Informatique - MPI

Exercice A1

Soit Σ un alphabet. u et v sont deux mots de Σ^*. L est un langage sur Σ.

Question 1. Le *langage quotient de v par u* est défini par : $u^{-1}v = \{w \in \Sigma^* \mid uw = v\}$. Montrer que le langage $u^{-1}v$ possèdre au plus un mot.

Question 2. Le *langage quotient de L par u* est défini par : $u^{-1}L = \{w \in \Sigma^* \mid uw \in L\}$. Si $\Sigma = \{a, b\}$ et si L est le langage des mots qui contiennent au moins une fois la lettre a, calculer les langages $\varepsilon^{-1}L$, $a^{-1}L$ et $b^{-1}L$.

Question 3. Montrer que $(uv)^{-1}L = v^{-1}u^{-1}L$.

Question 4. Montrer que si L est un langage reconnaissable alors le langage $u^{-1}L$ est reconnaissable.

Question 5. On note $\text{Quot}(L)$ la famille des langages définie par : $\text{Quot}(L) = \{u^{-1}L \mid u \in \Sigma^*\}$. On suppose que L est un langage reconnu par l'automate déterministe $\mathcal{A} = (Q, q_0, T, \delta)$ tel qu'il existe au moins un chemin de l'état initial q_0 à n'importe quel autre état q de Q. On note alors L_q le langage défini par :

$$L_q = \{w \in \Sigma^* \mid \delta^*(q, w) \in T\}$$

où δ^* est le prolongement naturel de δ sur $Q \times \Sigma^*$. Montrer que $\text{Quot}(L) = \{L_q \mid q \in L\}$.

Exercice A2

On considère les langages suivants définis sur l'alphabet $\Sigma = \{a, b\}$.

$$L_1 = \{a^n b^m \mid n \bmod 2 = 0 \text{ et } m \bmod 2 = 0\}$$
$$L_2 = \{a^n b^m \mid n \bmod 2 = 0, m \bmod 2 = 0 \text{ et } n \geqslant m\}$$

Question 1. Pour chacune des affirmations suivantes, dire sans justifier si elle est vraie ou fausse.
☐ **1.1.** $aaabab \in L_1$
☐ **1.2.** $aaabab \in L_2$
☐ **1.3.** $L_1 \subset L_2$
☐ **1.4.** L_1 est fini.

Question 2. Le langage L_2 est-il rationnel ? Justifier.

Question 3. On considère l'automate de la figure 1. Expliquer brièvement pourquoi il reconnaît L_1. Le déterminiser et le compléter. On note $\mathcal{A}_1$ l'automate résultant de ces opérations.

Question 4. À l'aide de l'algorithme de Berry-Sethi, construire un automate $\mathcal{A}_L$ reconnaissant le langage L dénoté par l'expression régulière $(aa)^*(bb)^*$. On détaillera les étapes.

Question 5. Construire l'automate produit entre $\mathcal{A}_L$ et un automate bien choisi reconnaissant le complémentaire de L_1. Quel est le langage reconnu par l'automate ainsi construit ? En déduire que $L \subset L1$.

Question 6. En utilisant les questions précédentes, justifier que L_1 est dénoté par $(aa)^*(bb)^*$.

Exercice A3

Un voyageur navigue dans un archipel à la recherche de l'île Maya. Les îles de cet archipel sont occupées par d'étranges habitants : les Pires, qui mentent systématiquement, et les Purs, qui disent toujours la vérité. Sur chaque île, le voyageur rencontre deux habitants, A et B qui acceptent de lui parler.

Question 1. Sur une première île, A et B tiennent les propos suivants.

