TP 4 : Algorithmes dichotomiques

On présente dans ce TP les algorithmes dichotomiques.

I. Algorithme dichotomique

I.1. Rechercher une valeur dans un tableau

On commence par un algorithme en apparence très simple : on dispose d'un tableau de valeurs, et on cherche à savoir si une valeur donnée est présente dans le tableau, ou non.

L'algorithme le plus naturel semble le meilleur :

```
1  def recherche(tableau, but) :
2      for valeur in tableau :
3          if valeur == but :
4              return True
5      return False
```

Notons qu'il n'est pas possible de savoir si la valeur à rechercher est absente du tableau avant d'avoir examiné *toutes* les valeurs du tableau. À cause de cela, on en déduit immédiatement que le nombre d'opérations à effectuer par cette fonction, dans le pire des cas, est proportionnelle à la taille du tableau N.
N.B. : on dit qu'il s'agit d'un algorithme de complexité linéaire. Nous y reviendrons lors du chapitre sur la complexité.

Comme la valeur à rechercher peut très bien se trouver à la dernière position, il est impossible de trouver un algorithme plus rapide, car dans le pire des cas, il faudra obligatoirement examiner toutes les valeurs présentes avant de conclure.

I.2. Recherche dans un tableau trié

Supposons à présent que les valeurs du tableau sont triées dans l'ordre croissant. Peut-on améliorer l'algorithme précédent afin de le rendre plus efficace ?

Une première amélioration vient à l'esprit : dès que l'on tombe sur une valeur supérieure à la valeur à rechercher, on peut stopper la recherche car toutes les valeurs suivantes seront elles aussi supérieures à la valeur recherchée.

Exercice 1
Modifier la fonction `recherche` en conséquence. On appellera cette nouvelle fonction `recherche_ordonnee`.

On peut se demander si le nombre d'opérations à effectuer pour trouver la valeur recherchée dans le tableau a diminué. Notons qu'il est toujours possible que la valeur recherchée soit la dernière du tableau. Ainsi, dans le pire des cas, ce nombre reste proportionnel à N (la taille du tableau trié).

I.3. Un algorithme plus efficace : la recherche dichotomique

Il est en fait possible de faire bien mieux que cela, *lorsque le tableau est trié* :

1) On sélectionne l'élément médian du tableau. Trois cas se présentent alors :

- × si l'élément médian est égal à la valeur recherchée, alors on stoppe la recherche (et on renvoie `True`).
- × si l'élément médian est strictement supérieur à la valeur recherchée, alors on poursuit la recherche seulement dans la 1ère moitié du tableau.
- × si l'élément médian est strictement inférieur à la valeur recherchée, alors on poursuit la recherche seulement dans la 2nde moitié du tableau.

2) On poursuit ainsi en subdivisant à chaque étape les sous-tableaux selon leurs valeurs médiances respectives.

3) On finit par aboutir à un tableau de longueur 1 contenant (ou non) la valeur recherchée.

Cette méthode consistant à séparer un problème en 2 parties est appelée **dichotomie** (du grec *dikhotomia* : couper en deux).
En toute généralité, l'idée est d'introduire deux variables `deb` et `fin` qui représentent le début et la fin de l'intervalle de recherche. On compare alors l'élément recherché avec la valeur médiane de l'intervalle. En cas d'échec, on réduit de moitié l'intervalle de recherche selon le procédé décrit ci-dessus. On continue tat que l'élément n'est pas trouvé et que l'intervalle n'est pas vide.

Plus précisément, supposons que l'on dispose d'un tableau ordonné `tab` de longueur `N`.

a) On fixe au départ : `deb = 0` (l'indice minimal du tableau `t`) et `fin = N-1` (l'indice maximal du tableau `t`).

b) On calcule l'indice médian `m = (deb + fin) // 2`.
On utilise une division euclidienne afin de s'assurer que cet indice est un entier.
Trois cas se présentent :

- × si `t[m] == but`, alors on a trouvé l'élément recherché.
- × si `t[m] > but`, alors l'élément recherché ne peut se trouver que dans la 1ère moitié du tableau `t` (si cet élément est bien présent dans le tableau).
  On conserve alors la valeur de `deb` mais on pose `fin = m - 1`.
  On effectue ensuite de nouveau l'étape ***b)***.
- × si `t[m] < but`, alors l'élément recherché ne peut se trouver que dans la 2nde moitié du tableau `t` (si cet élément est bien présent dans le tableau).
  On conserve alors la valeur de `fin` mais on pose `deb = m + 1`.
  On effectue ensuite de nouveau l'étape ***b)***.

c) Si, lors d'une des étapes de l'algorithme, la condition `fin - deb < 0` est réalisée (ce qui est possible avec les instruction `fin = m - 1` ou `deb = m + 1`), on en conclut que l'élément recherché ne se trouve pas dans le tableau. On interrompt alors la recherche.

