I Algorithme de tri fusion

On rappelle le principe du tri fusion sur une liste OCaml `l` de taille n :

- On divise la liste en deux sous-listes de taille $n/2$.
- On trie chaque sous-liste de manière récursive.
- On fusionne les deux sous-listes triées pour obtenir la liste triée.

1. Écrire l'algorithme de tri fusion en OCaml. On pourra décomposer en plusieurs fonctions.
2. Déterminer la complexité dans le pire cas de votre algorithme.

Solution :

1.

```
let rec fusion l1 l2 = match (l1, l2) with
    | ([], l2) -> l2
    | (l1, []) -> l1
    | (x1::l1, x2::l2) ->
        if x1 < x2 then x1::(fusion l1 (x2::l2))
        else x2::(fusion (x1::l1) l2)

let rec divise l = match l with
    | [] -> ([], [])
    | [x] -> ([x], [])
    | x::y::l ->
        let (l1, l2) = divise l in
        (x::l1, y::l2)

let rec tri_fusion l = match l with
    | [] -> []
    | [x] -> [x]
    | _ ->
        let (l1, l2) = divise l in
        let l1 = tri_fusion l1 in
        let l2 = tri_fusion l2 in
        fusion l1 l2
```

2. Soit $C(n)$ la complexité dans le pire cas de `tri fusion l` pour une liste de taille n. On a :

$$C(n) = 2 \underbrace{C\left(\frac{n}{2}\right)}_{\text{appels récursifs}} + \underbrace{Kn}_{\text{fusion et divise}}$$

En appliquant plusieurs fois la relation de récurrence, on obtient :

$$C(n) = 4C\left(\frac{n}{4}\right) + 2Kn = \ldots = 2^p C\left(\frac{n}{2^p}\right) + pKn$$

En prenant $p = \log_2(n)$, on a $\frac{n}{2^p} = 1$ et $2^p = n$. Donc $C(n) = nC(1) + Kn\log_2(n) = \mathrm{O}(n\log_2(n))$.

Soit A un algorithme de tri sur un tableau `T` de taille n. On considère l'arbre de décision a de A, où chaque noeud interne correspond à une comparaison entre deux éléments de `T` (c'est-à-dire un test de la forme `T[i] < T[j]`). L'exécution de A sur un tableau de taille n correspond à un chemin dans cet arbre, et la feuille atteinte correspond à l'ordre final des éléments (valeur de retour de A).
On note $f(a)$ et $h(a)$ respectivement le nombre de feuilles et la hauteur de l'arbre a.

3. Montrer que $f(a) \geq n!$.

Solution : Chaque feuille de a correspond à un résultat de tri, c'est-à-dire d'une permutation des n éléments de `T`. Il y a $n!$ permutations possibles, donc il faut au moins $n!$ feuilles pour représenter toutes les sorties possibles.

4. Montrer que $f(a) \leq 2^{h(a)}$.

Solution : Montrons par induction $P(a)$: « $f(a) \leq 2^{h(a)}$ » pour un arbre binaire a quelconque.
$P(V)$ est vrai pour un arbre vide V car $f(V) = 0 \leq 2^{h(V)} = 2^{-1}$.
Soit a un arbre binaire de sous-arbres g et d. Supposons $P(g)$ et $P(d)$. Alors :

$$f(a) = f(g) + f(d) \leq 2^{h(g)} + 2^{h(d)} \leq 2^{h(a)-1} + 2^{h(a)-1} = 2^{h(a)}$$

car $h(a) = \max(h(g), h(d)) + 1$.
Ainsi $P(a)$ est vrai et, par principe d'induction, $P(a)$ est vrai pour tout arbre binaire a.

5. En déduire qu'il n'existe pas d'algorithme de tri en complexité $o(n \log n)$ dans le pire cas.

Solution : D'après les deux résultats précédents, on a $2^{h(a)} \geq n!$ donc $h(a) \geq \log_2(n!)$. En utilisant une comparaison série-intégrale, on montre que $\log_2(n!) = \sum_{i=1}^{n} \log_2(i) \sim n \log_2(n)$. Donc pour n assez grand on a $h(a) \geq \frac{n}{2} \log_2(n)$ donc la complexité dans le pire cas de A est au moins $\frac{n}{2} \log_2(n)$.