- A : *B est un Pur et nous sommes sur l'île Maya.*
- B : *A est un Pire et nous sommes sur l'île Maya.*

☐ **1.1.** Modéliser la situation à l'aide d'une formule φ_1 du calcul propositionnel.
☐ **1.2.** Mettre φ_1 sous forme normale conjonctive (FNC) sans utiliser de table de vérité.
☐ **1.3.** Appliquer l'algorithme de Quine à la FNC obtenue à la question précédente et en déduire si le voyageur est arrivé sur l'île Maya et, si possible, la nature de A et B.

Question 2. Sur une deuxième île, A et B tiennent les propos suivants.

- A : *Nous sommes deux Pires et nous sommes sur l'île Maya.*
- B : *L'un de nous au moins est un Pire et nous ne sommes pas sur l'île Maya.*

□ **2.1.** Modéliser la situation à l'aide d'une formule φ_2 du calcul propositionnel.
□ **2.2.** Etablir sa table de vérité.
□ **2.3.** En déduire une formule équivalente à φ_2 sous forme normale disjonctive.
□ **2.4.** Déterminer si le voyageur est arrivé sur l'île Maya.

Exercice A4

On considère la suite définie par ses deux premiers termes $F_0 = F_1 = 1$ et, pour tout entier naturel n supérieur à 2, par la relation de récurrence $F_n = F_{n-1} + F_{n-2}$.

Question 1.
□ **1.1.** Donner un algorithme récursif naïf permettant de calculer le n-ième terme de cette suite.
□ **1.2.** On note N_n le nombre d'appels à cet algorithme sur l'entier n. Établir une relation entre N_n et F_n.
□ **1.3.** En déduire la complexité de cet algorithme. Commenter.

Question 2. Donner un algorithme de complexité linéaire en n qui calcule F_n.

Question 3. On pose $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ et, pour tout entier naturel n, $X_n = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$. Justifier l'existence d'un algorithme qui calcule F_n en temps $O(\log n)$.

Exercice B1

Cet exercice est à traiter dans le langage OCaml.

On considère un graphe non orienté $G = (V, E)$. Une *k-coloration* de G est une application $c : V \to [\![0, k-1]\!]$ telle que $(a, b) \in E \implies c(a) \neq c(b)$. Le problème de la coloration d'un graphe consiste à trouver une k-coloration de G de valeur de k minimale.

Les sommets d'un graphe G d'ordre n représentés désignés par les entiers $0, 1, \dots, n-1$ et G est représenté par les listes d'adjacence de ses sommets. On définit les types suivants.

```
1 type voisinage = int list ;;
2 type graphe = voisinage array ;;
```

Une k-coloration c est représentée par un tableau `couleur` de type `int array` tel que pour tout entier i de $\{0, \dots, n-1\}$, `couleur.(i)` $= c(i)$.

Question 1. Écrire une fonction `coloration_valide : graphe -> int array -> bool` qui prend en argument un graphe G, un tableau `couleur` et qui renvoie `true` si couleur est une coloration de G, `false` sinon. Évaluer sa complexité en fonction de $n = |V|$ et $p = |E|$.

Question 2. Un graphe qui possède une 2-coloration est dit *biparti* : il existe une partition de V en deux ensembles A et B telle que toute arête de E relie un sommet de A à un sommet de B.
Écrire une fonction `biparti : graphe -> int array` qui prend en argument un graphe supposé biparti et renvoie une 2-coloration de ce dernier. Évaluer sa complexité en fonction de n et p.

Question 3. En dehors du problème de la 2-coloration, tous les algorithmes connus garantissant une coloration optimale sont de complexité exponentielle. On s'intéresse alors à des solutions gloutonnes qui, à défaut de garantir une solution minimale, fournissent des colorations acceptables. L'*algorithme glouton* parcourt les sommets par ordre croissant des indices, en attribuant à chaque sommet la plus petite couleur disponible (c'est-à-dire non déjà donnée à un de ses voisins). Écrire la fonction `glouton : graphe -> int array` correspondante. Cette fonction renvoie-t-elle une coloration optimale pour le graphe biparti de la figure 2 ?

