Exercices 3

Philipp Ahrendt

January 7, 2026

Contents

1 Exercice 1: Palindromes

Un palindrome est une chaîne de caractères qui reste la même lorsqu'on inverse l'ordre des caractères. Par exemple, *kayak* et *elle* sont des palindromes. On considère aussi que la chaîne vide `""` est un palindrome, et que

toute chaîne à un seul caractère, comme par exemple `"a"` , est un palindrome. Écrire une fonction `palindrome` qui teste si une chaîne de caractères est un palindrome ou non.

```
def palindrome(c):
    if len(c) <= 1:
        return True
    else:
        return (c[0] == c[-1]) and palindrome(c[1:len(c)-1])
```

2 Exercice 2: Tri par insertion récursif

Dans cet exercice, on va reformuler le tri par insertion comme un algorithme récursif. Dans un premier temps, on va considérer une version qui ne manipule pas la liste d'entrée `L`, mais qui rend une nouvelle liste `L_t` triée contenant les mêmes entrées que `L`.

2.1 Question a:

Écrire une fonction `inser(L,n)`, qui prend en entrée une liste `L`, que l'on suppose triée, et un nombre `n`, et renvoie la liste triée `L_1` obtenue en rajoutant `n` à `L`.

Réponse:

```
def inser(L,n):
    L_1 = []
    i = 0
    while i < len(L) and L[i] < n:
        L_1.append(L[i])
        i += 1
    L_1.append(n)
    for i in range(i,len(L)):
        L_1.append(L[i])
    return L_1

print(inser([1,5,8,9],7))
```

2.2 Question b:

Utiliser la fonction `inser` pour écrire une fonction récursive `tri_insertion_rec` qui implémente le tri par insertion.

Réponse:

```
def tri_insertion_rec(L):
    if len(L) == 0:
        return L
    else:
        return inser(tri_insertion_rec(L[0:len(L)-1]),L[len(L)-1])
```

2.3 Question c:

Écrire une fonction récursive `tri_insertion_rec_direct(L)` qui trie directement la liste d'entrée `L`.

Réponse:

```
def inser(L,k):
    tmp = L[k]
    a = k-1
    while a > 0 and L[a] > tmp:
        L[a+1] = L[a]
        L[a] = tmp
        a = a - 1

def tri_insertion_rec_direct(L,k):
    if k > 0:
        tri_insertion_rec_direct(L,k-1)
        inser(L,k)

def insertion(L):
    tri_insertion_rec_direct(L,len(L)-1)
```

3 Exercice 3: Tours de Hanoï

Dans cet exercice, on va implémenter la solution récursive du jeu des *tours de Hanoï* vue en cours.

La représentation du jeu est la suivante:

- Les trois piliers sont représentés par les caractères `'A'` , `'B'` et `'C'`.

- Un nombre `n` représente le nombre de disques dans le jeu.

Écrire une fonction `hanoi(n,P1,P2)`, qui prend en entrée le nombre `n` de piliers, le pilier de départ `P1` et d'arrivée `P2`, et qui affiche une par une les instructions pour déplacer `n` piliers de `P1` vers `P2`.

Par exemple, `hanoi(3,'A','C')` doit afficher:

```
deplacer disque 1 de A vers C
deplacer disque 2 de A vers B
deplacer disque 1 de C vers B
deplacer disque 3 de A vers C
deplacer disque 1 de B vers A
deplacer disque 2 de B vers C
deplacer disque 1 de A vers C
```

Réponse:

```
def hanoi(n,P1,P2):
    if n > 0:

        if P1 == 'A':
            if P2 == 'B':
                hanoi(n-1,'A','C')
                print(f'deplacer disque {n} de A vers B')
                hanoi(n-1,'C','B')
        if P1 == 'A':
            if P2 == 'C':
                hanoi(n-1,'A','B')
                print(f'deplacer disque {n} de A vers C')
                hanoi(n-1,'B','C')
        if P1 == 'B':
            if P2 == 'C':
                hanoi(n-1,'B','A')
                print(f'deplacer disque {n} de B vers C')
                hanoi(n-1,'A','C')
        if P1 == 'B':
            if P2 == 'A':
                hanoi(n-1,'B','C')
                print(f'deplacer disque {n} de B vers A')
                hanoi(n-1,'C','A')
        if P1 == 'C':
```

```
            if P2 == 'B':
                hanoi(n-1,'C','A')
                print(f'deplacer disque {n} de C vers B')
                hanoi(n-1,'A','B')
        if P1 == 'C':
            if P2 == 'A':
                hanoi(n-1,'C','B')
                print(f'deplacer disque {n} de C vers A')
                hanoi(n-1,'B','A')
```

4 Exercice 4: Tri fusion

Cet exercice a pour but d'implémenter un algorithme de tri de type *diviser pour régner*; le **tri fusion**. Pour trier une liste `L`, l'idée est la suivante:

- On découpe `L` en deux morceaux `L1` et `L2`, et on les trie séparément.
- On fusionne les deux listes triées, afin d'obtenir un tri de la liste entière `L`.

4.1 Question a:

Écrire une fonction `fusion`, qui prend en entrée deux listes triées `L1` et `L2`, et renvoie une liste triée `L_fusion` contenant les entrées de `L1` et `L2`.

