TP de Python numéro 5 : Recherche dichotomique dans une liste triée.

*Semaine du **jeudi 28/11**.*

Dans ce TP, on continue le travail sur les listes. On donne un autre algorithme de tri, puis on s'intéresse à des algorithmes sur les listes triées qui permettent de gagner en efficacité. On va voir une première incarnation d'une méthode qui s'applique dans des situations variées, appelée la dichotomie.

I. Deux méthodes simples de tri

Dans cette partie, on va écrire des algorithmes assez simples permettant de trier une liste donnée, en modifiant la liste en entrée.

1. Tri par sélection du minimum

Cet algorithme est sûrement le plus simple pour trier une liste :

- On parcourt la liste pour trouver son plus petit élément,
- on échange cet élément avec le premier élément de la liste,
- on ne touche plus au premier élément de la liste, et on recommence à partir de la première étape, avec la sous liste obtenue en oubliant le premier élément.

Exemple :

Si on applique cet algorithme à la liste `[2,5,1,4,6,2,13]`, elle est modifiée étape par étape de la manière suivante :

- On met le plus petit élément au début à l'aide un échange : la liste devient `[1,5,2,4,6,2,13]`
- On recommence sans considérer le premier élément, donc seulement à partir de la flèche : `[1,5,2,4,6,2,13]`
  ↑
  La liste devient `[1,2,5,4,6,2,13]`
- On recommence sans considérer les deux premiers éléments, donc à partir de la flèche : `[1,2,5,4,6,2,13]`
  ↑
  La liste devient : `[1,2,2,4,6,5,13]`
- et ainsi de suite : `[1,2,2,4,6,5,13]` (*"4 est échangé avec lui-même".*)
- `[1,2,2,4,5,6,13]`
- (l'algorithme tourne encore 2 coups pour finir, mais plus rien ne se passe).

Exercice 1. Appliquer cet algorithme, à la main, sur la liste `[5,4,6,23,2,1,6]`

Exercice 2. Compléter le code `Python` ci-dessous afin d'implémenter le tri par sélection du minimum (remplacer les `...`), puis testez le sur des listes de votre choix.

```
def tri_minimum(L):
    for i in range(len(L)): #i représente la position de la flèche
        # Recherche du minimum de la liste à partir du i-ième
        min = ...
        for j in range(i,len(L)): #Boucle pour rechercher le minimum
            if L[j]<L[min]:
                ...
        # fin de boucle : min est la position du minimum, on échange
        ... , ... = ... , ...
    # fin de l'algorithme : L est triée
    return L
```

Remarque : on n'était pas forcé de mettre la ligne `return(L)`. Si on ne la met pas, la liste se fait quand même trier, mais la fonction ne renvoie pas la liste `L`.

2. Le tri à bulles

Le tri à bulles est un autre algorithme de tri. Le principe est assez simple :

- On parcourt les éléments de la liste du premier à l'avant dernier, et on les compare à leur successeur. S'ils ne sont pas dans l'ordre croissant, on les échange. Cette étape sera appelée un **parcours**.
- On répète assez de fois l'opération ci-dessus pour que la liste soit triée.

Exercice 3. Effectuer à la main le tri à bulles sur la liste `[2,1,4,3,8,1]`. Puis, écrire une fonction `tri_bulle` en s'aidant du code à trous ci-dessous (rajouter des commentaires). Puis, tester votre code sur des listes.

```
def tri_bulle(L):
    desordre=True
    # desordre contient True tant que la liste n'est pas triée.
    # A priori, la liste n'est pas triée.

    #Tant que la liste n'est pas triée, on effectue des parcours.
    while desordre:
        # Parcours avec détection de désordre
        desordre = False
        for i in range( ... ):
            if  ...   :
                desordre=True
                ...
```

Remarque. On peut montrer (théorème) qu'une liste de longueur n sera toujours triée au bout de $n-1$ parcours. Dans le code précédent, on a continué les parcours jusqu'à avoir une liste triée.

Exercice 4. Implémenter le tri à bulle sans variable `desordre`, en appliquant le théorème mentionné dans la remarque ci-dessus : si `L` est de longueur n, alors le tri à bulle trie la liste `L` après $n-1$ répétitions des parcours de `L`.

Remarque. Cette seconde version du tri à bulles est assez simple, et doit être connue.

