Python – Révisions

Parcours de listes et de textes

Pour rappel, voici les 3 façons de parcourir une liste (ou un tableau). Il est possible de faire la même chose avec des textes.

```
def parcours1(liste):
    for val in liste:
        print(val)
```

```
>>> parcours1([3, -2, 4])
3
-2
4
```

```
def parcours2(liste):
    for i in range(len(liste)):
        print(i, liste[i])
```

```
>>> parcours2([3, -2, 4])
0 3
1 -2
2 4
```

```
def parcours3(liste): # parcours à rebours
    for i in range(len(liste)-1, -1, -1): # on part du dernier indice
        print(i, liste[i])
```

```
>>> parcours3([3, -2, 4])
2 4
1 -2
0 3
```

Vous pouvez soit écrire chacune des fonctions en intégralité, soit utiliser le fichier `revisions-eleve.py` qui contient des versions à trous.
Vous pouvez, et devriez, rajouter des **`assert`** après chacunes de vos fonctions pour les tester automatiquement.

Exercice 1 : Écrire une fonction `compter(symbole, texte)` qui renvoie le nombre de fois où `symbole` apparaît dans `texte`.

```
>>> compter('b', 'bulle')
1
>>> compter('l', 'bulle')
2
>>> compter('v', 'bulle')
0
```

Exercice 2 : Écrire une fonction `position(val, liste)` qui renvoie la position de la première occurrence de `val` dans `liste`. La fonction renvoie `-1` si `val` n'est pas dans `liste`.

```
>>> position(7, [5, -1, 7, 4, 6, 4, 2])
2
>>> position(4, [5, -1, 7, 4, 6, 4, 2])
3
>>> position(0, [5, -1, 7, 4, 6, 4, 2])
-1
```

Exercice 3 : Écrire une fonction `derniere_position(val, liste)` qui renvoie la position de la dernière occurrence de `val` dans `liste`. La fonction renvoie `-1` si `val` n'est pas dans `liste`.

```
>>> derniere_position(7, [5, -1, 7, 4, 6, 4, 2])
2
>>> derniere_position(4, [5, -1, 7, 4, 6, 4, 2])
5
>>> derniere_position(0, [5, -1, 7, 4, 6, 4, 2])
-1
```

Exercice 4 : Écrire une fonction `indice_egal_valeur(liste)` qui renvoie **`True`** s'il existe un indice `i` tel que `liste[i] = i`. Sinon elle renvoie **`False`**.

```
>>> indice_egal_valeur([7, 1, 8])   # liste[1] = 1
True
>>> indice_egal_valeur([9, -7, 2, 9, 6])    # liste[2] = 2
True
>>> indice_egal_valeur([1, 2, 3, 4])
False
```

Exercice 5 : Écrire une fonction `moyenne(liste)` qui renvoie la moyenne des valeurs de `liste`, qui ne doit pas être vide.

```
>>> moyenne([5])
5.0
>>> moyenne([5, 15, 8])
9.333333333333334
>>> moyenne([5, 15, 10])
10.0
```

Exercice 6 : Écrire une fonction `moyenne_ponderee(liste)` qui renvoie la moyenne pondérée de la liste de couples `liste`. Chaque élément de `liste` est un couple `(note, coeff)`. Pour rappel, pour calculer une moyenne pondérée, il faut faire la somme des produits des notes avec leur coefficient et diviser le tout par la somme des coefficients. On suppose que toutes les notes sont sur 20 et qu'il y a au moins une note avec un coefficient non nul.

```
>>> moyenne_ponderee([(5, 1), (15, 1)])
10.0
>>> moyenne_ponderee([(5, 1), (15, 2)]) # 5*1 + 15*2 = 35 -> 35/3
11.666666666666666
>>> moyenne_ponderee([(5, 1), (15, 3)]) # 5*1 + 15*3 = 50 -> 50/4
12.5
>>> moyenne_ponderee([(5, 1), (15, 3), (20, 0)]) # le coeff 0 ne compte pas
12.5
```

Exercice 7 : Écrire une fonction `effectifs(liste)` qui renvoie un dictionnaire associant à chaque élément apparaissant dans `liste` son effectif.

```
>>> effectifs([4, 1, 2, 4, 2, 2, 6])
{4: 2, 1: 1, 2: 3, 6: 1}
```

