# EA4 – Éléments d'algorithmique II
# Examen de 1re session – 17 mai 2023
## 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 : hachage *(15-20 minutes)*

On considère une table de hachage `T` à adressage ouvert, de longueur initiale $m = 8$ avec la fonction de hachage $h(k) = k \mod m$. La résolution des collisions se fait par sondage linéaire, et la table a un taux de remplissage maximal autorisé de $\frac{3}{4}$.

1. Insérer successivement les clés 5, 28, 19, 15, 20, 35, 37, 26 et 10 en donnant toutes les indications nécessaires à la compréhension ; en particulier, indiquer clairement les cases sondées pour chaque insertion, et détailler précisément le cas de la clé 37.
2. Décrire précisément la suppression des clés 26, 19 puis 20, puis l'insertion de la clé 67.
3. On poursuit ensuite les insertions : 11, 17, 31, 8, 47, 25, 51... Quand le redimensionnement suivant se produit-il ?

### Exercice 2 : suites récurrentes linéaires *(15-20 minutes)*

On considère l'algorithme suivant :

```
def F(n) :
    return 0 if n < 3 else 1 if n == 3 else 2 * F(n-1) + 4 * F(n-3) + F(n-4)
```

1. Soit $A(n)$ le nombre d'opérations arithmétiques sur des entiers effectuées lors de l'exécution de `F(n)`. Montrer que $A(n) \in \Omega(3^{n/4})$.
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 la meilleure complexité en espace possible. Quelle est sa complexité en espace ? en temps ?
3. Proposer un algorithme `Fe(n)` calculant la même valeur de manière encore plus efficace. Quelle est sa complexité en temps ?

**Exercice 3 : élections à la majorité absolue** *(1 heure)*

Le but de cet exercice est de comparer plusieurs solutions au problème suivant :

> *Étant donné une liste de $n$ éléments, déterminer s'il existe parmi eux un élément* ***majoritaire****, c'est-à-dire apparaissant strictement plus que $\frac{n}{2}$ fois.*

Certaines solutions suggérées sont des applications directes des algorithmes vus en cours. Il vous est demandé de les utiliser, ***pas de les réécrire***. Vous pouvez énoncer sans démonstration (mais précisément) les résultats de complexité vus en cours.

Dans la suite, `T` désigne un tableau de taille $n$. Les différents algorithmes demandés devront renvoyer l'élément majoritaire s'il existe, et `None` sinon.

**1.** *Solution 1 (naïve, pour des éléments quelconques)* Décrire un algorithme permettant de résoudre le problème posé sans faire aucune hypothèse sur la nature des éléments de `T`, sans modifier `T`, et avec une complexité en espace constante. Quelle est sa complexité en temps ? Préciser le meilleur et le pire cas, selon que la recherche est fructueuse ou infructueuse.

**2.** *Solution 2 (pour des éléments comparables)* Proposer une solution plus efficace (en temps) dans le cas où il existe une relation d'ordre sur les éléments de `T`. Préciser les complexités en espace et en temps (au mieux, au pire et en moyenne).

**3.** *Solution 3 (pour des entiers)*

- **a.** Proposer une solution plus efficace (en temps) dans le cas où les éléments de `T` sont des entiers compris entre 0 et une constante `m`. Préciser les complexités en espace et en temps, dans le meilleur et le pire cas.
- **b.** Généraliser cette solution à des éléments entiers quelconques. Que deviennent les complexités en espace et en temps ?

**4.** *Solution 4 (pour des éléments quelconques)* Comment le principe de la solution précédente se généralise-t-il à des éléments quelconques ? Préciser les complexités en espace et en temps (au mieux, au pire et en moyenne).

***Dorénavant, la seule opération autorisée sur les éléments de `T` est le test d'égalité.***

**5.** *Solution 5 (« diviser pour régner »)*

- **a.** Soit `T1` et `T2` tels que `T = T1 + T2`. Si `x` est majoritaire dans `T`, qu'en est-il dans `T1` et dans `T2` ? Et réciproquement ?
- **b.** En déduire un algorithme de type « diviser pour régner » pour résoudre le problème.
- **c.** Soit $C(n)$ le nombre de comparaisons nécessaires, dans le pire cas, pour un tableau de longueur $n$. Écrire l'équation satisfaite par $C(n)$, et en déduire son ordre de grandeur.

