# Solution du devoir 1

**Auteur : Alexandre Blondin Massé**

| Question | 1 | 2 | 3 | 4 | Total |
|---|---|---|---|---|---|
| Sur | 20 | 20 | 30 | 30 | 100 |
| Note | | | | | |

1. (20 points) Pour chacune des expressions suivantes, indiquez si elle est vraie ou fausse. Dans tous les cas, justifiez votre réponse.
   (a) (5 points) $\log(n!) \in \Omega(n \log n)$;
   (b) (5 points) $n3^{2n} \in \omega(9^{n-4})$;
   (c) (5 points) $\sum_{i=1}^{n} i^2 \in \Theta(n^3)$;
   (d) (5 points) $\log^* n \in \mathcal{O}(1)$, où $\log^*$ est le logarithme itéré.

**Solution:** (a) C'est vrai. On a

$$\begin{aligned}
\log(n!) &= \log\left(\prod_{i=1}^{n} i\right) \\
&= \sum_{i=1}^{n} \log(i) \\
&\geq \sum_{i=\lfloor n/2 \rfloor}^{n} \log(i) \\
&\geq \sum_{i=\lfloor n/2 \rfloor}^{n} \log(\lfloor n/2 \rfloor) \\
&= \log(\lfloor n/2 \rfloor)(n - \lfloor n/2 \rfloor + 1) \\
&\in \Omega(n \log n).
\end{aligned}$$

(b) C'est vrai. D'une part,

$$n3^{2n} = n(3^2)^n = n9^n = \Theta(n9^n)$$

et d'autre part,

$$9^{n-4} = \frac{9^n}{9^4} = \Theta(9^n).$$

Or,

$$\lim_{n\to+\infty} \frac{n9^n}{9^n} = \lim_{n\to+\infty} \frac{n}{1} = +\infty,$$

ce qui entraîne $n3^{2n} \in \omega(9^{n-4})$.

(c) C'est vrai. Nous avons

$$\begin{aligned} \sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \\ &= \frac{2n^3+3n^2+n}{6} \\ &\in \Theta(n^3). \end{aligned}$$

(d) C'est faux. Supposons au contraire qu'il existe des constantes positives $C, k$ telle que $\log^*(n) \leq C$ pour $n \geq k$. Soit $D = \lceil C \rceil$ et

$$m = \max(k, 2 \uparrow (D+1)),$$

où $\uparrow$ est définie récursivement par

$$a \uparrow b = \begin{cases} 1, & \text{si } b = 0; \\ a^{a\uparrow(b-1)}, & \text{si } b \geq 1. \end{cases}$$

pour tout réel positif $a$ et pour tout entier positif $b$. Alors on aurait que

$$\begin{aligned} \log^*(m) &\geq \log^*(2 \uparrow (D+1)) \\ &= D+1 \\ &> D, \end{aligned}$$

malgré que $m \geq k$, ce qui contredit l'hypothèse.

2. (20 points) Donnez un algorithme qui affiche les sommets d'un graphe orienté $G$ rencontrés lors d'un parcours en profondeur, en respectant les contraintes suivantes :

   - Vous n'avez pas le droit d'utiliser la récursivité.
   - Vous devez utiliser la structure de données Pile, qui respecte la spécification suivante vue en classe :

| Type | Pile |
|---|---|
| Utilise | Element, Booleen |
| Opérations | PILEVIDE : $\emptyset \to$ Pile<br>ESTVIDE : Pile $\to$ Booleen<br>EMPILER : Pile $\times$ Element $\to$ Pile<br>DEPILER : Pile $\to$ Pile<br>TETE : Pile $\to$ Element |

- Vous devez utiliser la structure de données Graphe, qui respecte la spécification suivante :

| Type | Graphe |
|---|---|
| Utilise | Sommet, Ensemble |
| Opérations | SOMMETS : Graphe $\to$ Ensemble<br>SUCCESSEURS : Graphe $\times$ Sommet $\to$ Ensemble |

- Vous devez utiliser la structure de données Association, qui respecte la spécification suivante :

| Type | Association |
|---|---|
| Utilise | Sommet, Element |
| Opérations | ASSOCIATIONVIDE : $\emptyset \to$ Association<br>SET : Association $\times$ Sommet $\times$ Element $\to$ Association<br>GET : Association $\times$ Sommet $\to$ Element |

  et qui permet de marquer des sommets avec des valeurs quelconques (booléens, entiers, chaînes de caractères, etc.). Vous pouvez supposer que chacune des opérations s'effectuent en temps constant.

