Diviser pour régner - EXERCICES

Exercice 1 : Palindromes

Une chaîne est un *palindrome* si elle peut se lire de la même manière dans les deux sens, de gauche à droite et de droite à gauche. Par exemple :

- `"ressasser"`, `"radar"`, `"12321"` sont des palindromes ;
- `"nsi"`, `"toto"` n'en sont pas.

On se propose d'écrire une fonction récursive `est_palindrome(ch)` qui renvoie Vrai si la chaine `ch` passée en argument est un palindrome et Faux sinon.

Une première version utilisant le *slicing*

On considère la chaîne suivante.

In […
```
ch = "ressasser"
```

Question 1 : Quels *slices* permettent d'accéder :

- au premier caractère de `ch` ?
- au dernier caractère de `ch` ?
- à une chaîne privée du premier et du dernier caractères de `ch`.

In […
```
# à compléter
```

Question 2 : En utilisant les *slices*, complétez la fonction suivante.

In […
```
def est_palindrome(ch):
    if len(ch) <= 1:
        return ...
    else:
        return ...
```

Question 3 : Au moyen du mot clé `assert`, écrivez un bon jeu de tests pour cette fonction. Vérifiez que les tests passent avec succès.

In […
```

```

Question 4 : Expliquez pourquoi cet algorithme est un exemple de la méthode *diviser pour régner*.

Question 5 : Rendez-vous sur Python tutor (http://www.pythontutor.com/visualize.html#code=def%20est_palindrome%28ch%29%3A%0A%20%20%20%20if%20len%28ch%
1%5D%20and%20est_palindrome%28ch%5B1%3A-1%5D%29%0A%0Aest_palindrome%28%22ressasser%22%29&cumulative=false&curInstr=0&heapP
frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) et exécutez en mode pas à

pas le déroulement des appels récursifs de cette fonction pour la chaîne `"ressasser"`.

1. Expliquez pourquoi cet exemple constitue un *pire cas* dans l'étude du coût de l'algorithme ?
2. Combien de comparaisons on été effectuées ?
3. Si la chaîne de départ est un palindrome de longueur `n`, combien de comparaisons sont effectuées ?
4. Quel est le coût en temps de cet algorithme ?

Question 6 : Le *slicing* a un coût en mémoire caché. En effet, chaque appel `ch[1:-1]` induit la création d'une nouvelle chaîne qu'il faut stocker en mémoire. On peut le voir facilement avec Python tutor. Dans le pire cas, pour une chaîne de longueur `n` au départ, combien de chaînes doivent être créées pour répondre au problème ?

Une deuxième version améliorée

En réalité, on peut se passer de la création de toutes ces chaînes intermédiaires en utilisant et en faisant varier (comme pour la recherche dichotomique) les indices `g` (pour gauche) et `d` (pour droite) des caractères restants.

Question 7 : Ecrivez la fonction `palindrome(ch, g, d)` qui renvoie `True` si la chaine `ch[g..d]` (c'est-à-dire la chaîne `ch` limitée à ses caractères entre les positions `g` et `d`) est un palindrome et `False` sinon. Il suffit d'adapter la fonction de la première version.

In […
```
# à compléter
```

Question 8 : Quel appel à cette fonction faut-il faire pour tester si `mot` est un palindrome ?

In […
```
mot = "ressasser"
# à compléter par le bon appel :
```

Question 9 : Pour éviter cette écriture un peu lourde, il suffit de créer une autre fonction `est_palindrome2(ch)` qui est chargée de lancer le premier appel à la fonction récursive `palindrome`. Complétez le code de la fonction `est_palindrome2(ch)`.

In […
```
def est_palindrome2(ch):
    # à compléter
    pass
```

> La fonction `est_palindrome2` est appelée une *fonction d'interface* qui permet d'ajouter des arguments à une fonction sans que l'utilisateur ait à s'en préoccuper.

