Sujet Mines-Ponts 2020 : corrigé

1. `fonction1` est de type `texte -> int array`.
Montrons par récurrence que cette fonction renvoie un tableau T de taille λ tel que, étant donnée un texte $t = [u_1, \cdots, u_n]$ (où $n = |t|$), pour chaque $i \in [\![0, \lambda - 1]\!]$ $T.(i)$ est le nombre d'occurrences de la lettre σ_i dans le texte s.
Si $n = 0$, alors la liste t est vide ; on renvoie un tableau de taille λ dont toutes les valeurs sont vides ; en outre, $|t|_\sigma = 0$ pour toute lettre σ ; la tableau renvoyé est bien celui des valences des lettres.
Soit $n \in \mathbb{N}$. Supposons que le résultat soit celui décrit ci-dessus pour tous les textes de taille n. Soit $t = u :: t'$ un texte de taille $n+1$; t' est donc un texte de taille n. L'appel à `fonction1` pour t' renvoie donc le tableau des valences pour t' ; en outre, $|t|_\sigma = |t'|_\sigma$ pour tout $\sigma \neq \sigma_u$ et, pour $\sigma = \sigma_u$, $|t|_\sigma = |t'|_\sigma + 1$; par conséquent, le tableau T' obtenu par l'appel `fonction1 tprime` et l'instruction `theta.(u)<-theta.(u)+1` créent bien le tableau T des occurrences du texte t. Le résultat renvoyé par la fonction `fonction1` sur la liste $t = u :: t'$ est bien le tableau des occurrences des lettres de Σ dans t.
2.

```
let cree_repartition t = let l=ref [] and tab=fonction1 t in
    for k=0 to Array_length t -1 do
        if theta.(i)>0 then l:=(i,theta.(i)):: !l
    done;
    !l;;
```

3. La fonction `fonction1` a une complexité temporelle $O(\lambda) + O(|t|)$ car la création initiale du tableau prend un temps de l'ordre de $O(\lambda)$ et les modifications du tableau prennent une complexité $O(|t|)$.
La fonction `cree_repartition` a donc une complexité temporelle $O(\lambda + |t|)$ pour l'utilisation de `fonction1` puis $O(\lambda)$ pour la boucle suivante ; au total, on a donc une complexité temporelle de $O(\lambda + |t|) + O(\lambda) = O(\lambda + |t|)$.
4. La fonction `fonction1` crée et utilise un tableau de taille λ donc a une complexité spatiale de $O(\lambda)$. La complexité spatiale est donc en $O(\lambda)$.
5. La liste renvoyée est de taille la valence $[t]$. La complexité spatiale est donc $O([t])$.
6. Un courriel est de taille $|t|$ de l'ordre de quelques dizaines ou quelques centaines de caractères. En outre, λ est l'ordre du million. On a donc $|t| = o(\lambda)$. Les complexités des fonctions précédentes sont donc en $O(\lambda)$.
7.

```
let rec incremente_repartition r u= match r with
    | [] -> (u,1)::[]
    | (v,i)::rprime -> if u=v then (v,i+1)::rprime else (v,i)::incremente_repartition rprime u;;
```

8.

```
let rec cree_modulaire t m= match t with
    | [] -> Array_make m []
    | u::tprime -> let theta=cree_modulaire tprime m in
                   theta.(u mod m)<-incremente_repartition theta.(u mod m) m; theta;;
```

9.

```
let valence tab =let l=ref 0 in
    for k=0 to Array_length tab -1 do
        l:=!l+List_length tab.(k)
    done;
    !l;;
```

