Diviser-pour-régner

(Résumé du chapitre 2 du manuel)

Table des matières

2 Diviser-pour-régner **1**

2.1 Fouille binaire (dichotomique) 1
2.2 Tri par fusion (*mergesort*) 2
2.3 L'approche diviser-pour-régner 5
2.4 Tri rapide (*Quicksort*) 8
2.5 Algorithme de Strassen pour la multiplication de matrices 12
2.6 Arithmétiques à grande précision 13
2.7 Comment déterminer à quel moment cesser les appels récursifs 14
2.8 Quand ne pas utiliser l'approche diviser-pour-régner (avec récursivité) 14

A Procédures/fonctions récursives et équations de récurrence **16**

B Procédures récursives MPD avec tranches de tableaux **18**

B.1 Tranches de tableaux 18
B.2 Fouille binaire (dichotomique) 18
B.3 Tri par fusion 18

2 Diviser-pour-régner

– 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 obligatoirement), à un algorithme récursif :

- Question = comment obtenir la solution des sous-problèmes?
- Réponse = en appliquant la même approche diviser-pour-régner descendante aux sous-problèmes, et ce jusqu'à ce que le problème à résoudre 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 (dichotomique)

```
procedure trouverPosition( int A[*], int n, int v ) returns int pos
# PRECONDITION
#   SOME( k >= 0 :: n = 2^k ),
#   ALL( 1 <= i < n :: A[i] <= A[i+1] )
# POSTCONDITION
#   SOME( 1 <= i <= n :: A[i]  = v ) => (1 <= pos <= n) & A[pos] = v,
#   ALL ( 1 <= i <= n :: A[i] ~= v ) => pos = 0
{
  pos = trouverPosRec( A, 1, n, v );
}

procedure trouverPosRec( int A[*], int inf, int sup, int v ) returns int pos
{
  if (inf == sup) {
    if (A[inf] == v) {
      pos = inf;
    } else {
      pos = 0;
    }
  } else {
    int m = (inf+sup)/2;
    if (v <= A[m]) {
      pos = trouverPosRec(A, inf, m, v);
    } else {
      pos = trouverPosRec(A, m+1, sup, v);
    }
  }
}
```

Algorithme 1: Fouille binaire sur un tableau ordonné d'éléments

– Algorithme de la fouille binaire (recherche dichotomique) en notation MPD : Algorithme 1.

– Analyse de l'algorithme :

- Opération barométrique : comparaison de l'élément `v` avec un élément du tableau `A`.

  Justification : dans la partie `then`, cette comparaison est effectuée pour le même nombre de fois que les affectations à pos ; idem pour la partie `else` relativement à l'autre affectation. Vrai pour la combinaison de l'une et l'autre relativement à la comparaison des bornes.

- 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.

  Note : En fait, comme l'exécution de la procédure se termine nécessairement lorsque la taille de l'intervalle est d'un seul élément — il n'y a pas de test qui permet de terminer immédiatement la procédure si l'élément milieu est l'élément recherché — il s'agit plutôt d'une analyse de tous les cas.

- Équations de récurrence définissant la complexité asymptotique $W(n)$ du temps d'exécution :
  - $W(1) = 1$
  - $W(n) = W(\frac{n}{2}) + 1$, pour $n > 1$, n une puissance de 2
- Solution informelle :

Niveau de l'appel	Taille du problème	Nombre d'opérations barométriques à ce niveau
1	n	1
2	$n/2$	1
3	$n/4$	1
...	...	1
i	$n/2^{i-1}$	1
...	...	1
k	$n/2^{k-1}$	1
$k+1$	$n/2^k$	1

  Donc, complexité $\Theta(\lg n)$.

  On verra un peu plus tard (Annexe B du manuel) de quelle façon, en général, on peut *résoudre* des équations de récurrence.

