Chapitre ou TD ou TP un

# Un titre

MPI/MPI*, lycée Faidherbe

## I Fonction de distribution des lettres d'un texte

### I.1 Avec un tableau exhaustif et une liste

**Question 1** Le type de la variable est donné : `texte`. Le résultat en cas de texte vide est créé par `array make lambda 0`, ce doit donc être un tableau d'entiers :

```
fonction1 : int list -> int array}
```

On lit qu'on incrémente `t.(u)` quand on passe d'un texte `tprime` à `u :: tprime`, il est donc raisonnable de penser que `fonction1` calcule les occurrences des lettres d'un texte.
On démontre par récurrence sur $|t|$ que `fonction1 t` renvoie un tableau `theta` de $\lambda$ nombres entiers tel que `tab.(u)` donne le nombre d'occurrences de $u$ dans le texte $t$pour tout lettre $u$.

- Si $|t| = 0$, le texte est vide et la fonction renvoie bien un tableau de 0.
- On suppose que le résultat est vrai pour les textes de longueur $n$.
  Soit`t` un texte de longueur $n+1$, il est de la forme $u :: t'$, où $u$ est un caractère et $t'$ un texte de longueur $n$. D'après l'hypothèse de récurrence le résultat, `theta`, dénombre les occurrences des lettres de $t'$.
  Pour toute lettre $x \neq u$, les nombres d'occurences de $x$ dans $t'$ et $t$ sont égales et le nombre d'occurences de $u$ dans $t$ est supérieur de 1 au nombre d'occurences de $u$ dans $t'$. C'est bien ce qu'effectue la fonction : la propriété est vraie pour les textes de longueur $n+1$.

La récurrence est démontrée.

**Question 2**

```
let cree_repartition t  =
  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
```

**Question 3** La création de `theta` dans `fonction1` est de complexité linéaire en $\lambda$.
On effectue $|t|$ modifications du tableau dans `fonction1`.
`fonction1` est de complexité en $\mathcal{O}(\lambda + |t|)$.
`cree_repartition` appelle `fonction1` et parcourt ensuite tous les éléments du tableau pour réaliser à chaque fois des opérations à coût constant d'où une complexité ajoutée en $\mathcal{O}(\lambda)$.
La complexité totale est donc un $\mathcal{O}(\lambda + |t|)$.

**Question 4** La complexité spatiale de la fonction `cree_repartition` est celle de `fonction1` qui crée un tableau de taille $\lambda$ : c'est un $\mathcal{O}(\lambda)$.

**Question 5** Le résultat de la fonction `cree_repartition t` est une liste de $[t]$ couples.

**Question 6** Un texte de courriel comporte généralement moins de $\lambda$ caractères donc la complexité temporelles devient un $\mathcal{O}(\lambda)$ qui reste aussi la complexité spatiale. On ne peut pas diminuer $\lambda$ car on crée et parcourt un tableau de cette taille.
Un texte en français, même avec les signes de ponctuation, peut être codé dans une table ASCII de taille 256 : la taille de `cree_repartition t` est majorée par 256.

## I.2 Avec une table modulaire

**Question 7** L'énoncé est trop flou, presque incompréhensible.
Le contexte suggère qu'une répartition est une liste (ce que seul l'exemple permet de savoir) qui contient les couples $(x, k)$ où $k$ estle nombre d'occcurrences de $x$ dans le texte et $x$ appartenant à l'ensemble $\Sigma^{[\ell]}$ tel que la liste est à la $\ell$-ième case.
De plus la question ne dit pas si $u$ appartient à $\Sigma^{[\ell]}$ : on va le supposer.

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

**Question 8**

```
let rec cree_modulaire t m =
  let theta = Array.make m [] in
  let modif u =
    let k = u mod m in
    theta.(k) <- incremente_repartition theta.(k) u in
  List.iter modif t;
  theta
```

**Question 9**

```
let valence theta =
  let m = Array.length theta in
  let v = ref 0 in
  for i=0 to m-1 do
    v := !v + List.length theta.(i) done;
 !v
```

