UFR d'Informatique

Paris 7 – Paris Diderot

Année 2015–2016

Notes de cours d'algorithmique – M1

François Laroussinie

francois.laroussinie@liafa.univ-paris-diderot.fr

Page web du cours : http://www.liafa.univ-paris-diderot.fr/~francoisl/m1algo.html

DRAFT

1 Algorithmes Diviser-pour-régner

On va voir plusieurs exemples classiques (et surement déjà connus) d'algorithmes « diviser-pour-régner ». On donnera à chaque fois la complexité de l'algorithme mais son calcul est laissé de coté dans un premier temps, on verra ensuite le « master theorem » qui permettra de les trouver sans difficulté.

Le schéma d'un algorithme « diviser-pour-régner » est le suivant :

1. On *divise* le problème initial en plusieurs sous-problèmes (on fractionne le problème initial) que l'on *résout* (par des appels récursifs).
2. On construit une solution au problème initial à partir des solutions des sous-problèmes.

Ces différentes étapes se retrouvent dans l'appellation anglaise (plus complète) : « divise and conquer and combine »

1.1 Recherche dichotomique

C'est bien sûr le premier algorithme « diviser-pour-régner », il est connu de tous et il est particulièrement efficace : sans lui, pas de dictionnaire !

Le problème :

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 Python (fichier dicho.py) de cet algorithme peut s'écrire ainsi où `NBC` est une variable globale utilisée pour compter le nombre de comparaisons faites au cours du calcul :

```
def recherche(T,x,bg =0, bd=None) :
    global NBC
    if bd==None :
        NBC = 0
        return recherche(T,x,0,len(T)-1)
    if bg>bd : return None
    m = (bg+bd)/2
    v = T[m]
    NBC += 1
    if (x==v) : return m
    else :
        NBC +=1
        if x<v : return recherche(T,x,bg,m-1)
        else : return recherche(T,x,m+1,bd)
```

Notons qu'ici on fait deux tests (x==y et x<y) pour une paire d'éléments mais on pourrait utiliser un unique teste renvoyant -1,0 ou 1.

Et en itératif, cela donne :

```
def recherche2(T,x) :
    global NBC
    NBC=0
    bg, bd = 0, len(T)-1
    while (bg <= bd) :
        m = (bg+bd)/2
        v = T[m]
        NBC += 1
        if (x==v) : return m
        else :
            NBC +=1
            if x<v : bd = m-1
            else : bg = m+1
    return None
```

Pour calculer la complexité de l'algorithme, on part de l'équation suivante où $C(n)$ dénote la complexité pour un tableau de taille n *dans le pire cas* :

$$C(n) \;=\; C(\lceil\frac{n}{2}\rceil) + O(1)$$

Et $C(1) = 1$. On obtient alors : $C(n) = O(\log_2 n)$.

Cet algorithme est vraiment crucial : sans lui pas d'annuaire, pas de dictionnaire... Avec 1 million d'entrées, on obtient une réponse à une recherche en comparant au plus 20 paires d'éléments.

Peut-on faire mieux ? Non... Pour le voir, on peut utiliser la même idée que celle utilisée pour la preuve de la borne inférieure des algorithmes de tri. On prend un arbre de décision où chaque noeud interne correspond à un test (*i.e.* une comparaison de deux éléments du tableau), et où chaque noeud interne correspond à un résultat possible (un indice ou -1).

DRAFT

Comme il doit y avoir au moins $n+1$ noeuds externes (donc au moins n noeuds internes), et comme la hauteur de l'arbre (*i.e.* la longueur d'un chemin maximal, c'est-à-dire le coût de l'algorithme dans un pire cas) est, *au mieux*, en $\log_2(\mathtt{nb}_{\text{noeud}})$, on en déduit que tout algorithme a une complexité asymptotique au moins égale à $O(\log n)$.

1.2 Tri fusion

Là encore, un grand classique…

```
# Fusion de deux listes triees:
def Fusion (L1,L2) :
    res = [ ]
    i1 = i2 = 0
    while i1<len(L1) and i2 < len(L2) :
        if L1[i1] < L2[i2] :
            res.append(L1[i1])
            i1 = i1+1
        else :
            res.append(L2[i2])
            i2 = i2+1
    if i1==len(L1) :
        res += L2[i2:]
    else :
        res += L1[i1:]
    return res


def TriFusion(T) :
    if len(T) > 1 :
        if len(T) == 2 :
            if T[0]>T[1] : T[0],T[1] = T[1],T[0]
        else :
            m = len(T)/2
            T = Fusion(TriFusion(T[:m]),TriFusion(T[m:]))
    return T
```

DRAFT

Il y a deux appels récursifs sur deux moitiés de tableau et l'opération `Fusion` se fait en temps linéaire, la complexité est donc définie par la récurrence :

$$C(n) \;=\; C(\lceil \frac{n}{2} \rceil) + C(\lfloor \frac{n}{2} \rfloor) + O(n)$$

Et $C(1) = 0$. Et finalement= $C(n) = O(n \cdot \log n)$.

