A 2004 – INF – MP

ÉCOLE NATIONALE DES PONTS ET CHAUSSÉES,
ÉCOLES NATIONALES SUPÉRIEURES DE L'AÉRONAUTIQUE ET DE L'ESPACE,
DES TECHNIQUES AVANCÉES, DES TÉLÉCOMMUNICATIONS,
DES MINES DE PARIS, DES MINES DE SAINT-ÉTIENNE, DES MINES DE NANCY
DES TÉLÉCOMMUNICATIONS DE BRETAGNE,
ÉCOLE POLYTECHNIQUE
(Filière T.S.I.)

CONCOURS D'ADMISSION 2004

ÉPREUVE D'INFORMATIQUE

Filière MP
(Durée de l'épreuve : 3 heures)

Sujet mis à la disposition des concours Cycle International, ENSTIM et TPE-EIVP.

Les candidats et les candidates sont priés de mentionner de façon apparente sur la première page de la copie : « INFORMATIQUE – Filière MP »

RECOMMANDATIONS AUX CANDIDATS ET CANDIDATES

- **L'énoncé de cette épreuve, y compris cette page de garde, comporte 10 pages.**
- **Si, au cours de l'épreuve, un candidat ou une candidate repère ce qui lui semble être une erreur d'énoncé, il ou elle le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il ou elle a décidé de prendre.**
- **Tout résultat fourni dans l'énoncé peut être utilisé pour les questions ultérieures même s'il n'a pas été démontré.**
- **Il ne faut pas hésiter à formuler les commentaires qui semblent pertinents même lorsque l'énoncé ne le demande pas explicitement.**
- **L'utilisation d'une calculatrice ou d'un ordinateur est interdite.**

COMPOSITION DE L'ÉPREUVE

L'épreuve comporte deux parties indépendantes :

- un problème d'algorithmique et de programmation, page 2 à 7, à résoudre en 1 h 45 mn environ ;
- un problème sur les automates, pages 8 à 10, à résoudre en 1 h 15 mn environ.

1. Problème d'algorithmique et de programmation – 1 h 45 mn environ

Le tri par baquets

Préliminaire concernant la programmation : il faudra écrire des fonctions ou des procédures à l'aide d'un langage de programmation qui pourra être soit **Caml**, soit **Pascal**, tout autre langage étant exclu. **Indiquer en début de problème le langage de programmation choisi ; il est interdit de modifier ce choix au cours de l'épreuve**. Certaines questions du problème sont formulées différemment selon le langage de programmation ; cela est indiqué chaque fois que cela est nécessaire. Par ailleurs, lorsqu'un candidat ou une candidate écrira une fonction ou une procédure en langage de programmation, il ou elle précisera si nécessaire le rôle des variables locales et pourra définir des fonctions ou procédures auxiliaires qu'il ou elle explicitera. Enfin, lorsque le candidat ou la candidate écrira une fonction ou une procédure, il ou elle pourra, lorsque cela s'y prête, faire appel à une autre fonction ou procédure définie dans les questions précédentes.

Terminologie, notations et indications

a) Dans l'énoncé, les identificateurs sont écrits en police de caractères `Courier New` dans le contexte du langage de programmation et en *italique* sinon.

b) Un tableau est constitué de *cases* pouvant contenir des valeurs d'un type donné. Le nombre total de cases du tableau s'appelle ici sa *dimension*. La case d'indice i d'un tableau T sera notée $T[i]$ dans l'énoncé du problème.

c) Dans ce problème, nous appelons *opérations élémentaires* :

- un accès en mémoire pour lire ou écrire la valeur d'une variable ou d'une case d'un tableau ;
- une opération arithmétique entre entiers : addition, soustraction, multiplication, division entière, calcul du reste dans une division entière ;
- une comparaison ($>, <, \geq, \leq, =, \neq$) entre deux entiers.