10. On a $(Z_{l_0}|X_{i_0} = l_0) = |\{i \in [\![1, v]\!] \backslash \{i_0\} \,/\, X_i = l_0\}| + 1 = \sum_{i \neq i_0} \mathbb{1}_{X_i = l_0} + 1$ et, pour tout $l \neq l_0$, $(Z_l||X_{i_0} = l_0) = |\{i \in [\![1, v]\!] \backslash \{i_0\} \,/\, X_i = l_0\}| = \sum_{i \neq i_0} \mathbb{1}_{X_i = l_0}$. En outre, comme les X_i suivent une loi uniforme, $E(\mathbb{1}_{X_i = l_0}) = P(X_i = l_0) = \frac{1}{m}$.
Ainsi, $E(Z_{l_0}|X_{i_0} = l_0) = 1 + \sum_{i \neq i_0} \frac{1}{m} = 1 + \frac{v-1}{m}$ et, pour tout $l \neq l_0$, $E(Z_l|X_{i_0} = l_0) = \frac{v-1}{m}$.
11. La fonction `cree_modulaire` commence par créer un tableau de taille m ce qui prend une complexité temporelle de $O(m)$.
Pour chaque lettre du texte, la fonction `cree_modulaire` fait un appel à la fonction `incremente_repartition` pour une lettre l ; cette fonction parcourt la liste en position $l \mod m$ qui a une taille Z_l ; d'après la question précédente, cet appel a une complexité $\frac{v-1}{m}$ ou $1 + \frac{v-1}{m}$, c'est-à-dire au plus $1 + \frac{v-1}{m}$.
Comme la fonction `cree_modulaire` parcourt toutes les lettres du texte, il y a $|t|$ appels à la fonction `incremente_repartition` ce qui donne une complexité temporelle au plus de $|t| \times (1 + \frac{v-1}{m}) = |t| + \frac{(v-1)|t|}{m}$.
Au total, la complexité temporelle est donc de $O(m) + O(|t| + \frac{(v-1)|t|}{m}) = O(m + |t| + \frac{(v-1)|t|}{m})$.
12. Cette fonction crée un tableau de taille m ; chaque élément l du tableau est une liste. Chaque lettre apparait une seule fois dans une seule de ces listes. La somme des tailles des listes est donc la valence v. La complexité spatiale est donc $m + v$.
13. On cherche à minimiser $m + |t| + \frac{v-1}{m}|t|$ pour un nombre $|t|$ fixé de l'ordre de la centaine et un nombre v de l'ordre de la centaine

également.
En calculant la dérivée de la fonction $m \mapsto m + |t| + \frac{v-1}{m}|t|$, on trouve que le nombre m minimisant la complexité vérifie $1 - \frac{v-1}{m^2}|t| = 0$, c'est-à-dire $m = \sqrt{(v-1)|t|}$ qu'on peut approximer par $\sqrt{10^2 \times 10^2} = 10^2$.
14. Comme pour tout $\tau \neq \tau'$ telles que $\tau \leq \sigma_{v-1}$ et $\tau' \leq \sigma_{v-1}$, $A(I(\tau)) \neq A(I(\tau'))$, $I(\tau) \neq I(\tau')$. La fonction I restreinte aux τ tels que $\tau \leq \sigma_{v-1}$ est donc injective ; de plus, $I(\{\tau \,/\, \tau \leq \sigma_{v-1}\})$ contient les lettres du mot w, en nombre v, d'après (ii).
La fonction I restreinte aux τ tels que $\tau \leq \sigma_{v-1}$ et à valeurs dans les lettres du mot w est donc bijective. Ainsi, pour toute lettre τ telle que $\tau \leq \sigma_{v-1}$, $I(\tau)$ est une lettre du mot w.
15. Supposons que la lettre σ est présente dans le mot w. Alors il existe $\tau \leq \sigma_{v-1}$ tel que $I(\tau) = \sigma$. On a donc $A(\sigma) = A(I(\tau)) = \tau$; ainsi, $A(\sigma) = \tau \leq \sigma_{v-1}$ et $I(A(\sigma)) = I(\tau) = \sigma$.
Réciproquement, supposons que $A(\sigma) \leq \sigma_{v-1}$ et $I(A(\sigma)) = \sigma$. Alors $\sigma = I(A(\sigma)) = I(\tau)$ où $\tau \leq \sigma_{v-1}$. D'après la question précédente, $\sigma = I(\tau)$ est présente dans w.
16.

```
let est_present theta u = let (v,f,i,a)=theta in a.(u)<v && i.(a.(u))=u;;
```