Exercice 2
Écrire une fonction `recherche_dichotomique(tableau: list, but)` qui renvoie `True` si l'élément `but` est présent dans le tableau, et `False` sinon. On utilisera un algorithme dichotomique.

I.4. Terminaison et coût de l'algorithme de recherche dichotomique

On peut se demander combien de tours de boucle sont nécessaires pour obtenir le résultat. Pour le déterminer, il suffit d'avoir en tête les éléments suivants :

× l'intervalle de recherche initial (l'ensemble d'indices $[\![0, N-1]\!]$) est de longueur N.

× la longueur de l'intervalle de recherche est divisée par 2 à chaque tour de boucle.
À la fin du $m^{\text{ème}}$ tour de boucle, l'intervalle de recherche est donc de largeur $\left\lfloor \dfrac{N}{2^m} \right\rfloor$.

× l'algorithme s'arrête lorsque l'intervalle devient de longueur plus faible que 1.
Notons que l'algorithme se termine donc bien puisque : $\lim\limits_{m\to+\infty} \dfrac{N}{2^m} = 0$. *Il existe donc* $m_0 \in \mathbb{N}$ *tel que :* $\forall m \geqslant m_0$ *:* $\dfrac{N}{2^m} < 1$.

On obtient le nombre d'itérations nécessaires en procédant par équivalence :

$$\begin{aligned} \frac{N}{2^m} \leqslant 1 &\Leftrightarrow \frac{2^m}{N} \geqslant 1 && \text{(par stricte décroissance de la fonction inverse sur } \mathbb{R}_+^*\text{)} \\ &\Leftrightarrow 2^m \geqslant N \\ &\Leftrightarrow m\ \ln(2) \geqslant \ln(N) && \text{(par stricte croissance de la fonction } \ln \text{ sur } \mathbb{R}_+^*\text{)} \end{aligned}$$

Ainsi : $\left\lceil \dfrac{\ln(N)}{\ln(2)} \right\rceil$ tours de boucle suffisent.

On retiendra que si l'on souhaite chercher une valeur dans un tableau **trié** à N éléments, il est nécessaire d'effectuer au pire de l'ordre de $\ln(N)$ opérations, si on utilise un algorithme dichotomique. On dit que cet algorithme a une **complexité** en $O\big(\ln(N)\big)$ (on prononce « grand o de $\ln(N)$ »).

▶ Quel est la complexité de l'algorithme naïf utilisé dans la fonction `recherche` de l'Exercice 1 ?

II. Principe de dichotomie

II.1. Type de problèmes considérés

On considère un problème consistant à la recherche d'un élément `e` vérifiant une certaine propriété `Prop` dans un ensemble `E`. On fera les hypothèses suivantes :

× l'ensemble `E` est de « taille » finie.
Par exemple : un ensemble fini de valeurs ou un un intervalle réel de longueur finie.

× l'ensemble `E` peut être aisément découpé en deux parties.

× la propriété `Prop` peut être aisément testée.

II.2. Algorithme de dichotomie

Pour résoudre le type de problème précédent, on peut proposer la méthode itérative suivante.

- À chaque étape, on réduit l'ensemble de recherche (initialement `E`) en deux parties :
  × une partie utile dans laquelle on est sûr de trouver un élément `e` tel que `Prop(e)`.
  × un autre partie que l'on ne considèrera pas par la suite.
  Note : on n'exige pas que ces deux parties soient de même taille même si c'est souvent le cas.
- On arrête l'itération dès que l'on trouve un élément `e` dans `E` tel que `Prop(e)`.

