Analyse d'algorithmes

1 Algorithmes de tri

1.1 Tri bulle

Soit $t[1 \dots n]$ un tableau dont les éléments appartiennent à un ensemble E sur lequel il existe une relation d'ordre totale $\leq$ (par exemple $\mathbb{N}$ avec la relation $\leq$ traditionnelle.) Classiquement, pour $e_1, e_2 \in E$, on note $e_1 < e_2$ lorsque $e_1 \leq e_2$ et $e_1 \neq e_2$.

Le tri bulle de t consiste à classer t par ordre croissant de la manière suivante. On effectue des passages successifs sur t. À chaque étape d'un passage, si deux éléments sont en ordre inverse, on les permute. Ainsi, à la fin du premier passage, le plus petit élément de t se trouve en première position. De façon plus générale, à la fin du i-ième passage, le i-ième plus petit élément se trouve en i-ième position (en supposant que tous les éléments du tableau sont distincts.)

L'algorithme suivant trie un tableau donné selon le principe énoncé ci-dessus.

```
Tri_bulle (t[1...n]) :
 1 :  i ← 1
 2 :  tant que i ≤ n − 1 faire
 3 :      j ← n
 4 :      tant que j ≥ i + 1 faire
 5 :          si t[j] < t[j − 1] alors
 6 :              valeur ← t[j]
 7 :              t[j] ← t[j − 1]
 8 :              t[j − 1] ← valeur
 9 :          j ← j − 1
10 :      i ← i + 1
```

1. Montrez que l'algorithme termine lorsqu'on lui donne en entrée un tableau fini.
2. Soit n la taille du tableau à trier et k la valeur de la variable i à l'entrée dans la deuxième boucle **tant que**. Soit $C(n, k)$ le nombre de fois où la comparaison de la ligne 5 est effectuée. Exprimez $C(n, k)$ en fonction de n et k.
3. Soit $C(n)$ le nombre total de fois où la comparaison de la ligne 5 est effectuée lorsque l'algorithme prend en entrée un tableau de taille n. Exprimez $C(n)$ en fonction de n.
4. Quelle est la complexité en temps et en espace dans le pire des cas de l'algorithme ?

1.2 Tri par sélection itérée du maximum

Soit t un tableau de taille n dont les éléments appartiennent à un ensemble E sur lequel il existe une relation d'ordre totale $\leq$. Pour $e_1, e_2 \in E$, on note $e_1 < e_2$ lorsque $e_1 \leq e_2$ et $e_1 \neq e_2$.

Le tri de t *par sélection itérée du maximum* consiste à classer t par ordre croissant de la manière suivante.
- On recherche le plus grand élément parmi les n éléments de t et on permute ce plus grand élément avec le dernier élément du tableau.
- On réitère le processus sur les $n-1$ premiers éléments du tableau.

L'algorithme suivant trie un tableau donné selon le principe énoncé ci-dessus.

```
Tri_max (t[1...n]) :
1 :    i ← n
2 :    tant que i ≥ 2 faire
3 :        max ← t[1],   pos ← 1
4 :        j ← 1
5 :        tant que j ≤ i faire
6 :            si max < t[j] alors
7 :                pos ← j,   max ← t[j]
8 :            j ← j + 1
9 :        t[pos] ← t[i],   t[i] ← max
10 :       i ← i − 1
```

1. Montrez que l'algorithme termine lorsqu'on lui donne en entrée un tableau fini.
2. Soit n la taille du tableau à trier et k la valeur de la variable i à l'entrée dans la deuxième boucle **tant que**. Soit $C(n,k)$ le nombre de fois où la comparaison de la ligne 6 est effectuée. Exprimez $C(n,k)$ en fonction de n et k.
3. Soit $C(n)$ le nombre total de comparaisons effectuées par l'algorithme lorsqu'il prend en entrée un tableau de taille n. Exprimez $C(n)$ en fonction de n.
4. Quelle est la complexité en temps et en espace dans le pire des cas de l'algorithme ?

