Dichotomie

Le mot dichotomie veut dire “couper en deux”. C’est une méthode très courante pour faire des algorithmes plus efficaces : on coupe le problème en deux, et soit on traite les parties indépendamment (tri fusion, tri rapide …), soit on arrive à ne traiter qu’une seule des deux parties (recherches, exponentiation rapide…).

Recherche dichotomique

On a vu comment on peut trier une liste. Comment utiliser ce tri pour chercher un élément plus vite ?

Idée : on regarde au *milieu* de la liste.

- Si milieu < x, alors x est dans la deuxième moitié
- Sinon, dans la première moitié

−1	0	1	2	3	4	5	10	123	124
−1	0	1	2	3	4	5	10	123	124
−1	0	1	2	3	4	5	10	123	124
−1	0	1	2	3	4	5	10	123	124

```
def recherche_dicho(L, x):
    """Si L est une liste triée dans l'ordre croissant, renvoie l'indice de x dans L,
ou bien None si x n'est pas dans L."""
    i = 0 #borne inf inclue
    j = len(L) #borne sup exclue

    while (j - i) > 0:
        milieu = (i + j) // 2
        if L[milieu] > x:
            i = milieu + 1
        elif L[milieu] < x:
            j = milieu
        else: #cas ou L[milieu] == x
            return milieu
    return None
```

Combien de tour de boucle au maximum pour une liste de taille 8 ? De taille 16 ? Et pour 2^n en général ?

En fait, la complexité est *logarithmique*, soit en $O(\log(n))$.

Recherche du 0 d’une fonction

Supposons qu’on ai une fonction f croissante et qu’on cherche où elle s’annule sur un intervalle $[a, b]$, avec $f(a) < 0$ et $f(b) > 0$.

On peut appliquer le même principe !

Voir l’animation geogebra sur https://tpprepa.github.io/dichotomie.html .

```python
def recherche_zero(f, a, b, eps):
    """Si f(a) < 0 et f(b) > 0, renvoie un zero de f sur [a, b] à eps près."""
    while b - a > eps:
        milieu = a + b / 2
        if f(milieu) < 0:
            a = milieu
        else:
            b = milieu
    return (a + b) / 2
```

On veut trouver le nombre de tours de boucles avant que le programme s'arrête.

Le programme s'arrête quand :

$$\frac{b-a}{2^n} \leq \text{eps}$$

$$\text{Soit : } n \geq \log_2\left(\frac{b-a}{\text{eps}}\right)$$

On a donc, en ordre de grandeur de complexité, $O\left(\log\left(\frac{b-a}{\text{eps}}\right)\right)$ tours de boucles (et opération, car chaque tour de boucle fait un nombre constant d'opérations.)

Les concepteurs de sujet aiment bien tout ce qui touche aux applications physique, donc c'est vraiment le genre d'algorithme qui tombe !

Exponentiation rapide

On peut appliquer ce même principe de "couper en deux" sur des problèmes où c'est moins évident.

Ici, en prenant x un flottant (= un nombre à virgule) et n un entier positif, on veut calculer x^n de manière efficace.

méthode naive : on peut faire

$$\underbrace{x * x * x * \ldots * x * x}_{n \text{ fois}}$$

pour un total de $n-1$ multiplications.

Mais on peut faire mieux !

ex

- considérons x^4.
  On a $x * x * x * x = (x * x)^2$.
  Si on nomme $y = x * x$, il nous faut une multiplication pour calculer y, puis une multiplication pour calculer $y * y$.
  On a économisé une multiplication !
- Et si n est impair ? par exemple x^5
  Dans ce cas, on découpe comme suit : $x^5 = x * x^4$, et on calcule x^4 comme précédemment.

```
def exp_rapide(x, n):
    """Calcule x^n en O(log(n)), avec x un flottant et n un entier positif"""
    reste = 1
    while n ! = 0:
        if n % 2 = = 0:
            n / = 2
            x *= x
        else:
            reste *= x
            x -= 1
    return reste
```

x	n	reste
5	7	1
5	6	5
25	3	5
25	2	125
625	1	125
625	0	125*635

Tri fusion

On va réutiliser l'idée de couper en deux, mais cette fois pour trier une liste. Le tri fusion (ou "partition-fusion") est en 3 étapes :

1. Couper la liste à trier en deux part égales
2. Trier chaque partie séparément
3. Fusionner les deux parties triées

On va s'intéresser à ces étapes une par une.

1. Couper la liste en deux

```
def partition(L):
  """Prend une liste en argument, et renvoie deux nouvelles listes contenant chacune
la moitié des éléments (avec un de plus dans la deuxième si nécessaire)"""
  n = len(L)
  milieu = n//2

  moitie_1 = L[0:milieu]
  moitie_2 = L[milieu:n]

  return (moitie_1, moitie_2)
```

2. Trier chaque partie séparément

On va trier les parties en utilisant ... Le tri fusion lui-même !

3. Fusionner deux listes triées

Montrer animation.

Idée On aimerait placer tous les éléments contenus dans les deux listes `moitie_1` et `moitie_2` dans une nouvelle liste `resultat`.

On va prendre les éléments un par un !

```
def fusion(L1, L2):
    """fusionne deux listes triées en ordre croissant en une seule liste triée
également."""
    resultat = []
    i1 = 0
    i2 = 0
    #On transfère les éléments tant qu'on a terminé aucune des deux listes
    while i1 < len(L1) and i2 < len(L2):
        if L1[i1] < L2[i2]):
            resultat.append(L1[i1])
            i1 += 1
        else:
            resultat.append(L2[i2])
            i2 += 1

    #Une fois qu'une liste est terminée, on peut transférer tout ce qu'il reste dans
l'autre !
    reste1 = L1[i1:]
    reste2 = L2[i2:]

    resultat = resultat + reste1 + reste2

    #Une seule des deux boucles "for" sera non-vide (grâce à la condition du "while")

    return resultat
```

En combinant les trois

```
def tri_fusion(L):
    if len(L) < = 1:
        return L[:]
    milieu = len(L)//2

    #partition
    moitie_1, moitie_2 = partition(L)

    #tri récursif
    moitie_1 = tri_fusion(moitie_1)
    moitie_2 = tri_fusion(moitie_2)

    #fusion
    return fusion(moitie_1, moitie_2)
```