Soient f et g deux fonctions d'une même variable entière n (resp. de deux mêmes variables entières n et m). On dit que la fonction f a un ordre de grandeur au plus égal à celui de la fonction g s'il existe un entier strictement positif k et un entier N (resp. deux entiers N et M) tels qu'on ait, pour tout $n \geq N$ (resp. $n \geq N$ et $m \geq M$), $f(n) \leq k\, g(n)$ (resp. $f(n, m) \leq k\, g(n, m)$). On dit que les deux fonctions ont même ordre de grandeur si l'ordre de grandeur de l'une est au moins égal à l'ordre de grandeur de l'autre, et réciproquement. Par exemple, les fonctions $f(n) = 3\,n^2 - 5\,n + 4$ et $g(n) = n^2$ ont même ordre de grandeur. On pourra aussi dire que g est un ordre de grandeur de f.
Quand on calculera la complexité d'un algorithme $\mathcal{A}$, on l'exprimera sous la forme d'un ordre de grandeur du nombre d'opérations élémentaires effectuées pendant le déroulement de $\mathcal{A}$; cette complexité sera exprimée à l'aide de paramètres caractéristiques du problème à traiter.

d) Si l'on se préoccupe de trier une liste de n données, on sait alors que les algorithmes de tri **opérant par comparaisons et échanges des données de la liste** ont une complexité dans le cas le plus défavorable au moins de l'ordre de grandeur de $n\ln(n)$. On étudie dans ce problème des algorithmes de tri d'entiers qui n'opèrent pas par comparaisons entre les données à trier.

e) Tout au long du problème, on fera l'hypothèse qu'il n'y a pas de limitation sur la taille de la mémoire disponible et qu'on peut donc manipuler des tableaux de dimensions quelconques. On ne se préoccupera pas non plus du fait que les entiers codés sur un ordinateur sont bornés et on fera comme s'ils ne l'étaient pas.

f) **Indications pour la programmation**
Caml : Ce qui est appelé tableau dans l'énoncé correspond à ce qui est appelé *vecteur* en Caml. En Caml, `T.(i)` représente la case $T[i]$ d'un tableau T.

Pascal : Dans tout le problème, on supposera qu'on écrit les différentes fonctions dans un fichier contenant les définitions suivantes :

```
const
    MAX_DIM = 100;
    MAX_VAL = 1000;

type
    TABLE = array[0..MAX_DIM] of INTEGER;
    BACS = array[0..9] of TABLE;
```

PREMIÈRE PARTIE
Tri d'entiers compris entre 0 et *MAX_VAL*

Cette partie est consacrée à un algorithme qui peut être considéré comme la version la plus simple du tri par baquets étudié dans la seconde partie.

On suppose dans cette première partie qu'on veut trier par ordre croissant un ensemble de n entiers dont les valeurs sont toutes comprises entre 0 et une valeur maximum notée *MAX_VAL*. Certaines valeurs peuvent figurer plusieurs fois dans l'ensemble des données à trier ; ces valeurs figureront alors avec le même nombre d'occurrences après le tri.

Exemple :
Pour *MAX_VAL* = 10 et n = 9, avec les données à trier suivantes : 6, 4, 2, 8, 4, 2, 3, 6, 4, les données triées sont alors : 2, 2, 3, 4, 4, 4, 6, 6, 8.

¬ 1 – Expliciter un algorithme de tri, de complexité $MAX\_VAL + n$, fondé sur le principe suivant : on compte, pour chaque entier compris entre 0 à *MAX_VAL*, son nombre d'occurrences parmi les entiers à trier, puis on en déduit la liste triée. On justifiera sommairement la complexité de l'algorithme proposé.

¬ 2 – Il s'agit de programmer l'algorithme de la question ¬ 1. Les données à trier se trouvent initialement dans un tableau ; après le tri, les données doivent occuper globalement les mêmes cases du tableau mais en étant cette fois-ci ordonnées par ordre croissant.

Caml : Écrire en Caml une fonction `tri_simple` telle que, si
- `tableau` est un vecteur de dimension suffisante,
- `tableau.(0)` contient le nombre de données à trier,
- les données à trier, de valeurs comprises entre 0 et `MAX_VAL`, se trouvent dans `tableau` consécutivement à partir de l'indice 1,

alors `tri_simple tableau` ordonne les données du vecteur `tableau` par ordre croissant. La constante entière `MAX_VAL` est supposée déjà définie.

Pascal : Écrire en Pascal une procédure `tri_simple` telle que si
- `tableau` est de type `TABLE`,
- `tableau[0]` contient le nombre de données à trier, nombre qui ne dépasse pas la constante `MAX_DIM`,
- les données à trier, de valeurs comprises entre 0 et `MAX_VAL`, se trouvent dans `tableau` consécutivement à partir de l'indice 1,

alors `tri_simple(tableau)` ordonne par ordre croissant les données contenues dans `tableau`.

