Travaux pratiques 3

Recherche par dichotomie

Informatique tronc commun MPSI et PCSI

I Présentation

Effectuer une recherche dans une liste est une opération d'utilité courante. Par exemple avec une liste de nombres, on peut vouloir déterminer si un nombre est dans la liste, ou trouver l'encadrement le plus serré d'un nombre avec deux éléments de la liste.
En prévision des besoins de ce TP, ajoutez les importations suivantes en tête de votre code source :

```
import math
import time
import random as rd
import matplotlib.pyplot as plt
```

Afin de tester diverses méthodes et d'en comparer les performances, vous utiliserez la bibliothèque `time` avec le modèle de code suivant :

```
début = time.time()
# Code dont on veut mesurer le temps d'exécution
fin = time.time()
durée = fin-début
```

Ainsi, `durée` sera une estimation du temps d'exécution en secondes.
Effectuer de telles mesures sur des listes de quelques éléments, ou de quelques dizaines d'éléments, conduira à des valeurs non significatives. Pour générer facilement des listes plus longues vous pourrez utiliser le modèle de code suivant :

```
liste = rd.sample(range(5*n), n)
```

qui va créer une liste de `n` éléments dont les valeurs sont aléatoires entre 0 et `5*n-1`, de sorte qu'une valeur apparaît une fois au plus. Le choix de `5*n` est arbitraire et permet de s'assurer que la liste ne contient pas toutes les valeurs possibles.
Enfin, si on a besoin de trier une liste :

```
liste.sort()
```

va trier la variable `liste` en place dans l'ordre croissant.[1]

À retenir

On rappelle que la complexité d'un algorithme sera évaluée comme le nombre de tours de boucle dans le « pire des cas ».

[1] De ce fait, `sort` ne renvoie rien et il est inutile d'affecter sa valeur à une variable.

II Recherche linéaire

La recherche linéaire est un algorithme générique qui peut être utilisé avec n'importe quelle liste. Pour savoir s'il existe dans la liste un élément égal à `x`, on parcourt la liste dans l'ordre :

- Si on trouve un élément égal à `x`, la réponse est oui et on arrête immédiatement la recherche.
- Si on arrive au bout de la liste sans avoir rien trouvé, la réponse est non.

Exercice 1 - Recherche linéaire
Écrivez une fonction `recherche_linéaire` prenant en entrée une liste d'entiers et un nombre entier, et renvoyant en sortie `True` si le nombre est dans la liste, `False` sinon, en utilisant cet algorithme.

Indication : bouclez directement sur les éléments de la liste, puisque la connaissance de leur position est inutile.

Exercice 2 - Question annexe
Pourquoi est-il nécessaire que les nombres soient ici de type `int` ?

On veut maintenant évaluer son temps d'exécution en fonction de la longueur n de la liste.

Exercice 3 - Mesure pour une liste
Écrivez une fonction `mesure1_linéaire` qui prend en entrée le nombre d'éléments de la liste et l'élément à chercher.
Cette fonction doit générer aléatoirement la liste, effectuer la recherche linéaire en mesurant son temps d'exécution, et renvoyer ce temps.

Exercice 4 - Tracé du temps moyen d'exécution
Ajoutez le code ci-dessous à votre programme. Pour diverses valeurs de `n`, il appelle 10 fois `mesure1_linéaire` et calcule la moyenne des temps d'exécution renvoyés. Puis il trace le graphe de ce temps moyen en fonction de `n`, en échelles logarithmiques.

```
tirages = 10
liste_n = [2**k for k in range(1, 6)]
liste_temps = []

for n in liste_n:
    temps = 0
    for _ in range(tirages):
        temps += mesure1_linéaire(n, 1)
    liste_temps.append(temps/tirages)

plt.scatter(liste_n, liste_temps, label="Recherche linéaire")
plt.xlabel("n")
plt.ylabel("durée moyenne")
plt.xscale("log")
plt.yscale("log")
plt.legend()
plt.show()
```

Concluez sur la complexité de cet algorithme de recherche linéaire.

III Recherche dichotomique

III.1 Principe

Alice[2] pense à un entier x entre 0 et 100, disons 77, et Bob doit le deviner. Il peut seulement proposer des valeurs, et Alice répond en précisant si le nombre est plus grand ou plus petit.
Bob choisit de toujours proposer la valeur centrale de l'intervalle étudié.