Question 10 : Vérifiez avec Python tutor (http://www.pythontutor.com/visualize.html#code=def%20palindrome%28ch,%20g,%20d%29%3A%0A%20%20%20%20if%20d%20-%20g%20%3C%201%3A%0A%20%20%20%20%20%20%20%20return%20True%0A%20%20%20%20return%20palindrome%28ch,%20g%2B1,%20d-1%29%0A%0Adef%20est_palindrome2%28ch%29%3A%0A%20%20%20%20%23%20%C3%A0%20%200,%20len%28ch%29-1%29%0A%0Aprint%28est_palindrome2%28%22ressasser%22%29%29&cumulative=false&curInstr=frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) qu'avec cette version, il n'y a

qu'une chaîne `ch` à mémoriser (celle de départ). On obtient un algorithme avec un coût mémoire inférieur à la première version.

Exercice 2 : Calcul du minimum d'une liste

On peut appliquer la méthode *diviser pour régner* afin de déterminer la valeur minimale d'une liste (ou tableau) non vide d'entiers. Voici les trois étapes de la résolution du problème :

1. DIVISER : on "coupe" la liste en deux sous-listes de même taille (à un élément prêt) ;
2. REGNER : on cherche récursivement le minimum de chacune des deux sous-listes jusqu'à obtenir une liste d'un élément qui a pour minimum cet élément ;
3. COMBINER : on peut calculer la réponse au problème en utilisant le fait que le minimum d'une liste est le minimum des deux minimums de ses deux sous-listes.

> On peut écrire un algorithme utilisant le *slicing* mais, comme pour l'exercice précédent, cela induira un coût en mémoire supplémentaire (avec la création des toutes les listes intermédiaires). Pour éviter cela, on écrira uniquement une version "améliorée" qui utilise les indices `g` et `d` du premier et du dernier élément de chaque liste.

Question 1 : On note `m = (g + d) // 2` la position de l'élément central de la liste `L[g..d]` . On veut écrire une fonction récursive `mini(L, g, d)` qui renvoie la valeur minimale de `L[g..d]` .

Complétez la définition de cette fonction. On pourra utiliser la fonction `min(a, b)` qui renvoie la valeur minimale de deux nombres `a` et `b` .

$$\text{mini}(\text{L}, \text{g}, \text{d}) = \begin{cases} \text{L}[\ldots] & \text{si g} = \text{d} \\ \ldots & \text{sinon} \end{cases}$$

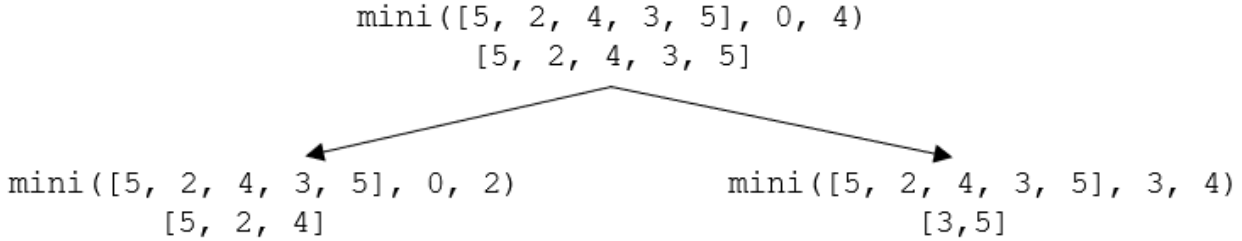
Question 2: Ecrivez la fonction Python correspondante.

In […
```
# à compléter
```

Question 3 : Ecrivez la fonction d'interface `minimum(L)` qui est chargée d'effectuer le premier appel à la fonction `mini` , et qui renvoie la réponse au problème, c'est-à-dire le minimum d'une liste `L` non vide).