2.2 Tri par fusion (*mergesort*)

– Objectif = trier un tableau d'éléments

– Idée générale du tri par fusion :

- On divise le tableau (de taille n) en deux-sous tableaux (de taille $n/2$).
- On trie (récursivement) les deux sous-tableaux.
- On fusionne (*merge*) les sous-tableaux triés.

```
procedure trier( ref int A[*], int n )
# PRECONDITION
#   SOME( k >= 0 :: n = 2^k )
# POSTCONDITION
#   A est une permutation de A',
#   ALL( 1 <= i < n :: A[i] <= A[i+1] )
{
  trierRec( A, 1, n );
}

procedure trierRec( ref int A[*], int inf, int sup )
{
  if (inf == sup) {
    # Rien a faire: deja trie.
  } else {
    # ASSERT( inf < sup )
    int mid = (inf + sup) / 2;
    trierRec ( A, inf,   mid );
    trierRec ( A, mid+1, sup );
    fusionner( A, inf, mid, sup );
  }
}

procedure fusionner( ref int A[*], int inf, int mid, int sup )
# PRECONDITION
#   inf <= mid <= sup,
#   ALL( inf   <= i < mid :: A[i] <= A[i+1] ),
#   ALL( mid+1 <= i < sup :: A[i] <= A[i+1] )
# POSTCONDITION
#   A[inf:sup] est une permutation de A'[inf:sup],
#   ALL( inf <= i < sup    :: A[i] <= A[i+1] )
{
  int C[inf:sup];  # Copie de travail pour fusion.
  int i1 = inf,    # Index pour la premiere moitie.
      i2 = mid+1,  # Index pour la deuxieme moitie.
      i  = inf;    # Index pour la copie.

  # On selectionne le plus petit element des deux moities en le copiant dans C.
  while( i1 <= mid & i2 <= sup ) {
    if (A[i1] < A[i2]) {
      C[i] = A[i1]; i1 += 1; i += 1;
    } else {
      # Autre facon d'ecrire l'affectation du prochain element.
      C[i++] = A[i2++];
    }
  }

  # ASSERT( i1 > mid | i2 > sup )
  # On transfere la partie non copiee dans C.
  for [k = i1 to mid] { C[i] = A[k]; i += 1; }
  for [k = i2 to sup] { C[i] = A[k]; i += 1; }

  # On recopie le tableau C dans A. On aurait pu ecrire: "A[inf:sup] = C;")
  for [i = inf to sup] { A[i] = C[i]; }
}
```

Algorithme 2: Tri par fusion

– Algorithme du tri par fusion et de la procédure de fusion : Algorithme 2.

Note : Dans une post-condition, une variable décorée d'un apostrophe, par ex., `x'`, dénote la valeur de la variable *avant* l'exécution de la procédure, c'est-à-dire, au moment de l'appel.

– Analyse de la complexité de la fusion (procédure `fusionner`) :

- Opération barométrique : affectation de `A[-]` vers `C[-]`.

  Justification : ces diverses affectations, pour un nombre non nul d'éléments, s'effectuent aussi souvent ou plus souvent que les initialisations au début de la procédure, aussi souvent que les comparaisons de la boucle `while`, et aussi souvent que les manipulations des variables d'index des boucles `for`.

- Taille du problème : nombre total d'items à fusionner dans les deux sous-listes, donc `sup-inf+1`.

- Type d'analyse : tous les cas.

  Soit n le nombre d'éléments qu'on veut fusionner, c'est-à-dire, $n =$ `sup-inf+1`. La complexité de la procédure `fusionner` sera donc linéaire :

$$T(n) = n \in \Theta(n)$$

Note : dans le manuel, une procédure légèrement différente de fusion est utilisée (Algorithme 2.3, p. 55). Cette procédure suppose que les éléments à trier sont reçus par l'intermédiaire de deux tableaux indépendants U et V, respectivement de taille h et m, et que le résultat est ensuite mis dans le tableau S. De plus, l'opération barométrique utilisée est celle de comparaison (dans la condition de la boucle **while**). Dans ce cas, il faut donc utiliser une analyse du pire cas (tous les cas possibles n'ont pas besoin du même nombre de comparaisons), analyse pour laquelle on conclut alors que la complexité est $W(h, m) = h + m - 1$.

