# EA4 – Éléments d'algorithmique
# Examen de 1re session – 24 mai 2016
## Durée : 3 heures

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

**Préliminaires :** *Ce sujet est constitué de ~~7~~ 6 exercices qui peuvent être traités dans l'ordre de votre choix, même si les exercices 4 à 6 sont complémentaires. Il est donc vivement conseillé de lire l'intégralité du sujet avant de commencer. Le sujet est trop long, le barème en tiendra compte. Il est préférable de ne faire qu'une partie du sujet correctement plutôt que de tout bâcler.*

***Sauf mention contraire explicite, toutes les réponses doivent être justifiées.***

*En l'absence de précision, les complexités demandées sont les ordres de grandeur ($\Theta$) des complexités en temps dans le pire cas des algorithmes concernés.*

*Vous êtes libres du langage utilisé pour décrire les algorithmes demandés, du moment que la description est suffisamment précise et lisible : python, pseudo-code, français, java...*

### Exercice 1 : répétitions

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

**1.** Pourquoi l'algorithme suivant est-il incorrect ?

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

**2.** Proposer une version corrigée, puis calculer sa complexité, en précisant quelles opérations élémentaires sont prises en compte.

**3.** Proposer un algorithme de complexité strictement meilleure permettant de résoudre le problème, et justifier sa complexité.

### Exercice 2 : fusion de listes triées

**1.** Rappeler (sans démonstration) l'algorithme optimal de fusion de deux listes triées.

**2.** En déduire un algorithme de complexité $\Theta(kn)$ permettant de fusionner $k$ listes triées, où $n$ est la longueur de la liste fusionnée, sans créer d'autre liste que la liste fusionnée.

**3.** En utilisant un tas auxiliaire, proposer un algorithme plus efficace. Quelle est sa complexité ?

**4.** Comment obtenir la même complexité sans tas auxiliaire, en s'autorisant à créer des listes intermédiaires ?

### Exercice 3 :

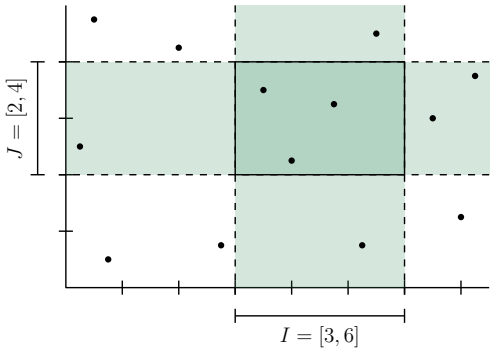
Soit `T` le tableau suivant : | 9 | 8 | 12 | 13 | 11 | 4 | 15 |. Détailler l'exécution sur `T`, en comptant (exactement) les comparaisons effectuées,

**1.** du tri rapide avec mémoire auxiliaire et pivot `T[0]` ;

**2.** du tri par tas *(ne pas hésiter à passer à une représentation sous forme d'arbre).*

## Recherche par plage

On s'intéresse au problème suivant : soit $\mathcal{N}$ un nuage (ensemble fini) de points du plan, et deux intervalles $I$ et $J$ ; déterminer le nombre de points de $\mathcal{N}$ situés dans le rectangle $I \times J$. Par exemple, le rectangle $[3,6] \times [2,4]$ de la figure ci-contre contient 3 points.




### Exercice 4 : cas à une dimension

L'analogue en dimension 1 est le suivant : étant donné un ensemble de nombres $\mathcal{N}$, 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.

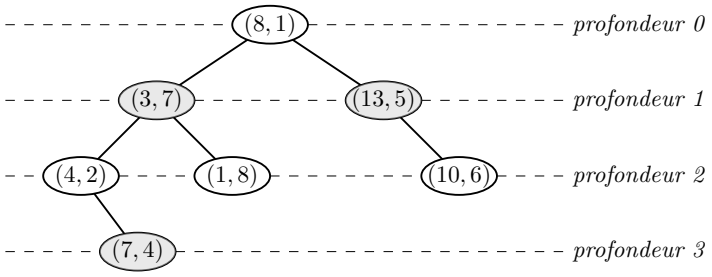
Un algorithme efficace en dimension 2 repose sur l'utilisation de *2D-arbres* ; un 2D-arbre est un arbre binaire dont chaque nœud `r` contient un point $(x_r, y_r)$, tel que :

– 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 5 : construction d'un 2D-arbre de hauteur minimale

**1.** Vérifier que l'arbre de hauteur 3 ci-dessous est un 2D-arbre.




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

Dessiner deux 2D-arbres contenant les mêmes points, 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(P)` permettant de construire un 2D-arbre de hauteur minimale à partir d'un tableau de points `P` et déterminer sa complexité.

### Exercice 6 : recherches dans un 2D-arbre

**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 l'arbre de la question 1, exercice 5.

**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)*

---