**Question 10** **La notion d'espérance conditionnelle est hors-programme.**
On va interpréter la question en notant $A = [\![1; v]\!] \setminus \{i_0\}$ et
$Z'_\ell = \left|\left\{i \in A, X_i = \ell\right\}\right|$ pour $\ell \neq \ell_0$ et $Z'_{\ell_0} = \left|\left\{i \in A, X_i = \ell_0\right\}\right| + 1$. On cherche $\mathbb{E}(Z'_\ell)$.
On note $Y_{\ell,i}$ la variable aléatoire qui vaut 1 si $X_i = \ell$ et 0 sinon ; on a $\mathbb{E}(Y_{\ell,i}) = \frac{1}{m}$.
Or, pour $\ell \neq \ell_0$, $Z'_\ell = \sum_{i\in A} Y_{\ell,i}$ donc $E(Z'\ell) = \sum_{i\in A} E(Y_{\ell,i}) = \dfrac{v-1}{m}$
De même $Z'_{\ell_0} = 1 + \sum_{i\in A} Y_{\ell_0,i}$ donc $E(Z'_{\ell_0}) = 1 + \dfrac{v-1}{m}$.

**Question 11** La notion de complexité moyenne est toujours difficile. La démonstration qui suit est sommaire.
Le coût de la fonction `incremente_repartition` est linéaire en la taille de la répartition donnée en entrée. La longueur moyenne des liste est $1 + \frac{v-1}{m}$ car on est certain qu'au moins un élément a été inséré : ainsi la complexité moyenne de l'appel de `incremente_repartition` pour tous les caractères est un $\mathcal{O}\left(|t| \cdot \left(1 + \frac{v-1}{m}\right)\right)$. On doit ajouter la complexité de la création du tableau

de taille $m$.
La complexité moyenne est donc un $\mathcal{O}\left(|t| \cdot \left(1 + \frac{v-1}{m}\right) + m\right)$.

**Question 12** La complexité spatiale au sens strict est constante car on renvoie les structures créées. La taille du résultat est la somme des tailles du tableau `theta` renvoyé et des tailles des sous-listes le composant, donc $\mathcal{O}(m+v)$.

**Question 13** Pour minimiser la complexité on cherche $m$ qui minimise $m \mapsto |t| \cdot (1 + \frac{v-1}{m}) + m$.
La dérivée est $-\frac{|t|(v-1)}{m^2} + m$ qui donne un minimum pour $m = \sqrt{|t|(v-1)}$
Pour un texte de courriel en français, $|t|$ est de l'ordre de 1000 et $v$ de l'ordre de 100 : on peut prendre $m$ de l'ordre de 300.
Ce n'est pas utile en comparaison de l'algorithme avec un tableau exhaustif avec $\lambda = 256$.

## I.3 Avec un tableau creux

**Question 14** On note $V$ l'ensemble des $v$ symboles du mot $w$ et $\Sigma_k = \{\sigma_i \ ; \ 0 \leqslant i < k\}$.
Le point (ii) se traduit par $V \subset I(\Sigma_v)$, on a donc $v|V| \leqslant |I(\Sigma_v)| \leqslant |\Sigma_v| = v$.
Il y a donc égalité des cardinaux donc des ensembles en raison de l'inclusion : $V = I(\Sigma_v)$ c'est-à-dire $I(\sigma_k) \in V$ pour $k < v$.

**Question 15** Si $I'$ est la restriction de $I$ à $\Sigma_v$ la question précédente montre que $I'$ est une surjection de $\Sigma_v$ vers $V$. Comme les deux ensembles ont le même cardinal, $I'$ est une bijection et la propriété (iii) signifie que $A'$, restriction de $A$ à $V$ est l'inverse de $I'$.
Ainsi, si $\sigma \in V$, $A(\sigma) = A'(\sigma) \in \Sigma_v$ et $I\big(A(\sigma)\big) = I'\big(A'(\sigma)\big) = \sigma$.
Inversement, si $A(\sigma) = \tau \in \Sigma_v$ et $I\big(A(\sigma)\big) = \sigma$ alors $\sigma = I(\tau) = I'(\tau) \in V$ car $\tau \in \Sigma_v$.

**Question 16** L'usage d'un tuple n'est pas très malin, il aurait mieux valu un enregistrement.

```
let est_present theta u =
  let v, tabF, tabI, tabA = theta in
  let s = tabA.(u) in
  s < v && tabI.(s) = u
```

La terminaison est triviale et la correction est le résultat de l'exercice 15.

**Question 17**

```
let incremente_tableaucreux theta u =
  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 = function
  |[] -> make_creux lambda
  |u :: reste -> let theta = cree_tableaucreux reste in
                 incremente_tableaucreux theta u
```

**Question 19** D'après l'énoncé, la complexité de `make_creux` est constante.
Chaque appel de `est_present` et de `incremente_tableaucreux` se fait en temps constant donc la complexité totale est en $\mathcal{O}(|t|)$.

