Chapitre 3
Tableaux & Tris à balayage

Le cours en questions

1 Les tableaux

1.1 Introduction

Implant Les listes en `python` (objets de la classe `<list>`) s'avèrent essentielle dans ce langage. Elles constituent une structure de donnée séquentielle permettant de stocker une collection d'éléments (d'objets) et d'y avoir accès via leur indice.

De leur implantation dépend le coût de leurs diverses méthodes et autres fonctions :

- Quel est le coût de l'accès à un élément par son indice ?
- Quel est le coût de l'ajout d'un élément à une liste ?
- Quel est le coût de la suppression d'un élément d'une liste ?
- Quel est le coût de la fonction `len` (qui retourne la longueur d'une liste passée en argument) ?
- Quel est le coût de l'extraction d'une sous-liste ?
- etc...

En parlant de *coût* c'est de leur complexité en temps (dans le pire, le meilleur des cas, ...) et de leur complexité en espace (dans le pire, le meilleur des cas, ...) dont nous parlons. Les complexités des algorithmes utilisant des listes en dépendent.

C'est de la **structure de donnée théorique** utilisée pour l'implantation des listes que dépendent ces coûts.


La complexité des opérations sur les listes dépend de la structure de donnée théorique utilisée pour leur implantation.

Et qu'en est-il des tableaux de `numpy` (objets de la classe `numpy.ndarray`) ?

1.2 Tableaux (statiques)

1.2.1 Structure de donnée

En informatique théorique un **tableau** (ou tableau statique) est une structure de donnée :

- *séquentielle* : les éléments contenues le sont à la suite : il y a un premier élément, un deuxième élément, etc..., un dernier élément,
- *indexée* : les éléments sont repérés grâce à un indice (intimement lié à leur position),
- *homogène* : toutes les données contenues sont de même type.

telle que les données sont contenues dans la mémoire principale de façon contiguë (à la suite les unes des autres).

Lors de la création d'un tableau de taille N, un espace contiguë de N fois la taille nécessaire pour stocker une donnée du type considéré est réservé en mémoire.

éléments	elt1	elt2	elt3	...	eltN
indices	0	1	2	...	N-1

- L'accès à un élément du tableau (en lecture ou en écriture) se fait à l'aide de son indice i en temps constant $O(1)$; en effet il suffit d'ajouter i fois la taille du type considéré à l'adresse du premier élément du tableau pour obtenir l'adresse où se situe l'élément d'indice i, et y accéder que ce soit en lecture ou en écriture.

C'est pour avoir un accès aux éléments en $O(1)$ que les données du tableaux doivent être de même type et enregistrées en mémoire de façon contigüe.

- On ne peut pas insérer dans un tableau un nouvel élément, ou y supprimer un élément. Sa taille est définitivement fixée lors de sa création. C'est pourquoi on parle de tableau statique.
- La longueur d'un tableau s'obtient en temps constant $O(1)$. Elle est mémorisée lors de la création du tableau.
- La création d'un tableau de N éléments avec initialisation (enregistrement des données) requiert un temps $\Theta(N)$. L'extraction d'un sous-tableau de n éléments requiert un temps $\Theta(n)$: il suffit de créer un nouveau tableau et d'y copier les éléments à extraire.

1.2.2 Implantation des tableaux de `numpy` (`numpy.ndarray`)

La structure de donnée utilisée pour implanter les tableaux `numpy` et celle de tableau statique. Ainsi :

- La taille d'un tableau `numpy` est fixée à sa création.
  Ajout et suppression d'éléments sont des opérations impossibles sur les tableaux `numpy`.
- Les données d'un tableau `numpy` doivent être de même type.
  Lorsque des données de type différents sont passés en paramètre, l'interpréteur tente de les homogénéiser (voir exemple plus bas.)
- On peut imposer à la création une homogénéisation en tout autre type de donnée, par l'option `dtype = type` où `type` est un type (ou une chaine de caractère pour certains types particuliers à `numpy`) (exemple : `dtype = int` pour des entiers signés, `dtype = 'uint8'` pour des entiers non signés sur 8 bits, etc...)

Les tableaux `numpy` sont implantés par des tableaux statiques.

- Ils ne peuvent contenir que des données de même type.
- L'ajout ou la suppression d'un élément est impossible.

• Exemples d'homogénéisation :

```
In[1] : import numpy as np

In[2] : np.zeros(4)  # Tableau de flottants
Out[2] : array([0.0, 0.0, 0.0, 0.0])
In[3] : np.zeros(4, dtype=int)  # Tableau d'entiers
Out[3] : array([0, 0, 0, 0])

In [4]: np.array([254,255,256], dtype='uint8')  # Entiers non-signés sur 8 bits
Out[4]: array([254, 255,   0], dtype=uint8)

In [5]: np.array([254,255,256], dtype='int8')  # Entiers signés sur 8 bits
Out[5]: array([-2, -1,  0], dtype=int8)

In[6]: In [29]: np.array([1,2,'toto'])  # Homogénéisation en chaines
Out[6]: array(['1', '2', 'toto'], dtype='<U4')

In [7]: np.array([True, 3])
Out[7]: array([1, 3])

In [8]: np.array([True, 3.])
Out[8]: array([ 1.,  3.])

In [9]: np.array([True, 3.], dtype=bool)
Out[9]: array([ True,  True], dtype=bool)
```

Les tableaux de `numpy` sont les conteneurs les plus efficients lorsque l'on manipule des tableaux de données numériques de taille et de type fixés à l'avance.

1.2.3 Complexité des opérations sur les tableaux `numpy`

Le tableau suivant résume la complexité temporelle des différentes opérations des tableaux de `numpy` :

Opération	Complexité
Création et initialisation d'un tableau de N éléments	$O(N)$
extraction d'un sous-tableau de n éléments	$O(n)$
Lecture d'un élément	$O(1)$
Modification d'un élément	$O(1)$
Obtention du nombre d'éléments	$O(1)$
Ajout d'un élément	Impossible
Suppression d'un élément	Impossible

La complexité spatiale de la création d'un tableau de N éléments est $\Theta(N)$.

1.3 Tableaux dynamiques

1.3.1 Structure de donnée

• En informatique théorique un **tableau dynamique** est un tableau,
donc une structure séquentielle, indexée, homogène, de données stockées en mémoire de façon contiguë,
qui de plus est redimensionnable.
Un espace contiguë de N fois la taille nécessaire est réservé en mémoire. Les éléments du tableau sont stockés en début de tableau ; leur nombre est la longueur n du tableau. Capacité N et longueur n sont mémorisées.

éléments	elt 1	elt 2	elt 3	⋯	⋯	elt n	-	⋯	⋯	⋯	⋯	-
indices	0	1	2	⋯	⋯	n-1	n	⋯	⋯	⋯	⋯	N-1
	←— éléments du tableau —→						←— espace réservé —→					

Le choix de la capacité N peut être défini à priori, ou dépendre directement de n ($N = n$ ou $N = 2n$ par exemple).

- L'accès à un élément du tableau (lecture ou écriture) se fait à l'aide de son indice en temps $O(1)$;
  pour les mêmes raisons qu'avec un tableau statique les éléments étant stockés en début de tableau.
- La longueur du tableau s'obtient en temps $O(1)$.
- La création et l'initialisation d'un tableau de n éléments s'effectue en $\Theta(n)$.

• Ajout est suppression d'un élément sont possibles :

- La suppression de l'élément d'indice `i` s'exécute en temps $\Theta(n-i)$: il faut décaler tous les éléments des indices `i+1` à `n-1` d'un cran sur la gauche, et décrémenter `n`.
  La suppression du dernier élément se fait en temps $O(1)$.
- Tant que la capacité n'est pas dépassée l'insertion d'un élément à l'indice `i` se fait en temps $\Theta(n-i)$: il faut décaler d'abord tous les éléments des indices `i+1` à `n-1` d'un cran sur la droite, et incrémenter `n`.
  Tant que la capacité n'est pas dépassée l'insertion d'un élément en fin de tableau (indice `n`) se fait en temps $O(1)$.
- Lorsque la capacité est dépassée il faut avant insertion copier dans un nouveau tableau, avec un coût $O(n)$ (si suffisamment d'espace non réservé est disponible à la suite du tableau il suffit d'augmenter la capacité). L'insertion est donc dans le pire des cas en $O(n)$.

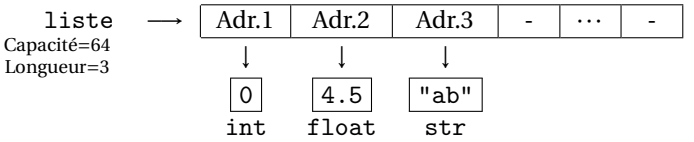
1.3.2 Implantation des listes en `python`

En `python` une liste est implantée comme un tableau dynamique.

Pourtant dans une liste `python` les données ne semblent pas homogènes (pas forcément du même type). Pour contourner cette limitation, ce ne sont pas les données qui sont stockées dans le tableau, mais des références vers ces données, c'est à dire les adresses mémoires des emplacements où sont stockées les données, de façon à obtenir des tableaux homogènes.

Lorsque le nombre d'éléments approche la capacité, la taille du tableau est augmentée, si nécessaire par copie sur un nouvel emplacement.

Exemple : l'implantation de `liste=[0, 4.5, "ab"]` peut se schématiser par :

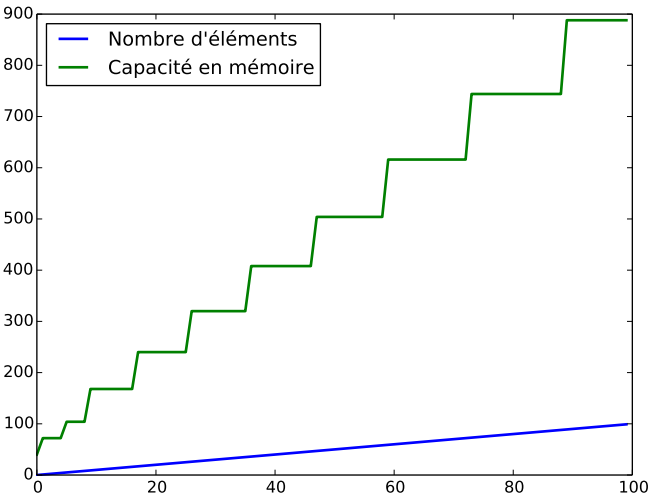



Les listes de `python` sont implantées à l'aide de tableaux dynamiques. Cela permet un accès aux éléments en lecture/écriture en $O(1)$, et autorise de plus l'ajout et la suppression d'éléments ; ces dernières opérations ne se font pas généralement en temps constant $O(1)$.

1.3.3 Mise en évidence de l'augmentation de capacité

La capacité d'une liste s'obtient par la méthode `__sizeof__`. On augmente le nombre d'éléments d'une liste `L`, on mémorise dans 2 listes `X` et `Y` ses nombre d'éléments et capacité :

```
L = []
X,Y=[],[]
for k in range(100):
    X.append(len(L))     # Nbre d'éléments
    Y.append(L.__sizeof__())     # Capacité
    L.append(1)
import matplotlib.pyplot as plt
plt.plot(X,X,X,Y,linewidth=2)
plt.legend(("Nombre d'éléments","Capacité en mémoire"),'upper left')
```



1.3.4 Complexité des opérations sur les listes en `python`

Ainsi de l'implantation des listes en `python` par des tableaux dynamiques, on déduit la complexité des opérations sur une liste de longueur n :

Opération	Complexité
Création d'une liste de n éléments	$\Theta(n)$
Extraction d'une sous-liste de m éléments	$\Theta(m)$
Lecture d'un élément	$O(1)$
Modification d'un élément	$O(1)$
Obtention du nombre d'éléments n	$O(1)$
Suppression de l'élément d'indice i	$\Theta(n-i)$
Suppression du dernier élément (`L.pop()`)	$O(1)$
Ajout d'un élément à l'indice i	$O(n-i)$ si $n<N$ $O(n)$ si $n=N$
Ajout d'un élément en fin de liste (`L.append(e)`)	$O(1)$ si $n<N$ $O(n)$ si $n=N$ $O(1)$ en temps amorti

L'ajout d'un élément en fin de liste (avec la méthode `append`) est dans le pire des cas en $O(n)$. On dit que c'est en O(1) "'*en temps amorti*" car le coût d'une éventuelle duplication sera "amortie" par les ajouts suivants. Plus formellement le coût moyen de l'ajout de n éléments en fin de liste est $O(1)$. Ajouter n éléments (ou plus généralement $a.n+b$ éléments, avec a, b des constantes) est aussi dans le pire des cas de complexité $O(n)$. D'où cette terminologie de **"Complexité en temps amorti"**.


La suppression d'un élément en fin de liste (avec `L.pop()`) s'effectue en temps $O(1)$.

L'ajout d'un élément en fin de liste (avec `L.append(e)`) s'effectue en $O(1)$ "en temps amorti" (et $O(n)$ dans le pire des cas).

Dans le calcul de complexité d'un algorithme utilisant des listes, il est plus simple de réserver au départ une liste de capacité suffisante et d'éviter l'usage des méthodes `append`, `extend` ou autre `+=`.

Les "listes" forment une structure de donnée hybride propre à `python`. Très souple d'utilisation, à utiliser lorsque l'on manipule des tableaux dynamiques ou de données non homogènes.

Ce n'est pas ce que l'on entend communément par "liste" en Informatique; la terminologie est impropre. On devrait parler de tableaux dynamique de références.

2 Algorithmes de tri à balayage

2.1 Introduction

Une opération d'usage très courant en Informatique est le tri d'une collection d'éléments ordonnables (rappelons qu'étymologiquement le terme français "ordinateur" signifie "qui crée de l'ordre"). Un tel algorithme s'applique à la collection pour renvoyer une séquence des éléments ordonnés dans le sens croissant ou décroissant.

Les éléments peuvent être des nombres (munis de l'ordre des réels), des chaines de caractère (munis de l'ordre lexicographique), et plus généralement toute famille d'éléments muni d'une relation de comparaison, comme par exemple des couples chaine de caractères/nombres que l'on ordonnerait selon les valeurs des nombres, ou des nombres complexes que l'on ordonnerait selon les valeurs de leur module.

Les tableaux se prêtent très bien aux algorithmes de tri, puisqu'ils permettent l'accès à leur élément et leur interversion avec des complexités en temps et en espace bornées $O(1)$.

Nous étudions dans cette partie les 3 algorithmes les plus célèbres de tri par balayage sur un tableau. Ce sont aussi les plus simples ; il s'agit de :

- tri par sélection,
- tri à bulle,
- tri par insertion.

On parle de tri à balayage car le tri s'opère sur un tableau en le balayant (c'est à dire en le parcourant) plusieurs fois pour réordonner les éléments qui ne seraient pas dans le bon ordre).
Le tri par insertion, est un exigible du programme de 2ème année : il faut savoir le programmer, l'adapter, prouver sa correction, connaitre et savoir calculer sa complexité. Les deux autres tris apparaissent aussi dans les concours, et pour eux aussi mieux vaut en maîtriser tous les aspects.

Toutes les versions que nous donnons des algorithmes de tri effectuent un **tri dans l'ordre croissant**. Il est facile de les adapter pour que le tri se fasse dans l'ordre inverse. Ils s'opèrent aussi sur des tableaux de nombres, mais s'adaptent rapidement à tout type de donnée munie d'un opérateur de comparaison.