Question 4. À un ensemble fini d'intervalles $[a_i, b_i[$, $0 \leqslant i \leqslant n-1$, on associe un graphe $G = (V, E)$ pour lequel les sommets de V sont les intervalles et dont les arêtes relient les couples d'intervalles dont l'intersection est non vide. Un tel graphe est appelé *graphe d'intervalles*. La figure 3 illustre cette définition.

□ **4.1.** Montrer que pour un graphe d'intervalles, l'algorithme glouton fournit une coloration minimale lorsque les sommets sont ordonnés par valeurs de a_i croissantes.

□ **4.2.** Montrer que pour un graphe G quelconque il existe toujours un ordonnancement des sommets pour lequel l'algorithme glouton fournit un résultat optimal.

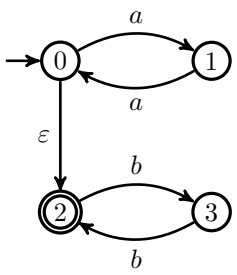


Figure 1

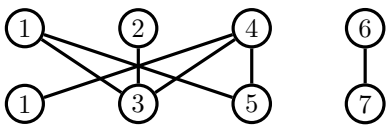
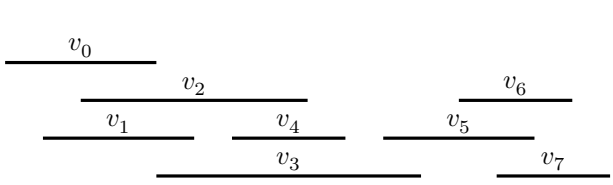


Figure 2 – Exemple de graphe biparti.



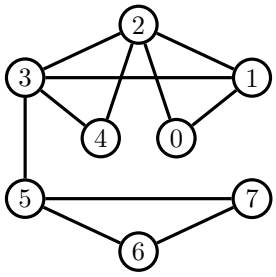


Figure 3 – Exemple de graphe d'intervalles.

Exercice B2

Cet exercice est à traiter dans le langage C.

On considère des arbres binaires non étiquetés. Un arbre est *binaire strict* si tout nœud interne a exactement deux enfants, ou autrement dit si tout nœud (interne ou non) a zéro ou deux enfants. Un arbre est *parfait* s'il est binaire strict et que toutes ses feuilles sont à même profondeur. Il est *presque-parfait* si tous les niveaux de profondeur sont remplis par des nœuds, sauf éventuellement le dernier, rempli par la gauche.

Question 1.

□ **1.1.** Combien existe-t-il d'arbres presque-parfaits de taille 6 ? Les représenter graphiquement.

□ **1.2.** Citer une structure de données qui peut s'implémenter avec des arbres presque-parfaits.

□ **1.3.** Proposer une définition par induction d'un arbre parfait de hauteur h, pour $h \in \mathbb{N}$. Justifier rigoureusement que cette définition coïncide avec celle de l'énoncé.

Question 2. Un arbre binaire est implémenté en C par le type `arbre` suivant.

```
1 struct Noeud {
2   struct Noeud* gauche;
3   struct Noeud* droite;
4 };
5 typedef struct Noeud arbre;
```

Ainsi, un arbre contient un pointeur vers son fils gauche et son fils droit.

□ **2.1.** Écrire une fonction `int hauteur(arbre* a)` qui prend en argument un pointeur vers un arbre, puis calcule et renvoie sa hauteur.

□ **2.2.** En déduire une `fonction bool est_parfait(arbre* a)` qui prend en argument un pointeur vers un arbre et renvoie un booléen qui vaut `true` si et seulement si l'arbre pointé par `a` est parfait. Quelle est sa complexité temporelle pour un arbre `a` effectivement parfait ?

□ **2.3.** Écrire une fonction `arbre* plus_grand_presque_parfait(arbre* a)` qui prend en argument un pointeur vers un arbre `a` et renvoie un pointeur vers le plus grand sous-arbre de `a` qui est presque-parfait. La complexité de cette fonction doit être linéaire en la taille de l'arbre `a`.