**Question 20** Comme on crée 3 tableaux de taille $\lambda$, la complexité spatiale est en $\mathcal{O}(\lambda)$.

**Question 21** La complexité spatiale en $\lambda$ est trop grosse pour un courriel de base.

# II Un encodeur optimal

## II.1 Codes et codages

**Question 22** On doit rechercher le code dans un arbre binaire de recherche.

```
let rec code u c =
  match c with
  |Nil -> failwith "Le code n'existe pas"
  |Noeud(s, bi, g , d) ->
    if u = s
    then bi
    else begin
      if u < s
      then code u g
      else code u d
```

Il suffit alors de coller les codes obtenus

```
let rec encodeur texte c =
  match texte with
  |[] -> []
  |x :: reste -> (code x c) @ (encodeur reste c)
```

**Question 23** La recherche du code d'une lettre $\sigma$ est proportionnelle à la profondeur de cette lettre dans l'arbre de code, $\text{prof}_{\mathcal{A}}(\sigma)$. Il y a $|t|_\sigma$ occurrence de $\sigma$ dans le texte.
Il faut ajouter la complexité de la concaténation, on ajoute en tête $|t|$ codes de longueurs $L$ au plus d'où une complexité en $\mathcal{O}\left(L\cdot|t| + \sum_{\sigma\in\Sigma} |t|_\sigma \cdot \text{prof}_{\mathcal{A}}(\sigma)\right)$.

## II.2 Arbre de code optimal

Pour les deux parties restantes, pour simplifier l'écriture, on identifie $\sigma_u$ et $u$ : les lettres sont des entiers.

**Question 24** Si $\sigma \in \mathcal{A}_g$ alors $\text{prof}_{\mathcal{A}}(\sigma) = \text{prof}_{\mathcal{A}_g}(\sigma) + 1$ car $\mathcal{A}_g$ est placé à une profondeur de 1. De même $\text{prof}_{\mathcal{A}}(\sigma) = \text{prof}_{\mathcal{A}_d}(\sigma) + 1$ pour $\sigma \in \mathcal{A}_d$. Pour la racine, on a $\text{prof}_{\mathcal{A}}(\rho) = 1$ d'où

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

Il y a un abus d'écriture car $\mathcal{A}_g$ et $\mathcal{A}_d$ n'ont pas $\Sigma$ comme ensemble de nœuds ; on généralise la notion de profondeur en posant $\text{prof}_{\mathcal{A}}(\sigma) = 0$ si sigma n'est pas dans $\mathcal{A}$.

**Question 25** On pose $\Pi_{k,k-1} = 0$, cela correspond à un arbre vide. r Pour une lettre $u$, l'arbre est réduit à sa racine et sa profondeur est $\Pi_{u,u} = f(u)$.

Pour les lettres $u$ à $v$, un arbre de code a une racine $\rho$ avec $u \leqslant \rho \leqslant v$, le fils gauche est un arbre de code pour les lettres de $u$ à $\rho - 1$ et le fils droit est un arbre de code pour les lettres de $\rho - 1$ à $v$. Pour une racine $\rho$ fixée, on a $\text{prof}(\mathcal{A}) = \sum_{\sigma=u}^{v} f(\sigma) + \text{prof}(\mathcal{A}_g) + \text{prof}(\mathcal{A}_d)$ ; cette valeur atteint son minimum pour $\text{prof}(\mathcal{A}_g) = \Pi_{u,\rho-1}$ et $\text{prof}(\mathcal{A}_d) = \Pi_{\rho+1,v}$.

Ainsi $\Pi_{u,v} = \sum_{\sigma=u}^{v} f(\sigma) + \min\{\Pi_{u,\rho-1} + \Pi_{\rho+1,v} \ ; \ u \leqslant \rho \leqslant v\}$.

**Question 26** Les questions précédentes suggèrent une programmation dynamique.

On va calculer dans une matrice les valeurs des sommes $\sum_{\sigma=u}^{v} f(\sigma)$ et des profondeurs minimales.