La fonction ne fait que des appels aux tableaux A et I de θ et des comparaisons sur ces valeurs. La fonction va donc bien se terminer. La correction découle directement de la question 15.
17.

```
let incremente_tableaucreux theta u= let (v,f,i,a)=theta in
    if est_present theta u then (f.(u)<-f.(u)+1; (v,f,i,a) )
    else ( i.(v)<-u; a.(u)<-v; f.(v)<-1; (v+1,f,i,a) );;
```

18.

```
let rec cree_tableaucreux t= match t with
    | [] -> make_creux lambda
    | u::tprime -> let theta=cree_tableaucreux tprime in incremente_tableaucreux theta u;;
```

19. La création initiale d'un tableau creux prend une complexité bornée. Chaque lettre de w, le mot considéré, exige un appel à la fonction d'incrémentation. Celle-ci effectue un appel à `est_present` et à un nombre borné d'appels à des fonctions de complexité constante. La fonction `est_present` a une complexité constante. La fonction `est_present` a donc une complexité bornée, toute comme la fonction d'incrémentation.
La fonction `cree_tableaucreux` a donc une complexité $O(|t|)$.
20. Cette complexité est $O(\lambda)$.
21. La pire complexité est la complexité spatiale qui est de $O(\lambda)$ avec λ de l'ordre du million, ce qui serait une mauvaise complexité. Toutefois, la complexité spatiale n'est pas limitante avec les grandes capacités de stockage modernes. La solution du tableau creux est faisable mais non réaliste.
22.

```
let rec encodeur t c =
    let rec encodeur_lettre u c = match c with
        | Nil -> failwith "lettre introuvable"
        | Noeud( v, i, c1, c2) when v=u -> i
        | Noeud( v, i, c1, c2) when v<u -> encodeur_lettre u c2
        | Noeud( _, _, c1, _) -> encodeur_lettre u c1 in
    match t with
    | [] -> []
    | u::tprime -> (encodeur_lettre u c)@(encodeur tprime c );;
```