DEUXIÈME PARTIE
Gestion de tables

Dans toute cette partie, on considérera des tableaux **dont le plus petit indice vaut 0** et tels que les données du problème n'occupent pas nécessairement toutes les cases. Plus précisément, pour un tableau *T* contenant *n* données significatives du problème (typiquement, des données à trier) :

- la case *T*[0] contiendra le nombre *n* ;
- les données significatives occuperont les cases *T*[1], *T*[2], …, *T*[*n*].

Par exemple, on pourrait, pour trier les entiers 6, 1 et 7, disposer d'un tableau *T* dont les indices varient de 0 à 8, mettre la valeur 3 (nombre de données à trier) dans *T*[0], dans *T*[1] la valeur 6, dans *T*[2] la valeur 1, dans *T*[3] la valeur 7 et ne pas se préoccuper des contenus des cases d'indice 4 à 8. Ce tableau peut être représenté ainsi :

T :

3	6	1	7					

On appellera *table* un tableau conçu selon ce modèle. Une table contiendra toujours **des entiers positifs ou nuls**. Ainsi, pour l'exemple précédent, la table *T* contient les données 6, 1 et 7. Une table *T* est dite *vide* si *T*[0] vaut 0. Une table sera dite *triée* si les entiers contenus dans les cases d'indices compris entre 1 et *T*[0] sont croissants au sens large.

Les questions ⌙3 à ⌙10 préparent la programmation du tri par baquets dont le principe sera indiqué après la question ⌙10.

⌙3 – Vider une table consiste à la transformer en une table vide.

Caml : Écrire en Caml une fonction `vider` telle que, si `T` est un vecteur contenant une table, `vider T` vide la table `T`.

Pascal : Écrire en Pascal une procédure `vider` telle que, si `T` est de type `TABLE`, `vider(T)` vide la table `T`.

⌙4 – *Ajouter un entier p* à une table consiste à ajouter *p* à la suite des données figurant déjà dans la table et à actualiser le nombre des données de la table. Par exemple, si la table *T* est réprésentée ci-dessous (les cases non remplies contiennent des valeurs non significatives) :

T :

3	6	1	7				

ajouter à la table *T* la valeur 5 consiste à transformer *T* en :

T :

4	6	1	7	5			

Caml : Écrire en Caml une fonction `ajouter` telle que, si `T` est un vecteur contenant une table et `p` un entier, `ajouter T p` transforme `T` pour ajouter `p` à la table `T`. On supposera que la dimension de la table *T* permet cet ajout.

Pascal : Écrire en Pascal une procédure `ajouter` telle que, si `T` est de type `TABLE` et `p` un entier, `ajouter(T, p)` ajoute l'entier `p` à la table `T`. On supposera que la dimension de la table *T* permet cet ajout.

⌙5 – *Concaténer une table T1 et une table T2* consiste à ajouter successivement dans *T1* toutes les données de *T2*. Par exemple, si les tables *T1* et *T2* sont réprésentées ci-dessous :

T1 :

3	6	1	7				

T2 :

2	12	4					

la table *T1* devient après concaténation :

T1 :

5	6	1	7	12	4		

La table *T2* est inchangée.

Caml : Écrire en Caml une fonction `concatener` telle que, si `T1` et `T2` sont deux vecteurs contenant des tables, `concatener T1 T2` concatène les tables `T1` et `T2`. On supposera que la dimension de `T1` est suffisante pour cette concaténation.

Pascal : Écrire en Pascal une procédure `concatener` telle que, si `T1` et `T2` sont de type `TABLE`, `concatener(T1, T2)` concatène les tables `T1` et `T2`. On supposera que la dimension de `T1` est suffisante pour cette concaténation.

→ 6 – Indiquer la complexité de la fonction ou de la procédure `concatener` en l'exprimant à l'aide des nombres de données contenues par les tables `T1` et `T2`. On justifiera sommairement le résultat.

→ 7 – Il s'agit ici de définir une fonction *max_valeurs* qui détermine la plus grande valeur d'une table *T* (c'est-à-dire le plus grand des entiers qui se trouvent entre les indices 1 et *T*[0]).

Caml : Écrire en Caml une fonction `max_valeurs` telle que, si `T` est un vecteur contenant une table, `max_valeurs T` renvoie la valeur du plus grand entier de `T`. La fonction `max_valeurs` renverra la valeur −1 si la table est vide.