Pour pouvoir aussi construire l'arbre qui donne la valeur minimale on garde aussi dans un tableau la valeur de la racine qui correspond à ce minimum.
On commence par le tableau des sommes des fréquences.

```
let tableauF f = (* f est la fonction de fréquence *)
  let tabF = Array.make_matrix lambda lambda 0 in
  for i = 0 to (lambda -1) do
    tabF.(i).(i) <- f i;
    for j = i+1 to (lambda - 1) do
      tabF.(i).(j) <- tabF.(i).(j-1) + f j done done;
  tabF
```

La complexité est en $\mathcal{O}(\lambda^2)$.
On calcule ensuite les profondeurs minimales et les racines que l'on remplit par diagonales.
On doit tenir compte des cas où la racine est au bord car $\Pi_{i,i-1}$ n'est pas défini pour $i = 0$ et $\Pi_{j+1,j}$ n'est pas défini pour $j = \lambda - 1$. On peut remplacer la lecture de `pi` par une fonction qui la généralise.

```
let fPi pi i j =
  if (j = 0) || (i = lambda)
  then 0
  else pi.(i).(i)
```

On calcule alors, dans un même temps, le tableau des valeurs $\Pi_{i,j}$ et des racines qui donnent les valeurs optimales. On ne renvoie que $\Pi_{0,\lambda-1} = \text{prof}(\mathcal{A})$ et les racines, pour calculer $\mathcal{A}$.

```
let piRacines f =
  let ff = tableauF f in
  let pi = Array.make_matrix lambda lambda 0 in
  let rac = Array.make_matrix lambda lambda 0 in
  (* initialisations sur la diagonale *)
  for i = 0 to (lambda -1) do
    pi.(i).(i) <- ff.(i).(i);
    rac.(i).(i) < - i done;
  (* pour chaque sur-diagonale à distance de d *)
  for d = 1 to (lambda - 1) do
    for i = 0 to (lambda - 1 - d) do
      (* on calcule Pi entre i et i+d *)
      let j = i + d in
      (* on initialise la valeur avec la racine en i *)
      pi.(i).(j) <- (fPi pi i (i-1)) + (fPi pi (i+1) j);
      rac.(i).(j) <- i;
      (* on calcule les valeurs pour les autres racines *)
      for r = i+1 to j do
        let som = (fPi pi i (r-1)) + (fPi pi (r+1) j) in
        (* si c'est mieux *)
        if som < pi.(i).(j)
        (* on actualise *)
        then (pi.(i).(j) <- som; rac.(i).(j) <- r) done;
      (* ne pas oublier la somme des f(k) *)
      pi.(i).(j) <- pi.(i).(j) + ff.(i).(j) done done;
  pi.(0).(lambda-1), rac
```

Les trois boucles imbriquées donnent une complexité en $\mathcal{O}(\lambda^3)$.
Il reste alors à déterminer $\mathcal{A}$.

```
let arbreOpt f c =
  let prof, rac = piRacines f in
  let rec construire u v =
    if u > v
    then Nil
    else begin
      if u = v
      then Noeud (k, c k, Nil, Nil)
      else let r = rac.(u).(v) in
        Noeud (r, c r, construire u (r-1), construire (r+1) v)
    end in
  prof, construire 0 (lambda -1)
```

## II.3 Un arbre de code optimal calculé plus rapidement

**Question 27** On modifie le calcul du minimum en ne cherchant la racine pour $\Pi_{i,j}$ qu'entre $r_{i,j-1}$ et $r_{i+1,j}$ ; les indices correspondants sont dans sur-diagonale précédente.
Pour un $d$ donné, pour chaque $i$, on fait $r_{i+1,j} - r_{i,j-1} + 1$ calculs pour chercher une valeur inférieure. Au total, pour $0 \leqslant i < \lambda - d$ et $j = i + d$, le nombre de calcules vérifie

$$\sum_{i=0}^{\lambda-d-1} (r_{i+1,j} - r_{i,j-1} + 1) = r_{\lambda-d,\lambda-1} - r_{0,d-1} + \lambda - d \leqslant \lambda - 1 - 0 + \lambda - d \leqslant 2\lambda.$$

Comme les autres calculs sont insérés dans deux boucles imbriquées, la complexité est en $\mathcal{O}(\lambda^2)$.

