Cours 17 – Algorithme de dichotomie (binary search algorithm)

Des volumes importants de données sont susceptibles d'être traitées par les ordinateurs. Des algorithmes efficaces sont alors nécessaires pour réaliser ces opérations comme, par exemple, la sélection et la récupération des données. Les algorithmes de recherche entrent dans cette catégorie. Leur rôle est de déterminer si une donnée est présente et, le cas échéant, d'en indiquer sa position, pour effectuer des traitements annexes. La recherche d'une information dans un annuaire illustre cette idée. On cherche si telle personne est présente dans l'annuaire afin d'en déterminer l'adresse. Plus généralement, c'est l'un des mécanismes principaux des bases de données : à l'aide d'un identifiant, on souhaite retrouver les informations correspondantes. Dans cette famille d'algorithmes, la recherche dichotomique permet de traiter efficacement des données représentées dans un tableau de façon ordonnée.

A / Recherche d'un élément dans un tableau

A.1 / Recherche dans un tableau NON TRIE: approche NAIF

Une première façon de rechercher une valeur dans un tableau est d'effectuer une **recherche naïve** à l'aide d'un parcours de tableau, que l'on peut programmer ainsi :

```
import numpy as np

# Masses approximatives de 6 quarks et de 3 leptons en Mev.c-2
# Masses des 3 autres Leptons (neutrino) environ 0
masse_particules=np.array([1777,106,0.5,7,3,1200,120,174000,4300]

def recherche_naive(tab, val):
    n=len(tab)              # nombre d'éléments du tableau
    for i in range(n):      # i varie entre 0 inclus et n exclu
        if tab[i] == val:
            return i
    return None

print(recherche_naive(masse_particules,120))
print(recherche_naive(masse_particules,300))

masse_triee=np.array(sorted(masse_particules))
print(masse_triee)
print(type(masse_triee))
```

Ici, on renvoie un entier positif ou nul en cas de succès, qui correspond à une position de la valeur recherchée dans la tableau, et `None` en cas d'échec.

```
6
None
```

Si l'on cherche à calculer le nombre d'opérations élémentaires dans le code précédent, on a :

- 2 opérations élémentaires : appel du nombre d'éléments de `tab` et affectation de `n`.
- Pour chaque étape de la boucle `for`, on a 2 opérations élémentaires : appel de l'élément `tab[i]` et comparaison.

La boucle `for` est exécutée n fois dans le pire des cas (si l'élément n'est pas présent dans le tableau par exemple).

Le nombre d'opération élémentaires vaut $2+2n$.

La complexité est linéaire, on note $O(n)$: le temps de recherche double lorsque la taille du tableau double.

A.2 / Recherche dans un tableau TRIE: approche par DICHOTOMIE

Dans l'approche précédente le tableau n'est pas trié (ordonné par valeur croissante). L'approche par dichotomie tire avantage **d'un tableau préalablement trié** (cf. cours d'informatique sur les algorithmes de tri). Nous utiliserons pour l'instant la fonction `sorted` de Python pour trier un tableau.

L'idée centrale de l'approche par dichotomie repose sur l'idée de **réduire de moitié l'espace de recherche à chaque étape** : on regarde la valeur du milieu et si ce n'est pas celle recherchée, on sait qu'il faut continuer de chercher dans la première moitié ou dans la seconde. Plus précisément, **en tenant compte du caractère trié** du tableau, il est possible d'améliorer l'efficacité d'une telle recherche de façon conséquente en procédant ainsi :