In […
```
# à compléter
```

Question 4 : Recopiez et complétez l'arbre des appels récursifs à la fonction `mini` si on effectue l'appel `minimum([5, 2, 4, 3, 5])` . Vous indiquerez par des numéros l'ordre des différents appels (1 pour le premier, 2 pour le deuxième,...). On pourra uniquement écrire la "portion de liste" sur laquelle porte chaque appel pour gagner du temps.



> On peut visualiser les appels successifs avec Python tutor (http://www.pythontutor.com/visualize.html#code=def%20mini%28L,%20g,%20d%29%3A%0A%20%20%20%20if%20d%20%3D%3D%20g%3A%0A%20%20%20%20%20%20%20%20return%20L%5Bg%5D%0A%20%20%20%20%20%20%20%20minG%20%3D%20mini%28L,%20g,%20m%29%0A%20%20%20%20%20%20%20%20minD%20%3D%20mini%28L,%20m%2B1,%20d%29%0A%20%20%20%20%20%20%20%20return%20min%28minG,%20minD%29%0A%0Adef%20minimum%28L%29%3A%0A%20%20%20%20return%20mini%28L,%200,%20len%28L%29-1%29%0A%0Aprint%28minimum%28%5B5,%202,%204,%203,%205%5D%29%29&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false). Dans le code présenté, on a pris soin de stocker les minimums des sous-listes gauche et droite dans deux variables pour plus de lisibilité et pour mieux suivre. Cela vous permettra de vérifier votre réponse à la question précédente.

Exercice 3 : Le tri fusion

> En anglais, *merge sort*.

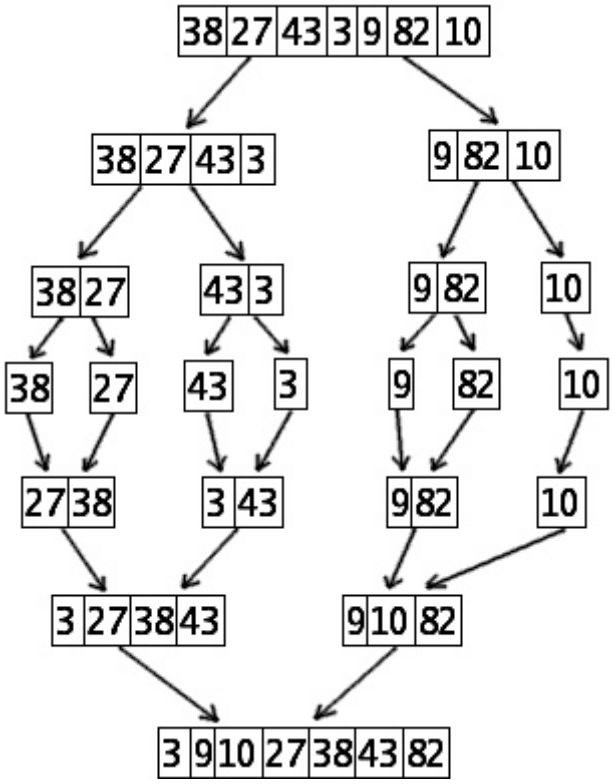
Vous avez découvert et implémenté en classe de Première deux algorithmes de tri : le tri par sélection et le tri par insertion. Nous avions vu que ces deux algorithmes ont un coût quadratique (de l'ordre de n^2, où n est la taille du tableau/liste à trier).

Il existe des algorithmes plus efficaces (de moindre coût) pour trier une liste. Le *tri fusion* en fait partie et c'est l'objet de cet exercice.

Principe du *tri fusion*

L'idée du *tri fusion* est la suivante : on sépare les éléments de la liste en deux sous-listes de tailles égales à un élément prêt. Ensuite, on tri *récursivement* chacune des deux listes avec le tri fusion, jusqu'à obtenir des listes d'au plus un élément qui sont déjà triées. Ensuite, on fusionne les deux listes triées.

Exemple



Question 1 : Justifier pourquoi cet algorithme repose sur principe *diviser pour régner*. Vous identifierez clairement les trois étapes de ce principe algorithmique.

