**ÉCOLE DES PONTS PARISTECH,
ISAE-SUPAERO, ENSTA PARIS,
TÉLÉCOM PARIS, MINES PARISTECH,
MINES SAINT-ÉTIENNE, MINES NANCY,
IMT ATLANTIQUE, ENSAE PARIS, CHIMIE PARISTECH.**

**Concours Centrale-Supélec (Cycle International),
Concours Mines-Télécom, Concours Commun TPE/EIVP.**

CONCOURS 2020

ÉPREUVE D'INFORMATIQUE MP

Durée de l'épreuve : 3 heures

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Cette épreuve concerne uniquement les candidats de la filière MP.

Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :

INFORMATIQUE - MP

L'énoncé de cette épreuve comporte 9 pages de texte.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Les techniques de compression sans perte permettent d'encoder des données telles que des textes afin de les stocker en occupant un minimum d'espace. Elles s'appuient sur une estimation de la fréquence d'apparition des symboles qui composent le texte. Dans ce problème, nous étudions comment compter efficacement le nombre d'occurrences de chaque symbole dans un texte, que l'on entendra dans la suite comme une suite finie de numéros (par exemple, des numéros Unicode), et comment optimiser le fonctionnement d'un encodeur pour passer d'un texte à une suite de bits.

Préliminaires

L'épreuve est composée d'un problème unique, comportant 33 questions. Après cette section de préliminaires, le problème est divisé en deux parties indépendantes. Pour répondre à une question, un candidat pourra réutiliser le résultat d'une question antérieure, même s'il n'est pas parvenu à établir ce résultat.

Concernant la programmation

Il faudra coder des fonctions à l'aide du langage de programmation OCaml, tout autre langage étant exclu. Lorsque le candidat écrira une fonction, il pourra faire appel à d'autres fonctions définies dans les questions précédentes de la même sous-partie ; il pourra aussi définir des fonctions auxiliaires. Quand l'énoncé demande de coder une fonction, il n'est pas nécessaire de justifier que celle-ci est correcte, sauf si l'énoncé le demande explicitement. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile de tester si les hypothèses sont bien vérifiées dans le code de la fonction.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

On suppose codées les fonctions suivantes :

— `list_length`, de type `'a list -> int`, de complexité en temps linéaire, qui renvoie la longueur d'une liste,

— `list_append`, de type `'a list -> 'a list -> 'a list`, de complexité en temps linéaire en la longueur du premier argument, qui concatène deux listes,

— `array_make`, de type `int -> 'a -> 'a array`, de complexité en temps linéaire en la valeur du premier argument, qui crée un tableau d'une longueur spécifiée en premier argument et dont chacune des cases est initialisée à une valeur donnée en second argument,

— `array_length`, de type `'a array -> int`, de complexité en temps constante, qui renvoie la longueur d'un tableau.

On rappelle que `a mod b` renvoie le reste de la division euclidienne de a par b.

Dans les calculs de complexité en espace, on considérera qu'un entier occupe une place constante en mémoire quelle que soit sa valeur.

Concernant les objets manipulés et leur type

L'intervalle des entiers compris entre a et b est noté $[\![a, b]\!]$.

La lettre Σ désigne un *alphabet fini ordonné* de cardinal λ ; ses lettres, numérotées dans l'ordre croissant, sont les éléments

$$\sigma_0,\ \sigma_1,\ \sigma_2,\ \ldots,\ \sigma_{\lambda-2},\ \sigma_{\lambda-1}.$$

Par exemple, la norme *Unicode* est une norme fréquemment utilisée en informatique qui consiste à travailler sur un alphabet de 1 114 112 symboles. Elle permet d'identifier toute sorte de caractères (lettre de l'alphabet latin, idéogramme chinois, émoticône, symbole de l'alphabet phonétique international, etc.) par un numéro unique entre 0 et 1 114 111 quelle que soit la plate-forme informatique employée.

Définitions : Un mot sur un alphabet Σ est une suite finie de lettres de Σ. Leur ensemble est noté Σ^*, le mot vide est noté ε. Un mot possède plusieurs caractéristiques : la *longueur* d'un mot $w \in \Sigma^*$, notée $|w|$, est le nombre de lettres qui composent w ; la *fréquence d'une lettre* $\sigma \in \Sigma$ dans un mot $w \in \Sigma^*$, notée $|w|_\sigma$, est le nombre d'occurrences de σ dans w ; la *valence* d'un mot $w \in \Sigma^*$, notée $[w]$, est le nombre de symboles distincts qui composent w.