– Analyse de la complexité du tri par fusion :

- Opération barométrique : opérations barométriques effectuées dans la procédure de fusion, dans la mesure où c'est là que le véritable travail est effectué.
- Taille du problème : n, le nombre d'éléments dans le tableau à trier.
- Type d'analyse : tous les cas.
- Équations de récurrence pour $T(n)$, en supposant que n est une puissance de 2 :
  - $T(1) = 0$
  - $T(n) = 2T(\frac{n}{2}) + n$, pour $n > 1$
- Solution informelle :

Niveau de l'appel	Taille du problème	Nombre d'opérations par sous-problème (division et combinaison)	Nombre de noeuds (de sous-problèmes) à ce niveau	Nombre total d'opérations barométriques (pour l'ensemble du niveau)
1	n	$n/2 + n/2 = n$	1	$1 \times (n) = n$
2	$n/2$	$n/4 + n/4 = n/2$	2	$2 \times (n/2) = n$
3	$n/4$	$n/8 + n/8 = n/4$	2^2	$2^2 \times (n/4) = n$
...	...	...	...	...
i	$n/2^{i-1}$	$n/2^{i-1}$	2^{i-1}	$2^{i-1} \times (n/2^{i-1}) = n$
...	...	...	...	...
k	$n/2^{k-1}$	$n/2^{k-1} - 1$	2^{k-1}	$2^{k-1} \times (n/2^{k-1}) = n$
$k+1$	$n/2^k$	0	2^{k-1}	$2^k \times 0 = 0$

Nombre total d'opérations :

$$\sum_{i=1}^{k} n = n \times k = n \lg n$$

Donc, l'algorithme est $\Theta(n \lg n)$.

Note : il est important de pouvoir faire, lorsque nécessaire, une telle analyse informelle (basée sur la structure de l'arbre, le nombre de noeuds, le travail effectué dans chaque noeud, etc.) 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

Les trois étapes de l'approche diviser-pour-régner :

1. Diviser un problème en sous-problèmes plus simples.
2. Résoudre les sous-problèmes.
3. Combiner les solutions des sous-problèmes pour obtenir la solution au problème de départ.

Dans certains cas, la majeure partie du travail se fait au niveau de la division en sous-problèmes (par ex., `quicksort`) alors que pour d'autres cas la majeure partie du travail se fait au niveau de la combinaison des solutions aux sous-problèmes (par ex., tri par fusion).

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, c'est-à-dire 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 à évaluer
```

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

utilier le raisonnement par *substitution* (raisonnement *équationnel*): "*equals can be replaced by equals*", comme en mathématique.

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

– De façon générique, la stratégie diviser-pour-régner peut être exprimée de la façon informelle (pseudocode) présentée à l'algorithme 3.

```
PROCEDURE resoudreDiviserPourRegner( p: Probleme ): Solution
  SI p est un problème simple ALORS
    sol <- on résout le problème simple p
  SINON
    (p1, ..., pk) <- on décompose le problème p en k sous-problèmes
     POUR i <- 1 A k FAIRE
       soli <- resoudreDiviserPourRegner( pi )
     FIN
     sol <- on combine les solutions sol1, ..., solk des sous-problèmes
  FIN
  RETOURNER( sol )
FIN
```

Algorithme 3: Diviser-pour-régner générique (décomposition en k sous-problèmes)

Code Haskell 1 Diviser-pour-régner générique en Haskell