Un algorithme de tri sur un tableau est dit :

en place si sa complexité en espace est bornée, c'est à dire si son application ne nécessite que l'usage d'une quantité bornée d'espace mémoire (quelques variables) indépendant du nombre d'éléments à trier.

stable si tout couple d'éléments égaux (pour la relation de comparaison) restent disposés dans le même ordre avant et après le tri. Sinon on dit qu'il est instable.


Un tri est en place si il s'opère directement sur le tableau en ne nécessitant que peu d'espace supplémentaire.
Un tri est stable si des éléments égaux restent disposés dans le même ordre avant et après le tri.

Le caractère en place d'un tri signifie que le tri est économe en espace mémoire. Le nombre d'éléments à trier pouvant être deja très important, il est intéressant de ne pas avoir à augmenter significativement l'encombrement dans la mémoire.
Le caractère stable est particulièrement intéressant lorsqu'on trie une collection d'éléments selon plusieurs caractéristiques. Imaginons par exemple que l'on dispose d'une famille de couples (nom,date de naissance) que l'on veuille trier par nom d'abord, et pour des homonymes par date de naissance ensuite. Avec un tri stable il suffit de trier d'abord les couples selon leur date de naissance, puis de refaire un tri sur le tableau obtenu, mais selon leur nom.

2.2 Le tri par sélection

C'est l'algorithme de tri dont le principe est le plus simple : sélectionner un à un les éléments du tableau par ordre décroissant pour les déplacer à leur position définitive. Son nom vient du fait qu'à chaque étape on sélectionne un nouvel élément pour le déplacer.

2.2.1 Principe

La sélection des éléments par ordre décroissant se fait à l'aide d'une recherche de maximum.

Donné un tableau `T` de N nombres :

- Chercher dans le tableau l'élément maximal.
- L'échanger avec celui placé en fin de tableau.
- Recommencer sur le sous-tableau constitué des $N-1$ premiers éléments, et ainsi de suite.

Il est bien sûr tout à fait possible de chercher plutôt le minimum pour le déplacer en début de tableau.

2.2.2 Exemple

Code de couleurs :

- ⓧ : maximum sélectionné dans le sous-tableau,
- x : élément à sa place définitive.

3	**1**	**2**	**5**	**6**	**4**	Tableau à trier
3	1	2	5	⑥	4	maximum sélectionné
3	1	2	5	4	6	déplacé en fin de liste
3	1	2	⑤	4	6	maximum sélectionné
3	1	2	4	5	6	déplacé en fin de sous-liste
3	1	2	④	5	6	maximum sélectionné
3	1	2	4	5	6	déplacé en fin de sous-liste
③	1	2	4	5	6	maximum sélectionné
2	1	3	4	5	6	déplacé en fin de sous-liste
②	1	3	4	5	6	maximum sélectionné
1	2	3	4	5	6	déplacé en fin de sous-liste
1	**2**	**3**	**4**	**5**	**6**	Tableau trié

2.2.3 Algorithme de tri par sélection

• L'algorithme du tri par sélection procède ainsi :

Pour un tableau `T` de longueur `N`, au sein d'une boucle for varie une variable `i` de `N-1` (indice du dernier élément) à `1` (indice du deuxième élément). A chaque itération de la boucle est recherché l'indice `m` de l'élément maximal parmi tous ceux d'indices $\leqslant$`i` avant d'échanger dans le tableau les éléments aux indices `i` et `m`.

```
Algorithme de Tri par Sélection (Paramètre : un tableau T)
    N = longueur(T)
    POUR i variant de N-1 à 1 par pas de -1:
        m = indice du Maximum dans le sous-tableau de T allant jusqu'à l'indice i
        Echanger T[i] et T[m]
    FIN POUR
```

• Ou encore par une recherche de minimum :

Pour un tableau `T` de longueur `N`, au sein d'une boucle for varie une variable `i` de `0` (indice du premier élément) à `N-2` (indice de l'avant-dernier élément). A chaque itération de la boucle est recherché l'indice `m` de l'élément minimal parmi tous ceux d'indices $\geqslant$`i` avant d'échanger dans le tableau les éléments aux indices `i` et `m`.

```
Algorithme de Tri par Sélection (Paramètre : un tableau T)
    N = longueur(T)
    POUR i variant de 0 à N-2:
        m = indice du Minimum dans le sous-tableau de T débutant à l'indice i
        Echanger T[i] et T[m]
    FIN POUR
```

2.2.4 Correction de l'algorithme

Supposons sans perte de généralité que l'algorithme est dans sa version recherche de maximum ; l'argument s'adapte aisément à l'autre version.
L'algorithme n'étant constitué que d'une boucle for sa terminaison est assurée.

Démontrons qu'on a l'**invariant de boucle** :

"Après le k-ième passage dans la boucle les k derniers éléments du tableau sont les k plus grands et sont ordonnés dans le sens croissant."


Un invariant de boucle est une propriété initialement vraie (ou une expression) qui demeure inchangée à chaque passage dans la boucle. Voir "Informatique MPSI, PCSI, PTSI" chez le même éditeur, chapitre 6.

Preuve. Par récurrence sur k.

Initialisation. Au premier passage, lorsque `i=N-1`, le maximum est déplacé à l'indice `N-1`, en dernière position. La proposition de récurrence est vérifiée.

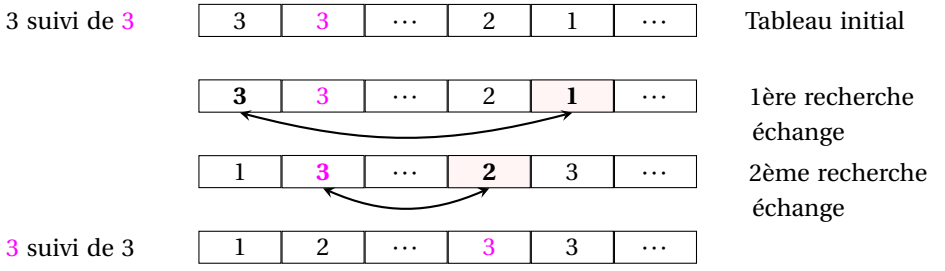
Hérédité. Supposons que les k derniers éléments soient les k plus grands et soient ordonnés dans le sens croissant. Au $(k+1)$-ème passage dans la boucle, lorsque $i = N-k-1$, `T[m]` est le plus grand élément parmi les $N-k$ plus petit éléments ; `T[m]` est donc à la fois plus grand que les $N-k$ premiers éléments et plus petit que les k derniers. On le déplace à l'indice `N-k-1` avant les k derniers éléments ; les $(k+1)$ derniers éléments du tableau sont alors d'une part les $(k+1)$ plus grands du tableau et d'autre part ordonnés dans le sens croissant. La proposition de récurrence demeure donc vraie. □

Déduisons-en la correction de l'algorithme. A la terminaison du programme sur un tableau de longueur N, la boucle s'est exécutée $N-1$ fois, et d'après l'invariant de boucle les $N-1$ derniers éléments du tableau sont les $N-1$ plus grands et sont ordonnés dans le sens croissant. Aussi le premier élément du tableau est le plus petit, et tous les éléments du tableau sont ordonnés dans le sens croissant.

Le tri sélection s'exécute en place. Par défaut il n'est pas stable, à cause de l'échange effectuée. Il est possible de l'implanter de façon stable, mais sur un tableau c'est pénalisant, et il vaut mieux pour cela utiliser d'autres structure de donnée (liste chainée).


Le tri par sélection est en place et instable.

Exemple : Mise en évidence de l'instabilité sur un tableau d'entiers (version recherche de minimum) :



L'échange des positions a inversé les deux éléments identiques 3 et 3 qui se retrouvent positionnés dans l'ordre inverse.

2.2.5 Code Python

• On peut effectuer la recherche de l'indice du maximum au sein d'une fonction extérieure.

```
def indiceMax(T,i):# Cherche l'indice du max parmi indices <= i
    iMax = 0
    for k in range(1,i+1):
        if T[iMax] < T[k]:
            iMax = k
```

```
    return iMax

def tri_selection(T):
    N = len(T)
    for i in range(N-1,0,-1):
        iMax = indiceMax(T,i)
        T[i], T[iMax] = T[iMax], T[i]
    return T
```


Le renvoi de T est facultatif : le tableau T aura été trié quoiqu'il en soit.

• Ou insérer la recherche du maximum dans la fonction principale.

```
def tri_selection(T):
    N = len(T)
    for i in range(N-1,0,-1):
        iMax = 0
        for k in range(1,i+1):
            if T[iMax] < T[k]:
                iMax = k
        T[i], T[iMax] = T[iMax], T[i]
    return T
```

• Ou procéder par une recherche de minimum à déplacer en début de tableau.

```
def tri_selection(T):
    N = len(T)
    for i in range(N-1):
        iMin = i
        for k in range(i+1,N):
            if T[iMin] > T[k]:
                iMin = k
        T[i], T[iMin] = T[iMin], T[i]
    return T
```

• Avec affichage des étapes :

```
def tri_selection(T):
    N = len(T)
    for i in range(N-1,0,-1):
        iMax = 0
        for k in range(1,i+1):
            if T[iMax] < T[k]:
                iMax = k
        print(T," Sélection dans [0;",i,"] :", sep='', end=' ')
        print("indice",iMax,"element",T[iMax])
        T[i], T[iMax] = T[iMax], T[i]
    print(T)
    return T
```

Exemple d'utilisation : avec la dernière fonction.

```
In [2]: tri_selection([6,3,2,1,5,4])
[6, 3, 2, 1, 5, 4] Sélection dans [0;5] : indice 0 element 6
[4, 3, 2, 1, 5, 6] Sélection dans [0;4] : indice 4 element 5
[4, 3, 2, 1, 5, 6] Sélection dans [0;3] : indice 0 element 4
[1, 3, 2, 4, 5, 6] Sélection dans [0;2] : indice 1 element 3
[1, 2, 3, 4, 5, 6] Sélection dans [0;1] : indice 1 element 2
```

```
[1, 2, 3, 4, 5, 6]
Out[2]: [1, 2, 3, 4, 5, 6]
```

2.2.6 Complexité

On effectue le calcul, par exemple sur l'algorithme par recherche de minimum. L'approche utilisée n'influe pas sur la complexité.

Dans tous les cas (pire ou meilleur) on effectue de l'ordre de :

$$\Theta(1)+\sum_{i=0}^{N-2}\left(\Theta(1)+\sum_{k=i}^{N-1}\Theta(1)\right)=\Theta(1)+\Theta(N-1)+\Theta\left(\sum_{i=0}^{N-2}\sum_{k=i}^{N-1}1\right)=\Theta(N)+\Theta\left(\sum_{i=0}^{N-2}N-1-i\right)$$
$$=\Theta(N)+\Theta\left((N-1)^2-\frac{(N-1)(N-2)}{2}\right)=\Theta(N)+\Theta\left(\frac{(N-1)N}{2}\right)=\Theta(N)+\Theta(N^2)=\Theta(N^2)$$

L'algorithme de tri par sélection sur un tableau de N éléments a une complexité quadratique $\Theta(N^2)$ dans le pire des cas, dans le meilleur des cas et en moyenne.


Le tri par sélection a une complexité quadratique dans tous les cas.

Le tri par sélection est considéré comme un mauvais tri. Sa complexité quadratique dans tous les cas, et son caractère instable, ne lui laissent comme seul point fort que sa simplicité. Il n'est utilisé que pour trier de petits tableaux de nombres, lorsque ses performances médiocres ne sont pas pénalisantes.

2.3 Le tri à bulle

C'est un algorithme de tri au principe très simple. Son nom provient d'une analogie avec les bulles de gaz dans le champagne qui remontent progressivement à la surface en échangeant leur position avec les bulles de liquide à densité plus importante.

2.3.1 Principe

Soit `T` un tableau de nombres.

- Parcourir les éléments du tableau `T` du premier au dernier.
  - Dès que l'on rencontre deux éléments consécutifs qui ne sont pas dans le bon ordre, on les intervertit. C'est à dire :

```
SI T[i] > T[i+1]
ALORS : intervertir T[i] et T[i+1]
```

- Recommencer un parcours tant que l'on a changé quelque chose au parcours précédent.


Dans le tri à bulle on parcourt le tableau tant que c'est nécessaire en intervertissant les éléments consécutifs qui ne sont pas dans le bon ordre.

2.3.2 Exemple

3	**2**	**1**	**5**	**4**	Tableau à trier
3	**2**	1	5	4	Parcours 1
2	**3**	1	5	4	interversion
2	**3**	**1**	5	4	
2	**1**	**3**	5	4	interversion
2	1	**3**	**5**	4	
2	1	3	**5**	**4**	
2	1	3	**4**	**5**	interversion
2	**1**	3	4	5	Parcours 2
1	**2**	3	4	5	interversion
1	**2**	**3**	4	5	
1	2	**3**	**4**	5	
1	2	3	**4**	**5**	
1	**2**	3	4	5	Parcours 3
1	**2**	**3**	4	5	
1	2	**3**	**4**	5	
1	2	3	**4**	**5**	aucun changement
1	**2**	**3**	**4**	**5**	Tableau trié

2.3.3 Algorithme. Première version

Le principe énoncé ci-dessus conduit à l'algorithme général pour l'exécution du tri à bulle. Chaque parcours consiste en une boucle for. Cette boucle est imbriquée dans une boucle while, les parcours s'exécutant au moins une fois, puis tant que le parcours précédent a modifié le tableau. Une variable booléenne `Changement`, condition de la boucle while, valant initialement `True`, passe à `False` au début de chaque parcours, et à `True` dès qu'un changement est opéré.

```
Algorithme du Tri à bulle (Paramètre : un tableau T)
    N = longueur(T)
    Changement = VRAI
    TANT QUE Changement est VRAI :
        Changement = FAUX
        POUR i variant de 0 à N-2
            SI T[i] > T[i+1]
            ALORS intervertir T[i] et T[i+1]
                  Changement = VRAI
        FIN POUR
    FIN TANT QUE
```

2.3.4 Correction de l'algorithme

Démontrons qu'on a l'**invariant de boucle** (pour la boucle while) :

"Après k parcours les k derniers éléments du tableau sont les k plus grands et sont ordonnés dans le sens croissant."

Preuve. Par récurrence sur k.

Initialisation. $k = 1$. Soit `T[m]` l'élément maximal dans le tableau ayant le plus grand indice parmi tous les éléments maximaux. Si `m=N-1` il n'y a rien à démontrer aussi supposons que `m<N-1`. Après toutes les comparaisons jusqu'à celle de `T[m-1]` et `T[m]` (si `m>0`), `T[m]` et tous les éléments à sa suite restent à la même position. Aux comparaisons suivantes il sera successivement déplacé aux indices `m+1`, `m+2`, ..., jusqu'à l'indice `N-1` ; en effet, `T[m]` étant le dernier élément maximal, toutes les comparaisons `T[m] > T[m+1]`, `T[m+1] > T[m+2]`, ..., `T[N-2] > T[N-1]` seront successivement vérifiées. Aussi après le 1er parcours le dernier élément du tableau est un élément maximal.

