Fabrication d'une liste d'entiers aléatoires :

Code Python : Voici une version sous forme d'une fonction Python, qui vérifie au préalable que les arguments que l'utilisateur donnera à la fonction sont acceptables.

```python
# fonction qui renvoie une liste de 'quantite' entiers aléatoires choisis entre 'debut' et 'fin' compris
def liste_entiers_aleatoires(debut : int, fin : int, quantite : int) -> list:
    assert isinstance(debut, int) and isinstance(fin, int) and isinstance(quantite, int)
    assert debut <= fin and quantite >= 1
    liste = []
    for entier in range(quantite):
        liste.append(randint(debut, fin))
    return liste
```

La ligne suivante est purement indicative.

```python
def liste_entiers_aleatoires(debut : int, fin : int, quantite : int) -> list:
```

Elle ne fait aucune vérification, mais elle permet de lire le type des arguments nécessaires à l'utilisation de la fonction et précise le type de la sortie.

Les lignes suivantes permettent de faire les vérifications.

```python
assert isinstance(debut, int) and isinstance(fin, int) and isinstance(quantite, int)
assert debut <= fin and quantite >= 1
```

On vérifie que les arguments **debut**, **fin** et **quantite** sont bien des entiers.
Puis que **debut**, **fin** et **quantite** sont cohérents.

La commande suivante provoque une erreur : **>>> liste_entiers_aleatoires** (49, 0, 25)

```
>>> liste_entiers_aleatoires(49, 0, 25)
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
  File "F:\@new_lab\@cours\@niveaux\@premieres-nsi\algorithmes\tri\pnsi_programmes_bases.py", line 20, in liste_entiers_aleatoires
    assert debut <= fin and quantite >= 1
AssertionError
>>>
```

Version simplifiée :

```python
# même fonction simplifiée, sans les vérifications sur les arguments de la fonction
def random_integer_list(lower, upper, length) -> list:
    return [ randint(lower, upper) for _ in range(length) ]
```

Remarque :

La variable compteur **i** étant inutile dans la ligne suivante :

```
return [ randint(lower, upper) for i in range(length) ]
```

On peut la remplacer par un underscore "_" comme dans l'exemple de la fonction donnée.

Python n'aime pas trop qu'on utilise des variables qui ne servent à aucun endroit.

Recherche d'une occurence dans une liste non triée :

On recherche la présence ou non d'une **valeur** dans un **tableau** de taille n.
A priori on va écrire un algorithme de ce genre pour renvoyer l'indice de **valeur** dans **tableau** si cette valeur y est présente, et -1 sinon :

Code Python, sous la forme d'une fonction :

```python
# recherche naïve d'une occurence dans une liste non triée
def occurence(tableau : list, valeur : int) -> int:
    """ la fonction renvoie la position de valeur dans tableau
        si la valeur est présente, et -1 sinon"""
    assert isintance(tableau, list) and isintance(valeur, int)
    for indice in range(len(tableau)):
        if tableau[indice] == valeur:
            return indice
    return -1
```

Complexité :

Comme dans le pire des cas on parcourt toute la liste une seule fois, cet algorithme est de **complexité linéaire** en la taille n de la liste, donc en $O(n)$.

C'est-à-dire que le nombre maximum d'opérations arithmétiques ou logiques nécessaires à la réalisation de cet algorithme est proportionnel à n.

Si la taille des données double, alors le temps d'exécution double également.

Mais si l'on dispose d'une liste triée au départ, on peut améliorer la complexité de la recherche, en utilisant justement le fait que cette liste soit triée.

Pour cela, on utilise le principe de la **dichotomie**, qui consiste à diviser par deux la taille des données restant à étudier à chaque étape.

Cet algorithme est traité plus loin.

