Informatique de Tronc Commun

Devoir surveillé n°3 (v4)

La calculatrice n'est pas autorisée

durée : 2 h

Toutes les réponses sont à donner sur la copie.

Exercice 1. Algorithme glouton : stratégie de concours.

Lors d'un devoir et encore plus lors d'une épreuve de concours, le but est de répondre juste à un maximum de questions afin d'obtenir au global une note maximale. Nous allons chercher à optimiser cette note maximale en choisissant judicieusement l'ordre de traitement des exercices d'un devoir.

Soit un sujet de devoir, noté sur 20, comportant six exercices et devant être traité en deux heures. Pour chaque exercice, sont donnés un barème et deux durées de traitement : la durée optimale et la durée raisonnable. La durée optimale est le temps à respecter pour traiter l'exercice si l'on souhaite aborder le sujet dans son intégralité. Seul un candidat très bien préparé est capable de finir l'exercice en un temps si cours. La durée raisonnable est le temps nécessaire à un candidat sérieux pour finir l'exercice. Malheureusement, il est impossible de faire tous les exercices du sujet dans ce cas.

Partant du principe qu'il ne pourra finir le sujet, le candidat se demande alors dans quel ordre de priorité traiter les exercices afin d'obtenir une note maximale.

On donne, sous forme d'un dictionnaire dont les clés sont le numéro (entier) de l'exercice et dont les valeurs sont des tuples contenant, dans cet ordre, les durées optimale et raisonnable en minutes, ainsi que le barème de l'exercice :

```
devoir = {1: (20, 30, 3), 2: (20, 40, 3), 3: (10, 20, 2), 4: (40, 55, 7),
          5: (10, 15, 2), 6: (20, 30, 3)}
```

Mise en œuvre de l'algorithme glouton

On souhaite associer à chaque exercice une pondération appelée « *efficacité* » qui est le rapport du nombre de points rapporté par l'exercice par la durée raisonnable de traitement de celui-ci.

Un algorithme intuitif consiste alors à traiter en priorité les exercices ayant l'efficacité la plus élevée.

1. Par quelle expression accède-t-on aux caractéristiques (durée optimale, durée raisonnable, barème) de l'exercice 4 à partir du dictionnaire `devoir` ?
2. Par le calcul de quelle expression peut-on obtenir l'efficacité de l'exercice numéro 2 à partir du dictionnaire `devoir` ?
3. Écrire un script construisant, à partir du dictionnaire `devoir`, à l'aide d'une boucle, le dictionnaire `d_eff` dont les clés sont les numéros d'exercices et les valeurs associées l'efficacité (de type `float`) de l'exercice.
   Sur l'exemple, le dictionnaire `d_eff` sera le dictionnaire `{1: 0.1, 2: 0.075, ..., 6: 0.1}`.

On donne l'instruction suivante : `sorted(d.items(), key = lambda x : x[1], reverse=True)` qui renvoie une liste de tuples `(clé, valeur)` du dictionnaire `d` ordonnés dans le sens décroissant des valeurs.

Exemple d'appels à la fonction :

```
>>> d = {"A": 0, "B": 4, "C": 2, "D": 6}
>>> sorted(d.items(), key = lambda x : x[1], reverse=True)
[('D', 6), ('B', 4), ('C', 2), ('A', 0)]
```

4. Utiliser cette instruction pour construire, à partir du dictionnaire `d_eff`, une liste `L_eff` dont les éléments sont les six couples (numéro d'exercice, efficacité), et ordonnés par efficacité décroissante.
5. Écrire un script, permettant d'obtenir, à partir de la liste `L_eff`, la liste `L_ex` des numéros des exercices qu'il est possible de traiter en moins de deux heures, si l'on commence par traiter les exercices de plus grande efficacité.
   On supposera que si des exercices ont la même efficacité, ils seront ordonnés suivant leur numéro.

Le script devra aussi calculer le temps, `T`, nécessaire pour traiter les exercices choisis.

6. Compléter le script de la question **5.** par le calcul du nombre de points, `Pts`, obtenus avec le choix d'exercices donné par `L_ex`.

7. Donner les valeurs des variables `L_ex` , `T` et `Pts` à la fin de l'exécution du script pour le devoir considéré en introduction.

On se propose de tester un autre choix d'exercice, par exemple de traiter les exercices 2, 3 et 4.

