TP3 RECHERCHE DICHOTOMIQUE

OBJECTIFS : Les objectifs de ce premier TP sont :

- De rechercher un élément dans une liste rapidement
- d'écrire un algorithme de recherche rapide,
- d'intégrer cet algorithme dans une fonction,
- de comparer le nombres d'itérations (passages dans la boucle) de recherche entre cet algorithme rapide et une recherche naïve.

DOCUMENT A RENDRE : Ce travail est évalué. Vous en rédigerez un compte-rendu au format *.pdf/.odt/.doc/ ;docx* pour le déposer en fin d'activité dans le répertoire *Devoir/TP3* du réseau avec le nom « tp3_nomfamille.pdf ». Ce compte-rendu contiendra :

- les réponses aux différentes questions posées,
- les copies d'écran des morceaux de codes écrits et celles des résultats des exécutions données dans le shell.

Pour commencer, ouvrir dans Pyzo un fichier nommé « recherche_mininmu.py » . Importer les bibliothèques random et time :

```
from random import *
import time
```

1. Recherche naïve dans une liste

- Créer une liste nommée « *petite_liste* » : [-47, -35, -75, -20, 45, 49, 90, -19, 14, 71]

Comment peut-on faire pour déterminer si la valeur 14 est comprise dans la liste ?

Combien de comparaisons seront faites au maximum si la valeur recherchée n'appartient pas à la liste ?

On désire maintenant que l'algorithme de recherche renvoie soit l'indice de la valeur recherchée soit -1 pour indiquer l'absence de la valeur dans la liste. Vous implémenterez le code d'une fonction « recherche naïve » qui corresponde à ce cahier des charges.

Appliquer recherche_naive pour les valeurs de 14 et 80

2. Recherche d'une méthode plus efficace.

Actuellement la liste contient des valeurs rangées dans un ordre aléatoire qui ne permet pas d'élaborer une méthode de recherche autre qu'une comparaison systématique de tous les éléments jusqu'à trouver la valeur recherchée ou pas dans un sens ou un autre.

Pour diminuer le temps de recherche il serait intéressant de ranger les éléments dans un ordre strict croissant ou décroissant.

- Rappeler la méthode associée aux listes qui effectue ce travail avec la plus grande efficacité.
- Appliquer cette méthode à petite liste.

- Afficher petite_liste triée.

- Si on reprend la recherche naïve de combien de comparaison a-t-on besoin pour trouver « 49 » et quel son indice.

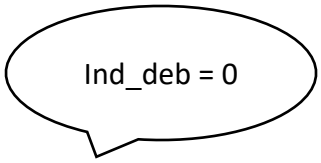
- Si on prend au hasard l'indice 2 dans quel sens sera-t-il plus judicieux de chercher ?

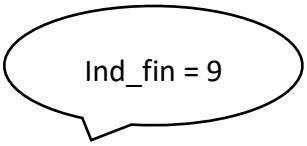
- Si on prend au hasard l'indice 8 dans quel sens sera-t-il plus judicieux de chercher ?

- De la même manière pour optimiser les chances de trouver la valeur recherchée quel est l'indice à voir en premier. Comment peut-on de calculer

On divise le tableau en deux parties en calculant l'indice milieu. Si la valeur recherchée correspond à celle de l'indice milieu on retourne cet indice. Dans le cas où la valeur de l'indice est supérieure à la valeur recherchée on s'occupera de la partie supérieure du tableau sinon de la partie inférieure. Et on recommence tant que l'on n'obtient pas le rang de l'élément recherché ou que la méthode ne puisse plus s'appliquée.

recherche de la valeur 49





-75	-47	-35	-20	-19	14	45	49	71	90

Calcul de l'indice milieu : m = partie entière (deb+fin)/2

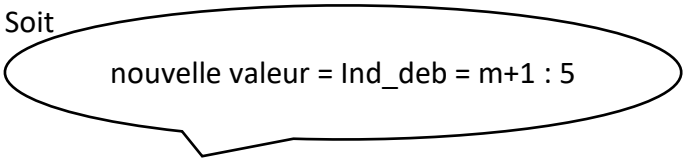
Valeur de m = (9-0)//2 =4

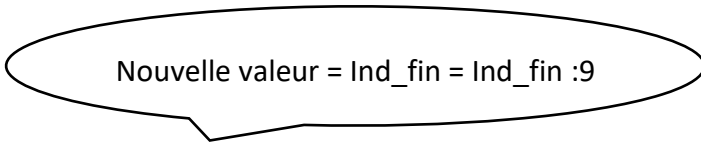
petite_liste [4] = -19 et -19 <14

Donc on prend la partie haute du tableau sans la valeur correspondant à l'indice milieu calculé précédemment :

