :

`http://www.cs.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/Knuth71.pdf`

1 Fonction de distribution des lettres d'un texte

1.1 Avec un tableau exhaustif

1. On impose à `t` le type `texte`, par ailleurs le type de `array_make lambda 0` est `int array`, on a donc :

   `fonction1 : texte -> int array`

   Montrons par récurrence sur $|t|$ que `fonction1 t` renvoie un tableau `tab` de λ nombres entiers tel pour tout `u` $\in [\![0,\lambda-1]\!]$, `tab.(u)` donne le nombre d'occurrences de `u` dans le texte `t`.

   - Le cas de base du filtrage montre que c'est vrai pour le texte vide, représenté par la liste vide.
   - On considère $n \in \mathbb{N}$, on suppose que le résultat est vrai sur tout texte de longueur n. On considère `t` un texte de longueur $n+1$, il est de la forme `u,t'`, où `u` est un caractère unicode et `t'` un texte de longueur n.

     En utilisant les notations introduites pour les mots, sur les textes, on a :

     $$|t|_u = |t'|_u + 1$$

     et pour tout caractère unicode `v` différent de `u` :

     $$|t|_v = |t'|_v$$

     ce qui correspond au cas inductif du filtrage dans le code de la fonction `fonction1`.

   Conclusion : Le résultat est vrai pour tout $n \in \mathbb{N}$.
2. On propose deux versions du code, avec et sans référence. Le principe consiste simplement à repérer dans `fonction1 t` les valeurs non-nulles du tableau.

```
(* avec ref *)
let cree_repartition (t:texte) : repartition  =
  let theta = fonction1 t in
  let l = ref [] in
   for i = 0 to lambda-1 do
    if theta.(i) > 0 then
    l := (i,theta.(i))::(!l)
   done;
  !l ;
```

```
(* sans ref *)
let cree_repartition (t:texte) : repartition =
  let tab = fonction1 t in
  let rec aux resultat k =
   match k with
    | -1               -> res
    | k when tab.(k)=0 -> aux resusltat (k-1)
    | k                -> aux ((k,tab.(k))::ressultat) (k-1)
  in aux [] (lambda-1);;
```

3. La complexité en temps de la fonction `cree_repartition` est celle de la fonction `fonction1` qui est linéaire sur le longueur du texte passé en argument ($\mathcal{O}(|t|)$).
On parcourt ensuite tous les éléments du tableau pour réaliser à chaque fois des opérations à coût constant. La complexité cherchée est donc :
$$\mathcal{O}(\lambda + |t|)$$
4. La complexité en espace de la fonction `cree_repartition` tient à la création du tableau de répartition par la fonction `fonction1`, cette complexité est donc $\mathcal{O}(\lambda)$.
5. La taille de la liste de retour de `cree_repartition t` est égale au nombre de caractères distincts du texte t. En étendant la notion de valence (définie formellement pour les mots) aux textes, `cree_repartition t` renvoie une liste de $[t]$ couples de deux nombres entiers naturels. La taille de la valeur de retour, au sens de son occupation mémoire, est donc de l'ordre de $\mathcal{O}([t])$.
6. En comptant les ponctuations, les caractères accentués et les caractères spéciaux, un courriel rédigé en langue française utilise une centaine de caractères distincts, pour un texte dont la longueur est de l'ordre de 1000. La taille de l'alphabet Unicode est donc environ 1000 fois plus grande que la taille du texte.
Dans les questions 3 et 4 les complexités sont donc du même ordre de grandeur $\mathcal{O}(\lambda)$, alors que la taille de la liste de retour (question 5) est très réduite par rapport à λ...Les complexités spatiale et temporelle de la fonction `cree_repartition` n'apparaissent donc pas optimales au regard de la taille de la valeur de retour.

1.2 Avec une table modulaire

7. On propose le code suivant dans lequel le filtrage dans la fonction auxiliaire test si le caractère unicode `u` est dans la répartition passée en argument :

```
let rec incremente_repartition (r:repartition) (u:unicode) :repartition =
  match r with
  |[]                          -> [(u,1)]
  |(v,n)::rprime when v = u -> (v,n+1)::rprime
  |(v,n)::rprime             -> (v,n)::(incremente_repartition rprime u) ;;
```

8. Il s'agit de lire le texte passé en argument et d'incrémenter de 1, la bonne liste de répartition, c'est le rôle de la fonction auxiliaire dans le code suivant :

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

9. Calculer la valence à partir de la table de comptage modulaire revient à additionner la longueur des listes composant la table modulaire de comptage. On propose ici encore deux options pour le code.

```
(* avec ref *)
let valence (theta:repartition array) =
  let m = array_length theta and v = ref 0 in
   for i=0 to m-1 do
    v := !v + List.length theta.(i)
  done;
 !v ;;
```

```
(* sans ref *)
let rec valence (theta:repartition array) =
  let rec aux res n = match n with
   | 0 -> res
   | k -> aux (res + list_length r.(k-1)) (k-1)
  in aux 0 (array_length r);;
```

