Travail sur la complexité : Analyse de tris

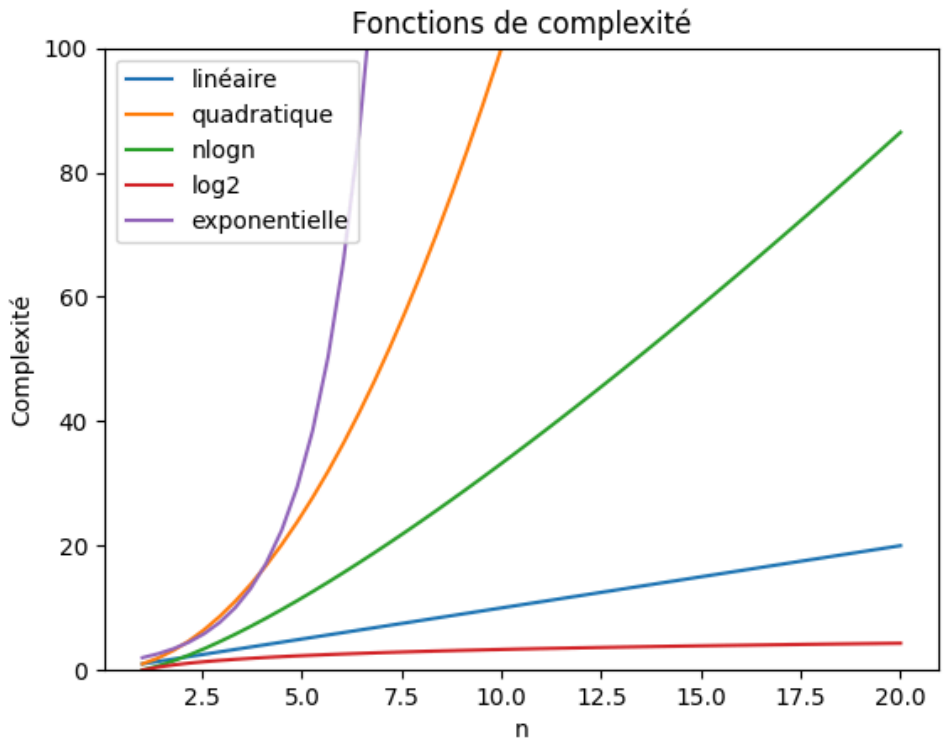
Tri par selection

La compléxité algorithmique est une notion assez abstraite en informatique.

Définition : La complexité d'un algorithme correspond au nombre d'opérations ***élémentaires*** *réalisées au court d'un algorithme* ***en fonction de la taille de la donnée utilisée****.*
La compléxité d'un algorithme peut être associé à une fonction de compléxité:

constante, linéaire, semi-linéaire, logarithmique, quadratique, exponentielle.

Exemple : pour la recherche d'une valeur dans une liste triée en utilisant le principe de dichotomie, la complexité est de l'ordre $O(log_2(n))$.



Cette activité permettra de savoir retrouver la complexité d'un algorithme via son temps d'exécution ou du nombre d'opérations élémentaires.

Vous disposez pour ce faire, un fichier `mesure_tri.py` et un fichier `ap_decorators.py` .
Le fichier `mesure_tri.py` sera votre fichier de rendu contenant chacune des fonctions à rédigier ou à compléter.
Le fichier `ap_decorators.py` quant à lui contient divers modules A NE PAS MODIFIER, permettant de compter certaines opérations.

Fonction de comparaison

Cette activité aura pour but de mesurer le nombre d'opérations élémentaires du tri par selection et du tri par insertion.
Ce nombre de comparaison doit pouvoir être mesuré de manière efficace. Pour cela, on doit se munir d'une fonction de comparaison.

- Écrire une fonction `comparer` qui prend en paramètre deux entiers a et b et renvoie -1 si $a < b$, 1 si $a > b$ ou 0 si $a = b$.

Exemple d'utilisation:

```
>>> comparer(1,2)
-1
>>> comparer(2,1)
1
>>> comparer(2,2)
0
```

Il ne faudra donc pas réaliser de comparaisons directement entre les éléments, mais utiliser la fonction de comparaisons à chaque fois que necessaire. Il faudra donc bien prendre en compte la valeur retournée de l'appel de la fonction de comparaison.

Implémentation du Tri par sélection

1. Expliquer comment fonctionne le tri par selection du minimum.
2. Écrire une fonction `indice_minimum` qui prend en paramètre une liste et un entier correspondant au début d'où l'on recherche le minimum et renvoie l'indice de l'élément minimum. Elle utilisera la fonction `comparer` pour réaliser la comparaison entre éléments.
3. Écrire une fonction `tri_selection_minimum` qui implémante l'algorithme du tri par selection du minimum.
   Elle prendra en entrée une liste d'entiers et modifie par effet de bord la liste pour trier les éléments dans l'ordre croissant.

Mesure du tri par selection du minimum par le temps d'exécution

On va essayer de mesurer le temps d'exécution de l'algorithme de tri par selection du minimum en faisant varier la taille de la liste.
Le but est de tracer une courbe et d'essayer d'approcher la complexité de ce tri.
On utilisera le module `timeit` qui permet de créer des chronomètres et le module `matplotlib` et plus précisément le sous-module `pyplot` pour créer le graphique.
On utilisera en plus le module `random` pour créer des listes aléatoires.

