Cours 2 — Algorithmes dichotomiques

Étude de cas

« Dichotomie : provient du grec $\delta\iota\chi\sigma\tau o\mu\iota\alpha$ dikhotomia « *division en deux parties* », mot composé des mots $\delta\iota\chi\alpha$ dikha « *en deux* » et $\tau o\mu o\varsigma$ tomos « *section, coupure* » »

L'objectif de ce cours est d'introduire une **approche algorithmique** importante : celles des algorithmes dichotomiques.

Démarche :

1. Nous allons commencer par introduire l'idée générale **(section I)** ;
2. Puis nous allons expliquer et analyser en détail un paradigme important d'algorithme dichotomique **(section II)** ;
3. Enfin nous allons donner deux autres exemples d'approches dichotomiques dans deux contexte différents **(section III)**.

Plan du cours

Prérequis :

1. Le cours sur la complexité.
2. Les TP sur les structures itératives.

Post-requis : nous retrouverons des approches dichotomiques à de nombreux moments en CPGE dont très bientôt :
— quand nous étudierons la forme récursive de certains algorithmes itératifs ;
— dans l'étude des algorithmes de tri.

1 Introduction

Une approche fondamentale en algorithmique quand on cherche à concevoir un algorithme est ce que l'on appelle **« diviser pour régner »** (*divide and conquer*). Elle procède en général en 3 étapes :

1. l'étape de **division** : où l'on découpe le problème en sous-problèmes de plus petite taille.
2. l'étape de **règne** : où l'on résout chacun des sous-problèmes indépendamment de manière directe où par des algorithmes itératifs ou récursifs. Cette étape s'adapte particulièrement bien au besoin de **parallélisation**, c'est-à-dire au fait de faire travailler une grappe d'ordinateurs ou de CPU en parallèle plutôt que de traiter le problème sur un seul et même ordinateur ou CPU.
3. l'étape de **combinaison** : où l'on calcule la solution finale du problème en combinant les « petites » solutions des « petits » sous-problèmes.

L'étape de division peut consister à subdiviser un problème donné en un nombre très grand de sous-problèmes. Mais on peut aussi ne diviser le problème qu'en deux. Dans un tel cas nous parlerons d'une **approche dichotomique**.

En mathématique, et notamment en algèbre linéaire, l'idée de résoudre un problème (comme une équation linéaire), en prenant des coordonnées dans une base, en résolvant des sous-problèmes pour chaque coordonnées et en recombinant les solutions obtenues pour chaque coordonnées en un vecteur de l'espace, constitue un exemple d'approche « diviser pour régner ». En Spé, vous étudierez notamment la diagonalisation et apprendrez à réduire un problème à l'étude de sous-problèmes sur chaque composantes propres, cela constitue aussi un autre exemple de stratégie « diviser pour régner ».

2 Paradigme de la recherche dans un tableau trié

Un des problèmes les plus caractéristique pour la mise en place d'une approche dichotomique est celui dit de la **recherche d'un élément dans une liste triée**.

Problème : étant donné une liste L d'éléments classés par ordre croissant et un autre élément x quelconque, on veut savoir si x est l'un des éléments de la liste L.

Approche naïve : l'algorithme de recherche séquentielle (*linear search* ou *sequential search*). Nous avons déjà vu cet algorithme. Un exemple de code s'écrit :

```
def SeqSearch(L,x):
# L : liste quelconque
# x : object quelconque
# sortie : True si x est dans L et False sinon
    res=False
    i=0
    while i<len(L) and not res:
        if L[i] == x:
            res=True
        i+=1
    return res
```

Listing 1 – Algorithme de recherche séquentiel

Nous considérerons que la taille des données d'entrée est donnée par la longueur n de la liste L. Nous mesurerons donc la complexité de cet algorithme en fonction de n. Dans le pire cas, (celui où x n'appartient pas à L ou celui où l'occurrence de x dans L n'a lieu qu'en $L[n-1]$) il y a n itérations de la boucle while et donc n tests d'égalité. La complexité du pire cas de cet algorithme est donc $O(n)$ et cet algorithme est donc de complexité linéaire. Nous allons voir qu'il s'agit d'un algorithme très lent pour un problème de recherche. En revanche, on remarque que cet algorithme ne nécessite pas, pour fonctionner, que la liste L soit triée par ordre croissant.