**6.** *Solution 6 (presque optimale)*

Pour améliorer la complexité, on relâche un peu la contrainte, et on cherche un algorithme `peutEtreMajoritaires(T)` qui renvoie un couple d'éléments de `T` (éventuellement identiques) tel qu'***aucun autre*** élément de `T` n'est majoritaire.

- **a.** Comment obtenir `majoritaire(T)` à partir de `peutEtreMajoritaires(T)` ?

On dit qu'un élément de `T` est ***quasi-majoritaire*** s'il y apparaît au moins $\frac{n}{2}$ fois.

- **b.** Que peut-on dire d'un tableau possédant deux éléments quasi-majoritaires ?
- **c.** Supposons que, pour tout $i$ tel que $2i+1 <$ `len(T)`, `T[2*`$i$`]` $\neq$ `T[2*`$i$`+1]`. À quelle condition `T` possède-t-il un élément quasi-majoritaire ? *(on pourra distinguer selon la parité de `len(T)`)*
- **d.** Supposons que, pour tout $i$ tel que $2i+1 <$ `len(T)`, `T[2*`$i$`]` $=$ `T[2*`$i$`+1]`. Si `T` possède un élément quasi-majoritaire `x`, qu'en est-il de `T[::2]` ?
- **e.** $(*)$ En déduire un algorithme récursif `peutEtreMajoritaires(T)` utilisant un sous-tableau de `T` de longueur au plus $\lceil \frac{n}{2} \rceil$.
- **f.** Quelle est la complexité (au pire) de l'algorithme `majoritaire(T)` obtenu ?

**Exercice 4 : sélection du $k^{\text{e}}$ élément** *(1 heure 20)*

Le but de cet exercice est de comparer plusieurs solutions au problème suivant, appelé *problème de la sélection* :

> *Étant donné un ensemble $\mathcal{E}$ de $n$ éléments comparables et un entier positif $k \leqslant n$, déterminer l'élément de* ***rang*** *$k$ de $\mathcal{E}$, c'est-à-dire l'élément $x$ de $\mathcal{E}$ tel que :*
> $$\#\{y \in \mathcal{E} \mid y \leqslant x\} = k \quad \text{et} \quad \#\{y \in \mathcal{E} \mid y > x\} = n - k$$

**Cas d'une liste triée** *(et sans doublons)*

**1.** Décrire dans chacun des cas suivants comment sélectionner *de manière optimale* le $k^{\text{e}}$ élément d'un ensemble de taille $n$ représenté par une liste triée sans doublons, plus précisément :

- **a.** par un tableau trié `T`,
- **b.** par une liste simplement chaînée triée `L`,
- **c.** par une liste *doublement* chaînée triée `D`.

Dans chaque cas, déterminer la complexité de l'algorithme en fonction de $n$ et $k$ et justifier son optimalité.

**Cas d'une liste quelconque** *(mais sans doublons)* Dans ce qui suit, on cherche à déterminer l'élément de rang $k$ d'une liste `L` de longueur $n$ sans doublons.

**2.** **a.** Décrire un algorithme `selection_naif(L, k)` inspiré du tri par sélection pour résoudre le problème. Déterminer sa complexité en fonction de $n$ et $k$.

- **b.** Quelle est la complexité de l'algorithme consistant à trier `L` (de manière optimale) avant de chercher l'élément de rang $k$ ?
- **c.** Comparer l'efficacité de ces deux algorithmes selon les valeurs de $n$ et $k$.

**3.** Une autre manière de résoudre ce problème est inspirée du tri rapide (*QuickSort*).

- **a.** Écrire une fonction `partition(L, pivot)` construisant (en temps optimal) un couple de listes contenant respectivement les éléments de `L` strictement inférieurs et strictement supérieurs à `pivot`.
- **b.** Soit `pivot` un élément quelconque de `L`, `(gauche, droite) = partition(L, pivot)`, et $g =$ `len(gauche)`. Que peut-on conclure concernant l'élément de rang $k$ de `L` selon la valeur de $g$ ?
- **c.** En déduire un algorithme récursif `selection_rapide(L, k)` renvoyant l'élément de rang $k$ de `L`.
- **d.** Quelle est sa complexité dans le meilleur cas ?
- **e.** Quelle est sa complexité dans le pire cas ?
- **f.** Quelle est (au pire) sa complexité si `partition` coupe systématiquement la liste en deux sous-listes de même taille *(à 1 près, naturellement)* ?