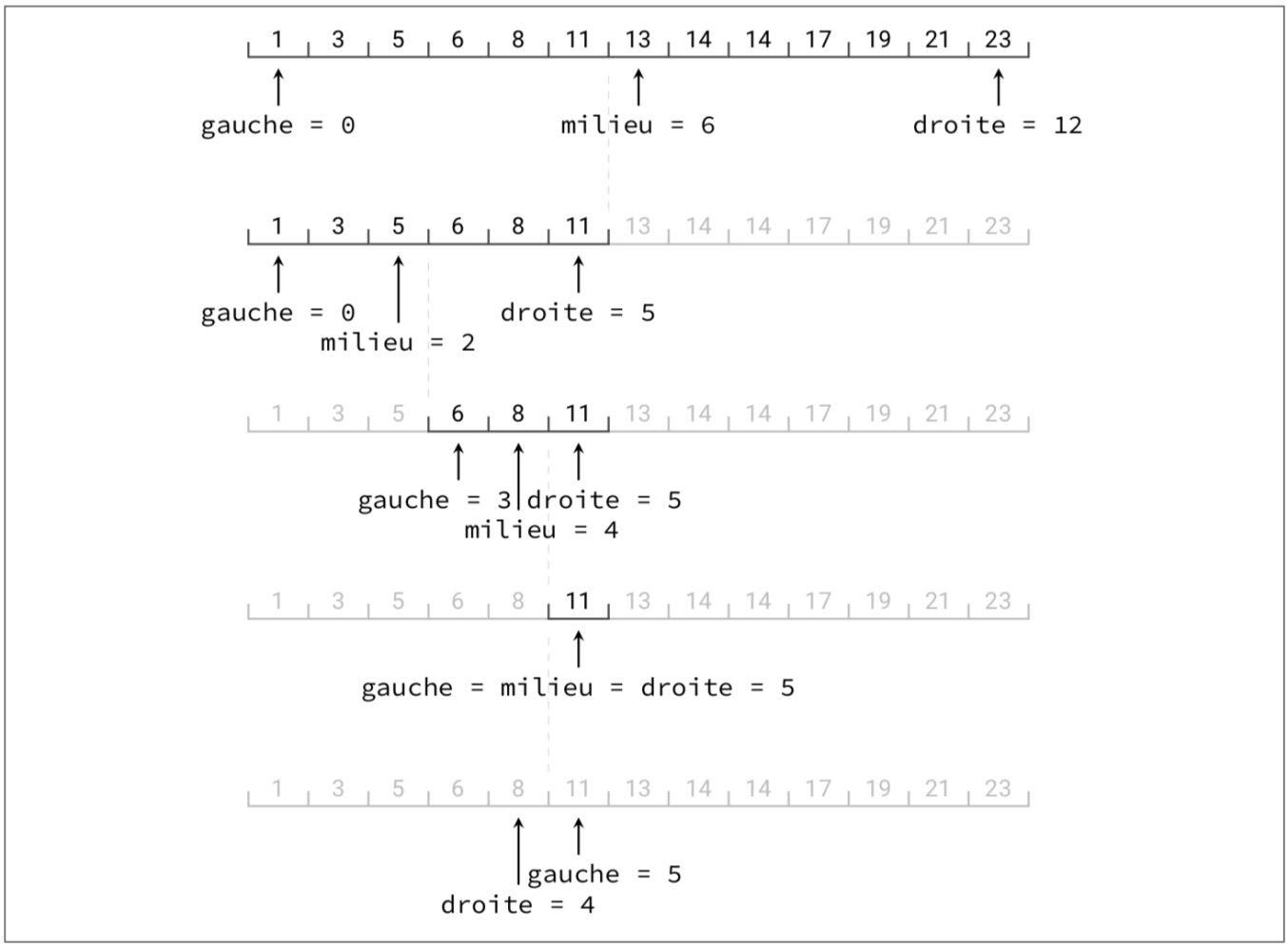
1) On détermine l'élément `m` au milieu du tableau.
2) Si c'est la valeur recherchée, on s'arrête avec un succès.
3) Si non, deux cas sont possibles:
a) Si `m` est plus grand que la valeur recherchée, comme le tableau est trié, cela signifie qu'il suffit de continuer à chercher dans la première moitié du tableau, la moitié gauche.
b) Si non, il suffit de chercher dans la moitié droite.
4) On répète cela jusqu'à avoir trouvé la valeur recherchée ou bien à avoir réduit l'intervalle de recherche à un intervalle vide, ce qui signifie que la valeur recherchée n'est pas présente.

À chaque étape, on coupe l'intervalle de recherche en deux, et on en choisit une moitié. On dit que l'on procède par ***dichotomie***, du grec ***dikha*** (en deux) et ***tomos*** (couper).
La méthode dichotomique divise le problème initial et élimine une partie des données. Vous verrez qu'il s'agit d'un exemple de la méthode plus générale **« diviser pour régner »** (divide and conquer) qui profite de la subdivision pour effectuer moins de calculs.