1.3 Tri par fusion

Soit $t[1 \ldots n]$ un tableau dont les éléments appartiennent à un ensemble E sur lequel il existe une relation d'ordre totale $\leq$. Pour $e_1, e_2 \in E$, on note $e_1 < e_2$ lorsque $e_1 \leq e_2$ et $e_1 \neq e_2$.

Le tri de t *par fusion* consiste à classer t par ordre croissant de manière récursive. Exemple d'application de la méthode "diviser pour régner", son principe est de séparer t en deux parties, de les trier, puis d'intercaler ces deux parties triées.

Soit $g, d \in [1, n]$ tels que $g \leq d$. L'algorithme suivant, basé sur le principe énoncé ci-dessus, permet de trier $t[g \ldots d]$ dans l'ordre croissant de ses éléments. Il utilise l'algorithme `Fusion` d'interclassement optimisé par une technique de recopie de $t[g \ldots d]$ dans un tableau t' de sorte que la première moitié de $t[g \ldots d]$ est recopiée dans le bon sens (ligne 1 de `Fusion`), mais la seconde dans le sens inverse (ligne 2 de `Fusion`).

```
Tri_fusion (g, d) :
1 :  si g < d alors
2 :      m ← ⌊(g+d)/2⌋
3 :      Tri_fusion(g, m)
4 :      Tri_fusion(m + 1, d)
5 :      Fusion(g, d, m)
```

```
Fusion (g, d, m) :
1 :  pour i allant de m à g faire t'[i] = t[i]
2 :  pour j allant de m + 1 à d faire t'[d + (m + 1) − j] = t[j]
3 :  i ← g,   j ← d
4 :  pour k allant de g à d faire
5 :      si t'[i] < t'[j] alors
6 :          t[k] ← t'[i],   i ← i + 1
7 :      sinon
8 :          t[k] ← t'[j],   j ← j − 1
```

Soit $g, d \in [1, n]$ avec $g \leq d$. On pose $p = (d - g) + 1$.

1. Calculez le nombre de comparaisons $f(p)$ effectuées lors de l'évaluation de `Fusion`(g, d, m).
2. Soit $c(p)$ le nombre de comparaisons effectuées lors de l'évaluation de `Tri_fusion`(g, d). Donnez une définition récurrente de la suite c sous la forme $c(p) = a.c(\lfloor \frac{p}{2} \rfloor) + b.c(\lceil \frac{p}{2} \rceil) + h(p)$ avec $a \in \mathbb{N}$, $b \in \mathbb{N}$, $a + b \geq 1$ et $h : \mathbb{N} \to \mathbb{R}^+$.
   Remarque : $\lfloor . \rfloor$ désigne la partie entière inférieure et $\lceil . \rceil$ la partie entière supérieure.
3. En posant $\alpha = \log_2(a + b)$, on a le résultat suivant :
   – si $h(p) = \Theta(p^\beta)$ avec $\beta < \alpha$, alors $c(p) = \Theta(p^\alpha)$,
   – si $h(p) = \Theta(p^\alpha)$, alors $c(p) = \Theta(p^\alpha \ln p)$,
   – si $h(p) = \Theta(p^\beta)$ avec $\beta > \alpha$ alors $c(p) = \Theta(p^\beta)$.
   Déduisez-en directement la classe de complexité en temps dans le pire des cas de l'algorithme `Tri_fusion`.
4. Comparez la complexité en temps dans le pire des cas de `Tri_fusion` à celle de `Tri_max`.

2 Approche "Diviser pour régner"

2.1 Puissance n-ième

Soit $a \in \mathbb{N}^*$ et $n \in \mathbb{N}$. L'objectif est de calculer a^n de manière efficace. Une technique consiste à utiliser la méthode "diviser pour régner" : on traite différemment les puissances paires et impaires, plus précisément, calculer a^{2p} (resp. a^{2p+1}) revient à calculer $(a^2)^p$ (resp. $a(a^2)^p$).

