# DS2 - documents de cours, TD, TP autorisés - Elements de correction

Note : lorsque, dans une question, est demandée la complexité, c'est une fonction de la taille de la donnée qui est attendue. Lorsque c'est le comportement asymptotique, la réponse attendue est soit une notation $\mathcal{O}$, soit une notation $\Theta$ soit une notation $\Omega$.

Vous pourrez utiliser toute fonction vue en cours, TD, TP à la condition expresse de spécifier correctement les entrées et les sorties.

## Exercice 1 : Recherche de mots

***Compétences évaluées : calculer des complexités, réinvestir des algorithmes connus.***

On s'intéresse dans cet exercice à la recherche d'un mot $u$ dans un texte $v$ (on suppose que la longueur de $u$ est plus petite que la longueur de $v$).

**Travail préliminaire.** On rappelle qu'un suffixe d'un mot $v$ de longueur $n$ est un sous-mot débutant à une position $i$, $0 \leq i \leq n$ et contenant toutes les lettres de $v$ de $i$ (compris) jusqu'à la fin du mot. La chaîne vide est un suffixe également.

Par exemple, les suffixes de `"abracadabra"` sont : `"abracadabra"` , `"bracadabra"` , `"racadabra"` , `"acadabra"` , `"cadabra"` , `"adabra"` , `"dabra"` , `"abra"` , `"bra"` , `"ra"` , `"a"` , `""`

**Q 1.1** Combien de suffixes contient une chaîne de caractères de longueur $n$ ?

*Corrigé*

Il existe $n + 1$ suffixes.

**Q 1.2** Ecrire une fonction `tableau_des_suffixes` qui étant donnée une chaine de caractère $v$ de longueur $n$ retourne un tableau trié dans l'ordre lexicographique contenant tous les suffixes de $v$ (à écrire en pseudo code, C ou Caml). Par exemple, pour `"abracadabra"` on obtient le tableau : [ `""` , `"a"` , `"abra"` , `"abracadabra"` , `"acadabra"` , `"adabra"` , `"bra"` , `"bracadabra"` , `"cadabra"` , `"dabra"` , `"ra"` , `"racadabra"` ]

*Corrigé*

```
compare (a,b):
  l1 = longueur de a
  l2 = longueur de b
  i = 0
  tant que i < l1 et i < l2 et a(i) <= b(i) faire
    i++
  fin tant que
  retourner i >= l1

tableau_des_suffixes (v):
  soit n la longueur de v
  soit t un tableau de chaînes de longueur n+1
  pour i allant de 0 à n faire
    t[i] := substring(v,i)
  fin pour
  trier t par le tri rapide en utilisant la fonction compare
  retourner t
```

On s'intéresse à la complexité en temps où l'opération qu'on compte est la comparaison de caractères.

**Q 1.3** Décrire un pire des cas. Quel est le comportement asymptotique en fonction de $n$ ?

*Corrigé*

Pour cette question, comme pour la suivante, il ne suffit pas de compter le coût des comparaisons de chaîne puisque ce sont les comparaisons de caractères qui nous intéressent.
Il faut donc imaginer la comparaison de deux chaînes de caractères écrite avec une boucle tant que (comme dans la fonction `compare` ci-dessus). Cette fonction a une complexité en $\Theta(min(l1, l2))$ dans le pire des cas.
Un pire des cas est lorsque le mot $v$ contient la même lettre à chaque position. Pour comparer deux chaînes il faut alors comparer toutes les lettres de la plus petite des deux chaînes.
Pour le tri rapide, la complexité dans le pire des cas est lorsque le pivot fait une partition en 0 éléments d'un côté et $n-1$ éléments d'un autre. Ce qui arrive ici lorsqu'on construit le tableau des suffixes par ordre des suffixes décroissant sur un mot dont toutes les lettres sont identiques. Dans ce cas la complexité du tri en en $\Theta(n^2)$ comparaisons.
Comme ici la comparaison de deux chaînes est en $\mathcal{O}(n)$, la complexité est alors $\mathcal{O}(n^3)$.