*Ce dernier cas, en apparence très favorable, est en fait assez représentatif du comportement de* `selection_rapide`*, qui fait* ***en moyenne*** *moins de deux fois plus de comparaisons – ce qu'on ne cherchera pas à démontrer ici. La même complexité (ou du moins, une complexité du même ordre de grandeur) peut être obtenue* ***dans le pire cas*** *par un algorithme inspiré de* `selection_rapide` *avec un choix de pivot bien particulier (et assez compliqué, ce qui rend cet algorithme moins efficace en pratique que* `selection_rapide`*), étudié à la question suivante.*

**4.** L'algorithme `BFPRT` (d'après les initiales de ses concepteurs) est inspiré de `selection_rapide`, à la différence près que le pivot est spécifiquement choisi comme étant la « médiane des médianes » de sous-listes de `L`, calculée récursivement de la manière suivante :

```
m = len(L) / 5          # découpe de L en m sous-listes de longueur 5 (+ un reste)
M = [ mediane_de_5_elts(L[5*i:5*(i+1)]) for i in range(m) ]       # calcul de la
liste des médianes des sous-listes de longueur 5 à l'aide d'une fonction spécifique
mom = BFPRT(M, m/2)     # calcul récursif de mom, médiane de ces m médianes (Median
Of Medians)
```

**a.** Quelle est la complexité du calcul de `M` ?

**b.** (∗) Dans le cas où $n$ est multiple de 5, donc où $n = 5m$, justifier qu'au moins $\frac{3}{10}$ des éléments de `L` sont inférieurs à `mom`, et au moins $\frac{3}{10}$ lui sont supérieurs.

**c.** En déduire une « inéquation de récurrence » pour le nombre $C(n)$ de comparaisons effectuées par `BFPRT` dans le pire cas pour un appel initial sur une liste de longueur $n$.
*(en négligeant les éléments non pris en compte dans le cas où n n'est pas multiple de 5).*

**d.** (∗) Conclure.

**Cas d'un arbre binaire de recherche** *(toujours sans doublons)* Dans ce qui suit, on cherche à déterminer l'élément de rang $k$ d'un ABR `A` sans doublons, de taille $n$ et de hauteur $h$. Un arbre (non vide) est assimilé à sa racine, et chaque nœud `n` est un objet ayant des champs `etiquette`, `fils_gauche` et `fils_droit`, accessibles en consultation par `n.etiquette`, `n.fils_gauche` et `n.fils_droit`.

**5. a.** Décrire un algorithme optimal pour résoudre le problème dans le cas particulier $k = 1$. Quelle est sa complexité pour un ABR de taille $n$ et hauteur $h$ ? Au pire pour un ABR de taille $n$ ? En moyenne pour un ABR de taille $n$ ? Quid du cas $k = n$ ?

**b.** Décrire un algorithme pour résoudre le problème dans le cas général avec une complexité en $O(n)$ dans tous les cas.

**c.** Exprimer plus précisément cette complexité en fonction du ou des paramètre(s) le(s) plus adapté(s) pour un rang $k$ et un arbre de taille $n$ et hauteur $h$.

Pour améliorer cette complexité, on enrichit la structure d'ABR en ajoutant, dans chaque nœud, la taille du sous-arbre correspondant – c'est-à-dire son nombre total de nœuds y compris le nœud lui-même (donc 1 pour les feuilles, par exemple). Chaque nœud `n` est donc un objet ayant des champs `etiquette`, `taille`, `fils_gauche` et `fils_droit`, accessibles en consultation comme en modification par `n.etiquette`, `n.taille`, etc. On supposera fourni un constructeur `Feuille(e)` qui crée un nouveau nœud d'étiquette `e`, de taille 1, et de fils `None`.

**6. a.** Réécrire l'algorithme d'insertion `insertion_ABR_enrichi(A, x)` pour que le champ `taille` soit correctement tenu à jour. Que devient la complexité ?
*(on supposera pour simplifier, et sans le vérifier, que tous les éléments sont distincts)*

**b.** Si les tailles des deux sous-arbres d'un ABR `A` sont respectivement égales à $g$ et $d$, où l'élément de rang $k$ de `A` se trouve-t-il, en fonction de $k$ ?

**c.** Décrire un algorithme `selection_ABR_enrichi(A, k)` aussi efficace que possible répondant au problème de la sélection pour ces ABR enrichis à l'aide du champ `taille`.

**d.** Analyser la complexité de cet algorithme, au pire et en moyenne pour les arbres de taille $n$.