Diviser pour régner

MPSI - Prytanée National Militaire

Pascal Delahaye

25 mai 2018

1 Le principe "diviser pour régner"

La stratégie "diviser pour régner" consiste à découper la donnée que l'on doit traiter en plusieurs parties à peu près égales, puis à appliquer l'algorithme à l'une ou à chacune des parties avant de combiner les résultats obtenus pour construire le résultat correspondant à la donnée initiale. Selon les cas, la donnée pourra être :

1. un nombre (algorithme d'exponentiation rapide)
2. un intervalle (dichotomie)
3. une liste ou un tableau (algorithmes de tri)
4. un tableau 2x2(algorithme de multiplication de deux matrices)
5. un tableau ligne (algorithme de multiplication de deux polynômes)

1.1 Evaluation "grossière" de la complexité

Notons $c(n)$ le coût de traitement d'un objet de taille n.
Pour simplifier l'étude :

1. nous considèrerons qu'un objet de taille n est partitionné en plusieurs objets de taille $n/2$.
2. nous noterons $f(n)$ le coût total du partage et de la recombinaison.

Ainsi, nous avons :

$$\boxed{c(n) = a.c(n/2) + f(n)} \quad \text{avec } a \in \mathbb{N}^*$$

avec :

1. $a = 1$ lorsque l'algorithme n'est appliqué qu'à l'une des deux parties (exponentiation rapide, dichotomie)
2. $a = 2$ lorsque l'algorithme est appliqué à chacune des deux parties (tri par fusion)
3. $a = k$ plus généralement ... (voir les exemples traités en fin de poly ...)

Remarque 1. Pour des raisons pratiques de calcul, comme n est supposé grand, nous nous dispenserons souvent de distinguer les cas où n est pair ou impair.

1.2 Cas où n est une puissance de 2 ($n=2^{k+1}$)

On a alors : $c(2^{k+1}) = a.c(2^k) + f(2^{k+1})$ qui donne $\dfrac{c(2^{k+1})}{a^{k+1}} = \dfrac{c(2^k)}{a^k} + \dfrac{f(2^{k+1})}{a^{k+1}}$.

Soit $p \in \mathbb{N}^*$.
Après sommations et élimination des termes on obtient :

$$\boxed{\frac{c(2^p)}{a^p} = \frac{c(2)}{a} + \sum_{k=2}^{p} \frac{f(2^k)}{a^k}}$$

On étudie alors le comportement de la $\sum$ au voisinage de $+\infty$ selon la nature de la fonction f et de la valeur de a.
Voir le tableau de la partie 1.4

Remarque 2. La formule précédente permet alors de déterminer $c(2^p)$ sous la forme d'un O ou d'un équivalent, ce qui nous permettra de trouver un O ou un équivalent de $c(n)$.

`Exemple 1.` **Algorithme d'exponentiation rapide**

1. Principe de l'algorithme :
   (a) Si $n = 2p$: on remarque que $x^n = (x^p)^2$ et on se ramène donc à calculer x^p
   (b) Si $n = 2p + 1$: on remarque que $x^n = x.(x^p)^2$ et on se ramène donc là aussi à calculer x^p.

2. Formule du côut :
   Utilisons la multiplication comme unité de coût :
   (a) Si n est pair, on a alors $c(n) = c(n/2) + 1$
   (b) Si n est impair, on a alors $c(n) = c((n-1)/2) + 2$

3. Comportement asymptotique du coût : avec $n = 2^{k+1}$, on obtient $c(2^{k+1}) = c(2^k) + O(1)$
   (a) D'après la relation précédente, on obtient : $c(2^p) \leq c(2) + \sum_{k=0}^{p-2} \lambda$ avec $\lambda \in \mathbb{R}^{+*}$.
   (b) Ainsi, $c(2^{p+1}) \leq \mu p$ avec $\mu \in \mathbb{R}^{+*}$.

1.3 Cas général

$$\boxed{c(n) = a.c(n/2) + f(n)} \quad \text{avec } a \in \mathbb{N}^*$$

On admettra que la suite $c(n)$ est croissante, ce qui est cohérent avec le bon sens !

