Numérique et Sciences Informatiques
Chapitre V - Les tableaux

Classe de première

I. Les tableaux indexés

I.1. Généralités

Définition

Un tableau est une collection ordonnée mutable d'éléments.

Remarque

On s'intéressera uniquement aux tableaux dont les éléments sont de même type.

Propriété

Un tableau étant ordonné, chacun de ses élément est indexé par un indice qui indique l'emplacement de l'élément dans le tableau.
Un tableau étant mutable, on peut changer la valeur d'un de ses éléments.

Remarque

En Python :

- un tableau est de type `list`.
- pour créer un tableau, on écrit les éléments entre crochets séparés par une virgule.
- pour créer un tableau vide, on écrit uniquement des crochets.
- pour travailler sur un élément dont on connaît l'indice, on écrit le tableau, suivi de crochets dans lesquels on indique l'indice.

Exemple

```
>>> mon_tableau = [7, 8, 3, 2, 5, 7, 9, 6]
>>> type(mon_tableau)
<class 'list'>
>>> mon_tableau[3]
 2
>>> mon_tableau[1] = 3
>>> mon_tableau
 [7, 3, 3, 2, 5, 7, 9, 6]
```

Remarque

Si une fonction modifie localement un tableau mis en paramètre de la fonction, ce dernier sera modifié globalement. C'est ce qu'on appelle un **effet de bord**.
Il n'y a pas besoin de renvoyer le tableau à la fin de la fonction.

Exemple

```
def ajoute1(tableau):
    indice = 0
    for element in tableau:
        tableau[indice] = element + 1
        indice += 1
```

Voyons ce qui se passe avec Pythontutor
Un autre exemple sans effet de bord.

II. Construction de tableaux en compréhension

Propriété

On peut créer des tableaux en compréhension, c'est-à-dire en utilisant une formule qui permet de calculer chaque éléments, une boucle `for` et éventuellement des instructions conditionnelles.

Exemple

```
>>> tab = [i for i in range(5)]
>>> tab
 [0, 1, 2, 3, 4]
>>> tab2 = [3 * chut + 1 for chut in range(3)]
>>> tab2
 [1, 4, 7]
>>> tab3 = [5 - i for i in range(5) if i % 2 ==0]
>>> tab3
 [5, 3, 1]
```

III. Tableau de tableaux

Définition

Les éléments d'un tableau peuvent être des tableaux.
Pour accéder à un élément intérieur à un tableau de tableaux, il faut une double indexation.

Propriété

Dans le cas où tous les éléments d'un tableau sont des tableaux ayant le même nombre d'éléments, le tableau est appelée **matrice**.
Le premier indice sera le numéro de ligne et le deuxième le numéro de colonne. On peut donc représenter une matrice ainsi :

$$\begin{bmatrix} elt_{(0;0)} & elt_{(0;1)} & \dots & elt_{(0;p)} \\ elt_{(1;0)} & elt_{(1;1)} & \dots & elt_{(1;p)} \\ \vdots & & \ddots & \vdots \\ elt_{(n;0)} & elt_{(n;1)} & \dots & elt_{(n;p)} \end{bmatrix}$$

Remarque

On peut créer des tableaux de tableaux en compréhension

Exemple

```
>>> tab = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> tab[0][0]
 1
>>> tab[1][2]
 6
>>> tab2 = [[ 3 * i + j for j in range(3)] for i in range(3)]
>>> tab2
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> tab3 = [[1 + j + 1 for j in range(i)] for i in range(4)]
>>> tab3
 [[], [2], [3, 4], [4, 5, 6]]
```

IV. Méthodes et fonctions avec des tableaux

IV.1. Quelques méthodes

Les tableaux possèdent des méthodes.

IV.1.a. append(x)

Définition

La méthode `mon_tableau.append(x)` ajoute l'élément `x` à la fin du tableau.

Exemple

```
>>> tab = [1, 2, 3]
>>> tab.append(8)
>>> tab
 [1, 2, 3, 8]
```

IV.1.b. pop([indice])

Définition

La méthode `mon_tableau.pop([indice])` enlève du tableau l'élément d'indice `indice` et la renvoie. Si aucun indice n'est spécifié, cette méthode enlève et renvoie le dernier élément du tableau.

Exemple

```
>>> tab = [1, 2, 3, 8]
>>> tab.pop(1)
 2
>>> tab
 [1, 3, 8]
>>> tab.pop()
 8
>>> tab
 [1, 3]
```

IV.1.c. copy()

Définition

La méthode `mon_tableau.copy()` renvoie une copie superficielle du tableau.