Indication : que peut-on dire des enfants gauche et droit d'un arbre presque-parfait de hauteur h ?

Exercice B3

Cet exercice est à traiter dans le langage OCaml. La fin du sujet fournit un code fournissant le type décrit par l'énoncé ainsi qu'une partie des fonctions décrites ci-après. Il est à compléter avec les fonctions demandées.

On note $|t|$ la taille d'un arbre. Un *arbre de Braun* est un arbre binaire qui est :

- soit l'arbre vide, noté $\perp$;
- soit de la forme $N(r, g, d)$ avec r une étiquette, g et d deux arbres de Braun vérifiant $|d| \leqslant |g| \leqslant |d| + 1$

Dans ce sujet, on se limite au cas où les étiquettes des arbres de Braun sont des entiers. De tels arbres sont représentés par le type suivant.

```
1 type braun = E | N of int * braun * braun
```

Question 1. Pour tout $n \in [\![1, 6]\!]$, dessiner les squelettes des arbres de Braun de taille n. Que remarque-t-on ? On admet dans la suite (et on pourra l'utiliser) que cette propriété est vraie pour tout $n \in \mathbb{N}$.

On admet que la hauteur h et la taille n d'un arbre de Braun vérifient $h = \Theta(\log n)$.

Question 2. Écrire une fonction `hauteur : braun -> int` calculant la hauteur d'un arbre de Braun. On demande une complexité logarihtmique en la taille de l'arbre en entrée, qu'on justifiera brièvement. Si vous ne parvenez pas à atteindre cette complexité, proposez néanmoins une fonction.

Un *tas de Braun* est un arbre de Braun vérifiant de surcroît que l'étiquette d'un nœud est toujours inférieure ou égale à celles de ses enfants éventuels. Dans la suite, on implémente des fonctions sur les tas de Braun de façon à les utiliser pour implémenter une file de priorité.

Question 3. Écrire une fonction `minimum : braun -> int` renvoyant l'élément minimal d'un tas de Braun. Dans le cas où l'arbre est vide, on renverra `max_int`.

Question 4. Écrire une fonction `inserer : braun -> int -> braun` qui insère un élément dans un tas de Braun. Attention, il faut bien sûr que le résultat soit un tas de Braun et en particulier un arbre de Braun.

On suppose que l'on dispose d'une fonction `fusionner : braun -> braun -> braun` prenant en entrée deux tas de Braun t et t' tels que $|t'| \leqslant |t| \leqslant |t'| + 1$ et qui renvoie un tas de Braun contenant les éléments de t et de t'.

Question 5. Écrire une fonction `extraire_min : braun -> int * braun` prenant en entrée un tas de Braun `t` et qui renvoie un couple `(m, t')` avec `m` le minimum de `t` et `t'` un tas de Braun contenant les étiquettes de `t` moins une occurrence de `m`.

Question 6. On cherche à présent à compléter la fonction fusionner fournie dans le code compagnon. Écrire une fonction `extraire_element : braun -> int * braun` prenant en entrée un tas de Braun `t` non vide et renvoyant `(x, t')` tels que `x` est un élément quelconque de `t` et `t'` est un tas de Braun contenant les éléments de `t` sauf une occurrence de `x`.

Question 7. Écrire une fonction `remplacer_min : braun -> int -> braun` prenant en entrée un tas de Braun `t`, un entier `x` et renvoyant un tas de Braun contenant les éléments de `t`, moins une occurrence du minimum de `t`, plus une occurrence de `x`.