10. *La notion d'espérance conditionnelle est hors-programme...*

La variable aléatoire Z_ℓ compte le nombre de variables aléatoires X_i égalent à ℓ. Pour ℓ_0 fixé dans dans $[\![1,m-1]\!]$ on a :

- Pour $\ell = \ell_o$, l'événement $(Z_\ell = k) \cap (X_{i_0} = \ell_0)$ est égal à l'événement :

$$(|\{i \in [\![1,v]\!] \setminus \{i_0\}\,, \quad X_i = \ell\}| = k-1)$$

La loi conditionnelle cherchée suit donc une loi binomiale de paramètre $(v-1,p)$. On a pour tout $k \in [\![1,v]\!]$, en posant $p \overset{\text{def}}{=} \dfrac{1}{m}$ et $q \overset{\text{def}}{=} 1-p$:

$$\mathrm{P}\left(Z_\ell = k \mid X_{i_0} = \ell_0\right) = \binom{v-1}{k-1} p^{k-1} q^{v-k}$$

On a donc :

$$\mathrm{E}\left[Z_\ell \mid X_{i_0}\right] = \sum_{k=1}^{v} k \binom{v-1}{k-1} p^{k-1} q^{v-k} = \underbrace{\sum_{k=1}^{v} (k-1) \binom{v-1}{k-1} p^{k-1} q^{v-k}}_{\overset{\text{def}}{=}\mu} + \underbrace{\sum_{k=1}^{v} \binom{v-1}{k-1} p^{k-1} q^{v-k}}_{=1}$$

μ est l'expression de l'espérance d'une variable aléatoire suivant une loi binomiale de paramètre $(v-1,p)$, on a donc :

$$\mathrm{E}\left[Z_\ell \mid X_{i_0} = \ell_0\right] = \frac{v-1}{m} + 1$$

- Pour $\ell \neq \ell_0$, le même type de raisonnement conduit à :

$$\mathrm{P}\left(Z_\ell = k \mid X_{i_0} = \ell_0\right) = \binom{v-1}{k} p^{k} q^{v-1-k}$$

On a donc :

$$\mathrm{E}\left[Z_\ell \mid X_{i_0} = \ell_0\right] = \underbrace{\sum_{k=0}^{v-1} k \binom{v-1}{k} p^{k} q^{v-1-k}}_{=\mu}$$

Finalement :

$$\mathrm{E}\left[Z_\ell \mid X_{i_0} = \ell_0\right] = \frac{v-1}{m}$$

11. *La notion de complexité moyenne est hors-programme...*

Le coût de la fonction `incremente_repartition` est linéaire en la taille de la répartition donnée en entrée. Pour un texte t donné, on suppose que les v lettres distinctes de t ont été choisies uniformément, et on les note $\sigma_{n_1}, \cdots, \sigma_{n_v}$.
On définit X_i la variable aléatoire valant $\mathrm{n_i} \bmod \mathrm{m}$.
Les variables aléatoires X_i sont des variables aléatoires à valeurs dans $[\![0,m-1]\!]$ et on fait l'hypothèse, suggérée par l'énoncé et liée à la taille de Σ, qu'elles suivent la loi uniforme.
Alors `creer_modulaire t m`, appelle récursivement `incremente_repartition` sur la liste `theta.(u mod m)`, qui est de longueur moyenne majorée par l'espérance conditionnelle de la question précédente (dans le cas $\ell = \ell_0$, car on connaît la lettre u de tête de liste), c'est-à-dire :

$$\frac{v-1}{m} + 1$$

Les $|t|$ appels à `incremente_repartition` engendre donc la complexité moyenne suivante :

$$\mathcal{O}\left(|t| + \frac{(v-1)\,|t|}{m}\right)$$

à laquelle s'ajoute la complexité de la création du tableau de taille m. On obtient finalement la complexité demandée :

$$\mathcal{O}\left(m + |t| + \frac{(v-1)\,|t|}{m}\right)$$

12. La complexité spatiale est la somme des tailles du tableau theta renvoyé et des tailles des sous-listes le composant, donc $\mathcal{O}(m + [t])$.
13. Le but est d'obtenir des complexités spatiale et temporelle "raisonnables".
En faisant les mêmes hypothèses que dans la question 6, en choisissant m du même ordre de grandeur que le nombre moyen de caractères utilisés dans un courriel (c'est-à-dire de l'ordre de 100), on obtient des complexités spatiale et temporelle de la fonction `cree_modulaire t m` de $\mathcal{O}(m)$.