Un *texte* est une suite finie d'entiers de l'intervalle $[\![0, \lambda - 1]\!]$. Le mot associé au texte de s entiers $[u_1, u_2, u_3, \ldots, u_s]$ est le mot $\sigma_{u_1}\sigma_{u_2}\ldots\sigma_{u_s} \in \Sigma^*$.

Indication OCaml : On définit les types `unicode`, `texte` et la constante globale `lambda` par les déclarations suivantes :

```
type unicode = int;;
type texte = unicode list;;
let lambda = 1114112;;
```

Une fonction de l'alphabet Σ vers un ensemble X est représentée par un tableau de longueur λ de type `'a array` où `'a` est le type par lequel on représente les éléments de X.

1 Fonction de distribution des lettres d'un texte

Le but de cette partie est de produire une structure de données qui permette de vérifier la présence d'une lettre dans un mot et de stocker les valeurs non nulles de la *fonction de distribution* des fréquences des lettres dans un mot $w \in \Sigma^*$ définie comme suit

$$\begin{cases} \Sigma & \to & \mathbb{N} \\ \sigma & \mapsto & |w|_\sigma \end{cases}$$

de trois manières différentes et de comparer les complexités en temps et en espace de chaque approche.

1.1 Avec un tableau exhaustif et une liste

☐ 1 – On définit une fonction `fonction1` par le code suivant

```
let rec fonction1 (t:texte) =
   match t with
   | [ ] -> array_make lambda 0
   | u::tprime -> let theta = fonction1 tprime in
                  theta.(u)<-theta.(u)+1;
                  theta;;
```

Inférer le type de la fonction `fonction1`. Expliquer ce que calcule `fonction1 t` et le démontrer par récurrence sur $|t|$.

Définition : Soit Σ' un sous-alphabet de l'alphabet Σ. On appelle *répartition* du sous-alphabet Σ' dans le mot $w \in \Sigma^*$ toute liste constituée de tous les couples $(\sigma, |w|_\sigma)$ tels que $\sigma \in \Sigma'$ et $|w|_\sigma > 0$. Cette liste peut être dans un ordre quelconque. La répartition d'un sous-alphabet dans un texte est la répartition de ce sous-alphabet dans le mot associé au texte.
Par exemple, une répartition des lettres du mot *acad* est la liste $[(a, 2), (c, 1), (d, 1)]$.

Indication OCaml : En OCaml, on pourra se servir du type

```
type repartition = (unicode * int) list;;
```

☐ 2 – Écrire une fonction `cree_repartition`, de type `texte -> repartition` qui renvoie une répartition d'un texte en utilisant la fonction `fonction1` définie à la question 1.

☐ 3 – Déterminer la complexité en temps de la fonction `cree_repartition` en fonction du cardinal λ de l'alphabet et d'une caractéristique du texte donné en entrée.

☐ 4 – Déterminer la complexité en espace de la fonction `cree_repartition` en fonction du cardinal λ de l'alphabet.

☐ 5 – Donner sans justification un ordre de grandeur de la taille de la valeur de retour de `cree_repartition` en fonction d'une caractéristique du texte donné en entrée.

☐ 6 – Comparer les ordres de grandeur obtenus dans les trois questions précédentes quand le texte donné en entrée est le texte d'un courriel de la vie courante rédigé en langue française et Σ est l'alphabet Unicode.

