EA4 – Éléments d'algorithmique II
Examen de 1re session – 10 mai 2022
Durée : 3 heures

Aucun document autorisé excepté une feuille A4 manuscrite
Appareils électroniques éteints et rangés

Le sujet est trop long. Ne paniquez pas, le barème en tiendra compte. Il est naturellement préférable de ne faire qu'une partie du sujet correctement plutôt que de tout bâcler.
Les (∗) marquent des questions plus difficiles que les autres. Elles seront hors barème.
Les indications de durée sont manifestement trop optimistes et ne sont là qu'à titre comparatif entre les différents exercices.
Vous êtes libres de choisir le langage utilisé pour décrire les algorithmes demandés, du moment que la description est suffisamment précise et lisible.

Lisez attentivement l'énoncé.
Sauf mention contraire explicite, les réponses doivent être justifiées.

Exercice 1 : suites récurrentes linéaires *(15-20 minutes)*

On considère l'algorithme suivant :

```
def F(n) :
    return 0 if n < 2 else 1 if n == 2 else 2 * F(n-1) + F(n-3)
```

1. Soit $A(n)$ le nombre d'opérations arithmétiques sur des entiers effectuées lors de l'exécution de `F(n)`. Montrer que $A(n) \in \Omega(2^{n/3})$.
2. Proposer un algorithme `Fd(n)` calculant la même valeur que `F(n)` en effectuant un nombre linéaire d'opérations arithmétiques sur des entiers, avec une complexité en espace linéaire. Quelle est sa complexité en temps ?
3. Proposer un algorithme `Fe(n)` calculant la même valeur de manière encore plus efficace. Quelle est sa complexité en temps ?

Exercice 2 : répétitions *(15-20 minutes)*

On s'intéresse au problème suivant : étant donné une liste `L` de n nombres, compter les valeurs qui apparaissent au moins deux fois dans `L`. On considère les deux algorithmes suivants :

```
def repetitions_naif(L) :
  cpt, l = 0, len(L)
  for i in range(l) :
    for j in range(i+1, l) :
      if L[i] == L[j] : cpt += 1
  return cpt
```

```
def repetitions_par_comptage(L) :
  cpt, m = 0, max(L)+1
  tab = [0] * m
  for elt in L : tab[elt] += 1
  for nb in tab : if nb >=2 : cpt += 1
  return cpt
```

1. Pourquoi `repetitions_naif` est-il incorrect ? Proposer une version corrigée, puis calculer sa complexité, en précisant quelles opérations élémentaires sont prises en compte.
2. À quelle condition sur `L` `repetitions_par_comptage` est-il correct ? Calculer sa complexité dans ce cas, en précisant quelles opérations élémentaires sont prises en compte.

Ex 1

```
def F(n)
    return 0 if n < 2 else 1 if n == 2 else
2 * F(n-1) + F(n-3)
```

$$A(n) \begin{cases} 0 & \text{si } n \leq 2 \\ A(n-1) + A(n-3) + 2 \end{cases}$$

$A(n-1)$ ↑ $F(n-1)$ — $A(n-3)$ ↑ $F(n-3)$ — 2 ↑ $F(n)$ fait 1 multiplication et 1 addition

3. Proposer un algorithme inspiré de `repetitions_par_comptage` adapté au cas général. Quelle est sa complexité dans le pire cas ? Rappeler (sans démonstration) sa complexité en moyenne.

4. Proposer enfin un algorithme de complexité strictement meilleure dans le pire cas, et justifier sa complexité.

Exercice 3 : intersections de segments *(20 minutes)*

On considère deux ensembles de n points du plan, $\{P_0, \dots, P_{n-1}\}$, sur la droite $y = 0$, et $\{Q_0, \dots, Q_{n-1}\}$, sur la droite $y = 1$, tous distincts, définissant n segments $[P_iQ_i]$. Le but de l'exercice est de calculer, le plus efficacement possible, le nombre de paires de segments qui s'intersectent.
Les abscisses des points sont données par deux tableaux `P` et `Q` de longueur n.

1. Si `P[`i`]` $<$ `P[`j`]`, à quelle condition les segments $[P_iQ_i]$ et $[P_jQ_j]$ s'intersectent-ils ?

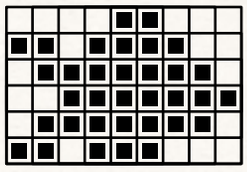
2. Décrire un algorithme permettant de calculer le nombre de paires de segments qui s'intersectent sans modifier `P` et `Q` et avec une mémoire auxiliaire constante. Quelle est sa complexité (en temps) ?

3. Supposons le tableau `P` croissant. Combien de segments $[P_iQ_i]$ intersecte-t-il dans ce cas ? À quelle grandeur caractéristique de `Q` le nombre de paires de segments qui s'intersectent est-il donc égal ?

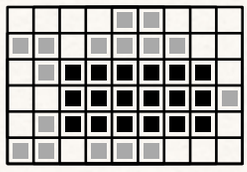
4. En déduire un algorithme le plus efficace possible, basé sur le principe « diviser pour régner », pour calculer le nombre de paires de segments qui s'intersectent.

