ITC – cours n°4
Algorithmes dichotomiques

Un algorithme dichotomique est un algorithme dans lequel on réduit de moitié la taille des données à traiter à chaque étape[1]. Ils se caractérisent généralement par une complexité logarithmique.

I. Recherche dans un tableau trié

1.1. Contexte

On considère un tableau de nombres (ou d'un autre type de données ordonnables). dans lequel on veut régulièrement trouver un élément ou sa position (s'il y est). Une première approche est par exemple de parcourir linéairement le tableau et de s'arrêter si on trouve l'élément :

```
def recherche_linéaire(tab: list, x: int) -> bool:
    n = len(tab)
    for elt in tab:
        if elt == x:
            return True
    return False
```

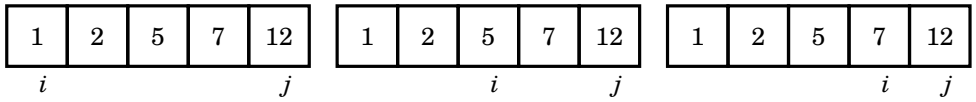
On constate assez directement que la complexité de cet algorithme est $\mathcal{O}(n)$.

1.2. Approche dichotomique

Si on a la garantie que le tableau en entrée est trié, on peut améliorer l'algorithme par une approche dichotomique :

- on regarde si le min (premier élément) est plus grand que le nombre cherché, ou si le max (dernier élément) est plus petit ; dans ces cas, on sait que le nombre cherché n'est pas présent ;
- on regarde si le min ou le max est égal au nombre cherché : si oui, on peut répondre ;
- dans le cas contraire, on regarde l'élément médian (au milieu du tableau), et on en déduit si le nombre cherché se trouverait dans la moitié supérieure ou inférieure ; si on trouve l'élément, on arrête immédiatement ;
- en répétant le processus, on arrive à un sous-tableau de taille 0 ne contenant pas l'élément cherché.

On présente ci-dessous l'exécution de l'algorithme pour un tableau de taille 5 dans lequel on recherche un 7 :



Étape 1 Étape 2 Étape 3

La mise en œuvre pratique de l'algorithme dichotomique emploie deux indices entiers i et j délimitant la séquence extraite $[t_i, \dots t_j]$dans lequel x est recherché. Les valeurs initiales de ces indices sont bien sûr $i = 0$ et $j = n - 1$. On compare ensuite x à l'élément médian d'indice $m = \lfloor (i + j)/2 \rfloor$. Tant que $t_m \neq x$ et $i \leqslant j$, on poursuit la recherche en répétant :

- si $t_m = x$, on s'arrête en renvoyant `True` ;
- si $t_m > x$, alors x peut se trouver dans le sous-tableau $[t_i, \dots t_{m-1}]$ donc on remplace j par $m - 1$;
- si $t_m < x$, alors x peut se trouver dans le sous-tableau $[t_{m+1}, \dots t_j]$ donc on remplace i par $m + 1$.

Si on atteint $i > j$ sans avoir trouvé, le domaine de recherche est vide donc l'élément ne se trouve pas dans la liste.

```
def recherche_dichotomique(tab: list, x: int) -> bool:
    i, j = 0, len(tab)-1  # positions de l'intervalle en cours
    if tab[i] > x or tab[j] < x:
        return False
    if tab[i] == x or tab[j] == x:
        return True
    while j-i > -1:
        # propriété P garantie à l'entrée : tab[i] < x < tab[j]
        m = (i+j) // 2
        if tab[m] == x:  # si on tombe dessus
            return True
        elif tab[m] > x:
            j = m - 1
        else:
            i = m + 1
        # la propriété P est maintenue
    return False
```

1.3. Complexité

Estimons la complexité : on comprend que la boucle while va faire k tours, et que la complexité est $C_n = \mathcal{O}(k)$. Il reste à estimer k.

On cherche l'élément x au sein d'un intervalle $[\![i; j]\!]$ dont la taille se réduit de moitié à chaque tour de boucle. Découpons la démonstration en deux cas :

- si la taille du tableau n peut s'écrire sous la forme $n = 2^k$, alors il faut k tours de

[1] En grec ancien,διχοτομια (dikhotomia) signifie « division en deux parties »

boucle pour arriver à un intervalle de taille 1. On en conclut que, dans le pire cas,

$$k = \log_2 n \quad \Rightarrow \quad C_N = \mathcal{O}(\log_2 n)$$

- sinon, on peut encadrer n pour se ramener au cas précédent :

$$\exists k \in \mathbb{N}, \quad 2^k \leqslant n < 2^{k+1} \quad \Rightarrow \quad k \leqslant \log_2 n < k+1$$

Puisqu'il faut toujours $\mathcal{O}(k)$ tours de boucle pour obtenir un résultat dans le pire cas, on obtient à nouveau une complexité $\mathcal{O}(\log_2 n)$.

> Quand on travaille par dichotomie sur des entiers, on pourra toujours supposer sans perte de généralité que la taille de l'entrée est de la forme 2^k.