Recherche des extremums dans un tableau :

On peut décider de déterminer les valeurs extrémales (minimum et maximum) d'un tableau ou, ce qui revient au même et est même plus souple, déterminer les indices de ces valeurs extrémales, ce qui est fait à travers l'algorithme suivant :

Code Python :

```
43  # fonction qui renvoie les indices des valeurs extrêmes d'une liste
44  def extremums(tableau : list) -> tuple:
45      assert isinstance(tableau, list) and len(tableau)
46      indice_mini = 0
47      indice_maxi = 0
48      for indice in range(1, len(tableau)):
49          if tableau[indice] < tableau[indice_mini]:
50              indice_mini = indice
51          elif tableau[indice] > tableau[indice_maxi]:
52              indice_maxi = indice
53      return indice_mini, indice_maxi
```

Complexité :

On parcourt une seule fois la liste complète, donc cet algorithme est de compléxité **linéaire**.

Remarque : La sortie est de type **tuple**, c'est-à-dire que la fonction renvoie un couple de valeurs qui est affiché sous la forme (ind_mini, ind_maxi) en sortie.

Pour accéder à l'indice du minimum ou du maximum, il faut utiliser :

mini = extremums(tableau)[0] et maxi = extremums(tableau)[1]

ou plus simplement :

mini, maxi = extremums(liste)

```
>>> mini, maxi = extremums(liste)
>>> mini
0
>>> maxi
49
>>>
```

L'algorithme de tri par insertion :

C'est la méthode qu'on utilise généralement pour classer les cartes d'un jeu.

Principe :

- On parcourt la liste du début à la fin.
- Au moment où l'on considère l'élément k , les éléments qui le précèdent sont déjà triés, et les suivants sont ceux qui restent à trier.
- L'objectif est d'insérer l'élément k au bon endroit parmi les précédents tant qu'il est plus petit que ces prédécesseurs, et qu'on ne sort pas de la liste.

En language naturel :

```
n ← longueur de la listeTab
pour k de 1 à n - 1 :
    x ← Tab[k]                      # on mémorise l'élément à trier
    j ← k                           # on se repère à partir de l'indice k
    tant que j > 0 et Tab[j - 1] > x :
        Tab[j] ← Tab[j - 1]         # on décale vers la droite les éléments Tab[0], Tab[1]
        j ← j - 1                   # Tab[2], …, Tab[i-1] qui sont plus grands que x
    Tab[j] ← x                      # on place ensuite x dans le trou laissé par le décalage
```

Complexité :

Comme cet algorithme comporte deux boucles imbriquées chacune d'une longueur maximale la taille n de la liste à trier, il est de **complexité quadratique** en $O(n^2)$, ce qui signifie que le nombre maximum d'opérations est proportionnel à n^2 .

Code Python :

```python
for i in range(1, len(tableau)):
    valeur = tableau[i]
    j = i
    while tableau[j-1] > valeur and j > 0:
        tableau[j] = tableau[j-1]
        j = j - 1
    tableau[j] = valeur
```

L'algorithme de tri par sélection :

Principe : Sur un tableau **Tab** de n éléments numérotés de 0 à $n-1$.

- Recherche du plus petit élément de la liste, et l'échanger avec l'élément d'indice 0.
- Recherche du second plus petit élément, et l'échanger avec l'élément d'indice 1.
- On continue ainsi jusqu'à ce que la liste soit entièrement triée.

En language naturel :

```
n ← longueur de la listeTab
pour i de 0 à n - 2 :
        ind_min ← i
        pour j de i + 1 à n - 1 :
                si Tab[ j ] < Tab[ ind_min ] :
                ind_min ← j
        si ind_min ≠ i :
                échanger Tab[ i ] avec Tab[ ind_min ]
```

Complexité :

Cet algorithme est constitué de deux boucles imbriquées de longueur maximale n, donc sa **complexité** est également **quadratique** en $O(n^2)$, donc proportionnelle à n^2.

Code Python :

```python
for i in range(len(tableau) - 1):
    index_mini = i
    for j in range(i + 1, len(tableau)):
        if tableau[j] < tableau[index_mini]:
            index_mini = j
    if index_mini != i:
        tableau[index_mini], tableau[i] = tableau[i], tableau[index_mini]
    print(tableau)
```

Remarques :

Le tri par sélection est un tri **en place**, c'est-à-dire que les éléments sont triés directement dans la liste elle-même.