**Q 1.4** Décrire un meilleur des cas. Quel est le comportement asymptotique en fonction de $n$ ?

*Corrigé*

Un meilleur des cas est lorsque le mot $v$ contient une lettre différente à chaque position. Ainsi la comparaison des chaînes de caractères s'arrête dès la comparaison de la première lettre de chaque suffixe. La complexité est alors $\mathcal{O}(n \log n)$.
Le fait qu'avant le tri, le tableau soi déjà trié dans l'odre lexicographique ne constitue pas un meilleur des cas puisque deux suffixes successifs peuvent avoir un préfixe plus long que 1, cela représente donc plus de comparaisons que le cas précédent.

### Algorithme de recherche

**Q 1.5** Proposer un algorithme de recherche d'un mot $u$ de taille $m$ dans un texte $v$ de taille $n$ qui exploite le tableau des suffixes.

*Corrigé*

Comme le tableau est trié, on peut faire une recherche dichotomique.

```
rechercher (u,v) =
  soit t = tablea_des_suffixes (v)
  rechercher par dichotomie u dans t
```

Une autre solution, moins efficace (et qui rapporte donc moins de points) est de faire une recherche linéaire, mais en s'arrêtant dès qu'on a la réponse.

**Q 1.6** Indiquer si il existe ou non un pire des cas, un meilleur des cas. Justifier. Donner la complexité de cet algorithme en fonction de $n$ et $m$.

*Corrigé*

Le pire des cas est celui où le mot n'est pas présent. Le mot $v$ devra être comparé à tous les suffixes de $u$ qui seront envisagés par la recherche dichotomique, soit $\mathcal{O}(m \times \log n)$ comparaisons.
Avec la recherche séquentielle on aura $\mathcal{O}(m \times n)$ comparaisons.
Le meilleur des cas est lorsque le mot $u$ correspond au suffixe se trouvant au milieu du tableau des suffixes. On a alors une seule comparaison de chaînes, on est en $\Theta(m)$.

## <u>Exercice 2</u> : Arbre équivalents

***Compétences évaluées : manipuler des arbres binaires, utiliser des structures de données adéquates pour résoudre un problème.***

Dans cet exercice on manipule des arbres binaires dont les étiquettes sont des entiers. Vous pourrez utiliser toutes les fonctions vues au cours du TP numéro 5 sans les redéfinir (voir annexes du sujet).

**Arbres à couches équivalentes** On dit que deux arbres $A$ et $B$ sont à couches équivalentes si, pour une profondeur donnée $p$, la somme des étiquettes des noeuds à profondeur $p$ est la même dans $A$ et $B$.

Par exemple les arbres $A$ et $B$ ci-dessous sont à couches équivalentes

puisque dans les deux arbres, la somme des valeurs à profondeur 0 est 4, à profondeur 1 est 9, à profondeur 2 est 10.

**Q 2.1** Quelle(s) structure(s) de données complémentaire utiliser pour construire un algorithme qui détermine si deux arbres binaires sont à couches équivalentes ? Justifier.

*Corrigé*

Pour le parcours d'arbre : On peut utiliser une file qui contiendra des couples (arbre, entier). On pourra ainsi faire un parcours en largeur et stocker dans l'entier la profondeur du noeud correspant à l'arbre.
Pour le test d'équivalence : sauvegarder la somme des valeurs à une profondeur dans une table de hachage (ajouter celles du premier, soustraire celles du second).

**Q 2.2** Proposer l'écriture d'un prédicat `a_couches_equivalentes` qui prend en paramètres deux arbres $A$ et $B$ (à écrire en pseudo code, C ou Caml).

*Corrigé*

Dans cet algorithme on parcourt les arbres en largeur grâce à des files. Grâce à la variable `p` on met en place une détection de changement de profondeur afin d'arrêter l'algorithme au plus vite.

```
a_couches_equivalentes (a,b) =
  soit t une table de hachage
  soit deux files fa et fb
  enfiler (a,0) à fa
  enfiler (b,0) à fb
  soit p un entier initialisé à 0
  tant fa n'est pas vide et fb n'est pas vide faire
    (xa,pa) = defiler fa
    (xb,pb) = defiler fb
    si pa et pb sont supérieurs à p alors
      retourner faux si t(p) est différent de 0
      p = min(pa,pb)
    fin si
    ajouter à t(pa) valeur(xa)
    retrancher à t(pb) valeur(xb)
    enfiler (fils_gauche(xa),pa+1) dans fa si fils_gauche(xa) n'est pas vide
    enfiler (fils_droit(xa),pa+1) dans fa si fils_droit(xa) n'est pas vide
    enfiler (fils_gauche(xb),pb+1) dans fb si fils_gauche(xb) n'est pas vide
    enfiler (fils_droit(xb),pb+1) dans fb si fils_droit(xb) n'est pas vide
  fin tant que
  retourner vrai si fa et fb sont vides toutes les deux et que toutes
    les valeurs de t sont à zéro
```

