Examen partiel

Durée : 2h.
Tous documents et matériels électroniques interdits.
Le sujet comporte deux exercices indépendants. N'hésitez pas à admettre le résultat d'une question pour aborder les questions qui suivent.

Exercice 1. Palindromes.
Une liste `L` de longueur n est un **palindrome** si pour tout $i \in \{0, ..., n-1\}$, on a $\texttt{L}[i] = \texttt{L}[n-1-i]$.

1. Écrire une fonction récursive qui permet de déterminer si une liste est un palindrome. On se basera sur l'idée suivante : les listes de longueur $\leqslant 1$ sont des palindromes, et si `L` est une liste de longueur n, alors `L` est un palindrome ssi $\texttt{L}[0] = \texttt{L}[n-1]$ et $\texttt{L}[1 : n-1]$ est un palindrome. On demande que la fonction soit sous forme récursive terminale.
2. Donner en justifiant une borne de complexité en O de votre fonction en fonction de la longueur n de la liste en entrée.
3. Montrer que la borne que vous avez donnée est optimale.
4. Écrire une version itérative (c'est-à-dire non récursive) de cette fonction.
5. On veut utiliser l'algorithme précédent sur des listes de longueur 10000. Quelle implémentation (récursive ou itérative) serait la plus adaptée ? On justifiera la réponse.

Exercice 2. Tri de Shell (shellsort).
Dans cet exercice, on s'intéresse au tri de Shell, qui est une manière d'améliorer le tri par insertion. On rappelle que le tri par insertion consiste à mettre au fur et à mesure les éléments de la liste à la bonne place parmi les éléments précédents. Voici une implémentation en Python du tri par insertion.

```
def triInsertion(L):
  for i in range(1,len(L)):
    j = i     # On va mettre L[i] a la bonne place parmi L[0]<=...<=L[i-1]
    while j >= 1 and L[j-1] > L[j]: # tant que l'element est trop petit
      (L[j],L[j-1]) = (L[j-1],L[j]) # on l'echange avec l'element precedent
      j = j-1
  return L
```

1. Donner l'invariant de boucle sur la boucle en `i` qui permet de montrer que le tri par insertion trie effectivement la liste `L` donnée en entrée.
2. Donner en justifiant la complexité du tri par insertion.

Les **tris de Shell** consistent en une amélioration du tri par insertion en effectuant tout d'abord des sauts plus grands afin d'amener les éléments à la bonne place plus rapidement. Ils se basent sur la fonction auxiliaire suivante :

```
def triInsertionSaut(L,k):
  for i in range(k,len(L)):
    j = i
    while j >= k and L[j-k] > L[j]:
      (L[j],L[j-k]) = (L[j-k],L[j])
      j = j - k
  return L
```

Étant donné un entier $k \geqslant 1$, une liste `L` de longueur n est dite k-**triée** si pour tout $i \in \{0, \ldots, n-1\}$ tel que $i + k \leqslant n-1$, on a $\texttt{L}[i] \leqslant \texttt{L}[i+k]$.

3. Que peut-on dire d'une liste 1-triée ?
4. Que renvoie `triInsertionSaut(L,2)` si `L = [7,3,6,2,5,1,4,0]` ?
5. Montrer que `triInsertionSaut(L,k)` renvoie une version k-triée de la liste `L`. On donnera notamment l'invariant de boucle sur la boucle en `i`.
6. Montrer qu'il existe une constante $C > 0$ telle que `triInsertionSaut(L,k)` prenne au plus $\frac{Cn^2}{k}$ étapes de calcul.

Le tri de Shell, dans sa version originelle, fonctionne de la manière suivante sur une liste `L` de taille n : soit $k \in \mathbb{N}$ le plus grand entier tel que $2^k < n$, alors pour l allant de k à 0 (inclus) par pas de -1, on effectue `triInsertionSaut(L,` 2^l`)`.

7. Écrire une fonction `plusGrandePuissance(n)` qui renvoie le plus grand entier k tel que $2^k < n$.
8. Écrire une fonction `triShell(L)` qui trie `L` en utilisant le tri de Shell.
9. Montrer que la fonction retourne bien une liste triée.
10. Montrer que la complexité est en $O(n^2)$, où n est la taille de la liste.
11. Montrer que la borne de complexité de la question précédente est optimale. On pourra s'inspirer de la liste de la question 4.

Le tri de Shell peut être grandement amélioré en utilisant une suite de sauts plus efficace, comme on le verra dans le DM. On sait notamment qu'avec une suite de sauts de la forme $2^l - 1$ *(au lieu de* 2^l*), la complexité est en* $O(n^{3/2})$ *!*