Pascal : Écrire en Pascal une fonction `max_valeurs` telle que, si `T` est de type `TABLE`, `max_valeurs(T)` renvoie la valeur du plus grand entier de `T`. La fonction `max_valeurs` renverra la valeur −1 si la table est vide.

→ 8 – Il s'agit de déterminer le nombre de chiffres d'un entier positif ou nul donné écrit dans la base 10 ; par exemple, le nombre de chiffres de 5973 vaut 4.

Caml : Écrire en Caml une fonction `nombre_chiffres` telle que, si `p` est un entier positif ou nul, `nombre_chiffres p` renvoie le nombre de chiffres de l'entier `p`.

Pascal : Écrire en Pascal une fonction `nombre_chiffres` telle que, si `p` est un entier positif ou nul, `nombre_chiffres(p)` renvoie le nombre de chiffres de l'entier `p`.

→ 9 – Il s'agit de définir une fonction *max_chiffres* qui calcule le nombre maximum de chiffres dans une écriture décimale des entiers contenus dans une table. Par exemple, si la table est :

T :	5	13	2408	1	97	892		

le résultat doit valoir 4.

Caml : Écrire en Caml une fonction `max_chiffres` telle que, si `T` est un vecteur contenant une table, `max_chiffres T` renvoie le nombre maximum de chiffres des données de la table `T`. La fonction renverra la valeur 0 si la table est vide.

Pascal : Écrire en Pascal une fonction `max_chiffres` telle que, si `T` est de type `TABLE`, `max_chiffres(T)` renvoie le nombre maximum de chiffres des données de la table `T`. La fonction renverra la valeur 0 si la table est vide.

→ 10 – Exprimer la complexité de la fonction *max_chiffres* appliquée à une table *T* à l'aide :

- du nombre *n* de données de la table *T*
- du nombre maximum *maxc* de chiffres des données de la table *T*.

On justifiera sommairement le résultat.

TROISIÈME PARTIE
Tri d'entiers écrits en base 10 à l'aide de baquets

Dans la description de l'algorithme du tri par baquets, on appelle *chiffre de rang r* ($r \geq 1$) d'un entier positif ou nul p le r-ième chiffre en partant de la **droite** de l'entier p écrit dans la base 10 ; par définition, si r est supérieur au nombre total de chiffres de p, le chiffre de rang r vaut alors 0. Par exemple, pour l'entier $p = 597$, le chiffre de rang 1 est 7, celui de rang 2 est 9, celui de rang 3 est 5, et le chiffre de rang i pour $i \geq 4$ est 0.
Les données sont dans une table T. L'objectif de l'algorithme est de transformer la table T en une table triée de manière croissante contenant les mêmes données. On dispose par ailleurs d'un tableau appelé *baquets*, dont les indices varient de 0 à 9, de 10 tables ; ces 10 tables peuvent contenir chacune au moins autant de données que le nombre de données contenues par la table T. L'algorithme est le suivant :

- Calculer le nombre maximum de chiffres des données de la table T ; le noter *maxc*.
- Pour r qui varie de 1 à *maxc*, faire :
  a) Vider les dix baquets, c'est-à-dire vider *baquets*[i] pour tout i appartenant à {0, 1, …, 9} ;
  b) considérer successivement, de l'indice 1 à l'indice T[0], les entiers de la table T : en notant p l'entier considéré, déterminer le chiffre de rang r de l'entier p ; en notant k ce chiffre, ajouter p à la table *baquets*[k] ;
  c) vider T ;
  d) pour i variant de 0 à 9, concaténer T et *baquets*[i] (le résultat de la concaténation se trouve dans T).

Indications : on pourra utiliser dans la suite du problème une fonction nommée *puiss10*, supposée déjà définie, qui calcule pour un entier p positif ou nul la valeur de 10^p. Dans les calculs de complexité, on considérera que les appels à cette fonction sont négligeables, ce qui signifie qu'on ne comptabilisera pas les opérations correspondantes. Lors de l'écriture d'une fonction ou d'une procédure en langage de programmation, si `p` est une variable entière positive ou nulle, on pourra obtenir 10^p par `puiss10(p)`.

→ 11 – On applique l'algorithme du tri par baquets à la table T ci-dessous, qui contient 8 données à trier.