1.3 Avec un tableau creux

14. On note $\mathcal{V}$ l'ensemble des v symboles distincts qui composent le mot w.
Le point (ii) de la définition assure que $I^{-1}(\mathcal{V}) \subset \{\sigma_0, \cdots, \sigma_{v-1}\}$. Puisque :

$$|\mathcal{V}| = |\{\sigma_0, \cdots, \sigma_{v-1}\}| = v$$

on a nécessairement :

$$I^{-1}(\mathcal{V}) = \{\sigma_0, \cdots, \sigma_{v-1}\}$$

L'application :

$$J : \begin{array}{ccc} \{\sigma_0, \cdots, \sigma_{v-1}\} & \to & \mathcal{V} \\ \tau & \mapsto & I(\tau) \end{array}$$

est donc bijective ce qui permet de conclure que pour tout $\tau \in \Sigma$ tel que $\tau \leqslant \sigma_{v-1}$, on a $I(\tau) \in \mathcal{V}$, ce qui revient à dire que $I(\tau)$ est une lettre présente dans le mot ω.
15. Avec les notations précédentes, on considère $\sigma \in \mathcal{V}$, il existe $i \in [\![0, v-1]\!]$ tel que $I(\sigma_i) = \sigma$.
le point (iii) de la définition assure que :

$$A(\sigma) = A(I(\sigma_i)) = \sigma_i$$

En particulier $A(\sigma) \leqslant \sigma_{v-1}$ et $I(A(\sigma)) = \sigma$.
Réciproquement, si $A(\sigma) \leqslant \sigma_{v-1}$ et $I(A(\sigma)) = \sigma$, le premier point et le résultat de la question précédente assurent que :

$$I(A(\sigma)) \in \mathcal{V}$$

Puisque $I(A(\sigma)) = \sigma$, on a donc $\sigma \in \mathcal{V}$. Ce qui permet de conclure que pour tout $\sigma \in \mathcal{V}$:

$$(\sigma \in \mathcal{V}) \Leftrightarrow (A(\sigma) \leqslant \sigma_{v-1} \quad \wedge \quad I(A(\sigma)) = \sigma)$$

16. On propose le code suivant directement inspiré du résultat de la question 15 :

```
let est_present (theta:creux) (u:unicode) =
 let v,_, tabI, tabA = theta in let s = tabA.(u) in
  s < v && tabI.(s) = u;;
```

La question de la terminaison du code est sans objet, sa correction est une conséquence directe du résultat de la question 15.
17. On propose le code suivant qui distingue les cas où u est ou n'est pas une nouvelle lettre.

```
let incremente_tableaucreux (theta:creux) (u:unicode) :creux =
  let v,tabF,tabI,tabA = theta in
   if est_present theta u then  (* sigma_u est déjà présente dans w *)
   (tabF.(u) <- tabF.(u) + 1;
```

```
    (v,tabF,tabI,tabA))
    else                              (* on ajoute une nouvelle lettre *)
    (tabF.(u) <- 1;
     tabI.(v) <- u;
     tabA.(u) <- v;
    (v+1,tabF,tabI,tabA));;
```

18. On suppose que le nombre entier passé en argument de la fonction `makecreux` est la taille de l'alphabet d'encodage. On s'appuie sur la fonction `incremente_tableaucreux` dans le code suivant :

```
let rec cree_tableaucreux (t:texte) :creux =
  match t with
   |[]        -> make_creux lambda
   |u::tprime -> incremente_tableaucreux (cree_tableaucreux tprime) u ;;
```

19. La fonction `cree_tableaucreux` appliquée au texte t fait $|t|$ appels à la fonction `incremente_tableaucreux`. L'énoncé précise que la complexité de `make_creux` est constante. La complexité de `incremente_tableaucreux` est constante (car `est_present` est de coût constant). Ainsi la complexité de `cree_tableaucreux` est :
$$\mathcal{O}(|t|)$$
donc linéaire sur la taille du texte passé en argument.
20. La fonction `cree_tableaucreux` renvoie 3 tableaux de taille λ, sa complexité en espace est :
$$\mathcal{O}(\lambda)$$
21. Pour les valeurs de $|t|$ et de v déjà évoquées, les complexités en temps, comme en espace se ramènent à $\mathcal{O}(\lambda)$, ce qui semble déraisonnable au regard de la valeur de λ.

2 Un encodeur optimal

2.1 Codes et codages

22. On commence par coder une fonction qui détermine le code d'un caractère unicode à partir de l'arbre de code.

```
let rec codeuni (u:unicode) (c:code) =
  match c with
  | Nil                      -> failwith "codeuni"
  | Noeud(s,bi,_,_) when s=u -> bi
  | Noeud(s,bi,cg,_)when s>u -> codeuni u cg
  | Noeud(s,bi,_,cd)         -> codeuni u cd;;
```

La fonction demandée peut s'écrire alors simplement :

```
let rec encodeur (t:texte) (c:code) = match t with
 | []        -> []
 | u::tprime -> let codeprime = encodeur tprime c
                in list_append (codeuni u c) codeprime;;
```

23. La fonction `encodeur` fait $|t|$ appels à la fonction `coduni`. Pour chaque lettre σ du texte passé en argument, la recherche du code de σ coûte $\text{prof}_{\mathcal{A}}(\sigma)$ comparaisons. On compte également les opérations de concaténation, on est conduit à la complexité suivante :