II.3. Exemple de problèmes qu'on peut résoudre en procédant par dichotomie

Calcul approché d'un zéro d'une fonction

Données : `E` $= [a, b]$, f, ε
`Prop(x)` : « `x` est une valeur approchée à ε près d'un zéro de f »

Jeu du plus ou moins

Données : `E` $= [\![1, N]\!]$
`Prop(x)` : « `x` est la valeur choisie par le Joueur `A` »

Savoir si un entier appartient à un tableau trié d'entiers

Données : `E` un tableau trié d'entiers
`Prop(x)` : « `x` est un élément de `E` »

III. Un exemple pour démarrer : Le jeu du « plus ou moins »

III.1. Principe

On s'intéresse dans ce TP au jeu du « plus ou moins ». Le principe de ce jeu est le suivant : un joueur A choisit un nombre u entre 1 et N (nombre fixé par les joueurs) et un joueur B doit deviner le nombre choisi. Lors de chaque tentative du joueur B, le joueur A doit lui signifier s'il est au-dessus de u ou en dessous. Ainsi, le joueur B peut se rapprocher du résultat à chaque tentative.

Voici un exemple de partie pour $N = 8$.

```
Joueur A : j'ai choisi un nombre entre 1 et 8. À toi de le deviner !
Joueur B : je pense que c'est 2.
Joueur A : raté ! C'est plus !
Joueur B : je pense que c'est 7.
Joueur A : raté ! C'est moins !
Joueur B : je pense que c'est 4.
Joueur A : raté ! C'est plus !
Joueur B : je pense que c'est 5.
Joueur A : c'est ça ! Bien joué :-)
```

III.2. Programmation en Python du jeu de « plus ou moins »

III.2.a) Choix du nombre u

► Importer la bibliothèque `random` sous l'alias `rd`. Que permet de faire la fonction `rd.randint` ?

Pour deux paramètres `a` et `b` entiers, la commande `rd.randint(a,b)` permet de simuler une variable aléatoire de loi $\mathcal{U}([\![a, b-1]\!])$.
Autrement dit, la commande `rd.randint` permet de renvoyer un entier choisi aléatoirement entre a et $b-1$.

III.3. Programmation du jeu

▶ Écrire une fonction `jeuPlusMoins` :

× qui prend en paramètre d'entrée un entier `N`,

× qui n'a pas d'argument de retour (le résultat sera un affichage non stocké dans une variable),

× qui implémente le jeu du plus ou moins.

Plus précisément, il faut effectuer les étapes suivantes.

1) Stocker dans une variable `u` un nombre choisi aléatoirement entre `1` et `N`.

2) Réaliser un affichage permettant d'écrire la phrase :

```
Joueur A : j'ai choisi un nombre entre 1 et ?? . À toi de le deviner !
```

Les symboles `??` seront remplacés à l'écran par la valeur `N` choisie comme paramètre d'entrée de la fonction `jeuPlusMoins`.

3) Écrire une structure itérative qui permet, tant que le joueur B échoue :

- d'afficher la tentative du joueur B (nombre entré au clavier),
- d'afficher la réponse du joueur A.

```
def jeuPlusMoins(N) :
    u = rd.randint(1, N+1)
    print('Joueur A : J ai choisi un nombre entre 1 et '
          + str(N) + '. À toi de le deviner ¡)
    rep = int(input('Joueur B : je pense que c est : '))
    while rep != u :
        if rep < u :
            print('Joueur A : raté ! C est plus ¡)
        else :
            print('Joueur A : raté ! C est moins ¡)
        rep = int(input('Joueur B : je pense que c est : '))
    return print('Joueur A : c est ça ! Bien joué ¡)
```

▶ Tester 3 fois de suite votre fonction pour `N = 128`.
Noter ci-dessous le nombre de tentatives effectuées pour trouver la valeur de `u`.

IV. Jeu du « plus ou moins » : aspect théorique

IV.1. Une stratégie efficace

On s'intéresse ici au jeu du « plus ou moins » pour `N = 128`

▶ En vous inspirant de l'algotihme dichotomique de la partie précédente, proposer une stratégie permettant de trouver la valeur de `u` en `7` essais ou moins.

a) On fixe au départ : `deb = 1` (la valeur minimale possible dans ce jeu) et `fin = 128` (la valeur maximale possible dans ce jeu).

b) Le Joueur B commence par proposer la réponse `rep = (deb + fin) // 2` (la valeur médiane dans la liste des entiers de 1 à 128, c'est-à-dire 64).
Trois cas se présentent :