Exemple

```
>>> tab = [1, 2, 3, 8]
>>> tab2 = tab.copy()
>>> tab2
 [1, 2, 3, 8]
>>> tab3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> tab4 = tab3.copy()
>>> tab4[0][1] = 5
>>> tab3
 [[1, 5, 3], [4, 5, 6], [7, 8, 9]]
```

Voici l'explication avec Pythontutor Pour éviter ce problème et avoir une copie en profondeur, on utilisera la fonction `deepcopy(mon_tableau)` de la bibliothèque `copy`.

```
>>> import copy
>>> tab3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> tab4 = copy.deepcopy(tab3)
>>> tab4[0][1] = 5
>>> tab3
 [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

IV.2. La fonction len()

Définition

La fonction `len(tableau)` renvoie la **longueur** du tableau mis en paramètre, c'est à dire le nombre d'éléments qui le composent.

Exemple

```
>>> tab = [1, 2, 3, 8]
>>> len(tab)
 4
>>> tab3 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> len(tab3)
 3
>>> tab4 = []
>>> len(tab4)
 0
```

IV.3. La fonction list()

Définition

La fonction `list(collection)` renvoie la collection mise en paramètre sous forme d'un tableau.

Exemple

```
>>> chaine = "abcd"
>>> list(chaine)
['a', 'b', 'c', 'd']
```

V. Algorithmes sur les tableaux

Correction d'un algorithme

Lorsqu'on s'intéresse aux algorithmes, il faut toujours vérifier que l'algorithme est **correct**, c'est-à-dire qu'il se termine.

Complexité en temps

De plus, on calculera toujours la **complexité en temps** de l'algorithme.
Cette complexité est calculée en fonction de la taille de l'entrée et compte le nombre d'étapes de calcul (ou le nombre de comparaisons) nécessaires à l'obtention du résultat. Ce nombre d'étape est appelé le **coût** de l'algorithme.
On peut calculer le coût dans le meilleur des cas (le coût est minimal), dans le pire des cas (le coût est maximal) ou en moyenne (moyenne de tous les coûts possibles).

Notation grand O de Landau

Soit une fonction $A : n \mapsto A(n)$.
On note $A(n) = \mathcal{O}(f(n))$ si et seulement si il existe deux réels k et p tels que $A(n) = k \times f(n) + p$.
Cette notation est appelée notation grand O de Landau.

Remarque

On ne s'intéressera en NSI qu'à la notation grand O de Landau où $f : n \mapsto f(n)$ une fonction du type :

- n^p avec $p \in \mathbb{N}$
- $log_2(n)$ où la fonction log_2 possède la propriété suivante :

$$\forall k \in \mathbb{N}, \log_2\left(2^k\right) = k$$

Exemple

$A(n) = 5,7n^2$
Alors $A(n) = \mathcal{O}\left(n^2\right)$.

V.1. Parcours séquentiel d'un tableau

V.1.a. Recherche d'un élément dans un tableau

Principe

Lorsqu'on recherche un élément dans un tableau :

- on compare l'élément cherché à chacun des éléments du tableau
- si un des éléments est égal à l'élément cherché alors on a trouvé un emplacement
- sinon, il n'est pas dans le tableau

Algorithme 1 : recherche d'un élément dans un tableau

Entrées : tab : un tableau d'éléments comparables
elt : un objet comparable aux éléments du tableau
res←False
pour chaque *element de tab* **faire**
  si *element* = *elt* **alors**
    res ←True
  fin
fin
Sorties : res

```
1 def recherche(tab, elt):
2     res = False
3     for element in tab:
4         if element == elt:
5             res = True
6     return res
```

Calcul de la complexité de l'algorithme
Soit n la taille du tableau.
Dans tous les cas, on parcourt le tableau entier donc le coût est n donc en $\mathcal{O}(n)$ ou encore linéaire.

Correction de l'algorithme
Dans tous les cas, on parcourt le tableau entier donc l'algorithme se termine. L'algorithme est donc correct.

Remarque

En général, on peut arrêter l'algorithme dès qu'on trouve un emplacement.

