

Résumé du chapitre 2 : Diviser-pour-régner

19 sept. 2002

2 Diviser-pour-régner

– Approche descendante à la résolution d'un problème :

- On *décompose* le problème à résoudre en sous-problèmes *plus simples*.
- On trouve la solution des sous-problèmes.
- On *combine* les solutions des sous-problèmes pour obtenir la solution du problème initial.

Cette approche conduit, de façon naturelle (mais pas nécessairement), à un algorithme récursif :

- Question : comment obtenir la solution des sous-problèmes?
- Réponse : en appliquant la même approche diviser-pour-régner descendante, et ce jusqu'à ce que le problème soit trivial.

Note importante : pour que l'approche diviser-pour-régner puisse conduire à un algorithme récursif, il faut que les sous-problèmes résultants soient *similaires* au problème initial. Si ce n'est pas le cas, on peut quand même considérer qu'on utilise une approche diviser-pour-régner, mais sans récursivité. (Voir section 2.8).

2.1 Fouille binaire

– Algorithme de la fouille binaire (recherche dichotomique) : pp. 48–49.

– Analyse de l'algorithme :

- Opération de base : comparaison d'un élément x avec $S[mid]$.
- Taille du problème : n , le nombre d'éléments du tableau.
- Hypothèse simplificatrice : $n = 2^k$, pour un certain $k \geq 0$.
- Type d'analyse : pire cas.
- Équations de récurrence définissant la fonction $W(n)$:
 - $W(1) = 1$
 - $W(n) = W(\frac{n}{2}) + 1$, pour $n > 1$, n une puissance de 2

- Solution informelle :

Niveau de l'appel	Taille du problème	Nombre total d'opérations à ce niveau
1		
2		
3		
...	...	
i		
...	...	
k		
$k + 1$		

Donc, complexité .

On verra un peu plus loin (rappel?) comment, en général, on peut résoudre des équations de récurrence.

2.2 Tri par fusion

- Objectif = trier un tableau d'éléments
- Idée générale du tri par fusion :
 - On divise le tableau (taille n) en deux-sous tableaux (taille $n/2$).
 - On trie (récursivement) les deux sous-tableaux.
 - On fusionne (*merge*) les sous-tableaux triés.
- Algorithme du tri par fusion et de la procédure de fusion : pp. 53-54 et 54-55.
- Analyse de la complexité de la fusion :
 - Opération de base : comparaison entre $U[i]$ et $V[j]$.
 - Taille du problème : h et m , le nombre d'items dans chacun des tableaux.
 - Type d'analyse : pire cas.
 - Le pire cas survient lorsque la fin d'un des tableaux est atteinte (par ex., $i = h$) alors qu'il ne reste qu'un seul élément de l'autre tableau à fusionner (par ex., $j = m - 1$).

$$W(h, m) = h + m - 1$$

- Analyse de la complexité du tri par fusion :
 - Opération de base : comparaisons effectuées dans la procédure de fusion.

- Taille du problème : n , le nombre d'éléments dans le tableau à trier.
- Type d'analyse : pire cas.
- Équations de récurrence pour $W(n)$:
 - $W(1) = 0$
 - $W(n) = 2W(\frac{n}{2}) + n - 1$, pour $n > 1$, n une puissance de 2

Dérivation de cette équation, en supposant que $n = 2^k$: p. 56.

- Solution informelle :

Niveau de l'appel	Taille du problème	Nombre d'opérations par sous-problème (division et combinaison)	Nombre total d'opérations (pour l'ensemble du niveau)
1			
2			
3			
...
i			
...
k			
$k + 1$			

Nombre total d'opérations :

Donc, l'algorithme est

Note : il est important de pouvoir faire, lorsque nécessaire, une telle analyse informelle de la complexité (même non exacte) ... parce que cela nous permet souvent de mieux comprendre le fonctionnement de l'algorithme (récursivité, nombre de niveaux d'appel, nombre de sous-tâches générées, etc.).

2.3 L'approche diviser-pour-régner ... dans le contexte de la programmation fonctionnelle

– Un langage de programmation *purement* fonctionnel — on dit aussi langage applicatif — est un langage basé uniquement sur l'utilisation de valeurs et de fonctions (au sens mathématique du terme, donc sans effet de bord), donc basé strictement *sur l'évaluation d'expressions*.

Exemples de tels langages : SML, Miranda, Haskell, Id/pH, sous-ensemble de Lisp/Scheme.