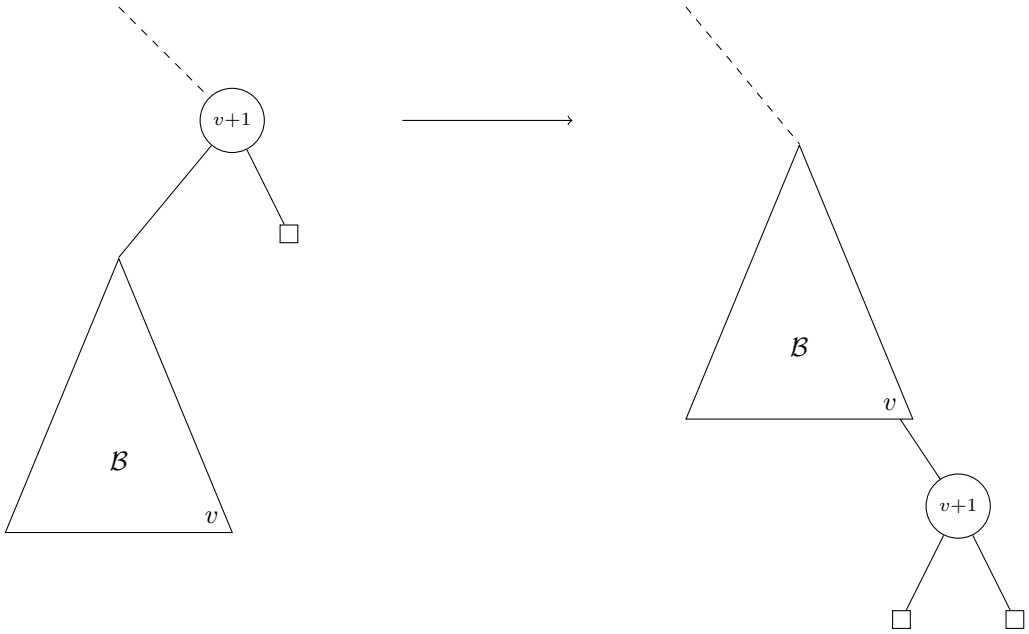
```
let piRacines f =
 let ff = tableauF f in
  let pi = Array.make_matrix lambda lambda 0 in
  let rac = Array.make_matrix lambda lambda 0 in
  for i = 0 to (lambda -1) do
    pi.(i).(i) <- ff.(i).(i);
    rac.(i).(i) <- i done;
  for d = 1 to (lambda - 1) do
    for i = 0 to (lambda - 1 - d) do
      let j = i + d in
      let r1 = rac.(i).(j-1) in
      let r2 = rac.(i+1).(j) in
      pi.(i).(j) <- fPi pi i (r1-1) + fPi pi (r1+1) j;
      rac.(i).(j) <- r1;
      for r = r1+1 to r2 do
        let som = fPi pi i (r-1) + fPi pi (r+1) j in
        if som < pi.(i).(j)
        then (pi.(i).(j) <- som; rac.(i).(j) <- r) done;
      pi.(i).(j) <- pi.(i).(j) + ff.(i).(j) done done;
  pi.(0).(lambda-1), rac
```

**Question 28** Un nœud d'un A.B.R. contient dans son fils droit des nœud de valeur supérieure. Si le nœud a une valeur maximale, son fils droit doit être vide.

**Question 29** On note que, si $\mathcal{A}$ est un arbre de code pour $c_{u,v}$ et si $\mathcal{A}'$ est l'arbre obtenu en ajoutant $v+1$ comme fils droit de $v$ alors $\text{prof}(\mathcal{A}') = \text{prof}(\mathcal{A})$.
Si $\mathcal{A}$ est optimal, alors $\text{prof}(\mathcal{A}') = \text{prof}(\mathcal{A}) = \Pi_{u,v}$ donc, comme $\mathcal{A}'$ est un arbre de code pour $c_{u,v+1}$, on a $\Pi_{u,v+1} \leqslant \Pi_{u,v}$.
Soit $\mathcal{A}_1$ un un arbre de code optimal pour $c_{u,v+1}$.
Si on effectue la recherche de $k$ ou de $k+1$ dans l'arbre binaire de recherche on parcourt les mêmes branches tant que la valeur du nœud n'est ni $k$, ni $k+1$. Quand on sépare $k$ et $k+1$, c'est donc qu'on est parvenu à un nœud de valeur $k$ ou $k+1$ : **soit $k+1$ est dans le fils droit de $k$, soit $k$ est dans le fils gauche de $k+1$.** Autrement, le plus petit ancêtre commun de $k$ et $k+1$ est soit $k$, soit $k+1$.