Notre programme de mesure s'organise en plusieurs points :

```python
import random
import timeit
import matplotlib.pyplot as plt

def mesure_tri_selection_temps(n):
    # On initialise un tableau qui servira à récolter nos temps d'exécutions
    tab_mesure = []
    for i in range(n):
        '''On créée une liste de nombre aléatoires de taille i qui variera
        jusqu'à la taille demandée en paramètre.
        '''
        liste_aleatoire = [random.randint(0,500) for _ in range(i)]
        # On prend un temps de départ
        s_time = timeit.default_timer()
        # On trie la liste créée
        ...
        # On prend un temps de fin
        e_time = ...
        # On ajoute dans le tableau de mesures, la différence de temps mesurée entre les deux chronomètres.
        tab_mesure.append(...)

    # On réalise un graphique grâce au tableau de mesure que l'on affiche
    plt.plot(tab_mesure)
    plt.show()
```

1. Compléter le programme ci-dessus pour afficher notre graphique.
2. Parmi les courbes représentatives de fonctions en page 1, laquelle est la plus proche de celle obtenue ? Quelle serait la complexité du tri par selection ?

Mesure du tri par selection du minimum par le nombre de comparaisons

L'utilisation de notre fonction comparer prend son sens ici.
Il existe en Python des décorateurs de fonction.
Un décorateur est une fonction qui permet de réaliser un traitement ou des calculs pour une fonction donnée.

Ici, on va utiliser un décorateur (déjà fourni) `count` qui permet de compter le nombre d'appels à la fonction que l'on souhaite décorer.

Par exemple, on se munit des fonctions suivantes.

```
@count
def comparer(valeur_1, valeur_2):
    ...

def recherche_lineaire(liste,element_recherche):
    present = False
    for elt in liste:
        if comparer(elt, element_recherche) == 0:
            present = False
    return present
```

Si l'on veut compter le nombre de comparaisons, plutot que d'utiliser un compteur, on va utiliser le décorateur `@count` de la fonction comparer et son compteur de comparaison nommé `counter`.

On pourra afficher ce nombre de comparaison de cette manière :

```
l = [1,3,5,15,42]
'''initialiser le compteur à 0'''
comparer.counter = 0
'''utilisation de la fonction qui utilise
les comparaisons comptées grâce au décorateur'''
recherche_lineaire(l,42)
'''affecter à une variable le compteur qui a évolué'''
nombre_comparaisons = comparer.counter
```

1. En vous inspirant de la fonction `mesure_tri_selection_temps`, compléter la fonction de mesure suivante :

```
def mesure_tri_selection_comparaisons(n):
    # On initialise un tableau qui servira à récolter nos temps d'exécutions
    tab_mesure = []
    for i in range(n):
        '''On créée une liste de nombre aléatoires de taille i qui variera
        jusqu'à la taille demandée en paramètre.
        '''
        liste_aleatoire = [random.randint(0,500) for _ in range(i)]
        # On initialise le compteur à 0
        ...
        # On trie la liste créée
        ...
        # On récupère le nombre de comparaisons grâce au compteur
        nombre_comparaisons = ...
        # On ajoute dans le tableau de mesures, le nombre de comparaisons.
        tab_mesure.append(...)

    # On réalise un graphique grâce au tableau de mesure que l'on affiche
    plt.plot(tab_mesure)
    plt.show()
```

2. Parmi les courbes représentatives de fonctions en page 1, laquelle est la plus proche de celle obtenue ? Quelle serait la complexité du tri par selection ?

Tri par insertion

Le complexité du tri par insertion dépend de la taille de l'entrée comme le tri par selection mais dépend aussi d'un autre facteur : la position des éléments dans la liste.
Elle dépend en majorité du nombre de comparaisons et celui-ci dépend du nombre de décalage de l'élément à placer au bon endroit, et ce, pour chaque élément de la liste.

Pour trier la liste [4,1,2,3,5], on doit réaliser 3 décalages :

i	liste	nb déplacements totaux
0	[**4**,1,2,3,5]	0
1	[**1**,**4**,2,3,5]	1
2	[**1**,**2**,**4**,3,5]	2
3	[**1**,**2**,**3**,**4**,5]	3
3	[**1**,**2**,**3**,**4**,**5**]	3

Par contre, pour la liste [4,5,3,2,1], on doit réaliser 8 déplacements.

i	liste	nb_deplacements_totaux
0	[**4**, 5, 3, 2, 1]	0
1	[**4**, 3, 5, 2, 1]	0
2	[**3, 4**, 5, 2, 1]	1
3	[**3, 4**, 2, 5, 1]	2
4	[**3, 2, 4**, 5, 1]	3
5	[**2, 3, 4**, 5, 1]	4
6	[**2, 3, 4**, 1, 5]	5
7	[**2, 3, 1, 4**, 5]	6
8	[**2, 1, 3, 4**, 5]	7
9	[**1, 2, 3, 4**, 5]	8
10	[**1, 2, 3, 4, 5**]	8

On distingue plusieurs cas pour le tri par insertion :

- Le meilleur cas : tous les éléments sont déjà triés, ainsi on ne réalise aucun décalage.
- Le pire cas : les éléments sont triés dans le sens contraire de celui dans lequel on souhaite trier (exemple : valeurs dans l'ordre décroissant et on cherche à trier dans l'ordre croissant).

Il faut donc évaluer les complexités des divers cas de ce tri et cela est possible grâce à l'activité précédente portant sur le tri par selection.
Cette partie ne sera pas guidée : En vous aidant de l'activité précédente, reprendre chacune des questions de l'étude du tri par selection mais les appliquer au tri par insertion et donner une courbe par cas étudié. Vous en déduirez les complexités de chacun des cas.