– Forme générale d’un programme fonctionnel = série de déclarations (constantes et fonctions), plus une expression à évaluer :

```
ident_1 = ...
ident_2 = ...
...
ident_k = ...
expression_a_evaluer
```

Note : une fonction, au sens mathématique du terme, n’est qu’une forme spéciale de constante!

– Dans un langage fonctionnel, les fonctions sont des valeurs manipulables comme toutes les autres (“citoyens de première classe”). Il est donc possible, et naturel, de transmettre des arguments à une fonction qui sont eux-mêmes des fonctions, de retourner un résultat qui est une fonction, de conserver une fonction dans une structure de données, etc.

– La programmation fonctionnelle est intéressante à cause de son style déclaratif, très près de ce qu’on écrirait en mathématiques. Il est donc plus facile de raisonner à propos d’un programme, c’est-à-dire, de déterminer les propriétés satisfaites par le programme) : on peut utiliser le raisonnement par *substitution* (raisonnement *équationnel*): “*equals can be replaced by equals*”, comme en mathématique.

– La plupart des langages fonctionnels modernes utilisent les séquences (listes) comme structures de données de base. Les algorithmes s’expriment donc souvent en termes de manipulations de listes.

– Dans un langage fonctionnel, l’utilisation de la stratégie diviser-pour-régner peut être exprimée de façon explicite et *générique* (donc favorisant la réutilisation) en définissant un groupe de fonctions appropriées.

```
diviserPourRegner
  estSimple           -- (Probleme -> Bool)           ->
  resoudreProblemeSimple -- (Probleme -> Solution)     ->
  decomposerProbleme  -- (Probleme -> [Probleme])     ->
  combinerSolutions   -- (Probleme -> [Solution] -> Solution) ->
  probleme            -- Probleme                    ->
                    -- Solution
= resoudreProbleme probleme
  where
  resoudreProbleme probleme
  | estSimple probleme = resoudreProblemeSimple probleme
  | otherwise          = combinerSolutions probleme sousSolutions
    where sousSolutions = map resoudreProbleme sousProblemes
          sousProblemes = decomposerProbleme probleme
```

Quelques exemples d'application:

```
quicksort :: [Int] -> [Int]
quicksort l = diviserPourRegner estVide vide partitionner combiner l
  where
    estVide l = l == []
    vide l = []
    partitionner (x : xs) = [ filter ((<=) x) xs,
                             [x],
                             filter ((>=) x) xs ]
    combiner probleme solutions = fold (++) [] solutions
```

```
fibonacci n = diviserPourRegner estCasBase un partitionner combiner n
  where
    estCasBase n = n <= 1
    un n = 1
    partitionner n = [n-1, n-2]
    combiner probleme solutions = fold (+) 0 solutions
```

2.4 Tri rapide (*Quicksort*)

– Algorithme célèbre développé par Hoare (1962).

– Son comportement dans le pire cas n'est pas très bon, mais en moyenne, en pratique sur des séquences typiques et en choisissant bien le pivot, son comportement est un des plus intéressants.

- Algorithme pour partitionnement : pp. 61–62.
- Complexité du partitionnement (tous les cas) : $T(n) = n - 1 = O(n)$.

Note : l'analyse du partitionnement peut être illustrée de deux façons :

1. En comptant, avec O , toutes les opérations mais en simplifiant à l'aide des propriétés de O .
2. En choisissant la comparaison comme opération de base.

En d'autres mots, dans la plupart des cas, le bon choix d'une opération de base rend l'analyse beaucoup plus facile.

- Algorithme pour tri rapide : p. 61
- Analyse du pire cas pour tri rapide :
 - Pire cas = le tableau est déjà trié
 - Équations définissant $T(n)$ si le tableau est déjà trié :
 - * $T(0) = 0$
 - * $T(n) = T(n - 1) + n - 1$, si $n > 0$.

– Solution par substitution :

$$\begin{aligned}T(n) &= T(n-1) + n - 1 \\&= (T(n-2) + n - 2) + n - 1 \\&= T(n-2) + 2n - 3 \\&= (T(n-3) + n - 3) + 2n - 3 \\&= T(n-3) + 3n - 6 \\&= \dots \\&= T(n-i) + in - (1 + 2 + \dots + i) \\&= \dots \\&= T(0) + n^2 - (1 + 2 + \dots + n) \\&= T(0) + n^2 - (n(n+1)/2) \\&= (n^2 - n)/2 \\&= n(n-1)/2\end{aligned}$$

Donc, le pire cas est $O(n^2)$.

Malgré tout, le tri rapide est intéressant. Ceci peut être montré par une analyse de la complexité moyenne (pp. 65–66). Ceci peut aussi être montré, informellement, de la façon illustrée dans l'exercice suivant.