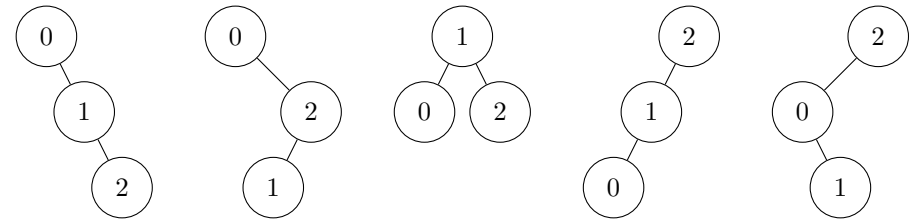
- Si $v+1$ est dans le fils droit de $v$ dans $\mathcal{A}_1$ alors, comme $v+1$ est la seule valeur supérieure à $v$, $v+1$ est le fils droit de $v$. Dans ce cas, il existe un arbre $\mathcal{B}$ tel que $\mathcal{B}' = \mathcal{A}_1$. On a alors $\Pi_{u,v+1} = \text{prof}(\mathcal{A}_1) = \text{prof}(\mathcal{B}') = \text{prof}(\mathcal{B}) \geqslant \Pi_{u,v}$.
- Si $v$ est dans le fils gauche de $v$ dans $\mathcal{A}_1$ que l'on note $\mathcal{B}$, alors $v+1$ n'a pas de fils droit et $v$ n'a pas de fils droit non plus. On peut donc construire l'arbre $\mathcal{B}'$ en ajoutant $v+1$ comme fils droit de $v$ et remplacer le nœud $v+1$ par $\mathcal{B}'$. On obtient ainsi un arbre $\mathcal{A}_2$ qui est encore un arbre de code pour $c_{u,v+1}$. Comme on a diminuée la profondeur des nœuds de $\mathcal{B}$, on a $\text{prof}(\mathcal{A}_2) \leqslant \text{prof}(\mathcal{A}_1)$. Or $\mathcal{A}_1$ est optimal donc $\mathcal{A}_2$ l'est aussi.
  On est ramené au cas précédent donc on a encore $\Pi_{u,v+1} \geqslant \Pi_{u,v}$.

Les deux inégalités $\Pi_{u,v+1} \leqslant \Pi_{u,v}$ et $\Pi_{u,v+1} \geqslant \Pi_{u,v}$ donnent l'égalité donc $\mathcal{A}'$, de profondeur $\Pi_{u,v}$ est optimal pour $c_{u,v+1}$




**Question 30** Soit $\mathcal{B}$ un arbre de code pour $c_{u,v+1}$, en isolant la contribution de $v+1$ et en notant $x = f(v+1)$, on a $\text{prof}(\mathcal{B}) = \sum_{k=u}^{v} \text{prof}_{\mathcal{B}}(k) \cdot f(k) + \text{prof}_{\mathcal{B}}(v+1) \cdot x = k_{\mathcal{B}} + \alpha_{\mathcal{B}} \cdot x$.
Alors $\pi_{u,v+1}(x) = \min\{\text{prof}(\mathcal{B}) \ ; \mathcal{B} \text{ arbre de code}\} = \min\{k_{\mathcal{B}} + \alpha_{\mathcal{B}} \cdot x \ ; \mathcal{B} \text{ arbre de code}\}$ est le minimum d'une famille finie de fonctions affines.
L'égalité $\min\{f,g\}(x) = \dfrac{f(x)+g(x)+|f(x)-g(x)|}{2}$ montre que le minimum de deux fonction continues est continu donc $\pi_{u,v+1}$ est continue.
De plus les graphes des fonctions affines sont des droites ; un nombre fini de droites admettent un nombre fini de points d'intersection et, entre deux points d'intersection consécutifs le minimum est un segment donc $\pi_{u,v+1}$ est affine par morceaux.
Dans un intervalle, plusieurs arbres peuvent donner la profondeur minimale ; dans ce cas, on choisit celui qui admet la racine maximale et on fait la même chose récursivement pour les sous-arbres. Dans ce cas les racines des sous-arbres sont les $r_{u',v'}$. On ne peut pas prouver la conservation de **tous** les $r_{u',v'}$.

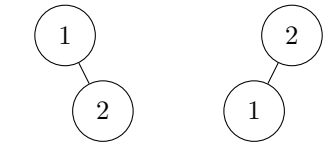
Voici un contre-exemple, $\lambda = 3$ et $f(0) = f(1) = 1$, $f(2) = x$.
On calcule les 5 arbres de codes et leur fonction de profondeur.