II. Recherche du zéro d'une fonction

2.1. Contexte

On considère une fonction f définie et continue sur un intervalle $[a, b]$. On suppose de plus que $f(a) < 0$ et $f(b) > 0$. En vertu du théorème des valeurs intermédiaires, l'équation $f(x) = 0$ admet au moins une solution dans $[a, b]$. L'objectif est de déterminer algorithmiquement une valeur approchée x_0 d'un zéro α de f à une tolérance ε près ; c'est-à-dire que

$$f(\alpha) = 0 \quad \text{et} \quad |x_0 - \alpha| < \varepsilon$$

> S'il y a plusieurs solutions, il est avantageux de trouver un intervalle plus petit dans lequel ne subsiste qu'un seul zéro. Cela nécessite souvent des informations spécifiques à la fonction f, donc au problème à traiter.

On peut envisager une recherche linéaire :

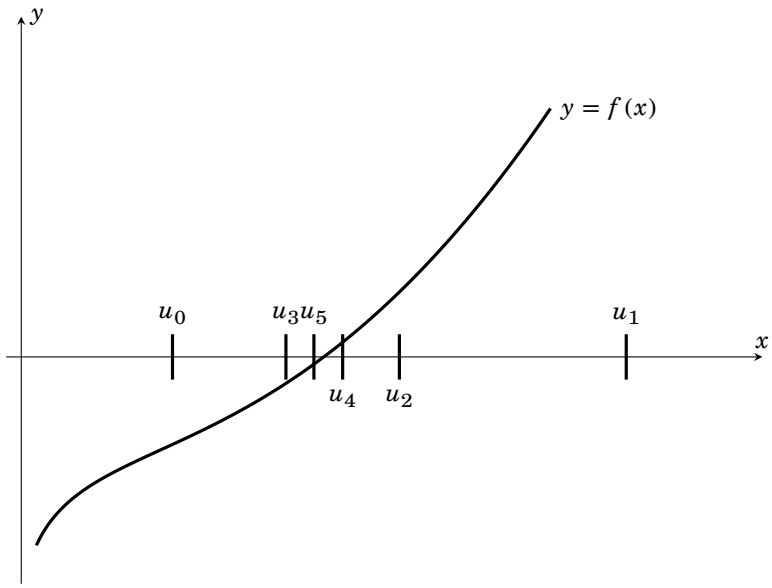
```
def zéro_linéaire(f, a0, b0, epsilon=1e-6):
    a = min(a0, b0)
    signeA = f(a)
    x = a + epsilon
    while f(x)*signeA > 0:
        x += epsilon
    return x - epsilon/2
```

On comprend que l'on teste tous les intervalles de taille ε les uns après les autres jusqu'à tomber sur le bon ; puisqu'il y a au maximum $|b-a|/\varepsilon$ intervalles à tester, la complexité dans le pire cas est donc $\mathcal{O}(|b-a|/\varepsilon)$.

2.2. Principe de la méthode dichotomique

L'idée de l'algorithme de dichotomie est de séparer l'intervalle en deux intervalles de longueur moitié, puis de localiser l'intervalle où se situe la racine. Dans le schéma ci-dessous :

- $[u_0; u_1]$ tel que $f(u_0) < 0$ et $f(u_1) > 0$ est un intervalle de départ qui convient ;
- on coupe l'intervalle en deux : $u_2 = (u_0 + u_1)/2$;
- on constate que $f(u_2)$ est du même signe que $f(u_1)$, donc la solution se trouve dans l'intervalle $[u_0; u_2]$;
- on recommence.



On s'arrête lorsque l'intervalle est plus petit que la précision demandée. On peut alors renvoyer comme résultat la moyenne des deux bornes, une des deux bornes ou les deux bornes (il faut préciser ce choix dans la documentation de la fonction).

```
def zéro_dichotomie(f, a0, b0, epsilon=1e-6):
    a, b = a0, b0
    signeA = f(a)
    while abs(b-a) > epsilon:
        c = (a+b) / 2
        if f(c)*signeA > 0:  # si f(c) est du meme signe que f(a)
            a = c
        else:
            b = c
    return (a+b) / 2
```

Cet algorithme est à connaître, mais surtout à comprendre : il faut pouvoir s'en inspirer pour créer de nouveaux algorithmes de dichotomie. En ce qui concerne l'utilisation concrète, l'algorithme de dichotomie est déjà implémenté dans la bibliothèque scipy : `scipy.optimize.bisect(f,a,b)`.