```
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
```

– 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, tel que cela est illustré dans l'extrait de code Haskell 1.

Quelques explications concernant cette fonction :

- La fonction reçoit cinq arguments, les quatre premiers étant eux-mêmes des fonctions — les commentaires (partie à droite de "-") indiquent le type de l'argument correspondant :
  - `estSimple` : fonction qui détermine si un `Probleme` est simple ou non (type `Bool`).
  - `resoudreProblemeSimple` : fonction qui reçoit un `Probleme` simple et qui produit la `Solution` appropriée.

- `decomposerProbleme` : fonction qui reçoit un `Probleme` et qui retourne une *séquence* (une liste) de sous-problèmes associés (les symboles "`[X]`" dénotent un type séquence dont les éléments sont de type `X`).
- `combinerSolutions` : fonction qui reçoit en argument le `Probleme` initial, une séquence de `Solutions` aux sous-problèmes, et qui produit la `Solution` globale.

- Le corps de la fonction `diviserPourRegner` consiste en une définition (locale) de la fonction `resoudreProbleme` combinée avec un appel à cette fonction. C'est cette fonction qui réalise, en Haskell, la logique décrite à l'algorithme 3.

  Notons que la fonction `map` est une fonction pré-définie du langage qui reçoit en arguments une fonction et une liste et qui retourne une liste résultant de l'application de la fonction à chacun des éléments de la liste. Quelques exemples :

```
map fois2 [10, 20, 30] => [20, 40, 60]
map plus1 [10, 20, 30] => [11, 21, 31]
map (plus 10) [1, 2, 3] = [11, 12, 13]
map additionner [(10, 20), (11, 21), (3, 20)] = [30, 32, 23]

fois2 x = 2 * x
plus x y = x + y
plus1 x = plus 1 x        -- Autre facon: plus1 = plus 1
additionner (x, y) = x+y
```

  Dans l'extrait de code Haskell 1, la fonction `map` applique donc la fonction `resoudreProbleme` à chacun des sous-problèmes de la liste `sousProblemes` et retourne une liste des `sousSolutions`.

Code Haskell 2 Quelques applications de la procédure `diviserPourRegner` générique

```
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


fibo 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
```

L'extrait de code Haskell 2 présente quelques exemples d'application de la procédure générique `diviserPourRegner` :

- Une fonction de tri de type `quicksort`. On verra plus en détail cet algorithme à la prochaine section.
- Une fonction pour calculer le nième nombre de Fibonnaci.

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 intéressant.

- Algorithme pour partitionnement (décomposition en deux sous-séquences dont les éléments sont, respectivement, inférieurs ou égaux, et supérieurs) : Algorithme 4, procédure `partitionner`.

  Complexité du partitionnement (tous les cas) : $T(n) = n - 1 = \Theta(n)$.

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

  1. En comptant *toutes* les opérations mais en simplifiant à l'aide des propriétés de Θ.
  2. En choisissant la comparaison avec l'élément pivot comme opération barométrique.

  Dans la plupart des cas, le bon choix d'une opération barométrique (alternative 2.) rend donc l'analyse beaucoup plus simple. Ici, la comparaison `A[i] <= pivot` est bien une opération barométrique puisque cette comparaison est exécutée au moins aussi souvent que n'importe quelle autre instruction (en se restreignant aux cas asymptotique, c'est-à-dire, lorsque le nombre d'éléments est grand) :

  – La comparaison va se faire plus souvent que l'initialisation du pivot.
  – La comparaison va se faire aussi souvent, ou plus souvent, que chacune des instructions du `if` dans la boucle `for`.
  – La comparaison va se faire aussi souvent que les opérations élémentaires associées à la manipulation de la variable d'itération.
  – La comparaison va se faire plus souvent que l'échange final.

- Algorithme pour tri rapide : Algorithme 4.

  Analyse du pire cas pour tri rapide :

  – Pire cas = le tableau est déjà trié. Dans ce cas, l'élément choisi comme pivot est le plus petit élément et la partie gauche (premier appel à `trierRec` sur l'intervalle `inf` à `posPivot-1`) est alors vide alors que la partie droite (appel récursif sur l'intervalle `posPivot+1` à `sup`) contient tous les éléments sauf le plus petit, donc contient $n - 1$ éléments.
  – Équations définissant $T(n)$ si le tableau est déjà trié (en comptant l'opération barométrique de comparaison avec le pivot) :
    - $T(0) = 0$
    - $T(1) = 0$
    - $T(n) = T(n-1) + n - 1$, si $n > 1$.
  – Solution par substitution :

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \\ &= [T(n-2) + n - 2] + n - 1 \\ &= T(n-2) + (n-2) + (n-1) \\ &= T(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \end{aligned}$$