T :	8	57	423	50	603	7	20	453	27	…

Le nombre maximum de chiffres des données de la table vaut 3.
Pendant l'itération correspondant à la valeur 1 de r :
- à l'issue du point b), les baquets d'indices 1, 2, 4, 5, 6, 8, 9 sont restés vides, *baquets*[0], *baquets*[3] et *baquets*[7] sont représentés ci-dessous :

baquets[0] :	2	50	20	…

baquets[3] :	3	423	603	453	…

baquets[7] :	3	57	7	27	…

- à l'issue du point d), la table T est :

T :	8	50	20	423	603	453	57	7	27	…

Détailler de même les itérations correspondant aux valeurs 2 et 3 de r.

→ 12 – Montrer qu'après l'exécution de l'algorithme du tri par baquets, la table T est triée.

→ 13 – Il s'agit d'écrire en langage de programmation une fonction ou une procédure nommée *distribuer* qui effectue le point b) de l'algorithme du tri par baquets.

Caml : écrire en Caml une fonction `distribuer` telle que si :
- `T` est un vecteur contenant une table,
- `r` est un entier au moins égal à 1,
- `baquets` est un vecteur de dimension 10 de vecteurs d'entiers ; chacun de ces 10 vecteurs contient une table vide et est supposé de dimension suffisante pour contenir les données qu'on voudra y mettre,

alors `distribuer T r baquets` effectue les opérations indiquées dans le point b) de l'algorithme du tri par baquets.

Pascal : écrire en Pascal une procédure `distribuer` telle que si :

- `T`, de type `Table`, contient une table,
- `r` est un entier au moins égal à 1,
- `baquets` est de type `BACS` (voir au début du problème) ; chacune des dix tables (de type `TABLE`) de `baquets` contient une table vide supposée de dimension suffisante pour contenir les données qu'on voudra y mettre,

alors `distribuer(T, r, baquets)` effectue les opérations indiquées dans le point b) de l'algorithme du tri par baquets.

→ 14 – Il s'agit d'écrire en langage de programmation une fonction ou une procédure nommée *tri_baquets* qui effectue le tri par baquets.

Caml : écrire en Caml une fonction `tri_baquets` telle que si `T` est un vecteur contenant une table, `tri_baquets T` transforme la table *T* en une table triée en utilisant l'algorithme du tri par baquets.

Pascal : écrire en Pascal une procédure `tri_baquets` telle que si `T`, de type `Table`, contient une table, `tri_baquets(T)` transforme la table *T* en une table triée en utilisant l'algorithme du tri par baquets.

→ 15 – On note *n* le nombre de données à trier et *maxc* le nombre maximum de chiffres des données de la table. Montrer que la complexité de l'algorithme du tri par baquets est de l'ordre de $maxc \times n$.

→ 16 – En supposant les données à trier toutes distinctes, donner une fonction exprimée uniquement à l'aide du nombre *n* de données à trier telle que :

- son ordre de grandeur minore la complexité du tri par baquets ;
- son ordre de grandeur peut être atteint.

→ 17 – Il s'agit dans cette question de modifier le tri par baquets pour que la complexité devienne de l'ordre de la somme du nombre des chiffres des données de la table. Cette nouvelle version du tri par baquets devra commencer directement par la mise dans les baquets sans nécessiter le calcul préalable du nombre maximum de chiffres des données ni l'utilisation d'un autre algorithme préparatoire. Pour définir un tel algorithme, donner son principe, justifier rapidement son exactitude et sa complexité, puis écrire ou réécrire en langage de programmation les fonctions ou procédures ajoutées ou modifiées ; on sera sans doute conduit à récrire `distribuer` et `tri_baquet` en les nommant `distribuer_bis` et `tri_baquet_bis`.

FIN DU PROBLÈME D'ALGORITHMIQUE ET DE PROGRAMMATION

2. Problème sur les automates – 1 h 15 mn environ

Longueur discriminante de deux automates

Deux automates $\mathcal{A}$ et $\mathcal{A}'$ sont équivalents s'ils reconnaissent le même langage. S'ils ne sont pas équivalents, alors il existe des mots qui sont reconnus par l'un et pas par l'autre. La longueur minimum des mots qui ont cette propriété est dite *discriminante*. L'objet de ce problème est d'évaluer, par deux méthodes, un majorant de la longueur discriminante de $\mathcal{A}$ et $\mathcal{A}'$ en fonction des nombres d'états de $\mathcal{A}$ et $\mathcal{A}'$.