– Question (variante améliorée du tri rapide) : Dans l'algorithme 2.7 (p. 62), le choix du pivot est fait à l'aide de l'instruction suivante :

```
pivotitem = S[low];
```

Supposons que l'on dispose d'une fonction `mediane` pouvant jouer un rôle d'*oracle* qui, en temps $O(f(n))$, puisse trouver l'élément médiane de `S`. Remplaçons alors la sélection du pivot par l'appel suivant :

```
pivotitem = mediane(S); // O(f(n))
```

Quelle serait alors la complexité du tri rapide (pire cas) en utilisant l'oracle `mediane`?

Rappel : La *médiane* m d'un groupe d'éléments S est l'élément tel qu'il y a autant d'éléments de S qui sont inférieurs à m que d'éléments qui sont supérieurs.

Réponse : L'équation de récurrence serait : ?

Il suffit que ? pour que la solution soit ?. Or, on verra ultérieurement qu'il est effectivement possible de trouver la médiane en temps ?.

Note : en pratique, le tri *Quicksort* est considéré comme l'un des plus rapides, et ce malgré son comportement dans le pire cas. Deux façons simples de voir que cette intuition est “raisonnable”, et ce sans entrer dans les détails mathématiques de calculs moyens ou autres, est de considérer les deux heuristiques suivantes pour le choix de l'élément pivot :

1. Approximation de l'oracle **mediane** : on calcule, en temps linéaire, une approximation de la médiane des éléments à trier.
2. Approximation aléatoire du comportement moyen : on sélectionne l'élément pivot *de façon aléatoire*.

Tri par fusion vs. tri rapide

Les algorithmes de tri par fusion et de tri rapide sont généralement présentés (avec la fouille binaire) comme les *archétypes* de l'approche diviser-pour-régner. Ces deux algorithmes se distinguent tout d'abord par leur complexité dans le pire cas ($O(n \lg n)$ vs. $O(n^2)$). Ils se distinguent aussi en fonction de l'espace requis pour exécuter chacun d'eux :

- Tri par fusion : nécessite de l'espace supplémentaire où la séquence fusionnée pourra être créée (la fusion *en place* n'est pas possible).
- Tri rapide : peut se faire *en place*.

Toutefois, dans le contexte de la présentation de l'approche diviser-pour-régner, ces deux algorithmes se distinguent particulièrement en termes des coûts associés aux différentes composantes (phases) de l'approche diviser-pour régner, et ce même si les deux satisfont (dans le meilleur cas) les mêmes équations de récurrence ($T(n) = 2T(n/2) + O(n)$) :

Algorithme	Coût de la décomposition en sous-problèmes	Coût de l'assemblage des sous-solutions	Coût au niveau de chaque sous-problème (noeud interne)
Tri par fusion			
Tri rapide fusion			

En d'autres mots, le gros du travail dans le cas de tri par fusion se fait au moment de la combinaison des solutions (la décomposition en sous-problèmes semblables au problème initial est triviale). Par contre, dans le tri rapide, c'est la décomposition en sous-problèmes qui est coûteuse alors que la combinaison des solutions est triviale.

On verra plus en détails (Annexe sur les équations de récurrence) les liens entre les coefficients et facteurs apparaissant dans les équations de récurrence et la façon dont la stratégie diviser-pour-régner est appliquée (coûts de la décomposition en sous-problèmes, résolution récursive des sous-problèmes, coûts de la combinaison/composition des sous-solutions).

2.5 Algorithme de Strassen pour la multiplication de matrices

– L’algorithme standard pour la multiplication de matrices est de complexité (tous les cas) $O(n^3)$. Toutefois, Strassen (1969) a conçu un algorithme qui, asymptotiquement, peut faire mieux.

– Stratégie utilisée par Strassen (p. 67) : la multiplication de matrices de taille 2×2 peut se faire à l’aide de sept (7) multiplications et 18 additions/soustractions plutôt qu’à l’aide de huit (8) multiplications et quatre (4) additions/soustractions.

– Faits :

- La multiplication est une opération plus coûteuse qu’une addition ou soustraction, tant pour les nombres entiers, réels, que pour les matrices.
- La stratégie de Strassen est valable lorsqu’elle est appliquée à des sous-matrices, donc pas seulement pour des matrices 2×2 .

– Algorithme de Strassen pour la multiplication de matrices : p. 69.

– Analyse de la complexité de l’algorithme de Strassen, en utilisant la multiplication comme opération de base :

- $T(n) = 7T(\frac{n}{2})$, pour $n > 1$, si n est une puissance de 2.
- $T(1) = 1$.