```
1 def recherche(tab, elt):
2     for element in tab:
3         if element == elt:
4             return True
5     return False
```

V.1.b. Recherche du minimum

Algorithme 2 : recherche du minimum

Entrées : tab : un tableau d'éléments comparables
mini←tab[0]
pour chaque *element de tab* **faire**
  si *element* < *mini* **alors**
    mini ←element
  fin
fin
Sorties : mini

```
def recherche_min(tab):
    mini = tab[0]
    for element in tab:
        if element < mini:
            mini = element
    return mini
```

Calcul de la complexité de l'algorithme
Soit n la taille du tableau.
Dans tous les cas, on parcourt le tableau entier donc le coût est n donc en $\mathcal{O}(n)$ ou encore linéaire.

Correction de l'algorithme
Dans tous les cas, on parcourt le tableau entier donc l'algorithme se termine. L'algorithme est donc correct.

V.1.c. Recherche du maximum

Algorithme 3 : recherche du maximum

Entrées : tab : un tableau d'éléments comparables
maxi←tab[0]
pour chaque *element de tab* **faire**
  si *element* > *maxi* **alors**
    maxi ←element
  fin
fin
Sorties : maxi

```
def recherche_max(tab):
    maxi = tab[0]
    for element in tab:
        if element > maxi:
            maxi = element
    return maxi
```

Calcul de la complexité de l'algorithme
Pour les mêmes raison que l'algorithme précédent, le coût est donc en $\mathcal{O}(n)$ ou encore linéaire.

Correction de l'algorithme
Dans tous les cas, on parcourt le tableau entier donc l'algorithme se termine. L'algorithme est donc correct.

V.1.d. Calcul de la somme

Algorithme 4 : somme des éléments d'un tableau

Entrées : tab : un tableau de nombres
resultat←0
pour chaque *element de tab* **faire**
| resultat ←resultat + element
fin
Sorties : resultat

```
1  def somme(tab):
2      resultat = 0
3      for element in tab:
4          resultat = resultat + element
5      return resultat
```

Calcul de la complexité de l'algorithme
Pour les mêmes raison que l'algorithme précédent, le coût est donc en $\mathcal{O}(n)$ ou encore linéaire.

Correction de l'algorithme
Dans tous les cas, on parcourt le tableau entier donc l'algorithme se termine. L'algorithme est donc correct.

V.1.e. Calcul de la moyenne

Algorithme 5 : moyenne des éléments d'un tableau

Entrées : tab : un tableau de nombres
resultat←0
compteur←0
pour chaque *element de tab* **faire**
| resultat ←resultat + element
| compteur ←compteur + 1
fin
Sorties : resultat / compteur

```
1  def moyenne(tab):
2      resultat = 0
3      compteur = 0
4      for element in tab:
5          resultat = resultat + element
6          compteur = compteur + 1
7      return resultat/compteur
```

Calcul de la complexité de l'algorithme
Pour les mêmes raison que l'algorithme précédent, le coût est donc en $\mathcal{O}(n)$ ou encore linéaire.

Correction de l'algorithme
Dans tous les cas, on parcourt le tableau entier donc l'algorithme se termine. L'algorithme est donc correct.

V.2. Tri de tableaux

V.2.a. par selection

Principe

Pour trier un tableau dans l'ordre croissant par l'algorithme de tri par sélection :

- on cherche le plus petit élément du sous-tableau
- on l'échange avec le premier élément du tableau
- on cherche le deuxième plus petit élément du tableau
- on l'échange avec le deuxième élément du tableau
- et ainsi de suite jusqu'au dernier élément du tableau

Algorithme 6 : tri par selection

Entrées : tab : un tableau d'éléments comparables
Résultat : un tableau d'éléments triés dans l'ordre croissant
n←len(tab)
pour *i* **allant de** *0* **à** *n-1* **faire**
  j ←indice du plus petit élément des n-j derniers éléments
  on échange tab[i] et tab[j]
fin