Hérédité. Supposons qu'après le k-ème parcours les k derniers éléments du tableaux soient les plus grands et ordonnés dans le sens croissant ; l'indice `N-k` est alors le plus petit indice de ces k plus grands éléments. Supposons aussi qu'un $(k+1)$-ème parcours ait lieu.
Soit `T[m]` l'élément maximal ayant plus grand indice parmi les $N-k$ premiers éléments du tableau. Après le $(k+1)$-ème parcours l'élément `T[m]` sera successivement déplacé aux indices `m+1`, `m+2`, $\cdots$, pour finir en `N-k-1` ; en effet toutes les

comparaisons `T[m] > T[m+1]`, `T[m+1] > T[m+2]`, ..., `T[N-k-2] > T[N-k-1]` seront successivement vérifiées, tandis que par hypothèse de récurrence `T[N-k-1] > T[N-k]` échouera ainsi que toutes les comparaisons suivantes. Ainsi à l'issue du $(k+1)$-ème parcours les $(k+1)$ derniers éléments `T[N-k-1]`$\leqslant$`T[N-k]` $\leqslant \ldots \leqslant$ `T[N-1]` sont les $(k+1)$ plus grands éléments du tableau et sont ordonnés dans le sens croissant. □

Ceci prouve l'invariant de boucle. Maintenant pour en déduire la terminaison et la correction du programme, remarquons que tant qu'un parcours s'applique à un tableau non ordonné un parcours supplémentaire aura lieu. Ainsi soit le tableau est ordonné avant le `N`-ième parcours, soit il y aura au moins `N` parcours. Mais à l'issue de ce dernier, d'après l'invariant de boucle, les `N` derniers éléments du tableau, c'est à dire tous, seront ordonnés dans le sens croissant. Aussi au plus tard après le parcours suivant, l'algorithme se terminera et à l'arrêt le tableau sera ordonné. Ceci prouve terminaison et correction de l'algorithme.

De plus remarquons que le tri à bulle se fait en place, et qu'il est stable (pour cela l'inégalité dans la comparaison doit être stricte).

Le tri à bulle est en place et stable.

2.3.5 Algorithme. Deuxième version améliorée

On peut améliorer l'algorithme donné ci-dessus : d'après l'invariant de boucle, après le k-ème parcours du tableau, tous les k derniers éléments sont à leur place définitive, et donc aucune inversion ne surviendra plus à ces positions. Donc à chaque parcours de tableau, le parcours pourra s'arrêter un indice avant le précédent. L'algorithme devient :

```
Algorithme du Tri à bulle (Paramètre : un tableau T)
    N = longueur(T)
    Changement = VRAI
    TANT QUE Changement est VRAI :
        Changement = FAUX
        POUR i variant de 0 à N-2
            SI T[i] > T[i+1]
            ALORS intervertir T[i] et T[i+1]
                  Changement = VRAI
        FIN POUR
        N = N-1
    FIN TANT QUE
```

Exemple :

3	**2**	**1**	**5**	**4**	Tableau à trier
3	2	1	5	4	Parcours 1
2	3	1	5	4	interversion
2	3	1	5	4	
2	1	3	5	4	interversion
2	1	3	5	4	
2	1	3	5	4	
2	1	3	4	5	interversion
2	1	3	4	**5**	Parcours 2
1	2	3	4	**5**	interversion
1	2	3	4	**5**	
1	2	3	4	**5**	
1	2	3	**4**	**5**	Parcours 3
1	2	3	**4**	**5**	aucun changement
1	**2**	**3**	**4**	**5**	Tableau trié

2.3.6 Code Python

• Fonction effectuant un tri à bulle :

```
def tri_bulle(T):
    N = len(T)
```

```
    changement = True
    while changement:
        changement = False
        for i in range(N-1):
            if T[i] > T[i+1]:
                T[i], T[i+1] = T[i+1], T[i]
                changement = True
        N -= 1
    return T
```


Le renvoi de T est facultatif : le tableau T aura été trié quoiqu'il en soit.

• Tri à bulle avec affichage des changements opérés.

Insérons dans le code précédent des instructions d'affichage.

```
def tri_bulle(T):
    N = len(T)
    changement = True
    k = 1
    while changement:
        changement = False
        print('-'*3*len(T) +' Parcours',k)      #Affichage parcours
        print(T, end = ' ')
        for i in range(N-1):
            if T[i] > T[i+1]:
                T[i], T[i+1] = T[i+1], T[i]
                print("interversion indices",i,'et',i+1 )  #Affichage interversion
                print(T, end = ' ')
                changement = True
        N -= 1
        k += 1
        if not changement:      #Affichage fin parcours
            print("aucun changement")
        else:
            print()
    return T
```

Exemple d'utilisation :

```
In [2]: tri_bulle([3,2,1,5,4])
--------------- Parcours 1
[3, 2, 1, 5, 4] interversion indices 0 et 1
[2, 3, 1, 5, 4] interversion indices 1 et 2
[2, 1, 3, 5, 4] interversion indices 3 et 4
[2, 1, 3, 4, 5]
--------------- Parcours 2
[2, 1, 3, 4, 5] interversion indices 0 et 1
[1, 2, 3, 4, 5]
--------------- Parcours 3
[1, 2, 3, 4, 5] aucun changement
Out[2]: [1, 2, 3, 4, 5]
```

2.3.7 Complexité

Soit N la longueur du tableau passé en paramètre. Puisque comme vu plus haut, après chaque passage un nouvel élément est à sa place définitive, la boucle `while` s'effectuera au plus N fois. A chaque exécution de la boucle, il y aura $\Theta(1)$

opérations élémentaires (1 affectation et 1 decrémentation) plus les opérations dans la boucle `for`. Au k-ème passage la boucle for s'itèrera $N-k$ fois, effectuant à chaque itération $\Theta(1)$ opérations élémentaires (1 comparaison et au plus 1 échange et une affectation). Le nombre total d'opérations élémentaires est donc dominé par :

$$\sum_{k=1}^{N}\left(\Theta(1)+\sum_{i=1}^{N-k}\Theta(1)\right)=\Theta\left(\sum_{k=1}^{N}1+\sum_{k=1}^{N}\sum_{i=1}^{N-k}1\right)=\Theta\left(N+\sum_{k=1}^{N}N-k\right)=\Theta\left(N+\frac{N(N-1)}{2}\right)=\Theta(N^2)$$

• Complexité dans le pire des cas

Dans le pire des cas, celui d'une liste triée dans le sens décroissant, la boucle while s'exécutera exactement `N` fois. Aussi le calcul ci-dessus donne l'ordre du nombre d'opérations élémentaires dans le pire des cas.

La complexité dans le pire des cas du tri à bulle est quadratique $\Theta(N^2)$.

Exemple d'implantation dans le pire des cas le pire des cas.

```
In [3]: tri_bulle([5,4,3,2,1])
--------------- Parcours 1
[5, 4, 3, 2, 1] interversionion indices 0 et 1
[4, 5, 3, 2, 1] interversion indices 1 et 2
[4, 3, 5, 2, 1] interversion indices 2 et 3
[4, 3, 2, 5, 1] interversion indices 3 et 4
[4, 3, 2, 1, 5]
--------------- Parcours 2
[4, 3, 2, 1, 5] interversion indices 0 et 1
[3, 4, 2, 1, 5] interversion indices 1 et 2
[3, 2, 4, 1, 5] interversion indices 2 et 3
[3, 2, 1, 4, 5]
--------------- Parcours 3
[3, 2, 1, 4, 5] interversion indices 0 et 1
[2, 3, 1, 4, 5] interversion indices 1 et 2
[2, 1, 3, 4, 5]
--------------- Parcours 4
[2, 1, 3, 4, 5] interversion indices 0 et 1
[1, 2, 3, 4, 5]
--------------- Parcours 5
[1, 2, 3, 4, 5] aucun changement
Out[3]: [1, 2, 3, 4, 5]
```

La complexité du tri à bulle est :

- quadratique dans le pire des cas,
- linéaire dans le meilleur des cas,
- quadratique en moyenne.

• Complexité dans le meilleur des cas.

Dans le meilleur des cas, celui d'une liste triée dans le sens croissant, la boucle while ne s'exécute qu'une seule fois. Un seul parcours permet de déterminer que la liste est ordonnée ; le nombre d'opérations élémentaires est alors :

$$\Theta(1)+\sum_{i=1}^{N-1}\Theta(1)=\Theta(N)$$

La complexité dans le meilleur des cas du tri à bulle est linéaire $\Theta(N)$.

Exemple d'implantation dans le meilleur des cas.

```
In [4]: tri_bulle([1,2,3,4,5])
--------------- Parcours 1
[1, 2, 3, 4, 5] aucun changement
Out[4]: [1, 2, 3, 4, 5]
```

• **Complexité en moyenne.**

On peut démontrer que la complexité en moyenne est encore quadratique.

La complexité en moyenne du tri à bulle est quadratique $\Theta(N^2)$.

Le tri à bulle, malgré sa complexité linéaire dans le meilleur des cas, et son caractère stable, et considéré comme un mauvais algorithme de tri. Son inefficacité provient notamment du phénomène des "tortues" : un élément proche d'être minimal et situé en fin de tableau nécessitera beaucoup de balayages pour revenir à sa place définitive. Ici encore, son avantage réside en sa simplicité.

Sa variante du tri cocktail résout en partie le problème des "tortues". Sa variante du tri à peigne (Comb sort) le résout totalement et peut rivaliser avec les meilleurs algorithmes de tri (voir Exercice 5).

2.4 Le tri par insertion

Le tri par insertion est un bon algorithme de tri. Pour un petit nombre d'éléments à trier, c'est celui en pratique qui est le plus efficace. Aussi c'est l'algorithme le plus couramment utilisé dans la vie courante, par exemple pour trier un paquet de carte.

2.4.1 Principe du tri par insertion

Le tri par insertion est souvent utilisé pour trier un paquet de carte :

On constitue (imaginairement) 2 tas :

- l'un dans la main droite, contenant toutes les cartes avant tri,
- l'autre dans la main gauche, contenant les cartes déjà triées,

Initialement la main gauche est vide.

Chaque étape consiste à :

- prendre la première carte du tas non trié
- L'insérer progressivement à sa bonne place dans le tas trié, en la faisant descendre d'une position dans le paquet tant que sa valeur reste inférieure à celle de la carte située en dessous-d'elle.

Après chaque étape le tas non trié contient une carte de moins, et le tas trié une carte de plus.

A la fin du tri la main droite est vide. La main gauche contient toutes les cartes, triées.

Dans son implantation sur un tableau, le tri s'effectuera en place. Les tas triés et non triés seront implantés par les début du tableau et fin du tableau. Initialement le tableau est non trié, le tas trié est vide, au cours du déroulement de l'algorithme les éléments seront insérés dans la partie à gauche du tableau, venant "grossir le tas trié". C'est de cette insertion que vient le nom du tri.

2.4.2 Exemple

• Code de couleurs :

- x : Partie du tableau non triée,
- x : Partie du tableau triée,
- ⓧ : Elément à insérer dans la partie triée.

3	2	1	5	6	4	Tableau à trier
③	2	1	5	6	4	première insertion
3	②	1	5	6	4	deuxième insertion
②	3	1	5	6	4	
2	3	①	5	6	4	troisième insertion
2	①	3	5	6	4	
①	2	3	5	6	4	
1	2	3	⑤	6	4	quatrième insertion
1	2	3	5	⑥	4	cinquième insertion
1	2	3	5	6	④	sixième insertion
1	2	3	5	④	6	
1	2	3	④	5	6	
1	**2**	**3**	**4**	**5**	**6**	Tableau trié

2.4.3 Algorithme

Le tri s'opère en place (directement sur le tableau passé en paramètre). Pour un tableau `T` de longueur `N` : au sein d'une boucle for varie une variable `i` de `1` (deuxième indice) à `N-1` (dernier indice). À une variable `k` est affectée la valeur de l'indice `i`, de façon à pouvoir décrémenter sa valeur durant l'insertion. L'insertion s'exécute au sein d'une boucle while : l'élément `T[k]` est comparé à l'élément `T[k-1]` à sa gauche pour éventuellement les permuter et poursuivre ensuite avec `T[k-1]`, et ainsi de suite tant que nécessaire.

• Approche simplifiée.

```
Alforithme de Tri par insertion (Paramètre : un tableau T)
    N = longueur(T)
    POUR i variant de 1 à N-1:
        x = T[i]
        k = i
        TANT QUE k > 0 et T[k-1] > x:
            échanger T[k] et T[k-1]
            k = k-1
        FIN TANT QUE
    FIN POUR
```

Cet algorithme n'est pas optimal car durant une insertion on copie la valeur `x` en plusieurs positions alors qu'à la fin de l'insertion de `x` il ne demeurera qu'en une seule position. L'algorithmique simplifié est cependant intéressant pour afficher l'état du tableau au fil de l'algorithme.

• Approche optimisée.

```
Alforithme de Tri par insertion (Paramètre : un tableau T)
    N = longueur(T)
    POUR i variant de 1 à N-1:
        x = T[i]
        k = i
        TANT QUE k > 0 et T[k-1] > x:
            T[k] = T[k-1]
            k = k-1
        T[k] = x
        FIN TANT QUE
    FIN POUR
```

Dans cette version `x` n'est copiée à sa position finale qu'en toute fin d'insertion. Les performances de l'algorithme sont sensiblement améliorées par rapport à la version précédente. C'est tant que possible sous cette forme que doit s'appliquer un tri par insertion.

Pour une implantation optimale du tri par insertion, durant son insertion, un élément `x`, ne doit être copié dans le tableau qu'en toute fin d'insertion.

2.4.4 Correction de l'algorithme

Terminaison de l'algorithme. Le programme principal est constitué d'une boucle for, au sein de laquelle est imbriquée une boucle while. Il suffit donc de prouver de la terminaison de la boucle while. Pour cela il suffit de remarquer que `k` en est un variant de boucle. D'où la terminaison de l'algorithme.

Un **variant de boucle** est une expression ne prenant que des valeurs entières et positives, et qui décroit strictement à chaque passage dans la boucle. Exhiber un variant de boucle démontre la terminaison de la boucle. (Voir "Informatique MPSI, PCSI, PTSI", chapitre 6 chez le même éditeur.)

Correction de l'algorithme. Démontrons qu'on a l'invariant de boucle :

"A l'issue de la k-ième insertion les $(k+1)$ premiers éléments du tableau sont ordonnés dans le sens croissant."

Preuve. Par récurrence sur k. Soit N le nombre d'éléments du tableau.

Initialisation. Pour $k=1$. L'élément à insérer est celui d'indice `k=1`. Il y a deux possibilités :

1er cas. `T[0]`$\leqslant$`T[1]` ; dans ce cas il n'y a pas d'insertion.
2ème cas. `T[0]>T[1]` ; il y a une seule insertion qui échange `T[0]` et `T[1]`, due dans la version simplifiée de l'algorithme à l'instruction :

```
echanger T[1] et T[0]
```

durant le (seul) passage dans la boucle while. Dans la version optimisée de l'algorithme l'échange est dû aux instructions :

```
x = T[1]          # avant la boucle while
T[1] = T[0]     # pendant la boucle while
T[0] = x         # après la boucle while
```

Dans les deux cas, après l'insertion les deux premiers éléments du tableau sont ordonnés dans le sens croissant, et la proposition de récurrence est vérifiée.