6. Montrer qu'il est possible de trier un tableau de taille n dont les éléments sont des entiers compris entre 0 et $n-1$ en complexité linéaire. Est-ce en contradiction avec le résultat précédent ?

Solution : On peut utiliser le tri par comptage : on crée un tableau `C` de taille n tel que `C[i]` contienne le nombre de fois que la valeur `i` apparaît dans `T`. Ensuite, on parcourt le tableau de comptage pour reconstruire le tableau trié.
Ce n'est pas en contradiction avec le résultat précédent car on ne compare pas les éléments de `T` entre eux, mais on utilise la connaissance de la plage des valeurs possibles pour les indexer dans le tableau de comptage.

7. Soit a un arbre binaire non vide et $S(a)$ la somme des profondeurs des feuilles de a. Montrer que $S(a) \geq f(a) \log_2(f(a))$, où $f(a)$ est le nombre de feuilles de a.

Solution : Par induction sur a. C'est vrai pour un arbre à un seul noeud : $S(a) = 0 \geq 1 \log_2(1)$.
Soit a un arbre binaire de sous-arbres g et d. Supposons la propriété est vraie pour g et d. Soit k le nombre de feuilles de g et p le nombre de feuilles de a. Si f est une feuille de g, la profondeur $p_a(f)$ de f dans a vérifie $p_a(f) = p_g(f) + 1$ (et de même pour une feuille de d). Donc :

$$S(a) = S(g) + f(g) + S(d) + f(d) = S(g) + S(d) + f(a) \geq f(g) \log_2(f(g)) + f(d) \log_2(f(d)) + f(a)$$

Comme $x \mapsto x \log_2(x)$ est convexe on a :

$$\frac{1}{2} f(g) \log_2(f(g)) + (1 - \frac{1}{2}) f(d) \log_2(f(d)) \geq \frac{1}{2}(f(g) + f(d)) \log_2\left(\frac{f(g) + f(d)}{2}\right) = \frac{1}{2} f(a) \log_2\left(\frac{f(a)}{2}\right)$$

Donc :

$$S(a) \geq \frac{1}{2} f(a) \log_2\left(\frac{f(a)}{2}\right) + f(a) = f(a) \log_2\left(\frac{f(a)}{2}\right)$$

8. Montrer qu'il n'existe pas d'algorithme de tri en complexité $o(n \log n)$ en moyenne.

Solution : Soit A un algorithme de tri d'arbre de décision a. D'après la question précédente, $S(a) \geq f(a) \log_2(f(a))$.
Donc la profondeur moyenne des feuilles de a, c'est-à-dire la complexité moyenne de A, est $\frac{S(a)}{f(a)} \geq \frac{f(a) \log_2(f(a))}{f(a)} = \log_2(f(a))$.
On conclut avec $f(a) \geq n!$ et $\log_2(n!) \sim n \log_2(n)$.

Exercice 4 *Activation de processus (type A)*

Soit un système temps réel à n processus asynchrones $i \in [\![1, n]\!]$ et m ressources r_j. Quand un processus i est actif, il bloque un certain nombre de ressources listées dans un ensemble P_i et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision `ACTIVATION` correspondant ajoute un entier k aux entrées et cherche à répondre à la question : "Est-il possible d'activer au moins k processus en même temps ?"

1. Soit $n = 4$ et $m = 5$. On suppose que $P_1 = \{r_1, r_2\}$, $P_2 = \{r_1, r_3\}$, $P_3 = \{r_2, r_4, r_5\}$ et $P_4 = \{r_1, r_2, r_4\}$. Est-il possible d'activer 2 processus en même temps ? Même question avec 3 processus.
2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème `ACTIVATION`. Évaluer la complexité de votre algorithme.