```
procedure partitionner( ref int A[*], int inf, int sup, res int posPivot )
# PRECONDITION
#   inf < sup
# POSTCONDITION
#   A est une permutation de A',
#   inf <= posPivot <= sup,
#   ALL( inf      <= i <= posPivot :: A[i]        <= A[posPivot] ),
#   ALL( posPivot < i  <= sup      :: A[posPivot] <  A[i]        )
{
  posPivot = inf;
  int pivot = A[posPivot];
  for [i = inf+1 to sup] {
    if (A[i] <= pivot) {
      posPivot += 1;
      A[i] :=: A[posPivot];  # Interchange les deux elements.
    }
  }
  # On deplace le pivot a sa bonne position.
  A[posPivot] :=: A[inf];    # Interchange les deux elements.
}


procedure trierRec( ref int A[*], int inf, int sup )
{
  if (inf >= sup) {
    # Tableau vide ou de taille 1: rien a faire
  } else {
    int posPivot;
    partitionner( A, inf, sup, posPivot ); # On partitionne A en deux parties.

    # ASSERT( A[posPivot] est a la bonne position )
    trierRec( A, inf, posPivot-1 );        # On effectue le tri de la partie gauche
                                           # (elements inferieurs ou egaux).
    trierRec( A, posPivot+1, sup );        # Idem pour la partie droite
                                           # (elements strictement superieurs).
  }
}


procedure trier( ref int A[*], int n )
{
  trierRec( A, 1, n );
}
```

Algorithme 4: Tri *quicksort*

$$
\begin{aligned}
&= T(n-i) + (n-i) + (n-i+1) + \ldots + (n-2) + (n-1) \\
&= \ldots \\
&= T(1) + 1 + 2 + \ldots + (n-2) + (n-1) \\
&= 0 + 1 + 2 + \ldots + (n-2) + (n-1) \\
&= \sum_{i=0}^{n-1} i \\
&= \frac{n(n-1)}{2}
\end{aligned}
$$

Donc, le pire cas est bien $\Theta(n^2)$.

– Malgré le résultat obtenu pour l'analyse du pire cas, le tri rapide est généralement considéré comme un tri intéressant. De façon relativement rigoureuse, ceci peut être montré en utilisant une analyse de la complexité *moyenne* (pp. 65–66 du manuel).

Ceci peut aussi être montré, de façon plus informelle, tel que cela est illustré dans les paragraphes qui suivent.

Variante améliorée du tri rapide

Dans l'algorithme 4, le choix du pivot dans la procédure `partitionner` est fait à l'aide des instructions suivantes :

```
posPivot = inf;
int pivot = A[posPivot];
```

La boucle `for` qui suit immédiatement le choix du pivot peut alors débuter son exécution sous la condition que le pivot est initialement à la position *inférieure* du tableau.

Supposons maintenant que l'on dispose d'une fonction `posMediane` jouant un rôle d'*oracle* et qui, en temps $\Theta(f(n))$, peut trouver *la position* de l'élément médiane de `A`.[1] Remplaçons alors les deux instructions mentionnées plus haut par la suite d'instructions suivantes :