- Vous pouvez supposer que l'instruction $u$.AFFICHER() affiche le contenu d'un sommet $u \in$ Sommet.
- Votre algorithme doit être linéaire au pire cas en temps et en espace.
- Le graphe $G$ n'est pas nécessairement connexe.

Indiquez les complexités temporelle et spatiale de votre algorithme en utilisant la notation asymptotique.

**Solution:** La contrainte de ne pas utiliser la récursivité et celle d'utiliser une pile suggèrent d'effectuer un parcours en profondeur de façon non récursive. L'idée consiste, lors du parcours en profondeur, à marquer les sommets visités de sorte qu'on s'assure de ne pas tourner en rond.

Voici ce qu'on obtient :

```
1: procedure ParcoursLargeur(G : graphe orienté)
2:     V ← associationVide()
3:     P ← pileVide()
```

```
 4:    pour u ∈ G.SOMMETS faire
 5:        V.SET(u, faux)
 6:        P.EMPILER(u)
 7:    fin pour
 8:    tant que ¬P.ESTVIDE() faire
 9:        u ← P.TETE()
10:        P.DEPILER()
11:        si ¬V.GET(u) alors
12:            u.AFFICHER()
13:            V.SET(u, vrai)
14:            pour v ∈ G.SUCCESSEURS(u) faire
15:                P.EMPILER(v)
16:            fin pour
17:        fin si
18:    fin tant que
19: fin procedure
```

Soient $n$ et $m$ le nombre de sommets et le nombre d'arêtes de $G$ respectivement.

Analysons d'abord la complexité spatiale. Les seules structures de données auxiliaires utilisées sont une pile et une association. La pile contient des sommets et chaque sommet $u$ peut apparaître plusieurs fois, mais jamais plus que $\deg(u)$. Quant à l'association, elle occupe un espace de $\Theta(n)$, puisque chaque sommet est marqué une seule fois comme visité ou non. Par conséquent, la complexité spatiale est

$$\sum_{u \in V} \Theta(\deg(u)) + \Theta(n) = \Theta(2m) + \Theta(n) = \Theta(n+m),$$

par le lemme des poignées de mains.

Pour la complexité temporelle, on remarque que toutes les opérations s'effectuent en temps constant et donc il suffit d'analyse le nombre de tours de boucle effectués. L'initialisation (lignes 4–7) s'effectue en $\Theta(n)$, puisqu'on parcourt chaque sommet. La boucle principale (lignes 8–18) est répétée tant qu'il y a des sommets dans la pile. Comme chaque sommet $u$ apparaît au plus $\deg(u)$ fois au total, on obtient $\Theta(m)$ tours de boucle, par le lemme des poignées de mains. Finalement, on a que la boucle 14–16 est répétée au total $\deg(u)$ pour chaque sommet $u$, ce qui entraîne une complexité temporelle totale de $\Theta(n+m)$.

3. (30 points) Dans cette question, nous nous intéressons à deux structures de données, nommées Arbre et Foret, qui implémentent respectivement une structure d'*arborescence* (arbre enraciné) et de *forêt enracinée*. Elles sont définies de récursivement, l'une par rapport à l'autre, de la façon suivante :

| Type | Arbre |
|---|---|
| Utilise | Naturel, Booleen, Element, Foret |
| Opérations | ARBRE : Element × Foret → Arbre<br>RACINE : Arbre → Element<br>ENFANTS : Arbre → Foret<br>NBNOEUDS : Arbre → Naturel<br>NBFEUILLES : Arbre → Naturel |

| Type | Foret |
|---|---|
| Utilise | Naturel, Booleen, Element, Arbre |
| Opérations | FORETVIDE : $\emptyset$ → Foret<br>ESTVIDE : Foret → Booleen<br>AJOUTERARBRE : Foret × Arbre → Foret<br>NBARBRES : Foret → Naturel<br>ARBRE : Foret × Naturel → Arbre<br>NBNOEUDS : Foret → Naturel<br>NBFEUILLES : Foret → Naturel |

Donnez les préconditions et les axiomes qui sont cohérents avec la sémantique suivante :

(1) Les enfants d'un arbre forment une forêt.