Maximum et minimum

Exercice 8 : Écrire une fonction `maximum(liste)` qui renvoie le maximum de `liste`. Si la liste est vide, il faut renvoyer **`None`**.

```
>>> maximum([4, 7, 1])
7
>>> maximum([3])
3
>>> maximum([-865, -6464, -4545, -121])
-121
>>> maximum([])   # None ne s'affiche pas
>>>
```

Exercice 9 : Écrire une fonction `ecart_max_min(liste)` qui renvoie la différence entre le maximum et le minimum de `liste`. On suppose que la liste n'est pas vide.
Vous ne pouvez pas utiliser les fonctions Python **`min`** et **`max`**.
Il est possible de faire cette fonction avec un seul parcours de la liste.

```
>>> ecart_max_min([1, 7])  # mini = 1, maxi = 7 -> 7-1 = 6
6
>>> ecart_max_min([3, 1, 9, 2])  # mini = 1, maxi = 9 -> 9-1 = 8
8
>>> ecart_max_min([15, 1, 9, 2])  # mini = 1, maxi = 15 -> 15-1 = 14
14
```

Exercice 10 : Écrire une fonction `max_avant_min(liste)` qui renvoie **`True`** si le maximum se trouve strictement avant le minimum de `liste`. Sinon, elle renvoie **`False`**. On suppose que la liste n'est pas vide.

```
>>> max_avant_min([4, 7, -2, 19, 5])   # 19 apres -2
False
>>> max_avant_min([-54, -85, -7, -45, -123, -26])   # -7 avant -123
True
>>> max_avant_min([7])   # 7 est le min et le max
False
```

Exercice 11 : Écrire une fonction `compter_maximum(liste)` qui prend une liste et renvoie le nombre d'occurrences du maximum de `liste`. Si la liste est vide, il faut renvoyer `0`. Vous ne pouvez ni utiliser les fonctions prédéfinies **`max`** et `count`. Il est possible de trouver le résultat en ne faisant qu'un seul parcours de la liste. Il faut bien réfléchir à ce qu'il faut faire lorsqu'on trouve un nouveau maximum.

```
>>> compter_maximum([3, 6, 1])
1
>>> compter_maximum([3, 6, 1, 6, 6, 3])
3
>>> compter_maximum([0, 0, 0, 0])
4
>>> compter_maximum([0])
1
```

Exercice 12 : Écrire une fonction `liste_positions_maximum(liste)` qui renvoie la liste des indices des occurrences du maximum dans `liste`. Là aussi, il est possible de le faire en un seul parcours.

```
>>> liste_positions_maximum([3, 6, 1])
[1]
>>> liste_positions_maximum([3, 6, 1, 6, 6, 3])
[1, 3, 4]
>>> liste_positions_maximum([0, 0, 0, 0])
[0, 1, 2, 3]
```

Liste en compréhension et boucles

Les deux fonctions suivantes renvoient le même résultat à chaque fois. On dit qu'elles sont équivalentes.

```
def affine1(liste):
    nouvelle_liste = []
    for v in liste:
        nouvelle_liste.append(3*v - 5)
    return nouvelle_liste

def affine2(liste):
    return [3*v - 5 for v in liste]      # liste en compréhension
```

```
>>> affine1([5, 2, 8, 10])
[10, 1, 19, 25]
>>> affine2([5, 2, 8, 10])
[10, 1, 19, 25]
```

Exercice 13 : Écrire une fonction `fonction2(liste)` équivalente à `fonction1(liste)`, mais en utilisant une liste en compréhension.

```
def fonction1(liste):
    nouvelle_liste = []
    for v in liste:
        nouvelle_liste.append(v*v + 3*v)
    return nouvelle_liste
```

Il est également possible de filtrer les éléments pour n'en garder que certains pour la nouvelle liste :

```
def filtrer1(a, liste):
    nouvelle_liste = []
    for v in liste:
        if v > a:
            nouvelle_liste.append(v)
    return nouvelle_liste

def filtrer2(a, liste):
    return [v for v in liste if v > a]
```

Exercice 14 : Écrire une fonction `triple_dans_intervalle1(a, b, liste)` qui est équivalente à la fonction suivante, mais en utilisant une boucle.

```
def triple_dans_intervalle2(a, b, liste):
    return [3*v for v in liste if a <= v <= b]
```

Génération d'une sous-liste

Exercice 15 : Écrire une fonction `que_les_negatifs(liste)` qui prend une liste `liste` et renvoie une nouvelle liste ne contenant que les éléments strictement negatifs de `liste`.

```
>>> que_les_negatifs([-3, 5, -6, 9])
[-3, -6]
>>> que_les_negatifs([2])
[]
>>> que_les_negatifs([-1, -7, -5, -3])
[-1, -7, -5, -3]
```

Exercice 16 : Écrire une fonction `que_croissant(liste)` qui prend une liste de nombres et renvoie une nouvelle liste commençant par le premier élément de `liste` et en ne rajoute un élément que s'il est strictement supérieur au dernier ajouté.

