DS1 - documents de cours, TD, TP autorisés - durée 2h - Éléments 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 Θ soit une notation Ω (le choix vous incombe).

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

Le barême est donné à titre indicatif.

Exercice 1 : Questions de cours [2 points] (réponses sans justification)

Q 1.1 Si je prouve qu'un algorithme est en $\Omega(n^2)$, peut-il être en $\mathcal{O}(n)$? Répondre par oui ou non. *Corrigé*

> Non bien sûr. Puisque l'algorithme est en $\Omega(n^2)$ il s'exécute dans le meilleur des cas en $\Theta(n^2)$, il ne peut donc pas y avoir d'instance qui s'exécutent en $\mathcal{O}(n)$ puisque $n < n^2$.

Q 1.2 Si je prouve qu'un algorithme est en $\mathcal{O}(n \log n)$, peut-il être en $\Omega(n)$? Répondre par oui ou non. *Corrigé*

> Oui. Puisque l'algorithme est en $\mathcal{O}n \log n$ il s'exécute dans le pire des cas en $\Theta(nlogn)$, rien n'empêche qu'une instance s'exécute en $\Omega(n)$ puisque $n < n \log n$.

Q 1.3 Un algorithme qui utilise la stratégie diviser pour régner s'exécutera d'autant plus vite qu'il divise le problème en un plus grand nombre de sous-problèmes. Répondre par vrai ou faux. *Corrigé*

> Faux. Nous en avons vu un exemple avec la fusion à 4.

Q 1.4 Une liste doublement chaînée permet l'accès au $i-$ème élément en $\Theta(1)$. Répondre par vrai ou faux. *Corrigé*

> Faux.

Exercice 2 : Complexité [6 points]

Compétence évaluée : calculer des complexités, définir pire et meilleur des cas.

On donne l'algorithme suivant, qui prend en entrée un tableau `t` de n entiers.

```
soit s = 0
soit p = 0
pour i allant de 0 à n-1 faire
  s = s + t[i]
  si t[i] est impair alors
    p = p + t[i]
  fin si
fin pour
afficher s et p
```

Q 2.1 Pourquoi n'existe-t-il pas de pire et de meilleur des cas pour le nombre de tests de parité ?

Corrigé

Le test est effectué à chaque tour de boucle. C'est une boucle pour, le nombre de tous est donc constant, fixé à l'avance.

Q 2.2 Donner exactement le nombre de tests de parité réalisés.

Corrigé

$$\sum_{i=0}^{n-1} 1 = n - 1 - 0 + 1 = n$$

Q 2.3 Quel est le comportement asymptotique de l'algorithme pour le nombre de tests de parité ?

Corrigé

L'algorithme est en $\Theta(n)$.

Q 2.4 Donner un pire des cas pour le nombre d'additions.

Corrigé

Si t est tel que tous ses éléments sont impairs : l'instruction `p = p + t[i]` est toujours exécutée.

Q 2.5 Donner exactement le nombre d'additions dans le pire des cas.

Corrigé

$$\sum_{i=0}^{n-1} 2 = 2n$$

Q 2.6 Donner un meilleur des cas pour le nombre d'additions.

Corrigé

Si t est tel que tous ses éléments sont pairs : l'instruction `p = p + t[i]` n'est jamais exécutée.

Q 2.7 Donner exactement le nombre d'additions dans le meilleur des cas.

Corrigé

$$\sum_{i=0}^{n-1} 1 = n$$

Q 2.8 Quel est le comportement asymptotique de l'algorithme pour le nombre d'additions ?

Corrigé

L'algorithme est en $\Theta(n)$.

Exercice 3 : Récursivité [5 points]

Compétence évaluée : établir l'équation de récurrence donnant la complexité d'un algorithme récursif, appliquer le théorème général.

On considère la fonction récursive suivant :

```
# CU: n est un entier positif ou nul
def f(n):
  if n < 2:
    return n
  else:
    m = 0
    for i in range (0,n):
      m = max(g(i),m)
    m = max(m,f(n-1))
    return m
```

où g est une fonction **non récursive** définie par ailleurs.

Dans l'exercice, on s'intéresse à la complexité en temps basée sur le décompte du nombre d'opérations de calcul du maximum entre deux entiers. On notera c_g la fonction de complexité en temps de g.

Q 3.1 Donner l'équation de récurrence de la complexité en temps de la fonction `f` qu'on notera c_f.

