Algorithmes dichotomiques

Cyril Charignon

Table des matières

Le principe d'un algorithme dichotomique est de « découper le problème en deux », puis de poursuivre la résolution sur un des deux sous-problèmes.

Ce qu'on appelle « découper le problème en deux » va dépendre du contexte.

1 Exemples

1.1 Zéro d'une fonction

Soit $(a, b) \in \mathbb{R}^2$ et $f : [a, b] \to \mathbb{R}$ qu'on suppose continue. Supposons de plus que $f(a) \leq 0$ et $f(b) \geqslant 0$. Alors le théorème des valeurs intermédiaires affirme qu'il existe $c \in [a, b]$ tel que $f(c) = 0$.

Écrivons une fonction prenant en entrée f, a, b, et un flottant $\varepsilon > 0$ et renvoyant un encadrement d'un flottant $c \in [a, b]$ tel que $f(c) = 0$ précis à ε près.

1.2 Calcul de puissance

1.3 Recherche dans un tableau trié

2 Nombre d'étapes

Proposition 2.1. *Soit $N \in \mathbb{N}$ et $(u_0, ..., u_N) \in \mathbb{R}^{N+1}$ une suite finie. On suppose :*

- $u_0 > 0$
- $\forall n \in [\![0, N[\![$, $u_{n+1} \leq \frac{1}{2} u_n$
- $u_N \geq 1$

Alors $N \leq \log_2(u_0)$

Rappel : La fonction $\log_2$ est définie ainsi : $\log_2 : x \mapsto \frac{\ln(x)}{\ln(2)}$. Elle vérifie $\forall x \in \mathbb{R}$, $\log_2(2^x) = x$ et $\forall x \in \mathbb{R}^{+*}$, $2^{\log_2(x)} = x$. Autrement dit c'est la réciproque de la fonction $x \mapsto 2 \times x$.

Démonstration : Par une récurrence immédiate, $u_N \leq \frac{u_0}{2^N}$. Or $n_N \geq 1$, donc $2^N \leq u_0$.

Puis en appliquant la fonction $\log_2$ qui est strictement croissante,

$$N \leq \log_2(u_0).$$

□

Ce théorème permet de majorer le nombre d'étapes effectuées par un algorithme dichotomique. On l'applique en prenant pour tout n u_n la « taille » du problème au bout de n étapes.

Utilisons-le pour calculer le nombre d'opérations effectuées par les exemples précédents :

- Calcul de puissance : soit x un nombre et $n \in \mathbb{N}$. Comptons le nombre de multiplications pour calculer `puissance(x,n)`. D'après la proposition 2.1 le nombre d'étapes pour calculer `puissance(x,n)` est au plus $\log_2(n)$. Comme à chaque étape il y a au plus deux multiplications, cela nous fait au plus $2\log_2(n)$ multiplications pour le calcul de x^n.
- Recherche dans un tableau trié : soit t un tableau, n sa longueur, x un élément. Comptons le nombre de comparaisons. Soit N le nombre d'étapes effectuées dans `appartientDicho(tx,t)`. Pour tout $k \in [\![0, N]\!]$, soit u_k la valeur de `fin-deb` après k étapes. On vérifie que la suite u satisfait les hypothèses de la proposition 2.1. D'où $N \leq \log_2(n)$. Comme il y a une comparaison par étapes, le nombre total de comparaisons est $\leq \log_2(n)$.
- Recherche d'un zéro d'une fonction : cette fois on prend pour u_k la valeur de `(fin-deb)/eps` après k étapes. On trouve que le nombre d'étapes est $\leq \log_2(\frac{b-a}{\varepsilon})$, donc le nombre d'appels à f est $\leq \log_2(\frac{b-a}{\varepsilon})$.

3 Diviser pour régner

La méthode « diviser pour régner » est une généralisation de la dichotomie. Dans celle-ci, après avoir découpé le problème en deux, on a besoin de résoudre les *deux* sous-problème et non un seul comme pour une dichotomie.

Cette fois, il est délicat de programmer à l'aide d'une boucle while : la récursivité est clairement plus adaptée.

Première partie

Exercices

1 Dichotomie

Exercice 1. ** Trichotomie

L'élève Z propose pour accélérer les programmes de dichotomie de découper le problème en trois plutôt qu'en deux. On tombera ainsi plus rapidement sur un cas trivial.

On propose d'appliquer cette technique à la recherche d'un zéro approché d'une fonction vérifiant les hypothèses du TVI.

1. Programmer cette méthode. On écrira donc une fonction prenant en entrée deux flottants a et b tels que $a < b$, une fonction $f \in \mathcal{C}\left([a,b],\mathbb{R}\right)$ telle que $f(a) \leq 0$ et $f(b) \geq 0$, et un flottant $\varepsilon > 0$, et on renverra un encadrement précis à ε près d'un flottant c tel que $f(c) = 0$.
2. Vérifier que le nombre d'étapes effectué par votre fonction est au plus $\log_3\left(\frac{b-a}{\varepsilon}\right)$.
3. Pour évaluer la vitesse de la fonction, on compte le nombre d'appels à f. Comparer votre fonction de trichotomie à celle de dichotomie vue en cours.