```
def tri_selection(tab):
    for k in range(len(tab)):
        indice_mini = k
        for j in range(k+1, len(tab))
            if tab[j] < tab[k]:
                indice_mini = j
        tab[k], tab[indice_mini] = tab[indice_mini], tab[k]
```

Calcul de la complexité de l'algorithme

À l'étape k, on compare toujours le k-ème élément avec les $n-k$ derniers éléments. Il y a donc $n-k$ comparaisons. En tout, le nombre de comparaisons est donc :

$$(n-1)+(n-2)+\ldots+2+1=\frac{n(n-1)}{2}\approx\frac{n\times n}{2}=\frac{1}{2}n^2=\mathcal{O}\left(n^2\right)$$

On dit que le coût est quadratique.

Correction de l'algorithme

- 1ère étape : déterminons un invariant de boucle, c'est-à-dire une propriété $\mathcal{P}_k$ qui reste vraie à l'étape k de la boucle.
  Pour notre algorithme de tri, l'invariant est $\mathcal{P}_k$: « à l'étape k, $tab[0:k]$, le sous-tableau de tab des k premiers éléments est trié dans l'ordre croissant et les éléments de $tab[k:]$ sont tous supérieurs ou égaux à $tab[k-1]$ ».
- 2ème étape : initialisation
  Avant l'étape 1, On échange le plus petit élément de tab avec le premier élément de tab donc :
  - $tab[0:1]$ ne contient qu'un élément donc est trié.
  - Les éléméents de $tab[1:]$ sont tous supérieurs ou égaux à $tab[0]$.

  Donc $\mathcal{P}_1$ est vraie.

- 3ème étape : conservation
  On suppose que $\mathcal{P}_k$ est vraie, c'est-à-dire que $tab[0 : k]$ est un tableau trié dans l'ordre croissant et que $tab[k :]$ ne contient que des éléments supérieurs ou égaux à $tab[k-1]$.
  Montrons que $tab[0 : k+1]$ sera triée dans l'ordre croissant à la fin de l'étape $k+1$ et que tous les éléments de $tab[k+1 :]$ sont supérieurs ou égaux à $tab[k]$.

  On échange le plus petit des éléments de $tab[k :]$ avec $tab[k]$ donc tous les éléments de $tab[k+1 :]$ sont supérieurs ou égaux à $tab[k]$.
  De plus, ce plus petit élément est supérieur ou égal à tous les éléments de $tab[0 : k]$ car $\mathcal{P}_k$ est vraie.
  Donc on a bien $tab[0 : k+1]$ est trié dans l'ordre croissant et tous les éléments de $tab[k+1 :]$ sont supérieurs ou égaux à $tab[k]$.
  Ainsi $\mathcal{P}_{k+1}$ est vraie.
  Il y a donc conservation de la propriété d'une étape à la suivante.

- 3ème étape : Correction
  La boucle se termine lorsque $k = n-1$.
  On a donc $tab[0 : n-1+1] = tab[0 : n] = tab$ qui est trié dans l'ordre croissant et les éléments de $tab[n :] = []$ sont tous supérieurs ou égaux à $tab[n-1]$.
  La propriété $\mathcal{P}_n$ est encore vraie.
  tab est trié dans l'ordre croissant.
  L'algorithme est donc correct.

V.2.b. par insertion

Principe

Pour trier un tableau dans l'ordre croissant par l'algorithme de tri par insertion :

- on prend un élément du tableau
- on le place dans le tableau résultat
- on prend un deuxième élément du tableau
- on le place dans le tableau résultat afin que ce dernier soit trié dans l'ordre croissant
- et ainsi de suite jusqu'au dernier élément du tableau

Algorithme 7 : tri par insertion

Entrées : tab : un tableau d'éléments comparables
Résultat : un tableau d'éléments triés dans l'ordre croissant

n←len(tab)
pour *j* **allant de** *1* **à** *n-1* **faire**
  insérer l'élément $tab[j]$ dans le tableau des j premiers éléments de tab afin que $tab[0 : j + 1]$ soit trié dans l'ordre croissant
fin

Algorithme 8 : inserer

Entrées : tab : un tableau d'éléments comparables dont les k premiers elements sont triés
k : l'indice d'un element de tab
Résultat : les $k + 1$ premiers éléments de tab sont triés

element←tab[k]
indice ←k-1
tant que *indice >=0* **et** *element < tab[indice]* **faire**
  tab[indice + 1] = tab[indice] indice ←indice - 1
fin
tab[indice + 1] = element