45	49	71	90

Soit





45	49	71	90

- Calcul de l'indice milieu : m = partie entière (deb+fin)/2

- Quelle serait une condition qui indiquerai que cette méthode n'a pas trouvé le résultat si on examine les indices du tableau :

Cette condition est appelée in invariant de boucle :
https://fr.wikipedia.org/wiki/Invariant_de_boucle

- A contrario quelle relation doit-elle être vérifiée pour continuer la méthode

- Quelle structure de boucle est la mieux adaptée pour autoriser une itération supplémentaire

- Indiquer les formules permettant de calculer les nouveaux indices de fin « fin » et de début « deb » par rapport à l'indice milieu m calculé précédemment dans le cas où la relation précédente est vérifiée

Dans le cas où la valeur recherchée est supérieure à celle de l'indice milieu :

Dans le cas où la valeur recherchée est inférieure à celle de l'indice milieu :

Compléter le tableau suivant pour la recherche de la valeur -35

	Itération 1	Itération 2	Itération 3
Indice Deb			
Indice Fin			
Fin – Deb >= 0	V		
m			
Tab[m]			
Tab[m] == -35	F		

Compléter le tableau suivant pour la recherche de la valeur 60

	Itération 1	Itération 2	Itération 2	Itération 3	Itération 4
Indice Deb					
Indice Fin					
Fin – Deb >= 0	V				
m					
Tab[m]					
Tab[m] == -35	F				

- Implémenter l'algorithme de recherche qui renvoira soit l'indice de la valeur recherchée soit -1 si elle n'est pas présente dans la liste déjà triée.

- Créer une fonction recherche_dichotomique qui automatise cette méthode de recherche.

3. Comparaison du nombre d'itération (passages dans la boucle) d'exécution

Dans recherche_naive et recherche_dichotomique ajouter un compteur pour le nombre d'itération/séquence effectuées dans le cas où la valeur recherchée n'est pas comprise dans petite liste.

Créer une liste avec n valeurs aléatoires de 0 à100 puis trier la avec la bonne méthode et faites et tester les nombre d'itération entre les deux méthodes de tris pour n = 10, 100 ,1000,10000,100000. Compléter le tableau :

Recherche/nombre itérations	10	100	1000	10 000	100 000
naïve					
Dichotomique					

4. Notion de complexité

La complexité (temporelle) d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme. Ce nombre s'exprime en fonction de la taille *n* des données. On s'intéresse au coût exact quand c'est possible, mais également au coût moyen (que se passe-t-il si on moyenne sur toutes les exécutions du programme sur des données de taille *n*), au cas le plus favorable, ou bien au cas le pire.

- Pour la recherche naïve on parcours entièrement tout la liste. Le nombre d'opérations de comparaison est donc dans le pire des cas égale au nombre d'éléments de la liste soit n

On écrit cette complexité est donc linéaire avec la notation suivante : $O(n)$

- Pour la recherche dichotomique on s'intéresse à la boucle tant que dans le cas le plus défavorable.

  Sachant qu'à chaque itération de la boucle on divise le tableau en 2, cela revient donc à se demander combien de fois
  faut-il diviser le tableau en 2 pour obtenir, à la fin, un tableau comportant un seul entier ? Autrement dit, combien de fois faut-il diviser n par 2 pour obtenir 1 ? Mathématiquement cela se traduit par l'équation $\frac{n}{2^a} = 1$ avec a le nombre de fois qu'il faut diviser n par 2 pour obtenir 1.

- Donner par approximation successives la valeur de a pour n = 100

$$\frac{100}{2^5} = 3.125 \quad \frac{100}{2^6} = 1.5625 \quad \frac{100}{2^7} = 0.78125$$

7 parait convenir

Pour systématiser ce calcul on introduit le logaritme de base 2 soit

$$\frac{n}{2^a} = 1 \; soit \; n = 2^a \; soit \; log_2(n) = log_2(2^a)$$

$$et\ enfin\ a\ =\ log_2(n) = \frac{\ln(n)}{\ln(2)}$$

- Dans ce cas la complexité est en $O(\log(n))$

- Vérifier pour les valeurs de n des différentes listes pour la recherche dichotomique

nombre d'éléments	10	100	1000	10 000	100 000
Dichotomique					
Formule					

5. Rapidité

On va comparer la recherche naïve sur une liste non triée avec la recherche dichotomique sur la même liste mais après l'avoir triée par la méthode sort(). Donc on va comparer :

temps recherche naïve == ? temps sort() + recherche dichotomique

Employer de nouveau les mêmes listes de 10 ,100,1 000, 10 000 et 100 éléments aléatoires pour mesurer le temps avant et après chacune des deux méthodes.

nombre d'éléments/temps	10 000	100 0000	1 000 000	10 000 000	10 0 000 000
Naïve					
Dichotomique					

Conclusion