université PARIS DIDEROT PARIS 7

L2 Informatique & Math-Info

Nom :

Prénom :

Groupe :

EA4 – Éléments d'algorithmique
Partiel du 22 mars 2017 – Sujet A
Durée : 1h45

Aucun document autorisé
Appareils électroniques éteints et rangés

Cet énoncé comporte 7 exercices indépendants. Ils ne sont absolument pas classés par ordre de difficulté, n'hésitez pas à les traiter dans l'ordre de votre choix. Les mentions de temps ne sont que des indications... mais devraient refléter à la fois les durées relatives des exercices et leur barème. Gardez du temps pour le dernier exercice !

Exercice 1 : *(5 min)*

Compléter le tableau ci-dessous avec les ordres de grandeur des complexités en temps des différents algorithmes de tri étudiés en cours, en fonction du nombre n d'éléments à trier.

	pire cas	meilleur cas	en moyenne
tri par sélection			
tri par fusion			
tri à bulles			
tri par insertion			
tri rapide			

Exercice 2 : *(10 min)*

Cocher ***toutes*** les assertions exactes.

		$f \in \Omega(g)$	$f \in O(g)$	$f \in \Theta(g)$	$f \notin \Theta(g)$
$f = 3(n^2-1)^2$	$g = 4n^3 + 5n$	☐	☐	☐	☐
$f = \log(3(n^2-1)^2)$	$g = \log(4n^3 + 5n)$	☐	☐	☐	☐
$f = 4n^3$	$g = 3^{(n+4)}$	☐	☐	☐	☐
$f = \log(4n^3)$	$g = 4(\log n)^3$	☐	☐	☐	☐
$f = 3^{(n+4)}$	$g = 3^n$	☐	☐	☐	☐
$f = 3^{(n+4)}$	$g = 3^{(4n)}$	☐	☐	☐	☐
$f = \log(n^4)$	$g = \sqrt[3]{n}$	☐	☐	☐	☐
$f = \log(4^n)$	$g = \sqrt[3]{n}$	☐	☐	☐	☐

(note : log *désigne le logarithme en base 2)*

Exercice 3 : *(10-15 min)*

Pour chacun des algorithmes suivants, donner une relation de récurrence satisfaite par le nombre $A_i(n)$ d'additions effectuées pour une entrée de taille n, et en déduire (sans démonstration) l'ordre de grandeur de $A_i(n)$.

```
def somme_1(T) :
  if len(T) == 0 : return 0
  else : return T[0] + somme_1(T[1:])
```

```
def somme_2(T) :
  if len(T) <= 1 : return 1
  else :
    m = len(T)//2
    return somme_2(T[:m]) + somme_2(T[m:])
```

```
def somme_3(T) :
  if len(T) <= 2 : return 1
  else :
    m = len(T)//3
    return somme_3(T[:m]) + 1
```

```
def somme_4(T) :
  if len(T) <= 1 : return 1
  else :
    tmp = somme_4(T[:-1])
    for elt in T : tmp = tmp + elt
    return tmp
```

```
def somme_5(T) :
  if len(T) <= 1 : return 1
  else :
    m = len(T)//2
    tmp = somme_5(T[:m]) + somme_5(T[m:])
    for elt in T : tmp = tmp + elt
    return tmp
```

Exercice 4 : *(10-15 min)*

On considère la suite définie de la manière suivante :

$$F_n = \begin{cases} 1 & \text{si } n < 3 \\ 3F_{n-1} + 2F_{n-3} & \text{sinon.} \end{cases}$$

Proposer un algorithme efficace pour calculer le terme d'ordre n de la suite.

Combien d'opérations arithmétiques sur des entiers cet algorithme effectue-t-il pour calculer F_n ?

Est-ce une mesure pertinente de sa complexité en temps ?

Exercice 5 : *(15-20 min)*

Soit `T` le tableau suivant :

81	141	145	41	123	117	27	83

Appliquer l'algorithme de tri fusion (*MergeSort*) à `T`. Combien de comparaisons d'éléments sont effectuées (exactement) ?

Déterminer l'élément de rang 5 dans `T` en appliquant l'algorithme de sélection rapide (*QuickSelect*), dans sa version avec `T[0]` comme pivot. Combien de comparaisons d'éléments sont effectuées (exactement) ?

Appliquer l'algorithme de tri par base (*RadixSort*) à `T` (en raisonnant en base 10).

Exercice 6 : *(15 min)*

On s'intéresse au problème suivant : étant donné une liste `L` de nombres (non nécessairement entiers) de longueur n, déterminer le *vainqueur* de `L`, *i.e.* l'élément de `L` qui y apparaît le plus de fois (ou l'un quelconque d'entre eux, en cas d'égalité).

Décrire un algorithme naïf permettant de résoudre ce problème sans modifier la liste `L`.

Quel est l'ordre de grandeur de la complexité (en temps) de cet algorithme ?

Comment résoudre ce problème avec une complexité en temps strictement meilleure ? Laquelle ?

Exercice 7 : *(20 min)*

On dit qu'un tableau `T` de n entiers est une *montagne* s'il est constitué d'une première partie strictement croissante, suivie d'une deuxième strictement décroissante, chacune pouvant éventuellement être vide ; autrement dit, `T` est une montagne s'il est strictement croissant ou décroissant, ou s'il existe un certain $m \in [\![1, n-2]\!]$ tel que :

$$\texttt{T[0]} < \texttt{T[1]} < \ldots < \texttt{T[m]} \quad \text{et} \quad \texttt{T[m]} > \texttt{T[m+1]} > \ldots > \texttt{T[n-1]}.$$

Proposer un algorithme `est_une_montagne(T)` *de complexité optimale*[1] qui teste si `T` est une montagne. Justifier rapidement sa correction et sa complexité.

1. c'est-à-dire l'algorithme qui vous semble le plus efficace ; il ne vous est pas demandé de prouver son optimalité.

On suppose maintenant que `T` est une montagne.

Proposer un algorithme `pied(T)` *de complexité optimale*[1] qui renvoie le plus petit élément de `T`. Justifier sa correction et sa complexité.

Étant donné une position i de `T`, comment tester *en temps constant* si $i < m$, où m est la position (inconnue *a priori*) du maximum de `T` ?

En déduire un algorithme `sommet(T)` *de complexité optimale*[1] qui renvoie le plus grand élément de `T`. Justifier rapidement sa correction et sa complexité.

Proposer un algorithme `nivelle(T)` *de complexité optimale*[1] qui renvoie un tableau trié contenant les mêmes éléments que `T`. Justifier rapidement sa correction et sa complexité.

(***bonus***) Justifier l'optimalité des algorithmes proposés.