Hérédité. Supposons la proposition vraie au rang $k-1$, et $k<N$. Les k premiers éléments du tableau sont alors ordonnés dans le sens croissant : `T[0]`$\leqslant$`T[1]`$\leqslant\cdots\leqslant$`T[k-1]`. Ils sont suivis dans le tableau par le $(k+1)$-élément : `x = T[k]`. A l'insertion suivante `x` doit être inséré au sein des $(k+1)$ premiers éléments.
• Premier cas : si `T[k-1]` $\leqslant$ `x` alors la condition du while : `(k > 0 et T[k-1] > x)` échoue et l'insertion s'arrête avec aux $(k+1)$ premières positions dans le tableau :

`T[0]` $\leqslant$ `T[1]`$\leqslant\cdots\leqslant$`T[k-1]`$\leqslant$`x`.

c'est à dire :

`T[0]` $\leqslant$ `T[1]`$\leqslant\cdots\leqslant$`T[k-1]`$\leqslant$`T[k]`.

La proposition de récurrence demeure alors vraie au rang k.

• Deuxième cas : si `T[k-1]>x`.

On traite dans la suite de la preuve de l'hérédité, deux cas, selon que l'algorithme est sous sa forme simplifiée ou optimisée.

⋆ Pour la forme simplifiée de l'algorithme :
si `T[k-1]>x` alors l'insertion permute ces deux éléments :

`T[0]`$\leqslant$`T[1]`$\leqslant\cdots\leqslant$`T[k-2]` et `x`<`T[k-1]`.

et l'insertion se poursuit en comparant `T[k-2]` et `x`, et ainsi de suite jusqu'à `j-1`, dernier indice d'un élément $\leqslant$`x`, s'il en est, `j`= 0 sinon. A l'issue de l'insertion, tous les éléments à gauche de `x`, s'il y en a, sont ordonnés et inférieurs à `x`, tous les éléments à droite de `x` parmi les $(k+1)$ premiers éléments du tableau, s'il y en a, sont ordonnés et strictement supérieurs à `x` :

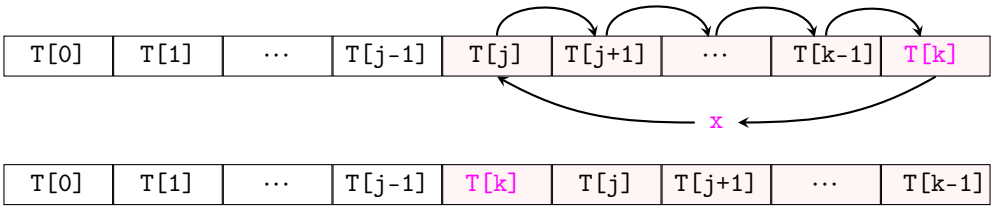
`T[0]`$\leqslant\cdots\leqslant$`T[j-1]`$\leqslant$`x`<`T[j]`$\leqslant\cdots\leqslant$`T[k-1]`.

c'est à dire :

`T[0]`$\leqslant\cdots\leqslant$`T[j-1]`$\leqslant$`T[k]`<`T[j]`$\leqslant\cdots\leqslant$`T[k-1]`.

Ainsi à l'issue de la k-ème insertion, les $(k+1)$ premiers éléments du tableau sont ordonnés dans le sens croissant. La proposition de récurrence demeure vraie au rang k.

⋆ Pour la forme optimisée de l'algorithme :
Soit `j-1` l'indice du dernier élément dans `T[0:k]` inférieur ou égal à x, s'il en est, -1 sinon. À l'issue de la k-ème itération de la boucle `for`, le tableau a alors été modifié comme ci-dessous :



Les déplacements au-dessus du tableau sont exécutées du plus à droite au plus à gauche à chaque passage dans la boucle `while` grace à l'instruction `T[k] = T[k-1]`. Les déplacements figurant au-dessous du tableau sont exécutés pour le premier avant le boucle `while` grâce à l'instruction `x = T[i]`, pour le second après la boucle grâce à `T[k] = x`. Le stockage de `T[i]` dans `x` mémorise la valeur de `T[i]` avant l'instruction `T[i] = T[i-1]`, pour la déplacer en toute fin d'insertion à la position `j` grace à `T[j] = x`. Ainsi à l'issue de l'itération de la boucle `for` les `(k+1)` premiers éléments de `T` sont désormais ordonnés dans le sens croissant :

$$T[0] \leqslant T[1] \leqslant \cdots \leqslant T[j-1] \leqslant T[k] \leqslant T[j] \leqslant T[j+1] \leqslant T[k-1].$$

L'invariant de boucle est vérifié au rang k. Cela conclut la récurrence. □

Reprenons la preuve de la correction de l'algorithme. Pour un tableau de N éléments, il y a $(N-1)$ insertions. A leur issue, d'après l'invariant de boucle, les N éléments du tableau sont ordonnés. Le tableau est donc trié.

De plus l'algorithme est en place, et stable (à cause de l'inégalité stricte dans la condition de la boucle while).

Le tri par insertion est en place et stable.

2.5 Code Python

Toutes les versions données sont en places et stables.

• Version optimisée.

```
def tri_insertion(T):
    N = len(T)
    for i in range(1,N):          # Balayage du 2ème au dernier
        x = T[i]                  # élément à insérer
        k = i                     # k est son indice
        while k > 0 and T[k-1] > x: # Insertion dans la partie triée
            T[k] = T[k-1]         # décalage à gauche
            k = k-1               # et poursuivre l'insertion
        T[k-1] = x                # copie finale de x
    return T    # Si l'on souhaite retourner le résultat
```


Le renvoi de T est facultatif : le tableau T aura été trié quoiqu'il en soit.

• Version avec affichage des insertions.

On utilise l'algorithme sous sa forme simplifiée pour pouvoir afficher les états intermédiaires du tableau durant le tri.

```
def tri_insertion(T):
    N = len(T)
    for i in range(1,N):
        k = i
        print(T,"insertion de",T[i],"d'indice",i)  # Affichage début insertion
        while k > 0 and T[k-1] > T[k]:
            T[k], T[k-1] = T[k-1], T[k]
            print(T)              # Affichage
            k = k-1
        print('-'*3*len(T))       # Affichage fin insertion
    return T
```

Exemple.

```
In [91]: tri_insertion([6,3,2,1,5,4])
[6, 3, 2, 1, 5, 4] insertion de 3 d'indice 1
[3, 6, 2, 1, 5, 4]
------------------
[3, 6, 2, 1, 5, 4] insertion de 2 d'indice 2
[3, 2, 6, 1, 5, 4]
[2, 3, 6, 1, 5, 4]
------------------
[2, 3, 6, 1, 5, 4] insertion de 1 d'indice 3
[2, 3, 1, 6, 5, 4]
[2, 1, 3, 6, 5, 4]
[1, 2, 3, 6, 5, 4]
------------------
[1, 2, 3, 6, 5, 4] insertion de 5 d'indice 4
[1, 2, 3, 5, 6, 4]
------------------
[1, 2, 3, 5, 6, 4] insertion de 4 d'indice 5
[1, 2, 3, 5, 4, 6]
[1, 2, 3, 4, 5, 6]
------------------
Out[91]: [1, 2, 3, 4, 5, 6]
```

2.5.1 Complexité

Pour calculer la complexité du tri par insertion, considérons qu'il s'applique sur un tableau `T` de N éléments. Au passage `i` dans la boucle for (à l'insertion de `T[i]`) la boucle while s'exécutera entre 0 et `i` fois. Le nombre d'opérations élémentaires est donc :

- au moins : $2+6(N-1)=\Theta(N)$
  (à chacun des $(N-1)$ passages dans la boucle for 1 incrémentation, 2 affectations, 2 comparaisons et un ET logique),
- au plus dominé par :

$$\Theta(1)+\sum_{i=1}^{N-1}(\Theta(1)+O(i))=\Theta(N)+O\left(\frac{N(N-1)}{2}\right)=\Theta(N)+O(N^2)=O(N^2)$$

Montrons que ces bornes, linéaire et quadratique, sont atteintes respectivement dans le meilleur des cas et dans le pire des cas.

- Dans le meilleur des cas : le cas d'un tableau ordonné dans le sens croissant, par exemple :

  $\mathtt{T}=$ | 1 | 2 | $\cdots$ | $(N-1)$ | N |

  Lors de la i-ème insertion l'élément `T[i]` est déjà à la bonne place, et la boucle while ne s'exécute pas ; de l'ordre d'1 opérations pour chaque insertion. La complexité est alors d'ordre :

$$\Theta(1)+\sum_{i=1}^{N-1}\Theta(1)=\Theta(1)+\Theta(N-1)=\Theta(N)$$

  c'est à dire linéaire.

– Dans le pire des cas : le cas d'un tableau ordonné dans le sens décroissant, par exemple :

  $\mathtt{T}=$ | N | $(N-1)$ | $\cdots$ | 2 | 1 |

  Lors de la i-ème insertion l'élément `T[i]` est strictement inférieur au `i` éléments qui le précèdent, il doit donc inséré en tout début de tableau : la boucle `while` s'exécute i fois. La complexité est alors d'ordre :

$$\Theta(1)+\sum_{i=1}^{N-1}(\Theta(1)+\Theta(i))=\Theta(N)+\Theta\left(\frac{N(N-1)}{2}\right)=\Theta(N)+\Theta(N^2)=\Theta(N^2)$$

  c'est à dire quadratique.

On peut démontrer aussi que la complexité en moyenne est quadratique.

La complexité du tri par insertion sur un tableau de N éléments est :

- **linéaire $\Theta(N)$ dans le meilleur des cas,**
- **quadratique $\Theta(N^2)$ dans le pire des cas,**
- **quadratique $\Theta(N^2)$ en moyenne.**


La complexité du tri par insertion est quadratique dans le pire des cas et linéaire dans le meilleur des cas.

Exemples :

• Tri par insertion sur une liste ordonnée dans le sens croissant.

```
In [4]: tri_insertion([1,2,3,4,5])
[1, 2, 3, 4, 5] insertion de 2 indice 1
---------------
[1, 2, 3, 4, 5] insertion de 3 indice 2
---------------
[1, 2, 3, 4, 5] insertion de 4 indice 3
---------------
[1, 2, 3, 4, 5] insertion de 5 indice 4
---------------
Out[4]: [1, 2, 3, 4, 5]
```

• Tri par insertion sur une liste ordonnée dans le sens décroissant.

```
In [3]: tri_insertion([5,4,3,2,1])
[5, 4, 3, 2, 1] insertion de 4 indice 1
[4, 5, 3, 2, 1]
---------------
[4, 5, 3, 2, 1] insertion de 3 indice 2
[4, 3, 5, 2, 1]
[3, 4, 5, 2, 1]
---------------
[3, 4, 5, 2, 1] insertion de 2 indice 3
[3, 4, 2, 5, 1]
[3, 2, 4, 5, 1]
[2, 3, 4, 5, 1]
---------------
[2, 3, 4, 5, 1] insertion de 1 indice 4
[2, 3, 4, 1, 5]
[2, 3, 1, 4, 5]
[2, 1, 3, 4, 5]
[1, 2, 3, 4, 5]
---------------
Out[3]: [1, 2, 3, 4, 5]
```

A contrario des tris par sélection et à bulle, le tri par insertion est considéré comme un tri relativement efficace. Ceci car :

- On démontre que dans le cas d'un petit nombre d'éléments à trier (quelques dizaines) c'est le plus rapide en moyenne, malgré que sa complexité en moyenne soit assez mauvaise, quadratique.
- Il est également très efficace lorsque le tableau est déjà presque trié. Par exemple la complexité est linéaire si de plus tous les éléments sont à une distance uniformément bornée (ne dépendant pas de N) de leur position finale. Ou lorsque le nombre d'éléments qui ne sont à la bonne place est uniformément borné.

Conclusion : on peut continuer à l'utiliser pour trier un paquet de carte, c'est le meilleur connu, mais nous verrons que lorsque le nombre d'éléments à trier devient grand, ce n'est plus celui à utiliser sur de grands tableaux très "désordonnés".

Dans l'exercice 6, nous voyons une amélioration du principe du tri par insertion, le tri de Shell (Shell Sort), dont les performances rivalisent avec les meilleurs algorithmes connus ; par contre cette amélioration a un coût : le tri de Shell est instable.

1) ■ Tris dans l'ordre décroissant.

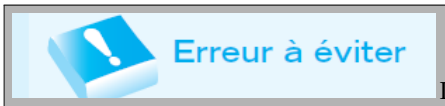
Écrire une version des algorithmes de :

1. Tri par insertion,
2. Tri à bulle,
3. Tri par sélection,

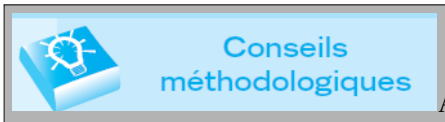
qui prennent en paramètre un tableaux de nombres et le trient dans l'ordre décroissant.

Pour cela :

- On écrira des fonctions `tri_insertionD`, `tri_bulleD`, `tri_selectionD` qui effectuent le tri demandé et affichent les étapes effectuées.
- Les tris opérés devront s'exécuter en place et pour les deux premiers être stables.
- On testera chaque fonction, par exemple sur la liste `[3,2,1,5,4]`.



Il ne faut pas effectuer un tri dans l'ordre croissant pour ensuite copier le tableau en sens inverse. D'abord ce ne serait pas en place. Ensuite même si la complexité temporelle ne changerait pas, ce serait très pénalisant pour l'efficacité de l'algorithme en mémoire et en temps.



Adapter les algorithmes vus en cours. Garder à l'esprit que les deux premiers tris devant rester stables, il faut bien choisir entre des inégalités strictes et larges.

1. Pour le tri par insertion : en vue de l'affichage l'algorithme échange les positions de x et de son voisin au fil de l'étape d'insertion.

```
def tri_insertionD(T):
    N = len(T)
    for i in range(1,N):
        x = T[i]
        k = i
        print(T,"insertion de",T[i],"indice",i)
        while k > 0 and T[k-1] < x:
            T[k] = T[k-1]
            T[k-1] = x
            print(T)
            k = k-1
        print('-'*3*len(T))
    return T
```

Inégalité stricte pour que le tri soit stable.

Exemple :

```
In [2]: tri_insertionD([3,2,1,5,4])
[3, 2, 1, 5, 4] insertion de 2 indice 1
---------------
[3, 2, 1, 5, 4] insertion de 1 indice 2
---------------
[3, 2, 1, 5, 4] insertion de 5 indice 3
```

```
[3, 2, 5, 1, 4]
[3, 5, 2, 1, 4]
[5, 3, 2, 1, 4]
---------------
[5, 3, 2, 1, 4] insertion de 4 indice 4
[5, 3, 2, 4, 1]
[5, 3, 4, 2, 1]
[5, 4, 3, 2, 1]
---------------
Out[2]: [5, 4, 3, 2, 1]
```

2. Pour le tri à bulle :

```
def tri_bulleD(T):
    N = len(T)
    changement = True
    k = 1
    while changement:
        changement = False
        print('-'*3*len(T)+' Parcours',k)
        print(T, end = ' ')
        for i in range(N-1):
            if T[i] < T[i+1]:
                T[i], T[i+1] = T[i+1], T[i]
                print("interversion indices",i,'et',i+1 )  #Affichage interversion
                print(T, end = ' ')
                changement = True
        N -= 1
        k += 1
        if not changement:
            print("aucun changement")
        else:
            print()
    return T
```

Inégalité stricte pour que le tri soit stable.