L'algorithme suivant est basé sur le principe énoncé ci-dessus.

```
Puissance (a, n) :
1 :  si n = 0 alors
2 :      retourner 1
3 :  sinon si n = 1 alors
4 :      retourner a
5 :  sinon si (n/2 ∈ ℕ) alors
6 :      retourner Puissance(a × a, n/2)
7 :  sinon
8 :      retourner a × Puissance(a × a, ⌊n/2⌋)
```

1. On note $c(n)$ le nombre d'appels récursifs lors de l'évaluation de `Puissance`(a, n). Donnez une définition récurrente de la suite c.
2. Montrez que pour tout $k \in \mathbb{N}$ on a $c(2^k) = k$. Déduisez-en que lorsque n est une puissance de 2 on a $c(n) = \log_2 n$.
3. Montrez par récurrence que la suite c est croissante.
4. En utilisant les questions précédentes montrez que lorsque $n \geq 1$ on a

$$\lfloor \log_2 n \rfloor \leq c(n) \leq \lfloor \log_2 n \rfloor + 1 \ .$$

5. Déduisez de la question 4 que l'algorithme termine.
6. Déduisez de la question 4 la classe de complexité en temps dans le pire des cas de l'algorithme.

2.2 Multiplication de deux entiers

On souhaite calculer de manière efficace le produit de deux entiers naturels sans utiliser la multiplication.

– **Approche naïve** : écrivez un algorithme récursif en vous basant sur les règles suivantes :

$$\begin{cases} m \times 0 = 0 \\ m \times 1 = m \\ m \times (n+1) = m \times n + m \ . \end{cases}$$

Calculez la complexité en nombre d'additions de votre algorithme. Déduisez-en sa classe de complexité en temps dans le pire des cas.

– **Approche "diviser pour régner"** : écrivez un algorithme récursif en vous basant sur les règles suivantes :

$$\begin{cases} m \times 0 = 0 \\ m \times 1 = m \\ m \times (2n+1) = m \times n + m \times n + m \\ m \times (2n) = m \times n + m \times n \ . \end{cases}$$

Soit $m, n \in \mathbb{N}$ et $a(n)$ le nombre d'additions effectuées lors de l'exécution de cet algorithme lorsqu'il prend en entrée m et n.

1. Donnez une définition récurrente de la suite a.
2. Encadrez la suite a par deux suites croissantes α et β de la forme

$$\alpha(n) = \alpha(\lfloor \frac{n}{2} \rfloor) + constante_1 \quad \text{et} \quad \beta(n) = \beta(\lfloor \frac{n}{2} \rfloor) + constante_2 \ .$$

3. Montrez que pour tout $k \in \mathbb{N}$ on a $\alpha(2^k) = k$ et $\beta(2^k) = 2 \times (k+1)$.
4. En utilisant les questions précédentes, montrez que lorsque $n \geq 1$ on a :

$$\lfloor \log_2 n \rfloor \leq a(n) \leq 2 \times (\lfloor \log_2 n \rfloor + 2) \ .$$

5. Déduisez de la question précédente la classe de complexité en temps dans le pire des cas de l'algorithme.
6. Cette approche est-elle plus efficace que l'approche naïve ?

3 Tours de Hanoï

On dispose de n disques de tailles distinctes et de trois piquets sur lesquels les disques peuvent être enfilés. Au départ, les n disques sont sur le premier piquet et sont enfilés dans l'ordre des diamètres décroissants, le plus large en bas. Le jeu consiste à déplacer les n disques sur le dernier piquet rangés comme au départ en déplaçant à chaque coup un disque au sommet d'une pile sur un autre piquet et en ne déposant jamais un disque sur un disque plus petit.
Un algorithme résolvant ce problème résulte d'un raisonnement par récurrence où :
- pour $n = 0$, il n'y a rien à faire ;
- le problème étant résolu pour $n-1$ disques, pour n disques il s'agit de déplacer les $n-1$ premiers du piquet 1 vers le piquet 2, puis de déplacer le n-ième disque du piquet 1 vers le piquet 3, puis de ramener les $n-1$ disques du piquet 2 vers le piquet 3.