8. Quels sont alors la durée de traitement (à une vitesse raisonnable) et le nombre total de points obtenus ? L'algorithme glouton proposé renvoie-t-il la solution optimale ?

Exercice 2. Algorithme d'addition, de multiplication de deux entiers en base 2

On rappelle que l'écriture canonique en base 2 d'un entier n peut se noter $(c_{m-1} \dots c_1 c_0)_2$, où m est le nombre de de chiffres de l'écriture en base 2 de n et les c_i $(0 \leq i < m)$ sont les chiffres de cette écriture, sont égaux à 0 ou à 1, sauf c_{m-1} qui nécessairement vaut 1.

Pour tout entier naturel $n > 0$, cette écriture, correspond à la décomposition suivante de n, et est unique :

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

1. Calculer la valeur de l'entier n dont l'écriture en base 2 est $(110111)_2$.

2. Quels sont, respectivement, le plus petit entier naturel dont l'écriture binaire canonique comporte exactement 4 chiffres ? le plus grand ?

3. Généraliser la réponse à la question **2.** pour des entiers dont l'écriture binaire canonique comporte exactement m chiffres ($m \geq 1$).

Dans les questions **4.a** et **4.b**, on considère des entiers dont l'écriture binaire est codée sur 4 bits, sous la forme de listes à 4 éléments valant 0 ou 1, quitte à la compléter par des zéros à gauche.

Ainsi, un entier n dont l'écriture binaire canonique est $(101)_2$, sera codé par la liste :

$$L = [0, 1, 0, 1].$$

4. On donne, en langage naturel, l'algorithme d'addition de deux entiers, n_1 et n_2, codés sur 4 bits par des listes de zéros, $L_1 = [b_1, b_2, b_3, b_4]$ et $L_2 = [b'_1, b'_2, b'_3, b'_4]$.

- Initialiser une variable r (retenue), à zéro.
- Construire une liste L de quatre zéros, notée ici $L = [d_1, d_2, d_3, d_4]$
- Pour i variant de 0 à 3 :
  - calculer la somme, s, égale à $b_{4-i} + b'_{4-i} + r$; affecter à r la valeur 0 ;
  - si cette somme, s, est supérieure ou égale à 2, affecter à r la valeur 1 et à remplacer la valeur de s par la valeur du reste dans la division euclidienne de s par 2 ;
  - affecter à d_{4-i} la valeur de s.

a. Appliquer cet algorithme pour effectuer l'addition des entiers n_1 et n_2, dont l'écriture binaire est $(101)_2$ et $(11)_2$, pour cela, on se contentera, sur la copie, de compléter le tableau suivant, en recopiant ses trois dernières colonnes et en y indiquant les valeurs des variables r et L à la fin de chaque itération d'indice i.

	i	r	L
Avant toute itération	-	0	[0, 0, 0, 0]
Fin de l'itération d'indice $i =$	0		
Fin de l'itération d'indice $i =$	1		
Fin de l'itération d'indice $i =$	2		
Fin de l'itération d'indice $i =$	3		

b. Vérifier que le résultat obtenu est correct en donnant la valeur des entiers n_1 et n_2 et l'entier codé par la liste L en fin d'exécution.

c. Implémenter l'algorithme d'addition précédent, pour des entiers codés sur un même nombre N de bits, sous la forme d'une fonction `addition(L1, L2)`, prenant en entrée deux listes de même longueur, et renvoyant une liste L, de même longueur que L_1 et L_2, et codant l'entier égal à la somme des entiers codés par L_1 et L_2. On ajoutera en première ligne du corps de la fonction une vérification, utilisant `assert`, du fait que les listes en entrée sont de même longueur.

Note : si le codage en binaire de la somme de deux entiers nécessite plus de bits que la longueur des listes en entrée, le bit supplémentaire sera ignoré (phénomène de dépassement ou *overflow*).

d. Évaluer la complexité de cet algorithme d'addition en fonction de N, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

On s'intéresse maintenant à un algorithme de multiplication.
Si l'on considère l'écriture binaire canonique d'un entier $n > 0$, notée $(c_{m-1} \dots c_1 c_0)_2$, correspondant à la décomposition suivante de n :