Exemple :

```
In [3]: tri_bulleD([3,2,1,5,4])
--------------- Parcours 1
[3, 2, 1, 5, 4] interversion indices 2 et 3
[3, 2, 5, 1, 4] interversion indices 3 et 4
[3, 2, 5, 4, 1]
--------------- Parcours 2
[3, 2, 5, 4, 1] interversion indices 1 et 2
[3, 5, 2, 4, 1] interversion indices 2 et 3
[3, 5, 4, 2, 1]
--------------- Parcours 3
[3, 5, 4, 2, 1] interversion indices 0 et 1
[5, 3, 4, 2, 1] interversion indices 1 et 2
[5, 4, 3, 2, 1]
--------------- Parcours 4
[5, 4, 3, 2, 1] aucun changement
Out[3]: [5, 4, 3, 2, 1]
```

3. Pour le tri par sélection :

```
def tri_selectionD(T):
    N = len(T)
    for i in range(N-1):
        iMax = i
        for k in range(i+1,N):
            if T[iMax] < T[k]:
                iMax = k
        print(T," Sélection dans [",i,";",N-1,"] :", sep='', end = ' ')
        print('indice",iMax,"element",T[iMax])
        T[i], T[iMax] = T[iMax], T[i]
    print(T)
    return T
```


Le tri par sélection est instable.

Exemple :

```
In [4]: tri_selectionD([3,2,1,5,4])
[3, 2, 1, 5, 4] Sélection dans [0;4] : indice 3 element 5
[5, 2, 1, 3, 4] Sélection dans [1;4] : indice 4 element 4
[5, 4, 1, 3, 2] Sélection dans [2;4] : indice 3 element 3
[5, 4, 3, 1, 2] Sélection dans [3;4] : indice 4 element 2
[5, 4, 3, 2, 1]
Out[4]: [5, 4, 3, 2, 1]
```

2) ■ Tris d'un tableau numérique avec choix de l'ordre.

Dans cet exercice les tris s'opèrent sur des tableaux de nombres (réels). Écrire les fonctions :

```
tri_insertion(T, croissant = True)
  tri_bulle(T, croissant = True)
tri_selection(T, croissant = True)
```

qui exécutent un tri en place sur un tableau numérique, respectivement par sélection, à bulle et par insertion. Les 3 fonctions prennent en paramètre un booléen `croissant` qui par défaut vaut `True`. Le tri s'exécutera dans le sens croissant ou décroissant selon la valeur de ce paramètre.

1. On n'impose pas aux tris d'être stables, privilégiant la rapidité.
2. On impose aux deux premiers tris d'être stables.

1. On peut utiliser la paramètre booléen pour tester les comparaisons.

2. On peut multiplier les opérandes des comparaisons par -1 pour renverser les inégalités en conservant leur caractère strict.

1. Des fonctions instables, mais quasiment sans perte de rapidité :

```
def tri_insertion(T,croissant = True):
    N = len(T)
    for i in range(1,N):
        x = T[i]
        k = i
        while k > 0 and (T[k-1] > x) == croissant:
            T[k] = T[k-1]
            k = k-1
        T[k] = x
    return T
```

```
def tri_bulle(T,croissant = True):
    N = len(T)
    changement = True
    while changement:
        changement = False
        for i in range(N-1):
            if (T[i] > T[i+1]) == croissant:
                T[i], T[i+1] = T[i+1], T[i]
                changement = True
        N -= 1
    return T

def tri_selection(T,croissant = True):
    N = len(T)
    for i in range(N-1,0,-1):
        iM = 0
        for k in range(i+1):
            if (T[iM] <= T[k]) == croissant:
                iM = k
        T[i], T[iM] = T[iM], T[i]
    return T
```

2. Des fonctions stables pour les deux premiers tris ; les opérandes des comparaison sont multipliés par un entier valant 1 (ordre croissant) ou -1 (ordre décroissant).

```
def tri_insertion(T,croissant = True):
    if croissant:
        s = 1
    else:
        s = -1
    N = len(T)
    for i in range(1,N):
        x = T[i]
        k = i
        while k > 0 and (T[k-1]*s > x*s):
            T[k] = T[k-1]
            k = k-1
        T[k] = x
    return T

def tri_bulle(T,croissant = True):
    if croissant:
        s = 1
    else:
        s = -1
    N = len(T)
    changement = True
    while changement:
        changement = False
        for i in range(N-1):
            if T[i]*s > T[i+1]*s :
                T[i], T[i+1] = T[i+1], T[i]
                changement = True
        N -= 1
    return T
```

3) ■ Tri par insertion selon un caractère.

1. Ecrire une fonction de tri par insertion, qui s'applique à un tableau de nombres complexes, et les trie par module croissant. Le tri devra être en place et stable.
2. On considère un tableau `Points` constitué de triplets de la forme `('A',0.5,-1.)`, `('B',1.,2.)`, etc... Chaque triplet `t` représente un point du plan euclidien, non nom `t[0]` (une chaine de caractère) et ses coordonnées dans un repère cartésien `t[1]`, `t[2]` (deux flottants).
   (a) Écrire un script permettant de trier le tableau `Points` à l'aide d'un tri par insertion et par distance croissante à l'origine $(0,0)$ du repère.
   (b) Índiquer comment modifier le script pour que de plus des points situés à même distance de l'origine apparaissent triés par ordre alphabétique sur leur nom.

1. La fonction `abs` permet d'obtenir le module d'un complexe passé en argument. La stabilité dépend de la comparaison employée : inégalité stricte ou large.

2.a. $d(O,A)=\sqrt{x_A^2+y_B^2}$; puisque la fonction $x\mapsto\sqrt{x}$ est croissante, $d(0,A)<d(O,B)\iff x_A^2+y_A^2<x_B^2+y_B^2$. Utiliser une fonction prenant en paramètre un triplet `t` et qui renvoie `t[1]**2+t[2]**2`.

2.b. Utiliser le fait que le tri par insertion est stable. L'opérateur < permet aussi la comparaison de chaines de caractères selon l'ordre lexicographique (en particulier l'ordre alphabétique pour les caractères de l'alphabet.)

1. Tri par insertion sur un tableau de complexes, par modules croissants :

```
def tri_insertion(T):
    N = len(T)
    for i in range(1,N):
        z = T[i]
        d = abs(z)       # module du complexe
        k = i
        while k > 0 and abs(T[k-1]) > d:
            T[k] = T[k-1]
            k = k-1
        T[k] = z
    return T
```

2. Tri par insertion appliqué au tableau `Points` :

```
def norm(t):
    return t[1]**2+t[2]**2

N = len(Points)
for i in range(1,N):
    t = Points[i]
    d = norm(t)
    k = i
    while k > 0 and norm(Points[k-1]) > d:
        Points[k] = Points[k-1]
        k = k-1
    Points[k] = t
```

3. Le tri par insertion étant stable, il suffit de commencer par trier le tableau `Points` selon le nom avant d'appliquer le tri selon la distance à l'origine. Au-dessus du script précédent on insère le script suivant :

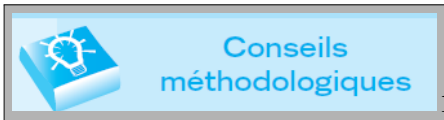
```
N = len(Points)
for i in range(1,N):
    t = Points[i]
    k = i
    while k > 0 and Points[k-1][0] > t[0]:
```

```
        Points[k] = Points[k-1]
        k = k-1
    Points[k] = t
```

4) ■ Test d'une variante du tri par sélection pour un tableau d'éléments distincts.

Le but de l'exercice est d'écrire et de tester une variante du tri par sélection, ne s'appliquant qu'à un tableau d'éléments deux à deux distincts, ou l'on recherche simultanément un minimum et un maximum pour les déplacer respectivement en première position et en dernière position.

1. Ecrire une fonction `echange(T,i,j)` prenant en paramètre un tableau `T` et deux entiers positifs `i`, `j`, et qui échange dans `T` ses éléments d'indice `i` et `j`.
2. Ecrire une fonction `indiceMinMax(T,i,j)` qui prend en paramètre un tableau `T` de nombres deux à deux distincts, et deux entiers positif `i<j` et qui retourne le couple des indices des éléments minimal et maximal dans le sous-tableau de `T` allant de l'indice `i` à l'indice `j`.
3. En appelant les fonctions précédentes, écrire une fonction `tri_double_selection(T)` prenant en paramètre un tableau de nombres deux à deux distincts et qui les ordonne dans le sens croissant de la façon suivante :
   - Chercher l'indice des éléments minimal et maximal de `T`,
   - échanger dans `T` l'élément minimal avec le premier élément de `T` et l'élément maximal avec le dernier élément de `T`,
   - Recommencer le même procédé pour les éléments de `L` du deuxième à l'avant-dernier élément
   - et ainsi de suite...
4. Déterminer la complexité dans le pire, le meilleur des cas et en moyenne de l'algorithme de tri obtenu.
5. Ecrire de même une fonction `indiceMin(T,i,j)` qui prend en paramètre un tableau `T` de nombres et qui renvoie l'indice d'un élément minimal dans le sous-tableau débutant à l'indice `i`.
6. Écrire une fonction `tri_selection` qui applique un tri par sélection en appelant les fonctions `echange` et `indiceMin` écrites ci-dessus.
7. Mesurer les temps d'exécution des deux fonctions de tri sur des tableaux aléatoires de longueurs 10, 20, ..., 990, 1000, et les stocker dans deux listes. Tracer dans une même fenêtre deux graphiques : sur le premier les deux courbes des temps d'exécution mesuré des deux algorithmes, et sur le deuxième les courbes des temps d'exécution restreints aux tableaux d'au plus 100 éléments.
8. Cette variante du tri par sélection présente-t-elle un intérêt?



1. La fonction ne renverra rien. L'échange au sein d'une fonction de deux éléments d'un tableau passé en paramètre, perdure à la sortie de la fonction.

2. Effectuer la recherche du minimum et du maximum simultanément ; autrement l'algorithme n'a aucun intérêt.

3. Structurer l'algorithme autour d'une boucle for où varie une variable `i` de `0` à `len(T)//2` ; autrement l'algorithme ne présente aucun intérêt.

3. La fonction `clock` du module `time` renvoie le temps processeur. Créer un seul tableau par compréhension de liste avant d'en faire une copie superficielle par slicing. La fonction `subplot` permet de générer des sous-graphique dans une figure.

1. Fonction `echange` :

```
def echange(T,i,j):
    T[i], T[j] = T[j], T[i]
```

2. Fonction `indiceMinMax` :

```
def indiceMinMax(T,i,j):
    Imin = Imax = j
    for k in range(i,j):
        if T[k] < T[Imin]:
```

```
            Imin = k
        elif T[k] > T[Imax]:
            Imax = k
    return Imin, Imax
```

3. Fonction `tri_double_selection` :

```
def tri_double_selection(T):
    n = len(T)
    for i in range(n//2):
        iMin, iMax = indiceMinMax(T,i, n-1-i)
        echange(T,i,iMin)
        echange(T,n-1-i,iMax)
    return T
```

4. Complexité de la fonction `tri_double_selection`. Soit un tableau `T` de longueur n. La fonction `echange(T,i,j)` a pour complexité $O(1)$ dans tous les cas. La fonction `indiceMinMax(T,i,j)` n'étant constitué que d'une boucle for, a pour complexité $\Theta(j-i)$. La complexité de `tri_double_selection(T)` est alors dans tous les cas :

$$\begin{aligned} O(1)+\sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} O(1)+\Theta(n-2i) &= O(1)+O(n)+\Theta\left(n\times\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)-\left\lfloor\frac{n}{2}\right\rfloor\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)\right) \\ &= O(1)+O(n)+\Theta\left(\frac{n}{2}\times\left(\frac{n}{2}+1\right)\right)=\Theta(n^2) \end{aligned}$$

Ainsi dans tous les cas, comme pour le tri par sélection, la complexité est quadratique.

5 et 6. Fonctions `indiceMin` et `tri_selection` :

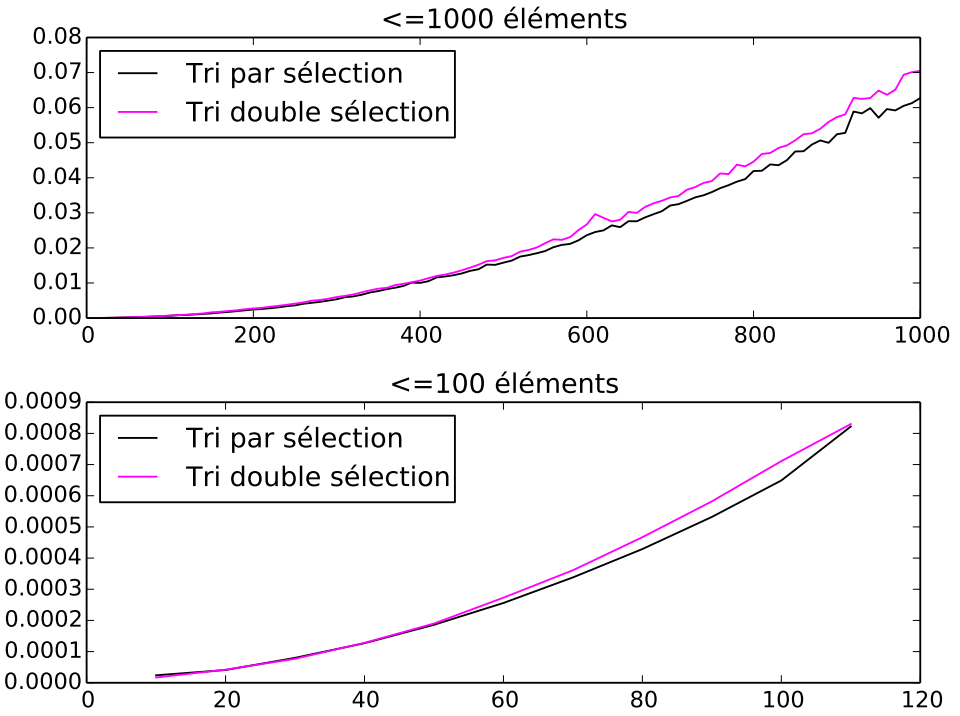
```
def indiceMin(T,i):
    Imin = i
    for k in range(i+1,len(T)):
        if T[k] < T[Imin]:
            Imin = k
    return Imin

def tri_selection(T):
    N = len(T)
    for i in range(N-1):
        iM = indiceMin(T,i)
        echange(T,i,iM)
    return T
```

7. Mesure et tracés des temps d'exécution :

```
from random import random
from time import clock
N = 1000
X, Y = [], []
for n in range(10,N+1,10):
    T0 = [random() for k in range(n)]
    T1 = T0[:]
    t0 = clock()
    tri_selection(T0)
    t1 = clock()
    tri_double_selection(T1)
    t2 = clock()
    X.append(t1-t0)
    Y.append(t2-t1)
import matplotlib.pyplot as plt
```

```
fig = plt.figure(1)
fig.subplots_adjust(hspace=0.3)  # ajustement de l'espace entre 2 graphiques
T = range(10,N+1,10)
plt.subplot(2,1,1)
plt.plot(T,X, color = 'black')
plt.plot(T,Y, color = 'magenta')
plt.legend(('Tri par sélection', 'Tri double sélection'),'upper left')
plt.title('<=1000 éléments')
plt.subplot(2,1,2)
T = T[:11]
plt.plot(T,X[:11], color = 'black'
plt.plot(T,Y[:11], color = 'magenta')
plt.legend(('Tri par sélection', 'Tri double sélection'),'upper left')
plt.title('<=100 éléments')
plt.show()
```



**** si possible mettre en couleur la courbe magenta***


La complexité quadratique des deux algorithmes apparaît clairement sur les graphiques

8. La mesure effectuée montre que cette variante ne semble présenter aucun intérêt ; le gain qu'on aurait pu escompter a largement souffert de l'augmentation des coûts à chaque itération des balayages. Oublions cette variante.