2.3. Complexité

On peut estimer la complexité de l'algorithme en prenant à nouveau le nombre d'itérations dans la boucle while. On trouve le nombre de boucles par récurrence : à chaque étape, l'intervalle $[a_n; b_n]$ a une largeur $\Delta_n = |b_n - a_n|$ qui est divisée par deux, et le nombre final n_f de boucles est tel que $\Delta_n < \varepsilon$, soit

$$\begin{cases} \Delta_n = \frac{\Delta_{n-1}}{2} \quad \text{et} \quad \Delta_0 = |b-a| \\ \Delta_{n_f} < \varepsilon < \Delta_{n_f-1} \end{cases} \Rightarrow \begin{cases} \Delta_n = \frac{\Delta_0}{2^n} \\ \Delta_{n_f} < \varepsilon < \Delta_{n_f-1} \end{cases}$$

$$\Rightarrow \quad \frac{|b-a|}{2^{n_f}} < \varepsilon < \frac{|b-a|}{2^{n_f-1}} \quad \Rightarrow \quad n_f > \log_2\left(\frac{|b-a|}{\varepsilon}\right) > n_f - 1$$

On a donc une complexité logarithmique en la longueur de l'intervalle $|b-a|$ et en $1/\varepsilon$. Par exemple, pour une précision à 10^{-10} sur un intervalle de largeur 1 au départ, on a 34 itérations.

On a donc une bien meilleure complexité qu'avec le premier algorithme lorsqu'on souhaite diminuer la tolérance ε.

III. Exponentiation rapide

3.1. Contexte

On cherche parfois à calculer de grandes puissances d'un nombre ou d'une matrice, soit calculer x^n avec n entier (nous prendrons le cas où x est un réel pour la suite). Une façon naïve de faire est la suivante :

```
def exponentiation_linéaire(x, n):
    y = 1
    for i in range(n):
        y *= x
    return y
```

qui remplit le contrat mais a une complexité $\mathcal{O}(n)$.

3.2. Principe de l'exponentiation rapide

On remarque que :

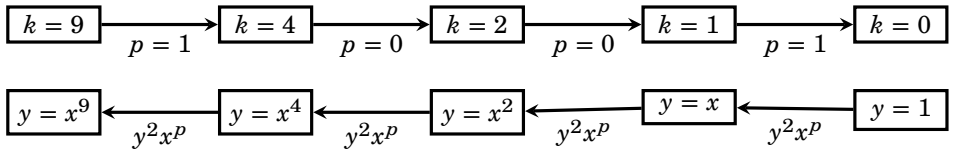
- si n est pair, alors on peut calculer x^n si on connaît $y = x^{n//2}$ en une seule opération : il suffit de mettre au carré, soit $x^n = y^2$;
- si n est impair, alors on peut calculer x^n si on connaît $y = x^{(n-1)//2}$ en deux opérations : on met au carré puis on multiplie par x : $x^n = y^2 x$.

Dans les deux cas, on remarque qu'on a gagné environ $n//2$ multiplications avec cette astuce. En répétant le processus jusqu'à arriver à $x^0 = 1$, on gagne ainsi en rapidité.

On peut écrire la démarche précédente dans les deux cas sous une forme unique :

$$x^n = y^2 x^p \quad \text{avec} \quad p \equiv n \ [2]$$

Par exemple, sur le cas d'un exponentiation par 9 :



La difficulté est de commencer par déterminer « à l'envers » la suite d'opérations à effectuer pour pouvoir ensuite effectuer les multiplications dans le bon ordre.

Une implémentation Python possible est la suivante :

```
def exponentiation_rapide(x, n):
    y, k = 1, n             # k est la puissance en cours de division
    liste_puissances = []  # 0 si puissance paire, 1 sinon
    while k > 0:
        if k%2 == 0:
            liste_puissances.append(0)
        else:
            liste_puissances.append(1)
        k = k//2
    # on reconstruit l'exponentielle à l'envers
    for p in liste_puissances[::-1]:
        y = y**2 * x**p
    return y
```

Une autre proposition est plus compacte, mais plus subtile :

```
def puissance(x, n):
    y = x
    k = n
    résultat = 1
    while k > 0:
        if k%2 == 1:
            résultat *= y
        y *= y
        k //= 2
    return résultat
```

Elle s'appuie sur la décomposition de n dans une base 2 :

$$n = \sum_{i=0}^{i_{\max}} p_i 2^i \quad \text{avec} \quad p_i \in \{0,1\} \quad \Rightarrow \quad x^n = \sum_{i=0}^{i_{\max}} \left(x^{2^i}\right)^{p_i}$$

qui montre que

- si p_i est nul alors la puissance x^{2^i} ne contribuera pas au résultat (d'où l'absence de mise à jour du résultat dans le `if`) ;
- si $p_i = 1$ alors la puissance x^{2^i} doit apparaître dans le produit final, et elle est obtenue en accumulant dans `résultat` y.

Chaque tour de boucle permet donc de calculer un terme du type $y = x^{2^i}$, et de l'accumuler uniquement si $p_i = 1$; k représente à chaque tour l'exposant n dont on a décalé la décomposition en base 2, obtenant ainsi p_i en prenant le plus petit terme de la décomposition par $k \,\%\, 2$, et on le divise par 2 pour à nouveau décaler la représentation en base 2.

3.3. Complexité

On peut, sans perte de généralité, supposer n comme une puissance de 2 : $n = 2^k$. Dans ce cas, on voit qu'il faut k tours de boucle pour diminuer la puissance jusqu'à 0, puis le même nombre pour recalculer l'exponentielle ; d'où une complexité $\mathcal{O}(k) = \mathcal{O}(\log_2 n)$.