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

alors,

$$2n = 2\left(\sum_{i=0}^{m-1} c_i 2^i\right) = 2\left(\sum_{i=0}^{m-1} c_i 2^i\right) = \sum_{i=0}^{m-1} c_i 2^{i+1} = \sum_{i=1}^{m} c_{i-1} 2^i + 0.2^0,$$

de sorte que $2n$ s'écrit $2n = \sum_{i=0}^{m} c'_i 2^i$, avec pour tout i compris entre 1 et m, $c'_i = c_{i-1}$ et $c'_0 = 0$, d'où l'on déduit que l'écriture binaire de $2n$ est

$$(c'_m \dots c'_1 c'_0)_2 = (c_{m-1} \dots c_1 c_0 0)_2.$$

On en déduit que si un entier n est codé sur N bits par une liste L de zéros et de uns, alors son double, $2n$, est codé sur N bits par la même liste, dans laquelle les valeurs ont été décalées d'un rang vers la gauche, le bit le plus à droite valant, lui, zéro.
Comme pour l'addition, on aura un possible phénomène d'*overflow* : le bit non nul le plus à gauche dans l'écriture de N, sera ignoré si l'écriture binaire canonique de $2n$ comporte $N + 1$ chiffres.

5.

a. Implémenter la multiplication par 2 d'un entier n, en base 2, sous la forme d'une fonction `multiplie2(L)`, prenant en entrée l'écriture binaire de n sous la forme d'une liste L de longueur N et renvoyant sous la forme d'une liste de zéros et de uns, une nouvelle liste de longueur N codant l'écriture binaire sur N bits de $2n$.

Exemples d'appels à la fonction :

```
>>> multiplie2([0, 0, 0, 1, 1, 1, 0, 1])
([0, 0, 1, 1, 1, 0, 1, 0])
```

```
>>> multiplie2([1, 1, 1, 1])
[1, 1, 1, 0])
```

Note : comme on le constate sur le dernier exemple, le bit le plus à gauche de la liste en entrée, appelé *bit de poids fort*, est « perdu » lors de cette opération, qu'il soit égal à zéro ou à un (phénomène d'*overflow*).

b. Évaluer la complexité, en fonction de N, de cet algorithme de multiplication par 2, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

On donne l'algorithme pour la multiplication de deux entiers sous la forme de la fonction suivante `mult(L1, L2)`, prenant en entrée deux listes de même longueur, L_1 et L_2, codant deux entiers sur un même nombre N de bits, et renvoyant la liste, de même longueur que L_1 et L_2, codant l'entier égal au produit des entiers codés par L_1 et L_2.

```
def mult(L1, L2):
    Lf = [0] * len(L1)
    L  = L2.copy()
    for i in range(len(L2)):
        if L2[len(L2) - 1 - i] == 1:
            Lf = addition(Lf, L)
        L = multiplie2(L)
    return Lf
```

c. Évaluer la complexité de cet algorithme de multiplication en fonction de N, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

Exercice 3. Recherche d'une valeur dans une liste

Recherche séquentielle dans une liste quelconque

1. Écrire une fonction `recherche1` qui prend en argument une liste `L` et une valeur `v` et qui renvoie `True` si la valeur est présente dans la liste `L` et `False` si ce n'est pas le cas.

Exemples d'appels à la fonction :

```
>>> recherche1([1, 10, 5, -6, 0], 5)
True
>>> recherche1(['Ghislain', 'Jebril', 'Laurène'], 'Nissrine')
False
```

2. Évaluer la complexité d'un appel à la fonction `recherche1` en fonction de la longueur n de la liste en entrée, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

3. Écrire une fonction `recherche2` qui prend en argument une liste `L` et une valeur `v` et qui renvoie la liste des indices des occurrences de la valeur `v` dans la liste `L` .
La liste renvoyée sera une liste vide si la valeur n'est pas présente dans la liste.

Exemples d'appels à la fonction :

```
>>> recherche2([1, 10, 5, -6, 0], 4)
[]
>>> recherche2(['A', 'B', 'C', 'E', 'D', 'C', 'D', 'C'], 'C')
[2, 5, 7]
```

Dans le second exemple, en effet, le caractère `'C'` est présent dans la liste, aux positions 2, 5 et 7.

4. Évaluer la complexité d'un appel à la fonction `recherche2` en fonction de la longueur n de la liste en entrée, en distinguant s'ils existent, un meilleur et un pire cas, que l'on caractérisera et dont on donnera les complexités respectives.

