Synthèse du cours 2 : les algorithmes diviser pour régner

21 septembre 2021

François Laroussinie

NB : Ces synthèses ont pour but de compléter les notes prises en cours. Elles ne les remplacent pas ! En particulier, la plupart des preuves n'y figurent pas. Rappel : il faut programmer les algorithmes vus en cours.

1 La recherche dichotomique

1.1 Recherche dichotomique classique

Input : un tableau trié $T[\ldots]$ et un élément x.
Output : l'indice de x dans T si il y est présent, et -1 (ou `None` dans le code Python) sinon.

La taille du problème est le nombre d'éléments dans T.

Le code « à la Python » de cet algorithme peut s'écrire comme ci-dessous où `bg` et `bd` délimite la zone de recherche dans le tableau `T` (on suppose que lorsque `bg` $\leq$ `bd`, la zone de recherche est compatible avec le tableau `T` et donc $0 \leq$ `bg` $\leq$ `bd` $\leq |$`T`$| - 1$). L'algorithme renvoie l'indice de x dans T si $x \in T$, et -1 sinon.

```
def recherche(T,x,bg,bd) :
    if bg>bd : return None
    m = (bg+bd)//2
    v = T[m]
    if (x==v) : return m
    else :
        if x<v : return recherche(T,x,bg,m-1)
        else : return recherche(T,x,m+1,bd)
```

La complexité (dans le pire cas) de l'algorithme est définie par l'équation suivante :

$$C(n) \;=\; C(\lceil \frac{n}{2} \rceil) + O(1) \quad \text{pour } n > 1 \qquad \text{et} \quad C(1) = 1$$

1.2 Application

Input : un tableau $T[\ldots]$ trié et une valeur v.
Output : Le nombre d'occurrences de v dans T.

Pour le résoudre, on peut modifier la recherche dichotomique pour trouver la borne gauche (fct `Bg`) et la borne droite (fct `Bd`) d'une zone contenant la valeur v. On repère l'absence de v dans T par le fait que la borne gauche (de la « zone v ») soit supérieure (strictement) à sa borne droite.

```
def Bd(T,g,d,v) :
    if (g>d) :
        return d
    m = (g+d)//2
    if T[m]> v :
        return Bd(T,g,m-1,v)
    else :
        return Bd(T,m+1,d,v)
```

```
def Bg(T,g,d,v) :
    if (g>d) :
        return g
    m = (g+d)//2
    if T[m]< v :
        return Bg(T,m+1,d,v)
    else :
        return Bg(T,g,m-1,v)
```

```
def NbOcc(T,v) :
    bg=Bg(T,0,len(T)-1,v)
    bd=Bd(T,0,len(T)-1,v)
    if bg>bd :
        return 0
    else :
        return bd-bg+1
```

La complexité de `NbOcc` est en $O(\log n)$. Sa correction repose alors sur deux propriétés sur `Bd` et `Bg` (preuves vues en cours) :

Propriété 1

$$\texttt{Bd(T,g,d,v)}\ \textit{renvoie} : \begin{cases} j & \textit{si } g \leq j \leq d \textit{ et } T[j] = v \textit{ et } (j = d \textit{ ou } T[j+1] > v), \\ d & \textit{si } v > T[d] \textit{ ou } g > d, \\ g-1 & \textit{si } v < T[g]. \end{cases}$$

$$\texttt{Bg(T,g,d,v)}\ \textit{renvoie} : \begin{cases} i & \textit{si } \ g \leq i \leq d \ \textit{ et } \ T[i] = v \ \textit{ et } \ (i = g \textit{ ou } T[i-1] < v), \\ g & \textit{si } \ v < T[g] \textit{ ou } g > d, \\ d+1 & \textit{si } \ v > T[d]. \end{cases}$$

2 Karatsuba

Input : deux entiers a et b de n chiffres dans une base r.
Output : le résultat du produit $a \cdot b$.

La taille du problème est n.

On note a_0, a_1, $\ldots a_{n-1}$ les chiffres de a de telle sorte que : $a = \sum_{i=0}^{n-1} a_i \cdot r^i$. On a de même pour b avec $b = \sum_{i=0}^{n-1} b_i \cdot r^i$.

L'algorithme classique requiert $\Theta(n^2)$ opérations élémentaires (ici les décalages, les additions et multiplications de chiffres) : $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i \cdot b_j \cdot r^{i+j}$.

En fait, on peut se ramener à $\Theta(n^{\log_2 3})$ opérations avec un algorithme diviser-pour-régner.