Etape 1 :
Soit $p \in \mathbb{N}$.
On se place dans le cas où $n = 2^p$ et on détermine un ordre de grandeur de $C(2^p)$ avec la méthode précédente.
On en déduit une majoration précise de la forme :

$$c(2^{p+1}) \leq G(p)$$

Etape 2 :
Soit $n \in \mathbb{N}$.
Il est possible d'encadrer n par deux puissances successives de 2 : $2^p \leq n \leq 2^{p+1}$ en prenant $p = \lfloor \dfrac{\ln n}{\ln 2} \rfloor$.
On a alors :

$$2^p \leq n \quad \text{et} \quad p \leq \frac{\ln n}{\ln 2}$$

Comme la fonction c est considérée croissante, alors on a :

$$c(2^p) \leq c(n) \leq c(2^{p+1})$$

Etape 3 :
On peut alors majorer la fonction $G(p)$ par une fonction de n et obtenir ainsi :

$$0 \leq c(n) \leq c(2^{p+1}) \leq G(p) \leq H(n) = O(K(n)) \quad \text{et donc} \quad c(n) = O(K(n))$$

`Exemple 2.` Prouver que l'algorithme d'exponentiation rapide, admet pour complexité : $c(n) = O(\ln n)$

1.4 Exemples théoriques

Déterminer la complexité des algorithmes basés sur le principe "diviser pour règner" dans les cas suivants :

Valeur de a (nbr d'appels récursifs)	1	1	2	2	3
Expression de f	$f(n) = O(1)$	$f(n) = O(n)$	$f(n) = O(1)$	$f(n) = O(n)$	$f(n) = O(1)$
Complexité de l'algorithme	$O(\ln n)$	AF	$O(n)$	$O(n \ln n)$	$O(n^{\log_2(3)})$

Remarque 3.
Plus généralement, on constate que si la complexité vérifie la relation $c(n) = a.c(\frac{n}{2}) + O(1)$ (avec $a \geq 2$) alors on a :

$$c(n) = O(n^{\log_2(a)})$$

Remarque 4.
Vous devez être capable de refaire les calculs précédents pour $a > 2$ et des fonctions f différentes.

2 situations classiques à retenir !

$$c(n) = 2c(\frac{n}{2}) + O(1) \quad \Rightarrow \quad c(n) = O(\ln n)$$
$$c(n) = 2c(\frac{n}{2}) + O(n) \quad \Rightarrow \quad c(n) = O(n \ln n)$$

`Exemple 3.` Calculer la complexité $c(n)$ d'un algorithme sachant que celle-ci vérifie la relation de récurrence :

$$c(n) = 2c(\frac{n}{2}) + O(n \ln n)$$

2 Exemple 1 : le tri rapide

Il s'agit de trier les entiers d'une liste l par ordre croissant. Cette méthode s'apparente à un algorithme du type *diviser pour règner* dans la mesure où l'idée est de commencer par décomposer la liste initiale en deux sous listes. Seulement, dans l'algorithme de tri rapide, le critère de division ne porte pas uniquement sur la taille.

Le principe du tri rapide est le suivant :

1. On considère le premier élément e de la liste l
2. On décompose le reste de la liste en deux listes l_1 et l_2 :
   (a) l_1 comprenant tous les entiers inférieurs à e
   (b) l_2 comprenant tous les entiers supérieurs ou égaux à e
3. Puis on recommence l'opération précédente sur les deux listes l_1 et l_2 jusqu'à obtenir une liste nulle.
4. Il s'agit enfin d'assembler dans l'ordre suivant les éléments l_1 - e - l_2.

1. Construire une première procédure récursive `partition` d'arguments l et e effectuant la partition d'une liste l en les deux listes l_1 et l_2 contenant, l'une (l_1) tous les éléments de l inférieur à e et l'autre (l_2) tous les éléments de l strictement supérieurs à e.
2. En déduire une procédure récursive `tri_rapide` effectuant le tri rapide d'une liste l.
3. Effectuer la preuve de correction et de terminaison de cet algorithme
4. Etude de la complexité du tri rapide.
   (a) Déterminer la complexité $c_1(n)$ (en nombre de comparaisons) de l'algorithme dans le cas où la liste est d'ores et déjà triée.
   Montrer que $c_1(n)$ correspond à une complexité maximale en vérifiant par récurrence sur n que $c(n) \leq c_1(n)$ où $c(n)$ est la complexité associée à une liste quelconque de longueur n donnée quelconque.
   (b) Prouver que la complexité en moyenne du tri rapide est un $O(n \ln n)$.