Approche dichotomique : l'intérêt est ici d'accélérer grandement l'algorithme de recherche, en revanche, il sera nécessaire que la liste L soit triée. L'idée de **l'algorithme de recherche dichotomique** (*binary search*) est de découper le tableau en deux (ou ce qui s'en rapproche le plus) et de comparer x uniquement au milieu du tableau afin de savoir de quel côté se situe potentiellement x puis de recommencer. Ainsi par exemple : si

$$L = [1,2,3,4,5,6,7,8,9,10] \quad \text{et} \quad x = 7$$

alors :

- étape (1) : test $5 < 7$, on ne regarde que le tableau de droite $[6,7,8,9,10]$
- étape (2) : test $7 < 8$, on ne regarde que le tableau de gauche $[6,7]$
- étape (3) : test $6 <, 7$ on ne regarde que le tableau de droite $[7]$
- étape (4) : test $7 = 7$, c'est terminé.

Il s'agit d'un algorithme, en fait très intuitif, que notre cerveau a appris à pratiquer instinctivement : c'est celui que nous utilisons pour « tâtonner » à la recherche d'un objet sur une table ou pour jouer au jeu du « chaud/froid », etc... Un exemple de code s'écrit :

```
def BinSearch(L,x):
# L : liste triée par ordre croissant
# x : object quelconque
# sortie : True si x est dans L et False sinon
g = 0
d = len(L)-1
while g <= d:
    m=(g+d)//2 # milieu ou partie entière du milieu
    if L[m] == x:
        return True
    elif L[m]<x:
        g=m+1
    else:
```

```
            d=m-1
    return False
```

Listing 2 – Algorithme de recherche dichotomique

Étude de l'algorithme de recherche dichotomique :

1. **Terminaison de l'algorithme :** afin de prouver que l'algorithme termine bien, il nous faut identifier un *variant de boucle* et montrer que ce variant est un entier qui décroit strictement à chaque itération de la boucle while jusqu'à ce qu'il devienne négatif et que le test d'arrêt termine la boucle. Considérons le variant $d-g$ et posons v_k la valeur de ce variant à la k-ième itération. Alors :

$$v_0 = n-1 \text{ (où } n \text{ est la longueur de } L), v_1 \leqslant \frac{v_0}{2} < v_0, v_2 \leqslant \frac{v_1}{2} < v_1, \dots$$

   Donc la suite v_k décroit strictement. En conséquence l'algorithme termine : ou bien la boucle termine sur un return ou bien v_k devient négatif et la condition d'arrêt sera satisfaite.

2. **Correction de l'algorithme :** afin de prouver que l'algorithme fournit un résultat correct, nous devons identifier une propriété (ou prédicat) qui sera toujours vrai à chaque itération. Séparons les cas :
   — si x n'appartient pas à L, alors la propriété

$$\mathcal{P}(k) = "g \leqslant d \quad \text{et} \quad L[m] \neq x \text{ pour } m = E(\frac{d+g}{2})"$$

   est toujours vrai à l'itération k.
   — si x appartient à L, alors la propriété

$$\mathcal{P}(k) = "g \leqslant d \quad \text{et} \quad L[g] \leqslant x \leqslant L[d]"$$

   est toujours vrai à l'itération k. On peut montrer cela par récurrence :
   — pour $k = 1$ c'est vrai car nous avons supposé que x est dans L.
   — supposons que $\mathcal{P}(k)$ est vrai à la k-ième itération. Posons pour simplifier g_k la valeur de g et d_k la valeur de d lors de cette k-ième itération. Posons m la valeur du milieu (soit $E(\frac{d_k+g_k}{2})$). Si $L[m] = x$, alors il n'y a pas d'itération suivante. Traitons par exemple le cas $L[m] < x$, alors x se situe dans la demi-liste de droite. Dans ce cas,