23. La fonction fait appel pour chaque lettre σ d'un mot w à la fonction `encodeur_lettre` qui a une complexité $prof_c(\sigma)$ et effectue une concaténation avec une première liste de taille $c(\sigma)$. La complexité totale est donc $\sum\limits_{\sigma\in w} |w|_\sigma(c(\sigma) + prof_c(\sigma)) = \sum\limits_{\sigma\in w} c(\sigma)|w|_\sigma + \sum\limits_{\sigma\in w} prof_c(\sigma) \leq L|w| + \sum\limits_{\sigma\in w} |w|_\sigma prof_{\mathcal{A}}(\sigma)$.
24. Si τ est une lettre de $\mathcal{A}' = \mathcal{A}_g$ ou $\mathcal{A}' = \mathcal{A}_d$, $prof_{\mathcal{A}}(\sigma) = 1 + prof_{\mathcal{A}'}$. Si σ est la racine de $\mathcal{A}$, Σ_g et Σ_d les lettres du sous-arbre gauche et droit de $\mathcal{A}$, on a $Prof(\mathcal{A}) = f(\sigma) \times prof(\sigma) + \sum\limits_{\sigma\in\Sigma_g} f(\sigma)(prof_{\mathcal{A}_g}(\sigma) + 1) + \sum\limits_{\sigma\in\Sigma_d} f(\sigma)(prof_{\mathcal{A}_d}(\sigma) + 1) = f(\sigma) + \sum\limits_{\sigma\in\Sigma_g} f(\sigma) + \sum\limits_{\sigma\in\Sigma_d} f(\sigma) + Prof(\mathcal{A}_g) + Prof(\mathcal{A}_d) = \sum\limits_{\sigma\in\Sigma} f(\sigma) + Prof(\mathcal{A}_g) + Prof(\mathcal{A}_d)$.
25. L'alphabet $\Sigma_{u,u}$ est $\{\sigma_u\}$, représenté par l'arbre de racine u, de sous-arbres droit et gauche vides. La profondeur pondérée de cet arbre est $f(\sigma_u)$. On a donc $\Pi_{u,u} = f(\sigma_u)$.
En outre, un arbre de code optimal est constitué d'une racine r, $r \in [\![u, v]\!]$, d'un sous-arbre gauche $\mathcal{A}_g$ représentant un code de $\{\sigma_u, \cdots, \sigma_{r-1}\}$ et d'un sous-arbre droit $\mathcal{A}_d$ représentant un code de $\{\sigma_{r+1}, \cdots, \sigma_v\}$. On a donc $\Pi_{u,v} = \sum\limits_{u\leq i\leq v} f(\sigma_i) + Prof(\mathcal{A}_g) + Prof(\mathcal{A}_d)$ avec $Prof(\mathcal{A}_g) \geq \Pi_{u,r-1}$, $Prof(\mathcal{A}_d) \geq \Pi_{r+1,v}$. On a donc $\Pi_{u,v} \geq \sum\limits_{u\leq i\leq v} f(\sigma_i) + \Pi_{u,r-1} + \Pi_{r+1,v}$.
On a donc $\Pi_{u,v} \geq \sum\limits_{u\leq i\leq v} f(\sigma_i) + \min\limits_{u\leq r\leq v}(\Pi_{u,r-1} + \Pi_{r+1,v})$.
En outre, soit $r_0 \in [\![u, v]\!]$ tel que $\Pi_{u,r_0-1} + \Pi_{r_0+1,v} = \min\limits_{u\leq r\leq v}(\Pi_{u,r-1} + \Pi_{r+1,v})$. Soient $\mathcal{A}_g$ un arbre optimal représentant un code de Σ_{u,r_0-1} et $\mathcal{A}_d$ un arbre optimal représentant un code de $\Sigma_{r_0+1,v}$. Alors l'arbre, noté $\mathcal{A}$, de racine r_0, de sous-arbre gauche $\mathcal{A}_g$ et de sous-arbre droit $\mathcal{A}_d$, est un arbre représentant un code de $\Sigma_{u,r_0-1} \cup \{\sigma_{r_0}\} \cup \Sigma_{r_0+1,v} = \Sigma_{u,v}$. La profondeur pondérée de cet arbre est $Prof(\mathcal{A}) = \sum\limits_{u\leq i\leq v} f(\sigma_i) + Prof(\mathcal{A}_g) + Prof(\mathcal{A}_d)$; comme $\mathcal{A}_g$ et $\mathcal{A}_d$ sont optimaux, $Prof(\mathcal{A}_g) = \Pi_{u,r_0-1}$, $Prof(\mathcal{A}_d) = \Pi_{r_0+1,v}$. On a donc $Prof(\mathcal{A}) = \sum\limits_{u\leq i\leq v} f(\sigma_i) + \Pi_{u,r_0-1} + \Pi_{r_0+1,v}$; par conséquent, $\Pi_{u,v} \leq Prof(\mathcal{A}) = \sum\limits_{u\leq i\leq v} f(\sigma_i) + \Pi_{u,r_0-1} + \Pi_{r_0+1,v}$.

En conclusion, $\Pi_{u,v} = \sum_{u \leq i \leq v} f(\sigma_i) + \min_{u \leq r \leq v} (+\Pi_{u,r-1} + \Pi_{r+1,v})$.