Exercice 4 : blocs monochromes *(50-60 minutes)*

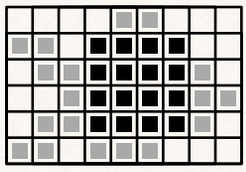
On considère une image en noir et blanc pixélisée, représentée par un tableau `M` de bits (0 pour blanc et 1 pour noir) de m lignes et n colonnes. On appelle *bloc* un sous-tableau rectangulaire de `M` constitué uniquement de 1, et on cherche à déterminer l'aire maximale d'un bloc de `M`.
Par exemple, dans l'image ci-dessous, l'aire maximale d'un bloc est 18 :



(a) exemple de bitmap



(b) bloc maximal



(c) bloc carré maximal

0	0	0	0	1	2	0	0	0
1	2	0	1	2	3	4	0	0
0	1	2	3	4	5	6	7	0
0	0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7	0
1	2	0	1	2	3	0	0	0

(d) longueur des blocs-lignes

1. Cas particulier des blocs-lignes (*i.e.* de hauteur 1) : soit $L(i, j)$ la longueur du plus grand bloc du sous-tableau `M[`i`][:`j`+1]` contenant le pixel `M[`i`][`j`]`.

a. Quelle est la complexité du calcul de $L(i, j)$? Et donc du calcul naïf de la longueur maximale d'un bloc-ligne de `M` ?

b. Comment calculer $L(i, 0), L(i, 1), \dots L(i, n-1)$ en temps *cumulé* $\Theta(n)$? Dans quel sens peut-on dire que le calcul de $L(i, j)$ est en $O(1)$?

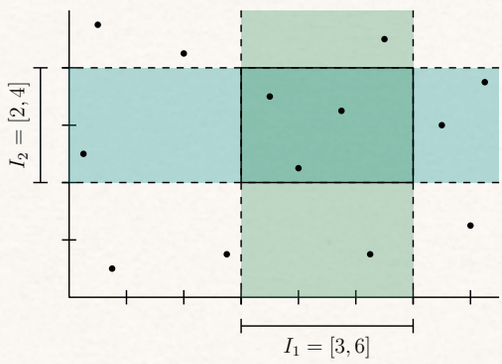
c. En déduire un algorithme de calcul de la longueur maximale d'un bloc-ligne de `M` plus efficace que le précédent. Quelle est sa complexité dans le pire et le meilleur cas ? Cet algorithme est-il optimal ?

Dans la suite, on note (abusivement) `M[:`i`][:`j`]` le sous-tableau rectangulaire de `M` constitué des j premières colonnes de ses i premières lignes.

2. **Cas particulier des blocs carrés :** soit $C(i,j)$ la largeur du plus grand bloc *carré* de `M[:`i`+1][:`j`+1]` contenant le pixel `M[`i`][`j`]`.
   a. Donner une définition récursive de $C(i,j)$ en fonction de $C(i-1,j-1)$, $C(i-1,j)$ et $C(i,j-1)$.
   b. En déduire un algorithme optimal de calcul du plus grand bloc carré de `M`. Justifier sa complexité.
3. **Cas général – méthode 1 :** soit $L_h(i,j)$ la largeur maximale d'un bloc de `M[:`i`+1][:`j`+1]` de hauteur h et contenant le pixel `M[`i`][`j`]`.
   a. Exprimer $L_h(i,j)$ en fonction de $L_{h-1}(i-1,j)$ et $L_{h-1}(i,j-1)$.
   b. En déduire un algorithme de calcul de l'aire maximale d'un bloc de `M` en temps $\Theta(m^2 n)$.
4. **Cas général – méthode 2 :**
   a. (*) Comment calculer l'aire maximale d'un bloc « à cheval » sur les lignes `m//2` et `m//2+1` en temps $\Theta(mn)$?
   b. En déduire un algorithme de calcul de l'aire maximale d'un bloc en temps $\Theta(mn \log mn)$.

Recherche par plage

Dans les trois derniers exercices, on s'intéresse au problème suivant : soit $k \in \mathbb{N}^*$, $\mathcal{N}$ un nuage de points de $\mathbb{R}^k$, et k intervalles $I_1, \ldots, I_k$; déterminer le nombre de points de $\mathcal{N}$ situés dans le pavé rectangulaire $I_1 \times \cdots \times I_k$.
Par exemple, le rectangle $[3,6] \times [2,4]$ de la figure ci-contre contient 3 points du plan.



Exercice 5 : cas à une seule dimension *(10-15 minutes)*

Dans cet exercice, on considère le problème en dimension 1 : étant donné un ensemble $\mathcal{N}$ de nombres réels, et un intervalle I, compter les éléments de $\mathcal{N} \cap I$.
Pour simplifier, on suppose toutes les valeurs distinctes.

