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 c'est le programme qui 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 aussi apporter diverses modifications comme par exemple choisir les nombres dans un intervalle plus grand, ce qui augmente la difficulté.
La réponse à la question suivante semble évidente même à un enfant...

Exercice 3. Quelle semble être la meilleure stratégie possible, en rapport avec le titre de ce TP ? Pour un nombre entre 1 et 100, en combien de coups êtes-vous absolument certain de gagner ? Pouvez-vous le démontrer ?

II Application aux solutions d'équations

On s'inspire de cette méthode pour trouver les solutions d'une équation. Soit une fonction f définie sur un intervalle $[a, b]$ à valeurs dans $\mathbb{R}$, continue et croissante, on cherche une solution à l'équation $f(x) = y$ en supposant $f(a) < y < f(b)$. Alors on calcule $m = \frac{a+b}{2}$ et on compare $f(m)$ avec y. Si $f(m) < y$ on doit chercher x entre m et b, et si $f(m) > y$ on doit chercher x entre a et m. Cela divise par deux la taille de l'intervalle de recherche.

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 4. On cherche une solution x à l'équation $x^3 + x = 5$.

1. Avec un tableau de variations, justifier que cette équation admet une unique solution et qu'elle est dans l'intervalle $[1, 2]$.
2. Écrire une fonction `solution(n)` qui applique la méthode de dichotomie ici décrite en n étapes (n est un entier) et qui affiche à chaque fois les bornes a et b de l'intervalle de recherche, telles qu'on soit sûr que la solution x est entre a et b.
3. Ré-écrire la fonction mais prenant en argument non pas `n` mais un *seuil* `s` supposé strictement positif et qui s'arrête quand on est sûr que la solution cherchée est dans un intervalle $[a, b]$ avec $|b - a| < s$ (la fonction n'affiche rien, mais renvoie le couple (a, b)) et donner une valeur approchée de la solution à 10^{-6} près. La fonction s'appelle `solution2(s)`.
4. Bonus : la dichotomie est naturellement un algorithme récursif, car chercher par dichotomie la solution entre a et b revient à chercher par dichotomie la solution soit entre a et m soit entre m et b. Il faut alors mettre les bornes de l'intervalle en tant qu'arguments de la fonction. Écrire la fonction récursive `solution3(s, a, b)` qui renvoie le couple (a, b) tel que la solution est garantie être dans (a, b) avec un écart strictement inférieur à s.

III Application à la recherche dans une liste

La dichotomie est une méthode terriblement efficace pour rechercher un élément dans une liste à condition qu'elle soit **triée**.

Exercice 5. Révisions : écrire une fonction `est_croissante(L)` qui renvoie `True` si la liste `L` est triée, et `False` sinon.

Exercice 6. Révisions : écrire une fonction `cherche_iteratif(L, x)` qui parcourt la liste et donne le premier indice où l'élément `x` apparait, et renvoie `None` s'il n'apparait pas.

La méthode par dichotomie pour chercher à quel indice apparait l'élément `x` consiste à partir d'une liste triée `L` de longueur n et à comparer l'élément `x` avec un élément au milieu de `L`. Si `x` est plus grand, c'est qu'il faut chercher dans la moitié de droite de la liste, ceux qui sont plus grand que le milieu, et sinon il faut chercher dans la moitié de gauche. On travaillera donc en initialisant deux variables : `a` à 0 et `b` à $n-1$ (le dernier indice de la liste). Par rapport au cas des fonctions continues, les nouveaux problèmes suivants se posent :

1. Il n'y a pas toujours exactement un élément au milieu de la liste (cela dépend de la parité de n), on ne peut donc pas à n'importe quel moment comparer `x` avec `L[(a+b)/2]`. D'une part `L[3.5]` affichera une erreur, mais en fait même `L[3.0]`, il faut donc travailler avec la division en nombres entiers `(a+b) // 2`. Cela revient à comparer `x` avec l'élément de `L` au milieu dans la moitié gauche si n est pair ; par exemple dans une liste de longueur 8 le dernier indice est 7, la première moitié correspond aux indices de 0 à 3 et la deuxième moitié de 4 à 7, et `(0+7) // 2 = 3` donc on comparera `x` avec `L[3]`.
2. L'élément `x` n'est pas nécessairement dans la liste, et on ne peut pas diviser par deux les intervalles à l'inifini. Il faut donc savoir s'arrêter quand l'écart entre a et b devient exactement 1 et détecter alors si on trouve l'élément `x` à l'indice a ou b ou pas du tout.
3. Dans le cas où `x` serait strictement plus grand que tous les éléments de la liste, l'algorithme décrit ici s'appliquerait jusqu'au bout pour faire remonter la borne a jusqu'à b ; de même si `x` était plus petit que le minimum, l'algorithme décrit descendrait jusqu'à a. Il est donc préférable de tester dès le départ si `x` est bien compris entre `L[0]` et `L[n-1]`.

Exercice 7. Écrire la fonction `cherche_dichotomie(L, x)` qui recherche par cette méthode de dichotomie si l'élément `x` est dans la liste `L`, et renvoie son indice si elle le trouve et `None` s'il n'est pas dans la liste.

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.

Exercice 8. Avec le fichier joint, tester la recherche dans le dictionnaire, à la fois itératif et par dichotomie. On pourra modifier la fonction `cherche_dichotomie(L, x)` pour afficher le nombre d'étapes qu'elle réalise, et comparer cette valeur avec le logarithme en base 2 de la longueur du dictionnaire. Éventuellement renommer `cherche_dichotomie_print` la version qui affiche le nombre de coups.

Exercice 9. Le fichier joint contient un morceau de programme pour chronométrer le code Python, avec la fonction `timeit` du module `timeit`. Elle prend en argument le nombre de répétitions qu'elle va effectuer, et le résultat est exprimé en secondes. Tester le chronométrage, sur divers mots, à la fois avec les recherches itératives et dichotomiques (dans les versions sans `print`), avec au maximum un millier de répétitions.

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 10. 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.