26.

fonction 1 : prend en argument deux indices u et v, la fonction de poids f et un tableau Π de taille $\lambda \times \lambda$ contenant toutes les valeurs $\Pi_{u,r}$ pour $u \leq r \leq v-1$ et $\Pi_{r,v}$ pour $u+1 \leq r \leq v$. Cette fonction calcule $\Pi_{u,v}$ et le met dans le tableau Π en position (u,v).

```
début fonction 1
minimum:=Pi(u+1,v) ; somme:=f(u) ; r0:=u
pour r=u+1 à v faire
    somme:=somme+f(r) ; nouveau:=Pi(u,r-1)+Pi(r+1,v)
    si nouveau<minimum alors (minimum:=nouveau ; r0:=r)
fin faire
Pi(u,v):=minimum+somme
A(u,v):=Noeud(r0, c(r0), A(u,r0-1), A(r0+1,v))
fin fonction 1
```

fonction 2 : crée un tableau Π contenant les valeurs $\Pi_{u,u}$ en indices (u,u) et 0 ailleurs ainsi qu'un tableau A contenant les arbres ayant pour seule étiquette σ_u ; prend en arguments la fonction de pondération f et le code c.

```
début fonction 2
Pi:=créer matrice de taille (lambda,lambda) de coefficients 0
A:=crée matrice de taille (lambda,lambda) de coefficients Nil
pour u=0 à lambda-1 faire
    Pi(u,u):=f(u)
    A(u,u):=Noeud(u, c(u),Nil,Nil)
fin faire
renvoyer Pi
fin fonction 2
```

fonction 3 : renvoie des tableaux Π et A avec, pour tout $u \leq v$, $\Pi(u,v) = \Pi_{u,v}$ et $A(u,v)$ est un arbre de code optimal pour $c_{u,v}$; prend en argument la fonction f.

```
début fonction 3
Pi:=fonction 2(f)
pour u=0 à lambda-2 faire
    pour v=u+1 à lambda-1 faire
        fonction 1(u,v,f,Pi)
    fin faire
fin faire
renvoyer Pi
fin fonction 3
```

La fonction principale, fonction 3, effectue un appel à fonction 2, qui a une complexité $O(\lambda)$ pour la modification de la diagonale et $O(\lambda^2)$ pour la création de la matrice initiale ; la fonction principale est ensuite constituée de deux boucles imbriquées énumérant toutes les valeurs $0 \leq u < v \leq \lambda - 1$; pour chaque valeur $u < v$, un appel à la fonction 1 est effectuée ; cette fonction étant constituée d'une boucle de longueur $v-u$, cet appel à la fonction 1 a une complexité de l'ordre de $v-u$. La complexité totale est donc $\sum_{0 \leq u < v \leq \lambda-1} (v-u)$. Or, pour tout u, v, $v-u = O(\lambda)$, et le nombre de couples (u,v) est $O(\lambda^2)$. On a donc $\sum_{0 \leq u < v \leq \lambda-1} (v-u) = \sum_{0 \leq i < v \leq \lambda-1} O(\lambda) \leq \lambda^2 \times O(\lambda) = O(\lambda^3)$.
La complexité temporelle est donc bien $O(\lambda^3)$.

27. On définit initialement trois tableaux `A`, `R`, `S` et `P` de taille $\lambda \times \lambda$ tels que $A(u,v)$, $R(u,v)$, $S(u,v)$, $P(u,v)$ contiendront respectivement un arbre de code optimal pour $c_{u,v}$, $r_{u,v}$, $\sum_{u \leq i \leq v} f(\sigma_i)$ et $\Pi_{u,v}$.

```
début algorithme
pour u allant de 0 à lambda -1 faire
   R.(u,u):=u ; A.(u,u)=Noeud(u,c(u),Nil,Nil) ; S(u,u)=f(u) ; Pi(u,u):=f(u)
fin faire
pour u allant de 0 à lambda-1 faire
    pour v allant de u+1 à lambda-1 faire
        S(u,v)=S(u,v-1)+f(v)
    fin faire
fin faire
pour u allant de 0 à lambda-1 faire
    pour v allant de u+1 à lambda-1 faire
        r0:=R(u,v-1) ; minimum:=Pi(u,r0-1)+Pi(r0+1,v)
        pour r allant de R(u,v-1)+1 à R(u+1,v) faire
            si minimum<=Pi(u,r-1)+Pi(r+1,v) alors (r0:=r ; minimum:=Pi(u,r-1)+Pi(r+1,v) )
        fin faire
        R(u,v):=r0 ; Pi(u,v)=S(u,v)+minimum ; A(u,v):=Noeud(r0,c(r0),A(u,r0-1),A(r0+1,v))
    fin faire
fin faire
```