$$g_{k+1} = m+1 \quad \text{et} \quad d_{k+1} = d_k.$$

   Ainsi $L[g_{k+1}] \leqslant x$ car sinon $L[m] < x < L[m+1]$ et donc x ne serait pas dans la liste ce qui est exclu ici. De plus on a bien $x \leqslant L[d_{k+1}]$ par hypothèse de récurrence. Enfin si $d_{k+1} < g_{k+1}$, alors cela signifie que $m \leqslant d_k < m+1$ et donc $m = d_k$ ce qui impose que $m = g_k$ mais encore que $L[g_k] = x = L[d_k]$ par hypothèse de récurrence. Ce cas est exclu car nous avons supposé $L[m] < x$. Il reste le cas $x < L[m]$ qui se démontre de manière similaire.

   Dans chacun des deux cas, nous venons de prouver qu'un prédicat reste vrai lors de chaque itération, ce qui prouve la correction de l'algorithme.

3. **Complexité de l'algorithme** : Reprenons les v_k (valeur du variant $d-g$ lors de la k-ième itération) que nous avons introduit. On obtient par récurrence immédiate que :

$$v_k \leqslant \frac{v_0}{2^k}.$$

   Posons $k = E(\log_2(v_0)) + 1$ alors

$$2^{k-1} \leqslant v_0 < 2^k$$

   Ainsi $\frac{v_0}{2^k} < 1$ et donc $v_{k+1} < 0$ et l'algorithme s'arrête (s'il ne s'est pas arrêté avant). L'algorithme termine donc au plus en k étapes. Il reste à observer que $k = E(\frac{\ln(n-1)}{\ln(2)}) + 1$ et donc

$$k \leqslant \frac{\ln(n)}{\ln(2)} + 1$$

   ou encore $k = O(\ln(n))$. Ceci prouve que la complexité de l'algorithme de recherche dichotomique est en $O(\ln(n))$. Notons qu'il s'agit d'une complexité remarquablement (exponentiellement plus) rapide en comparaison à celle de la recherche séquentielle ou de la simple lecture du tableau. Il est donc infiniment plus rapide de chercher un élément que de lire l'intégralité du tableau.

3 Deux autres exemples d'approche dichotomique dans deux contextes différents

3.1 L'exponentiation rapide

Problème : étant donné un entier g et un entier n, on veut calculer g^n.

Approche naïve : elle consiste à réaliser le produit $g \cdot g$ $n-1$ fois. Un exemple de code s'écrit :

```
def LinearExp(g,n):
    # g : entier
    # n : entier positif supérieur à 1
    # sortie : g**n
    res=1
    for i in range(n):
        res=res*g
    return res
```

Listing 3 – Exponentiation naive

Cet algorithme réalise n multiplications. On va voir que l'on peut faire bien plus efficace que cela.

Approche dichotomique : l'algorithme d'exponentiation rapide. L'idée est ici de réaliser les multiplications par carrés itérés d'où l'autre nom de l'algorithme « *algorithme des carrés itérés* » (*repeated squaring* ou *binary squaring*). Expliquons cela sur un exemple simple : mettons que nous souhaitions calculer g^{11}. Alors :

$$\begin{aligned} g^{11} &= g \cdot g^{10} \\ &= g \cdot (g^5)^2 \\ &= g \cdot \left(g \cdot g^4\right)^2 \\ &= g \cdot \left(g \cdot (g^2)^2\right)^2 \end{aligned}$$

Nous avons donc eu besoin de réaliser deux multiplication par g et 3 mise au carré, soit 5 multiplications en tout au lieu de 11. Nous pouvons rédiger cet algorithme de bien des façons. Si l'on souhaite le rédiger sous la forme d'un processus itératif conditionnel, on obtient un code du type :

```
def BinaryExp(g,n):
    # g : entier
    # n : entier supérieur à 1
    # sortie : g**n
    res=1
    k=n
    while k!=0:
        if k%2==1:
            res=res*g
        res=res*res
        k=k//2
    return res
```