$\pi(x) = 3+3x$ $\pi(x) = 3+3x$ $\pi(x) = 3+2x$ $\pi(x) = 5+x$ $\pi(x) = 5+x$

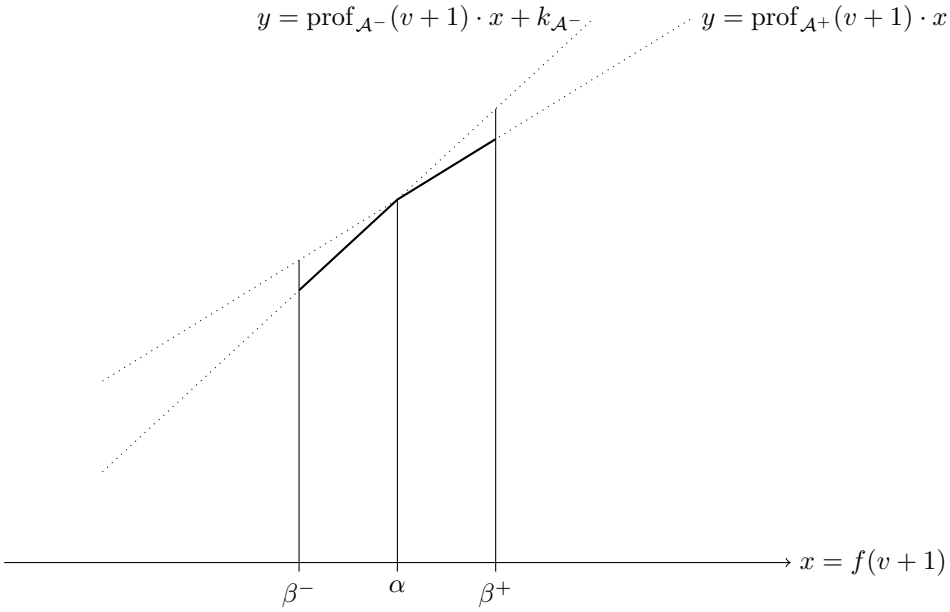
Pour $x < 2$, l'arbre optimal est le troisième avec $r_{0,2} = 1$ (et $r_{0,0} = 0$, $r_{2,2} = 2$).
Pour $x > 2$, le quatrième et le cinquième arbres conviennent avec $r_{0,2} = 2$ mais on doit choisir le quatrième pour prendre $r_{0,1} = 1$.

Cependant, si veut calculer $r_{1,2}$ dans le cas $x < 2$, on a deux arbres possibles et la valeur du minimum est
$1 + 2x$ pour $x < 1$ avec $r_{1,2} = 1$
ou $2 + x$ pour $x \in ]1; 2[$ avec $r_{1,2} = 2$.
$r_{1,2}$ n'est pas constant.




$\pi(x) = 1 + 2x$ $\pi(x) = 2 + x$

**Question 31** On remarque d'abord que les suites $\left(d_k^-\right)$ et $\left(d_k^+\right)$ sont par construction strictement croissante tant qu'elles n'atteignent pas la valeur $v - 1$. De plus elles sont à valeurs entières ; ce sont donc des suites finies dont la dernière valeur est $v + 1$.
On considère 2 arbres optimums pour $c_{u,v+1}$, $\mathcal{A}^-$ et $\mathcal{A}^+$, le premier est obtenu pour $x = f(v+1)$ dans $I^-$, le second pour $x \in I^+$.
Par définition $d_0^-$ est la racine de $\mathcal{A}^-$, puis $d_1^-$ est la racine du fils droit et ainsi de suite jusqu'à $d_{m-}^- = v + 1$ ; ainsi $m^- + 1$ est la profondeur de $v + 1$ dans $\mathcal{A}^-$ (la racine est à la profondeur 1).
De même $m^+ + 1$ est la profondeur de $v + 1$ dans $\mathcal{A}^+$.
Or, d'après la question précédente, l'équation de $\pi_{u,v+1}$ sur $[\beta^-; \alpha]$ est $y = \text{prof}_{\mathcal{A}^-}(v+1) \cdot x + k_{\mathcal{A}^-}$ et sur $[\alpha; \beta^+]$ l'équation est $y = \text{prof}_{\mathcal{A}^+}(v+1) \cdot x + k_{\mathcal{A}^+}$.
En utilisant la continuité en $\alpha$, on peut écrire ces équations $y = \text{prof}_{\mathcal{A}^-}(v+1) \cdot (x - \alpha) + \pi_{u,v+1}(\alpha)$ et $y = \text{prof}_{\mathcal{A}^+}(v+1) \cdot (x - \alpha) + \pi_{u,v+1}(\alpha)$ respectivement.
Comme l'équation $\text{prof}_{\mathcal{A}^+}(v+1) \cdot (x - \alpha) + \pi_{u,v+1}(\alpha)$ est la valeur minimale des profondeurs des arbres de code $[\alpha; \beta^+]$, on doit avoir $\text{prof}_{\mathcal{A}^-}(v+1) \cdot (x - \alpha) + \pi_{u,v+1}(\alpha) > \text{prof}_{\mathcal{A}^+}(v+1) \cdot (x - \alpha) + \pi_{u,v+1}(\alpha)$ pour $x \in [\alpha; \beta^+]$ d'où $m^- + 1 = \text{prof}_{\mathcal{A}^-}(v+1) > \text{prof}_{\mathcal{A}^+}(v+1) = m^+ + 1$.
On a bien $m^- > m^+$.