5) ■ Variantes efficaces du tri à bulle.

Le tri à bulle est très pénalisé par le phénomène des "tortues" : si un élément grand situé en début de tableau remonte rapidement à sa position finale, ce n'est pas le cas d'un élément petit situé en fin de tableau ; c'est ce qu'on appelle une "tortue". Certaines variantes du tri à bulle sont conçues pour résoudre ce problème des tortues.

1. **Le tri cocktail.** Le tri cocktail est une variante du tri à bulle qui consiste à chaque balayage à comparer les éléments adjacents (et les intervertir si nécessaire) successivement en montant puis en redescendant. Comme dans le tri à bulle, on recommence un balayage tant que l'on a changé quelque chose au balayage précédant. Il résout en partie le problème des tortues, mais l'amélioration qu'il apporte au tri à bulle n'est que légère.
   (a) Écrire une fonction `tri_cocktail` qui applique l'algorithme sur un tableau de nombre à retourner trié dans le sens croissant. L'algorithme devra être stable et en place, et aussi optimisé que possible.
   (b) Justifier la terminaison et la correction de l'algorithme en choisissant un invariant de boucle approprié.

   Il n'est pas difficile de montrer que le tri cocktail a mêmes complexités que le tri à bulle. Le gain est très modéré.
2. **Le tri à peigne (ou Comb sort).** Dans le tri à peigne les éléments sont comparés non pas avec leur voisin, mais avec les éléments espacés d'un certain pas. Ce pas décroit au fur et à mesure de l'exécution jusqu'au pas 1. En pratique le pas vaut initialement (la partie entière de) la longueur du tableau divisé par 1,3. On procède comme dans le tri à bulle, par balayage, en comparant chaque élément avec celui espacé du pas et situé à sa droite, et en les intervertissant si nécessaire. Au balayage suivant le pas est encore divisé par 1,3, et ainsi de suite, jusqu'à un pas minimal de 1. Lorsque le pas est de 1 (on retrouve un tri à bulle), l'algorithme s'arrête dès que le balayage n'a rien changé au tableau.

   Cet algorithme permet d'éliminer le phénomène des "tortues". Il a d'excellentes performances, comparables aux meilleurs algorithmes de tri, tout en restant de principe et à écrire, très simple.
   (a) Écrire une fonction `tri_peigne` qui applique l'algorithme sur un tableau de nombres à retourner trié dans le sens croissant. L'algorithme devra être en place.
   (b) Justifier de la terminaison et de la correction de l'algorithme.
   (c) Donner un contre-exemple qui montre que l'algorithme est instable.

   Sa complexité est assez délicate à calculer. Dans le pire des cas elle s'avère quadratique $O(N^2)$, dans le meilleur des cas linéaire $\Theta(N)$ et en moyenne $O(N\log(N))$. Le gain est très important : cet algorithme rivalise en termes de performances avec les meilleurs algorithmes de tri.
3. Mesurer pour un même tableau de 10 000 nombres aléatoires le temps d'exécution des tris à bulle, cocktail et à peigne.


Le tri à peigne (ou Comb sort), basé sur le tri à bulle et l'élimination des "tortues", en constitue une amélioration tout à fait remarquable. Ses performances rivalisent avec les meilleurs algorithmes de tri connus, malgré un principe qui demeure très simple.

1.a. Pour l'optimisation remarquer qu'après un balayage ascendant le plus grand élément a été déplacé à sa position définitive, et après le balayage descendant il en est de même du plus petit élément.

1.b. Se ramener à la terminaison et correction du tri à bulle en choisissant bien l'invariant de boucle.

2.a. Modifier l'algorithme du tri à bulle selon la description du tri à peigne.

2.b. Se ramener à la terminaison et correction du tri à bulle.

3. La fonction `clock` du module `time` renvoie le temps processeur. Créer un seul tableau par compréhension de liste avant d'en faire deux copies superficielles par slicing.

1.a.

```
def tri_cocktail(T):
    debut = 0
    fin = len(T)-1
    changement = True
```

```
    while changement:    # Balayage
        changement = False
        for i in range(debut,fin):   # montant
            if T[i] > T[i+1]:
                T[i], T[i+1] = T[i+1], T[i]
                changement = True
        fin -= 1
        for i in range(fin-1,debut-1,-1):   # descendant
            if T[i] > T[i+1]:
                T[i], T[i+1] = T[i+1], T[i]
                changement = True
        debut += 1
    return T
```

 Utiliser deux variables `debut` et `fin` pour délimiter la partie du tableau balayée.

1.b. Le plus simple est de choisir le même invariant de boucle que pour le tri à bulle : "Après le k-ème balayage les k derniers éléments du tableau sont les k plus grands et sont ordonnés dans le sens croissant".

 On pourrait démontrer que l'on a en fait l'invariant de boucle : "Après le k-ème balayage les k premiers éléments du tableau ainsi que les k derniers éléments sont à leur place définitive."

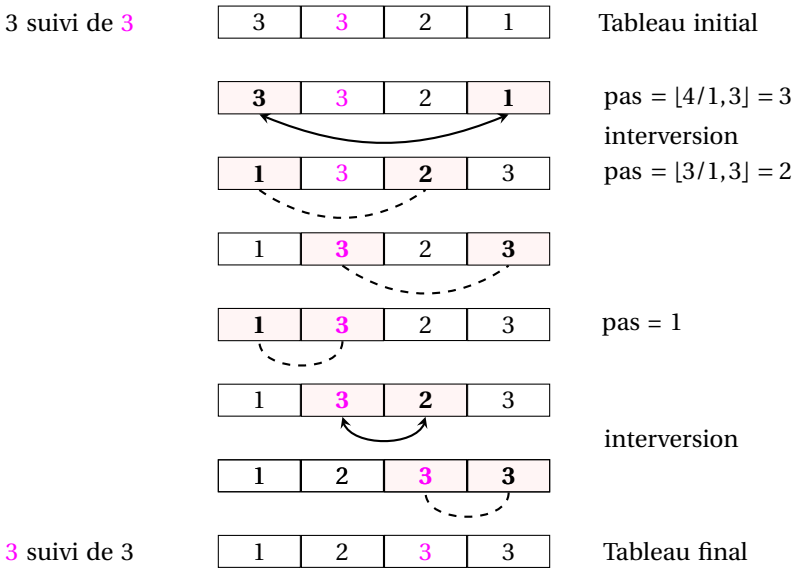
En effet, tout comme pour le tri à bulle, la partie ascendante du balayage déplace un nouvel élément en fin de tableau à sa place définitive. Or la partie descendante du balayage ne change rien aux k derniers éléments, puisqu'ils sont ordonnés et les k plus grands. La suite de l'argument utilisé pour la terminaison et la correction du tri à bulle s'applique alors aussi ici.

2.a. Fonction implantant un tri à peigne :

```
def tri_peigne(T):
    pas = len(T)
    changement = True
    while pas > 1 or changement:
        pas = max(1,int(pas/1.3))
        changement = False
        for i in range(len(T)-pas):
            if T[i] > T[i+pas]:
                T[i], T[i+pas] = T[i+pas], T[i]
                changement = True
    return T
```

2.b. Tout d'abord le pas `pas` est à valeur entière et diminue strictement jusqu'à atteindre la valeur `1` (instruction : `pas = max(1,int(pas/1.3))`) ; cette valeur `1` finit donc par être atteinte. Or dès que le pas égale 1 l'algorithme devient un tri à bulle dont on a déjà prouvé (cf. §2.3.4) la terminaison et correction. On en déduit la terminaison et correction de l'algorithme du tri à peigne.

2.c. Donnons pour contre-exemple le tableau à 4 éléments suivant :



Les deux éléments 3, 3 ne sont plus dans le même ordre.

3. Comparaison des temps d'exécution des 3 tris :

```
from random import random
from time import clock
N = 10000
T0 = [random() for k in range(N)]
T1 = T0[:]
T2 = T0[:]
t0 = clock()
tri_bulle(T0)
t1 = clock()
tri_cocktail(T1)
t2 = clock()
tri_peigne(T2)
t3 = clock()
print("Comparaison pour un tableau de",N,"éléments :")
print('Tri à bulle :',t1-t0,'s')
print('Tri cocktail :',t2-t1,'s')
print('Tri à peigne :',t3-t2,'s')
```

ce qui produit :

```
Comparaison pour un tableau de 10000 éléments :
Tri à bulle : 14.350749000000178 s
Tri cocktail : 12.45773199999985 s
Tri à peigne : 0.06959600000027422 s
```

Comme on le voit (seul) le tri à peigne donne de très bonnes performances. Son seul défaut est d'être instable.

6) ■ Le tri de Shell (Shell Sort).

Le tri de shell est une variante améliorant le tri par insertion. Il est construit à partir des observations suivantes :

- Le tri par insertion est très efficace sur des tableaux presque triées.
- Ce qui pénalise le tri par insertion est que chaque élément n'est déplacé que d'une position pas à pas .

Il est construit sur le tri par insertion comme le tri à peigne est construit sur le tri à bulle (voir exercice 5). C'est à dire qu'il s'opère comme un tri par insertion à ceci près que l'insertion s'effectue sur les sous-tableaux des éléments espacés d'un même pas $\geqslant 1$. Initialement le pas est important, et successivement après chaque balayage le pas diminue, d'un facteur d'environ 2,3, jusqu'à obtenir un pas de 1, dans quel cas il devient un tri par insertion sur un tableau presque trié.
Les pas optimaux ont été déterminés empiriquement, et sont :

$$1, 4, 10, 23, 57, 132, 301, 701.$$

au-delà, si la longueur du tableau est supérieure à 701, chaque pas suivant s'obtient en multipliant le plus grand pas par le facteur 2,3.

1. Écrire une fonction `pasOptimaux` prenant en paramètre un entier positif `N` et qui retourne la liste des pas comme définis ci-dessus adaptés à un tableau de longueur `N`.
2. Écrire la fonction `tri_shell` qui exécute un tri de Shell en place sur un tableau de nombre.
3. Justifier de la terminaison et correction du tri de Shell.
4. Mesurer le temps d'exécution des tri par insertion et tri de shell sur des tableaux aléatoires identiques de 10, 20, ..., 990, 1000 éléments. Tracer sur un graphique les courbes des temps d'exécution obtenus.
5. Construire un contre-exemple qui montre que le tri de Shell n'est pas stable.

1. Renvoyer une liste d'au moins 8 éléments.

2. `for x in reversed(liste)` permet de parcourir `liste` en sens inverse. Adapter l'algorithme vu en cours en faisant varier le pas au sein d'un tableau prédéfini de valeurs.

3. Se ramener au tri par insertion.

4. Écrire une fonction de tri par insertion.

1. Fonction `pasOptimaux` :

```
def pasOptimaux(N):
    L = [1,4,10, 23, 57, 132, 301, 701]
    while L[-1] < N:
        L.append(int(L[-1]*2.3))
    return L
```

2. Fonction `tri_shell` :

```
def tri_shell(T):
    N = len(T)
    PO = pasOptimaux(N)
    for pas in reversed(PO):
        for i in range(pas,N):
            x = T[i]
            k = i
            while k >= pas and T[k-pas] > x:
                T[k] = T[k-pas]
                k = k-pas
            T[k] = x
    return T
```

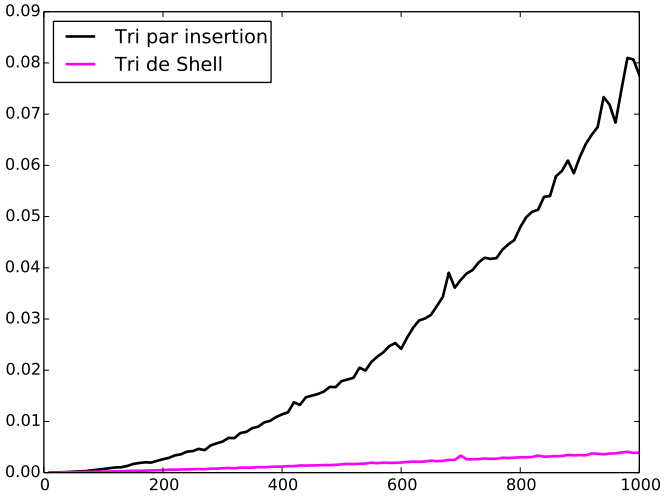
3. A chaque passage dans la boucle for principale la variable `pas` diminue strictement pour finir par valoir 1. Dès que c'est le cas le tri devient un tri par insertion, dont on a déjà prouvé la terminaison et correction (cf. §2.4.4).

4. Pour le tracé il faut d'abord écrire une fonction de tri par insertion.

```
def tri_insertion(T):
    N = len(T)
    for i in range(1,N):
        x = T[i]
        k = i
        while k > 0 and T[k-1] > x:
            T[k] = T[k-1]
            k = k-1
        T[k] = x
    return T

from random import random
from time import clock
import matplotlib.pyplot as plt
N = 1000
X = []
Y = []
for n in range(10,N+1,10):
    T0 = [random() for k in range(n)]
    T1 = T0[:]
    t0 = clock()
    tri_insertion(T0)
    t1 = clock()
    tri_shell(T1)
    t2 = clock()
    X.append(t1-t0)
    Y.append(t2-t1)
plt.figure(1)
T = range(10,N+1,10)
plt.plot(T,X, linewidth=2, color = 'black')
plt.plot(T,Y, linewidth=2, color = 'magenta')
plt.legend(("Tri par insertion","Tri de Shell"),'left upper')
plt.show()
```

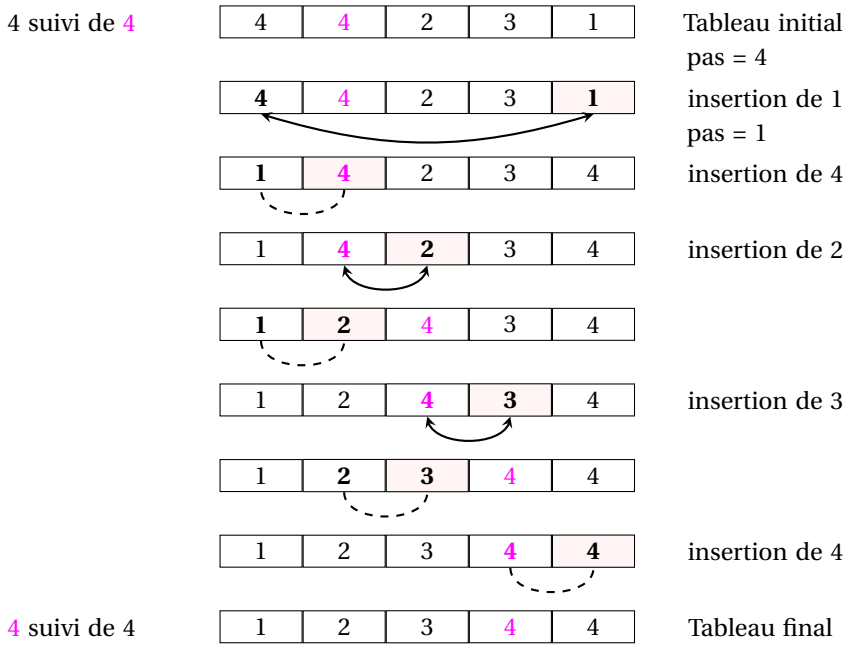
On obtient :



****si possible mettre en couleur la courbe magenta*****

Comme le montre le graphique, le tri de Shell est très performant. C'est une amélioration notable du tri par insertion. On visualise bien sur le graphique la complexité moyenne quadratique du tri par insertion. La complexité du tri de Shell dépend du tableau des pas employé; on connait des suites de pas donnant des complexités dans le pire des cas en $O(n^{3/2})$ ou en $O(n\log(n)^2)$. Le tableau des pas donné ici est empiriquement vérifié comme étant très efficace, mais sa complexité demeure un problème ouvert. Un autre problème ouvert est l'existence de pas donnant pour complexité la borne théorique optimale $\Theta(n\log(n))$.