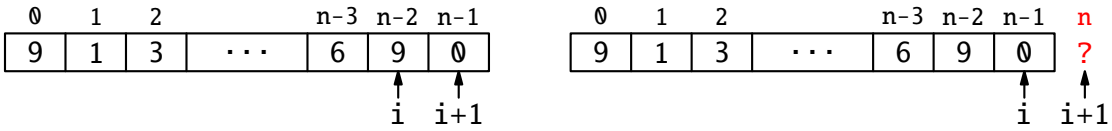
```
>>> que_croissant([1, 2, 5, 3, 10, -1]) # 3 et -1 inférieur au précédent
[1, 2, 5, 10]
>>> que_croissant([11, 2, 5, 3, 10, -1]) # aucun n'est supérieur à 11
[11]
>>> que_croissant([11, 2, 125, 3, 15, 120])
[11, 125]
```

Exercice 17 : Des escrocs ont réussi à voler un numéro de carte bancaire. Ne sachant pas quelle somme est disponible sur le compte, ils passent un maximum de commandes sur des sites Internet sans même attendre la confirmation de la Banque. Cette dernière traite les transactions les unes après les autres et bloque toutes les suivantes dès que la limite disponible est dépassée. La transaction faisant dépasser la limite n'est pas validée.
Écrire une fonction `validees(commandes, limite)` qui renvoie la liste des commandes validées jusqu'à dépasser la limite. Si la limite n'est pas atteinte, toutes les commandes sont validées. Les commandes sont représentées par une liste de prix et la limite est également un nombre.

```
>>> validees([20,30,60,40,10], 130) # 20+30+60=110 20+30+60+40=150
[20, 30, 60]
>>> validees([20,30,60,40,10], 1000) # 20+30+40+10<1000
[20, 30, 60, 40, 10]
>>> validees([20,30,60,40,10], 10) # La première commande bloque la carte
[]
```

Comparaison d'éléments

Pour les fonctions suivantes, il faut parcourir les listes en comparant les valeurs deux à deux. Il faut faire attention aux valeurs utilisées de **range** :



Exercice 18 : Écrire une fonction `valeurs_consecutives_egales(liste)` qui renvoie **True** si deux éléments consécutifs de `liste` sont égaux. Sinon elle renvoie **False**.

```
>>> valeurs_consecutives_egales([5, 3, 5]) # les deux 5 ne se suivent pas directement
False
>>> valeurs_consecutives_egales([3, 5, 5])
True
>>> valeurs_consecutives_egales([3])
False
```

Exercice 19 : Écrire une fonction `deux_entiers_successifs(liste)` qui renvoie la liste des couples d'entiers consécutifs contenus dans `liste`.

```
>>> deux_entiers_successifs([3, 4])  # 3 et 4 sont des entiers consécutifs
[(3, 4)]
>>> deux_entiers_successifs([1, 3, 4, 2])
[(3, 4)]
>>> deux_entiers_successifs([1, 3, 7, 4, 2])  # 3 et 4, et 1 et 2 ne se suivent pas
[]
>>> deux_entiers_successifs([4, 3, 2, 1]) # les entiers ne sont pas dans le bon ordre
[]
>>> deux_entiers_successifs([1, 2, 3])
[(1, 2), (2, 3)]
```

Exercice 20 : Un texte est un palindrome s'il se lit de la même manière de droite à gauche et de gauche à droite, comme "kayak". Écrire une fonction `palindrome(texte)` qui renvoie un booléen indiquant si `texte` est un palindrome ou pas. Il est possible de ne parcourir que la moitié du texte.

```
>>> palindrome("kayak")
True
>>> palindrome("ABBA")
True
>>> palindrome("chateauaetahc")
True
>>> palindrome("kay ak")
False
>>> palindrome("abcd")
False
```

