

XII – Algorithmes dichotomiques

1/ Le principe de dichotomie

1.1) Préambule

Programmer le Jeu du Devin, qui fonctionne comme suit :

- ❶ L'ordinateur choisit au hasard un entier entre 1 et 20, que l'utilisateur doit deviner.
On pourra utiliser la commande `randint()` de la bibliothèque `random` : l'instruction `randint(a, b)` renvoie un nombre entier aléatoire pris entre a et b .
- ❷ À chaque essai, l'ordinateur précise si le nombre proposé est trop grand ou trop petit.
- ❸ On répète ceci jusqu'à ce que l'utilisateur trouve le bon nombre.

Tester le jeu, en essayant d'élaborer une stratégie pour minimiser le nombre de tentatives. Normalement, on tombe naturellement sur la méthode exposée ci-dessous.

1.2) Recherche par dichotomie

Supposons que l'on cherche un élément précis dans un ensemble :

- ❶ on coupe alors en deux l'ensemble examiné.
- ❷ on effectue un test pour savoir dans quelle moitié poursuivre la recherche ;
- ❸ on réitère le procédé.

Cette méthode de **dichotomie** permet de trouver l'élément recherché (ensemble fini) ou d'en obtenir un encadrement précis (ensemble infini) assez rapidement. On va l'appliquer à :

- 1) la recherche d'un élément dans un tableau trié ;
- 2) l'approximation du point d'annulation d'une fonction ;
- 3) l'exponentiation d'un nombre ou d'une matrice.

1.3) Complexité des algorithmes dichotomiques

Contrairement aux algorithmes à boucles simples déjà croisés, qui ont une complexité linéaire, les algorithmes dichotomiques sont de complexité logarithmique donc plus efficaces. Prenons le cas de la recherche d'un élément dans une liste finie de longueur n : elle s'arrête quand il ne reste plus qu'un élément, donc le nombre maximal d'étapes k vérifie

$$2^k \simeq n \quad \text{soit} \quad k \simeq \log_2(n)$$

Dans le cas d'un intervalle de longueur ℓ , la précision de l'encadrement atteindra ε lorsque

$$\ell/2^k \simeq \varepsilon \quad \text{soit} \quad k \simeq \log_2(\ell/\varepsilon)$$

2/ Recherche d'un élément dans une liste triée

On veut savoir si un élément a appartient à une liste ordonnée L de n réels.

- ❶ On commence par vérifier que a est compris entre les valeurs extrêmes de la liste.
- ❷ On prend ensuite une valeur au milieu de la liste :
 - Si elle est plus petite que a , on conserve la seconde moitié de la liste ;
 - sinon, on conserve la première moitié.
 Ainsi, l'élément a reste toujours compris entre les valeurs extrêmes de la liste.
- ❸ On réitère ce procédé, jusqu'à ce que l'on obtienne une liste à un élément.
C'est forcément a si ce dernier fait partie de la liste. On peut alors conclure.

La méthode est simple à mettre en œuvre à la main. Il y a deux opérations coûteuses au niveau informatique : choisir une valeur au milieu de la liste et manipuler des sous-listes. Pour cette raison, on va travailler sur les indices des éléments dans la liste de départ :

- 1) on stocke les indices extrêmes de la sous-liste courante dans deux variables g et d ;
- 2) la partie entière de leur moyenne donne l'indice d'une valeur au milieu de la liste ;
- 3) on la stocke dans d si l'on choisit la première moitié de la liste, et dans g sinon.

```

L, a : liste triée, élément recherché
d ← taille(L)-1
g ← 0
Si a > L[d] OU a < L[g] ALORS
    Affichage taquin
    Renvoyer 'Faux'
TANT QUE d-g > 0
    i ← ⌊(d+g)/2⌋
    SI L[i] < a ALORS
        g ← i+1
    SINON
        d ← i
SI L[g] = a ALORS
    Renvoyer 'Vrai'
SINON
    Renvoyer 'Faux'
  
```

```

def Dichotomie(L,a):
    d=len(L)-1
    g=0
    if a>L[d] or a<L[g]:
        print("C'est mal barré !")
        return False
    while d-g>0:
        i=int((d+g)/2)
        if L[i]<a:
            g=i+1
        else:
            d=i
    if L[g]==a:
        return True
    else:
        return False
  
```

Remarque : il ne faut surtout pas effectuer des extractions de sous-listes car ces opérations ont un coût linéaire et vont faire exploser la complexité.

Exemple : tester ceci sur une liste de votre choix.