On obtient le code suivant :

```
import numpy as np

# Masses approximatives de 6 quarks et de 3 leptons en Mev.c-2
# Masses des 3 autres Leptons (neutrino) environ 0
masse_particules=np.array([1777,106,0.5,7,3,1200,120,174000,4300])

masses_triees=np.array(sorted(masse_particules))
# on a la tableau masses_triees=[0.5,3,7,106,120,1200,4300,1777,17400(

def recherche_dichotomique(tab, val):
    gauche = 0
    droite = len(tab) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if tab[milieu] == val:
            # on a trouvé val dans le tableau,
            # à la position milieu
            return milieu
        elif tab[milieu] > val:
            # on cherche entre gauche et milieu - 1
            droite = milieu - 1
        else: # on a tab[milieu] < val
            # on cherche entre milieu + 1 et droite
            gauche = milieu + 1
     # on est sorti de la boucle sans trouver val
    return None

print(recherche_dichotomique(masses_triees,174000))
print(recherche_dichotomique(masses_triees,300))
```

Le schéma ci-dessous illustre le fonctionnement de l'algorithme dans le cas de la recherche de l'élément 10 dans le tableau trié suivant. Ici le programme retourne `None`, 10 ne figure pas dans le tableau.



Pour s'assurer que le programme ci-dessus fonctionne correctement, il faut se poser la question suivante importante : le programme renvoie-t-il bien un résultat ? Ce dernier comportant une boucle non bornée, est-on sûr d'en sortir à un moment donné ? il faut s'assurer que le programme termine, que l'on ne reste pas bloqué infiniment dans la boucle. On peut montrer que c'est bien le cas.

On peut aussi montrer que la complexité de l'algorithme par dichotomie est **logarithmique** en $O(\log n)$. On a donc une recherche dans un tableau beaucoup plus rapide que par la méthode naïf en particulier pour des tableaux comportant un grand nombre d'éléments.

B / Recherche dichotomique du zéro d'une fonction : résolution de l'équation $f(x)=0$

B.1 / Analyse du problème

On considère une fonction continue ($\mathbb{R}\to\mathbb{R}$) sur le segment $[a,b]$ telle que $f(a)f(b)<0$. Par le théorème des valeurs intermédiaires, l'équation $f(x)=0$ admet au moins une solution sur l'intervalle $[a,b]$. Comme la fonction est strictement monotone sur le segment $[a,b]$ alors cette solution est unique (on la notera x_{sol} par la suite) sur le segment $[a,b]$. Nous allons utiliser la méthode dichotomique pour rechercher cette solution.

On divise l'intervalle en deux et on calcule la valeur de la fonction au milieu de l'intervalle. On garde alors une moitié de cet intervalle. On réitère la procédure sur ce sous–intervalle.

Remarque : Le nombre flottant ne permet pas un calcul exact à cause de la représentation des nombres à virgules. Un test du type `a==b` n'a en générale pas de sens si `a` et `b` dont des nombres à virgule flottante. On remplacera donc un tel test par une condition de la forma `abs(a-b)<epsilon` où `epsilon` est une valeur poche de zéro, choisi en fonction du problème à traiter.

B.2 / Principe de l'algorithme

1) On sait que $a < x_{xol} < b$, on va alors poser $x_m = \frac{a+b}{2}$ (abscisse du milieu).

2) On regarde quel est le signe de $f(x_m)$:

- Si $f(x_m)$ est du même signe que $f(a)$, a prend la valeur de x_m (b reste inchangé).
- Si $f(x_m)$ est du même signe que $f(b)$, b prend la valeur de x_m (a reste inchangé).