```python
def inserer(tab, k):
    element = tab[j]
    indice = k-1
    while indice >= 0 and element < tab[indice] :
        tab[indice + 1] = tab[indice]
        indice -= 1
    tab[indice + 1] = element

def tri_insertion(tab):
    n = len(tab)
    for k in range(1, n):
        inserer(tab, k)
```

Calcul de la complexité de l'algorithme

- Dans le meilleur des cas : le tableau est déjà trié.
  Donc le fait d'insérer l'élément un element dans le sous-tableau de k éléments ne fera qu'une comparaison.
  On a alors, en tout, $n-1 = \mathcal{O}(n)$ comparaisons.
  Dans le meilleur des cas, le coût est linéaire.

- Dans le pire des cas : le tableau est trié dans l'ordre décroissant.
  Donc le fait d'insérer l'élément un element dans le sous-tableau de k éléments fera k comparaisons.
  On a alors, en tout, $(n-1)+(n-2)+...+2+1 = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$ comparaisons.
  Dans le pire des cas, le coût est quadratique.

- On admet que le coût en moyenne est $\mathcal{O}(n^2)$, c'est-à-dire quadratique.

Correction de l'algorithme

- 1ère étape : déterminons un invariant de boucle, c'est-à-dire une propriété $\mathcal{P}_k$ qui reste vraie à l'étape k de la boucle.
  Pour notre algorithme de tri par insertion, l'invariant est $\mathcal{P}_k$: « à l'étape k, $tab[0:k]$, le sous-tableau de tab des k premiers éléments est trié ».

- 2ème étape : initialisation
  Avant l'étape 1, $tab[0:1]$ est composé d'un seul élément donc est trié dans l'ordre croissant. Donc $\mathcal{P}_1$ est vraie.

- 3ème étape : conservation
  On suppose que $\mathcal{P}_k$ est vraie, c'est-à-dire que $tab[0:k]$ est un tableau trié dans l'ordre croissant.
  Montrons que $tab[0:k+1]$ sera triée dans l'ordre croissant à la fin de l'étape $k+1$.
  On insère l'élément $tab[j]$ dans le tableau des j premiers éléments, ce qui donne un tableau de $j+1$ éléments.
  L'insertion est telle que le tableau reste trié dans l'ordre croissant.
  Donc le tableau des $j+1$ premiers éléments de tab est trié dans l'ordre croissant.
  Il y a bien conservation de la propriété.
  La propriété $\mathcal{P}_k$ est donc vraie.

- 3ème étape : Correction
  La boucle se termine lorsque $k = n-1$.
  On a donc $tab[0:n-1+1] = tab[0:n] = tab$ qui est trié dans l'ordre croissant.
  La propriété $\mathcal{P}_n$ est encore vraie.
  tab est trié dans l'ordre croissant.
  L'algorithme est donc correct.

V.3. Recherche par dichotomie dans un tableau trié

Principe

Lorsqu'on recherche un élément par dichotomie dans un tableau trié :

- on compare l'élément cherché à un autre élément du tableau (souvent celui du milieu).
- si les deux éléments comparés sont égaux, alors on a trouvé l'élément chérché
- sinon, on s'intéresse au bon sous-tableau (de droite ou de gauche) dans lequel l'élément cherché est susceptible d'être
- on réitère les étapes 1 à 3 avec le sous-tableau choisi.

Algorithme 9 : recherche dichotomique

Entrées : tab : un tableau d'éléments triés dans l'ordre croissant
element : un element comparable aux elements de *tab*
Résultat : l'indice de l'élément cherché s'il est dans le tableau, -1 sinon

gauche $\leftarrow$0
droite $\leftarrow$len(tab) - 1 milieu $\leftarrow$(gauche + droite) // 2
tant que *gauche* $\leq$ *droite* **et** *tab[milieu]* $\neq$ *element* **faire**
  si *tab[milieu]* $>$ *element* **alors**
    droite $\leftarrow$milieu - 1
  sinon
    gauche $\leftarrow$milieu + 1
  fin
fin
si *tab[milieu]* $=$ *element* **alors**
  resultat $\leftarrow$milieu
sinon
  resultat $\leftarrow -1$
fin
Sorties : resultat