Exercice 2. ** Koko mange des bananes

Le gardien du zoo est parti pour h heures, et il a oublié de fermer la porte de la réserve de bananes. Koko bondit sur l'occasion et sur les bananes. Il y a n piles de bananes, et pour tout $i \in [\![0, n[\![$, la pile i contient `nb_bananes_dans_pile[i]` bananes.

Comme Koko est épicurienne, elle souhaite savourer ses bananes et donc les manger le moins vite possible, tout en faisant en sorte d'avoir tout fini avant le retour du gardien.

Elle choisit une vitesse k exprimée en bananes par heure. Chaque heure, elle choisit une pile de bananes et en mange, si possible, k. S'il y en a moins que k, elle mange ce qu'il reste et n'entame pas d'autre pile durant cette heure.

Notons que le temps total mis pour manger toutes les bananes est indépendant de l'ordre dans lequel elle attaque les piles.

On suppose $h \geq n$.

1. Écrire une fonction `vitesse_valide` qui indique si une vitesse k passée en argument est suffisante pour finir toutes les bananes avant le retour du gardien.
2. Méthode naïve : trouver un minorant et un majorant pour la vitesse k. Tester toutes les valeurs, et garder la plus petite qui convienne.
3. Par dichotomie.

Exercice 3. * Élément manquant**

Écrire une fonction `élémManquant` prenant en entrée un tableau t tel qu'il existe $(a,b) \in \mathbb{Z}^2$ tel que t est l'intervalle $[\![a,b[\![$ privé d'un élément, et renvoyant cet élément.

Par exemple `élémManquant([2,3,5,6])` renverra 4.

On supposera que l'élément manquant n'est pas le premier, ainsi `élémManquant([2,3,4])` renverra 5.

On demande naturellement une fonction efficace.

2 Diviser pour régner

Exercice 4. ** Calcul d'un maximum

1. Écrire une fonction calculant le maximum d'un tableau basée sur le principe « diviser pour régner ».

   On pourra écrire une fonction auxiliaire `max_entre` prenant un tableau t et deux indices `de` et `fin` et renvoyant le maximum de `t[deb:fin]`.
2. On pose pour tout $n \in \mathbb{N}$ C_n le nombre de comparaisons nécessaires pour calculer le maximum d'un tableau de longueur n. Montrer que pour tout $n \in \mathbb{N}^*$, $C_n = n - 1$. Conclusion ?

   Par « comparaison », on entend ici « comparaison entre deux éléments du tableau ».

Exercice 5. * Algorithme de Karatsuba**

On étudie ici une méthode pour multiplier deux nombres. Le but est d'effectuer le moins possibles de multiplications à un chiffre.

1. Soient a et b deux entiers, p le nombre de chiffres pour écrire a dans la base choisie et q le nombre de chiffres pour écrire b. Combien de multiplications à un chiffre sont-elles effectuées par la méthode enseignée au CE2 pour calculer le produit $a \times b$?
2. L'algorithme de Karatsuba est de type diviser pour régner. Soient a et b deux entiers, soit n tel que le nombre de chiffre pour écrire a et b en base 2 soit inférieur ou égal à n (ce qui revient à dire que $a < 2^n$ et $b < 2^n$).
   On pose $k = \left\lfloor \frac{n}{2} \right\rfloor$. Soient α, β, γ, δ tels que $a = \alpha + \beta \cdot 2^k$ et $b = \gamma + \delta \cdot 2^k$.
   (a) Comment obtenir α, β, γ, δ en Python ?
   (b) Trouver une formule permettant d'écrire le produit $a \times b$ en trois multiplications. On pourra utiliser la quantité $(\alpha - \beta) \times (\gamma - \delta)$. On ne comptera pas les multiplications par des puissances de 2 qui sont faciles à effectuer par un ordinateur.
   (c) En déduire une fonction de type « diviser pour régner » pour multiplier deux entiers. Cette fonction prendra en entrée les entiers a et b ainsi qu'un entier n tel que a et b sont inférieurs à 2^n.
   Pour multiplier un entier u par 2^k id existe une commande rapide : `u<<k`. En fait cette opération consiste à décaler les bits de u de k cases vers la droite.

Quelques indications

4 1.

2. Récurrence forte. Utiliser $P(n)$: « tout appel à `max_entre` avec des paramètres `deb` et `fin` tels que `fin-deb==n` nécessite $n-1$ comparaisons. »

5 On aura $\beta < 2^k$ et $\alpha < 2^{n-k}$.