Valeur proposée	Réponse d'Alice	Conclusion de Bob
50	trop petit	$x \in [\![51, 100]\!]$
75	trop petit	$x \in [\![76, 100]\!]$
88	trop grand	$x \in [\![76, 87]\!]$
81	trop grand	$x \in [\![76, 80]\!]$
78	trop grand	$x \in [\![76, 77]\!]$
76	trop petit	$x = 77$

Le mot *dichotomie* signifie « couper en deux » : on découpe l'intervalle de recherche, et grâce au fait que les nombres de l'intervalle $[\![0, 100]\!]$ sont ordonnés, Bob a pu conclure en 6 essais, là où une recherche linéaire en aurait nécessité… 78.

III.2 Implémentation

On se limite à une liste d'entiers de longueur n ordonnée dans le sens croissant, le plus petit élément étant en position 0 et le plus grand en position $n - 1$. Le principe de l'algorithme est le suivant :

1. On étudie la liste constituée des éléments entre les positions $a = 0$ et $b = n - 1$. On calcule alors la position de l'élément central.
2. Par comparaison, on détermine si l'élément central est égal à la valeur cherchée ou, à défaut, si la valeur cherchée peut être dans le sous-intervalle de gauche ou de droite.
3. On redéfinit a ou b pour restreindre l'intervalle étudié lors de l'itération suivante.
4. On s'arrête quand on trouve l'élément cherché, ou quand l'intervalle étudié devient vide.

Exercice 5 - Recherche dichotomique
Écrivez une fonction `recherche_dichotomique` ayant la même interface que `recherche_linéaire` (mêmes entrées et même sortie), mais procédant par dichotomie.
Pour tester la fonction, on rappelle que la liste doit être préalablement triée.

III.3 Performances comparées

À retenir

Pour placer plusieurs courbes ou nuages de points sur le même graphe, il suffit que les appels à `plt.plot` ou `plt.scatter` soient placés avant l'appel à `plt.show`.
À l'inverse, pour qu'ils soient sur des graphes séparés, il suffit que chacun ait son propre appel à `plt.show`.

[2] Il est d'usage international, en informatique, d'appeler « Alice » et « Bob » les deux personnes intervenant dans une communication, ce qui est un peu plus convivial que « personne A » et « personne B ».

Exercice 6 - Comparaison graphique
Reprenez le code de l'exercice 4 et ajoutez-y le nécessaire pour pouvoir placer sur le même graphe (log-log) les temps moyens d'exécution des deux algorithmes.
Commentez.

Exercice 7 - Performances relatives
Évaluez numériquement le facteur gagné en performances pour $n = 10^5$.

Exercice 8 - Complexité théorique
Justifiez théoriquement que la complexité de la recherche dichotomique est $O(\log_2(n))$.

IV Encadrement d'un flottant

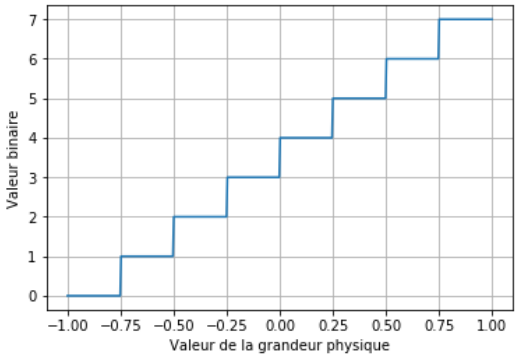
Prenons par exemple la liste triée `[2, 4, 6, 9]`. L'encadrement le plus serré de `5.2` par deux éléments de la liste est entre `4` et `6`.
On pourrait bien sûr parcourir la liste linéairement à la recherche du premier élément qui dépasse `5.2`, mais par dichotomie on va pouvoir faire mieux.

Exercice 9 - Encadrement par dichotomie
Donnez-vous une liste d'entiers tous différents (une dizaine d'éléments suffit) et triés dans l'ordre croissant. Soit a le plus petit élément et b le plus grand.
Donnez-vous un flottant dans $]a, b[$.
Écrivez une fonction `encadrement_dichotomique` qui prend en entrée la liste d'entiers et le flottant et qui renvoie les indices de position des deux éléments consécutifs encadrant le flottant. Si le flottant n'est pas encadrable dans cette liste, la fonction doit renvoyer `-1,-1`.
La fonction devra bien sûr procéder par dichotomie.

Remarque : renvoyer un résultat manifestement « absurde » (ici `-1, -1`) en cas de problème est peu élégant… Python permet de traiter ça plus proprement, mais ce n'est pas le sujet ici.