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

2.2 Arbre de code optimal

24. On suppose que la fonction Prof est définie inductivement, en particulier pour tout $\sigma \in \Sigma$ et pour tout arbre de code $\mathcal{A}$, on a :

$$\text{prof}_{\mathcal{A}}(\sigma) = \begin{cases} 1 + \text{prof}_{\mathcal{A}_g}(\sigma) & \text{si } \sigma \text{ appartient à } \mathcal{A}_g \\ 1 + \text{prof}_{\mathcal{A}_d}(\sigma) & \text{si } \sigma \text{ appartient à } \mathcal{A}_d \end{cases}$$

En notant ρ le sommet de $\mathcal{A}$, on a donc :

$$\text{Prof}(\mathcal{A}) = f(\rho) + \sum_{\sigma\in\mathcal{A}_g} f(\sigma)\left(1 + \text{prof}_{\mathcal{A}_g}(\sigma)\right) + \sum_{\sigma\in\mathcal{A}_d} f(\sigma)\left(1 + \text{prof}_{\mathcal{A}_d}(\sigma)\right)$$

Finalement :

$$\text{Prof}(\mathcal{A}) = \sum_{\sigma\in\Sigma} f(\sigma) + \text{Prof}(\mathcal{A}_g) + \text{Prof}(\mathcal{A}_d) \tag{1}$$

25. Pour tout nombre entier $u \in [\![0,\lambda-1]\!]$, on a :

$$\Pi_{u,u} = f(\sigma_u) \tag{2}$$

On considère $u,v \in [\![0,\lambda-1]\!]$ tel que $u < v$ et $\mathcal{A}$ un arbre de code optimal de racine ρ pour le code $c_{u,v}$, alors et $\mathcal{A}_g$ (resp. $\mathcal{A}_d$) est un arbre de code optimal pour le code $c_{u,\rho-1}$ (resp. $c_{\rho+1,v}$). Le résultat de la question 24 assure donc que :

$$\Pi_{u,v} = \sum_{i=u}^{v} f(\sigma_i) + \Pi_{u,\rho-1} + \Pi_{\rho+1,v} \tag{3}$$

En particulier :

$$\Pi_{u,v} = \sum_{i=u}^{v} f(\sigma_i) + \min_{r\in[\![u,v]\!]} \{\Pi_{u,r-1} + \Pi_{r+1,v}\} \tag{4}$$

On généralise les égalités (3) et (4), on prenant comme convention $\Pi_{u,v} \overset{\text{def}}{=} 0$ si $u > v$.

26. Le principe de l'algorithme consiste à utiliser la formule (4) afin de remplir dynamiquement trois matrices Π, A et F de la façon suivante :

(1) Pour tout $u \in [\![0,\lambda-1]\!]$, on fait :

$$F_{u,u} \leftarrow f(\sigma_u)$$

Puis pour tout $k \in [\![1,\lambda-2]\!]$, on fait :

$$F(u,u+k) = F(u,u+k-1) + f(\sigma_{u+k})$$

(2) On initialise les deux matrices Π et A de taille $\lambda\times\lambda$ avec des coefficients égaux à 0 pour la matrice Π, avec `Nil` pour la matrice A.

(3) Pour tout $u \in [\![0,\lambda-1]\!]$, on fait :

$$\Pi_{u,u} \leftarrow f(\sigma_u)$$

On initialise aussi $A_{u,u}$ avec l'arbre dont la racine est σ_u étiquetée par $c(\sigma_u)$ et dont les fils gauche et droit sont `Nil`.

(4) Pour tout $k \in [\![1,\lambda-2]\!]$, on fait :

* Pour tout $u \in [\![1,\lambda-k-1]\!]$, on fait :

$$\Pi_{u,u+k} \leftarrow F_{u,u+k} + \min_{r\in[\![u,u+k]\!]} \{\Pi_{u,r-1} + \Pi_{r+1,u+k}\}$$

∗ Pour $r_0 \in [\![u,u+k]\!]$ tel que :

$$\Pi_{u,r_0-1} + \Pi_{r_0+1,u+k} = \min_{r\in[\![u,u+k]\!]} \{\Pi_{u,r-1} + \Pi_{r+1,u+k}\}$$

On stocke en $A_{u,u+k}$ l'arbre de code dont la racine est r_0 étiquetée par $c(\sigma_{r_0})$ et dont les sous-arbres gauches et droits sont respectivement A_{u,r_0-1} et $A_{r_0+1,u+k}$.

La complexité des étapes (1) et (2) est $\mathcal{O}(\lambda^2)$.
La complexité de l'étape (3) est $\mathcal{O}(\lambda)$.
Pour l'étape (4), la complexité du calcul du minimum à chaque étape est $\mathcal{O}(\lambda)$.
On remplit dynamiquement 3 demi-matrices de taille $\lambda \times \lambda$, la complexité de l'algorithme est bien $\mathcal{O}(\lambda^3)$.