Notations et terminologie

Un *alphabet* Σ est un ensemble fini d'éléments appelés *lettres*. Un *mot* sur Σ est une suite finie de lettres de Σ ; le mot vide est noté ε. On désigne par Σ^* l'ensemble des mots sur Σ, y compris le mot vide. La *longueur* d'un mot m, notée $|m|$, est le nombre de lettres qui le composent. Un *langage* est une partie de Σ^*.

Un *automate* $\mathcal{A}$ est décrit par une structure $<\Sigma, Q, T, I, F>$, où :

- Σ est un alphabet ;
- Q est un ensemble fini et non vide appelé *ensemble des états* de $\mathcal{A}$;
- $T \subseteq Q \times \Sigma \times Q$ est appelé l'ensemble des transitions ; étant donnée une transition $(p, a, q) \in T$, on dit qu'elle va de l'état p à l'état q et qu'elle est d'étiquette a ; on pourra la noter $p \xrightarrow{a} q$;
- $I \subseteq Q$ est appelé ensemble des *états initiaux* de $\mathcal{A}$;
- $F \subseteq Q$ est appelé ensemble des *états finals* de $\mathcal{A}$.

On représente graphiquement l'automate $\mathcal{A}$ ainsi :

- un état p est figuré par un cercle marqué en son centre par p ; si p appartient à I, cela est figuré par une flèche entrante sans origine ; si un état q appartient à F, cela est figuré par une flèche sortante sans but ;
- une transition $(p, a, q) \in T$ est figurée par une flèche allant de l'état p vers l'état q et étiquetée par la lettre a.

Un calcul c de $\mathcal{A}$ est un chemin de la forme $p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \ldots \xrightarrow{a_k} p_k$, avec $p_{i-1} \xrightarrow{a_i} p_i \in T$ pour $1 \leq i \leq k$; p_0 est l'*origine* du calcul, p_k son *extrémité*. L'*étiquette* de c est le mot formé par la suite des étiquettes des transitions successives du chemin.

Un calcul de $\mathcal{A}$ d'origine p, d'extrémité q et d'étiquette m est dit *réussi* si on a $p \in I$ et $q \in F$. Un mot $m \in \Sigma^*$ est *reconnu* par $\mathcal{A}$ s'il est l'étiquette d'un calcul réussi. Le *langage reconnu* par $\mathcal{A}$, noté $L(\mathcal{A})$, est l'ensemble des mots reconnus par $\mathcal{A}$. Deux automates $\mathcal{A}$ et $\mathcal{A}'$ sont dits *équivalents* si on a $L(\mathcal{A}) = L(\mathcal{A}')$.

L'automate $\mathcal{A}$ est dit *déterministe* si I ne contient qu'un élément et si, pour tout $(p, a) \in Q \times \Sigma$, il existe au plus un état $q \in Q$ avec $(p, a, q) \in T$. L'automate $\mathcal{A}$ est dit *complet* si et seulement si, pour tout $p \in Q$ et tout $a \in \Sigma$, il existe $q \in Q$ avec $(p, a, q) \in T$.

On rappelle que tout automate ayant n états est équivalent à un automate déterministe complet ayant au plus 2^n états.

PREMIÈRE PARTIE
Approche naïve

→ 18 – Soit $\mathcal{A}$ un automate, déterministe ou non déterministe, avec n états. Montrer que $L(\mathcal{A})$ est vide si et seulement s'il ne contient aucun mot de longueur inférieure ou égale à $n - 1$.

→ 19 – Soit $\mathcal{A}$ un automate déterministe complet ayant n états. Donner un automate ayant aussi n états qui reconnaît $\overline{L(\mathcal{A})}$, le complémentaire dans Σ^* de $L(\mathcal{A})$. Justifier votre réponse.

→ 20 – Soient $\mathcal{A}$ et $\mathcal{A}'$ deux automates utilisant le même alphabet Σ avec respectivement n et n' états. Donner un automate ayant $n \times n'$ états qui reconnaît $L(\mathcal{A}) \cap L(\mathcal{A}')$. Justifier votre réponse.