Réponse:

```
def fusion(L1,L2):
    L = []
    i1 = 0
    i2 = 0
    while i1 < len(L1) and i2 < len(L2):
        if L1[i1] < L2[i2]:
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1

    if i1 == len(L1):
        L.extend(L2[i2:len(L2)])
    else:
```

```
        L.extend(L1[i1:len(L1)])

    return L
```

4.2 Question b:

Utiliser la fonction `fusion` pour écrire une fonction `tri_fusion` qui implémente le tri fusion.

Réponse:

```
def tri_fusion(L):
    L1 = L[0:len(L)//2]
    L2 = L[len(L)//2:len(L)]

    return fusion(tri_fusion(L1),tri_fusion(L2))
```

5 Exercice 5: Chemins dans un graphe

Implémenter l'algorithme pour trouver le chemin le plus court dans un graphe, vu en cours.

```
def chemins(n,E):

    # On commence par créer chemins_aux(n,E,0)
    M = []
    for i in range(n):
        L = []
        for j in range(n):
            if (i,j) in E:
                L.append(1)
            else:
                # On utilise une valeur "trop grande" pour représenter "infini"
                # Cela facilite l'écriture du min dans l'algorithme
                L.append(2*len(E)+1)
        M.append(L)

    # L'algorithme dynamique principal
    for k in range(n):
        for i in range(n):
            for j in range(n):
```

```
            M[i][j] = min(M[i][j], M[i][k] + M[k][j])

    # On finit par mettre des "infini" là où il faut
    for i in range(n):
        for j in range(n):
            if M[i][j] == 2*len(E)+1:
                M[i][j] = "infini"

    return M
```

6 Exercice 6: Sous-listes croissantes

Si `L` est une liste de nombres, une *sous-liste croissante* de `L` est donnée par une suite d'indices `i_1 < i_2 ... < i_k` tels que `L[i_1] <= L[i_2] ... <= L[i_k]`. Le but de cet exercice est de construire un algorithme `crois(L)` qui trouve, pour une liste `L`, la longueur de la sous-liste croissante la plus longue.

Exemple:

- `L = [1,8,6,4,5,7,2,5]`
- sous-liste la plus longue: `[1,4,5,7]`
- `crois(L) == 4`

6.1 Question a:

Pour un indice i de la liste, notons `crois(L,i)` la longueur de la sous-liste croissante la plus longue qui commence à l'indice i. Donner une formule pour `crois(L)` en termes des `crois(L,i)`.

Réponse: Si `L_0` est la sous-liste la plus longue de `L`, et i_0 est l'indice du premier élément de `L_0`, alors on a `crois(L) = len(L_0) = crois(L,i_0)`; `L_0` est en particulier la sous-liste la plus longue qui commence en i_0. De l'autre côté on a `crois(L) >= crois(L,i)` pour tout indice i. Cela montre que

`crois(L) = max(crois(L,i) pour 0 <= i < len(L))`

6.2 Question b:

Notons I l'ensemble des indices j > i tels que L[j] >= L[i]. Donner une formule pour `crois(L,i)` en termes des `crois(L,j)`, pour j ∈ I.

Réponse: Toute liste croissante qui commence en i a son prochain élément à une position j ∈ I. Pour trouver la sous-liste la plus longue qui commence en i, il suffit donc de trouver la plus longue parmi les listes qui commencent en un des j ∈ I, et d'y rajouter l'élément i. En utilisant le raisonnement de la question précédente, on trouve que

`crois(L,i) = 1 + max(crois(L,j) pour j dans I)`

6.3 Question c:

En déduire un algorithme récursif (diviser pour régner) pour calculer `crois(L)` et l'implémenter. Estimer sa complexité.

Réponse:

```
# On implémente d'abord la "vraie fonction" récursive crois(L,i)
def crois_aux(L,i):
    sous_longueurs = []

    for j in range(i+1,len(L)):
        if L[j] >= L[i]:
            sous_longueurs.append(crois_aux(L,j))

    if len(sous_longueurs) > 0:
        return 1 + max(sous_longueurs)
    else:
        return 1

def crois(L):
   return max([crois_aux(L,i) for i in range(len(L))])
```

Au pire des cas, pour une liste de longueur n, cet algorithme fait un appel récursif à tous les cas 0 <= i <= n (avec un nombre constant d'opérations pour diviser et combiner). La formule pour la complexité C(n) est donc

$C(n) = \sum_{0 \leq i \leq n} C(i) + K$

Pas besoin de calculer la complexité exacte (pour info, c'est de l'ordre 2^n), il suffit de voir que c'est pire que l'algorithme de fibonacci naïf, dont on sait que la complexité est exponentielle, et qu'on a donc une complexité au moins exponentielle, ou encore pire, pour cet algorithme.

6.4 Question d:

Implémenter un algorithme suivant une approche de programmation dynamique. Estimer sa complexité

```
def crois(L):
    sous_longueurs = [1 for e in L]

    for k in range(len(L)):
        i = len(L) - k - 2
        m = 0
        for j in range(i+1,len(L)):
            if L[j] >= L[i] and sous_longueurs[j] > m:
                m = sous_longueurs[j]
        sous_longueurs[i] = m+1

    return max(sous_longueurs)
```

On peut remarquer que les 2 boucles imbriquées de cet algorithme sont très similaires à ceux du tri par insertion, ce qui nous donne une complexité de n^2, bien meilleure que l'approche naïve de diviser pour régner.