2.3 Un arbre de code optimal calculé plus rapidement

27. En supposant l'inégalité $\mathcal{E}(n)$ vérifiée pour tout $n \in [\![0,\lambda-2]\!]$, on peut optimiser l'étape (4) de l'algorithme de la question précédente de la façon suivante :

- Pour $u = v$, $r_{u,u}$ est égal à u.
- Pour $n \in [\![0,\lambda-2]\!]$, en supposant les $r_{u,v}$ connus pour toutes les couples (u,v) tels $v-u \leqslant n$. On cherche $r_{u,v+1}$ dans l'intervalle $[\![r_{u,v},r_{u+1,v+1}]\!]$ de longueur :

$$r_{u+1,v+1} - r_{u,v} + 1$$

Pour n fixé, déterminer tous les $r_{u,v+1}$ coûte donc le nombre de comparaisons suivant :

$$\sum_{u=0}^{\lambda-2-n} r_{u+1,v+1} - r_{u,v} + 1$$

Or, la somme est télescopique :

$$\sum_{u=0}^{\lambda-2-n} r_{u+1,v+1} - r_{u,v} + 1 = \lambda - 1 - n + r_{\lambda-1-n,\lambda-1} - r_{0,n}$$

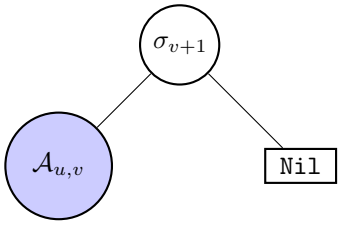
- Or :

$$\sum_{n=0}^{\lambda-2} \lambda - 1 - n + r_{\lambda-1-n,\lambda-1} - r_{0,n} = \frac{\lambda(\lambda-1)}{2} + \sum_{n=0}^{\lambda-2} \underbrace{r_{\lambda-1-n,\lambda-1} - r_{0,n}}_{\leqslant\lambda}$$

Finalement, en limitant les comparaisons aux intervalles $[\![r_{u,v},r_{u+1,v+1}]\!]$, le nombre de comparaisons effectué est $\mathcal{O}(\lambda^2)$.

28. La définition d'un arbre de code assure que si τ appartient au sous-arbre droit de σ alors $\tau > \sigma$. Puisque σ_v est un maximum strict de l'ensemble $\{\sigma_u, \cdots, \sigma_v\}$, σ_v ne possède pas de fils droit dans l'arbre de code qui représente le code $c_{u,v}$.
29. On considère $\mathcal{B}$ un arbre de code représentant le code $c_{u,v+1}$. Dans cet arbre, le sommet σ_{v+1} est le sommet le plus à droite.

- Si σ_{v+1} est le sommet de l'arbre $\mathcal{B}$, celui-ci est de la forme ci-contre :
où $\mathcal{A}_{u,v}$ est un arbre de code pour $c_{u,v}$.



Dans le l'arbre de code $\mathcal{B}$ tous les sommets de $\mathcal{A}_{u,v}$ ont une profondeur majorée de 1 par rapport à leur profondeur dans l'arbre $\mathcal{A}_{u,v}$. On a donc :

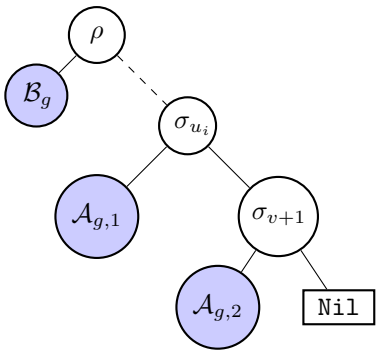
$$\text{Prof}(\mathcal{B}) = \sum_{i=u}^{v} f(\sigma_i) + \text{Prof}(\mathcal{A}_{u,v})$$

En considérant l'arbre $\mathcal{A}'_{u,v}$ obtenu à partir de $\mathcal{A}_{u,v}$ en ajoutant le sommet σ_{v+1} comme fils droit du sommet qui contient la lettre σ_v, on a :

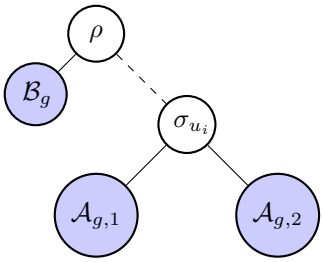
$$\text{Prof}\left(\mathcal{A}'_{u,v}\right) = \text{Prof}\left(\mathcal{A}_{u,v}\right)$$

En particulier :

$$\text{Prof}\left(\mathcal{B}\right) \geqslant \text{Prof}\left(\mathcal{A}'_{u,v}\right)$$



- Si σ_{v+1} n'est le sommet de l'arbre $\mathcal{B}$, celui-ci est de la forme ci-contre :
  Dans cet arbre si $\sigma_{u_i} \neq \sigma_v$, σ_v est nécessairement contenu dans le sommet le plus à droite du sous-arbre $\mathcal{A}_{g,2}$.
  Si $\sigma_{u_i} = \sigma_v$, alors nécessairement $\mathcal{A}_{g,2} = \texttt{Nil}$.
  * Dans la cas où $\sigma_{u_i} \neq \sigma_v$, on note $\mathcal{A}$ l'arbre obtenu à partir de $\mathcal{B}$ en remplaçant le sommet σ_{v+1} par la racine du sous-arbre $\mathcal{A}_{g,2}$:



l'arbre $\mathcal{A}$ dans le cas où $\sigma_{u_i} \neq \sigma_v$

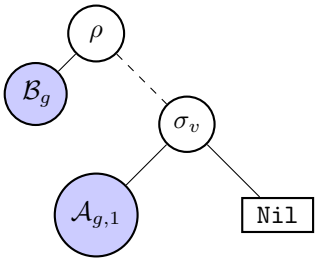
En considérant l'arbre $\mathcal{A}'$ obtenu à partir de $\mathcal{A}$ en ajoutant le sommet σ_{v+1} comme fils droit du sommet qui contient la lettre σ_v, $\mathcal{A}'$ est un arbre de code pour $c_{u,v+1}$ et on a :

$$\text{Prof}\left(\mathcal{A}'\right) = \text{Prof}\left(\mathcal{A}\right)$$

Par ailleurs, puisque les sommets du sous-arbre $\mathcal{A}_{g,2}$ dans l'arbre $\mathcal{B}$ ont une profondeur majorée de 1 par rapport aux mêmes sommets dans l'arbre $\mathcal{A}'$, on a :

$$\text{Prof}\left(\mathcal{B}\right) \geqslant \text{Prof}\left(\mathcal{A}'\right)$$

  * Dans le cas où $\sigma_{u_i} = \sigma_v$, en notant $\mathcal{A}$ l'arbre défini par :



l'arbre $\mathcal{A}$ dans le cas où $\sigma_{u_i} = \sigma_v$

L'arbre $\mathcal{B}$ est l'arbre obtenu à partir de $\mathcal{A}$ en ajoutant à $\mathcal{A}$ le sommet σ_{v+1} comme fils droit du sommet qui contient la lettre σ_v. En particulier :

$$\text{Prof}\left(\mathcal{B}\right) = \text{Prof}\left(\mathcal{A}'\right) = \text{Prof}\left(\mathcal{A}\right)$$

L'étude précédente assure donc que si $\mathcal{A}$ est un arbre de code optimal pour le code $c_{u,v}$, l'arbre de code $\mathcal{A}'$, obtenu à partir de $\mathcal{A}$ en ajoutant à $\mathcal{A}$ le sommet σ_{v+1} comme fils droit du sommet qui contient la lettre σ_v, est un arbre de code optimal pour le code $c_{u,v+1}$.

30. Commençons par un cas particulier :
Si pour tout $x \in [\![u,v]\!]$, on a $f(\sigma_x) = 0$, on montre facilement que l'application $\pi_{u,v+1}$ est définie par :

$$\pi_{u,v+1} : \begin{array}{ccc} \mathbb{R}_+ & \to & \mathbb{R}_+ \\ x & \mapsto & x \end{array}$$

Il suffit pour cela de considérer un arbre de code dont σ_{v+1} est la racine.
Par ailleurs, pour tout couple de nombres entiers (u',v') tels que $u \leqslant u' \leqslant v' \leqslant v+1$, on a $I = \mathbb{R}_+$ et :

$$r_{u',v'} = \sigma_{v'}$$

Cas général :

Pour tout arbre de code $\mathcal{B}$ associé à $c_{u,v+1}$, on a, en posant $x \overset{\text{def}}{=} f(\sigma_{v+1})$:

$$\text{Prof}(\mathcal{B}) = \sum_{k=u}^{v} f(\sigma_k)\,\text{prof}_{\mathcal{B}}(\sigma_k) + x \cdot \text{prof}_{\mathcal{B}}(\sigma_{v+1})$$

En notant $\varphi_{\mathcal{B}}$ l'application définie par :

$$\varphi_{\mathcal{B}} : \begin{array}{ccc} \mathbb{R}_+ & \to & \mathbb{R}_+ \\ x & \mapsto & \displaystyle\sum_{k=u}^{v} f(\sigma_k)\,\text{prof}_{\mathcal{B}}(\sigma_k) + x \cdot \text{prof}_{\mathcal{B}}(\sigma_{v+1}) \end{array}$$

$\varphi_{\mathcal{B}}$ est une application affine sur $\mathbb{R}_+$ de pente $\text{prof}_{\mathcal{B}}(\sigma_{v+1})$.
L'ensemble $\mathcal{E}_{u,v+1}$ des arbres de code représentant $c_{u,v+1}$ est fini et, par définition, on a :

$$\forall x \in \mathbb{R}_+, \quad \pi_{u,v+1}(x) = \min_{\mathcal{B}\in\mathcal{E}_{u,v+1}} \{\varphi_{\mathcal{B}}(x)\} \tag{5}$$