Preuve :
Pour des raisons pratiques de calcul, nous supposerons ici que l'algorithme s'applique à un tableau contenant l'image de $[\![1, n]\!]$ par l'une des $n!$ permutations possibles de $\mathfrak{S}_n$.
L'algorithme de tri rapide partage une liste l_n en deux sous listes l_{k-1} et l_{n-k} avec k pouvant prendre toutes les valeurs de $[\![1, n]\!]$ avec une probabilité de $1/n$.
Le coût moyen est alors donné par la formule :

$$C(n) = \frac{1}{n}\sum_{k=1}^{n}(C(k-1) + C(n-k) + n - 1) \quad \text{ou encore :} \quad C(n) = \frac{2}{n}\sum_{k=1}^{n-1} C(k) + n - 1$$

On élimine la $\sum$ de l'expression, on pose $u_n = \dfrac{C(n)}{n+1}$ puis on exprime $C(n)$ en fonction de $H_n = \displaystyle\sum_{k=1}^{n} \frac{1}{k}$.
On obtient alors : $C(n) = 2(n+1)H_n - 4n$ d'où : $C(n) \sim 2n \ln n$

Remarque 5. Le tri rapide est très apprécié en raison de la valeur de sa complexité moyenne qui est un $O(n \ln n)$.

3 Exemple 2 : le tri par fusion

Le principe général du tri par fusion est le suivant :

1. On divise en deux moitiés la liste à trier.
2. On tri chacune d'entre elles
3. On fusionne les deux moitiés obtenues pour reconstituer la liste complète triée.

1. Construction d'une fonction récursive de tri par fusion :
   (a) Construire une fonction récursive `divise` qui divise une liste donnée en deux sous-listes de longueur égale ou presque (si la longueur est impaire !).
   (b) Construire une fonction récursive `fusion` qui fusionne deux listes triées en une seule liste triée.
   (c) En déduire la fonction `tri_fusion`.
   (d) Développer les différentes étapes du tri de la liste `[2;5;4;3;1]` par la fonction précédentes.

2. Etude de la complexité :
   (a) On note $c(n)$ le nombre de comparaisons effectuées par la fonction `tri_fusion` pour trier une liste de longueur n. Montrer que $c(n)$ vérifie une récurrence du type *diviser pour règner*.
   (b) En déduire que la complexité moyenne et dans le pire des cas du tri par fusion est un $O(n \ln n)$.

Remarque 6. Bilan sur les tris (en prenant la comparaison comme Opération Significative) :

Tri par sélection	Tri par insertion	tri à bulle	tri rapide	tri fusion
$\sim \frac{n^2}{2}$	$\sim \frac{n^2}{4}$	$\sim \frac{n^2}{2}$	$O(n \ln n)$	$O(n \ln n)$

Remarque 7. Il existe un théorème affirmant que pour le tri d'une liste quelconque, il ne peut exister d'algorithme dont la complexité en terme de nombre de comparaisons soit inférieure à un $O(n \ln n)$.

4 Exemple 3 : Recherche dichotomique dans un tableau trié

Lorsque le tableau considéré est d'ores et déjà trié, on peut effectuer une recherche dichotomique en procédant de la façon suivante.

Principe de l'algorithme de recherche dichotomique :

On compare l'élément e recherché à la valeur médiane du tableau. Selon que e est plus grand ou plus petit que cette valeur, on recommence la même opération sur la première moitié ou la seconde moitié du tableau. On s'arrête une fois l'élément trouvé ou lorsque le tableau testé à une longueur nulle.

1. **Fonction itérative :**
   Construire une fonction itérative de recherche dichotomique dans un tableau déjà trié.

2. **Fonction récursive :**
   (a) Construire une fonction récursive `cherche` de recherche dichotomique qui permet de savoir si un élément e est compris entre les élément d'indice i et j d'un tableau v.
   (b) En déduire une fonction de recherche dichotomique d'un élément e dans un tableau v.

3. Etudier la complexité des algorithmes précédents.

Remarque 8. Intérêt de la recherche dichotomique :
1. Aucun si l'on a une unique recherche à effectuer puisqu'il faut commencer par trier le tableau et qu'un algorithme de tri est au mieux un $O(n \ln n)$.
2. En revanche on pourra l'utiliser si l'on a un grand nombre de recherches à effectuer.