1.2 Avec une table modulaire

Définition : Soient $w \in \Sigma^*$ un mot et $m \in \mathbb{N}^\star$ un entier non nul. Pour tout entier ℓ compris entre 0 et $m-1$, on note $\Sigma^{[\ell]}$ le sous-alphabet de Σ défini par

$$\Sigma^{[\ell]} = \{\sigma_u,\ u \in [\![0, \lambda - 1]\!]\ ;\ (u - \ell) \text{ est divisible par } m\}\,.$$

Une *table modulaire de comptage d'ordre m du mot w* est un tableau de longueur m dont la ℓ-ième case contient une répartition des lettres de $\Sigma^{[\ell]}$ dans w.
Par exemple, avec l'alphabet $\Sigma = \{a, b, c, d, e\}$ muni de l'ordre alphabétique et l'entier $m = 3$, une table modulaire de comptage d'ordre 3 du mot *acad* est le tableau de trois listes

$$\boxed{[(a,2);(d,1)]\ |\ []\ |\ [(c,1)]}.$$

☐ 7 – Écrire une fonction `incremente_repartition`, de type `repartition -> unicode -> repartition` telle que `incremente_repartition r u` renvoie la répartition d'un mot qui contient la lettre σ_u une fois de plus qu'un mot de répartition r.

☐ 8 – Écrire une fonction `cree_modulaire`, dont le type est `texte -> int -> repartition array`, qui utilise la fonction définie à la question 7, telle que la valeur de retour de `cree_modulaire t m` est une table modulaire de comptage d'ordre m du texte t.
Le recours à la fonction `fonction1` de la question 1 n'est pas autorisé pour répondre à cette question.

☐ 9 – Écrire une fonction `valence`, de type `repartition array -> int`, qui renvoie la valence d'un mot à partir de sa table modulaire de comptage.

☐ 10 – Soient m et v deux entiers non nuls et X_1, X_2, …, X_v des variables aléatoires discrètes à valeurs dans l'intervalle entier $[\![0, m-1]\!]$, mutuellement indépendantes et qui suivent la loi de distribution uniforme. Soient encore un entier i_0 fixé dans l'intervalle $[\![1, v]\!]$ et un entier ℓ_0 fixé dans l'intervalle $[\![0, m-1]\!]$. Pour tout entier ℓ compris entre 0 et $m-1$, on appelle Z_ℓ le cardinal

$$Z_\ell = |\{i \in [\![1, v]\!];\ X_i = \ell\}|.$$

Pour tout entier ℓ compris entre 0 et $m-1$, calculer l'espérance

$$\mathbb{E}[Z_\ell | X_{i_0} = \ell_0].$$

☐ 11 – Soit $w \in \Sigma^*$ le mot d'un texte t de valence v. En supposant que l'ensemble des v lettres distinctes présentes dans le mot w a été choisi uniformément dans l'ensemble des parties à v éléments de l'alphabet Σ, montrer que la complexité moyenne en temps de la fonction `cree_modulaire t m` est

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

☐ 12 – Donner sans justification la complexité en espace de la fonction `cree_modulaire t m` en fonction de m et d'une caractéristique du texte t.

☐ 13 – Quelle valeur numérique de m suggérez-vous de choisir lorsque t est le texte d'un courriel de la vie courante rédigé en langue française et Σ est l'alphabet Unicode ?

1.3 Avec un tableau creux

Définition : Un *tableau creux de comptage du mot* $w \in \Sigma^*$ est un quadruplet (v, F, I, A) formé d'un entier v et de trois fonctions $F : \Sigma \to \mathbb{N}$, $I : \Sigma \to \Sigma$ et $A : \Sigma \to \Sigma$ tels que

(i) l'entier v est la valence du mot w,

(ii) pour toute lettre $\sigma \in \Sigma$ présente dans le mot w, il existe une lettre $\tau \in \Sigma$ telle que $\tau \leqslant \sigma_{v-1}$ et $I(\tau) = \sigma$.

(iii) pour toute lettre $\tau \in \Sigma$ telle que $\tau \leqslant \sigma_{v-1}$, on a $A(I(\tau)) = \tau$,

(iv) pour toute lettre $\sigma \in \Sigma$ présente dans le mot w, on a $F(\sigma) = |w|_\sigma$.

Par exemple, avec l'alphabet $\Sigma = \{a, b, c, d, e, f, g, h\}$, le mot *bbbbbbbbbbbbehhhhhhhhhh* admet comme tableau de comptage le quadruplet (v, F, I, A)

$v = 3$ et

	a	b	c	d	e	f	g	h
F	$*$	12	$*$	$*$	1	$*$	$*$	9
I	b	e	h	$*$	$*$	$*$	$*$	$*$
A	$*$	a	$*$	$*$	b	$*$	$*$	c

où chaque symbole $*$ représente une valeur particulière mais non significative.