Corrigé

$$c_f(n) = \left(\sum_{i=0}^{n-1} c_g(i) + 1\right) + 1 + c_f(n-1)$$

Q 3.2 Proposer une implantation de la fonction g telle que $c_f(n) = \Theta(n^2)$.

Q 3.3 Donner la complexité de g. Justifier.

Q 3.4 Justifier le comportement asymptotique de `f`.

Corrigé

Il suffit que g ne fasse un nombre d'appels à `max` qui soit constant, par exemple 0

```
def g(n):
  return n
```

ainsi $c_g(i) = 0$ et :

$$c_f(n) = \left(\sum_{i=0}^{n-1} 1\right) + 1 + c_f(n-1) = n + 1 + c_f(n-1) = \Theta(n^2)$$

On modifie maintenant l'appel à `f(n-1)` par `f(n//2)` tout en conservant la fonction g de la question précédente.

Q 3.5 Donner le comportement asymptotique de `f`. Justifier. (Si vous n'avez pas réussi la question précédente, vous résoudrez alors l'équation $c_f(n) = n^2 + c_f(n/2)$)

Corrigé

$c_f(n) = n + 1 + c_f(n/2)$. $a = 1, b = 2$. $f(n) = n + 1 = \Omega(n^{\log_2 1+0.5})$. Cas 3 du théorème général. Comme on a $1 \times \frac{n}{2} < 0.9n$, on en déduit $c_f(n) = \Theta(f(n)) = \Theta(n)$.

Exercice 4 : Tri par inversions [9 points]

Compétence évaluée : comprendre un algorithme, évaluer sa complexité, analyser expérimentalement sa complexité.

On présente ci-dessous un nouvel algorithme de tri sur des **permutations**[1] basé sur une opération d'inversion d'une suite d'éléments dans une permutation. Les permutations seront représentées dans des listes natives Python.

La procédure `inverser(l,i,j)` a pour effet de transformer la suite d'éléments :

```
l[i], l[i+1], ... , l[j]
```

en :

```
l[j], ... , l[i+1], l[i]
```

dans la liste `l`.

Pour que l'algorithme fonctionne, il est **impératif** que le premier élément soit la plus petite valeur et le dernier la plus grande valeur de la permutation. On ne considèrera donc que ce type de permutation.

```
def tri_inversion (l):
    n = len(l)
    assert(l[0] == 1)
    assert(l[n-1] == n)
    for i in range (0,n-2):
        j = i + 1
        while not l[j] == l[i] + 1:
            j = j + 1
        inverser(l,i+1,j)
```

Q 4.1 Dérouler l'algorithme sur la permutation `1 7 4 2 6 5 3 8` (donner l'état de la liste `l` à la fin de chaque tour de la boucle pour).

Corrigé

```
i = 0 [1, 2, 4, 7, 6, 5, 3, 8] après inversion de 7 .. 3
i = 1 [1, 2, 3, 5, 6, 7, 4, 8] après inversion de 4 .. 3
i = 2 [1, 2, 3, 4, 7, 6, 5, 8] après inversion de 5 .. 4
i = 3 [1, 2, 3, 4, 5, 6, 7, 8] après inversion de 7 .. 6
i = 4 [1, 2, 3, 4, 5, 6, 7, 8]
i = 5 [1, 2, 3, 4, 5, 6, 7, 8]
```

Q 4.2 Proposer en Python l'écriture de la procédure d'inversion.

Corrigé

```
def inverser (l,i,j):
   for k in range (0,(j-i+1)//2):
     tmp = l[i+k]
     l[i+k] = l[j-k]
     l[j-k] = tmp
```

On suppose que l'accès au $i-$ème élément (en lecture ou écriture) d'une liste l se fait en temps constant (l'obtention de la longueur sera négligé).

1. Une permutation de longueur n est une séquence d'entiers composée des éléments 1 à n, différents deux à deux.

Q 4.3 Indiquer où ajouter les compteurs dans le code ainsi que le valeur des incréments pour compter le nombre d'accès.

Corrigé

On ajoute un incrément de compteur de 4 à chaque tour de la boucle `pour` de la procédure `inverser` (pour l'échange des valeurs), et un incrément de compteur de 2 à chaque tour de la boucle `pour` de la procédure `tri_inversion` pour le test d'égalité du `tant que` (on pourrait aussi ajouter un incrément de compteur de 2 pour les accès réalisés dans les `assert`).