II. Algorithmes dichotomiques

Le mot *dichotomie* vient du grec : *tomós* signifie "section" ou "coupure", *dikha* signifie "en deux". Le principe de dichotomie en mathématiques, c'est le principe de couper en deux une "zone de recherche". Nous retrouverons ce principe dans des TPs ultérieurs, et dans le chapitre sur la continuité pour démontrer le théorème des valeurs intermédiaires.

1. Recherche d'éléments

Recherche d'un élément dans une liste (rappel)

La commande Python `a in L` teste si un élément `a` appartient à une liste `L`, et renvoie `True` si c'est le cas, `False` sinon.

Le problème auquel réponds la recherche dichotomique est que le temps d'exécution de cette commande peut être très long si la liste `L` est grande.

Exercice 5.

1. Définir la liste `L7` des nombres inférieurs à dix millions, et la liste `L8` des nombres inférieurs à cent millions. Que remarquez vous sur le comportement de l'ordinateur quand on définit `L8` ?
2. Définir la liste `C8` des carrés des nombres inférieurs à cent millions. Que remarquer ?

Un petit exercice déjà fait pour vous rafraichir la mémoire.

Exercice 6. Coder une fonction `appartient` prenant en entrée un argument `a` et une liste `L` et renvoyant `True` si `a` est un élément de la liste `L`, et `False` sinon. Ne pas utiliser la commande `in` pour coder `appartient`.

Dans l'invite de commande, tester la fonction `appartient`, et remarquer les temps d'exécutions, avec les listes de l'exercice précédent :

```
a=10**14
b=10**16
appartient(a,L8)
appartient(b,L8)
appartient(a,C8)
appartient(b,C8)
```

Si on avait fait les mêmes tests avec un milliard au lieu de cent millions, il aurait fallu attendre un certain temps (essayez si vous voulez). Quand on traite un grand nombre de données (ce qui arrive souvent en analyse de données, un domaine prenant de plus en plus d'importance de nos jours), cette attente a un impacte sérieux sur le déroulement du reste des opérations. Pour décrire cela, on a une notion de **complexité** d'un programme informatique, qui correspond au nombre d'opérations qu'un programme donné effectue pour arriver à ses fins.

La dichotomie est un principe général qu'on retrouve dans pas mal d'algorithmes particulièrement efficaces, mais aussi dans des démonstrations mathématiques.

2. Recherche dichotomique d'un élément dans une liste triée

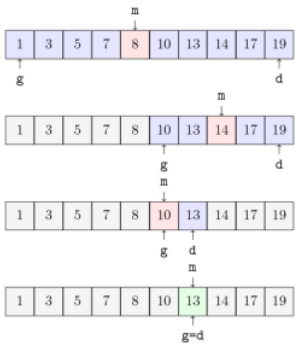
On va appliquer le principe de dichotomie pour écrire une fonction `recherche_dicho` prenant en entrée une liste de nombres `L`, **triée dans l'ordre croissant**, et un nombre `a`, et renvoyant `True` si `a` est un élément de `L`, et `False` sinon.

La recherche dichotomique d'un élément dans une liste triée

L'idée est de couper successivement la zone de recherche en deux, en utilisant que la `L` est triée. Voici l'algorithme :

1. A priori, on cherche l'élément `a` entre les indices `g=0` et `d = len(L)-1`.
2. Tant que la zone de recherche entre `g` et `d` contient des éléments, on suit les étapes 3 et 4. Sinon, on va a l'étape 5.
3. On regarde l'indice `m` au milieu de `g` et `d` (si la longueur de `L[g:d+1]` est paire, on choisit une des deux valeurs centrales, ça n'a pas d'importance).
4. On compare `L[m]` et `a`.
   - Si `L[m]==a`, on s'arrête et on renvoie `True`.
   - Sinon, si `L[m]<a`, on cherche dans la sous liste à droite de `m` car `L` est triée. On reprends à l'étape 1 avec `g=m+1` sans changer `d`
   - Sinon, `L[m]>a` : on cherche dans la sous liste à gauche de `m` : on reprends à l'étape 1 avec `d=m-1` sans changer `g`.
5. Si, lors de notre recherche, la zone de recherche ne contient plus d'éléments, c'est que `a` n'est pas dans `L`.

Par exemple, voici l'illustration de l'algorithme pour la recherche de 13 dans la liste `[1,3,5,7,8,10,13,14,17,19]`.