- × si le Joueur *A* répond que c'est gagné, alors on a trouvé l'entier recherché.
- × si le Joueur A répond que c'est moins, alors l'entier recherché ne peut se trouver que dans la $1^{\text{ère}}$ moitié de la liste des valeurs entre 1 et 128.
  On conserve alors la valeur de `deb` mais on pose `fin = m - 1`.
  On effectue ensuite de nouveau l'étape ***b)***.
- × si le Joueur A répond que c'est plus, alors l'entier recherché ne peut se trouver que dans la 2^{nde} moitié de la liste des valeurs entre 1 et 128.
  On conserve alors la valeur de `fin` mais on pose `deb = m + 1`.
  On effectue ensuite de nouveau l'étape ***b)***.

▶ Si on choisit maintenant `N = 129`, combien de tentatives faut-il (au maximum) avec cette méthode ? Et si `N = 145` ? Et si `N = 256` ? Et si `N = 1048576` ?

▶ De manière générale, si `u` est choisi entre `1` et `N` (quelconque), combien de tentatives sont nécessaires pour trouver la valeur de `u` avec la stratégie précédente ?

IV.2. Retour au clavier

On modifie ici la fonction `jeuPlusMoins` afin que l'on puisse se moquer du joueur B s'il utilise une stratégie moins efficace que celle évoquée au-dessus.

- Ajouter une variable `nbT` permettant de compter le nombre de tentatives effectuées lors du jeu. *(initialement* `nbT=0` *puis* `nbT` *est incrémenté à chaque tour de boucle)*

- En début de programme, ajouter une variable `tmax` calculant le nombre maximum de tentatives nécessaires si l'on joue avec la stratégie précédente.

- En fin de programme, ajouter une structure conditionnelle permettant d'afficher, en fonction du nombre de tentatives nécessaires au joueur B :
  - soit la phrase `Joueur A : tu as gagné en moins de ?? essais. Respect !`
  - soit la phrase `Joueur A : il t'a fallu plus de ?? essais. Looooooooooooooser !`
    (on veillera à respecter le nombre de « o » à afficher)

  Les symboles `??` devront être remplacés, lors de l'affichage à l'écran, par le nombre de tentatives effectuées au cours du jeu.

V. Je veux encore jouer !

▶ Écrire un programme :

× qui demande initialement à l'utilisateur :
`Veux-tu jouer au jeu du plus ou moins ? (o = oui, n = non)`
et stocke le résultat dans une variable `rep`.

× qui permet au joueur B de s'exercer au jeu du « plus ou moins » **tant qu**'il en a envie.

Pour ce faire :

- on devra faire appel à la fonction `jeuPlusMoins`.
  Le paramètre d'entrée `N` sera entré au clavier par l'utilisateur à chaque nouveau jeu.
- À chaque fin de jeu, on demandera à l'utilisateur s'il veut jouer de nouveau :
  `Veux-tu jouer de nouveau ? (o = oui, n = non)`

VI. Calcul approché d'un zéro d'une fonction

VI.1. Théorème des valeurs intermédiaires

On commence par rappeler l'énoncé du TVI vu en cours ainsi que la méthode par dichotomie utilisée pour le démontrer.

TVI : énoncé

Soit $f : [a, b] \to \mathbb{R}$ continue sur l' **intervalle** $[a, b]$.

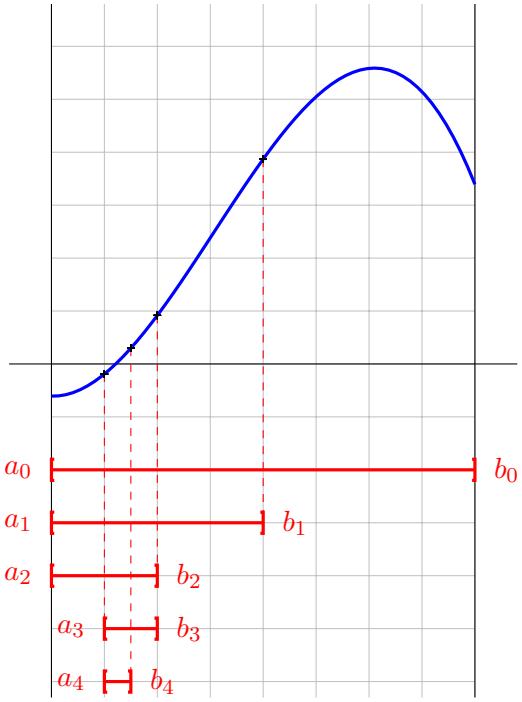
Supposons que : $f(a)f(b) \leqslant 0$.

