EA4 – Éléments d'algorithmique II
Devoir maison n° 1 – groupe INFO 3
(Correction)

Exercice 1 : définitions et notations

1. Qu'est-ce qu'un tri *en place* ?
2. Qu'est-ce qu'une *permutation* ?
3. Donner la notation et la définition de « f est *négligeable* par rapport à g. »
4. Donner la notation et la définition de « f est *minorée* par g. »

Correction :

1. *Un tri en place modifie directement le tableau donné en entrée en n'utilisant qu'un espace de stockage auxiliaire borné.*
2. *Une permutation de taille n est une bijection de l'ensemble $[\![1, n]\!]$ sur lui-même correspondant à un ordre de ces éléments.*
3. $f = o(g)$ *si* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. *Ou de façon équivalente,* $\forall c > 0, \exists n_0$, *tel que* $\forall n > n_0, c \cdot f(n) < g(n)$.
4. $f = \Omega(g)$ *si* $\exists c > 0, n_0$, *tels que* $\forall n > n_0, c \cdot g(n) < f(n)$.

Exercice 2 : complexité des tris

Pour chacun des algorithmes de tri vus jusqu'à présent (par insertion*, par sélection, à bulles*, fusion, rapide*), indiquer la complexité en meilleur cas, en pire cas, en moyenne (si elle a été étudiée) exprimée en nombre de (C) comparaisons, (A) assignations/échanges et (T) le total.
* Pour les tris marqués d'une étoile, indiquer la version/variante choisie.

▷ *Complexités données en ordres de grandeur (Θ).*

<table>
<tr><th></th><th>C_{meil}</th><th>C_{moy}</th><th>C_{pire}</th><th>A_{meil}</th><th>A_{moy}</th><th>A_{pire}</th><th>T_{meil}</th><th>T_{moy}</th><th>T_{pire}</th></tr>
<tr><td>À bulles (naïf)</td><td colspan="3" rowspan="2">n^2</td><td rowspan="11">0</td><td colspan="3" rowspan="2">n^2</td><td colspan="2" rowspan="2">n^2</td><td rowspan="4">n^2</td></tr>
<tr><td>À bulles (optimisé)</td></tr>
<tr><td>À bulles (shaker)</td><td rowspan="3">n</td><td colspan="2" rowspan="3">n^2</td><td colspan="3" rowspan="3">n^2</td><td rowspan="3">n</td></tr>
<tr><td>Insertion (normal)</td></tr>
<tr><td>Insertion (dichotomie)</td><td colspan="3">$n \log n$</td><td>$n \log n$</td></tr>
<tr><td>Sélection (en place)</td><td colspan="3">n^2</td><td colspan="3">n</td><td colspan="2">n^2</td></tr>
<tr><td>Rapide (normal)</td><td colspan="2" rowspan="6">$n \log n$</td><td rowspan="2">n^2</td><td colspan="2" rowspan="6">$n \log n$</td><td rowspan="2">n^2</td><td colspan="3" rowspan="6">$n \log n$</td></tr>
<tr><td>Rapide (randomisé)</td></tr>
<tr><td>Rapide (médiane de 3)</td><td rowspan="2">n^2</td><td rowspan="3">$n \log n$</td></tr>
<tr><td>Rapide (médiane)</td></tr>
<tr><td>Fusion</td><td rowspan="2">$n \log n$</td><td rowspan="2">$n \log n$</td></tr>
<tr><td>Rapide (pas en place)</td><td>n^2</td><td>n^2</td><td>n^2</td></tr>
</table>

Exercice 3 : équations de récurrence connues

Pour chacune des équations de récurrence suivantes, identifier un ou plusieurs algorithmes dont la complexité s'exprime par cette équation et en donner un ordre de grandeur (sans le démontrer). Enfin, trier ces ordres de grandeur.