Recherche par dichotomie

La fonction `recherche3` définie ci-dessous permet une recherche par dichotomie d'un élément dans une liste triée. Cette fonction prend deux arguments, le premier est une liste dont les éléments sont classés dans l'ordre croissant et le second la valeur à rechercher dans la liste.

```
1.  def recherche3(L,v):
2.      bas, haut = 0, len(L) - 1
3.      while bas <= haut:
4.          milieu = (bas + haut) // 2    #détermination de la valeur de milieu
5.          print(bas, milieu, haut)      #affichage des valeurs bas, milieu, haut
6.          if L[milieu] == v:     #
7.              return True        #valeur trouvée
8.          elif L[milieu] < v:    #valeur v supérieure strictement à L[milieu]
9.              bas = milieu + 1   #
10.         else:                  #valeur v inférieure à L[milieu]
11.             haut = milieu - 1  #
12.     return False               #valeur non trouvée dans la liste
```

On définit une liste `L` comme suit :

```
>>> L = [10, 20, 30, 50, 60, 80, 90, 100, 120]
```

L'exécution de l'instruction `print(recherche3(L, 30))` (recherche de la valeur 30 dans la liste `L`), induit les affichages suivants :

```
>>> print(recherche3(L, 30))
0 4 8
0 1 3
2 2 3
True
```

On a donc 3 itérations de la boucle `while`, au cours desquelles les valeurs affichées des variables `bas`, `milieu` et `haut` sont les suivantes :

	`bas`	`milieu`	`haut`
Itération 1	0	4	8
Itération 2	0	1	3
Itération 3	2	2	3

À l'issue de la 3ème itération, la valeur 30 est trouvée, donc la fonction renvoie `True`.

5. À l'exemple de ce qui précède, déterminer quels sont les affichages produits et la valeur renvoyée lors de l'exécution de l'appel `recherche3(L, 20)`.

6. Faire de même pour l'appel `recherche3(L, 0)`.

On peut s'assurer de la correction d'un algorithme utilisant une boucle, en produisant pour cette boucle un invariant de boucle.

7. Rappeler ce qu'est un invariant de boucle et en donner un pour la boucle de la fonction `recherche3`.

La fonction `recherche3` proposé comportant une boucle `while`, il est primordial de s'assurer de la fin de celle-ci, pour démontrer que l'algorithme termine.

8. Considérant l'implémentation de la fonction `recherche3`, indiquer à quelle(s) condition(s) une exécution de la fonction termine.

9. En considérant les valeurs initiales, b_0 et h_0, des variables `bas` et `haut`, et en notant respectivement, b_i et h_i, leurs valeurs, à la fin de la $i^{\text{ème}}$ itération de la boucle `while`, quelle propriété de la suite $(h_i - b_i)_{i\geq 0}$ assure que la boucle termine ? Pourquoi ? (il n'est pas demandé de démontrer cette propriété).

10. Déduire de la question **8.** un variant de boucle pour l'algorithme et lui donner un sens concret.

Complexité de la recherche dichotomique dans une liste triée.

11. Pour une liste de 1000 éléments, peut-on estimer le nombre d'itérations nécessaires pour que la boucle `while` termine ? Donner et justifier par des calculs votre estimation.

12. Pour une liste L de longueur n, donner, sans la démontrer, une majoration du nombre d'itérations nécessaires à la fonction `recherche3` pour renvoyer le résultat.

13. Donner alors la complexité d'exécution de cette fonction `recherche3`.

14. En comparant la complexité de la fonction `recherche3` à celle de la fonction `recherche1` définie à la question 1, expliquer en une ou deux phrases, l'intérêt de la recherche dichotomique.
On pourra en particulier comparer la façon dont on peut s'attendre à ce que varie le temps d'exécution lorsque la longueur de la liste en entrée double en considérant le nombre d'itérations nécessaires dans le pire cas.

Exercice 4. Algorithme de tri sélection

PARTIE A

On donne ici une implémentation du tri sélection légèrement différente de celle du cours

```
1.  def tri_selection1(L):
2.      n = len(L)
3.      for i in range(n - 1):
4.          p = 0
5.          for j in range(1, n - i):
6.              if L[j] > L[p]:
7.                  p = j
8.          L[n - 1 - i], L[p] = L[p], L[n - 1 - i]
9.          #print(L)
10.      return L
```

1. On décommente la ligne 9 et on exécute l'appel `tri_selection1([3, 5, 2, 8, 3, 4, 6])`.
   Donner tous affichages produits par l'instruction en ligne 9.
2. Quel est le rôle des instructions lignes 4 à 7 ?
3. Donner un invariant de boucle pour la boucle d'indice i.
4. Expliquer en une ou deux phrases pour quelle raison, si l'on se limite à appeler la fonction `tri_selection1` sur des listes auxquelles on a préalablement attaché un nom de variable, l'instruction `return L` en ligne 10 peut être omise.