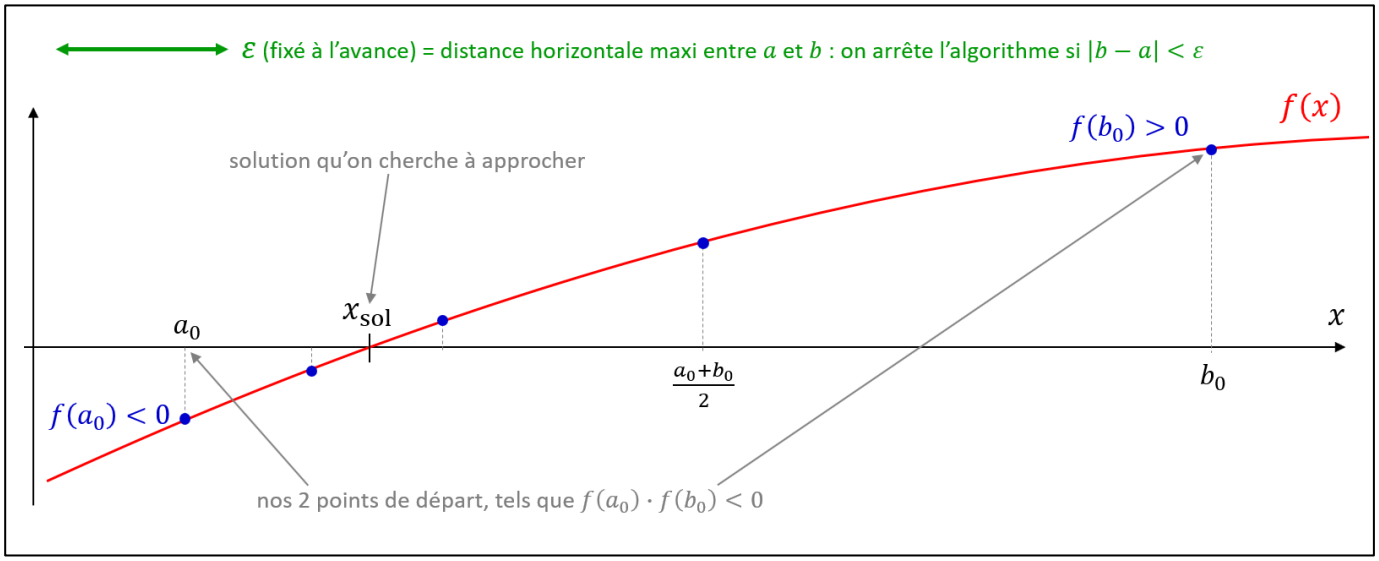
3) Ainsi l'intervalle $[a,b]$ a diminué de moitié et on a à nouveau $f(a)f(b)<0$ donc x_{sol} est toujours entre a et b et on réitère donc en reprenant à l'étape **1)**.

À chaque itération, l'intervalle $[a,b]$ se resserre comme un étau autour de x_{sol} de tel sorte qu'après un certain nombre d'itérations, il s'écrit $[a,b]=[x_{sol}-\varepsilon, x_{sol}+\varepsilon']$ avec ε et ε' très petits. Auquel cas tout x de l'intervalle est à peu près une solution de l'équation $f(x)\approx 0$. On renvoie alors $\frac{a+b}{2} \approx x_{sol}$ après cette dernière itération.

Remarque : Comme on l'a déjà noté, En pratique, on fixera un ε et on arrêtera l'algorithme (boucle `while`) quand $|a-b|<\varepsilon$.

On donne ci dessous une représentation graphique de la méthode. Remarquez que l'on notera, par lisibilité graphique, $a_0, a_1, \ldots$ (resp. $b_0, b_1, \ldots$) les valeurs successives de a (resp. b), alors qu'en pratique il n'y a pas de suite (a_i) (resp. (b_i)), il ne sert à rien de stocker les valeurs intermédiaires de a (resp. b), qui est écrasé par une nouvelle valeur à chaque itération.

En pratique b sera toujours $>a$ donc $|b-a|=b-a$ ce qui évite de se poser la question des valeurs absolues.



Voici un exemple de code Python : on admet que la fonction `f(x)` a été définie à l'extérieur (en amont de cette fonction).

```
def dichotomie(a,b,epsilon):     # pts de départ a et b, et ε fixés à l'avance
    while b-a > epsilon:
        m = (a+b)/2
        if f(a) * f(m) > 0:      # si f(a) et f(m) sont de même signe
            a = m                # point milieu (correspond au x milieu)
        else:                    # si f(b) et f(m) sont de même signe
            b = m
    return (a+b)/2               # correspond au x final
```

(cf. TD17 pour une mise en œuvre pratique de l'algorithme sur un exemple.)

La méthode de recherche par dichotomie est assez simple à coder, elle ne pose pas de problème de terminaison (boucle infinie), et elle est de **complexité logarithmique** en $O\left(\log\frac{(b-a)}{\varepsilon}\right)$.