5. Un contre-exemple à 5 éléments qui montre que le tri est instable :



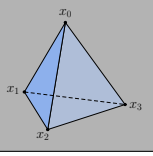
Les deux éléments 4, 4 ne sont plus positionnés dans le même ordre.

1. Tri par sélection. ..... **D'après Modélisation - Filières PC, TPC - 2016.**

Un simplexe dans $\mathbb{R}^p$ est défini par une famille libre de $p+1$ vecteurs de $\mathbb{R}^p$; le simplexe défini par la famille $\{x_0, x_1, \ldots, x_p\}$ est l'ensemble des barycentres :

$$\left\{ \sum_{i=0}^{p} \lambda_i \cdot x_i \in \mathbb{R}^p \;\middle|\; \forall i, \lambda_i \geqslant 0 \quad \text{et} \quad \sum_{i=0}^{p} \lambda_i = 1 \right\}$$

La famille est l'ensemble des sommets du simplexe. (Exemples : dans $\mathbb{R}$ un segment, dans $\mathbb{R}^2$ un triangle, dans $\mathbb{R}^3$ un tétraèdre).



Soit $z : \mathbb{R}^p \longrightarrow \mathbb{R}$ une fonction de p variables réelles. On souhaite écrire un algorithme de tri permettant de réindexer les sommets d'un simplexe à $p+1$ sommets $\left(X = \{x_0, x_1, \ldots, x_p\}\right)$ de sorte que les valeurs que prend la fonction $z(x)$ en chaque sommet du simplexe soient classées de manière croissante.

La méthode du tri par sélection repose sur la recherche du plus petit élément d'une liste. Une fois identifié, on échange cet élément avec le premier élément de la liste. Il s'agit d'une procédure itérative (cf. figure suivante). Lors de la première itération, on échange le premier élément ($i = 0$) avec le plus petit de la liste. Lors de la i-ème itération, on échange le i-ème élément par le plus petit élément en ne considérant que ceux à partir de la i-ème position.

Vecteur initial	10	2	- 3	5	- 1	20	60	48	- 15	9
1re itération	**- 15**	2	- 3	5	- 1	20	60	48	**10**	9
2e itération	*- 15*	**- 3**	**2**	5	- 1	20	60	48	10	9
3e itération	*- 15*	*- 3*	**- 1**	5	**2**	20	60	48	10	9

Ecrire le code permettant de réindexer les sommets du simplexe $X = \{x_0, x_1, \ldots, x_p\}$ de sorte que les valeurs que prend la fonction $z(x)$ en vchaque sommet du simplexe soient classées de manière croissante. On supposera que la fonction $z(x)$ est déjà définie.

Conseils méthodologiques

Appliquer un tri par sélection, avec recherche de minimum, dans l'ordre des valeurs de z.

Tri des sommets du simplexe :

```
n = len(X)
for k in range(1,n):
   ind = k
   for i in range(k+1,n):
      if z(X[i]) < z(X[ind]):
         ind = i
   X[k], X[ind] = X[ind], X[k]
```

2. Tri par insertion.....D'après épreuve écrite E.N.A.C. 2015.

On considère la fonction suivante qui s'applique à une liste `L` de nombres :

```
def Ordonnom(L):
    for i, s in enumerate(L):
        j=i
        while 0<j and s<L[j-1]:
            L[j]=L[j-1]
            j=j-1
            L[j]=s
    return L
```

Parmi les assertions suivantes lesquelles sont vraies?

A) `Ordonnom(L)` renvoie la liste `L` ordonnées dans le sens croissant.

B) `Ordonnom(L)` renvoie la liste `L` ordonnées dans le sens décroissant.

C) La complexité dans le cas le pire de `Ordonnom` est en $O(n^3)$ où n désigne la taille de la liste `L`.

D) La complexité dans le cas le pire de `Ordonnom` est en $O(n^2)$ où n désigne la taille de la liste `L`.

E) La complexité dans le cas le pire de `Ordonnom` est en $O(n)$ où n désigne la taille de la liste `L`.

On rappelle que l'instruction `for i, s in enumerate(L):` permet d'exécuter une boucle for pour laquelle la variable `s` parcourt les éléments de la liste `L`, tandis que la variable `i` parcourt les indices correspondants.

A) est vrai. (C'est ce qu'on appelle le *tri par insertion*).

B) est faux.

C), D) et E). Calcul de la complexité : la boucle `for` s'exécute pour chaque élément de la liste. Pour chacun d'entre eux, on l'insère à la bonne place en le comparant successivement aux éléments qui lui précèdent dans la liste (au sein de la boucle `while`).

Ainsi, dans le pire des cas, celui où la liste est ordonnée dans le sens décroissant, l'élément à l'indice `i` devra être inséré jusqu'en début de liste, soit `i-1` insertion. On a alors pour complexité :

$$\sum_{i=1}^{n}\Theta(i-1)=\Theta\left(\sum_{i=1}^{n}(i-1)\right)=\Theta\left(\sum_{i=0}^{n-1}i\right)=\Theta\left(\frac{n(n-1)}{2}\right)=\Theta(n^2).$$

La complexité dans le pire des cas est quadratique.

Ainsi :
C) est vrai bien qu'imprécis, puisque $n^2=O(n^3)$.

D) est vrai.

E) est faux. (Cependant la complexité du tri par insertion est linéaire dans le meilleur des cas).

3. VRAI/FAUX sur le Tri à bulle ..... D'après Écrit E.N.A.C. 2015.

A. On définit par le script suivant une fonction `ORDONNE` :

```
def ORDONNE(L):
    n=len(L)
    for i in range(0,n-1):
        if L[i]>L[i+1]:
            c=L[i]
            L[i]=L[i+1]
            L[i+1]=c
    return L
```

Parmi les assertions suivantes lesquelles sont vraies :

1. `ORDONNE([1,3,2,5,4])` renvoie `[1,2,3,4,5]`
2. Si `L` est une liste de nombres `ORDONNE(L)` renvoie une liste de nombres réordonnée dans l'ordre croissant.
3. `ORDONNE([1,3,5,2,4])` renvoie `[1,2,4,3,5]`
4. `ORDONNE(ORDONNE([3,2,1]))` renvoie `[1,2,3]`.

B. L est une liste de n entiers quelconques. Parmi les assertions suivantes lesquelles sont vraies :

1. Pour ordonner `L`, il suffit d'utiliser $n-1$ fois la fonction `ORDONNE`.
2. Pour ordonner `L`, il suffit d'utiliser n fois la fonction `ORDONNE`.
3. Une liste `L` ordonnée dans le sens décroissant n'est pas modifiée par la fonction `ORDONNE`.
4. Une liste `L` ordonnée dans le sens croissant est un invariant pour la fonction `ORDONNE`.

NB : La fonction ORDONNE est sauvegardée dans un fichier.

C. On définit à la suite du fichier, par le script suivant, une fonction Ordonnum :

```
def Ordonnum(L):
    n=len(L)
    for i in range(0,n-1):
        L=ORDONNE(L)
    return L
```

L est une liste de n entiers quelconques. Parmi les assertions suivantes lesquelles sont vraies :

1. `Ordonnum` est une fonction dont l'argument est un réel.
2. `Ordonnum` est une fonction dont la valeur de sortie est un réel.
3. `Ordonnum(L)` renvoie `True` si `L` est ordonné dans le sens croissant.
4. `Ordonnum(L)` renvoie les éléments de la liste `L` ordonnés dans le sens croissant.

B.1. Remarquer qu'un exécution de `ORDONNE(L)` déplace le plus grand élément en dernière position.

A.

1. Vrai.

2. Faux. Par exemple si `L = [3, 2, 1]`, `ORDONNE(L)` renverra `[2, 1, 3]`.

3. Faux. Il renvoie `[1, 3, 2, 4, 5]`.

4. Vrai. `ORDONNE(L)` renvoie `[2, 1, 3]` et `ORDONNE([2, 1, 3])` renvoie `[1, 2, 3]`.

B. 1. Vrai. Après l'exécution de `ORDONNE(L)` le plus grand élément de `L` a été déplacé en dernière position. Après une deuxième exécution les deux plus grands éléments sont aux dernières positions, et dans ordonnés le sens croissant. Après $n-1$ exécutions de `ORDONNE(L)` les $n-1$ éléments sont les $n-1$ plus grands, en dernières positions et ordonnés d ans le sens croissant. Le premier élément de la liste est donc le plus petit.

2. Vrai, mais $n-1$ exécutions suffisent.

3. Faux. Vois l'exemple de `[3, 2, 1]` qui devient `[2, 1, 3]`.

4. Vrai. Rien n'est changé pour une liste ordonnée dans le sens croissant.

C.

1. Faux, ce doit être une liste d'éléments ordonnables.

2. Faux. `Ordonnum(L)` renvoie une liste.

3. Faux.

4. Vrai.


C'est un *tri à bulle* qu'exécute `Ordonnum`.

4. Authentification des usagers . . . . . Modélisation, concours CCP, Filière TSI, 2016.

Des usagers sont détenteurs de badge portant un numéro pour identification pouvant être lu par un lecteur RFID(1).


(1) : RFID (de l'anglais radio frequency identification) : technologie d'identification automatique qui utilise le rayonnement radiofréquence pour identifier les objets porteurs d'étiquettes lorsqu'ils passent à proximité d'un interrogateur. (source : CNRFID - Centre national de référence RFID)

Un fichier ASCII `fichier_badges_auto.txt` contient l'ensemble des numéros des badges autorisés; le contenu du fichier est de la forme suivante :

12020501
12020508
12020506
12020502
12020504
12020509
12020503
.....

1. Écrire un script qui permet de stocker au préalable l'ensemble des numéros des badges issu du fichier dans un tableau `tab` d'entiers.
2. En pseudo langage, on considérera que les indices commencent à 1. Soit l'algorithme de la fonction `travail_sur_tableau()` suivant :

```
travail_sur_tableau(tab)

Fonction  permettant de ……..
Entrée :        tab[] tableau contenant les valeurs des badges.
Sortie :        ……..

Début
    n←longueur(tab)
    i←2
    Tant que ( i<=n )
        x ← tab[i]
        j ← i-1
        Tant que ( j>0 ET tab[j]>x )
            tab[j+1] ←tab[j]
            j ← j-1
        Fin Tant que
        tab[j+1]← x
        i←i+1
    Fin Tant que
Fin
```

   Expliquer le rôle de cet algorithme.
3. Pour la boucle externe, indiquer quelle variable joue le rôle de variant de boucle. Justifier que cette boucle se termine.
4. En justifiant votre réponse, préciser si la classe de la complexité de cet algorithme est linéaire, quadratique, cubique, linéarithmique (quasi linéaire) ou logarithmique.
5. Nous désirons à présent rechercher la position de l'occurrence du numéro du badge d'un utilisateur dans le fichier des badges autorisés matérialisé ici par le tableau supposé trié `tab`.

   Écrire une fonction `recherche(x, tab)` qui renvoie l'indice où l'élément `x` apparaît dans `tab`, et `None` s'il n'apparaît pas. La recherche se fera par dichotomie.

Conseils méthodologiques

1. Réviser la lecture de fichier texte (cf. chapitre 1).
2. Reconnaître un algorithme de tri connu.
3. Pour la terminaison de l'algorithme il faudra aussi déterminer un variant pour la deuxième boucle.
4. Considérer les coplexités dans le pire et dans le meilleur des cas.
5. La dichotomie se fait sur les indices délimitant la partie du tableau où poursuivre la recherche.

1. Lecture du fichier et création du tableau `tab` :

```
file = open('fichier_badges_auto.txt', 'r')
tab = [ ]
for ligne in file:
    tab.append(int(ligne))
file.close()
```

2. La fonction exécute un tri par insertion. Il permet de trier le tableau `tab` dans l'ordre croissant. En sortie le tableau `tab` a été trié en place.

3. L'expression `n-i+1` est un variant de boucle. En effet, initialement elle vaut la longueur du tableau -1, à cause de l'instruction `i += 1` la quantité diminue strictement à chaque passage dans la boucle, et la condition `i <= n` assure que `n-i+1` reste positif. C'est donc un variant de boucle. Si la deuxième boucle s'achève, le programme se termine alors. Or la deuxième boucle admet `j` comme variant de boucle évident. Ceci justifie la terminaison du programme.

4. La complexité de l'algorithme dans le pire des cas est quadratique. La première boucle s'itèrera `n-1` fois, la deuxième au plus `i` fois. Dans le cas le plus défavorable, celui d'une liste triée dans le sens décroissant, la seconde boucle s'itèrera exactement `i-1` fois, car en début de boucle l'élément `x` sera strictement inférieur à tous ceux qui le précédent. A l'intérieur de la boucle le nombre d'opérations élémentaires est bornée. Le calcul de la complexité devient alors :

$$O(1)+\sum_{i=2}^{n}\left(\Theta(1)+\sum_{k=1}^{i-1}\Theta(1)\right)=O(1)+\Theta\left(\frac{(n+2)(n-1)}{2}\right)=\Theta(n^2)$$

Dans le meilleur des cas, celui d'une liste triée dans le sens croissant, la boucle while ne s'exécute pas (puisque en début de boucle l'élément `x` est supérieur ou égal à l'élément qui le précède). La complexité est alors linéaire.

5. Fonction de recherche par dichotomie :

```
def recherche(x, tab):
    debut = 0                    # indice du début
    fin = len(x)-1               # indice de la fin
    while fin-debut >= 0:
        milieu = (debut+fin)//2  # indice au milieu
        if tab[milieu] == x:     # élément trouvé
            return milieu
        elif tab[milieu] < x:
            debut = milieu + 1   # dichotomie à droite
        else:
            fin = milieu -1      # dichotomie à gauche
    return None                  # élément non trouvé
```

5. Tri par insertion. ..... D'après Mines-Ponts 2016.

On se donne la fonction tri suivante, écrite en Python :

```
def tri(L):
    n= len(L):
    for i in range(1, n):
        j=i
```

```
    x = L[i]
    while 0 < j and x < L[j-1]:
        L[j] = L[j-1]
        j = j-1
    L[j] = x
```

1. Lors de l'appel `tri(L)` lorsque `L` est la liste `[5, 2, 3, 1, 4]`, donner le contenu de la liste `L` à la fin de chaque itération de la boucle `for`.
2. Soit `L` une liste non vide d'entiers ou de flottants. Montrer que "la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération i" est un invariant de boucle. En déduire que `tri(L)` trie la liste `L`.
3. Évaluer la complexité dans le meilleur et dans le pire des cas de l'appel `tri(L)` en fonction du nombre n d'éléments de `L`.

