TP 13
Dichotomie

Le mot *dichotomie* provient du grec et signifie « couper en deux ». En informatique, il s'agit ici d'une méthode générale de recherche de solution d'un problème en coupant l'intervalle de recherche en deux à chaque étape. Nous allons d'abord l'illustrer par un jeu simple et bien connu.

I Préliminaire : un jeu

On connait le jeu suivant entre deux joueurs : l'un pense à un nombre entre 1 et 100 et l'autre doit le deviner en formulant ses propositions, auxquelles la réponse sera seulement « plus petit », « plus grand » ou bien « trouvé ! ».
Il est tout à fait possible de programmer ce jeu, où le programme choisit le nombre au hasard !

Exercice 1. Compléter le programme fourni : il choisit un nombre au hasard, l'affiche (au début, pour faire des tests, et ensuite on supprimera la ligne) puis demande à l'utilisateur une proposition de nombre. Il dit ensuite si le nombre à trouver est plus grand, plus petit, ou si c'est bon.

Jouez à ce jeu pendant quelques minutes.

Exercice 2. Modifier le programme pour qu'en plus il compte le nombre de tentatives du joueur, affichant à la fin un « trouvé en n coups », et tenter de gagner en un minimum de coups possibles.

On peut apporter diverses modifications comme par exemple choisir un intervalle plus grand pour `n`, ce qui augmente la difficulté.
Qu'observe-t-on ? On comprend vite, même sans arriver à le démontrer rigoureusement (cela est possible) que la meilleure stratégie est de choisir à chaque étape le nombre le plus au milieu possible, ce qui à chaque étape divise par deux l'intervalle dans lequel on cherche.

II Valeurs intermédiaires

On s'inspire de cette méthode pour trouver les zéros d'une fonction : soit une fonction continue f définie sur un intervalle $[a, b]$ à valeurs dans $\mathbb{R}$, supposons par exemple que $f(a) < 0$ et $f(b) > 0$, on cherche une valeur x entre a et b pour laquelle $f(x) = 0$. Alors on calcule $m = \frac{a+b}{2}$ et on teste le signe de $f(m)$. Si $f(m) < 0$ alors on cherchera plutôt entre m et b, et si $f(m) > 0$ on cherchera entre a et m. À chaque étape la longueur de l'intervalle est divisé par 2.

Remarque 1. On peut traduire cela en une *preuve* du théorème des valeurs intermédiaires. Mais cela nécessité d'abord quelques pré-requis sur la continuité...

Exercice 3. On cherche une solution x à l'équation $x^3 = 2$, sachant que x est entre 1 et 2.

1. Écrire une fonction `solution(n)` qui applique la méthode de dichotomie ici décrite en n étapes (n est un entier) et qui retourne un tuple `(a, b)` tel que après n étapes de dichotomie on est sûr que x est entre a et b.
2. Ré-écrire la fonction mais prenant en argument non pas `n` mais un *seuil* `s` qui s'arrête quand on est sûr que la solution cherchée est dans un intervalle $[a, b]$ avec $|b - a| < s$, et donner une valeur approchée de la solution à 10^{-6} près. La fonction s'appelle `solution2(s)`
3. Ceci peut aussi naturellement s'écrire de façon récursive ! C'est alors une fonction `solution3(s, a, b)` qui va tester s'il faut chercher la solution entre a et m ou bien entre m et b, puis s'appeler récursivement, et qui s'arrête si $|b-a| < s$. Écrire cette fonction de façon récursive.

Exercice 4. Écrire des fonctions pour déterminer les solutions des équations suivantes, garanties à 10^{-6} près :

1. $x^5 + x = 7$ ($x > 0$)
2. $3e^x = x + 5$ (pur chacune des deux solutions !) en important l'exponentielle avec `from math import exp`

III Recherche dans une liste triée

Supposons que l'on ait une liste triée, disons de nombres entiers, par ordre croissant. On cherche à savoir si un élément `x` appartient à la liste, et si oui à quel indice. Pour simplifier on suppose que `x` apparait *au plus une fois* dans la liste, donc qu'elle est en fait strictement croissante ; ainsi on ne se préoccupe pas de savoir si l'indice qu'on donne est bien le plus petit indice auquel il apparait.

Exercice 5. Révisions : écrire la fonction `est_strict_croissante(L)` qui teste (c'est à dire, renvoie un booléen `True` ou `False`) si la liste `L` est bien en ordre strictement croissant.

Exercice 6. Révisions : écrire une fonction `cherche_iteratif(L, x)` qui parcourt la liste et teste si l'élément `x` appartient à la liste `L` ; la fonction retourne son indice si elle le trouve, et `-1` sinon.