Implémenté de cette manière, ce tri n'est pas **stable**, c'est-à-dire que l'ordre d'apparition des éléments égaux n'est pas préservé.

Recherche d'une occurrence dans une liste triée :

On dispose de la liste **liste** de longueur *n* et on y recherche la présence de la valeur **valeur**. Les éléments de liste sont supposés rangés dans l'ordre croissant.

Principe :

- On détermine l'élément **milieu** de la **liste**
- Si c'est **valeur**, on s'arrête avec succès
- Sinon :
  - si **milieu** est plus grand que **valeur**, on continue à chercher dans la moitié gauche de **liste**
  - si **milieu** est plus petit que **valeur**, on continue à chercher dans la moitié droite de **liste**
- On continue jusqu'à obtenir la valeur recherchée, ou une liste vide, ce qui signifie que la valeur n'est pas présente.

Mise en oeuvre sur un exemple :

On fait la moyenne des indices des valeurs, et non des valeurs elles mêmes, car la moyenne des valeurs ne donne pas à priori une valeur qui est au milieu de la liste.

Recherche de **valeur** = **15** dans **liste** = **[1, 5, 7, 9, 11, 11, 15, 17]**
On a pour la taille des données : **n** = **8**

- L'indice minimum de la recherche est au départ: **ind_min** = **0**
- L'indice maximum de la recherche est au départ: **ind_max** = 8 - 1 = **7**

- On recherche donc 15 entre liste[**0**] = 1 et liste[**7**] = 17 à la première étape
- On détermine l'indice au milieu de la liste: **ind_milieu** = (0 + 7) // 2 = **3**
- Comme liste[3] = 9 et que 9 < 15

- On recherche maintenant 15 entre liste[**3+1**] = 11 et liste[**7**] = 17 à la deuxième étape
- On détermine l'indice au milieu de la liste: **ind_milieu** = (4 + 7) // 2 = **5**
- Comme liste[5] = 11 et que 11 < 15

- On recherche maintenant 15 entre liste[**5+1**] = 15 et liste[**7**] = 17 à la troisième étape
- On détermine l'indice au milieu de la liste: **ind_milieu** = (6 + 7) // 2 = **6**
- Comme liste[6] = 15 **on s'arrête** et **on renvoie la valeur** position **6**.

On voit ici qu'il a suffit de **trois étapes** (trois boucles) pour terminer l'algorithme.

Si l'on avait recherché la présence de **valeur = 17**, comme liste[6] = 15 < 17, il aurait fallu **une étape supplémentaire**, pour finir la recherche entre liste[7] = 17 et liste[7] = 17.

Et si l'on avait recherché la présence de **valeur = 20**, on se retrouve à devoir chercher valeur entre liste[8] et liste[7], ce qui doit être rendu impossible, car alors ind_min > ind_max.

Donc **au maximum 4 étapes** pour répondre dans tous les cas, avec cet exemple de liste de longueur **n = 8**.

En language naturel : on cherche **valeur** dans **Tab**

```
ind_min ← 0
ind_max ← n - 1
tant que ind_mini ⩽ ind_maxi :
        ind_milieu = ( ind_mini + ind_maxi ) // 2
        si Tab [ ind_milieu ] = valeur :
                s'arrêter et renvoyer ind_milieu
        sinon si  Tab [ ind_milieu ] > valeur :
                ind_maxi = ind_milieu - 1
        sinon :
                ind_mini = ind_mileu + 1
renvoyer - 1
```

Code Python :

```python
# recherche dichotomique dans une liste triée
def recherche_dicho(tableau : list, valeur : int) -> int:
    indice_mini = 0
    indice_maxi = len(tableau) - 1
    while indice_mini <= indice_maxi:
        indice_milieu = (indice_mini + indice_maxi) // 2
        if tableau[indice_milieu] == valeur:
            return indice_milieu
        elif tableau[indice_milieu] > valeur:
            indice_maxi = indice_milieu - 1
        else:
            indice_mini = indice_milieu + 1
    return -1
```