La première boucle est en $O(\lambda)$, la deuxième en $O(\lambda^2)$.
Concernant la troisième, elle parcourt, pour chaque $0 \leq u < v \leq \lambda - 1$, l'intervalle $[\![r_{u,v-1}, r_{u+1,v}]\!]$ qui est de taille $r_{u+1,v} - r_{u,v-1} + 1$;

elle contient un nombre d'itérations égal à $\sum_{0\le u<v\le\lambda-1}(r_{u+1,v}-r_{u,v-1}+1)=\sum_{0\le u<v\le\lambda-1}r_{u+1,v}-\sum_{0\le u<v\le\lambda-1}r_{u,v-1}+\sum_{0\le u<v\le\lambda-1}1=\sum_{1\le u\le v\le\lambda-1}r_{u,v}-\sum_{0\le u\le v\le\lambda-2}r_{u,v}+\sum_{v=1}^{\lambda-1}\sum_{u=0}^{v-1}1=\sum_{u=1}^{\lambda-1}r_{u,v}-\sum_{v=0}^{\lambda-2}r_{u,v}+\sum_{v=1}^{\lambda-1}v=\sum_{u=1}^{\lambda-1}r_{u,v}-\sum_{v=0}^{\lambda-2}r_{u,v}+\frac{\lambda(\lambda-1)}{2}$. En outre, $0\le r_{u,v}\lambda-1$ donc $\sum_{u=1}^{\lambda-1}r_{u,v}\le\lambda^2=O(\lambda^2)$.

Par conséquent, le nombre d'itérations de la troisième boucle est $O(\lambda^2)$. Comme chaque itération ne comprend qu'un nombre borné d'opérations, la complexité de cette boucle est $O(\lambda^2)$. La complexité totale est donc la somme de ces trois dernières, à savoir $O(\lambda^2)$.

28. Pour toute lettre τ du sous-arbre droit de la racine σ_v, on a $\tau>\sigma_v$ (voir la définition d'un arbre de code). Or l'arbre considéré est un arbre de code $c_{u,v}$ donc toutes les lettres du code τ vérifient $\tau\le\sigma_v$. Ainsi, il n'y a aucune lettre dans le sous-arbre droit, qui est donc vide.

29. Comme $f(\sigma_{v+1})=0$, l'arbre considéré a pour poids $Prof(\mathcal{A})=\Pi_{u,v}$.

Soit un arbre $\mathcal{A}'$ de code représentant $c_{u,v+1}$. Alors, enlevant l'étiquette σ_{v+1}, on obtient un arbre de code représentant $c_{u,v}$. Comme $f(\sigma_{v+1})=0$, cet arbre est de profondeur pondérée $Prof(\mathcal{A}')$. Par conséquent, $Prof(\mathcal{A}')\ge\Pi_{u,v}$.

En résumé, pour tout arbre $\mathcal{A}'$ de code représentant $c_{u,v+1}$, $Prof(\mathcal{A})\ge\Pi_{u,v}$ avec égalité pour l'arbre $\mathcal{A}$ proposé par l'énoncé. Ainsi, cet arbre est optimal.

30. Par définition, $Prof(\mathcal{A})=f(\sigma_{v+1})prof(\sigma_{v+1})+\sum_{u\le\sigma\le v}f(\sigma)prof_{\mathcal{A}}(\sigma)$.

