TP de Python numéro 4 : Recherche dichotomique dans une liste triée.

*Semaine du **jeudi 23 novembre**.*

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. 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 : `[1,5,2,4,6,2,13]`
- On recommence sans considérer le premier élément. La liste devient `[1,2,5,4,6,2,13]`
- On recommence sans considérer les deux premiers éléments : `[1,2,2,4,6,5,13]`
- et ainsi de suite : `[1,2,2,4,6,5,13]`
- `[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 `??`). Rajoutez des commentaires sur votre code, et testez le sur des listes de votre choix.

```
def tri_minimum(L):
        for i in range(len(L)): #On trie à partir du i-ième élément de L
                min = ??        #On va chercher le minimum des éléments au dessus du i-ème
                for j in range(i,len(L)): #Boucle pour rechercher le minimum
                        if L[j]<L[min]:
                                ??
                ??,??=??,??                                     #Ici, on fait l'échange
        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, et on les compare à leur successeur. S'ils ne sont pas dans l'ordre croissant, on les échange.
- On répète assez de fois l'opération ci-dessus pour que la liste soit triée.

On peut montrer qu'une liste de longueur n sera toujours triée au bout de $n-1$ itérations, mais à la place, on va plutôt dire que si un parcourt de la liste est fait sans échanges, alors la liste est 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. Puis, tester votre code sur des listes.

```
def tri_bulle(L):
        desordre=True                          #desordre=True tant que la liste n'est pas triée
        while desordre:
                desordre = False
                for i in range(??):
                        if ?? :
                                desordre=True
                                ??
```

Exercice 4. Implémenter le tri à bulle sans variable `desordre`, en appliquant le théorème mentionné 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` décrits ci-dessus.

II. Algorithmes dichotomiques

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.
3. Définir la liste `T` des entiers de la forme 2^n+1, pour n entier pair entre 0 et 30.

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 de l'indice d'un élément dans une liste triée

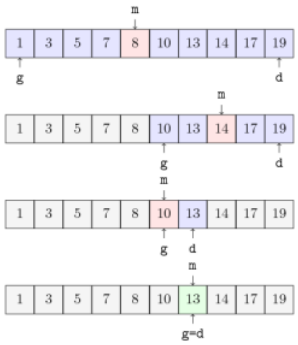
On va appliquer la dichotomie pour écrire une fonction `indice_dicho` prenant en entrée une liste **triée** `L`, et un élément `a` supposé dans la liste `L`, et renvoyant l'indice d'une occurrence de `a` dans `L`.

La recherche dichotomique d'indice dans une liste triée

L'idée est de couper la zone de recherche en deux à chaque étapes. Voici l'algorithme :

1. On cherche l'élément `a` entre les indices `g=0` et `d = len(L)-1`. Pour cela, on regarde l'indice du milieu `m` (si la longueur de `L` est paire, on choisit une des deux valeurs centrales, ça n'a pas d'importance).
2. On compare `L[m]` et `a`.
   - Si `L[m]==a`, on s'arrête et on renvoie `m`
   - sinon, si `L[m]<a`, on cherche dans la sous liste à droite de `m`. On reprends 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`.

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 `indice_dicho` prenant en entrée une liste `L` et un élément `a` de `L`, et renvoyant l'indice d'une occurrence de `a` dans `L`, avec cette recherche dichotomique. On pourra s'aider du code à trous ci-dessous.

```
indice_dicho(a,L):
        g,d=0,len(L)-1
        while g<d:
                m=int((g+d)/2)
                if ??:
                        return ??
                elif ??
                        g=??
                else:
                        ??
        return ??
```

3. Recherche d'un élément dans une liste

On peut appliquer le même principe pour rechercher un élément dans une liste. On obtiendra une fonction `appartient_dicho` de complexité particulièrement intéressante, qui nous permettra largement de faire les tests de l'exercice 5 avec cent milliards au lieu de cent millions.

Exercice 8. En s'inspirant de la recherche dichotomique d'indice dans une liste, écrire le code d'une fonction `appartient_dicho` prenant en entrée un nombre `a` et une liste triée `L` et renvoyant `True` si `a` est un élément de `L`, et `False` sinon.

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

Ensuite, reprendre l'exercice 5 avec `appartient_dicho`, et comparez les temps d'exécution.