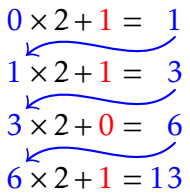
Conversion en binaire

Pour cette partie, les nombres binaires seront représentés par des listes de `0` et de `1`. Ainsi 13_{10} sera représenté par `[1, 1, 0, 1]` :

$$13_{10} = 8 + 4 + 1 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1101_2$$

Pour convertir un nombre binaire en base 10, on peut soit additionner les puissances de 2, soit utiliser l'algorithme suivant :

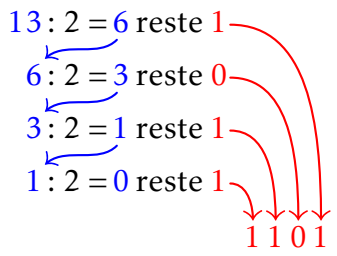
resultat ← 0
Pour tous les chiffres *c* en allant de gauche à droite **:**
    resultat ← 2 × *resultat* + *c*
Renvoyer *resultat*



Exercice 21 : Écrire une fonction `bin_vers_entier(nb_bin)` qui prend une liste non vide de `0` et de 1 `nb_bin` et qui renvoie l'entier correspondant.

```
>>> bin_vers_entier([0])
0
>>> bin_vers_entier([1])
1
>>> bin_vers_entier([1, 0])
2
>>> bin_vers_entier([1, 1, 0, 1])
13
```

Pour convertir un nombre entier en binaire, il est possible de chercher la plus grande puissance de 2 qu'il est possible d'enlever, puis de faire les soustractions successives avec les puissances décroissantes, ou alors on peut utiliser l'algorithme représenté ci contre. Attention, il faut rajouter les restes de droite à gauche. Pour rappel la division entière s'écrit `n // 2` et le reste s'obtient en faisant `n % 2`.



Exercice 22 : Écrire une fonction `entier_vers_bin(nb_entier)` qui prend un entier et renvoie la liste des bits correspondant à ce nombre en binaire.

```
>>> entier_vers_bin(0)
[0]
>>> entier_vers_bin(1)
[1]
>>> entier_vers_bin(2)
[1, 0]
>>> entier_vers_bin(13)
[1, 1, 0, 1]
```

Suites numériques

Aucune des fonctions suivantes n'est récursive.
On considère la suite (u_n) définie par $u_0 = 2$ et $u_{n+1} = 0{,}8u_n + 4$ pour tout $n \geq 0$.

Exercice 23 : Écrire une fonction `terme(n)` qui renvoie la valeur de u_n.

```
>>> terme(0)
2
>>> terme(1)
5.6
>>> terme(10)
18.067264716800004
```

Exercice 24 : Écrire une fonction `seuil(limite)` qui renvoie le premier rang, c'est-à-dire la valeur de n, telle que $u_n \geq$ `limite`.

```
>>> seuil(0)
0
>>> seuil(10)
3
>>> seuil(19)
13
```

Exercice 25 : La suite u_n converge vers 20 lorsque n devient très grand. Écrire une fonction `seuil2(marge)` qui renvoie le premier rang tel que $20 - u_n <$ `marge`.

```
>>> seuil2(0.01)
34
>>> seuil2(0.0001)
55
>>> seuil2(0.0000001)
86
```

La suite de Fibonacci est définie par $f_0 = 1$, $f_1 = 1$ et $f_{n+2} = f_{n+1} + f_n$ pour tout $n \geq 0$.

Exercice 26 : Écrire une fonction `fibo(n)` qui renvoie la valeur du terme de rang `n` de la suite de Fibonacci.

```
>>> fibo(0)
1
>>> fibo(1)
1
>>> fibo(2)
2
>>> fibo(10)
89
>>> fibo(100)
573147844013817084101
```

La suite de Syracuse est définie de la manière suivante :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Exercice 27 : Écrire une fonction `prochain_syracuse(u)` qui calcule la valeur suivant `u` dans la suite de Syracuse.

```
>>> prochain_syracuse(5)
16
>>> prochain_syracuse(100)
50
```

Exercice 28 : Écrire une fonction `suite_syracuse(u)` qui renvoie la liste des valeurs de la suite de Syracuse obtenue en partant de `u`.

```
>>> suite_syracuse(5)
[5, 16, 8, 4, 2, 1]
>>> suite_syracuse(136)
[136, 68, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

On appelle **altitude maximale** d'une suite de Syracuse le plus grand nombre de la suite.

Exercice 29 : Écrire une fonction `altitude_max(u)` qui renvoie l'altitude maximale de la suite en partant de `u`.