Pour un arbre donné $\mathcal{A}$, la fonction $\pi_{\mathcal{A}}:f(\sigma_{v+1})\mapsto Prof(\mathcal{A})=f(\sigma_{v+1})prof_{\mathcal{A}}(\sigma_{v+1})$ est une fonction affine de pente $prof_{\mathcal{A}}(\sigma)$.

De plus, les arbres représentant le code $c_{u,v+1}$ sont en nombre fini. Si ces arbres sont $\mathcal{A}_1,\cdots,\mathcal{A}_N$, on a alors $\pi_{u,v+1}=\min_{1\le i\le N}\pi_{\mathcal{A}_i}$.

La fonction $\pi_{u,v+1}$ est donc le minimum d'un nombre fini de fonctions affines. Une telle fonction est continue et affine par morceaux.

Sur chaque intervalle où $\pi_{u,v+1}$ est affine par morceaux, un arbre $\mathcal{A}_i$ est un arbre de code optimal sur tout l'intervalle. Sa racine ne varie pas. Ainsi $r_{u,v+1}$ ne varie pas ; en outre, comme en enlevant les étiquettes.

En outre, pour $u'>u$, $\pi_{u',v+1}$ est aussi le minimum de fonctions affines qui ont même pente que dans le cas de $r_{u,v+1}$. Son graphe est donc parallèle à celui de $\pi_{u,v+1}$. On en déduit que, sur un intervalle où la pente de $\pi_{u,v+1}$ est constante, celle de $\pi_{u',v+1}$ l'est aussi, ce qui correspond au fait que $r_{u',v+1}$ est constant.

Enfin, par hypothèse, les $r_{u,v'}$, $v'<v+1$ sont constantes.

31. Les suites (d_k^+) et (d_k^-) sont strictement croissantes (car $d_{k+1}=r_{d_k+1,v+1}\le d_k+1$ pour tout k) et majorées par $v+1$. Il n'y a donc qu'un nombre fini de valeurs à ces suites.

Par définition, $d_m=\sigma_{v+1}$.

En outre, un arbre de code optimal de $c_{u,v+1}$ peut-être construit à partir de la racine d_0, d'un sous-arbre gauche et d'un sous-arbre droit qui est un arbre de code optimal pour $c_{d_0+1,v+1}$; la racine de ce dernier peut-être constitué de d_1 et d'un sous-arbre droit de code optimal pour $c_{d_1+1,v+1}$. Par récurrence, on peut construire un arbre de code optimal de $c_{u,v+1}$ ayant pour racines successives des sous-arbres droits $d_0,d_1,\cdots,d_m$. Ainsi, m est la profondeur de $d_m=\sigma_{v+1}$. Cette profondeur est la pente de la fonction $\pi_{u,v+1}$.

Enfin, un minimum de fonctions affines est convexe, donc la pente des fonctions décroît, et même strictement lors d'une rupture de pente. Ainsi, la pente de la fonction $\pi_{u,v+1}$ est supérieure strictement sur I^- à celle sur I^+. Par conséquent, la profondeur $prof(\sigma_{v+1})$, qui vaut m^- sur I^- et m^+ sur I^+, est supérieure strictement sur I^- à celle sur I^+. On a donc $m^->m^+$.

32. On a $d^+_{m^+}=\sigma_{v+1}=d^-_{m^-}$; or $m^->m^+$ donc $d^+_{m^+}>d^-_{m^+}$. Supposons que $d_0^+<d_0^-$.

On a $d_0^+\ge u$ donc $v-(d_0^++1)\le v-u-1\le n+1-1=n$; d'après $\mathcal{E}(n)$, on a $r_{d_0^++1,v+1}\le r_{d_0^+2,v+1}$.

De même, $v-(d_0^++2)\le n$ donc $r_{d_0^++2,v+1}\le r_{d_0^++3,v+1}$. Par récurrence, pour tout $p\ge d_0^++1$, $r_{d_0^++1,v+1}\le r_{p,v+1}$; en particulier, pour $p=d_0^-+1>d_0^->d_0^+$, $r_{d_0^-+1,v+1}\ge r_{d_0^++1,v+1}$ donc $d_1^-=r_{d_0^-+1,v+1}\ge r_{d_0^++1,v+1}=d_1^+$.