Listing 4 – Exponentiation rapide

De même que dans le cas de la recherche dichotomique sur une liste, le variant de boucle k s'annule à la $k+1$-ième itération si

$$2^{k-1} \leqslant n < 2^k$$

soit si $k = E(\log_2(n)) + 1$. Ainsi, dans le pire cas, l'algorithme réalise $2(E(\log_2(n)) + 1)$ multiplications. Cette quantité est un $O(\ln(n))$. On observe donc que l'algorithme naïf réalise exponentiellement plus de multiplications que l'exponentiation rapide !

Remarque : nous avons énoncé le problème de l'exponentiation rapide sur les entiers (g est ici un entier) mais en réalité nous pouvons énoncer ce problème ainsi que l'algorithme d'exponentiation rapide pour une classe d'objets bien plus vaste, par exemple : g pourrait être une matrice d'entiers, g pourrait être une permutation sur un ensemble fini ou plus

généralement g pourrait être n'importe quel élément dans un groupe mathématique dont on peut représenter en Python les éléments ainsi que la loi. Mentionnons juste que cet algorithme est absolument fondamental en cryptographie (RSA, SSH/SSL, logarithme discret).

3.2 Résolution d'une équation $f(x) = 0$

Problème : étant donné une fonction $f : [a, b] \longrightarrow \mathbb{R}$ continue non nulle en a et en b. On suppose que f admet un changement de signe entre a et b, autrement dit $f(a)f(b) < 0$. Alors le théorème des valeurs intermédiaires garantit le fait qu'il existe un réel $x \in]a, b[$ tel que $f(x) = 0$. Si $\varepsilon > 0$, alors nous souhaitons calculer une approximation décimale de x (un flottant) à ε près.

Cette fois-ci, il n'y a pas d'approche naïve pour résoudre ce problème. Si l'on rajoute des hypothèses de régularités sur la fonction f, d'autres méthodes vont exister comme la méthode de la sécante ou la méthode de Newton. Ces méthodes sont plus rapides que l'algorithme par dichotomie que nous allons rappeler ici (ce dernier a été vu en cours de mathématiques).

Nous souhaitons procéder selon la même idée que l'algorithme de recherche dichotomique dans un tableau. L'idée de la *méthode de dichotomie* (*bisection method*) est de découper l'intervalle $[a, b]$ en 2 et de comparer le signe de $f(a)$ et $f(b)$ uniquement au signe de $f(m)$ où m est le milieu de l'intervalle afin de savoir de quel côté se situe potentiellement x puis de recommencer. On obtient un code du type suivant :

```
def Bisection(f,eps,a,b):
    # f : une fonction python valide sur des flottants de [a,b]
    # eps : flottant strictement positif
    # a,b : deux flottants avec a<b
    # sortie : un flottant qui approxime x à eps près tel que f(x)=0.
    g,d=a,b
    while d-g>eps:
        m=(g+d)/2
        if f(g)*f(m)<0:
            d=m
        else:
            g=m
    return (g+d)/2
```

Listing 5 – Méthode de dichotomie

De même que dans le cas de la recherche dichotomique sur une liste, le variant de boucle $d-g$ à la k-ième itération vérifie :

$$d - g < \frac{b-a}{2^k}.$$

Ainsi la boucle s'arrête dès que

$$\frac{b-a}{2^k} < \varepsilon.$$

Autrement dit, si l'on souhaite une approximation à 10^{-N} près, il sera nécessaire d'avoir réalisé au moins k étape avec :

$$10^N(b-a) < 2^k$$

soit k supérieur à $N\frac{\ln(10)}{\ln(2)} + \frac{\ln(b-1)}{\ln(2)}$. Le nombre d'itérations minimal pour obtenir une approximation à 10^{-N} près est donc linéaire en N, ce qui ne constitue pas une approximation rapide. Pour autant, on retiendra que cet algorithme fonctionne avec des hypothèses de régularité minimales sur la fonction f.