1. $T(n) = T(n-1) + 1$
2. $T(n) = T(n-1) + n$
3. $T(n) = T(n-1) + T(n-2)$
4. $T(n) = T(\frac{n}{2}) + 1$
5. $T(n) = T(\frac{n}{2}) + n$
6. $T(n) = 2\,T(\frac{n}{2}) + n$
7. $T(n) = 3\,T(\frac{n}{2}) + n$
8. $T(n) = 4\,T(\frac{n}{2}) + n$

Correction :

1. $\Theta(n)$. *Fonctions récursives simples telles que exponentiation naïve, factorielle.*

2. $\Theta(n^2)$. *Multiplication naïve de polynômes, tris par sélection, insertion, à bulles implémentés de façon récursive, tri rapide dans le pire cas.*
3. $\Theta(\phi^n)$ *avec* $\phi = \frac{1+\sqrt{5}}{2} \approx 1,618$ *(le nombre d'or). Fibonacci. Comme la fonction est croissante, on peut borner par* $T_\Omega(n) = 2T_\Omega(n-2)$ *et* $T_O(n) = 2T_O(n-1)$*, donc par* $\Omega((\sqrt{2})^n)$ *et* $O(2^n)$.
4. $\Theta(\log n)$. *Exponentiation rapide, dichotomie, recherche dans un ABR équilibré.*
5. $\Theta(n)$. *Médiane ou* k^e *élément* QuickSort*-like avec pivots parfaits.*
6. $\Theta(n \log n)$. *Tri fusion, tri rapide avec pivots parfaits.*
7. $\Theta(n^{\log_2 3})$. *Karatsuba.*
8. $\Theta(n^2)$. *Multiplication naïve de polynômes en diviser pour régner.*
9. $\log n \prec n \prec n \log n \prec n^{\log_2 3} \prec n^2 \prec \phi^n$

Exercice 4 : équation de récurrence

Résoudre la récurrence $\begin{cases} T(0) = T(1) = 1 \\ T(n) = 2\,T(\frac{n}{3}) + 1 \end{cases}$

▷ *On pose* $t(m) = T(3^m)$. *Résolvons* $t(m) = 2t(m-1) + 1$. *En développant* k *fois :* $t(m) = 2^k t(m-k) + k$. *Donc* $t(m) = 2^m t(0) + m = \Theta(2^m)$. *D'où* $T(n) = \Theta(n^{\log_3 2})$ *avec* $\log_3 2 \approx 0,631$ *(ou directement par le master theorem).*

Exercice 5 : ordres de grandeur, vrai ou faux, le retour

Dans cet exercice f et g désignent des fonctions croissantes à valeurs strictement supérieures à 1. Les affirmations suivantes sont-elles vraies ou fausses ? Justifier.

1. $f = \Theta(g) \implies \sqrt{f} = \Theta(\sqrt{g})$
2. $\sqrt{f} = \Theta(\sqrt{g}) \implies f = \Theta(g)$
3. $f = \Theta(g) \implies \log f = \Theta(\log g)$
4. $\log f = \Theta(\log g) \implies f = \Theta(g)$

Correction :

1. *Par hypothèse* $\exists n_0, c, d > 0$*, tels que* $\forall n > n_0, c \cdot g(n) < f(n) < d \cdot g(n)$.
   Donc $\forall n > n_0$*, on a* $\sqrt{c} \cdot \sqrt{g(n)} < \sqrt{f(n)} < \sqrt{d} \cdot \sqrt{g(n)}$. *VRAI.*
2. *Par hypothèse* $\exists n_0, c, d > 0$*, tels que* $\forall n > n_0, c \cdot \sqrt{g(n)} < \sqrt{f(n)} < d \cdot \sqrt{g(n)}$.
   Donc $\forall n > n_0$*, on a* $c^2 \cdot g(n) < f(n) < d^2 \cdot g(n)$. *VRAI.*
3. *La valeur* 1 *pose problème quand on passe au* log. *Ainsi* $1 = \Theta(2)$ *mais* $\log(1) = 0 = o(\log(2))$. *Mais les fonctions sont à valeurs strictement supérieures à* 1.
   Si les fonctions sont bornées alors leur logarithme l'est aussi et est non nul, elles sont donc toutes deux de l'ordre de grandeur $\Theta(1)$.
   Sinon, par hypothèse, $\exists n_0, c, d > 0$*, tels que* $\forall n > n_0, c \cdot g(n) < f(n) < d \cdot g(n)$. *Alors* $\forall n > n_0$*, on a* $\log(c) + \log(g(n)) < \log(f(n)) < \log(d) + \log(g(n))$. *Comme* g *est croissante et non bornée,* $\exists n_1 \geq n_0$*, tel que* $\forall n > n_1, \log(c) \geq -\frac{1}{2}\log(g(n))$ *(utile uniquement si* $c < 1$*). De même,* $\exists n_2 \geq n_1$*, tel que* $\forall n > n_2, \log(d) \leq \log(g(n))$ *(utile uniquement si* $d > 1$*). Alors* $\forall n > n_2$*, on a* $\frac{1}{2}\log(g(n)) < \log(f(n)) < 2 \cdot \log(g(n))$. *En conclusion* $\log f = \Theta(\log g)$. *VRAI.*
4. $\log n = \Theta(\log n^2)$ *mais* $n = o(n^2)$. *FAUX.*
   On l'a également vu pour $n!$ *et* n^n.