Le minimum d'un ensemble fini de fonctions continues est continue. Ce résultat ce démontre par induction sur le cardinal de l'ensemble des fonctions considérées et s'appuie sur le fait que, pour deux applications f et g à valeurs réelles, on a :

$$\min(f,g) = \frac{1}{2}(|f-g| + f - g)$$

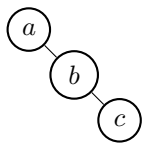
On montre aussi par récurrence sur le cardinal de l'ensemble des fonctions considérées que le minimum d'un ensemble de fonctions affines est une fonction affine par morceaux.
Il résulte de ces résultats que l'application $\pi_{u,v+1}$ est une application **continue** et **affine par morceaux**.
Sur chacun des intervalles ouverts I sur lequel la restriction de $\pi_{u,v+1}$ est affine, il existe $\mathcal{B} \in \mathcal{E}_{u,v+1}$ tel que :

$$\forall x \in I, \quad \pi_{u,v+1}(x) = \sum_{k=u}^{v} f(\sigma_k)\,\text{prof}_{\mathcal{B}}(\sigma_k) + x \cdot \text{prof}_{\mathcal{B}}(\sigma_{v+1})$$

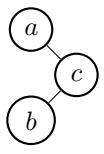
La pente est donc égale à $\text{prof}_{\mathcal{B}}(\sigma_{v+1})$.
Un résultat mathématique issu de l'égalité (5) assure également que :

$$\boxed{\text{prof}_{\mathcal{B}}(\sigma_{v+1}) \text{ décroît strictement à chaque changement de pente.}} \tag{6}$$

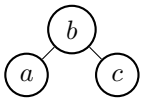
Intermède :
Illustrons la situation pour trois lettres a, b, c avec $f(a) = f(b) = \gamma$ et $f(c) = x$ variable.
On a 5 arbres de code possibles :



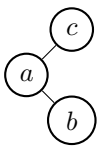
$\text{Prof}(\mathcal{A}) = 3\gamma + 3x$



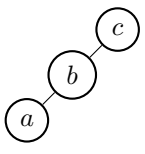
$\text{Prof}(\mathcal{A}) = 4\gamma + 2x$



$\text{Prof}(\mathcal{A}) = 3\gamma + 2x$

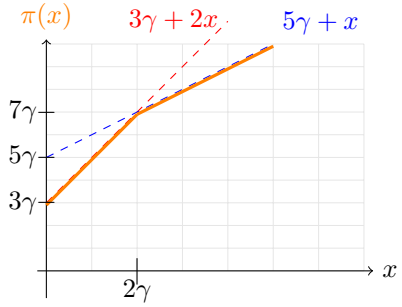


$\text{Prof}(\mathcal{A}) = 5\gamma + x$



$\text{Prof}(\mathcal{A}) = 5\gamma + x$

On peut facilement représenter la fonction π :



On a donc à considérer deux intervalles :

$$I_1 \overset{\text{def}}{=}]0,2\gamma[, \qquad I_2 \overset{\text{def}}{=}]2\gamma, +\infty[$$

En considérant cette fois la restriction du code à l'alphabet $\{b,c\}$, toujours avec $f(b) = \gamma$ et $f(c) = x$, on a deux arbres de code possibles conduisant aux deux fonctions :

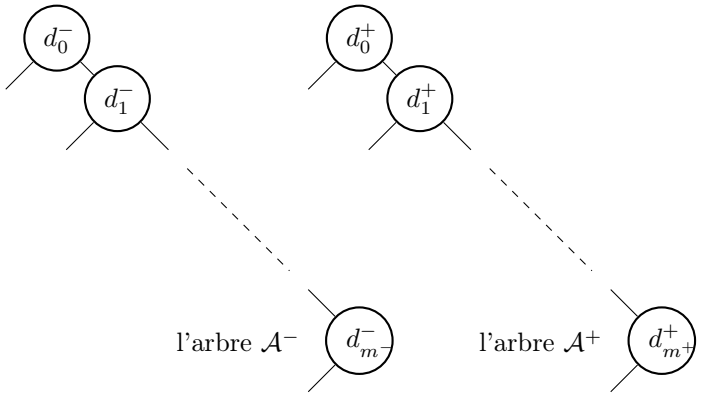
$$x \mapsto \gamma + 2x, \qquad x \mapsto 2\gamma + x$$

Ici, les deux intervalles à considérer sont $]0,\gamma[$ et $]\gamma, +\infty[$, le résultat proposé est donc trivial si $v' \leqslant v$ et **faux** pour $v' = v+1$.

31. Par construction les suites $\left(d_k^-\right)$ et $\left(d_k^+\right)$ sont des suites strictement croissantes de nombres entiers strictement inférieur à $v+1$, l'algorithme de construction se termine donc et les deux suites sont finies.
En ce qui concerne la deuxième partie du résultat, commençons par observer :
En prenant l'exemple de notre intermède avec $\alpha \overset{\text{def}}{=} 2\gamma$, on a :

$$\left(d_0^-, d_1^-\right) = (b,c), \qquad \left(d_0^+\right) = (c)$$

On a donc $m^- = 1$ et $m^+ = 0$. Le résultat est donc vérifié dans ce cas particulier.
Dans le cas général, il s'agit de démontrer que la suite $\left(d_k^-\right)$ est strictement plus longue que la suite $\left(d_k^+\right)$.
L'idée consiste à remarquer qu'un arbre de code optimal $\mathcal{A}^-$ associé à $c_{u,v+1}$ pour $f(\sigma_{v+1}) \in I^-$ et un arbre de code optimal $\mathcal{A}^+$ associé à $c_{u,v+1}$ pour $f(\sigma_{v+1}) \in I^+$ ont de la forme suivante :



La longueur des suites $\left(d_k^-\right)$ et $\left(d_k^+\right)$ sont donc respectivement égales à la profondeur de d_{m^-} et d_{m^+}, c'est-à-dire σ_{v+1} dans les arbres $\mathcal{A}^-$ et $\mathcal{A}^+$.

