SÉANCE 8

RAPPELS SUR LES ALGORITHMES DICHOTOMIQUES

I - Recherche dans une liste triée

I.1 - Le principe de la recherche

On considère une liste `a` de n nombres, triés par *ordre croissant*. Un nombre `x` quelconque étant donné, on cherche à savoir si ce nombre est dans la liste `a`.
On utilise la méthode suivante : on compare x avec l'élément qui est au milieu de la liste dans laquelle on le cherche. Plus précisément si on cherche x dans la sous-liste dont les indices constituent l'intervalle $[\![m, n]\!]$, on comparera x à l'élément d'indice `(n - m)//2`.
Si `x` lui est égal, alors `x` appartient à la liste et on s'arrête là (on a trouvé `x` à la position `(n - m)//2`).
Sinon, si `x` lui est supérieur, alors comme tous les éléments placés avant $a[(n-m)//2]$ sont plus petits que $a[(n-m)//2]$, `x` leur est aussi supérieur. On continue alors en cherchant `x` parmi les éléments situés strictement après $a[(n-m)//2]$.
On procéderait de la même manière si on avait `x` inférieur à $a[(n-m)//2]$, en cherchant parmi les éléments situés strictement avant $a[(n-m)//2]$.
En procédant ainsi on cherche `x` dans des listes de plus en plus petites (le nombre d'éléments des listes successives dans lesquelles on cherche `x` est une suite strictement décroissante). Ainsi soit on trouve `x`, soit on finit par le chercher dans une liste vide et la conclusion est alors immédiate : `x` n'est pas dans la liste.

I.2 - Une version itérative

```
def recherche_dicho(x, a):
    """
    Entrée : x (float ou int)
             a (list) une liste de nombres (float ou int)
             supposée triée par ordre croissant
    Sortie : l'indice d'une occurrence de x dans a si x est dans a
             sinon None
    Méthode par dichotomie (séparation de la liste de recherche en
    deux parties égales à une unité près).
    """

    m = 0
    n = len(a) - 1

    while m <= n:
        i0 = (m + n)//2
        if x == a[i0]:
            return i0
        elif x > a[i0]:
            m = i0 + 1
        else:
            n = i0 - 1

    return None
```

I.3 - L'approche récursive

On reprend l'algorithme de recherche dichotomique et on en propose une version récursive :

```
def dicho1(x, a):
    """
    Entrée : x (float ou int)
             a (list) une liste de nombres (float ou int)
             supposée triée par ordre croissant
    Sortie : True si x est dans a sinon False (bool)
    """

    if len(a) == 0:
        return False

    i0 = len(a)//2
    if x == a[i0]:
        return True
    elif x > a[i0]:
       return dicho1(x, a[i0 + 1:])
    else:
        return dicho1(x, a[:i0])
```

Quel est le problème dans l'implémentation ci-dessus ? Quel est l'avantage de celui ci-dessous ?

```
def dicho2(x, a, m, n):
    """
    Entrée : x (float ou int)
             a (list) une liste de nombres (float ou int)
             supposée triée par ordre croissant
    Sortie : True si x est dans a sinon False (bool)
    Recherche dichotomique récursive.
    Implémentation optimisée conservant la complexité en O(ln(n))
    """

    if m >= n:
        return False

    i0 = (m + n)//2

    if x == a[i0]:
        return True
    elif x > a[i0]:
        return dicho2(x, a, i0 + 1, n)
    else:
        return dicho2(x, a, m, i0)
```

II - Exponentiation rapide

Au lieu d'écrire que $x^n = x \times x^{n-1}$, on va décomposer le calcul suivant deux cas principaux :

$$x^n = \begin{cases} x & \text{si } n = 1 \\ (x^2)^{\frac{n}{2}} & \text{si n est pair} \\ x \cdot (x^2)^{\frac{n-1}{2}} & \text{si n est impair} \end{cases}$$

Les deuxièmes et troisièmes cas ci-dessus sont les cas principaux car ils peuvent à nouveau se décomposer en d'autres cas. Le premier est un cas d'arrêt de l'algorithme.

```
def expor(x, n):
    """
    Entrée : x (float), n (int) strictement positif
    Sortie : x**n
    Algorithme d'exponentiation rapide
    """

    p = 1
    y = x
    m = n

    while m != 1:
        if m % 2 == 1:
            p = p*y
        y = y*y
        m = m//2

    return p*y
```

```
def expor_rec(x, n):
    """
    Entrée : x (float), n (int)
    Sortie : x**n (float)
    Calcul récursif d'exponetiation rapide.
    """

    if n == 0: # cas de base
        return 1
    elif n%2 == 0:
        return expor_rec(x*x, n//2)
    else:
        return x*expor_rec(x*x, n//2)
```