PARTIE B

On indique que, lorsque la position k existe dans une liste L, l'instruction `del L[k]` supprime la valeur en position k dans la liste L.

On rappelle que, pour les listes Python, la méthode `.pop()`, sans paramètre, supprime et renvoie la dernière valeur dans une liste non vide, et on indique qu'utilisée avec un paramètre k, `L.pop(k)`, a le même effet que l'instruction `del L[k]`, mais en renvoyant la valeur supprimée.

1. Définir une fonction `supprime`, prenant en argument une liste L, et un indice de position k, vérifiant à l'aide d'une assertion que la valeur de k est correspond à l'indice d'une position existante dans L, et décalant d'un rang vers la gauche toutes les valeurs de L situées à la position k ou au-delà, et supprimant, après que tous les décalages ont été effectués, la dernière valeur de la liste L.
   On ajoutera une brève chaîne de documentation dans la définition de la fonction supprime donnant la spécification de la fonction.
2. Donner la signature de la fonction supprime.
3. Étant rappelé que l'instruction `L.pop()` sur une liste de longueur $n \geq 1$ a une complexité en O(1) (indépendante de la longueur n de la liste), déterminer la complexité de la fonction `supprime` en fonction de n et k.

On considère une implémentation du tri sélection différente de celle du cours, donnée par la fonction suivante :

```
1.  def tri_selection2(L):
2.      n = len(L)
3.      L1 = []
4.      for i in range(n):
5.          v, k = L[0], 0
6.          for j in range(len(L)):
7.              if L[j] < v:
8.                  v = L[j]
9.                  k = j
10.         L1.append(v)
11.         L.pop(k)
12.     return L1
```

4. Expliquer en une phrase le principe de fonctionnement de cet algorithme. Rappeler pourquoi il termine.
5. Évaluer sa complexité, sachant que la complexité de `L.pop(k)` est celle de `supprime(L, k)`.
6. Donner un invariant de boucle vérifié par le couple de listes (L, L_1) au fil des itérations de la boucle d'indice i.
7. Expliquer ce qui différencie les algorithmes implémentés par les fonctions `tri_selection1` et `tri_selection2`.
   Donner deux différences, l'une d'elle tenant au fait que l'un des deux tris est un tri en place et l'autre non.

8. Redonner le sens de l'expression « tri en place » et quel est l'avantage d'un tri en place.

Exercice 5. Algorithme de tri insertion

On donne la fonction suivante qui est version altérée de l'implémentation du tri insertion donnée en classe :

```
1.  def tri_insertionF(L):
2.      n = len(L)
3.      for i in range(1, n):
4.          tmp = L[i]
5.          j = i
6.          while L[j - 1] > L[i]:
7.              L[j] = L[j - 1]
8.              j = j - 1
9.          L[j] = tmp
```

On indique qu'en Python, une position négative, j, dans une liste L, et comprise entre $-\text{len}(L)$ et -1 désigne la position $i = \text{len}(L) - 1 - j$. Ainsi, par exemple, si $L = [5,\ 2,\ 3]$, $L[-1]$ vaudra 3 et $L[-3]$ vaudra 5.

1. Donner l'état de la liste L à la fin de chaque itération d'indice i lorsque l'on appelle la fonction `tri_insertionF` sur la liste $L = [1,\ 4,\ 3,\ 2,\ 1]$.

2. Donner l'état de la liste L à la fin de chaque itération d'indice i lorsque l'on appelle la fonction `tri_insertionF` sur la liste $L = [4,\ 1,\ 5]$.

3. Apporter des modifications entre les lignes 4 à 8 de la fonction `tri_insertionF` pour obtenir une fonction `tri_insertionT` dans laquelle l'invariant de boucle du tri insertion, à savoir le fait qu'avant toute itération et à la fin de chaque itération d'indice i, la portion de liste $L[:i+1]$ est triée.

