TP 6. Recherche dichotomique.

Créez, comme d'habitude, un sous-répertoire `TP06` dans votre répertoire personnel `Python` ou `Info`.

Introduction

Dans le cadre de la gestion d'un très grand nombre de données (base de données, gestion de stock, carnet d'adresses...), on peut être amené à faire des recherches régulières sur celles-ci. Pour faire une recherche efficace, une possibilité est de trier les données selon un critère afin de pouvoir effectuer des recherches plus efficaces par la suite.
Dans ce TP, on va comparer la complexité entre une recherche séquentielle et une recherche par dichotomie d'un élément dans une liste en fonction de sa taille.
Au début de votre programme, vous aurez besoin :

— d'importer la bibliothèque random afin de créer une liste aléatoire de taille `n` :

```
import random as r
L=[r.randint(0,500) for i in range(30)] #création d'une liste de 30 entiers
aléatoires compris entre 0 et 500

# pour trier la liste, on utilise une méthode de liste : sort
L.sort()
```

— de plus pour comparer la complexité des deux méthodes, vous utiliserez l'outil graphique afin de représenter le temps de calcul en fonction de la taille de la liste en utilisant la bibliothèque `matplotlib` :

```
import matplotlib.pyplot as plt
```

1 Recherche séquentielle

On commence par réaliser une algorithme séquentiel qui parcourt l'ensemble des valeurs de la liste.

Question 1. Écrire une fonction `recherche_sequentielle(L, val)` prenant en argument une liste `L` d'entiers et une valeur `val` à rechercher dans celle-ci qui renvoie `True` **aussitôt** que la valeur est trouvée et `False` sinon. Quel est son avantage ?

Supposons maintenant que nous souhaitions faire une recherche séquentielle dans une liste triée.
On peut réaliser quelques optimisations, en effet la liste étant triée :

— on peut vérifier si la valeur à chercher n'est pas strictement inférieur au premier terme de la liste,

— on peut vérifier si la valeur à chercher n'est pas strictement supérieure au dernier terme de la liste,

Question 2. Écrire une fonction `recherche_sequentielle_triee(L, val)` prenant en argument une liste **triée** `L` d'entiers et une valeur `val` à rechercher dans celle-ci qui renvoie `True` **aussitôt** que la valeur est trouvée et `False` sinon. Mettre en place les optimisations proposées ci-dessus.

Un élément important en vérification des programmations est d'être capable de choisir un jeu de test permettant de vérifier que l'algorithme fonctionne pour tous les cas et notamment les cas limites.
C'est le rôle des tests dit unitaires réalisés dans tout développement de logiciels.

Question 3. Proposer un jeu de tests qui doit comporter au moins 4 cas. Tester votre programme à l'aide de ces différents tests et le modifier au besoin.

On souhaite déterminer le temps de calcul pour une liste de taille significative : 10 000 000 de valeurs.

```
import time
>>> temps = time.perf_counter() # mesure un temps de référence en secondes
>>> recherche_naive(L,val)
>>> print(time.perf_counter()-temps)
```

Question 4. Écrire les instructions pour mesurer le temps de calcul de la fonction `recherche_sequentielle_triee(L, val)` pour une liste triée L de 10000000 termes dont la valeur cherchée est le dernier terme de la liste : `L[-1]`, comparer le temps obtenu à celui de la recherche interne de python : `valeur in L`. Conclure.

2 Recherche par dichotomie

La liste étant triée, l'idée est de tester si l'élément du milieu de la liste est :

- inférieur strictement à la valeur recherchée, on cherche la valeur dans la demi-liste de gauche ;
- supérieur strictement à la valeur recherchée, on cherche la valeur dans la demi-liste de droite ;
- égale à la valeur recherchée, l'algorithme se termine et renvoie `True`.

On itère jusqu'à ce que la liste de recherche soit vide.

Question 5. Proposer une fonction `recherche_dicho(L:list, val:int) -> bool:` basée sur cette méthode. Vous utiliserez une boucle `while` en réfléchissant bien à la condition d'arrêt.

Question 6. Définir un jeu de tests pour valider votre algorithme.

Question 7. Mesurer le temps de calcul de la fonction et le comparer aux temps précédents. Vous utiliserez la liste créée en **question 4**.

Question 8. Modifier la fonction précédente pour qu'elle renvoie le nombre d'itérations réalisées.

3 Représentation graphique

On souhaite maintenant tracer l'évolution du temps mis par la fonction pour réaliser la recherche en fonction de la taille de la liste.

Question 9. Construire la liste des temps `Les_temps_seq` et la liste `Les_temps_dicho` des fonctions `recherche_sequentielle_triee(L, val)` et `recherche_dicho(L, val)` pour `n` variant de 100 à 1 000 000 avec un pas de 50000.

Question 10. Construire la liste `tailles` des valeurs de n.

Question 11. Représenter sur le même graphique les `Les_temps_seq` et les `Les_temps_dicho` en fonction de la liste `tailles` des `n`, que peut-on conclure ?

Question 12. Représenter graphiquement les `Les_temps_dicho` en fonction de n. Quelle est la forme de la courbe obtenue ?

4 Introduction à la complexité

On souhaite approximer la courbe des itérations effectuées pour une recherche dichotomique par la loi mathématique `log en base 2`.

Pour cela, vous aurez besoin :

— de définir en abscisse une liste de puissances de 2, de longueur 20. Cette liste contient les nombres d'éléments des listes utilisées pour la recherche dichotomique.

```
les_x=[2**i for i in range (5,24)] # création de la liste des d'abscisses
```

— de construire la loi mathématique associée à cette liste.

```
import math as m
les_y=[m.log2(x) for x in range les_x] # liste des ordonnées associée à la fonction
mathématique log2.
```

Question 13. Construire la liste `itérations_dicho` contenant le nombre d'itérations nécessaires pour chercher le dernier élément d'une liste triée de taille définie dans `les_x` avec la méthode de dichotomie.

Question 14. Représenter sur un même graphique le nombre d'itérations en fonction de la taille de la liste utilisée ainsi que la loi mathématique `log2`. Que peut-on conclure ?