1.3 Algorithme de Karatsuba

Cet algorithme date de 1960. Il a été publié en 1962 par Andreï Kolmogorov (1903–1987) qui pensait que l'algorithme classique en $O(n^2)$ était optimal jusqu'à ce qu'un étudiant, Anatolii Alexevich Karatsuba (1937–2008), propose un algorithme « diviser pour régner » une

semaine après un séminaire de Kolmogorov. Ce dernier a écrit et publié un article[1] dont Karatsuba n'a eu connaissance qu'à sa publication.

Le problème est le suivant :

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.

Si 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) : $\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i \cdot b_j \cdot r^{i+j}$. Si on prend $n = 2^{10} = 1024$, l'algorithme classique demande $2^{20} = 1.048.576$ multiplications, et autant (ou presque) de décalages et d'additions.

En fait, on peut « facilement » 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 clairement :

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

Si on en déduit naïvement un algorithme diviser-pour-régner, on fait quatre appels récursifs sur des problèmes de taille $\frac{n}{2}$ (pour calculer $\alpha_1 \cdot \beta_1$, $\alpha_0 \cdot \beta_1$, $\alpha_1 \cdot \beta_0$ et $\alpha_0 \cdot \beta_0$) et on obtient une complexité définie par la récurrence $C(n) = 4C(\frac{n}{2}) + O(n)$, ce qui entrainerait une complexité en $\Theta(n^2)$ comme l'algorithme naïf (non récursif) précédent. L'idée de Karatsuba consiste à faire 3 (et non 4) appels à des produits de taille $\frac{n}{2}$.
En effet ; on peut voir que K_1 s'obtient par :

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

et donc par un seul produit de taille $\frac{n}{2}$ en plus des deux nécessaires pour K_0 et K_2...

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

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

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

1. A. Karatsuba and Yu. Ofman (1962). « Multiplication of Many-Digital Numbers by Automatic Computers ». Proceedings of the USSR Academy of Sciences 145 : 293-294. Translation in Physics-Doklady, 7 (1963), pp. 595-596

DRAFT

Exemple. Prenons la base $r = 10$ et $n = 4$: $a = 1234$ et $b = 4321$.

Premier niveau : On doit résoudre les trois sous-problèmes $K_2 = 12 \times 43$, $K_0 = 34 \times 21$ et $K_1 = (12 + 34) \times (43 + 21) - K2 - K_0$.
Second niveau :

— $12 \times 43 : \begin{cases} 1 \times 4 = 4 & K_2^1 \\ 2 \times 3 = 6 & K_0^1 \\ (1+2)(4+3) - K_2^1 - K_0^1 = 21 - 4 - 6 = 11 \end{cases}$

$\rightarrow K_2 = 4 \cdot 10^2 + 11 \cdot 10 + 6 = 516$

— $34 \times 21 : \begin{cases} 3 \times 2 = 6 & K_2^2 \\ 4 \times 1 = 4 & K_0^2 \\ (3+4)(2+1) - K_2^2 - K_0^2 = 21 - 6 - 4 = 11 \end{cases}$

$\rightarrow K_0 = 6 \cdot 10^2 + 11 \cdot 10 + 4 = 714$

— $46 \times 64 : \begin{cases} 4 \times 6 = 24 & K_2^3 \\ 6 \times 4 = 24 & K_0^3 \\ (4+6)(6+4) - K_2^2 - K_0^2 = 100 - 24 - 24 = 52 \end{cases}$

$\rightarrow 24 \cdot 10^2 + 52 \cdot 10 + 24 = 2944$

D'où $K_1 = 2944 - 516 - 714 = 1714$.
Et finalement : $516 \cdot 10^4 + 1714 \cdot 10^2 + 714 = 5.332.114$

Algorithme. On peut écrire l'algorithme de Karatsuba comme ci-dessous. Attention : lorsqu'on écrit 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 coef de poids faible, `A[1]` celui de de b^1, *etc*.

```
def karatsuba(A,B) :
    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
```

DRAFT

1.4 Master Theorem

Remarque : il ne couvre pas tous les cas... La preuve se trouve dans « Introduction à l'algorithmique »[2]. D'autres variantes se trouvent dans « Éléments d'algorithmique »[3].

ThÃ©orÃ¨me 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 $t(n) = \Theta(f(n))$.*

La preuve de ce théorème consiste en deux parties : on montre le résultat lorsque n est une puissance de b, puis on l'étend aux autres entiers (en montrant que le passage aux $\lceil\ \rceil$ et $\lfloor\ \rfloor$ ne change pas la complexité asymptotique).