5 Exemple 5 : Algorithme de Strassen

La multiplication de deux matrices 2×2 requiert 8 multiplications et 4 additions scalaires. Plus généralement, la multiplication de deux matrices $n \times n$ requiert un $O(n^3)$ opérations arithmétiques.
Le nombre de multiplications peut être réduit par une méthode utilisant le principe "diviser pour règner".

Principe de l'algorithme de Strassen :

Pour multiplier deux matrices M et N de $\mathfrak{M}_n(\mathbb{R})$, on commence par effectuer une décomposition des matrices en 4 blocs.

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad \text{et} \quad N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

On montre que la matrice produit vaut alors : $MN = \begin{pmatrix} -P_2 + P_4 + P_5 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{pmatrix}$ avec :

$P_1 = A(F - H)$ $P_4 = D(G - E)$ $P_7 = (A - C)(E + F)$
$P_2 = (A + B)H$ $P_5 = (A + D)(E + H)$
$P_3 = (C + D)E$ $P_6 = (B - D)(G + H)$

1. Quel est le nombre d'opérations arithmétiques effectuées sur les matrices ?
2. En appliquant le principe précédent récursivement (on supposera que n est une puissance de 2), calculer $c(n)$ le coût pour effectuer la multiplication de deux matrices $n \times n$.
   On donnera autant d'importance aux additions qu'aux multiplications.
3. Estimer le gain par rapport à une méthode naïve.

Remarque 9. Le meilleur algorithme de calcul du produit de deux matrices de taille n a actuellement une complexité de l'ordre de $n^{2,79..}$

6 Exemple 6 : Alg. de Karatsuba pour la multiplication de Polynômes

Cet algorithme donne une méthode rapide de multiplication de deux polynômes selon le principe "diviser pour régner".

Principe de l'algorithme de Karatsuba :

Si on note P et Q les deux polynômes à multiplier, n le plus grand des degrés et $m = E(n/2)$.

On décompose alors P et Q de la façon suivante : $\begin{cases} P = X^m P_1 + P_2 \\ Q = X^m Q_1 + Q_2 \end{cases}$ avec $\begin{cases} \deg P_2 \leq m \\ \deg Q_2 \leq m \end{cases}$.

En notant $\begin{cases} S_1 = P_1 Q_1 \\ S_2 = (P_1 - P_2)(Q_1 - Q_2) \\ S_3 = P_2 Q_2 \end{cases}$, on obtient : $\boxed{PQ = S_1 X^{2m} + (S_1 + S_3 - S_2)X^m + S_3}$

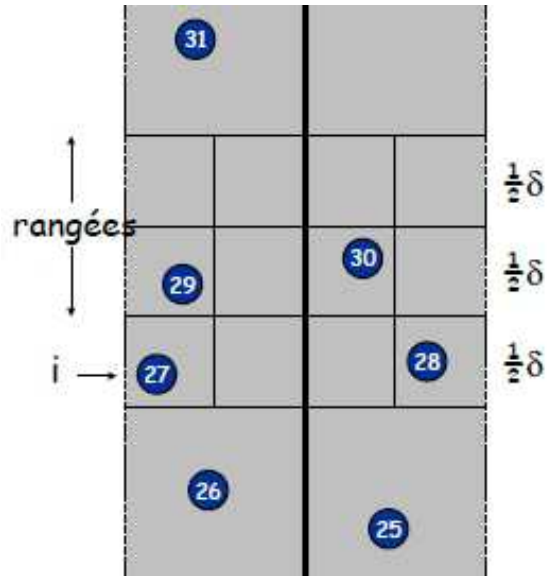
1. Expliquer le fonctionnement de l'algorithme.
2. Etudier sa complexité.

7 Exemple 7 : Distance minimale dans un nuage de points

Il s'agit ici de déterminer la plus petite distance entre deux points contenus dans un nuage de points donné sous la forme d'un tableau ligne de points triés selon leur abscisse.

Principe de l'algorithme :

1. On détermine la médiane M des abscisses des points du nuage (un algorithme de tri permet de le faire).
2. On répartit les points en deux sous-nuages : l'un contenant les points d'abscisse inférieure à M et l'autre les autres.
3. On applique la procédure aux deux nuages de points précédents et on considère d la distance minimale obtenue sur les deux nuages.
4. On considère alors un troisième nuage contenant tous les points d'abscisse comprise entre $M-d$ et $M+d$ (il suffit pour cela d'éliminer les autres points du nuage initial) et on regarde si ce nouveau nuage contient deux points dont la distance est inférieure à d.
5. Pour cela, on classe les points obtenus selon leur ordonnée et pour éviter de calculer inutilement trop de distances, on remarque (en examinant les différentes situations possibles) qu'il suffit de ne considérer que les distances entre un point et ses 5 successeurs dans la liste.