L'algorithme suivant est basé sur le principe énoncé ci-dessus (départ, intermédiaire et arrivée sont respectivement le piquet de départ où les disques sont rangés initialement, le piquet intermédiaire et le piquet d'arrivée où les disques doivent être rangés à la fin.)

```
Hanoï (n, départ, intermédiaire, arrivée) :
1 :  si n > 0 alors
2 :      Hanoï (n − 1, départ, arrivée, intermédiaire)
3 :      Écrire (départ, "→", arrivée)
4 :      Hanoï (n − 1, intermédiaire, départ, arrivée)
```

1. On note $c(n)$ le nombre d'appels récursifs effectués lors de l'évaluation de

   Hanoï(n, départ, intermédiaire, arrivée).

   Donnez une définition récurrente de la suite c sous la forme $c(n+1) = \alpha.c(n) + f(n)$ où $\alpha \in \mathbb{R}$, $\alpha \geq 1$ et $f : \mathbb{N} \to \mathbb{R}^+$.
2. **[Complexité en temps]** Lorsque $\alpha > 1$, on a le résultat suivant :
   - si la série de terme général $\frac{f(k)}{\alpha^k}$ converge, $c(n) = O(\alpha^n)$,
   - si $f(n) = O(\alpha^n)$ alors $c(n) = O(n\alpha^n)$,
   - si $f(n) = O(\beta^n)$ avec $\beta > \alpha$, $c(n) = O(\beta^n)$.

   Déduisez-en directement un ordre de grandeur de la complexité en temps dans le pire des cas de l'algorithme Hanoï.
3. **[Terminaison]**
   (a) Montrez que pour tout $n \in \mathbb{N}$ on a $\sum_{k=1}^{n} \frac{1}{2^k} = 1 - \frac{1}{2^n}$.
   (b) Montrez que pour tout entier naturel $n \geq 2$ on a $c(n) = \alpha^{n-1}[c(1) + \sum_{k=1}^{n-1} \frac{f(k)}{\alpha^k}]$.
   (c) Déduisez-en que pour tout entier naturel $n \geq 2$ on a $c(n) = 2^{n+1} - 2$.
   (d) Déduisez-en que l'algorithme Hanoï termine toujours.

4 Algorithme d'Euclide

On se propose de calculer la complexité en temps de l'algorithme d'Euclide de calcul du pgcd basé sur la relation :

$$\text{pgcd}(a, b) = \text{pgcd}(b, r)$$

où r est le reste de la division euclidienne de a par b.

```
pgcd (a, b) :
{on suppose que a ≥ b}
1 :  soit (q, r) tel que a = b.q + r
2 :  si r = 0 alors
3 :      retourner b
4 :  sinon
5 :      retourner pgcd(b, r)
```

1. On considère les couples d'entiers (a_k, b_k) où $a_k \geq b_k > 0$ pour lesquels le calcul du pgcd nécessite k étapes. Étant donnés trois couples successifs dans le processus de calcul de $\text{pgcd}(a, b)$:

   $$(a_{k+1}, b_{k+1}) \rightarrow (a_k, b_k) \rightarrow (a_{k-1}, b_{k-1})$$

   démontrez que $b_{k+1} \geq b_k + b_{k-1}$.

2. On considère la suite de Fibonacci définie par :

   $$\begin{cases} \text{Fibo}(0) = 0, \\ \text{Fibo}(1) = 1, \\ \text{Fibo}(n+2) = \text{Fibo}(n+1) + \text{Fibo}(n) \qquad \forall n \in \mathbb{N} \end{cases}$$

   En utilisant le résultat de la question précédente, montrez par récurrence que pour tout entier k on a $b_k \geq \text{Fibo}(k)$.

3. On suppose que pour tout entier naturel k, on a $\text{Fibo}(k) \geq \Phi^k$ où $\Phi > 1$ est un réel dont on ne précise pas la valeur ici. Déduisez de la question précédente un ordre de grandeur du coût en temps de calcul de $\text{pgcd}(a, b)$.