(2) Une feuille est un arbre dont les enfants sont une forêt vide.

(3) Une forêt est une liste ordonnée d'arbres enracinés, indicée de 0 à $k-1$, où $k$ est le nombre d'arbres dans la forêt.

(4) Lorsqu'on ajoute un arbre à une forêt, il est inséré en dernière position.

Votre ensemble d'axiomes doit être complet et non redondant.

**Solution:** La seule précondition est que ARBRE$(F, i)$ est définie si et seulement si $0 \leq i <$ NBARBRES$(F)$, pour toute $F \in$ Foret et $i \in$ Naturel.

Les axiomes sont les suivants. Pour $F \in$ Foret, $A \in$ Arbre, $e \in$ Element et $i \in$ Naturel, on a :

1. RACINE(ARBRE$(e, F)) = e$
2. ENFANTS(ARBRE$(e, F)) = F$
3. NBNOEUDS(ARBRE$(e, F)) = 1 +$ NBNOEUDS$(F)$
4. NBFEUILLES(ARBRE$(e, F)) = \begin{cases} 1, & \text{si ESTVIDE}(F); \\ \text{NBFEUILLES}(F), & \text{sinon.} \end{cases}$
5. ESTVIDE(FORETVIDE)

6. $\neg$estVide(ajouterArbre$(F, A)$)
7. nbArbres(foretVide) $= 0$
8. nbArbres(ajouterArbre$(F, A)$) $= 1 +$ nbArbres$(F)$
9. arbre(ajouterArbre$(F, A), i) = \begin{cases} A, & \text{si } i = \text{nbArbres}(F) - 1; \\ \text{arbre}(F, i), & \text{sinon.} \end{cases}$
10. nbNoeuds(foretVide) $= 0$
11. nbNoeuds(ajouterArbre$(F, A)$) $=$ nbNoeuds$(A) +$ nbNoeuds$(F)$
12. nbFeuilles(foretVide) $= 0$
13. nbFeuilles(ajouterArbre$(F, A)$) $=$ nbFeuilles$(A) +$ nbFeuilles$(F)$

4. Pour chacun des ensembles suivants $E$, calculez sa cardinalité $n$, donnez une fonction de hachage $h : E \to \{0, 1, 2, \ldots, n-1\}$ qui est parfaite et implémentez la fonction de hachage $h$ à l'aide de votre langage préféré parmi Python, Java, C et C++. En particulier, montrez que votre implémentation est correcte en l'illustrant sur plusieurs exemples.
   (a) (15 points) $E$ est l'ensemble des points à coordonnées entières dans le rectangle induit par les points $(0, 0)$ et $(a, b)$, où $a, b \geq 1$ sont des entiers donnés, c'est-à-dire
   $$E = \{(x, y) \mid 0 \leq x \leq a \wedge 0 \leq y \leq b\}.$$
   (b) (15 points) $E$ est l'ensemble des palindromes de longueur $n$ sur l'alphabet $\{0, 1\}$, où $n \geq 0$ est un nombre entier donné.

**Solution:** Dans les deux cas, il s'agit simplement de proposer une bijection simple à implémenter de l'ensemble $E$ vers $\{0, 1, 2, \ldots, n-1\}$.

(a) Il suffit d'ordonner les couples du rectangle. Il y a plusieurs façons de faire, par exemple en prenant l'ordre lexicographique

$$(0, 0), (0, 1), (0, 2), \ldots, (1, 0), (1, 1), \ldots (a, b).$$

La règle de correspondance pour cet ordre est simplement

$$h(x, y) = xb + y.$$

(b) Encore une fois, il y a plusieurs solutions possibles. On constate tout d'abord qu'un palindrome est complètement déterminé par sa première ou sa deuxième moitié,

c'est-à-dire que pour tout entier $n \geq 0$, il y a une bijection entre les palindromes de longueur $n$ et les mots binaires de longueur $\lceil n/2 \rceil$. Ensuite, chaque mot binaire peut être transformé en nombre naturel (bref, le mot binaire est la représentation binaire du nombre!). Par conséquent, une solution possible, en utilisant l'ordre lexicographique sur la deuxième moitié, est :

$$h(w) = \begin{cases} 0, & \text{si } w = \varepsilon \text{ ou } w = 0; \\ 1, & \text{si } w = 1; \\ 2h(u) + a, & \text{si } w = aua, \text{ où } a \text{ est une lettre.} \end{cases}$$