Indication OCaml : En OCaml, on pourra se servir du type

```
type creux = int * int array * unicode array * unicode array;;
```

On pourra utiliser une fonction `make_creux`, de type `int -> creux` qui crée et renvoie un tableau creux de comptage du mot vide en temps constant, aucune hypothèse ne pouvant être faite sur le contenu des trois tableaux constituant la valeur de retour de cette fonction.

☐ 14 – Soit (v, F, I, A) un tableau creux de comptage du mot $w \in \Sigma^*$. Montrer que pour toute lettre $\tau \in \Sigma$ avec $\tau \leqslant \sigma_{v-1}$, la lettre $I(\tau)$ est une lettre présente dans le mot w.

☐ 15 – Soit (v, F, I, A) un tableau creux de comptage du mot $w \in \Sigma^*$. Montrer que pour toute lettre $\sigma \in \Sigma$, la lettre σ est présente dans le mot w si et seulement si on a $A(\sigma) \leqslant \sigma_{v-1}$ et $I(A(\sigma)) = \sigma$.

☐ 16 – Écrire une fonction `est_present` de type `creux -> unicode -> bool` telle que la valeur de retour de `tableau_creux theta u` est vraie si et seulement si la lettre σ_u est une lettre présente dans un mot de tableau creux de comptage θ. Justifier la correction et la terminaison de la fonction.

☐ 17 – Écrire une fonction `incremente_tableaucreux`, de type `creux -> unicode -> creux`, telle que la valeur de retour de `incremente_tableaucreux theta u` est un tableau creux de comptage d'un mot qui contient la lettre σ_u une fois de plus qu'un mot de tableau creux de comptage θ.

☐ 18 – Écrire une fonction `cree_tableaucreux` de type `texte -> creux` qui renvoie le tableau creux de comptage du texte donné en argument.
Le recours à la fonction `fonction1` de la question 1 n'est pas autorisé pour répondre à cette question.

☐ 19 – Déterminer la complexité en temps de la fonction `cree_tableaucreux`.

☐ 20 – Donner sans justification la complexité en espace de la fonction `cree_tableaucreux`.

☐ 21 – Est-il réaliste d'utiliser cette fonction pour le texte d'un courriel de la vie courante rédigé en langue française encodé avec l'alphabet Unicode lorsqu'on utilise un ordinateur ou un téléphone actuel ?

2 Un encodeur optimal

2.1 Codes et codages

Définitions : Un *code* c est une application

$$c : \Sigma \to \{0,1\}^*.$$

Le *codage* associé à un code c est l'application

$$e : \begin{cases} \Sigma^* & \to & \{0,1\}^* \\ \tau_1\tau_2\dots\tau_n & \mapsto & c(\tau_1)c(\tau_2)\cdots c(\tau_n) \end{cases}$$

Un *arbre de code* représentant un code c est un arbre binaire, dont l'ensemble des sommets est Σ, tel que

— pour toute lettre $\sigma \in \Sigma$, l'étiquette du sommet σ est $c(\sigma)$,

— pour toutes lettres σ et τ de l'alphabet Σ, si τ appartient au sous-arbre gauche de σ, alors on a $\tau < \sigma$ dans l'ordre de l'alphabet Σ ; si τ appartient au sous-arbre droit de σ, alors on a $\tau > \sigma$ dans l'ordre de l'alphabet Σ.

Indication OCaml : On adopte les déclarations de type suivantes afin de représenter les mots binaires et les arbres de code en OCaml :

```
type binaire = bool list;;
type code =
   | Nil
   | Noeud of unicode * binaire * code * code;;
```

☐ 22 – Écrire une fonction `encodeur`, de type `texte -> code -> bool list`, qui, à partir d'un texte t et d'un code c, renvoie le codage du mot du texte t par le codage associé au code c.

On note $\mathsf{prof}_{\mathcal{A}}(\sigma)$ la profondeur d'un sommet σ dans un arbre $\mathcal{A}$, c'est-à-dire le nombre de sommets de l'unique chemin entre le sommet σ et la racine de $\mathcal{A}$. En particulier, la racine ρ d'un arbre $\mathcal{A}$ est de profondeur $\mathsf{prof}_{\mathcal{A}}(\rho) = 1$.