III - Autres exemples de stratégie «diviser pour régner»

La méthode «Diviser pour régner» consiste à diviser un problème de taille n en deux sous-problème de taille $\frac{n}{2}$ (ou à peu près) ; puis de résoudre séparément ces problèmes avant de rassembler les résultats pour obtenir la réponse finale. L'idée, pour avoir des gains de performances importants, étant de recommencer récursivement cette méthode sur les sous-problèmes jusqu'à obtenir des cas assez simples pouvant être traités de manière direct :

Algorithme : fonction DR – Diviser pour régner

Données : x
début
  si x est suffisament petit ou simple **alors**
    retourner directement le résultat
  sinon
    Décomposer x en 2 : x_1 et x_2
    $y_1 \leftarrow \mathrm{DR}(x_1)$
    $y_2 \leftarrow \mathrm{DR}(x_2)$
    Rassembler y_1 et y_2 pour trouver la solution y
    retourner y

Il est parfois nécessaire de diviser le problème initial en plus que deux sous-problèmes et il se peut que les sous-problèmes soient choisis de tailles différentes.

Si l'on note $\mathrm{C}(n)$ la complexité requise pour traiter un problème de taille n alors, avec la méthode «diviser pour régner», on a :

$$\mathrm{C}(n) = a\mathrm{C}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + b\mathrm{C}\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n);$$

où :

- a et b sont des coefficients qui peuvent varier suivant les algorithmes. Par exemple,
  - $a = b = 1$ s'il faut traiter les deux sous-problèmes de manière indépendante,
  - $a = 1$ et $b = 0$ s'il ne faut s'occuper que du premier sous-problème,
  - $a + b = 1$ s'il ne faut s'occuper que d'un sous-problème, sans savoir lequel *a priori*;

  On a toujours $a + b \geqslant 1$.
- $f(n)$ représente le coût pour séparer le problème et recombiner les deux résultats.

III.1 - L'exemple de la multiplication des grands entiers

On veut multiplier deux entiers comportant chacun n chiffres : $x = a_{n-1}\dots a_2 a_1 a_0$ et $y = b_{n-1}\dots b_2 b_1 b_0$. En posant la multiplication comme appris à l'école primaire :

$$\begin{array}{rr} & a_{n-1}\dots a_2 a_1 a_0 \\ \times & b_{n-1}\dots b_2 b_1 b_0 \\ \hline \end{array}$$

on s'aperçoit que l'on va effectuer n^2 multiplications simples $(a_i \times b_j)$ – et aussi quelques additions (celles-ci étant beaucoup plus simples pour le processeur, on les laisse de côté dans l'évaluation de la complexité).
On cherche maintenant à découper le problème. Si n est pair (*i.e.* $n = 2m$), on peut écrire :

$$\begin{cases} x = 10^m a + b \\ y = 10^m c + d, \end{cases} \quad \text{avec } a, b, c, d \text{ quatre nombres entiers à } m \text{ chiffres.}$$

Le produit xy s'écrit alors $10^{2m}ac + 10^m(ad + bc) + bd$. On doit donc évaluer quatre produits : ac, ad, bc, bd ; soit $4 \times m^2 = n^2$ multiplications simples. On n'y gagne rien. Mais en étant plus malin, on peut poser :

$$p = ac, \qquad q = bd, \qquad r = (a + b)(c + d),$$

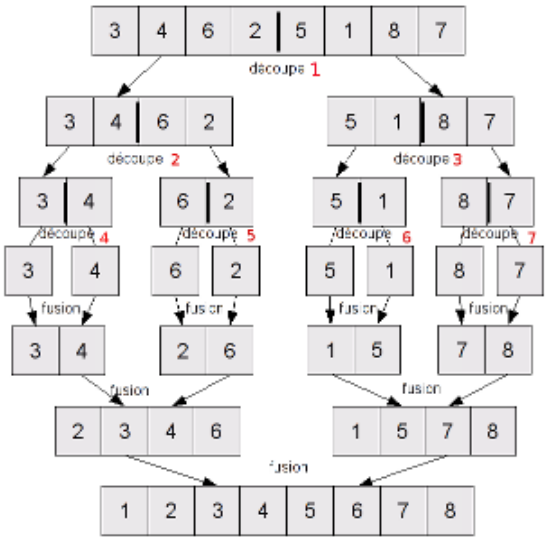
on a alors :