**Remarques**

- La remarque de la fin de la question 30 impose de considérer non seulement des changement de pente mais aussi des changement de position de racine maximale. Comme le nombre de sous-arbres reste fini, il n'y aura qu'un nombre fini de point de changement. Par contre on aura une inégalité large : $m^- \geqslant m^+$.
- On considère des racines maximales dans 2 arbres ; pour marquer la différence entre ces racines on les note $r^-$ ou $r^+$ pour $\mathcal{A}^-$ et $\mathcal{A}^+$ respectivement.
- La question 32 demande de prouver que si $\mathcal{E}(n)$ est vérifiée alors $d_0^- = r^- u, v + 1 \leqslant r^+ u, v + 1 = d_0^+$ pour $v = u + n + 1$. Je ne sais pas le prouver sans une récurrence sur ce dernier résultat.

Je pose donc la propriété

$$\mathcal{P}(n)\ ;\ \forall a,b\in[\![0;\lambda-1]\!]\ a\leqslant b\leqslant a+n \Longrightarrow \begin{cases} r^-_{a,b}\leqslant r^-_{a,b+1}\leqslant r^-_{a+1,b+1}\\ r^+_{a,b}\leqslant r^+_{a,b+1}\leqslant r^+_{a+1,b+1}\\ r^-_{a,b+1}\leqslant r^+_{a,b+1}\end{cases}$$

Dans la question 32, on prouve que $\mathcal{P}(n)$ implique $r^-u,v+1\leqslant r^+u,v+1$ pour $v=u+n+1$, et dans la question 33 on conclut la récurrence.

**Question 32**

On suppose que la propriété $\mathcal{P}(n)$ est vérifiée et on considère $u$ et $v$ avec $v=u+n+1$.
On suppose qu'on a $\mathcal{E}(n)$, c'est-à-dire
$r^-_{a,b}\leqslant r^-_{a,b+1}\leqslant r^-_{a+1,b+1}$ et $r^+_{a,b}\leqslant r^+_{a,b+1}\leqslant r^+_{a+1,b+1}$ pour $a\leqslant b\leqslant a+n$.
Si on a $d^-_0>d^+_0$, comme on a toujours $d^+_0>u$ donc $|v-d^+_0|\leqslant n$, on peut appliquer la seconde inégalité de $\mathcal{E}(n)$ : $r^+_{d^+_0,v+1}\leqslant r^+_{d^+_0+1,v+1}$.
De même $|v-(d^+_0+1)|\leqslant n$ donc $r^+_{d^+_0+1,v+1}\leqslant r^+_{d^+_0+2,v+1}$.
On peut prolonger jusqu'à $r^+_{d^-_0-1,v+1}\leqslant r^+_{d^-_0,v+1}$ d'où $r^+_{d^+_0,v+1}\leqslant r^+_{d^-_0,v+1}$
On a $d^-_0<d^-_1<\cdots<d^-_{m^+}<\cdots<d^-_{m^-}=v+1$.
En particulier $d^-_{m^+}<v+1=d^+_{m^+}$.
Si on a $d^-_0>d^+_0$, on considère le premier entier $s$ tel que $d^-_s\leqslant d^+_s$.

**Question 33**