Alors il existe $c \in [a, b]$ tel que $f(c) = 0$.

Calcul des suites (a_n), (b_n), (c_n)

Cas $f(a) \leqslant 0$ et $f(b) \geqslant 0$

- Initialement, $a_0 = a$, $b_0 = b$, $c_0 = \dfrac{a_0 + b_0}{2}$
- Si $f(c_n) \leqslant 0$ alors :
  - $a_{n+1} = c_n$
  - $b_{n+1} = b_n$
- Si $f(c_n) > 0$ alors :
  - $a_{n+1} = a_n$
  - $b_{n+1} = c_n$
- $c_{n+1} = \dfrac{a_{n+1} + b_{n+1}}{2}$



VI.2. Mise en œuvre en Scilab

Étant donnés un intervalle $[a, b]$, une fonction f et une précision ε, notre but est de calculer une valeur approchée à ε près d'un zéro de f (un élément c tel que $f(c) = 0$).

▶ Dans le cours, on a étudié uniquement le cas où $f(a) \leqslant 0$ et $f(b) \geqslant 0$. Si on considère une fonction f est telle que $f(a) \geqslant 0$ et $f(b) \leqslant 0$, comment se ramener au cas précédent ?

▶ À chaque itération, on sait que $I_n = [a_n, b_n]$ contient un zéro de f. Quelle taille de I_n permet d'assurer que toute valeur contenue dans I_n est une valeur approchée à ε près de c ?

▶ En déduire la condition d'arrêt de l'algorithme.

L'algorithme doit s'arrêter dès que :

▶ En déduire la condition d'itération de l'algorithme.

On doit itérer tant que :

- Écrire la fonction `dichotomie` qui :
  - × prend en paramètres d'entrée une fonction `f`, deux réels `a` et `b` (`a` $<$ `b`) et une valeur `eps`,
  - × renvoie en sortie une variable `res` contenant une valeur approchée à `eps` près d'un zéro de la fonction `f`,
  - × utilise une variable `signe` qui contient -1 si `f(a)` est négatif et 1 si `f(a)` est positif,
  - × utilise une variable `g` initialement affectée à `a` puis mise à jour dans la boucle,
  - × utilise une variable `d` initialement affectée à `b` puis mise à jour dans la boucle,
  - × utilise une variable `m` contenant les valeurs successives du point milieu.

- Dans un nouvel onglet **SciNotes**, coder la fonction $f : x \mapsto x^2 - 2$.
- Utiliser la fonction `dichotomie` afin de calculer une valeur approchée du zéro positif de cette fonction f. On demande une précision de 10^{-10}. Notez le résultat obtenu.
  (à vous de choisir convenablement les paramètres `a` *et* `b`*)*

- Quels sont les zéros de cette fonction f ? Vérifier alors le résultat de l'appel précédent.

- Dans un nouvel onglet **SciNotes**, coder la fonction $g : x \mapsto -x - \mathrm{e}^x$.
- Utiliser la fonction `dichotomie` afin de calculer une valeur approchée du zéro positif de cette fonction g. On demande une précision de 10^{-10}. Notez le résultat obtenu.
  (à vous de choisir convenablement les paramètres `a` *et* `b`*)*

VI.3. Nombre d'itérations nécessaires

- Modifier la fonction `dichotomie` en lui rajoutant un paramètre de sortie `n` permettant de calculant le nombre d'itérations effectuées.
- On peut en fait prévoir le nombre `n` d'itérations effectuées.
  Rappeler tout d'abord la taille de l'intervalle $[a_n, b_n]$ lors de la $n^{\text{ème}}$ itération.

- Rappeler la relation qui lie cette taille et la précision `eps`.

- Isoler `n` dans cette relation.
  En déduire la formule permettant de prévoir le nombre d'itérations nécessaires.

- Comparer le nombre d'itérations donné par cette formule avec le nombre d'itérations obtenu par votre fonction `dichotomie` (deuxième version) sur les fonctions f et g.