```
# On identifie l'element mediane et on le selectionne comme pivot.
posPivot = posMediane( A, inf, sup );
int pivot = A[posPivot];
# On ramene l'element pivot au debut du tableau.
A[inf] :=: A[posPivot];
posPivot = inf;
```

En supposant que les deux moitiés sont effectivement réparties de façon égale (à cause du choix de la médiane comme pivot), on obtiendrait alors les équations de récurrence suivantes pour décrire la complexité asymptotique du tri rapide[2] :

- $T(1) = \Theta(1)$
- $T(n) = 2T(\frac{n}{2}) + \Theta(f(n)) + \Theta(n) + \Theta(1)$

  (temps pour obtenir la médiane, suivi du temps pour le partitionnement, suivi du temps pour la combinaison des deux parties triées, en plus du temps pour les appels récursifs)

[1] Rappelons que l'élément médiane est celui qui, dans une liste ordonnée des éléments, se trouve au milieu, c'est-à-dire qu'il y a autant d'éléments qui lui sont inférieurs qu'il y en a qui lui sont supérieurs.

[2] En fait, pour être exact, l'équation de récurrence devrait plutôt être la suivante :

- $T(n) = T(\frac{n}{2}) + T(\frac{n}{2} - 1) + \Theta(f(n)) + \Theta(n) + \Theta(1)$

Or, par monotonicité, on a que $T(\frac{n}{2} - 1) \leq T(\frac{n}{2})$. On devrait donc plutôt conclure que $T(n) \in O(n \lg n)$. Toutefois, pour simplifier la présentation, on ignore cette petite subtilité.

Or, on peut voir (cf. analyse du tri par fusion) qu'il suffit alors que $f(n) \in \Theta(n)$ pour que l'algorithme de tri rapide résultant soit $O(n \lg n)$ — en d'autres mots, il suffit de s'assurer que le temps pour le calcul de la médiane *ne domine pas* celui requis pour le partitionnement.

La question qui se pose alors est de savoir s'il est possible de déterminer la médiane *en temps linéaire*. En d'autres mots, est-il possible de réaliser, ou même simplement approximer, le comportement de l'*oracle* calculant la médiane? La réponse à cette question est positive :

1. En fait, il existe un algorithme (section 8.5 du manuel) qui permet effectivement de déterminer, en temps linéaire, la médiane d'un groupe d'éléments.

2. Une autre façon d'obtenir une *approximation* de la médiane pourrait être d'utiliser un algorithme de type Las Vegas, c'est-à-dire, un algorithme *aléatoire* (probabiliste). Par exemple, on pourrait choisir, de façon aléatoire, un certain nombre fixe d'éléments du tableau (par ex., cinq éléments choisis au hasard) et ensuite identifier, en temps constant (puisqu'on a un nombre fixe d'éléments), la médiane parmi ces cinq éléments. La valeur espérée serait alors une approximation de la médiane qui ferait en sorte que, en moyenne, les deux sous-tableaux générés lors du partitionnement seraient de tailles équivalentes.

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 : $O(n \lg n)$ par opposition à $O(n^2)$. Ils se distinguent aussi en fonction de l'espace requis pour exécuter chacun d'eux :

- Tri par fusion : nécessite de l'espace supplémentaire ($\Theta(n)$) où la séquence fusionnée (`C`) pourra être créée — la fusion *en place* n'est pas possible.
- Tri rapide : peut se faire *en place*, donc ne demande aucun espace mémoire additionnel (sauf pour les variables locales).

Dans le contexte de la présentation de l'approche diviser-pour-régner, ces deux algorithmes se distinguent plus 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 (choix approximatif de la médiane comme pivot), les mêmes équations de récurrence, à savoir :

- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, si $n > 1$
- $T(1) = \Theta(1)$

Algorithme (coût pour *chaque* appel)	Coût de la décomposition en sous-problèmes	Coût de l'assemblage des sous-solutions	Coût total pour la partie non-récursive
Tri par fusion	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Tri rapide	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$

En d'autres mots, le gros du travail dans le cas de tri par fusion se fait au moment de la combinaison des solutions (fusion des parties déjà triées), alors que la décomposition en sous-problèmes semblables au problème initial est trivial (on divise en deux parties de taille équivalente, indépendamment du contenu, en utilisant simplement l'index milieu). Par contre, dans le tri rapide, c'est la décomposition en sous-problèmes qui est coûteuse (partitionnement en deux parties comportant les éléments inférieurs vs. supérieurs au pivot) alors que la combinaison des solutions est triviale (les deux parties sont déjà triées et dans le bon ordre).

On verra plus en détails (Annexe B du manuel 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 $\Theta(n^3)$ (tous les cas), la multiplication naïve de deux matrices s'effectuant à l'aide de trois boucles imbriquées :