On propose l'implémentation suivante du tri insertion

```
1.  def tri_insertion2(L):
2.      n = len(L)
3.      for i in range(n - 2, -1, -1):
4.          j = i
5.          while j < n - 1 and L[j] > L[j + 1]:
6.              L[j + 1], L[j] = L[j], L[j + 1]
7.              j = j + 1
```

On indique que pour a entier positif, `range(a, -1, -1)` génère la liste de valeurs $[a, a-1, \ldots, 1, 0]$.

4. Donner l'état de la liste L à la fin de chaque itération d'indice i lorsque l'on appelle la fonction `tri_insertion2` sur la liste $L = [7,\ 4,\ 1,\ 6,\ 5]$.

5. Donner l'invariant de boucle vérifié par la liste L au fil des itérations de la boucle `for` d'indice i.

6. Quel est le rôle des lignes 4 à 7 ?

Exercice 6. Diviser pour régner

Tri fusion

On considère la fonction `g` suivante :

```
def g(L1, L2):
    n1, n2 = len(L1), len(L2)
    i1, i2 = 0, 0
    L1 = L1 + [float('inf')]
    L2 = L2 + [float('inf')]
    L = []
    while i1 + i2 <= n1 + n2:
        if L1[i1] < L2[i2]:
            L = L + [L1[i1]]
            i1 = i1 + 1
        else:
            L = L + [L2[i2]]
            i2 = i2 + 1
    L.pop()
```

```
    return L
```

On indique que la valeur `float('inf')` est une valeur flottante spéciale ayant la propriété d'être supérieure ou égale à toute autre valeur flottante ou entière.

Remarque : cette valeur est affichée sous la forme de la chaîne de caractère `inf`

```
>>> float('inf')
inf
```

On rappelle que l'instruction `L.pop()` supprime et renvoie la valeur en dernière position dans une liste L non vide.

1. Donner l'état de la liste L au fil des itérations lors d'un appel à la fonction g avec pour argument les listes $L_1 = [4, 1, 2]$ et $L_2 = [3, 4, 5]$.

2. Reprendre la question **1.**, lorsque $L_1 = [1, 2, 4]$ et $L_2 = [3, 4, 5]$.

3. Quelles que soient les listes L_1 et L_2 en argument de la fonction, quel est, en fonction des longueurs respectives n_1 et n_2 de ces deux listes, le nombre d'itérations effectuées par la boucle `while`. Justifier.

4. Quelles que soient les listes L_1 et L_2 en argument de la fonction, quelle sera le nombre d'éléments de la liste L renvoyée par la fonction `g` ? Justifier.

5. Si les listes L_1 et L_2 sont triées, décrire, en fonction des valeurs i_1 et i_2 des variables `i1` et `i2`, décrire le contenu de la liste L à la fin de chaque itération.
En déduire l'état de la liste L lorsque la fonction termine.

On souhaite utiliser la fonction `g` précédente pour donner une implémentation récursive sous la forme d'une fonction `tri_fusion` du tri par fusion, qui, on le rappelle, consiste à appeler récursivement la fonction sur les moitiés gauche et droite d'une liste non triée, et à construire une version triée de la liste en entrée à partir des listes triées renvoyées par ces deux appels récursifs.

6. Compléter sur votre copie le squelette de fonction suivant, de sorte à obtenir une telle implémentation du tri par fusion

```
1.  def tri_fusion(L):
2.      n = len(L)
3.      ## traitement des cas de base
4.      ...
5.
6.      ## traitement du cas général par deux appels récursifs
7.      Lgauche = ... # Lgauche est une copie de la partie gauche de la liste,
8.                    # obtenue par slicing
9.      Ldroite = ... # Lgauche est une copie de la partie droite de la liste
10.                    # obtenue par slicing
11.     ...
12.
13.     return ...
```

7. Quelle est la complexité, en fonction de la longueur n de la liste L des opérations en lignes 7 et 9 ?

8. Rappeler sans justification la complexité du tri fusion et la comparer à celle du tri insertion.

L'algorithme de tri par fusion utilise le principe « diviser pour régner ».

9. Rappeler ce que recouvre ce principe.

10. Proposer une fonction `maximum`, dont le squelette est le même que celui proposé en question **6.** et implémentant récursivement la recherche du maximum d'une liste L en appliquant le principe « diviser pour régner ».

11. Que pensez-vous de l'intérêt de la fonction `maximum` proposée en termes de complexité ?