```python
def recherche_dicho(tab, element):
    gauche = 0
    droite = len(tab) - 1
    milieu = (gauche + droite) // 2
    while gauche <= droite and tab[milieu] != element:
        if tab[milieu] > element:
            droite = milieu - 1
        else :
            gauche = milieu + 1
        milieu = (gauche + droite) // 2
    if tab[milieu] == element:
        resultat = milieu
    else:
        resultat = -1
    return resultat
```

Calcul de la complexité de l'algorithme

- Dans le meilleur des cas, l'élément cherché est l'élément du milieu du tableau donc le coût est 1.
- Dans le pire des cas, l'élément n'appartient pas au tableau.
  A chaque passage dans la boucle, on s'intéresse à un sous-tableu de taille moitié moindre. Il faut donc se poser la question suivante : au bout de combien d'itération, arrivons-nous à un sous-tableau de taille 1 ?
  En effet avec 1 élément restant on saura si l'élément cherché est dans le tableau.
  Le problème revient donc mathématiquement, en posant k le nombre d'itérations et n la taille du tableau, à résoudre l'équation suivante :
  $\frac{n}{2^k} = 1$
  ou encore $2^k = n$
  La fonction mathématique logarithme en base 2 va permettre de répondre à ce problème :
  $\log_2(2^k) = \log_2(n)$
  $k = \log_2(n) = \mathcal{O}(\log_2(n))$
  Ainsi le coût dans le pire des cas est logarithmique.
- Dans tous les autres cas, c'est-à-dire dans le cas où l'élément cherché est dans le tableau, il faut résoudre la même équation puisqu'on cherche au bout de combien d'itérations on arrive à une sous-liste de 1 élément (l'élément cherché).
  Donc le coût est logarithmique.

Correction de l'algorithme
Posons ig_k et id_k les indices qui déterminent le sous-tableau lors de la k-ième itération.
Posons ensuite $im_k = \frac{ig_k + id_k}{2}$.
On a alors $ig_0 = 0$ et $id_0 = n - 1$.

Supposons que nous sommes à la k-ième itération. On a quatre possibilités :

- $ig_k > id_k$: l'algorithme se termine car on n'entre pas dans la boucle.
- $tab[im_k] = elt$: l'algorithme se termine car on n'entre pas dans la boucle.
- $ig_k \leqslant id_k$ et $elt < liste[im_k]$:
  On a alors $ig_{k+1} = ig_k$ et $id_{k+1} = im_k - 1$
  On peut en déduire que $id_{k+1} - ig_{k+1} = im_k - 1 - ig_k$
  $id_{k+1} - ig_{k+1} = \frac{ig_k + id_k}{2} - ig_k - 1$
  $id_{k+1} - ig_{k+1} = \frac{id_k - ig_k}{2} - 1$
  $id_{k+1} - ig_{k+1} < id_k - ig_k$
- $ig_k \leqslant id_k$ et $elt > liste[im_k]$:
  On a alors $ig_{k+1} = im_k + 1$ et $id_{k+1} = id_k$
  On peut en déduire que $id_{k+1} - ig_{k+1} = id_k - (im_k + 1)$
  $id_{k+1} - ig_{k+1} = id_k - im_k - 1$
  $id_{k+1} - ig_{k+1} = id_k - \frac{ig_k + id_k}{2} - 1$
  $id_{k+1} - ig_{k+1} = \frac{id_k - ig_k}{2} - 1$
  $id_{k+1} - ig_{k+1} < id_k - ig_k$

Dans ces deux derniers cas, nous avons $id_{k+1} - ig_{k+1} < id_k - ig_k$.
Donc la suite $(id_k - ig_k)$ est strictement décroissante.
Or $\exists p \in \mathbb{N}, \forall k \in \mathbb{N}_p, ig_k \leqslant id_k$ ou encore $id_k - ig_k \geq 0$.
Donc la suite est convergente et donc « s'arrête » au bout d'un moment.

Donc $\exists q \in \mathbb{N}$ tel que :

- Soit $ig_q > id_q$: l'algorithme se termine car on n'entre pas dans la boucle.
- Soit $tab[im_q] = elt$: l'algorithme se termine car on n'entre pas dans la boucle.

VI. Exercices

VI.1. Exercice 1

Créer des tableaux en compréhension : les nombres pairs , les 10 premières puissances de 2

1. Le tableau des 10 premiers nombres pairs positifs.
2. Le tableau des 10 premières puissances de 2.
3. la matrice carrée comprenant 6 lignes des issues d'un lancer de deux dés à 6 faces.
4. la matrice comprenant 7 lignes et 10 colonnes des nombres de 0 à 69.

VI.2. Exercice 2

Créer une fonction `nb_occurrences(tab, element)` qui renvoie le n'ombre d'occurrences de la valeur `element` dans le tableau `tab`.
Exemple :