Solution (détails lors de la présentation de l’annexe sur les équations de récurrence) : $O(n^{\lg 7}) = O(n^{2.81})$.

Note : le même résultat asymptotique est obtenu si on utilise plutôt toutes les opérations arithmétiques (multiplication, addition, soustraction) comme opérations de base.

2.6 Arithmétiques à grande précision

Section omise.

2.7 Comment déterminer à quel moment cesser les appels récursifs

– Dans la mise en oeuvre directe de la stratégie diviser-pour-régner avec récursivité, on cesse la récursivité (la décomposition en sous-problèmes) lorsque le problème à résoudre est vraiment trivial, c’est-à-dire, ne peut plus du tout être décomposé (typiquement, de taille 1).

– Dans l’abstrait et l’idéal, les surcoûts (*overhead*) associés à la gestion des appels récursifs (par ex., allocation et copie des arguments sur la pile) peuvent être ignorés. En pratique, ces surcoûts peuvent devenir significatifs et il peut devenir plus efficace, à partir d’une certaine taille de problème, d’utiliser une approche asymptotiquement moins

efficace, mais avec un coefficient plus faible au niveau des surcoûts. La stratégie diviser-pour-régner *avec seuil* (avec coupure) devient donc, en gros, la suivante (en supposant une décomposition dichotomique) :

```
Solution resoudre_dpr( Probleme p )
{
  if (taille(p) <= TAILLE_MIN) {
    return( resoudre_algo_simple(p) ); // Solution non-recursive
  } else {
    Probleme p1, p2;
    (p1, p2) = decomposer(p);
    Solution s1 = resoudre_dpr(p1); // Appels recursifs
    Solution s2 = resoudre_dpr(p2);
    return( combiner(p, p1, p2) );
  }
}
```

La sélection de la valeur seuil (*threshold*) à partir de laquelle la récursivité doit se terminer dépend de nombreux facteurs :

- Mise en oeuvre exact de l'algorithme.
- Langage et compilateur utilisés.
- Machine et S.E. sur lesquels s'exécute le programme.
- Données sur lesquelles le programme est exécuté.

En d'autres mots, l'analyse *théorique* de l'algorithme ne suffit plus. On doit plutôt utiliser diverses techniques et outils plus pratiques, par exemple, on pourrait exécuter le programme avec différentes données et mesurer son temps d'exécution, utiliser l'outil `profile` (sur Unix/Linux) pour déterminer les fonctions et procédures les plus fréquemment appelées et déterminer où sont les points chauds, etc.

2.8 Quand ne pas utiliser l'approche diviser-pour-régner (avec récursivité)

Pour que l'approche diviser-pour-régner *avec* récursivité conduise à une solution efficace, il ne faut pas que l'une ou l'autre des conditions suivantes survienne :

1. Un problème de taille n est décomposé en deux ou plusieurs sous-problèmes eux-même de taille presque n (par ex., $n - 1$).
2. Un problème de taille n est décomposé en n sous-problèmes de taille n/c (pour une constante $c \geq 2$).

Rappelons aussi que l'approche diviser-pour-régner *avec récursivité* ne peut être utilisée que si les sous-problèmes sont *du même type* que le problème initial (par ex., le tri d'un tableau se fait en triant ses sous-tableaux). Toutefois, dans de nombreux cas, on peut dire qu'une approche diviser-pour-régner est utilisée, et ce même si aucune récursivité n'est nécessaire.

Exemple :

- Un fichier `commentaires.txt` contient des lignes de la forme suivante, où les différents noms peuvent ou non être distincts, et où les différentes lignes sont ordonnées selon la date (croissante) :

```
Date  Nom  Commentaire
Date  Nom  Commentaire
Date  Nom  Commentaire
...
```

On désire déterminer le nombre de noms distincts présents dans le fichier `commentaires.txt`.

- Algorithme non récursif utilisant une approche diviser-pour-régner (parce qu'il y a décomposition en sous-problèmes *plus simples* résolus de façon indépendante) :

```
DEBUT
  noms <- obtenir la liste des noms contenus dans le fichier
  noms_tries_et_uniques <- trier_unique(noms)
  RETOURNER( taille(noms_tries_et_uniques) )
FIN
```

- Mise en oeuvre (très simple) de cet algorithme sur Unix/Linux :

```
awk '{print $2;}' < commentaires.txt | sort -u | wc -l
```

On revient sur des exemples de ce style de décomposition lors de l'étude des stratégies de base de programmation parallèle.