→ 21 – Soient $\mathcal{A}$ et $\mathcal{A}'$ deux automates déterministes complets utilisant le même alphabet Σ avec respectivement n et n' états. Montrer que si $\mathcal{A}$ et $\mathcal{A}'$ ne sont pas équivalents, il existe un mot de longueur au plus $n \times n' - 1$ qui est reconnu par l'un et non par l'autre, *i.e.* la longueur discriminante de $\mathcal{A}$ et $\mathcal{A}'$ est inférieure ou égale à $n \times n' - 1$.

→ 22 – En déduire un majorant pour la longueur discriminante de deux automates non équivalents quelconques avec n et n' états respectivement.

SECONDE PARTIE
Approche plus fine

L'objectif de cette partie est de démontrer que la longueur discriminante de deux automates déterministes non équivalents, avec n et n' états respectivement, est inférieure ou égale à $n + n' - 1$.

→ 23 – Montrer sur un exemple, avec un alphabet à une seule lettre, qu'il existe des automates déterministes $\mathcal{A}$ et $\mathcal{A}'$ non équivalents et qui reconnaissent les mêmes mots de longueur strictement inférieure à $n + n' - 1$, où n (resp. n') désigne le nombre d'états de $\mathcal{A}$ (resp. de $\mathcal{A}'$).

On introduit un ensemble de définitions et de notations.
Soit $\mathcal{A} = <\Sigma, Q, T, I, F>$ un automate déterministe avec n états. On identifie l'ensemble Q des états de $\mathcal{A}$ avec l'ensemble $\{1, 2, \ldots, n\}$ des n premiers entiers naturels non nuls de sorte que chaque état est identifié par un entier compris entre 1 et n. **On suppose que l'on a $I = \{1\}$**.
On note $\{e_1, e_2, \ldots, e_n\}$ la base canonique de l'espace vectoriel $\mathbf{R}^n$. Pour un entier i compris entre 1 et n, le vecteur e_i est donc le vecteur de $\mathbf{R}^n$ dont toutes les composantes sont nulles sauf la i-ième qui vaut 1. On note 0 le vecteur nul de $\mathbf{R}^n$.

Pour chaque lettre a de l'alphabet Σ, on définit une application linéaire φ_a de $\mathbf{R}^n$ dans $\mathbf{R}^n$ par :

pour tout i avec $1 \le i \le n$:
- s'il existe j, $1 \le j \le n$, tel que $(i, a, j) \in T$, alors $\varphi_a(e_i) = e_j$,
- sinon $\varphi_a(e_i) = 0$.

L'existence et l'unicité de l'application φ_a découlent du fait que $\mathcal{A}$ est déterministe et de la linéarité de φ_a.

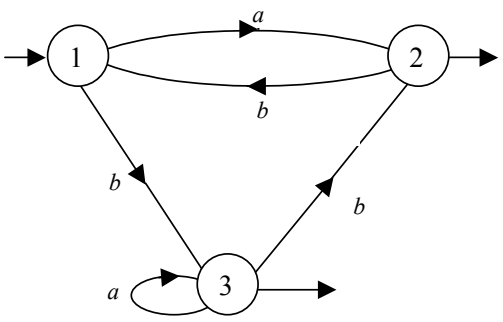
Si $m = a_1 a_2 \ldots a_{k-1} a_k$ est un mot non vide de Σ^*, où $a_1, a_2, \ldots, a_{k-1}, a_k$ sont des lettres de Σ, on pose :

$$\varphi_m = \varphi_{a_k} \circ \varphi_{a_{k-1}} \circ \cdots \circ \varphi_{a_2} \circ \varphi_{a_1}$$

On note φ_ε l'identité de $\mathbf{R}^n$ dans $\mathbf{R}^n$.
On appelle z la somme des vecteurs e_i quand i décrit l'ensemble F des états finals : $z = \sum_{i \in F} e_i$.

Enfin, si u et v sont deux vecteurs de $\mathbf{R}^n$, on note $u.v$ le produit scalaire de u et de v : $u.v = \sum_{i=1}^{n} u_i v_i$.

Exemple :



$\varphi_a(e_1) = e_2$, $\varphi_a(e_2) = 0$, $\varphi_a(e_3) = e_3$
$\varphi_b(e_1) = e_3$, $\varphi_b(e_2) = e_1$, $\varphi_b(e_3) = e_2$
$z = (0, 1, 1)$.

On peut aussi constater les égalités suivantes :
$\varphi_{abb}(e_1) = e_3$, $\varphi_{ba}(e_3) = 0$, $\varphi_{ba}(e_2) = e_2$.