On souhaite, partant d'une liste constituée de couples (chaîne, entier), trier la liste par ordre croissant de l'entier associé suivant le fonctionnement suivant :

```
>>> L = [['Bresil', 76], ['Kenya', 26017], ['Ouganda', 8431]]
>>> tri_chaine(L)
>>> L
[['Bresil', 76], ['Ouganda', 8431], ['Kenya', 26017]]
```

4. On dispose d'une liste Python stockée dans la variable `deces2010` et constituée de couples (chaîne, entier). C'est la liste des nombres de décès dus au paludisme en 2010 pour chaque pays.

   Écrire un script en Python pour trier la liste `deces2010` par ordre croissant du nombre de décès dus au paludisme en 2010.

2. Procéder par récurrence.

3. Les pire et meilleur des cas sont réalisée pour des listes triées, dans le bon et mauvais sens ; ceux pour lesquels chaque insertion est minimale, respectivement maximale.

3. Adapter le code de la fonction `tri`, en modifiant ce qui doit l'être.

1. Contenu de la liste `L` après chaque insertion :

5	2	3	1	4	Tableau initial
2	5	3	1	4	`i = 1`. Insertion de 2
2	**3**	5	1	4	`i = 2`. Insertion de 3
1	2	3	5	4	`i = 3`. Insertion de 1
1	2	3	**4**	5	`i = 4`. Insertion de 4
1	2	3	4	5	Tableau final

2. Montrons que "la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération i" et un invariant de boucle. Par récurrence sur `i` :

Initialisation. Après la 1ère itération, `i=1`, et `L[0:2]` contient les deux premiers éléments de `L` dans le même ordre.

Premier cas : si au début de l'itération `L[0]` $\leqslant$ `L[1]`. Alors `x = L[1]` et la condition `x < L[0]` n'est pas vérifiée. Ainsi il n'y pas de passage dans la boucle `while`, et l'instruction `L[1] = x` ne change rien à la liste.

Deuxième cas : si au début de l'itération `L[0]`>`L[1]`, alors la condition `0 < j and x < L[0]` est vérifiée ; il y a un seul passage dans la boucle `while` et la suite d'instructions exécutés lors de l'itération est :

```
    j = 1
    x = L[1]
    # debut boucle while
    L[1] = L[0]
    j = 0
    # fin boucle while
    L[0] = x
```

qui échange les éléments `L[0]` et `L[1]` dans `L`.

Dans les deux cas, à l'issue de l'itération `i=1`, la liste extraite `L[0:2]` est constituée de `L[0]` et `L[1]` classés dans l'ordre croissant. L'invariant de boucle est vérifié.

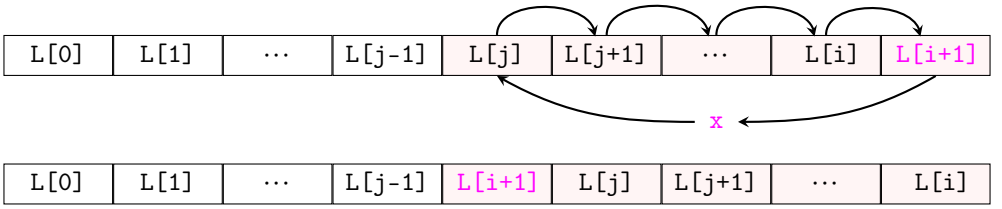
Hérédité. Supposons qu'à l'issue de la `i`-ème itération de la boucle for l'invariant de boucle soit vérifié et qu'une `(i+1)`-ème itération se produise.

Par hypothèse de récurrence : $L[0] \leqslant L[1] \leqslant \cdots \leqslant L[i]$.

En début d'itération, `x` est initialisé avec la valeur `L[i+1]` et `j` avec `i+1`. Il y a encore deux cas :

Premier cas : si $L[i] \leqslant L[i+1]$. Alors la condition de la boucle `while` n'est pas vérifiée, et l'itération termine sur l'instruction `L[i+1] = x`, qui ne change rien à `L`. Ainsi : $L[0] \leqslant L[1] \leqslant \cdots \leqslant L[i] \leqslant L[i+1]$.
L'invariant de boucle est vérifié au rang `i+1`.

Deuxième cas : si $L[i] > L[i+1]$. Alors la condition de la boucle `while 0 < j and x < L[i]` est vérifiée. Soit `j-1` l'indice du dernier élément de `L[0:i+1]` qui soit $\leqslant$ `x`, s'il en est, `-1` sinon. A l'issue de la boucle `while`, et de l'instruction `L[j] = x` qui la suit, le tableau a été modifié selon le schéma suivant :



Les déplacements schématisés au-dessus du tableau sont exécutées du plus à droite au plus à gauche à chaque passage dans la boucle `while` grace à l'instruction `L[j] = L[j-1]`. Les déplacements figurant au-dessous du tableau sont exécutés avant le boucle `while`, grâce à l'instruction `x = L[i+1]`, et après la boucle, grâce à `L[j] = x` ; le stockage de `L[i+1]` dans `x`, permet de mémoriser la valeur de `L[i+1]` avant l'instruction `L[i+1] = L[i]`, pour la déplacer à la fin de l'insertion, à la position `j` grace à `L[j] = x`. Ainsi à l'issue de l'itération de la boucle `for` les `(i+2)` premiers éléments de `L` (`L[0:i+2]`) sont désormais ordonnés dans le sens croissant :

$$L[0] \leqslant L[1] \leqslant \cdots \leqslant L[j-1] \leqslant L[i+1] \leqslant L[j] \leqslant L[j+1] \leqslant L[i].$$

L'invariant de boucle est vérifié au rang `i+1`. Cela conclut la récurrence. □

En particulier, grâce à l'invariant de boucle, si `L` est de longueur n, alors après la $(n-1)$-ème et dernière itération de la boucle `for`, la liste `L` est triée.

3. Calcul de la complexité, pour une liste de n éléments :
– À cause de la boucle `for i in range(1,n)` l'algorithme est au moins de complexité $\Theta(n)$.
– À cause de la boucle `while`, qui s'exécute au plus $n-1$ fois, l'algorithme est au plus de complexité $\Theta(n^2)$.
Montrons que ces bornes sont atteintes dans les meilleur et pire des cas.

Pour le meilleur des cas, considérons une liste de n éléments déjà triés dans l'ordre croissant. La condition `x < L[j-1]` n'étant jamais réalisée dans ce cas, la boucle `while` ne s'exécute pas et la complexité est $\Theta(n)$.

Pour le pire des cas, considérons une liste de n éléments triés dans le sens décroissant. Chaque élément de cette liste est alors inférieur à tous ceux qui le précèdent. Pour cela, à l'itération `i` de la boucle `for`, la boucle `while` s'itèrera exactement `(i-1)` fois. La complexité est alors d'ordre :

$$\sum_{i=1}^{n-1}\left(O(1)+\sum_{j=1}^{i-1}\Theta(1)\right)=\sum_{i=1}^{n-1}\Theta(i)=\Theta\left(\sum_{i=1}^{n-1}i\right)=\Theta(n^2)$$

En résumé : l'algorithme de tri est de complexité linéaire dans le meilleur des cas, quadratique dans le pire des cas.

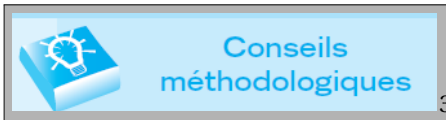
4. Script permettant de trier la liste `deces2010` par ordre croissant des paramètres entiers.

```
n = len(deces2010):
for i in range(1, n):
    j = i
    x = L[i]
    while 0 < j and x[1] < L[j-1][1]:
        L[j] = L[j-1]
        j = j-1
    L[j] = x
```

6. Tri comptage.....Epreuve Type - Oral Banque PT.

Soit N un entier naturel non nul. On cherche à trier une liste `L` d'entiers naturels strictement inférieurs à `N`.

1. Écrire une fonction comptage, d'arguments `L` et `N`, renvoyant une liste `P` dont le k-ième élément désigne le nombre d'occurences de l'entier k dans la liste `L`.
2. Utiliser la liste `P` pour en déduire une fonction `tri`, d'arguments `L` et `N`, renvoyant la liste `L` triée dans l'ordre croissant.
3. Tester la fonction `tri` sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement.
4. Quelle est la complexité temporelle de cet algorithme ?



3. On pourra utiliser la fonction `randint(0,5)` du module `random` qui retourne un nombre entier pseudo-aléatoire entre 0 et 5 inclus.
4. On exprimera la complexité en fonction du couple (N,n). On commencer par étudier celle de la fonction `comptage` avant de calculer la complexité de la fonction `tri`.

1) Fonction `comptage`.

```
def comptage(L,N):
    P = [0] * N
    for n in L:
        P[n] += 1
    return P
```

2) Fonction `tri`.

```
def tri(L,N):
    P = comptage(L,N)
    i = 0
    for j in range(N):
        x = P[j]
        for k in range(x):
            L[i] = j
            i += 1
    return L
```

3) La fonction `randint(0,5)` du module `random` retourne un nombre entier pseudo aléatoire entre 0 et 5 inclus.
On pourrait aussi utiliser la fonction `random()` qui retourne un float dans $[0,1[$ (loi pseudo-uniforme), et `int(6*random())`.

```
In [1]: from random import randint
In [2]: L = [ randint(0,5) for i in range(20) ]
In [3]: tri(L, 6)
Out[3]:
[0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 3, 3, 5, 5, 5, 5, 5, 5, 5]
```

4) Complexité de `comptage` :

N opérations pour créer le tableau P ; parcours de `L` par la boucle `for` : `len(L)` opérations : incrémentations d'un élément de P (l'écriture d'un élément dans une liste est en $O(1)$).

Pour un tableau de n éléments , constituées de valeurs dans $[[0, N]]$, la complexité temporelle de `comptage()` est en $\Theta(\max(n, N))$ (et spatiale en $O(N)$).
Complexité de `tri` :

C'est la complexité de comptage ($\max(n, N)$) plus la complexité de la reconstitution de la liste `L` triée :
Reconstitution de `L` triée : de l'ordre de :

$$P[0] + P[1] + \cdots + P[N-1] = n \quad \text{opérations}$$

(puisque la somme du nombre d'occurrences des éléments apparaissant dans `L` égale son nombre d'éléments au total).

Reconstitution de `L` trié de complexité : $\Theta(n)$.

Ainsi l'algorithme est de complexité : $\Theta(\max(N, n))$ (où n est la longueur de la liste L à trier.

L'algorithme est intéressant seulement lorsque les valeurs sont des entiers positifs peu étalés (N peu grand devant n).
Pour des entiers uniformément bornés, il est intéressant car de complexité linéaire.
Autrement lorsque $N >> n$ il faut l'éviter et privilégier d'autres algorithmes de tri.

7. Compréhension de code . . . . . d'après : Épreuve type de l'épreuve spécifique d'Informatique de Mines-Telecom.

On considère le code suivant :

```
def f (x) :
    i = 0
    j = 0
    k = len(x) - 1
    while (i <= k) :
        if (x[i] == 0) :
            x[i], x[j] = x[j], x[i]
            j += 1
            i += 1
        elif (x[i] == 1) :
            x[i] = x[k]
            x[k] = 1
            k -= 1
        else :
            i += 1
```

1. De quel type doit être le paramètre `x` ?
2. Justifier la terminaison de l'algorithme.
3. Évaluer la complexité de la fonction f en fonction d'un paramètre que l'on définira.
4. Que fait la fonction f ? Justifier votre réponse.
5. Lorsque `x` est un tableau de nombre réels, interprétez l'algorithme comme un algorithme de tri selon un caractère que l'on définira.

2. Montrer que `k-i+1` est un variant de la boucle while.
3. Utiliser le variant établi en 2) pour évaluer le nombre de passages dans la boucle.
4. Mettre en évidence des invariants de boucle portant sur les variables `j`, `k` et `i`.
5. Définir une fonction $f : \mathbb{R} \longrightarrow \mathbb{R}$ appropriée pour définir l'ordre employé.

1. Le paramètre doit être une séquence (à cause de l'indexation `x[i]` et doit être mutable (à cause de l'affectation `x[i], x[j] = x[j], x[i]`). Il faudra que `x` soit un tableau : liste ou tableau `numpy`.

2. On a pour variant de boucle `k-i+1` : valant initialement `len(x)`, cette expression décroit strictement à chaque passage dans la boucle `while` : exactement une des 2 instructions `i += 1` et `k -= 1` est exécuté à chaque passage. La condition `i <= k` assure que `k-i+1` reste positif à l'exécution du programme. Ainsi l'algorithme se termine.

3. L'expression `k-i+1`, comme nous l'avons vu à la question précédente vaut initialement `len(x)`, décrémente à chaque passage dans la boucle, jusqu'à ce que sa valeur atteigne 0, après quoi l'algorithme s'arrête. Il y a donc exactement `len(x)` passages dans la boucle. Puisqu'en amont de la boucle, et au sein de la boucle, seules $\Theta(1)$ opérations élémentaires sont exécutées, la complexité est dans tous les cas linéaire $\Theta(n)$ en fonction du nombre d'éléments n =`len(x)` du tableau `x`.

4. La fonction `f` parcourt le tableau `x` et déplace les `0` en début de tableau et les `1` en fin de tableau.
En effet on a l'invariant de boucle :
• "Avant l'indice `j` se trouvent tous les zéros ayant été déplacés en début de tableau."
En effet c'est vrai initialement lorsque `j=0` et qu'aucun zéro n'a été déplacé, et ça reste vrai à chaque passage dans la boucle, puisque les seules instructions modifiant `j` et insérant des zéros en début de tableau sont :

```
    ...
if (x[i] == 0) :
    x[i], x[j] = x[j], x[i]
    j += 1
    ...
```

qui insère un zéro à la position `j` avant d'incrémenter `j`.

De même on a pour invariant de boucle :
• "Après l'indice `k` se trouvent tous les uns ayant été déplacés en fin de tableau."
C'est vrai initialement puisque `k = len(x)-1`, et ça reste vrai ç chaque passage dans la boucle puisque la seule partie du code modifiant `k` et insérant des zéros en fin de tableau est :

```
    ...
elif (x[i] == 1) :
    x[i] = x[k]
    x[k] = 1
    k -= 1
    ...
```

qui insère un `1` en fin de tableau à l'indice `k` avant de décrémenter `k`.

Finalement on a l'invariant :

• "Tous les zéros et les uns situés avant l'indice `i` ont été déplacés en début ou fin de tableau".

à cause des deux portions de code donnés ci-dessus.

Remarquons aussi que toutes les modifications apportées au tableau sont des échanges qui ont eu lieu durant les déplacements des zéros et des uns.

Puisque la fin de l'algorithme le variant de boucle `k-i+1` vaut `0`, `i = k+1`, et d'après les invariants de boucles établis ci-dessus tous les zéros ont été déplacés en début de tableau, tous les uns en fin de tableau, entre les deux tous les autres éléments du tableau.

5. En considérant par exemple l'application $f : \mathbb{R} \longrightarrow \mathbb{R}$ définie par :

$$x \longmapsto f(x) = \begin{cases} 0 & \text{si x = 0} \\ 1 & \text{si x = 1} \\ 1/2 & \text{sinon} \end{cases}$$

l'algorithme exécuté par `f(x)` exécute un tri d'un tableau numérique `x` par ordre croissant des valeurs prises par f.

L'algorithme de tri utilisé est alors un tri comptage.