Exercice 6 : le tri flou d'intervalles, c'est clair

On considère un tableau `T` de données à trier dont la valeur n'est pas connue de manière exacte, mais seulement par un encadrement : `T` est donc un tableau de couples représentant des intervalles `T[`i`]` $=$ [`T[`i`][0]`, `T[`i`][1]`] tels que le i^{e} élément x_i de l'ensemble appartient à `T[`i`]` :

$$\texttt{T[}i\texttt{][0]} \leqslant x_i \leqslant \texttt{T[}i\texttt{][1]}.$$

Le *tri flou* d'un tel tableau de longueur n consiste à réordonner les éléments de `T` en un tableau `S` tel qu'il existe $(x_0, x_1, \ldots, x_{n-1})$ vérifiant :

$$\forall i,\ x_i \in \texttt{S[}i\texttt{]} \quad \text{et} \quad x_0 \leqslant x_1 \leqslant \cdots \leqslant x_{n-1}.$$

1. Effectuer un tri flou des intervalles suivants :

$$[11,15],\ [19,21],\ [5,8],\ [10,13],\ [17,20],\ [2,4],\ [14,16],\ [3,6],\ [17,18],\ [9,12]$$

▷ *De nombreuses solutions sont possibles. Par exemple :*
$[2,4],[5,8],[3,6],[11,15],[10,13],[9,12],[14,16],[17,18],[19,21],[17,20]$

2. Quels sont tous les résultats possibles du tri flou des intervalles suivants ?

$$A=[4,7],\ B=[8,10],\ C=[6,9],\ D=[11,13],\ E=[12,15].$$

▷ *ACBDE, ACBED, ABCDE, ABCED, CABDE, CABED*

On dit qu'un problème P est ***moins dur*** qu'un problème Q si tout algorithme qui résout le problème Q peut être modifié pour résoudre le problème P sans augmentation de complexité.

3. Expliquer pourquoi le problème du tri flou est moins dur que le problème du tri usuel.
▷ *On peut se contenter d'appliquer un tri usuel sur la première composante de chaque intervalle.*

4. Que peut-on dire lorsque certains intervalles à trier ont une intersection non vide ?
▷ *Dans ce cas, on n'est pas obligé d'e ectuer tous les échanges. En e et, les intervalles qui partagent une intersection non vide peuvent être placés dans n'importe quel ordre.*

On souhaite tirer parti de cette propriété pour obtenir un algorithme de tri flou, basé sur le tri rapide (`QuickSort`), mais de complexité moindre lorsque les instances du problème sont plus simples. Pour cela, on va modifier l'étape de partitionnement du tri rapide. Une fois l'intervalle `pivot` choisi, on définit un sous-intervalle ***non vide*** de `pivot`, appelé `chevauchement`, ayant la propriété suivante :

> *`chevauchement` est égal à l'intersection des intervalles de `T` qui l'intersectent.* (*)

En particulier, si un intervalle `I` appartenant à `T` ne contient pas `chevauchement`, alors ils sont disjoints.

5. Quels sont tous les intervalles `chevauchement` possibles si `T` contient les intervalles de l'exemple de la question 2 et si `pivot`$=[6,13]$?
▷ $[6,7]=A\cap C$, $[8,9]=B\cap C$, $[12,13]=D\cap E$.

L'ensemble `T` d'intervalles est alors partitionné en trois sous-ensembles :
- les `petits` intervalles, dont tous les éléments sont plus petits que ceux de `chevauchement`,
- les intervalles `moyens`, qui intersectent (et donc chevauchent) `chevauchement`,
- les `grands` intervalles, dont tous les éléments sont plus grands que ceux de `chevauchement`.

Dans un premier temps, nous allons décrire un algorithme de tri flou en admettant l'existence de `chevauchement`. Plus précisément, on supposera que `trouveChevauchement(T, pivot)` calcule un tel intervalle en temps linéaire en la longueur de `T`.

Pour simplifier, on construira le tableau trié dans un nouveau tableau.