```
>>> altitude_max(11)
52
>>> suite_syracuse(11)
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
>>> altitude_max(32)
32
>>> suite_syracuse(32)
[32, 16, 8, 4, 2, 1]
```

On appelle **durée de vol** la longueur d'une suite de Syracuse.

Exercice 30 : Écrire une fonction `duree_vol(u)` qui renvoie la durée de vol en partant de `u`.

```
>>> duree_vol(7)
17
>>> suite_syracuse(7)
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Exercice 31 : Écrire une fonction `record_altitude(u_limite)` qui renvoie la valeur de `u` qui donne la suite avec l'altitude maximale, en partant entre `1` et `u_limite`. La fonction renvoie également l'altitude maximale atteinte. En cas d'égalité en partant de deux valeurs différentes de `u`, c'est la plus petite qui est renvoyée.

```
>>> record_altitude(10)
(7, 52)      # le record est en partant de 7 et on atteint 52
>>> suite_syracuse(7)
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Manipulation de tableaux

Les tableaux sont similaires aux suites, sauf que la longueur est fixée à la création et on ne peut ni ajouter, ni enlever des éléments. En Python, on représente les tableaux par des suites, mais on n'utilise pas `append`.

Exercice 32 : Écrire une fonction `normaliser(tab, a, b)` qui prend deux nombres `a` et `b` tels que `a < b` et qui remplace tous les éléments de `tab` inférieurs à `a` par `a` et tous ceux supérieurs à `b` par `b`.

```
>>> tab = [15, -7, 1, 9, 11]
>>> normaliser(tab, 0, 10)
>>> tab
[10, 0, 1, 9, 10]
```

Exercice 33 : Écrire une fonction `remplacer(val1, val2, tableau)` qui modifie `tableau` en remplaçant toutes les occurrences de `val1` par `val2`.

```
>>> tableau = [2, 3, 7, 1, 4, 2, 5]
>>> remplacer(2, 1, tableau)
>>> tableau
[1, 3, 7, 1, 4, 1, 5]
```

Exercice 34 : Écrire une fonction `echange(tab, i, j)` qui permute la valeur d'indice `i` avec celui d'indice `j` dans `tab`.

```
>>> tab = [15, -7, 1, 9, 11]
>>> echange(tab, 0, 2)
>>> tab
[1, -7, 15, 9, 11]
```

Exercice 35 : Écrire une fonction `miroir(tab)` qui inverse l'ordre des éléments de `tab` Le premier devenant le dernier et ainsi de suite. Vous pouvez utiliser `echange`.

```
>>> tab = [1, 5, 2 ,9, -1, 8]
>>> miroir(tab)
>>> tab
[8, -1, 9, 2, 5, 1]
```

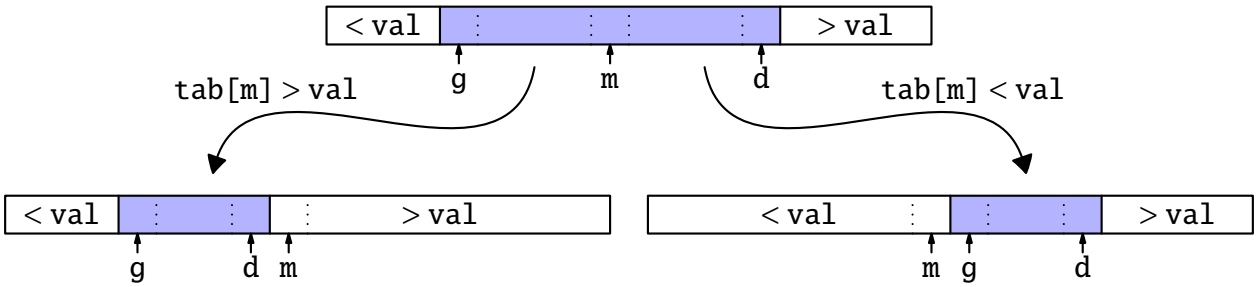
Exercice 36 : Écrire une fonction `ecarts(tab)` qui renvoie un nouveau tableau dont la valeur du premier élément est `tab[0]` et la valeur de l'élément d'indice `i` > 0 est la différence entre `tab[i]` et `tab[i-1]`. On suppose que `tab` n'est pas vide.