Question 8. Justifier brièvement la correction de la fonction fusionner fournie par l'énoncé.

```
1  (*Type décrit par l'énoncé*)
2  type braun = E | N of int * braun * braun
3
4  (*Un exemple de tas de Braun à 4 éléments*)
5  let t = N(1,N(3,N(7,E,E),E),N(4,E,E))
6
7  (**
8  Entrée : deux tas de Braun t et t' tels que |t'| <= |t| <= |t'| + 1.
9  Sortie : un tas de Braun contenant l'union des éléments contenus dans t et t'.
       *)
10 (***ATTENTION : on ne peut pas utiliser cette fonction
11 avant d'avoir écrit extraire_element et remplacer_min***)
12 let rec fusionner (g:braun) (d:braun) :braun = match g, d with
13     |E, E | N(_,E,E), E -> g
14     |N(rg, gg, dg), N(rd, gd, dd) ->
15       if rg <= rd
16       then N(rg, d, fusionner gg dg)
17       else let x, g' = extraire_element g in N(rd, remplacer_min d x, g')
18     |_ -> failwith "conditions sur les arbres non respectées dans fusionner"
```

Exercice B4

On s'intéresse à un bus touristique pouvant contenir n_p passagers. On suppose que $n > n_p$ passagers alternent indéfiniment entre attendre de pouvoir monter dans le bus, faire leur tour touristique, puis se remettre en attente à l'arrêt de bus. Pour formaliser ce problème, on considère des fonctions fictives :

- `board` et `unboard` sont les fonctions permettant à un passager de monter et descendre ;
- `load` et `unload` sont les fonctions permettant au bus de charger et décharger ses passagers, et `run` est la fonction qui consiste à lancer le tour touristique.

On fait les hypothèses suivantes.

- Le bus ne démarre son tour que lorsqu'il est plein.
- Les passagers doivent attendre l'exécution de `load` pour monter et de `unload` pour descendre du bus.
- Le chargement du bus ne peut commencer qu'une fois que le déchargement est terminé (et qu'il n'y a plus de passagers dans le bus).
- Deux passagers ne peuvent pas monter (resp. descendre) simultanément dans le bus.

Question 1. Écrire des fonctions en pseudo-code correspondant aux actions du bus et d'un passager, sans prendre en compte les problèmes de synchronisation.

Question 2. Détailler les fonctions `load`, `unload`, `board` et `unboard` en prenant en compte la synchronisation. On pourra utiliser des compteurs, des mutex et des sémaphores.

Question 3. Cette solution peut-elle être utilisée dans le cas où il y a plusieurs bus ? Justifier.

Question 4. On souhaite gérer le cas où il y a m bus de capacité n_p, numérotés de 0 à $m-1$, en respectant les règles suivantes.

- Un seul bus peut ouvrir ses portes aux passagers à la fois (pour charger ou décharger).
- Un bus ne peut pas en doubler un autre.

On suppose que $m \times n_p \leqslant n$.
□ **4.1.** Que peut-il se passer si on n'impose pas la contrainte $m \times n_p \leqslant n$?
□ **4.2.** Proposer une solution. On pourra utiliser des tableaux de sémaphores.

Exercice B5

Cet exercice est à traiter dans le langage OCaml. Si nécessaire, on notera c_{mod} le coût temporel de l'appel à la fonction mod *de OCaml (fonction modulo).*

Question 1.
□ **1.1.** Écrire une fonction *récursive non terminale* `dernier_chiffre : int -> int` qui reçoit un entier naturel n et qui renvoie la valeur du dernier chiffre dans la représentation en base 10 de 2^n.
□ **1.2.** Prouver votre solution. Analyser sa complexité.

Question 2.
□ **2.1.** Reprendre la question en écrivant une fonction *récursive terminale.*
□ **2.2.** Prouver votre solution. Analyser sa complexité.

Question 3.
□ **3.1.** Pour $n \geqslant 1$, quelle observation peut-on faire sur les derniers chiffres de 2^n ?
□ **3.2.** En déduire une fonction plus efficace que les précédentes. Quelle est sa complexité ?