3/ Recherche du point d'annulation d'une fonction

On dispose d'une fonction f continue sur un intervalle $[a; b]$. On veut savoir si elle s'annule sur cet intervalle et, si oui, trouver une valeur approchée de l'un de ses points d'annulation à $\varepsilon > 0$ près.

- ❶ On vérifie dans un premier temps que $f(a) \times f(b) < 0$.

Le TVI nous assure alors que f s'annule sur $[a; b]$.

- ❷ On note ensuite c le milieu de $[a; b]$:

- si $f(a) \times f(c) \leq 0$, on conserve l'intervalle $[a; c]$.
- sinon, on conserve l'intervalle $[c; b]$.

De la sorte, l'intervalle conservé contient un point d'annulation de la fonction f .

- ❸ On réitère le procédé, jusqu'à ce que la longueur de l'intervalle soit plus petite que 2ε . Le milieu c de l'intervalle $[a; b]$ fournit alors la valeur recherchée.

```
f,a,b,ε : fonction,réel,réel,réel
SI a ≥ b ou f(a)f(b)>0 ALORS
    Affichage taquin
    Interrompre la fonction
TANT QUE b-a>2ε
    c ← (a+b)/2
    SI f(a)f(c) ≤ 0 ALORS
        b ← c
    SINON
        a ← c
Renvoyer (a+b)/2
```

```
def Zero(f,a,b,eps):
    if a>=b or f(a)*f(b)>0:
        print("Changer l'intervalle")
        return
    while (b-a)>2*eps:
        c=(a+b)/2
        if f(a)*f(c)<=0:
            b=c
        else:
            a=c
    return (a+b)/2
```

Exemples : appliquer ce procédé à la résolution sur $[0; 1]$ des équations ci-dessous

(a) $\cos(x) - x = 0$

(b) $1 - \sqrt{\sin(x)} = x$

(c) $x^2 = 2$

4/ L'exponentiation rapide

Soient $a \in \mathbb{R}$ et $n \in \mathbb{N}$. Le calcul de a^n par produits nécessite $n - 1$ multiplications, d'où une complexité linéaire. On peut cependant accélérer le calcul en utilisant des élévations au carré successives, qui ne demandent chacune qu'une seule multiplication. Il suffit pour cela de décomposer l'exposant en somme de puissances de 2 (ou encore d'utiliser son écriture binaire, pour les puristes). Par exemple, on voit que

$$a^{13} = a^{1+4+8} = a \times a^4 \times a^8 = a \times (a^2)^2 \times ((a^2)^2)^2$$

peut ainsi se calculer avec seulement de 5 produits. On retrouvera cette écriture de façon plus systématique en réitérant des divisions des exposants par 2 comme ci-dessous :

$$a^{13} = a^{1+2 \times 6} = a \times (a^2)^6 = a \times (a^2)^{2 \times 3} = a \times ((a^2)^2)^3 = a \times ((a^2)^2)^{1+2} = a \times (a^2)^2 \times ((a^2)^2)^2$$

Cette idée brillante conduit à l'algorithme déjà croisé lors des études de terminaison et validité. Voici son principe général, qui permet de calculer a^n par récurrence :

- Si $n = 2k$ est pair, alors $a^n = a^{(2k)} = (a^2)^k = (a^2)^{\lfloor n/2 \rfloor}$.
- Si $n = 2k + 1$ est impair, alors $a^n = a^{(2k+1)} = a \times (a^k)^2 = a \times (a^2)^{\lfloor n/2 \rfloor}$.

On peut l'écrire simplement comme ceci :

<pre> a, n : réel, entier naturel s ← a N ← n r ← 1 TANT QUE N>0 SI N est pair ALORS s ← s² N ← N/2 SINON r ← r*s N ← N - 1 Renvoyer r </pre>	<pre> def expoRapide(a,n): s = a N = n r = 1 while N>0: if N%2==0: s=s**2 N=N//2 else: s=r*s N=N-1 return r </pre>
---	---

Remarque : on a déjà vu que la proposition « $s^N \times r = a^n$ » est un invariant de boucle, ce qui nous assure que la variable r vaut bien a^n en sortie de boucle.

Examinons la complexité de cet algorithme. Notons p le nombre de divisions par 2 subies par la variable N lors du déroulement de la boucle.

- On peut déjà noter que $2^p \leq n$ si bien que $p \leq \log_2(n)$.
- Dans le pire des cas, chaque division est précédée d'une soustraction : ceci nous donne alors p soustractions, les p divisions et la soustraction finale quand $N = 1$. Le nombre de passages dans la boucle est donc $2p + 1$.
- Ainsi, le nombre de répétitions de la boucle est majoré par $2 \log_2(n) + 1$: on obtient bien une complexité logarithmique.