Nom :

Prénom :

Groupe :

L2 Informatique & Math-Info

EA4 – Éléments d'algorithmique
Partiel du 6 mars 2019 – Sujet D

Durée : 2 heures

Aucun document autorisé
Appareils électroniques éteints et rangés

Le sujet est (trop) long, il en sera tenu compte dans la notation. Les 6 exercices sont indépendants et ne sont absolument pas classés par ordre de difficulté, n'hésitez pas à les traiter dans l'ordre de votre choix. Gardez du temps pour le dernier exercice !

Exercice 1 :

Décrire le déroulement de l'algorithme de Karatsuba pour calculer le produit 1201×2011, en représentant en particulier l'arbre des appels récursifs. Prendre le produit de chiffres comme cas de base.

Exercice 2 :

Soit `T` le tableau suivant :

13	19	27	62	8	51	5	34

Appliquer l'algorithme de tri fusion (*MergeSort*) à `T`. Combien de comparaisons d'éléments sont effectuées ? *(on demande un décompte exact)*

Appliquer l'algorithme de tri rapide (*QuickSort*) à `T` dans sa version ***en place***, avec `T[0]` comme pivot. Combien de comparaisons d'éléments sont effectuées ? *(on demande un décompte exact)*

Exercice 3 :

Pour chacun des algorithmes `foo_`i`_` 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 foo_1_(T) :
  return 0 if len(T) == 0 else foo_1_(T[3:]) + 1
```

```
def foo_2_(T) :
  m = len(T)//2
  return 0 if len(T) == 0 else foo_2_(T[:m]) + 1
```

```
def foo_3_(T) :
  return 0 if len(T) == 0 else foo_1_(T) + foo_3_(T[2:])
```

```
def foo_4_(T) :
  m = len(T)//2
  return 0 if len(T) == 0 else foo_1_(T) + foo_4_(T[:m]) + foo_4_(T[m:])
```

Exercice 4 :

On s'intéresse au problème suivant : étant donné une liste `L` de nombres (non nécessairement entiers) de longueur n, déterminer le nombre de singletons, *i.e.* d'éléments apparaissant exactement une fois dans `L`.

Décrire un algorithme naïf permettant de résoudre ce problème sans modifier la liste `L`, et avec mémoire auxiliaire constante.

Quel est l'ordre de grandeur de la complexité en temps de cet algorithme ? Justifier.

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

Exercice 5 :

Classer les fonctions suivantes en fonction de leur ordre de grandeur dans les classes Θ_1 à Θ_7 : les fonctions appartiennent à la même classe Θ_i si et seulement si elles sont du même ordre de grandeur, et les classes Θ_i sont rangées en ordre croissant.
Liste des fonctions à traiter (où log désigne le logarithme en base 2) :

$$n^3 + 2n^2, \quad n^3 + n^4, \quad n^3 \times 2^n, \quad n^3 + \log n, \quad n^2 \times 4^{\log n},$$

$$\log(\sqrt{n}), \quad \log(n^3), \quad (\log n)^3, \quad 2^{n-1}, \quad 2^{2n}, \quad 2^{n+4}, \quad 4^{n+1}$$

Θ_1	Θ_2	Θ_3	Θ_4	Θ_5	Θ_6	Θ_7

Exercice 6 :

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.