Exemples d'application. On peut maintenant reprendre les algorithmes vus précédemment :

- recherche dichotomique : $a = 1$, $b = 2$, $f(n) = O(1)$: $\log_2(1) = 0$, **cas 2** $\rightarrow t(n) = \Theta(\log n)$.
- tri fusion : $a = 2$, $b = 2$, $f(n) = O(n)$: $\log_2(2) = 1$, **cas 2** $\rightarrow t(n) = \Theta(n \cdot \log n)$.
- recherche min/max : $a = 2$, $b = 2$, $f(n) = O(1)$: $\log_2(2) = 1$, **cas 1** (avec un $\varepsilon \in]0; 1[$) $\rightarrow t(n) = \Theta(n)$.
- algorithme de Karatsuba : $a = 3$, $b = 2$, $f(n) = O(n)$: $\log_2(3) \sim 1.58$, **cas 1** (avec un $\varepsilon \in]0; \log_2 3 - 1[$) $\rightarrow t(n) = \Theta(n^{\log_2 3})$.

Ici on se contentera de prouver le théorème pour les valeurs de n correspondant à des puissances de b. On a donc le lemme suivant :

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

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

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 $t(n) = \Theta(f(n))$.*

DRAFT

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

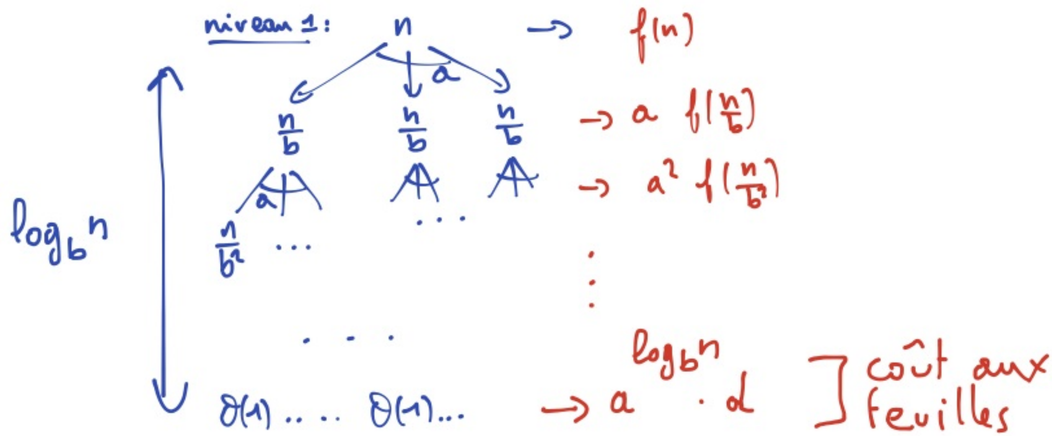

FIGURE 1 – Représentation des appels récursifs

DRAFT

Pour comprendre d'abord informellement le théorème, prenons un algorithme diviser pour régner qui, à chaque étape, fait a appels récursifs sur des problèmes de taille $\frac{n}{b}$ et qui, ensuite, en déduit une solution pour le problème de taille n en temps $f(n)$. On voit facilement que $t(n)$ représente le coût (la complexité) d'un tel algorithme. On peut alors représenter graphiquement ce que représente la quantité $t(n)$: considérons la Figure 1. A la racine, on a un problème de taille n, au niveau 1 on a a problèmes de taille $\frac{n}{b}$, au second a^2 problèmes de taille $\frac{n}{b^2}$, *etc.* La hauteur de l'arbre est $\log_b n$ et il y a donc $a^{\log_b n}$ feuilles. Maintenant la quantité $t(n)$ est clairement égale à la somme des complexités dues à la construction des solutions globales à chaque niveau : $f(n)$, puis $a \cdot f(\frac{n}{b})$, puis $a^2 \cdot f(\frac{n}{b^2})$, ..., à laquelle on ajoute la complexité des cas de base (coût constant d pour chaque feuille – $\Theta(1)$ – d'après la définition). Les trois cas du Master theorem vont correspondre à trois situations possibles : (1) le coût aux feuilles domine celui de la construction des solutions globales (car la fonction f est suffisamment « petite »), (2) le coût de la construction des solutions globale domine tout juste, et (3) c'est le coût de la construction des solutions globales qui domine nettement le reste...

En fait, on peut facilement représenter le coût total correspondant à $t(n)$ en

Preuve : Soit $n = b^p$, et donc : $p = \log_b n$. 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 $d = t(1)$. 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 :

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

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 $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$.

□

DRAFT