On souhaite montrer que `ACTIVATION` est NP-complet.

3. Donner un certificat pour ce problème.
4. Écrire en pseudo code un algorithme de vérification polynomial. On supposera disposer de trois primitives, toutes trois de complexité polynomiale :
   (a) `appartient(c,i)` qui renvoie `Vrai` si le processus `i` est dans l'ensemble d'entiers `c`.
   (b) `intersecte(`$\mathtt{P}_i$`,R)` qui renvoie `Vrai` si le processus `i` utilise une ressource incluse dans un ensemble de ressources `R`.
   (c) `ajoute(`$\mathtt{P}_i$`,R)` qui ajoute les ressources $\mathtt{P}_i$ dans l'ensemble `R` et renvoie ce nouvel ensemble.

En théorie des graphes, le problème `STABLE` se pose la question de l'existence dans un graphe non orienté $G = (S, A)$ d'un ensemble d'au moins k sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il $S' \subset S$, $|S'| \geq k$ tel que $s, p \in S' \Rightarrow (s, p) \notin A$?

5. En utilisant le fait que `STABLE` est NP-complet, montrer par réduction que le problème `ACTIVATION` est également NP-complet.

Commentaires du jury

1. On attend du candidat une réponse en oui / non avec une précision de quels processus activer dans le cas où la réponse est oui et une très rapide justification dans le cas où la réponse est non. Une réponse orale suffit.
2. De multiples réponses sont possibles sur cette question et sont acceptées du moment qu'elles sont clairement décrites et correctement analysées. On n'attend pas une complexité optimale.
3. Une courte phrase décrivant la nature d'un tel certificat et indiquant sa polynomialité en la taille de l'entrée suffit à obtenir tous les points.
4. On attend un algorithme utilisant à bon escient les primitives proposées par le sujet.
5. Le jury est attentif au fait que tous les arguments soient bien présents : description de la réduction, preuve que c'en est une et justification de son caractère polynomial pour le caractère NP-difficile ; caractère NP pour pouvoir conclure quant à la NP-complétude.

Solution :

1. P_2 et P_3 peuvent être activés simultanément. Il n'est pas possible d'activer 3 processus en même temps car $r_1 \in P_1, P_2, P_4$ donc il n'est pas possible de prendre 2 processus parmi P_1, P_2, P_4.
2. On créé un tableau de booléens `T` de taille n initialisé à `false`. Pour chaque processus, on met `T[i]` à `true` si le processus utilise la ressource r_i. Le nombre maximum de processus activables est alors le nombre de `true` dans `T`. Il suffit de renvoyer `true` si le nombre de `true` est supérieur ou égal à k.
3. On choisit comme certificat un sous-ensemble de processus, par exemple sous forme d'un tableau de booléens `T` tel que `T[i]` est `true` si le processus P_i appartient au sous-ensemble.
4.

```
Entrée : Tableau de booléens T de taille n
R ← ∅
for i = 0 à n − 1 :
    if T[i] = true :
        if intersecte(i, R) :
            Renvoyer false
        ajoute(i, R)
Renvoyer true
```

5. Montrons que `STABLE` se réduit à `ACTIVATION`. Soit $G = (S, A)$, k une instance de `STABLE`. On construit une instance de `ACTIVATION`, composée d'un ensemble R de ressources et P de processus :
   - Chaque arête est une ressource.
   - Pour chaque sommet s, l'ensemble $V(s)$ des arêtes incidentes à s est un processus.

   Supposons que G, k soit une instance positive de `STABLE`, c'est-à-dire qu'il existe $S \subset G$ tel que $|S| \geq k$ et il n'y a pas d'arête entre deux sommets de S. Alors l'ensemble des processus de la forme $V(s)$ pour $s \in S$ est un ensemble de $\geq k$ processus activables.
   Inversement, supposons qu'il existe un ensemble P de processus activables tel que $|P| \geq k$. Alors l'ensemble des sommets s tels que $V(s) \in P$ est une instance positive de `STABLE`.