Voici une implémentation des deux fonctions en Python :

```
from itertools import product

# ------------- #
# Question 4(a) #
# ------------- #

def rectange(a, b):
    r"""
    Retourne l'ensemble des points a coordonnees entieres dans le
    rectangle de coins (0,0) et (a,b).
    """
    return set((x,y) for x in range(a) for y in range(b))

def ha(a, b):
    r"""
    Retourne une fonction de hachage parfaite pour rectangle(a, b)
    """
    return lambda x, y: x * b + y

def tester_ha():
    r"""
    Verifie si ``ha`` est une fonction de hachage parfaite
    """
    for (a, b) in [(2, 7), (4, 6)]:
        h = ha(a, b)
        print 'Test de la fonction h pour a = %s et b = %s' % (a, b)
        print '========================================='
        for x in range(a):
            for y in range(b):
                print 'h(%s, %s) = %s' % (x, y, h(x, y))
        print

tester_ha()

# ------------- #
# Question 4(b) #
# ------------- #
```

```
def mots(alphabet, n):
    r"""
    Genere l'ensemble des mots de longueur n sur l'alphabet donne
    """
    assert n >= 0
    for w in product(alphabet, repeat=n):
        yield ''.join(w)

def palindromes(alphabet, n):
    r"""
    Genere l'ensemble des palindromes de longueur n sur l'alphabet
    donne.
    """
    assert n >= 0
    for w in mots(alphabet, (n + 1) // 2):
        suffix = ''.join(reversed(w))
        if n % 2 == 1:
            suffix = suffix[1:]
        yield w + suffix

def hb(alphabet):
    r"""
    Retourne une fonction de hachage parfaite pour les palindromes
    binaires.
    """
    assert len(alphabet) == 2
    def h(w):
        if w == '' or w == alphabet[0]:
            return 0
        elif w == alphabet[1]:
            return 1
        else:
            a = 0 if w[0] == alphabet[0] else 1
            return 2 * h(w[1:-1]) + a
    return h

def tester_hb():
    r"""
    Verifie si ``hb`` est une fonction de hachage parfaite
    """
    h = hb('01')
    for n in [1, 2, 3, 6]:
        print 'Test de la fonction h pour n = %s' % n
        print '================================='
        for p in palindromes('01', n):
            print 'h(%s) = %s' % (p, h(p))
        print

tester_hb()
```

Et le résultat est

```
Test de la fonction h pour a = 2 et b = 7
=========================================
h(0, 0) = 0
h(0, 1) = 1
h(0, 2) = 2
h(0, 3) = 3
h(0, 4) = 4
h(0, 5) = 5
h(0, 6) = 6
h(1, 0) = 7
h(1, 1) = 8
h(1, 2) = 9
h(1, 3) = 10
h(1, 4) = 11
h(1, 5) = 12
h(1, 6) = 13

Test de la fonction h pour a = 4 et b = 6
=========================================
h(0, 0) = 0
h(0, 1) = 1
h(0, 2) = 2
h(0, 3) = 3
h(0, 4) = 4
h(0, 5) = 5
h(1, 0) = 6
h(1, 1) = 7
h(1, 2) = 8
h(1, 3) = 9
h(1, 4) = 10
h(1, 5) = 11
h(2, 0) = 12
h(2, 1) = 13
h(2, 2) = 14
h(2, 3) = 15
h(2, 4) = 16
h(2, 5) = 17
h(3, 0) = 18
h(3, 1) = 19
h(3, 2) = 20
h(3, 3) = 21
h(3, 4) = 22
h(3, 5) = 23

Test de la fonction h pour n = 1
================================
h(0) = 0
h(1) = 1
```

```
Test de la fonction h pour n = 2
================================
h(00) = 0
h(11) = 1

Test de la fonction h pour n = 3
================================
h(000) = 0
h(010) = 2
h(101) = 1
h(111) = 3

Test de la fonction h pour n = 6
================================
h(000000) = 0
h(001100) = 4
h(010010) = 2
h(011110) = 6
h(100001) = 1
h(101101) = 5
h(110011) = 3
h(111111) = 7
```