En refaisant la démonstration précédente pour le couple (d_1^+,d_1^-), on montre que si $d_1^->d_1^+$, alors $d_2^-\ge d_2^+$.

Par récurrence, s'il n'existe pas $s<m^+$ tel que $d_s^+=d_s^-$, alors, pour tout $0\le l<m^+$, $d_l^->d_l^+$ donc $d^-_{m^+}\ge d^+_{m^+}$ ce qui contredit le fait que $d^+_{m^+}>d^-_{m^+}$.

Par conséquent, il existe s tel que $d_s^+=d_s^-$. En choisissant s minimal pour cette propriété, on a donc pour tout $0\le l<s$, $d_l^+<d_l^-$.

Comme vu précédemment, on peut considérer un arbre de code optimal de $c_{u,v+1}$ $\mathcal{A}^+$ ayant pour racines des sous-arbres droits successifs $d_0^+,d_1^+,\cdots,c^+_{m^+}$ pour l'intervalle I^+ et un arbre $\mathcal{A}^-$ de code optimal $c_{u,v+1}$ ayant pour racines des sous-arbres droits successifs $d_0^-,\cdots,d^-_{m^-}$.

Considérons l'arbre $\mathcal{A}_0$ obtenu en échangeant, dans l'arbre $\mathcal{A}^-$, le sous-arbre de $\mathcal{A}^-$ de racine $d_s^-=d_s^+$ par celui de racine d_s^+ dans $\mathcal{A}^+$. Cet arbre a pour racines des sous-arbres droits successifs $d_0^-,\cdots,d_s^-=d_s^+,d^+_{s+1},\cdots,d^+_{m^+}=\sigma_{v+1}$.

Cet arbre a pour sous-arbre droit issu de $d_s^-=d_s^+$ un arbre de code optimal pour $c_{d_s^-+1,v+1}$. L'arbre obtenu en enlevant ce sous-arbre est un arbre de code optimal pour c_{u,d_s} (d'après la constance des $c_{u,',v'}$ de la question 30). L'arbre $\mathcal{A}_0$ est donc un arbre de code optimal pour $c_{u,v+1}$ de racine d_0^- pour $\sigma_{v+1}=d^+_{m^+}$ et $f(\sigma_{v+1})\in I^+$. Or l'hypothèse $d_0^+<d_0^-$ contredit la minimalité de $d_0^+=r_{u,v+1}$.

On a donc $d_0^-\le d_0^+$. Par conséquent la fonction $f(\sigma_{v+1})\mapsto r_{u,v+1}$ est une fonction constante par morceaux et croissante !

33. Pour $n=0$, si $v-u\le n$, alors $v=u$; $r_{u,u}=u$; $r_{u,u+1}=u$ ou $r_{u,u+1}$ donc $r_{u,u}=u\le r_{u,u+1}\le u+1=r_{u+1,u+1}$. On a donc $\mathcal{E}(0)$.

Soit $0\le n\le\lambda-2$. Supposons $\mathcal{E}(n)$. Soit (u,v) tel que $v-u\le n+1$.

Si $f(\sigma_{v+1})=0$, d'après la question 29, il y a un arbre de code pour $c_{u,v+1}$ optimal de racine $r_{u,v}$ donc $r_{u,v+1}\ge r_{u,v}$.

D'après la question précédente, par croissance de la fonction $f(\sigma_{v+1})\mapsto r_{u,v}$, si $f(\sigma_{v+1})\ge0$, $r_{u,v+1}\ge r_{u,v}$.

L'autre inégalité $r_{u,v+1}\le r_{u+1,v+1}$ se montre de même en étudiant la fonction $\pi_{u,v}:f(\sigma_u)\mapsto\Pi_{u,v}$.