☐ 23 – Pour un code c donné, exprimer la complexité en temps de la fonction `encodeur` à l'aide de $L = \max_{\sigma\in\Sigma} |c(\sigma)|$, des profondeurs des sommets de l'arbre de code $\mathcal{A}$ et de la distribution des fréquences dans le mot w donnés en argument

2.2 Arbre de code optimal

On fixe un code c de l'alphabet Σ ainsi que des poids réels positifs sous la forme d'une application $f : \Sigma \to \mathbb{R}_+$. La *profondeur pondérée* d'un arbre de code $\mathcal{A}$ représentant le code c est la somme

$$\mathsf{Prof}(\mathcal{A}) = \sum_{\sigma\in\Sigma} f(\sigma)\ \mathsf{prof}_{\mathcal{A}}(\sigma).$$

☐ 24 – Exprimer la profondeur pondérée $\mathsf{Prof}(\mathcal{A})$ d'un arbre de code $\mathcal{A}$ en fonction de f, de la profondeur pondérée $\mathsf{Prof}(\mathcal{A}_g)$ du sous-arbre gauche $\mathcal{A}_g$ de l'arbre de code $\mathcal{A}$ et de la profondeur pondérée $\mathsf{Prof}(\mathcal{A}_d)$ du sous-arbre droit $\mathcal{A}_d$ de l'arbre de code $\mathcal{A}$.

On dit qu'un arbre de code est *optimal* si sa profondeur pondérée est la plus petite des profondeurs pondérées des arbres de code représentant un code de l'alphabet Σ. Pour tous entiers u et v compris entre 0 et $\lambda - 1$ avec $u \leqslant v$, on note $c_{u,v}$ la restriction du code c à l'alphabet $\Sigma_{u,v} = \{\sigma_u, \sigma_{u+1}, \ldots, \sigma_{v-1}, \sigma_v\}$. On note $\Pi_{u,v}$ la profondeur pondérée d'un arbre de code représentant le code $c_{u,v}$ optimal.

☐ 25 – Pour tout entier u compris entre 0 et $\lambda - 1$, calculer $\Pi_{u,u}$.
Pour tous entiers u et v compris entre 0 et $\lambda - 1$ vérifiant $u < v$, exprimer une relation entre $\Pi_{u,v}$ et les quantités $(\Pi_{u,r})_{u\leqslant r\leqslant v-1}$ et $(\Pi_{r,v})_{u+1\leqslant r\leqslant v}$.

☐ 26 – Concevoir et présenter un algorithme qui reçoive en entrée un code $c : \Sigma \to \{0,1\}^+$ et une distribution de fréquences $f : \Sigma \to \mathbb{N}$ sous la forme de tableaux et qui construise un arbre de code optimal en temps $O(\lambda^3)$. Il n'est pas exigé d'utiliser la syntaxe OCaml pour répondre à cette question. On décomposera l'algorithme en sous-programmes élémentaires dont on caractérisera les entrées, les sorties ou les effets suffisamment pour que l'établissement d'une preuve de correction de l'algorithme soit facile. Cette preuve de correction n'est pas exigée.
On justifiera la complexité en temps.

2.3 Un arbre de code optimal calculé plus rapidement

Pour tous entiers u et v avec $0 \leqslant u \leqslant v \leqslant \lambda - 1$, on appelle $r_{u,v}$ le plus grand entier r inférieur à $\lambda - 1$ tel qu'il existe un arbre optimal pour le code $c_{u,v}$ ayant la lettre σ_r comme racine. On se propose de démontrer, pour tout n compris entre 0 et $\lambda - 2$, l'encadrement

$$\forall\, 0 \leqslant u \leqslant v \leqslant \lambda - 2 \text{ avec } v - u \leqslant n, \qquad r_{u,v} \leqslant r_{u,v+1} \leqslant r_{u+1,v+1}. \tag{$\mathcal{E}(n)$}$$

☐ 27 – Déduire de l'encadrement $\mathcal{E}(\lambda - 2)$ une modification de l'algorithme proposé en réponse à la question 26 afin que son temps d'exécution soit $O(\lambda^2)$. Détailler le calcul de la complexité en temps.

Soient u et v deux entiers avec $1 \leqslant u \leqslant v \leqslant \lambda - 2$.