Le résultat (6) assure donc que $m^- < m^+$.

32. On suppose que l'encadrement $\mathcal{E}(n)$ est vérifié pour $n \in \mathbb{N}$. On considère deux nombres entiers naturels u et v tels que $v - u \leqslant n+1$.
On suppose enfin que $d_0^+ < d_0^-$.
On peut donc considérer $p \in \mathbb{N}^*$ tel que $d_0^+ + p = d_0^-$.
Puisque pour tout $k \in [\![1,p]\!]$, on a :

$$v + 1 - \left(d_0^+ + k\right) \leqslant n$$

En utilisant la deuxième inégalité de $\mathcal{E}(n)$, on obtient :

$$\forall k \in [\![1,p]\!], \quad r_{d_0^+ + k, v+1} \leqslant r_{d_0^+ + k + 1, v+1}$$

Par transitivité sur les p inégalités précédentes, on obtient en particulier :

$$r_{d_0^+ + 1, v+1} \leqslant r_{d_0^+ + p + 1, v+1}$$

Ce qui se lit encore :

$$\underbrace{r_{d_0^+ +1,v+1}}_{=d_1^+} \leqslant \underbrace{r_{d_0^- +1,v+1}}_{=d_1^-} \tag{7}$$

On a donc établi par (7) que $d_1^+ \leqslant d_1^-$. Si on avait $d_1^+ < d_1^-$, on obtiendrait par le même raisonnement $d_2^+ \leqslant d_2^-$.
Or, puisque $d_{m^+}^+ = d_{m^-}^- = \sigma_{v+1}$ et que $m^- > m^+$, on ne peut pas avoir des inégalités strictes à chaque étape.
On peut donc considérer le plus petit indice $s < m^+$ tel que $d_s^- = d_s^+$. On a alors, par définition de s :

$$\forall \ell \in [\![0,s-1]\!], \quad d_\ell^+ < d_\ell^-$$

En remplaçant dans $\mathcal{A}^-$ le **sous-arbre arbre droit** du sous-arbre enraciné en d_s^- par le **sous arbre droit** du sous-arbre de $\mathcal{A}^+$ enraciné en d_s^+, on obtient un arbre $\mathcal{B}$ de code optimal pour $c_{u,v+1}$ pour $f(\sigma_{v+1}) \in I^+$.
En effet les deux arbres obtenus par émondage des deux sous-arbres droits sont des arbres de code optimaux associés à c_{u,d_s} où d_s est la valeur commune de d_s^- et d_s^+ :

$$\sum_{k=u}^{d_s} f(\sigma_k) \operatorname{prof}_{\mathcal{A}^-}(\sigma_k) = \sum_{k=u}^{d_s} f(\sigma_k) \operatorname{prof}_{\mathcal{A}^+}(\sigma_k)$$

Les deux sous-arbres droits sont quant à eux des sous-arbres de code associés à $c_{d_s+1,v+1}$.
La racine de $\mathcal{B}$ est d_0^- avec $d_0^+ < d_0^-$ ce qui contredit la définition de $r_{u,v+1}$ pour $f(\sigma_{v+1}) \in I^+$.
Finalement :

$$d_0^+ \geqslant d_0^-$$

33. On montre le résultat par induction sur n.
On remarque préalablement que pour $n = 0$, on a $u = v$ et :

$$r_{u,u} = u, \quad r_{u,u+1} \in \{u, u+1\}, \quad r_{u+1,u+1} = u+1$$

L'encadrement $\mathcal{E}(0)$ est donc vérifié.
On considère $n \in [\![0,\lambda-3]\!]$ et on suppose $\mathcal{E}(n)$.
On considère également $u \in [\![0,\lambda-2]\!]$ tel que $u+n+1 \leq \lambda-2$. Les résultats précédents assurent qu'en ajoutant σ_{u+n+2} à $\Sigma_{u,u+n+1}$, on a $r_{u,u+n+2} \geqslant r_{u,u+n+1}$.
En effet si $f(\sigma_{u+n+2}) = 0$, le résultat de la question 29 assure que $r_{u,u+n+2}$ est au moins égal à $r_{u,u+n+1}$.
Par ailleurs, le résultat de la question 32 assure que la fonction $r_{u,u+n+2}(x)$ est une fonction croissante de x, valeur de $f(\sigma_{u+n+2})$.
En particulier :

$$\forall x \in \mathbb{R}^+, \quad r_{u,u+n+2}(x) \geqslant r_{u,u+n+1}$$

On applique symétriquement le résultat à l'ajout d'une lettre strictement inférieure à toutes les autres lettres pour obtenir la deuxième inégalité cherchée.