Prenons $\begin{cases} a \stackrel{\text{def}}{=} \alpha_1 \cdot r^m + \alpha_0 \\ b \stackrel{\text{def}}{==} \beta_1 r^m + \beta_0 \end{cases}$ avec $m = \frac{n}{2}$ et $\alpha_0, \beta_0, \alpha_1, \beta_1 < r^m$.

On a :$a \cdot b \;=\; \underbrace{\alpha_1 \cdot \beta_1}_{K_2} r^{2m} + \underbrace{(\alpha_0 \cdot \beta_1 + \alpha_1 \cdot \beta_0)}_{K_1} r^m + \underbrace{\alpha_0 \cdot \beta_0}_{K_0}$

Et K_1 peut s'obtenir qu'avec un unique produit (de nombres de taille $\frac{n}{2}$) avec :

$$K_1 \;=\; (\alpha_1 + \alpha_0) \cdot (\beta_1 + \beta_0) - K_2 - K_0$$

ou avec :

$$K_1 \;=\; K_2 + K_0 - (\alpha_1 - \alpha_0) \cdot (\beta_1 - \beta_0)$$

L'algorithme de Karatsuba consiste donc à calculer K_0, K_2 selon leur définition, puis K_1 selon une des définitions ci-dessus. On obtient donc la complexité suivante (le $O(n)$ vient des additions et des décalages sur des nombres de tailles n) :

$$T(n) \;=\; 3 \cdot T(\frac{n}{2}) + O(n)$$

ce qui donne du $\Theta(n^{\log_2 3})$ ($\sim \Theta(n^{1.58})$) (voir le Master Theorem).

Algorithme. On peut écrire l'algorithme de Karatsuba comme ci-dessous. Attention : lorsqu'on écrit A+B ou A-B, il s'agit d'une opération sur les tableaux A et B avec propagation de retenue de coût en $O(max(|A|, |B|))$, à écrire ! De plus, on écrit "`([0]*m)>>K1`" pour indiquer que l'on ajoute `m` chiffres `0` à `K1` (*ie* on le multiplie par b^m). On suppose que `A[0]` est le coefficient de poids faible, `A[1]` celui de de b^1, *etc.*

```
def karatsuba(A,B) :   // version 1
    si |A| != |B|, on complète avec des 0 pour obtenir |A| == |B|
    si |A|==1 :
        res = A.B  // res est un tableau de taille 1 ou 2
    sinon :
        m = (|A|+1)/2
        A0 = A[0...m-1],    A1 = A[m...|A|-1]
        B0 = B[0...m-1],    B1 = B[m...|A|-1]

        K2 = karatsuba(A1,B1)
        K0 = karatsuba(A0,B0)
        aux = karatsuba(A0+A1,B0+B1)
        K1 = aux - (K0+K2)
        K1=([0]*m)>>K1
        K2=([0]*(2*m))>>K2
        res = K2+K1+K0
    retourner res
```

3 Master theorem

Remarque : il ne couvre pas tous les cas... La preuve se trouve dans « Introduction à l'algorithmique »[1]. D'autres variantes se trouvent dans « Éléments d'algorithmique »[2].

Théorème 1 *Soient deux rationnels $a \geq 1$ et $b > 1$. Soit $f(n)$ une fonction positive. On considère la fonction $t(n)$ définie par :*

$$t(n) \;=\; \begin{cases} a \cdot t(\frac{n}{b}) + f(n) & si\; n > 1 \\ \Theta(1) & si\; n = 1 \end{cases}.$$

où $\frac{n}{b}$ peut aussi désigner $\lceil \frac{n}{b} \rceil$ ou $\lfloor \frac{n}{b} \rfloor$.
Alors on a :

1. *Si $f(n) = O(n^{\log_b(a)-\varepsilon})$ pour $\varepsilon > 0$, on a : $t(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, on a : $t(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
3. *Si $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ pour $\varepsilon > 0$ et si il existe $c < 1$ tel que $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ pour n assez grand, alors $t(n) = \Theta(f(n))$.*

Preuve : On le montre lorsque n est une puissance de b ($n = b^p$, et donc : $p = \log_b n$). On a :

On a :

$$t(n) \;=\; f(n) + a \cdot f(\frac{n}{b}) + a^2 \cdot f(\frac{n}{b^2}) + \ldots + a^{p-1} \cdot f(\frac{n}{b^{p-1}}) + a^p \cdot d$$

où $d = t(1)$. Le terme $a^p \cdot d$ correspond au coût des appels sur un problème de taille 1 (si on voit les appels récursifs sous la forme d'arbre, c'est le coût associé aux feuilles). Les autres termes représentent le coût cumulé de la fonction f pour tous les niveaux d'appels (l'appel initial de taille n, les a appels sur des problèmes $\frac{n}{b}$, les a^2 appels sur des problèmes de taille $\frac{n}{b^2}$ *etc.*).