☐ 28 – Montrer que, dans un arbre de code qui représente le code $c_{u,v}$, le sommet qui contient la lettre σ_v ne possède jamais de fils droit.

☐ 29 – Dans cette question, on suppose que le poids $f(\sigma_{v+1})$ est nul. Soit $\mathcal{A}$ un arbre de code optimal pour le code $c_{u,v}$ et $\mathcal{A}'$ l'arbre obtenu en ajoutant à $\mathcal{A}$ le sommet σ_{v+1} comme fils droit du sommet qui contient la lettre σ_v. Montrer que $\mathcal{A}'$ est un arbre de code optimal pour le code $c_{u,v+1}$.

Dans les trois questions qui suivent, on suppose que les poids $(f(\sigma_x))_{u \leqslant x \leqslant v}$ restent constants tandis que le poids $f(\sigma_{v+1})$ est variable.

☐ 30 – Montrer que l'application

$$\pi_{u,v+1} : \begin{cases} \mathbb{R}_+ & \to & \mathbb{R}_+ \\ f(\sigma_{v+1}) & \mapsto & \Pi_{u,v+1} \end{cases}$$

est une application continue et affine par morceaux sur $\mathbb{R}_+$ dont on précisera la pente pour chaque morceau et que, pour tout intervalle ouvert $I \subseteq \mathbb{R}_+$ sur lesquels l'application $\pi_{u,v+1}$ est affine, les indices $(r_{u',v'})_{u \leqslant u' \leqslant v' \leqslant v+1}$ sont constants lorsque $f(\sigma_{v+1})$ varie dans I.

Soit $\alpha \in \mathbb{R}_+^\star$ un point de changement de pente de l'application $\pi_{u,v+1}$, soit $\beta^- < \alpha$ un réel tel que l'application $\pi_{u,v+1}$ soit affine sur l'intervalle $I^- =]\beta^-, \alpha[$ et soit $\beta^+ > \alpha$ un réel tel que l'application $\pi_{u,v+1}$ soit affine sur l'intervalle $I^+ =]\alpha, \beta^+[$.

On construit deux suites (d_k^-) et (d_k^+) par récurrence. Chaque suite s'arrête si le terme suivant n'est plus défini. Pour construire la première suite, on choisit $f(\sigma_{v+1})$ dans l'intervalle I^- et on pose

$$\begin{cases} & d_0^- &=& r_{u,v+1} \\ \text{si } d_k^- < v+1, & d_{k+1}^- &=& r_{d_k^-+1,v+1} \end{cases}.$$

Pour construire la seconde suite, on choisit $f(\sigma_{v+1})$ dans l'intervalle I^+ et on pose

$$\begin{cases} & d_0^+ &=& r_{u,v+1} \\ \text{si } d_k^+ < v+1, & d_{k+1}^+ &=& r_{d_k^++1,v+1} \end{cases}.$$

Dans la mesure où les entiers $r_{u,v}$ dépendent de la valeur de $f(\sigma_{v+1})$, les deux suites (d_k^-) et (d_k^+) ne sont pas forcément égales et ne dépendent que du fait que $f(\sigma_{v+1})$ appartient à I^- ou à I^+.

☐ 31 – Montrer que les suites (d_k^-) et (d_k^+) sont finies et que, en appelant m^- l'indice du dernier terme défini de la suite (d_k^-) et m^+ l'indice du dernier terme défini de la suite (d_k^+), on a $m^- > m^+$.

On suppose avoir démontré la validité de l'encadrement $\mathcal{E}(n)$ pour un certain entier n. On suppose que les entiers u et v vérifient $v - u \leqslant n + 1$.

☐ 32 – En faisant l'hypothèse $d_0^+ < d_0^-$, montrer qu'il existe un indice $s < m^+$ tel qu'on ait $d_s^- = d_s^+$ et tel que, pour tout entier ℓ compris entre 0 et $s-1$, on ait $d_\ell^+ < d_\ell^-$. En déduire une contradiction en échangeant les descendants du sommet $\sigma_{d_s^-}$ et du sommet $\sigma_{d_s^+}$ dans certains arbres.

☐ 33 – Montrer que l'encadrement $(\mathcal{E}(n))$ est satisfait pour tous entiers n avec $n \leqslant \lambda - 2$.

Fin de l'épreuve