Exercice 10 - Question annexe
Est-il encore important que la liste soit constituée d'entiers ?

Une application importante est la numérisation d'une donnée, opération typiquement menée avant un traitement informatique.
Prenons un voltmètre numérique sait mesurer une tension dans l'intervalle $[-1, 1]$ V. Il travaille sur 3 bits, ce qui signifie qu'une tension, une fois traitée pour être stockée en mémoire, ne peut avoir que $2^3 = 8$ valeurs distinctes, représentées par les entiers de 0 à 7. La correspondance est donnée par le graphe :



Par exemple, une tension mesurée de 0,32 V sera stockée en mémoire sous la forme de l'entier 5.

Exercice 11 - Numérisation
Générez la liste des valeurs de tension correspondant à cet exemple, et utilisez la fonction `encadrement_dichotomique` sur quelques exemples.

V Complément : et `in`, dans tout ça ?

Python fournit l'opérateur **`in`**, qui permet d'écrire des clauses booléennes comme :

```
présence = 3 in [1, 7, -3, 0, 2]
```

ici `présence` vaut `False` après exécution car 3 n'est pas dans la liste.
C'est évidemment bien plus concis et agréable à lire que les algorithmes développés ci-dessus, mais que fait cet opérateur ?

Exercice 12 - Performances de `in`
À l'aide du code déjà écrit, comparez le temps moyen d'exécution de **`in`** à ceux de `recherche_linéaire` et `recherche_dichotomique`. Conclusion ?

VI Exercice bonus : recherche d'un maximum dans une liste

On considère une liste d'entiers telle que :

- Tous ses éléments sont uniques (pas de doublon).
- Ces éléments sont d'abord croissants, puis décroissants (un seul changement de monotonie).

Ceci assure l'existence et, surtout, l'unicité d'un élément maximal. Dans ce cas, on peut rechercher cet élément par une méthode inspirée à la dichotomie : la *trichotomie*. Le principe est le suivant.

1. On découpe la liste en trois sous-listes de longueurs égales (éventuellement à un élément près) en calculant les positions `m1` et `m2` des deux éléments situés aux frontières.
2. On distingue alors trois cas :
   - Si `liste[m1]` est strictement supérieur à `liste[m2]`, alors le maximum ne peut pas se trouver dans la sous-liste de gauche.
   - Si `liste[m1]` est strictement inférieur à `liste[m2]`, alors le maximum ne peut pas se trouver dans la sous-liste de droite.
   - Si les deux éléments sont égaux, alors le maximum est dans la sous-liste du milieu.

   On réduit ainsi la liste considérée pour la recherche.
3. Par itérations successives, on réduit la taille de la liste jusqu'à ce qu'elle ne contienne plus qu'un élément qui est le maximum cherché.

Pour implémenter cet algorithme, demandez-vous en particulier si vous pouvez exclure les éléments de position `m1` et/ou `m2` à chaque itération, comme vous l'avez fait pour l'élément central avec la dichotomie.

Exercice 13 - Création d'une liste croissante puis décroissante
Créez une liste conforme aux contraintes posées ci-dessus, par exemple en concaténant deux listes, une croissante et une décroissante, produites par deux appels bien choisi à **`range`**. Une douzaine d'éléments suffit.

Exercice 14
Écrivez une fonction qui reçoit une liste du type ci-dessus et renvoie la valeur de l'élément maximal en procédant par trichotomie.
Ne cherchez pas forcément à optimiser votre code tout de suite. Vous pouvez par exemple proposer d'abord une version qui réécrit la liste à chaque tour de boucle et/ou utilise du slicing, bien que ce ne soit pas souhaitable dans l'absolu.

VII Solutions

Solution de l'exercice 1 - Recherche linéaire

```
def recherche_linéaire(liste, nombre):
    for e in liste:
        if e == nombre:
            return True
    return False
```

Solution de l'exercice 2 - Question annexe

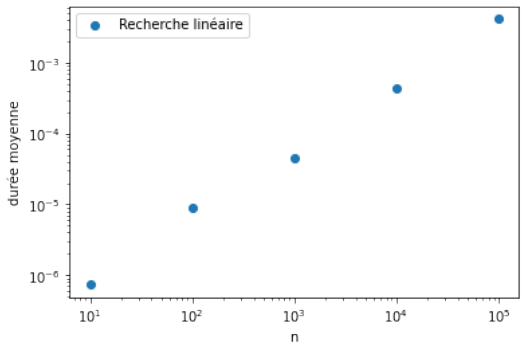
Le test d'égalité entre deux flottants renvoie un résultat indéterminé à cause de la manière dont les flottants sont représentés en mémoire, avec une précision limitée. Donc la valeur renvoyée par la fonction n'aurait aucun intérêt.