On souhaite maintenant écrire cette dernière fonction par dichotomie. Pour cela, on initialise deux variables : `a` à zéro et `b` à la longueur de la liste (moins un !) et, dans une boucle, on calcule $m = \frac{a+b}{2}$ et on compare l'élément `x` avec `L[m]`, puis on continue la recherche soit entre a et m soit entre m et b. Il faut alors faire attention à deux choses :

1. Il faut calculer la division $\frac{a+b}{2}$ en nombres entiers ! Sinon on se retrouvera à demander `L[3.5]` ce qui provoquera une erreur...
2. Il faut savoir quand on s'arrête, en ne sachant pas si l'élément `x` est dans la liste ou pas. Mais dans le pire des cas les variables a et b sont deux nombres consécutifs, et soit `x` est égal à `L[a]` ou à `L[b]` soit c'est que `x` n'est pas du tout dans la liste ; de plus si on calculait $\frac{a+b}{2}$ en nombre entier le résultat donnerait encore a et risque de provoquer une boucle infinie.

En fait, dans la boucle on comparera `x` à `L[m]` uniquement ; la comparaison avec `L[a]` ou avec `L[b]` aura été faite au passage précédent dans la boucle... il est donc préférable de comparer dès le début `x` avec les deux bornes de la liste, ce qui permet aussi de détecter dès le début si on cherche hors de la liste (par exemple si on cherche le nombre 200 dans une liste de nombres tous entre 1 et 100).

Exercice 7.

1. Écrire la fonction `cherche_dichotomie(L, x)` qui recherche par cette méthode de dichotomie si l'élément `x` est dans la liste `L`, et retourne son indice si elle le trouve et `-1` sinon. La fonction est à compléter dans le fichier fourni.
2. Amélioration : modifier la fonction pour qu'en plus elle compte combien de passage dans la boucle elle effectue, et affiche (juste avant l'instruction `return`) le nombre de passages nécessaires (un message du type « trouvé en n coups »).

Les fichiers ci-joints contiennent un dictionnaire français de plus de 600 000 mots issu du logiciel libre de correction orthographique GNU Aspell, et un bout de programme qui va charger le dictionnaire comme une liste de mots. Sur les mots, l'opération Python `<` correspond à l'ordre lexicographique, ainsi la fonction précédente s'applique directement si `L` est la variable nommée `dictionnaire` et si `x` est un mot. Si tout se passe bien, le dictionnaire a déjà été trié dans l'ordre de Python. Cela provoque quelques étrangetés pour un humain : les mots avec des majuscules sont rangés d'abord, les mots commençant par des lettres accentuées sont rangés à la fin. Peu importe cependant, tant que le dictionnaire est bien rangé par ordre croissant *au sens* de l'opération `<` telle qu'elle est construite en Python. Testez !

```
>>> len(dictionnaire)
629286
>>> est_strict_croissant(dictionnaire)
True # sinon, il y a un problème, le régler avant de continuer
>>> cherche_iteratif(dictionnaire, "mathématiques")
322819
>>> cherche_iteratif(dictionnaire, "pythoniser")
-1 # ce mot n'existe pas !
>>> for i in [612066, 423643, 156198, 22868, 600164]: print(dictionnaire[i])
```

Ce programme indique que pour trouver le mot `"mathématiques"` dans le dictionnaire, il a fallu comparer ce mot à 322 819 mots successifs. Pour vérifier qu'un mot n'existe pas, il a même fallu le comparer à chacun des plus de 600 000 présents ! Si tout se passe bien, vous pouvez aussi tester la recherche dichotomique dans le dictionnaire. Combien de comparaisons sont nécessaires pour trouver ce même mot ?

```
>>> cherche_dichotomie(dictionnaire, "mathématiques")
```

IV Retour au jeu

On reprend maintenant le jeu de la première partie, mais on inverse les rôles : cette fois, c'est l'utilisateur qui pense à un nombre, et c'est le programme qui formule ses propositions et demande à l'utilisateur « plus grand ou plus petit ? ». On

pourra alors considérer que l'utilisateur doit répondre simplement `"+"` pour dire « plus grand », `"-"` pour dire « plus petit », et `"="` pour dire « trouvé ! ».
La différence avec la première partie, c'est que le programme pourrait mal se comporter si l'utilisateur change en cours de route le nombre qu'il a en tête… ou s'il ne respecte pas les consignes, choisit un nombre hors des bornes… mais le programme devrait être capable de détecter un tel problème !

Exercice 8. Écrire le programme `jeu_inverse()` qui devine un nombre auquel pense l'utilisateur, entre 1 et 100. On pourra compléter la fonction dans le fichier fourni.