1 Recherche dichotomique

1.1 Recherche dans un tableau trié

Le principe de l'algorithme est le suivant : on maintient à jour une zone de recherche comprise entre deux indices `a` et `b` (intialisés aux extrémités de la liste). En fonction de la valeur de la liste comprise au milieu de la zone de recherche, on sait à partir de l'hyptohèse selon laquelle la liste est triée que la valeur cherchée ne peut se trouver que dans la demi-partie gauche ou demi-partie droite de la zone de recherche. On met à jour les indices jusqu'à trouver l'élément ou obtenir une zone de recherche vide.

```
def rech_dicho(L,x):
    d=0; f=len(L)-1
    while (d>=f):
        m=(d+f)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            d=m+1
        else:
            f=m-1
    return False
```

Remarque 1

On pourra envisager plus tard une version purement récursive de cette fonction en coupant la liste en deux avant chaque appel. On écrit ainsi

```
def rech_dicho_rec(L,x):
    n=len(L)
    if L[n//2]==x:
        return True
    elif L[n//2]<x:
        return rech_dico_rec(L[(n//2):n],x)
    else:
        return rech_dico_rec(L[:(n//2)],x)
```

Il ne faut toutefois toujours pas perdre de vue qu'une instruction de la forme `L[(n//2):n]` n'est pas gratuite à priroi ! En pratique, l'ordinateur recopie la partie de la liste extraite ce qui rajoute un surcoût linéaire en n.

1.2 Recherche du zéro d'une fonction continue et monotone

Soit f une fonction continue sur un segment $[a;b]$. On suppose que $f(a)$ et $f(b)$ sont de signes contraires. Le théorème des valeurs intermédiaires assure l'existence d'un réel x dans $[a;b]$ tel que $f(x)=0$. Pour obtenir une valeur approchée de x, l'algorithme de recherche par dichotomie consiste à couper l'intervalle $[a;b]$ en deux à chaque tour de boucle en conservant systèmatiquement l'intervalle pour lequel f prend des valeurs de signes différents aux extrémités. On arrête l'algorithme lorsque l'intervalle est de largeur inférieure à la précision souhaitée. On peut l'écrire de la manière suivante :

```
def zero_dicho(f,a,b,eps):
    x=a; y=b
    while(y-x>=eps):
        m=(x+y)/2
        if f(x)*f(m)<0:
            y=m
        else:
            x=m
    return (x+y)/2
```

1.3 Complexité

Dans les deux cas, la « taille » de la zone de recherche (l'entier $f-d+1$ dans le premier cas, le réel $y-x$ dans le second) est divisée au moins par deux à chaque nouveau tour de boucle. Initialement, ces tailles valent respectivement n (le nombre d'éléments de `L`) et $b-a$. Après k tours de boucle, elles sont donc majorée respectivement par $n/2^k$ et $(b-a)/2^k$.

- Le premier code s'arrête automatiquement lorsque $d=f$ soit puisqu'il s'agit d'entiers lorsque $n/2^k<1$, ce qui a lieu au plus tard en $\log_2(n)$ étapes. Il s'agit donc d'un code de complexité logarithmique.
- Le second s'arrête lorsque $(b-a)/2^k<\epsilon$, soit lorsque $k>\log_2(b-a)-\log_2(\epsilon)$. Pour une précision à p chiffres, il faut prendre $\epsilon=10^{-p}$ auquel cas $-\log_2(\epsilon)=O(p)$. On a donc une complexité linéaire pour obtenir une telle précision.

2 Exponentiation rapide

Un algorithme d'exponentiation a pour objectif de calculer $\alpha^n=\overbrace{\alpha\times\alpha\cdots\alpha}^{n \text{ termes}}$ où α est un élément d'un ensemble quelconque muni d'une multiplication interne. La méthode naïve consisterait à effectuer $n-1$ multiplications, ce qui serait de coût linéaire. L'exponentiation rapide est une méthode plus efficace consistant à utiliser la formule suivante, où $\mathbb{1}$ désigne le neutre pour la multiplication :

$$\alpha^n=\begin{cases}\mathbb{1} & \text{si } n=0\\ \alpha^{n/2}*\alpha^{n/2} & \text{si } n \text{ est pair}\\ \alpha*\alpha^{(n-1)/2}*\alpha^{(n-1)/2} & \text{si } n \text{ est impair}\end{cases}$$

Le programme suivant présente une implémentation de cette méthode. Il prend en argument l'élément α, la puissance n et la fonction de multiplication utilisée. Si on veut pouvoir traiter le cas $n=0$, il faut également fournir l'élément neutre (soit en argument, soit en définissant une variable globale extérieure au programme).

```
1 def expo_rapide(a,n,mul):
2     if n==1:
3         return a
4     else:
5         b=expo_rapide(a,n//2,mul)
6         if n%2==0:
7             return mul(b,b)
8         else:
9             return mul(a,mul(b,b))
```

Attention à ne pas lancer deux fois le même appel récursif, ce qui nuirait à la complexité ! On remarquera donc dans le programme ci-dessus l'utilisation d'une variable `a` de stockage.