TP6 - Dichotomie

→ *Q.1) Créez un fichier* `TP6.py` *dans lequel vous pourrez sauvegarder votre travail.*

A) Recherche dans un tableau trié

On dispose dans cette partie d'un tableau (une liste) trié par ordre croissant. On souhaite déterminer si un élément se trouve ou non dans ce tableau.
On rappelle dans un premier temps un algorithme de recherche qui n'utilise pas le fait que le tableau est trié. On étudie dans un deuxième temps un algorithme qui utilise l'ordre du tableau et est beaucoup plus rapide.

A.1 Recherche naïve

Nous avons déjà vu un algorithme de recherche d'un élément dans une liste dans le TP5.

→ *Q.2) Écrire une fonction* `recherche(L, e)` *qui renvoie* `True` *si l'élément* `e` *est dans la liste* `L` *et* `False` *sinon.*

→ *Q.3) Rappelez la complexité de cet algorithme dans le pire des cas.*

A.2 Recherche dichotomique

L'algorithme précédent fonctionne à tous les coups, mais n'est pas très rapide. Lorsqu'on sait a priori que la liste L est triée, on dispose d'un algorithme bien meilleur qui utilise le principe de **dichotomie** (qui vient du grec pour division en deux parties).
L'idée est simple :

- pour déterminer si un élément `e` est dans une liste triée `L`, on compare `e` à l'élément `m` qui est au milieu de `L`.
- si `e` est égal à `m`, on a trouvé `e`!
- si `e` est inférieur strictement à `m`, on sait alors que `e` ne peut se trouver qu'à gauche de `m`.
- si `e` est supérieur strictement à `m`, on sait alors que `e` ne peut se trouver qu'à droite de `m`.

À l'aide d'une seule comparaison, on élimine la moitié des éléments du tableau! Il ne reste plus qu'à recommencer l'opération avec la liste de gauche ou de droite suivant le cas.
Nous allons coder cet algorithme sous Python de la façon suivante :

1. on définit une plage d'indice délimitée par `g` (premier indice) et `d` (dernier indice) dans laquelle on cherche l'élément.

→ *Q.4) Que valent* `g` *et* `d` *au début de l'algorithme?*

2. On calcule l'indice `m` du milieu de la plage.

→ *Q.5) Étant donnés deux indices* `g` *et* `d`*, avec* $g \leq d$*, comment obtient-on l'indice du milieu? Attention aux types!*

3. On compare `e` à `L[m]` :
   - si `e` est égal à `L[m]`, on a trouvé l'élément, on s'arrête!
   - si `e` est inférieur/supérieur à `L[m]`, on met à jour les valeurs de `g` et `d`.

→ *Q.6) Comment doit-on changer les valeurs de* `g` *et* `d` *suivant les cas?*

4. On recommence à partir de l'étape 2.

→ *Q.7) De quels types d'instructions a-t'on besoin pour coder cet algorithme? De combien de variables?*

→ *Q.8) Quelle condition sur* `g` *et* `d` *doit-on vérifier avant de reprendre à l'étape 2?*

→ *Q.9) Écrire une fonction* `recherche_dichotomique(L, e)` *qui teste si l'élément* `e` *est dans la liste* `L` *(supposée triée) en utilisant la dichotomie.*

→ *Q.10) Commenter la fonction.*

A.3 Tests et comparaisons

L'algorithme précédent n'est pas forcément parfait. Pour s'assurer qu'il fait correctement ce qu'on lui demande, on va le tester sur plusieurs cas. Pour cela, on pourra créer des listes de nombres aléatoires à l'aide du module `random` et utiliser la méthode `sort()` qui trie une liste.

→ *Q.11) Écrire une fonction* `liste_alea(n)` *qui renvoie une liste triée de n entiers choisis aux hasard entre* `0` *et n.*

→ *Q.12) Tester la fonction* `recherche_dichotomique` *sur une liste contenant 1 élément, sur une liste contenant 2 éléments, sur une liste contenant un nombre pair d'éléments et enfin sur une liste contenant un nombre impair d'éléments. Dans chacun des cas, tester avec au moins trois valeurs de* `e` *: le premier et le dernier élément la liste et une qui n'est pas dans la liste. Corriger la fonction si besoin.*

→ *Q.13) Que se passe-t-il si la liste* `L` *est vide ? Modifier la fonction pour prendre en compte ce cas.*

Nous allons maintenant comparer les temps d'exécution des fonctions `recherche` et `recherche_dichotomique` en utilisant la fonction `process_time` du module `time` :

— on stocke l'heure juste avant d'exécuter la fonction ;

— on exécute la fonction ;

— on stocke l'heure juste après l'exécution.

→ *Q.14) Comparer les temps d'exécution des deux fonctions dans le pire des cas avec des listes de taille* 10^4.

A.4 Terminaison et correction

Pour justifier que la boucle `while` s'arrête bien et évaluer par la même occasion la complexité de la recherche dichotomique, nous allons utiliser la valeur `d-g`.

→ *Q.15) Justifier qu'après chaque itération, la valeur de* `d-g` *est au moins divisée par* 2.

→ *Q.16) Comment s'appelle* `d-g` *pour la boucle* `while` *?*

On obtient ainsi par récurrence que si n est la longueur de `L`, alors après k itérations de la boucle, on a `d-g` $\leq \frac{n}{2^k}$.

→ *Q.17) Justifier la terminaison de l'algorithme.*

→ *Q.18) Combien d'itérations fera-t-on au maximum ? Quelle est la complexité de la fonction ?*

Considérons maintenant la propriété suivante : `e` ne se trouve ni dans `L[0:g]` ni dans `L[d+1:len(L)]`.

→ *Q.19) Justifier que la propriété est bien vraie avant de rentrer dans la boucle.*

→ *Q.20) Justifier que si la propriété est vraie au début d'une itération, elle est encore vraie à la fin.*

→ *Q.21) Comment appelle-t-on une telle propriété ?*

On peut maintenant justifier la correction de la fonction :

— la fonction renvoie `True` lorsque `e` est égal à `L[m]` ce qui est correct ;

— la fonction renvoie `False` lorsqu'on a parcouru toute la boucle et alors on a `g = d + 1`. En utilisant la propriété précédente, on sait alors que `e` est ni dans `L[0:d+1]` ni dans `L[d:len(L)]`, autrement dit, `e` n'est pas dans la liste ce qui est aussi correct.

B) Pour les plus motivés

→ *Q.22) Écrire une fonction* `insertion(L, x)` *qui prend en paramètres une liste de nombres triée* `L` *et un nombre* `x`*, et qui insère* `x` *dans la liste* `L` *de sorte que la liste reste triée. La fonction ne renvoie rien.*

→ *Q.23) En utilisant la fonction précédente, écrire une fonction* `tri_insertion(L)` *qui prend une liste de nombres* `L` *en paramètre et renvoie une nouvelle liste triée contenant les mêmes éléments que* `L`*. Cette fonction ne doit pas modifier* `L`.

→ *Q.24) Évaluer la complexité de ce tri.*