Une case ne peut contenir qu'un seul point !
Mais encore ?...

1. Développer les différentes étapes de cet algorithme.

2. Justifiez que si l'on quadrille le dernier nuage de points par des carrés de côté de longueur $d/2$, on est assuré que chaque carré peut contenir au plus 1 point du nuage.

3. En déduire pourquoi il suffit de ne considérer que les 11 points suivants un point donné dans le calcul des distances.
   On peut même prouver que 7 points suffisent.

4. Etudier la complexité de l'algorithme et la comparer à la complexité d'un algorithme utilisant la "force brute".

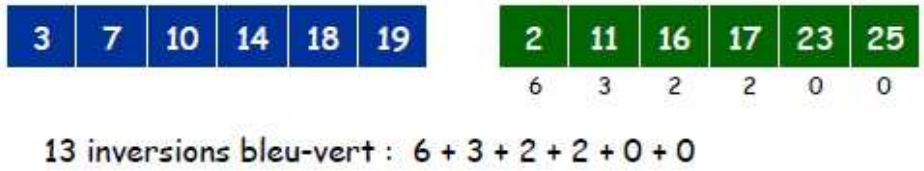
8 Exemple 8 : Compter le nombre d'inversions dans une liste d'entiers

Il s'agit ici de déterminer le nombre d'inversions contenues dans une liste de nombres entiers.
Cet algorithme est en particulier utilisé dans les cas suivants :

1. Comparaison des goûts de deux personnes ayant classés des objets par ordre de préférence
2. Mesure du degré de rangement d'un tableau
3. Analyse de la sensibilité du ranking de google

Principe général de l'algorithme :

1. On sépare le tableau en deux tableaux de taille égale (ou presque).
2. On applique la fonction pour compter le nombre d'inversions dans chacun des deux tableaux.
3. Il reste alors à déterminer le nombre d'inversions du type $(i,\ j)$ où i est un élément du premier tableau et j un élément du deuxième.
   (a) Pour cela, on pourrait comparer tous les couples possibles mais cela donnerait $\frac{n^2}{4}$ comparaisons et cela remettrait en cause la pertinence de notre algorithme.
   (b) On décide plutôt de :
   — commencer par ordonner les éléments de chacun des tableaux (complexité en $O(n \ln n)$), puis
   — par déterminer le nombre d'inversions à l'aide d'un algorithme de complexité $O(n)$.



Algorithme en $O(n)$ de dénombrement des inversions entre les 2 sous-tableaux triés

- On note n_1 la longueur du premier tableau T_1 et n_2 la longueur du premier tableau T_2.
- On souhaite compter le nombre d'inversion de type $(i,\ j)$ où $i \in [\![0, n_1 - 1]\!]$ et $j \in [\![0, n_2 - 1]\!]$.
- Pour j allant de 0 à $n_2 - 1$:
  On détermine la première valeur de i telle que $T_2(j) < T_1(i)$, c'est à dire telle que $(i,\ j)$ soit une inversion.
  Dans ce cas, le tableau T_1 étant trié, tous les couples $(k,\ j)$ tels que $k > i$ seront aussi des inversions.
  Nous aurons alors déterminer $n_1 - i$ inversions du type $(x,\ j)$, nombre à ajouter à un compteur S.
  Le tableau T_2 étant trié, on cherchera la valeur de i en partant de la valeur i précédemment trouvée car si $(x,\ j)$ n'était pas une inversion, alors $(x,\ j+1)$ ne le sera pas non plus.
- La complexité de cet algorithme est bien un $O(n)$ car la valeur de j prends n_2 valeurs et en tout, nous aurons n_2 comparaisons initiales $T_2(j) < T_1(i)$ auxquelles s'ajoutent au plus n_1 comparaisons dues à la recherche de la valeur de i.

1. Programmer cet algorithme en utilisant bien sûr la structure de tableau pour stocker la permutation étudiée.
2. Montrer que sa complexité est $O(n \ln^2 n)$ et comparer-la à la complexité d'un algorithme utilisant la "force brute".