Notons que $a^p = a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a}) = n^{\log_b a}$.

Il reste donc à évaluer $g(n) \stackrel{\text{def}}{=} \sum_{i=0}^{p-1} a^i \cdot f(\frac{n}{b^i})$ dans les trois cas du lemme :

Cas 1. $f(n) = O(n^{\log_b a-\varepsilon})$. D'où $f(\frac{n}{b^i}) = O\Big((\frac{n}{b^i})^{\log_b a-\varepsilon}\Big)$. Et :

$$g(n) \;=\; O\Big(\sum_{i=0}^{p-1} a^i \cdot (\frac{n}{b^i})^{\log_b a-\varepsilon}\Big)$$

1. *Introduction à l'Algorithmique*, T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Dunod.
2. *Eléments d'algorithmique*, D. Beauquier, J. Berstel, Ph. Chrétienne, Edition Masson.

Et :

$$\begin{aligned}
\sum_{i=0}^{p-1} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b a-\varepsilon} &= n^{\log_b a-\varepsilon} \cdot \sum_{i=0}^{p-1} \frac{a^i}{(b^{\log_b a-\varepsilon})^i} \\
&= n^{\log_b a-\varepsilon} \cdot \sum_{i=0}^{p-1} \frac{a^i}{a^i \cdot b^{-i\varepsilon}} = n^{\log_b a-\varepsilon} \cdot \sum_{i=0}^{p-1} b^{i\varepsilon} \\
&= n^{\log_b a-\varepsilon} \cdot \frac{1-b^{\varepsilon p}}{1-b^{\varepsilon}} \;=\; n^{\log_b a-\varepsilon} \cdot \frac{1-(b^{\log_b n})^{\varepsilon}}{1-b^{\varepsilon}} \\
&= n^{\log_b a-\varepsilon} \cdot \frac{1-n^{\varepsilon}}{1-b^{\varepsilon}} \;=\; \frac{1}{b^{\varepsilon}-1}(n^{\log_b a} - n^{\log_b a-\varepsilon}) \\
&= O(n^{\log_b a})
\end{aligned} \tag{1}$$

Donc $t(n) = g(n) + a^p \cdot d = O(n^{\log_b a}) + \Theta(n^{\log_b a}) = \Theta(n^{\log_b a})$

Cas 2. $f(n) = \Theta(n^{\log_b a})$. D'où $f(\frac{n}{b^i}) = \Theta((\frac{n}{b^i})^{\log_b a})$. Et :

$$g(n) = \Theta\Big(\sum_{i=0}^{p-1} a^i \cdot (\frac{n}{b^i})^{\log_b a}\Big)$$

Et :

$$\sum_{i=0}^{p-1} a^i \cdot \left(\frac{n}{b^i}\right)^{\log_b a} \;=\; n^{\log_b a} \cdot \sum_{i=0}^{p-1} \frac{a^i}{(b^{\log_b a})^i} \;=\; n^{\log_b a} \cdot p \;=\; n^{\log_b a} \cdot \log_b n$$

D'où on obtient : $f(n) = g(n) + a^p \cdot d = \Theta(n^{\log_b a} \cdot \log n) + \Theta(n^{\log_b a}) = \Theta(n^{\log_b a} \cdot \log n)$.

Cas 3. On a $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ avec $c < 1$. Cela entraîne qu'à chaque niveau des appels récursifs, le coût global dû à f diminue… De $f(\frac{n}{b}) \leq \frac{c}{a} \cdot f(n)$, on obtient :

$$f(\frac{n}{b^i}) \leq \frac{c}{a} \cdot f(\frac{n}{b^{i-1}}) \leq \left(\frac{c}{a}\right)^2 f(\frac{n}{b^{i-2}}) \leq \ldots \leq \left(\frac{c}{a}\right)^i \cdot f(n)$$

Donc : $a^i \cdot f(\frac{n}{b^i}) \leq c^i \cdot f(n)$. Et :

$$g(n) = \sum_{i=0}^{p-1} a^i \cdot f(\frac{n}{b^i}) \;\leq\; \sum_{i=0}^{p-1} c^i \cdot f(n) \;=\; f(n) \cdot \sum_{i=0}^{p-1} c^i$$

Or $\sum_{i=0}^{p-1} c^i \leq \frac{1}{1-c}$. D'où $g(n) = O\Big(f(n)\Big)$ et même $g(n) = \Theta\Big(f(n)\Big)$ car $f(n)$ est dans la somme définissant $g(n)$.

Finalement $t(n) = g(n) + a^p \cdot d = \Theta(f(n)) + \Theta(n^{\log_b a}) = \Theta(f(n))$ car $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$.

□