Une autre manière de faire, moins efficace, consiste à disposer d'une fonction calculant la somme des valeurs des étiquettes à une profondeur donnée puis de tester toutes les profondeurs.

**Q 2.3** Y a-t-il un pire des cas ? Si oui le donner, sinon justifier.

*Corrigé*

Oui, il y a un pire des cas si les deux arbres sont identiques. On ne pourra pas arrêter le calcul avant d'avoir parcouru tous les noeuds des deux arbres.
Si on n'avait pas mis en place la détection d'arrêt grâce à la variable `p` alors il n'y aurait pas de meilleur et de pire des cas. La décision étant prise une fois les deux arbres parcourus.

**Q 2.4** Donner le comportement asymptotique dans le pire des cas ou dans le cas général suivant la réponse à la question précédente en fonction du nombre $n$ des noeuds de $A$ et du nombre $m$ des noeuds de $B$.

*Corrigé*

Dans le meilleur des cas on s'arrête dès la racine des arbres. On est donc en $\Omega(1)$. Dans le pire des cas on parcourt une et une seule fois chaque noeud des deux arbres. De plus le nombre de valeurs dans la table ne peut exéder $m + n$. On est en $\mathcal{O}(n + m)$.

**Arbres à feuilles équivalentes** Etant donné un arbre binaire $A$, on associe à chaque feuille $x$ de $A$ la fonction $f(x)$ définie comme la somme des valeurs de noeuds sur la branche menant de la racine $A$ à $x$ (racine et feuille incluses).

Etant donné un arbre binaire $A$, on définit l'ensemble $\mathcal{S}(A) = \{f(x)\text{pour toute feuille } x \text{ de } A\}$.

On dit que deux arbres $A$ et $B$ sont à feuilles équivalentes si $A$ et $B$ ont même nombre de feuilles et que $\mathcal{S}(A) = \mathcal{S}(B)$.

Par exemple, pour les deux arbres $A$ et $B$ ci-dessous

$\mathcal{S}(A) = \{7, 9, 17\}$ et $\mathcal{S}(B) = \{9, 17, 7\}$. Ils sont donc à feuilles équivalentes.

**Q 2.5** Quelle(s) structures(s) de données utiliser pour construire un algorithme qui détermine si deux arbres sont à feuilles équivalentes ? Justifier.

*Corrigé*

Pour le parcours d'arbre : on peut utiliser deux piles pour réaliser un parcours en profondeur.
Pour le test d'équivalence : on peut utiliser deux listes dans lesquelles on ajoutera les valeurs calculées aux feuilles. On peut aussi utiliser des tables de hachage.

**Q 2.6** Proposer l'écriture d'un prédicat `a_feuilles_equivalentes` qui prend en paramètres deux arbres $A$ et $B$ (à écrire en pseudo code, C ou Caml).

*Corrigé*

```
a_feuilles_equivalentes (a,b) =
  la := liste_des_valeurs (a)
  lb := liste_des_valeurs (b)
  trier la
  trier lb
  tant que la n'est pas vide et que lb n'est pas vide alors
    valeur = tete (la)
    tant que la n'est pas vide et tete (la) = valeur alors
      la = reste (la)
    fin tant que
    tant que lb n'est pas vide et tete (lb) = valeur alors
      lb = reste (lb)
    fin tant que
  fin tant que
  retourner vrai si la est vide et lb est vide sinon faux

liste_des_valeurs (a) =
  soit la une liste
  soit pa une pile
  empiler (a,0) dans pa
  tant pa n'est pas vide faire
    (xa,sa) := depiler (pa)
    si xa est une feuille alors
      ajouter à la (sa+valeur(a))
    sinon
    empiler dans pa (fils_gauche(xa),sa+valeur(xa))
    empiler dans pa (fils_droit(xa),sa+valeur(xa))
  fint tant que
  retourner la
```