→ 24 – Soient $m \in \Sigma^*$ et deux entiers i et j vérifiant $1 \leq i \leq n$ et $1 \leq j \leq n$. Montrer qu'il existe un calcul d'origine i, d'extrémité j et d'étiquette m si et seulement si on a $\varphi_m(e_i) = e_j$.

→ 25 – Soit $m \in \Sigma^*$. Donner les valeurs possibles du produit scalaire $\varphi_m(e_1).z$ et indiquer une condition nécessaire et suffisante portant sur ce produit scalaire pour que m soit un mot reconnu par $\mathcal{A}$.

Dans la suite du problème, on considère en plus de l'automate déterministe $\mathcal{A}$ un automate déterministe $\mathcal{A}'$. Les notations utilisées pour $\mathcal{A}'$ sont les mêmes que celles utilisées pour $\mathcal{A}$ à cela près que tous les identificateurs sont dotés d'un « prime » (on a donc : n', φ'_a, φ'_m, z' et les vecteurs de la base canonique de $\mathbf{R}^{n'}$ sont notés e'_i).
Si u et v sont des vecteurs respectivement de $\mathbf{R}^n$ et de $\mathbf{R}^{n'}$, on note $(u\ ;\ v)$ le vecteur w de $\mathbf{R}^{n+n'}$ obtenu en concaténant u et v ; plus précisément, le vecteur w est défini par :

$$\begin{cases} \text{si } 1 \leq i \leq n, & w_i = u_i \\ \text{si } n+1 \leq i \leq n+n', & w_i = v_{i-n} \end{cases}$$

Par exemple, si $u = (0, 1, 0)$ et $v = (1, 2)$, $(u\ ;\ v)$ est le vecteur $(0, 1, 0, 1, 2)$.
On pose :

$E = (e_1; -e'_1)$ (bien noter le signe –)
$Z = (z; z')$.

Pour tout mot m de Σ^*, on définit une application linéaire Φ_m de $\mathbf{R}^{n+n'}$ dans $\mathbf{R}^{n+n'}$ par :

pour tout $u \in \mathbf{R}^n$ et pour tout $u' \in \mathbf{R}^{n'}$, $\Phi_m((u\ ;\ u')) = \left(\varphi_m(u)\ ;\ \varphi'_m(u')\right)$

(on admet la linéarité de Φ_m).

→ 26 – Soit $m \in \Sigma^*$; indiquer les valeurs possibles du produit scalaire $\Phi_m(E).Z$ et, selon ces valeurs, préciser l'appartenance de m à $L(\mathcal{A})$ et $L(\mathcal{A}')$.

Pour $k \in \mathbf{N}$, on note V_k le sous-espace vectoriel de $\mathbf{R}^{n+n'}$ engendré par $\{\Phi_m(E) \mid m \in \Sigma^*, |m| \leq k\}$; V_0 est donc engendré par le vecteur $(e_1\ ;\ -e'_1)$.

→ 27 – Monter que, pour $k \geq 0$, on a $V_k \subseteq V_{k+1}$.

→ 28 – Soit $m \in \Sigma^*$; on suppose que m s'écrit sous la forme : $m = \mu a$, où $\mu \in \Sigma^*$ et $a \in \Sigma$. Montrer l'égalité $\Phi_m = \Phi_a \circ \Phi_\mu$.

→ 29 – Soit $w \in V_k$ et $a \in \Sigma$; montrer : $\Phi_a(w) \in V_{k+1}$.

→ 30 – On suppose qu'il existe $k \geq 0$ tel que $V_k = V_{k+1}$; montrer l'égalité $V_{k+2} = V_{k+1}$.

→ 31 – Montrer qu'il existe un entier $h \leq n + n' - 1$ tel que, pour tout $k \geq h$, $V_k = V_h$.

→ 32 – Montrer que si $\mathcal{A}$ et $\mathcal{A}'$ ne sont pas équivalents, il existe un mot de longueur inférieure ou égale à $n + n' - 1$ qui est accepté par l'un et pas par l'autre, *i.e.* que $n + n' - 1$ est un majorant de la longueur discriminante de $\mathcal{A}$ et $\mathcal{A}'$.

→ 33 – En déduire un majorant pour la longueur discriminante de deux automates quelconques ayant n et n' états respectivement.

FIN DU PROBLÈME SUR LES AUTOMATES