Solution de l'exercice 3 - Mesure pour une liste

```
def mesure1_linéaire(valeurs, nombre):
    liste = rd.sample(range(5*valeurs), valeurs)
    début = time.time()
    recherche_linéaire(liste, nombre)
    fin = time.time()
    return fin-début
```

Solution de l'exercice 4 - Tracé du temps moyen d'exécution

On trouve une figure comme celle-ci :



sachant qu'il y a une certaine variabilité d'une exécution à l'autre et d'une machine à l'autre.
Avec les valeurs proposées, il est très probable que l'élément cherché n'est pas dans la liste, et donc qu'on évalue la complexité « au pire ».
Le graphe ayant une allure à peu près linéaire avec une pente de 1, on peut conjecturer que cette complexité est linéaire. Ceci était prévisible puisque la liste est parcourue linéairement.

Solution de l'exercice 5 - Recherche dichotomique

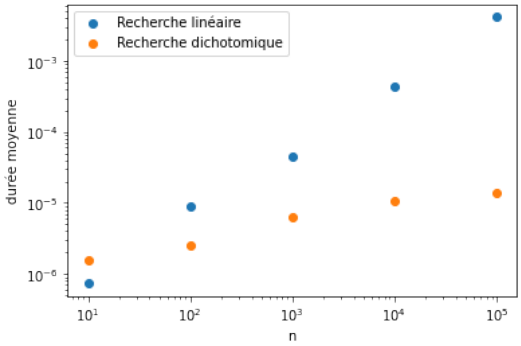
L'algorithme conduit naturellement vers une boucle **`while`**. Lors de la détermination de la position de l'élément milieu, il faut utiliser la division euclidienne pour obtenir un entier.
Si l'élément milieu n'est pas égal à la valeur cherchée, on peut définir le sous-intervalle à garder en excluant cet élément milieu.

```
def recherche_dichotomique(liste, nombre):
    a = 0
    b = len(liste)-1
    while a <= b:
        moy = (a+b)//2
        if liste[moy] == nombre:
            return True
        elif liste[moy] < nombre:
            a = moy+1
        else:
            b = moy-1
    return False
```

Solution de l'exercice 6 - Comparaison graphique
Les fonctions `recherche_linéaire`, `mesure1_linéaire`, `recherche_dichotomique` sont présentes mais n'ont pas été recopiées dans le code ci-dessous

```
def mesure1_dichotomique(valeurs, nombre):
    liste = rd.sample(range(5*valeurs), valeurs)
    liste.sort()
    début = time.time()
    recherche_dichotomique(liste, nombre)
    fin = time.time()
    return fin-début

tirages = 10
liste_n = [10**k for k in range(1, 6)]
liste_t_linéaire = []
liste_t_dichotomique = []

for n in liste_n:
    temps_linéaire = 0
    temps_dichotomique = 0
    for _ in range(tirages):
        temps_linéaire += mesure1_linéaire(n, 1)
        temps_dichotomique += mesure1_dichotomique(n, 1)
    liste_t_linéaire.append(temps_linéaire/tirages)
    liste_t_dichotomique.append(temps_dichotomique/tirages)

plt.scatter(liste_n, liste_t_linéaire, label="Recherche linéaire")
plt.scatter(liste_n, liste_t_dichotomique, label="Recherche
   dichotomique")
plt.xlabel("n")
plt.ylabel("durée moyenne")
plt.xscale("log")
plt.yscale("log")
plt.legend()
plt.show()
```

Ce code produit la figure ci-dessous. On en conclut, empiriquement, que la complexité de la recherche dichotomique est bien plus basse, et que plus que n augmente, plus cet algorithme est intéressant comparé à la recherche linéaire.



Solution de l'exercice 7 - Performances relatives
On utilise les deux listes `liste_t_linéaire` et `liste_t_dichotomique` : le rapport de leurs derniers éléments vaut environ 300. La différence de performance devient rapidement spectaculaire !