```
>>> ecarts([7, 9, 1, 6, 3])
[7, 2, -8, 5, -3]
```

Exercice 37 : Écrire une fonction `ranger(tab)` qui prend un tableau contenant des 0 et des 1 et modifie ce tableau en mettant tous les 0 au début et les 1 à la fin.

```
>>> tab = [1, 0, 0, 0, 1, 0]
>>> ecarts(tab)
>>> tab
[0, 0, 0, 0, 1, 1]
```

Pour faire une recherche dichotomique dans un tableau trié, on part de l'indice au milieu et selon si la valeur cherchée est plus petite ou plus grande que la valeur observée, on continue la recherche à gauche ou à droite. Pour cela, on mémorise le plus petit indice (`g`) et l'indice le plus grand (`d`) pouvant contenir la valeur. On résume cela par le schéma suivant :



Exercice 38 : Écrire une fonction `dichotomie(val, tab)` qui prend un tableau trié dans l'ordre croissant et qui renvoie l'indice de `val` s'il est dans `tab` s'il y est et **`None`** sinon. La recherche est faite par dichotomie. En plus de l'indice, la fonction renvoie également le nombre d'étapes nécessaires pour arriver à ce résultat.

```
>>> tab = [1, 2, 3, 5, 7, 9, 12, 35, 79, 90, 179]
>>> recherche_dichotomique(79, tab)
(8, 2)
>>> recherche_dichotomique(1, tab)
(0, 3)
>>> recherche_dichotomique(2, tab)
(1, 4)
>>> recherche_dichotomique(4, tab)
(None, 3)
>>> recherche_dichotomique(100, tab)
(None, 4)
```

Algorithmes gloutons

Exercice 39 : Écrire la fonction `rendu_monnaie(montant, liste_valeurs)` qui renvoie la liste des nombres de billets et pièces nécessaire pour obtenir `montant`, en en utilisant le moins possible. Les valeurs des billets et des pièces sont données dans l'ordre décroissant. Vous pourrez par exemple utiliser un algorithme glouton. Vous pouvez définir la liste `euros` dans votre fichier :

```
euros = [500, 200, 100, 50, 20, 10, 5, 2, 1]
```

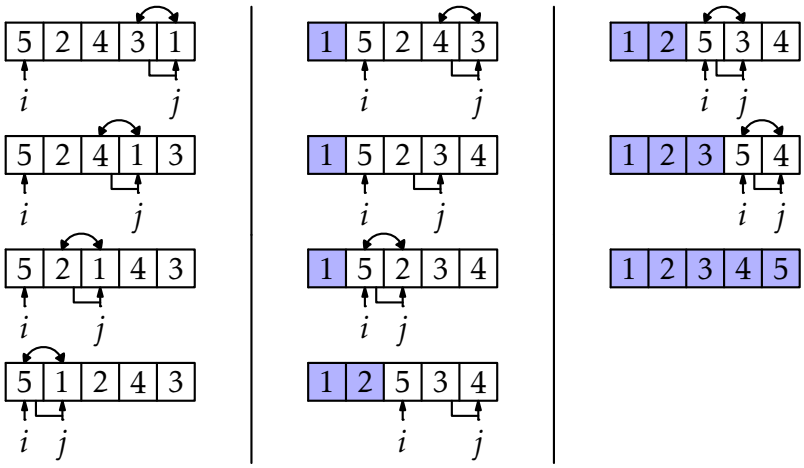
```
>>>
>>> rendu_monnaie(358, euros)
[200, 100, 50, 5, 2, 1]
>>> sum([200, 100, 50, 5, 2, 1])
358
>>> rendu_monnaie(740, euros)
[500, 200, 20, 20]
>>> sum([500, 200, 20, 20])
740
```

La fonction **`sum`**`(liste)` renvoie la somme de tous les éléments d'une liste de nombres.

Tris de tableaux

Pour cette partie, on pourra utiliser la fonction `echange(tab, i, j)` de l'exercice 34.

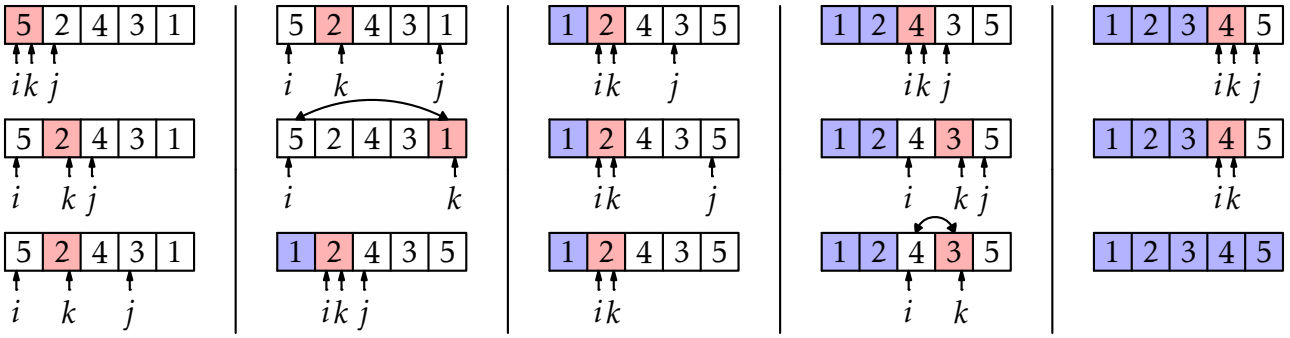
Le principe du **tri à bulles** consiste à partir de la fin et de remonter jusqu'au début du tableau en permuttant les couples de valeurs successifs qui ne sont pas dans le bon ordre. Puis on recommence mais en s'arrêtant à l'élément d'indice 2. Et ainsi de suite, en allant de moins en moins loin à chaque itération.



Exercice 40 : Écrire une fonction `tri_bulles(tab)` qui trie les valeurs de `tab` en utilisant un tri à bulles.