**Q 2.7** Y a-t-il un pire des cas ? Si oui le donner, sinon justifier.

*Corrigé*

Il ne peut y avoir de pire des cas pour le parcours puisqu'on est obligé de parcourir toutes les feuilles des deux arbres pour décider de l'équivalence.
Pour le test d'équivalence, le pire des cas correspond à deux arbres équivalents puisqu'on sera obligé de parcourir les deux listes.

**Q 2.8** Donner le comportement asymptotique dans le pire des cas ou dans le cas général suivant la réponse à la question précédente en fonction du nombre $n$ des noeuds de $A$ et du nombre $m$ des noeuds de $B$.

*Corrigé*

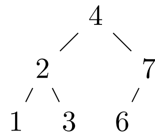
Dans le pire des cas on est en $\Theta(n+m)$ pour le parcours et $\Theta(n)$ pour le test, soit une complexité en $\mathcal{O}(n+m)$ pour l'algorithme.
Dans le meilleur des cas on est en $\Theta(n+m)$ pour le parcours et $\Theta(1)$ pour le test, soit une complexité en $\Omega(n+m)$ pour l'algorithme.
Le tri des listes se fait en $\Theta(n \log n)$ et $\Theta(m\ logm)$ si on utilise le tri fusion.
La conclusion est que l'algorithme est en $\Theta(n+m+m\log m+n\log n)$.

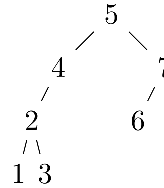
## Exercice 3 : Arbre à insertion par la racine

***Compétences évaluées : manipuler des structures de données connues, réinvestir des raisonnements vus en cours.***

On considère dans cet exercice des arbres binaires de recherche. L'application qui utilisera ces arbres effectue des opérations de recherche d'étiquettes dans l'arbre. Elle présente la particularité de réaliser plus de recherches sur les derniers éléments insérés que sur les premiers. On aura ainsi tout intérêt à ajouter une nouvelle étiquette à la racine.

Ci-dessous l'exemple d'un arbre avant et après insertion de la valeur 5.






L'arbre résultant est toujours un ABR, mail il a été réorganisé.

La procédure d'ajout peut-être décrite ainsi :

```
ajouter (e,a):
  ag, ad = partitionner_en_arbres (a, e)
  retourner nouveau_noeud (e,ag,ad)
```

où `partitionner_en_arbres` crée deux arbres binaires à partir de l'arbre $a$ en fonction de l'étiquette $e$. Pour partionner en arbes, nous commencerons par partitionner en deux listes d'étiquettes.

Vous disposez dans cet exercice de toutes les fonctions de manipulations des arbres binaires vues dans le TP numéro 5 (voir annexes du sujet). N'oubliez pas que pour faciliter le travail, il peut être utile, voire nécessaire d'écrire des sous-fonctions.

**Q 3.1** Ecrire une fonction `partitionner_en_listes` qui prend en entrée un arbre binaire de recherche $a$ et une étiquette $e$ et retourne un couple de listes, la première contenant toutes les étiquettes de $a$ inférieures ou égales à $e$, la seconde les autres.

*Corrigé*

C'est ici un algorithme très classique à écrire qui va réaliser un parcours d'arbre et ranger les étiquettes des noeuds dans les listes. Une version itérative avec pile est donnée. La version récursive se fait simplement grâce à une fonction auxiliaire.

```
partitionner_en_listes (a, e) =
  soit l1 une liste vide
  soit l2 une liste vide
  soit p une pile
  empiler a dans p
  tant que p n'est pas vide faire
    x := depiler(p)
    si x n'est pas un arbre vide faire
      si valeur(x) < e alors
        ajouter_en_tete(valeur(x),l1)
      sinon
        ajouter_en_tete(valeur(x),l2)
      fin si
      empiler fils_gauche(x) dans p
      empiler fils_droit(x) dans p
    fin si
  fin tant que
  retourner (l1,l2)
```

**Q 3.2** Ecrire une fonction `partitionner_en_arbres` qui prend en entrée un arbre binaire de recherche $a$ et une étiquette $e$ et retourne un couple d'arbres binaires, le premier contenant toutes les étiquettes inférieures ou égales $e$ de $a$, le second les autres.