Solution de l'exercice 8 - Complexité théorique
À chaque étape, on divise par 2 la longueur de la liste étudiée donc, au k^{e} tour de boucle, la longueur restante est inférieure ou égale à $n/2^k$. Donc quand l'algorithme se termine la longueur restante est inférieure ou égale à 1 :

$$\frac{n}{2^k} \leqslant 1 \quad \Rightarrow \quad k \geqslant \log_2(n)$$

Solution de l'exercice 9 - Encadrement par dichotomie
On peut recycler l'essentiel de la fonction `recherche_dichotomique`, à deux détails près :

- Il faut garder l'élément central lors de la réduction de l'intervalle étudié pour le tour de boucle suivant.
- La condition d'arrêt intervient quand l'intervalle étudié est de largeur 1.

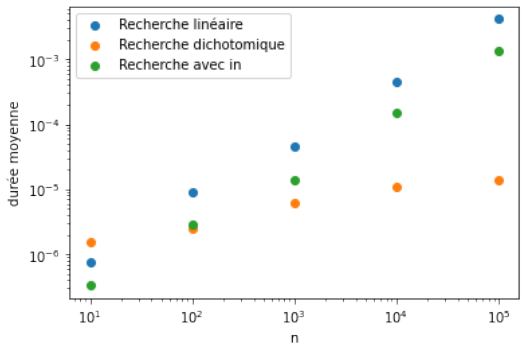
```
def encadrement_dichotomique(liste, nombre):
    a = 0
    b = len(liste)-1
    while a <= b:
        if b-a == 1:
            return a, b
        else:
            moy = (a+b)//2
            if liste[moy] < nombre:
                a = moy
            else:
                b = moy
    return -1, -1

liste = [9, 14, 22, 24, 28, 35, 40, 42, 44, 45]
x = 43.3
a, b = encadrement_dichotomique(liste, x)
print("{} est entre {} et {}".format(x, liste[a], liste[b]))
```

Solution de l'exercice 10 - Question annexe
Comme la fonction ne fait plus aucun test d'égalité avec un élément de la liste, il n'est plus nécessaire de se limiter à des listes d'entiers.

Solution de l'exercice 11 - Numérisation
Comme expliqué à l'exercice précédent, on peut utiliser `encadrement_dichotomique` sans modification avec une liste de flottants.
Le résultat cherché est le minorant de l'encadrement donc :

```
bits = 3
Vmin = -1
Vmax = 1
pas = (Vmax-Vmin)/2**bits
tensions = [Vmin+k*pas for k in range(2**bits+1)]
u = 0.32
a, _ = encadrement_dichotomique(tensions, u)
print("{} est associé à l'entier {}".format(u, a))
```

Solution de l'exercice 12 - Performances de `in`
On obtient :



`in` semble être proche d'une complexité linéaire. Ça n'a rien d'étonnant : cet opérateur peut être utilisé pour une grande diversité de types composites, y compris pour des types non numériques ou non ordonnables, donc il doit être le plus générique possible.
La dichotomie est plus efficace… quand elle est applicable. Elle nécessite un type composite où les éléments ont une position définie et peuvent être ordonnés selon une relation d'ordre. Situation courante mais pas universelle !

Solution de l'exercice 13 - Création d'une liste croissante puis décroissante
Par exemple : `liste = [k for k in range(3, 12)]+[k for k in range(17, 6, -2)]`

Solution de l'exercice 14
Les réponses peuvent varier selon que les deux éléments choisis comme « frontières » sont considérés comme rattachés à la sous-liste sur leur gauche ou sur leur droite.
Voici une première version qui écrase la liste à chaque tour de boucle :

```
def recherche_maximum_redef(liste):
    while len(liste) > 1:
        m1 = len(liste)//3
        m2 = 2*m1 if m1 > 0 else 1
        if liste[m1] < liste[m2]:
            liste = liste[m1+1:]
        elif liste[m1] > liste[m2]:
            liste = liste[:m2]
        else:
            liste = [liste[m1+1:m2]]
    return liste[0]
```

Petite subtilité avec la création de la deuxième frontière `m2`, pour éviter le cas où `m1` et `m2` seraient nuls tous les deux, ce qui ferait partir le **`while`** en boucle infinie.
Deuxième version qui travaille plus proprement en laissant la liste inchangée :

```
def recherche_maximum(liste):
    # a et b sont les indices de position des extrémités
    # de l'intervalle considéré (incluses)
    a = 0
    b = len(liste)-1
    while b-a+1 > 1:
        c = (b-a+1)//3
        c2 = 2*c if c > 0 else 1
        m1 = a+c
        m2 = a+c2
        if liste[m1] < liste[m2]:
            a = m1+1
        elif liste[m1] > liste[m2]:
            b = m2-1
        else:
            a = m1
            b = m2
    return liste[a]
```