```
>>> tab = [9, 2, 4, 1, 7, 6]
>>> tri_bulles(tab)
>>> tab
[1, 2, 4, 6, 7, 9]
```

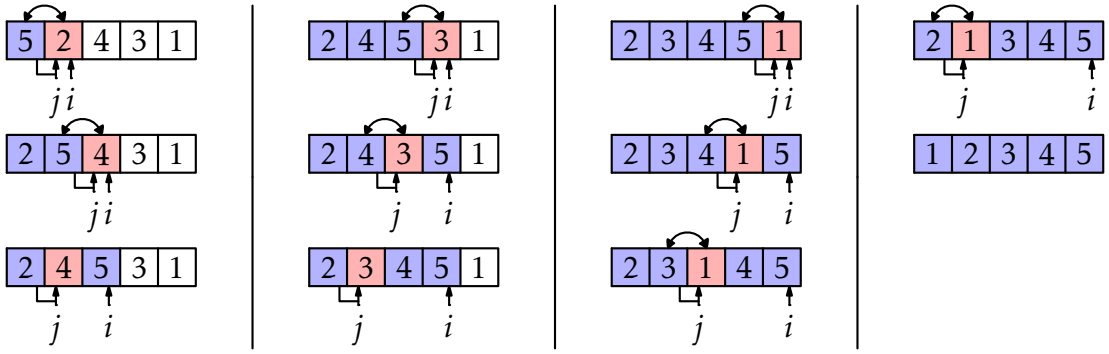
Le **tri par sélection** consiste à parcourir le tableau pour trouver le minimum et à l'échanger avec l'élément d'indice 0. Puis le minimum entre l'indice 1 et l'indice final est échangé avec l'élément d'indice 1, et ainsi de suite.



Exercice 41 : Écrire une fonction `tri_selection(tab)` qui trie les valeurs de `tab` en utilisant un tri par sélection.

```
>>> tab = [9, 2, 4, 1, 7, 6]
>>> tri_selection(tab)
>>> tab
[1, 2, 4, 6, 7, 9]
```

Le **tri par insertion** consiste à parcourir le tableau de gauche à droite et pour chaque nouvelle valeur vue, l'insérer à gauche au bon endroit et ainsi de suite.



Exercice 42 : Écrire une fonction `tri_insertion(tab)` qui trie les valeurs de `tab` en utilisant un tri par insertion.

```
>>> tab = [9, 2, 4, 1, 7, 6]
>>> tri_insertion(tab)
>>> tab
[1, 2, 4, 6, 7, 9]
```