L2 Informatique

Nom :

Prénom :

Numéro :

Groupe :

EA4 – Éléments d'algorithmique
Contrôle du 18 mars 2022
Durée : 1 heure 30

Aucun document autorisé
Appareils électroniques éteints et rangés

Les 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.
Il n'est pas nécessaire de réécrire les algorithmes du cours utilisés sans modification.

Exercice 1 :

On considère l'algorithme suivant :

```
def F(n) :
    return 0 if n < 3 else 1 if n == 3 else 2 * F(n-1) + 5 * F(n-3) + 1
```

Soit $A(n)$ le nombre d'opérations arithmétiques sur des entiers effectuées lors de l'exécution de `F(n)`. Donner une définition récursive de $A(n)$.

En déduire que $A(n)$ est croissante, puis que $A(n) \in \Omega(2^{n/3})$.

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.

Quel est l'ordre de grandeur de la complexité en temps de `Fd(n)` ? et de sa complexité en espace ?

Proposer un algorithme `Fe(n)` calculant la même valeur de manière encore plus efficace.

Évaluer la complexité en temps de `Fe(n)`.

Exercice 2 :

On s'intéresse au problème suivant : étant donné une liste `L` de nombres (non nécessairement entiers) de longueur $n \geqslant 2$, déterminer la *maille* de `L`, *i.e.* la plus petite différence (en valeur absolue) entre deux éléments de `L` (éléments à des positions différentes : en particulier, la maille est nulle si et seulement si `L` contient un doublon).

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 3 :

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 indice $m \in [\![1, n-2]\!]$ 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.

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.

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

Étant donné un indice i, comment tester *en temps constant* si $i < m$, où m est l'indice (inconnu *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.

Exercice 4 :

On considère l'algorithme `foo` ci-dessous. Décrire son déroulement pour chacun des deux appels.

```
def foo(T, deb = 0, fin = None) :
   if fin == None : fin = len(T)
   if fin - deb < 2 : return T
   if fin - deb == 2 :
      if T[deb] > T[deb+1] :
         T[deb], T[deb+1] = T[deb+1], T[deb]
      return T
   un_tiers = (fin - deb) // 3
   b1, b2 = deb + un_tiers, fin - un_tiers
   foo(T, deb, b2)
   foo(T, b1, fin)
   foo(T, deb, b2)
   return T
```

(1) de manière détaillée
`foo([4, 3, 2, 1])`

(2) sans détailler les appels pour `fin - deb < 5`
`foo([6, 5, 4, 3, 2, 1])`

Émettre une conjecture $\mathcal{C}$ sur l'état de `T` après exécution de `foo`.

On souhaite démontrer $\mathcal{C}$. Soit P_n la propriété « $\mathcal{C}$ est vraie pour tout tableau de taille au plus n ».
Soit $n > 2$ un entier tel que P_n est vraie, et considérons un tableau `T` de taille $n+1$.
Comparer les tailles des trois sous-tableaux `T0 = T[deb:b1]`, `T1 = T[b1:b2]` et `T2 = T[b2:fin]`.

Que peut-on dire de ces trois sous-tableaux après le premier appel `foo(T, deb, b2)` ?

Après l'appel `foo(T, b1, fin)` ?

Après le deuxième appel `foo(T, deb, b2)` ?

Conclure.

Soit $S(n)$ le nombre de comparaisons effectuées lors d'un appel à `foo` sur un tableau de taille n.
Donner une définition récursive de $S(n)$.

Comparer la complexité de `foo` à celle des algorithmes classiques résolvant le même problème.