Exercice 7. Écrire le code d'une fonction `recherche_dicho` prenant en entrée une liste de nombres `L` triée dans l'ordre croissant et un nombre `a`, et renvoyant `True` si `a` est un élément de `L`, et `False` sinon. On s'aidera du code à trou suivant.

```
def recherche_dicho(a,L):
    g,d=0,len(L)-1 # On recherche initialement dans toute la liste

    while g<=d: # Tant que la zone de recherche est non vide
        m=(g+d)//2 # "milieu" de g et d
        if  L[m]== a : # Si on trouve a au milieu
            return True
        elif  ... :
            g= ... # Recherche dans la sous liste de droite
        else :
            ...    # Recherche dans la sous liste de gauche

    # Si la boucle while s'est arrêté sans trouver a :
    return(False)
```

Remarque. On obtient une fonction `recherche_dicho` de complexité particulièrement intéressante (c'est-à-dire à l'exécution rapide), qui nous permet largement de faire les tests de l'exercice 6 avec cent milliards au lieu de cent millions.

Exercice 8. Reprendre l'exercice 6 avec cette fonction et comparer les temps d'exécutions.

3. Recherche de l'indice d'un élément dans une liste triée

On peut appliquer le même principe pour donner l'indice d'une occurrence d'élément dans une liste de nombres `triée`.

Exercice 9. En s'inspirant de la recherche dichotomique d'élément dans une liste triée, écrire le code d'une fonction `indice_dicho` prenant en entrée un nombre `a` et une liste de nombres triée (dans l'ordre croissant) `L`, et renvoyant en sortie l'indice d'une occurrence de `a` dans `L` si `a` est un élément de `L`, et affichant (avec `print`) le message "élément non trouvé" sinon.

Indication : On applique exactement le même principe que pour `recherche_dicho`, donc le code sera très proche. Commencez par comprendre comment l'algorithme s'adapte sur une liste simple, puis adaptez le code de `recherche_dicho`.

III. Enrichissement d'une méthode de tri (TP n°5)

Le but de cette partie et d'enrichir l'implémentation du tri à bulles pour trier des listes plus complexes que de simples listes de nombres, selon des critères particuliers.

Exercice 10. Écrire le code d'une fonction Python d'entête `def TriLongueur(L):` prenant en entrée une liste `L` dont les éléments sont des listes, et triant `L` de sorte que ces listes soient rangées par longueurs décroissantes.

Par exemple, à la suite du code suivant :

```
L= [ [3,2,1], [5,4,4,1], [1], [2,1] ]
TriLongueur(L)
```

la liste `L` devra contenir : `[ [5,4,4,1], [3,2,1], [2,1], [1] ]`.

Exercice 11. Un centre de conférence reçoit des demandes de la part de divers intervenants voulant utiliser sa salle principale. Afin de choisir parmi ces possibilités, ce centre veut trier ces demandes selon certains critères.

Chaque demande est encodée par un triplet `(d,f,p)` où `d` est l'heure de début souhaité de la conférence, `f` son heure de fin, et `p` une note de prestige, de 1 à 10, qu'attribue (assez arbitrairement, si on y regarde de plus près) le centre de conférence à chaque conférencier.

Le centre de conférence rentre toutes ces demandes dans une liste Python. Par exemple, la liste considérée sera

$$[(13, 15, 5), (12, 17, 6)]$$

s'il a reçu deux demandes : une demande pour une conférence de 13h à 15h pour une conférence de prestige 5, et une demande de 12h à 17h pour une conférence de prestige 6.

1. Écrire le code d'une fonction Python `TriDuree` prenant en entrée une telle liste et renvoyant en sortie cette liste triée par durée de conférence croissante. Les cas d'égalités seront sans importance, et la fonction modifiera son argument en la triant au passage.
2. Écrire le code d'une fonction Python `TriPrestige` prenant en entrée une telle liste et renvoyant en sortie cette liste triée par prestige décroissant. Les cas d'égalités seront sans importance, et la fonction modifiera son argument en la triant au passage.
3. Écrire le code d'une fonction Python `TriPrestigeParHeure` prenant en entrée une telle liste et renvoyant en sortie cette liste triée par "prestige par heure" décroissant. Par exemple, une conférence de 2h avec un prestige de 6 comptera pour 3 prestige par heure. Les cas d'égalités seront sans importance, et la fonction modifiera son argument en la triant au passage.