Question 2 : Dans l'exemple précédent, identifiez la phase d'appels récursifs et la phase de fusions des listes. A quelle(s) étape(s) du principe *diviser pour régner* correspondent-elles ?

Ecriture de la fonction récursive `tri_fusion`

Question 3 : Si on appelle `fusion` la fonction qui fusionne deux listes, complétez la fonction récursive `tri_fusion(lst)` qui trie une liste `lst` d'entiers selon la méthode du tri fusion.

In […

```
def tri_fusion(lst):
    """Renvoie une nouvelle liste avec les éléments triés (dans l'ordre croissant)
    # à compléter
    if ... :
        return lst
    else:
        m =                  # m est la position médiane
        l1 =                 # l1 est la sous-liste de gauche (lst[0...m-1]) trié
        l2 =                 # l2 est la sous-liste de droite (lst[m..]) triée pa
        return fusion(l1, l2)
```

Question 4 : Sur le schéma de l'exemple précédent (tri de la liste `[38, 27, 43, 3, 9, 82, 10]`), indiquez par des numéros l'ordre des différents appels à la fonction `tri_fusion` ainsi qu'à la fonction `fusion` (1 pour le premier appel, 2 pour le second, ...).

Ecriture de la fonction `fusion`

Il reste à écrire la fonction `fusion(l1, l2)` qui doit créer une nouvelle liste triée en fusionnant les listes triées `l1` et `l2` .

Question 5 : Observez l'animation suivante pour en déduire l'algorithme de la fonction `fusion(l1, l2)` qui suit.

6 5 3 1 8 7 2 4

Source : Wikipedia (https://fr.wikipedia.org/wiki/Tri_fusion#/media/Fichier:Merge-sort-example-300px.gif), Swfung8, CC-BY-SA 3.0.

In […

```
def fusion(l1, l2):
    """Renvoie une liste triée résultant de la fusion des listes triées l1 et l2."
    L = []  # création de la liste à renvoyer
    n1, n2 = len(l1), len(l2)  # tailles des deux listes
    i1, i2 = 0, 0 # positions de départ dans les deux listes
    # à compléter
    while ... :
        pass
```

Aide n°1 : Il faut comparer les plus petits éléments respectifs des deux listes et ajouter le plus petit d'entre eux à la liste `L`, jusqu'à épuisement de l'une des deux listes. Il ne reste plus qu'à complèter la liste `L` avec tous les éléments restants. Si besoin, regardez l'aide n°2.

Aide n°2 :

```
fonction fusion(l1, l2)
    L ← liste vide
    n1 ← taille(l1)
    n2 ← taille(l2)
    i ← 0
    j ← 0
    Tant que i1 < n1 et i2 < n2 faire
        Si ... alors
            Ajouter ... à la fin de L
            i1 ← ...
        sinon
            Ajouter ... à la fin de L
            i2 ← ...
    Pour i allant de i1 à n1-1 faire  # s'il reste des éléments da
ns l1
        Ajouter ... à la fin de L
    Pour j allant de i2 à n2-1 faire  # s'il reste des éléments da
ns l2
        Ajouter ... à la fin de L
    Renvoyer ...
```

Exercice 4 : Rotation d'un quart de tour d'une image

Cette exercice est tiré du livre Prépabac NSI, Terminale, activité 14 page 52 (60 minutes). L'objectif de l'activité est d'implémenter un algorithme simple de rotation d'un quart de tour puis

d'implémenter un algorithme basé sur la méthode *diviser pour régner* qui permet d'effectuer la rotation avec un coût en mémoire constant.

Image à télécharger : elle s'intitule `joconde.jpg` et est située dans le dossier `data` situé dans le répertoire de ce notebook.

Petite précision : si vous utilisez un notebook pour coder vos réponses, il peut être intéressant de remplacer l'instruction

```
img.show()
```
par

```
display(img)
```
afin d'afficher directement l'image dans le notebook.