6. Écrire une fonction `compare(I, J)` qui renvoie -1, 0 ou 1 selon la position de l'intervalle `I` par rapport à l'intervalle `J` :
- -1 si tous les éléments de `I` sont strictement plus petits que ceux de `J`,
- 0 si `I` et `J` s'intersectent,
- 1 si tous les éléments de `I` sont strictement plus grands que ceux de `J`.

▷
```
def compare(I, J) :
  if I[1] < J[0] : return -1
  if J[1] < I[0] : return 1
  return 0
```

```
# Ou en une ligne pour les plus motivés :
def compare(I, J) :
  return int(I[0] > J[1]) - int(I[1] < J[0])
```

7. Écrire la fonction `partition(T, chevauchement)` qui partitionne le tableau `T` par rapport à un intervalle `chevauchement` ayant la propriété (*).

▷
```
def partition(T, chevauchement) :
  petits = [ I for I in T if compare(I, chevauchement) == -1 ]
  moyens = [ I for I in T if compare(I, chevauchement) == 0 ]
  grands = [ I for I in T if compare(I, chevauchement) == 1 ]
  return petits, moyens, grands
```

```
# Autre possibilité en un seul passage :
def partition(T, chevauchement) :
  res = [[], [], []]
  for I in T :
    res[compare(I, chevauchement) + 1].append(I)
  return res
```

8. Écrire un algorithme inspiré de QuickSort réalisant le tri flou d'un tableau d'intervalles.

   ▷
   ```
   def triFlou(T) :
     if len(T) <= 1 : return T
     pivot = T[0]
     chevauchement = trouveChevauchement(T, pivot)
     petits, moyens, grands = partition(T, chevauchement)
     return triFlou(petits) + moyens + triFlou(grands)
   ```

9. Quelle est sa complexité dans le pire des cas ?

   ▷ *Sa complexité dans le pire cas est*

   $$T(n) = \max_{p\geq 0; m\geq 1} (T(p) + T(n-p-m)) + n$$

   où p est la taille de `petits`*, m celle de* `moyens`*. Le pire cas est exactement comme celui de* QuickSort*, c'est-à-dire $p = 0$ et $m = 1$, donc une complexité en $O(n^2)$.*

10. Quelle est sa complexité si les intervalles à trier ont une intersection non vide ?

    ▷ *Attention, ce cas est di érent de celui de la question 4. Si l'ensemble des intervalles a une intersection non vide, il n'y a rien à faire (on peut choisir tous les x_i égaux à n'importe quel entier contenu dans l'intersection).* `partition` *retourne* `[], T, []`*. Dans ce cas $p = 0$ et $m = n$ donc la complexité est linéaire.*

11. En vous basant sur la complexité en moyenne de QuickSort, borner la complexité en moyenne de votre algorithme.

    ▷ `triFlou` *se comporte toujours au moins aussi bien que* QuickSort*, mieux dans le cas où* `moyens` *comporte strictement plus d'un élément, donc on peut borner la complexité en moyenne par celle de* QuickSort*, c'est-à-dire $\Theta(n \log n)$.*

Nous allons maintenant écrire la fonction `trouveChevauchement(T, pivot)`.

12. L'algorithme décrit aux questions 7 et 8 fonctionne-t-il toujours si `chevauchement` est remplacé par `pivot` ?

    ▷ *Non, dans l'exemple si le pivot est C, alors* `moyens` $= \{A, B, C\}$ *donc on risque de placer B avant A.*

13. Écrire une fonction `intersection(I, J)` qui renvoie l'intervalle intersection de `I` et `J` si ces deux intervalles ont une intersection non vide.

    ▷
    ```
    def intersection(I, J) : # on suppose compare(I, J) == 0
      return [max(I[0], J[0]), min(I[1], J[1])]
    ```

14. Écrire une fonction `trouveChevauchement(T, pivot)` qui renvoie un sous-intervalle (non vide) `chevauchement` de `pivot` ayant la propriété (*).

    On ne demande pas que `chevauchement` soit « optimal » parmi les sous-intervalles de `pivot` ayant cette propriété (par exemple du point de vue du nombre d'intervalles qui l'intersectent). En revanche, on exige que la complexité de la fonction `trouveChevauchement(T, pivot)` soit au plus linéaire en la longueur de `T`.

    ▷
    ```
    def trouveChevauchement(T, pivot) :
      res = pivot
      for I in T :
        if compare(res, I) == 0 :
          res = intersection(res, I)
      return res
    ```