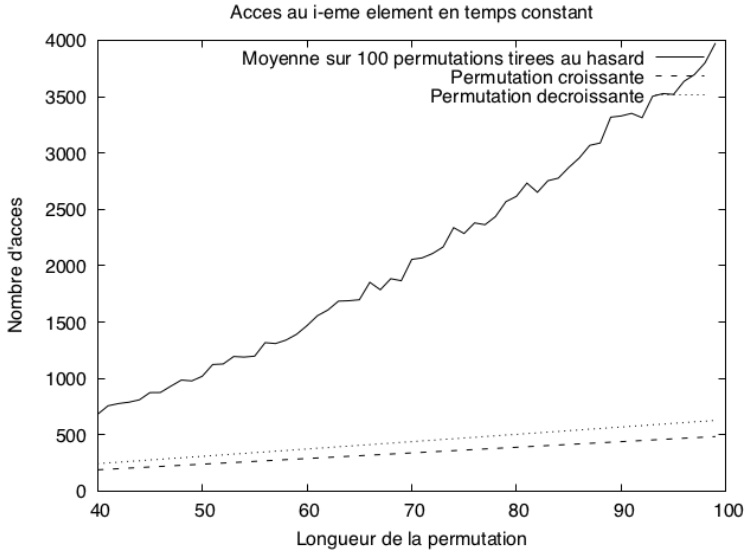
Q 4.4 Quelle est la complexité en nombre d'accès à la liste de la procédure `inverser` ?

Corrigé

$$\sum_{i=0}^{\frac{n}{2}-1} 4 = 4\frac{n}{2} = 2n = \Theta(n)$$

On notera que dans l'écriture d'inverser donnée ici, le cas où n est pair ou impair donne un nombre exact différent.

Le tracé du nombre d'accès sur différents types et tailles de permutations donne ceci :



Q 4.5 Qu'en déduisez-vous ?

Corrigé

- Il existe un pire et un meilleur des cas.
- Le meilleur des cas semble être pour les permutations croissantes et de complexité linéaire.
- L'algorithme semble avoir une complexité en moyenne en $\Theta(n^2)$.

Q 4.6 Expliquer pourquoi le tri sur une permutation « décroissante » de longueur n (donc de la forme $1, n-1, n-2, ..., 2, n$) coûte légèrement plus.

Corrigé

Contrairement à la permutaton croissante, il faudra une fois renverser les $n-2$ éléments compris entre les indices 0 et $n-1$.

Q 4.7 Décrire un pire des cas.

Corrigé

Le pire des cas est de devoir renverser $n-2$ éléments à la première étape, $n-3$ éléments à la seconde étape et ainsi de suite : on ne place correctement qu'un seul élément à chaque étape. Cela arrive quand la permutation est de la forme $(1, 3, 5, ..., 4, 2, n)$.

Q 4.8 Donner la complexité en nombre d'accès à la liste de l'algorithme de tri dans le pire des cas. Justifier.

Corrigé

On renverse $n-2$ éléments à l'itération 0, $n-3$ à l'itération 1, ..., $n-i-2$ à l'itération i. Le renversement de m éléments requiert $2m$ accès. A chaque itération la comparaison du tant que requiert 2 accès.
Soit

$$c(n) \leq \sum_{i=0}^{n-3} \left(4\frac{(n-i-2)}{2} + 2 \right) = (n-2)(n+1) = \Theta(n^2)$$

$\leq$ à cause de la parité de la longueur de l'intervalle renversé.

On suppose maintenant que l'accès au $i-$ème élément d'une liste l se fait en temps linéaire.

Q 4.9 Quelle est la complexité en nombre d'accès à la liste de la procédure `inverser` ?

Corrigé

Chaque élément k entre i et j est accédé 2 fois. On a donc

$$\sum_{k=i}^{j} 2k = 2\left(\sum_{k=0}^{j} k - \sum_{k=0}^{i-1} k \right) = 2\left(\frac{j(j+1)}{2} - \frac{(i-1)(i)}{2} \right) = \Theta(j^2 - i^2)$$

Q 4.10 Donner les nouveaux incréments de compteur et estimez le nombre d'accès sur une permutation « décroissante » de longueur n (donc de la forme $1, n-1, n-2, ..., 2, n$).

Corrigé

$2(i+k) + 2(j-k)$ pour inverser, et $i+j$ pour le tri.

On rappelle que :

$$\sum_{i=0}^{n} 1 = n+1, \sum_{i=0}^{n} i = \frac{n(n+1)}{2}, \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$