```
>>> nb_occurrences([1, 2, 3, 4, 1, 6, 2, 1, 8, 3], 1)
3
>>> nb_occurrences([1, 2, 3, 4, 1, 6, 2, 1, 8, 3], 3)
2
>>> nb_occurrences([1, 2, 3, 4, 1, 6, 2, 1, 8, 3], 8)
1
>>> nb_occurrences([1, 2, 3, 4, 1, 6, 2, 1, 8, 3], 9)
0
```

VI.3. Exercice 3

Le tri à bulles est un algorithme de tri consistant à comparer répétitivement les éléments consécutifs d'un tableau et à les permuter s'ils sont mal triés.

Principe du tri à bulles

I. (a) en commençant à l'indice 0
   (b) on compare un élément et son suivant.
   (c) s'ils ne sont pas dans l'ordre croissant, on les permute, sinon on ne fait rien.
   (d) on incrémente l'indice.
   (e) on réitère les étapes (b) à (d) jusqu'à avoir parcouru tout le tableau.

II. le dernier élément est donc bien placé.

III. on réitère l'étape I mais on s'arrête après avoir parcouru le tableau à l'exception des éléments bien placés.

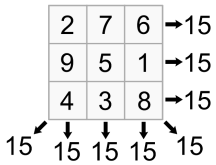
1. Déterminer la complexité de l'algorithme de tri à bulles.
2. Implémenter une fonction `tri_bulles(tab)` qui trie un tableau `tab` selon la méthode du tri à bulle.
3. Améliorer l'algorithme pour qu'il s'arrête dès qu'il n'y a plus de permutation lors de l'étape I.

VI.4. Exercice 4

1. Implémenter une fonction `dec_to_hex(nb: int) -> str` qui convertit un nombre du système décimal en sa représentation en hexadécimal.
2. Implémenter une fonction `hex_to_dec(nb: str) -> int` qui convertit un nombre du système hexadécimal en sa valeur dans le système décimal.
3. Implémenter une fonction `bin_to_hex(nb: str) -> str` qui convertit un nombre du système binaire en sa représentation en hexadécimal.
4. Implémenter une fonction `hex_to_dec(nb: str) -> str` qui convertit un nombre du système hexadécimal en sa valeur dans le système binaire.

VI.5. Exercice 5

En mathématiques, un carré magique d'**ordre n** est composé de n^2 entiers strictement positifs, écrits sous la forme d'un tableau carré.
Ces nombres sont disposés de sorte que leurs sommes sur chaque rangée, sur chaque colonne et sur chaque diagonale principale soient égales.
On nomme alors **constante magique** (et parfois densité) la valeur de ces sommes.
Un carré magique **normal** est un cas particulier de carré magique, constitué de tous les nombres entiers de 1 à n^2, où n est l'ordre du carré.
Exemple de carré magique :



Pour plus d'information sur les carré magiques, vous pouvez consulter la page Wikipedia.

1. (a) Compléter le carré suivant pour qu'il soit magique.

4	14	15	1
	7		12
5			
16			13

   (b) Quelle est sa constante magique ?
   (c) Est-il normal ?

2. (a) Quelle est la constante magique d'un carré magique normal d'ordre 3 ? d'ordre 4 ? d'ordre 5 ?
   (b) Conjecturer le lien entre n et la constante magique d'un carré magique normal d'ordre n.
   (c) Démontrer la conjecture.

3. On considère les variables suivantes :
```
Carre1 = [[1, 15, 24, 14], [12, 28, 3, 5], [21, 6, 10, 4], [20, 2, 7, 18]]
Carre2 = [[16, 3, 2, 13], [10, 8, 11, 5], [7, 9, 6, 12], [1, 14, 15, 4]]
```
   (a) Crer une fonction `ordre(carre: list) -> int` que renvoie l'ordre du carré.
   (b) Créer une fonction `somme_ligne(carre: list) -> list` qui renvoie le tableau des sommes des lignes du carré.
   (c) Créer une fonction `somme_colonne(carre: list) -> list` qui renvoie le tableau des sommes des colonnes du carré.

(d) Créer une fonction `somme_diago(carre: list) -> list` qui renvoie le tableau des sommes des deux diagonales du carré.

(e) Créer un prédicat `est_magique(carre:list) -> bool` qui renvoie `True` sir le carré est magique, `False` sinon.

(f) Créer un prédicat `est_magique_normal(carre:list) -> bool` qui renvoie `True` sir le carré est magique et normal, `False` sinon.

(g) Dans le fichier `carre.csv`, se trouve un carré de nombres.

i. Créer une fonction qui récupère les données du fichier et les renvoie sous forme d'un tableau de tableau.

ii. Les données du fichier `carre.csv` correspondent-elles à un carré magique ?