*Corrigé*

C'est immédiat en utilisant la fonction précédente. Il suffit à partir de la liste des étiquettes et de construire un ABR en ajoutant les valeurs des listes. Remarquez qu'on ne demandait pas de construire un ABR particulier.

```
partitionner_en_arbres (a, e) =
  soit (l1, l2) = partitionner_en_listes (a,e)
  retourner (liste_en_abr(l1),liste_en_abr(l2)

liste_en_abr (l) =
  soit a un arbre vide
  tant que l n'est pas vide faire
    ajouter_abr (a, tete(l))
    l := reste(l)
  fin tant que
  retourner a
```

La fonction `ajouter_abr` correspond à la fonction `ajouter` écrite dans le TP5 question 12.

**Q 3.3** Donner un encadrement de la hauteur de l'arbre obtenu après un appel à ajouter sur un arbre $a$ contenant $n$ noeuds et de hauteur $h$. Justifiez très clairement.

*Corrigé*

Un pire des cas est que la valeur ajoutée soit plus petite que toutes les valeurs de l'arbre initial et que la partition en arbres crée un arbre vide est un arbre de hauteur $n$. On aboutit ainsi à un arbre de hauteur maximale $n + 1$.
Le meilleur des cas est la valeur ajoutée soit la médiane des valeurs de l'arbre initial et que la partition en arbres crée deux arbres de hauteur $\log_2(n)$. On aboutit ainsi à un arbre de hauteur minimale $1 + \log_2(n)$.

**Q 3.4** On suppose qu'après l'ajout de $n = 2^p$ étiquettes avec la procédure ajouter, on aboutit à un arbre équilibré. On suppose que la probabilité de recherche de l'étiquette ajoutée à l'étape $i$ est $p(i) = 1 - \frac{1}{2^i}$. Déterminer le coût moyen d'une recherche d'une des étiquettes $n$ présente dans l'arbre.

*Corrigé*

La complexité moyenne d'une recherche fructueuse s'écrit :

$$\sum_{i=1}^{n} p(i)c(i)$$

où $c(i)$ est la complexité de la recherche de la $i$-ème étiquette insérée.
Calculons d'abord le coût de recherche d'une étiquette.
La recherche de la dernière étiquette insérée, soit la $n = 2^p$-ième nécessite un seul test. La recherche des étiquettes numéros $n-1$ et $n-2$ nécessitent 2 tests chacune. On peut donc calculer le nombre de tests nécessaires pour chaque étiquette en fonction de l'étape d'insertion :

| étape | nombre de tests |
|---|---|
| n = $2^p$ | 1 |
| n-1 = $2^p - 1$ | 2 |
| n-2 = $2^p - 2$ | 2 |
| n-3 = $2^p - 3$ | 3 |
| ... | |
| n-6 = $2^p - 6$ | 3 |
| n-7 = $2^p - 7$ | 4 |
| ... | |

On peut en déduire que la recherche de l'étiquette insérée à l'étape $i = n - k$ nécessite $\log_2(k+1)$ tests.
On aboutit ainsi à

$$\sum_{i=1}^{n} p(i)c(i) = \sum_{k=0}^{n-1} p(n-k)c(n-k) = \sum_{k=0}^{n-1} \left(1 - \frac{1}{2^{n-k}}\right) \log_2(k+1) < \sum_{k=0}^{n-1} \log_2(k+1)$$

Maintenant

$$\sum_{k=0}^{n-1} \log_2(k+1) = \sum_{k=1}^{n} \log_2(k) = \log_2(\prod_{k=1}^{n} k) = \log_2 n!$$

Avec des souvenirs mathématiques on a que

$$\log_2 n! \approx n \log_2(n)$$

D'où on déduit que la recherche est en moyenne en $\mathcal{O}(n \log n)$. Mais il ne fallait pas aller si loin pour obtenir tous les points à la question.