$$xy = 10^{2m}p + 10^m(r - p - q) + q.$$

Il n'y a plus que trois produits à évaluer ! On a gagner 25% de temps de calcul ! (Cette affirmation est un peu rapide, car il y a des calculs supplémentaires pour découper les nombres et rassembler les résultats, mais l'idée est là).
De plus, on peut recommencer ce découpage pour calculer ces trois produits… On arrive au final à une complexité en $\mathrm{O}\left(n^{\log_2 3}\right)$.

III.2 - L'exemple du tri fusion

◇ L'idée derrière le tri fusion est de séparer le tableau en deux sous-tableaux, de trier (récursivement) ces deux tableaux, puis de fusionner ces tableaux triés pour obtenir le résultat final.



On fait donc les étapes suivantes :

- on coupe la liste en deux sous-listes de longueurs (à peu près) égales ;
- on trie les deux sous-listes ;
- on fusionne les deux sous-listes triées en une liste triée.

◇ Programmons cela en Python. On commence par la fonction fusionnant les deux sous-listes triées :

```
def fusion(s, t):
    """ entrée : deux tableaux triés
        sortie : un tableau trié constitué par les éléments de s et t
    """
    l = []
    i, j = 0, 0                      # indices de parcours des deux listes
    while i < len(s) or j < len(t):
        if i == len(s):              # liste s intégralement traitée
            l.append(t[j])
            j += 1
        elif j == len(t):            # liste t intégralement traitée
            l.append(s[i])
            i += 1
        else:                      # listes s et t encore exploitables
            if s[i] < t[j]:
                l.append(s[i])
                i += 1
            else:
                l.append(t[j])
                j += 1
    return(l)
```

On utilise deux indices `i` et `j` et, à chaque tour de boucle on fait avancer soit l'un soit l'autre indice. L'algorithme termine car `i+j` est strictement croissant.

```
>>> a = [1,3,89,102]
>>> b = [0,2,6,8,9,10,42]
>>> fusion(a, b)
[0, 1, 2, 3, 6, 8, 9, 10, 42, 89, 102]
```

On programme alors la fonction de tri :

```
def triFusion(L):
    n = len(L)
    if n <= 1:       # on finit par considérer une liste vide ou à 1 élément
        return L
    else:
        m = n//2
        l1 = triFusion(L[:m])
        l2 = triFusion(L[m:])
        return(fusion(l1, l2))
```

Voici une autre version, quelle différence y a-t-il?

```
def fusionne(T, debut, milieu, fin):
    aux = [0]*(fin-debut+1)
    p1 , p2 = debut, milieu+1
    for i in range(fin-debut+1):
        if p2 == fin+1 or (p1 <= milieu and T[p1] <= T[p2]) :
            aux[i] = T[p1]
            p1 += 1
        else:
            aux[i] = T[p2]
            p2 +=1
    for i in range(debut, fin+1):
        T[i] = aux[i-debut]

def tri(T, debut, fin):
    if debut < fin:
        milieu = (debut + fin)//2
        tri(T, debut, milieu)
        tri(T, milieu+1, fin)
        fusionne(T, debut, milieu, fin)

def trifusion(T):
    tri(T, 0, len(T)-1)
```

Étudions la complexité dans le pire des cas de cet algorithme. Tout d'abord, pour la fonction fusion, étudions le nombre de comparaisons nécessaires. Les longueurs des deux tableaux à fusionner ne diffèrent qu'au plus de 1. Le pire des cas est celui où il faut intercaler alternativement des éléments de chacun des sous-tableaux : si la taille du tableau fusionné est n, cela peut donc nécessiter $(n-1)$ comparaisons entre éléments du tableau.
Ainsi, la relation de récurrence sur la complexité s'écrit pour tout $n \in \mathbb{N}^*$:

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + (n-1).$$

On prouve alors que $C(n) = O(n \ln n)$.