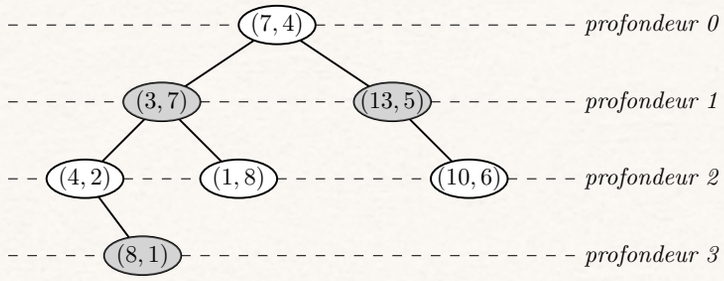
1. Décrire un algorithme simple `comptePoints(N, a, b)` permettant de résoudre ce problème si $\mathcal{N}$ est représenté par un tableau `N` non trié et I par un couple `(a, b)`. Quelle est sa complexité ?
2. Décrire un algorithme plus efficace `comptePointsTrie(N, a, b)` dans le cas où `N` est trié. Justifier sa complexité.
3. Supposons qu'on doive répondre à un nombre m de requêtes successives sur un même tableau `N` *initialement non trié*. Pour quels ordres de grandeur de m est-il judicieux de trier `N` ?
4. Les deux algorithmes précédents peuvent-ils être adaptés à la dimension 2, avec un nuage représenté par un tableau `N` de points ? Justifier.

En dimension $k \geqslant 2$, un algorithme efficace repose sur l'utilisation de *kD-arbres* : pour $k = 2$, un 2D-arbre est un arbre binaire dont chaque nœud `r` contient un point (x_r, y_r), tel que :

– si `r` est situé à une ***profondeur paire***, alors tout point contenu dans son sous-arbre gauche a une abscisse $x < x_r$, et tout point contenu dans son sous-arbre droit a une abscisse $x > x_r$;
– si `r` est situé à une ***profondeur impaire***, alors tout point contenu dans son sous-arbre gauche a une ordonnée $y < y_r$, et tout point contenu dans son sous-arbre droit a une ordonnée $y > y_r$.

Exercice 6 : construction d'un 2D-arbre de hauteur minimale *(20-25 minutes)*

1. Expliquer pourquoi l'arbre de hauteur 3 ci-dessous ***n'est pas*** un 2D-arbre.



(en grisé, les nœuds à profondeur impaire ; la racine est à profondeur 0, donc paire)

Dessiner deux 2D-arbres contenant les mêmes points que cet arbre, l'un de hauteur minimale, l'autre de hauteur maximale.

2. Soit `N` un tableau de n points, et `A` un 2D-arbre de hauteur *minimale* contenant les mêmes points que `N`. Quelle est la hauteur de `A` ? Quel point r faut-il placer à sa racine ? Quels points g et d doivent être à la racine de son sous-arbre gauche et de son sous-arbre droit ? Quelle est la complexité (au pire et en moyenne) de l'algorithme vu en cours permettant de calculer r à partir de `N` ?

3. Pour faciliter le calcul de r, g et d, on suppose que `N` a été préalablement trié selon les abscisses croissantes (tableau `Nx`) ***et*** selon les ordonnées croissantes (tableau `Ny`). Avec quelle complexité peut-on alors trouver r ? Décrire un algorithme de partitionnement de `Nx` et `Ny` permettant de faire de même pour trouver g et d. Quelle est sa complexité ?

4. En déduire un algorithme `construire2Darbre(N)` permettant de construire un 2D-arbre de hauteur minimale à partir d'un tableau de points `N` quelconque fixé, et déterminer sa complexité.

Exercice 7 : recherches dans un 2D-arbre *(25-30 minutes)*

1. Décrire un algorithme (récursif) `recherche2Darbre(A, p)` testant l'appartenance d'un point `p` à un nuage représenté par un 2D-arbre `A`. Quelle est sa complexité si `A` est de taille n et de hauteur h ? *(il pourra être commode d'utiliser un paramètre optionnel supplémentaire pour tenir compte du niveau :* `recherche2Darbre(A, p, niveau = 0)`*)*

Chaque nœud `p` d'un 2D-arbre `A` est la racine d'un sous-arbre dont tous les points appartiennent à une région rectangulaire `cellule(A, p)`, éventuellement non bornée. En particulier, la cellule correspondant à la racine est le plan entier, celles de ses deux fils sont les demi-plans définis par la droite verticale passant par le point racine.

2. Faire un schéma des cellules correspondant aux feuilles de chacun des deux 2D-arbres de la question 1, exercice 6.

3. Comment modifier l'algorithme de la question 1 pour déterminer `cellule(A, p)`, représentée par un quadruplet `(xmin, ymin, xmax, ymax)` ? *(en particulier, le quadruplet correspondant à la racine est* `(None, None, None, None)`*)*

4. (*) En étudiant les différentes positions relatives possibles de `cellule(A, p)` et d'une plage rectangulaire fixée, en déduire un algorithme (récursif) `comptePoints(A, a, b, c, d)` qui compte les points d'un 2D-arbre `A` situés dans le rectangle $[a, b] \times [c, d]$. Justifier sa correction. *(on peut montrer que l'algorithme demandé est de complexité $O(\sqrt{n} + m)$, si m est la valeur calculée par l'algorithme)*

1. Ce n'est pas un 2D-arbre car il respecte pas la condition que tous les noeuds à gauche d'un noeud donné doivent avoir une coordonnée