```
for [i = 1 to n, j = 1 to n]  {
  c[i,j] = 0
  for [k = 1 to n] {
     c[i,j] += a[i,k] * b[k,j]
  }
}
```

Toutefois, Strassen (1969) a conçu un algorithme qui, *asymptotiquement*, peut faire mieux.

La stratégie utilisée par Strassen (p. 67 du manuel) est basée sur l'idée suivante : 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.

Ainsi, soit les matrices A, B et C de taille 2×2 définies comme suit :

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Strassen a montré que la matrice C peut être obtenue de la façon suivante. Soit les valeurs m_i $(i = 1, \ldots, 7)$ définies comme suit :

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) \\ m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

La matrice C peut alors être définie comme suit :

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

L'autre idée importante de l'algorithme de Strassen est que cette stratégie de réarrangement des opérations arithmétiques peut aussi être utilisée, *récursivement*, pour des sous-matrices :

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

- Fait : La multiplication est une opération plus coûteuse qu'une addition ou soustraction, et ce autant pour les nombres entiers ou réels que pour les matrices.

- Corollaire : il est avantageux de minimiser, autant qu'il est possible de le faire, le nombre de multiplications.

– Une description informelle de l'algorithme de Strassen pour la multiplication de matrices serait donc telle que présentée à l'algorithme 5 (les sommes et différences de matrices étant simplement représentées à l'aide des opérateurs binaires usuels).

```
PROCEDURE produitStrassen( A, B: Matrice,  n: Nat ): Matrice
# PRECONDITION
#   SOME( k >= 0 :: n = 2^k )
DEBUT
  C <- new Matrice[n,n]
  SI n == 1 ALORS
    C[1,1] <- A[1,1] * B[1,1]
  SINON
     // Décomposition récursive du problème
    (A11, A12, A21, A22) <- décomposer la matrice A en 4 sous-matrices de taille n/2
    (B11, B12, B21, B22) <- décomposer la matrice B en 4 sous-matrices de taille n/2
     // Appels récursifs
     M1 <- produitStrassen( A11+A22, B11+B22, n/2 )
     M2 <- produitStrassen( A21+A22, B11, n/2 )
     ...
     M7 <- produitStrassen( A12-A22, B21+B22, n/2 )
     // Combinaison des solutions aux sous-problèmes
     C11 <- M1+M4-M5+M7
     C12 <- M3+M5
     C21 <- M2+M4
     C22 <- M1+M3-M2+M6
     C <- composer les sous-matrices C11, ..., C22
  FIN
  RETOURNER( C )
FIN
```

Algorithme 5: Algorithme de Strassen (description informelle)

– Analyse de la complexité de l'algorithme de Strassen, en utilisant la multiplication de nombres comme opération barométrique :

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

Solution (cf. Annexe B du manuel sur les équations de récurrence) :

$$T(n) \in \Theta(n^{\lg 7}) = \Theta(n^{2.81})$$

Note : en fait, on obtient le même résultat asymptotique si on utilise plutôt les opérations arithmétiques d'addition et soustraction comme opérations élémentaires. Dans ce cas, les équations de récurrence sont plutôt le suivantes :

- $T(n) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$, pour $n > 1$, si n est une puissance de 2.
- $T(1) = 0$.

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 récursion (la décomposition en sous-problèmes) lorsque le problème à résoudre est vraiment trivial, c'est-à-dire qu'il ne peut plus du tout être décomposé (typiquement, de taille 1).

– Dans l'abstrait, 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 alors, en gros, celle illustrée à l'algorithme 6 (en supposant une décomposition *dichotomique*, c'est-à-dire en deux sous-problèmes) :

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

Algorithme 6: Diviser-pour-régner dichotomique (deux sous-problèmes) avec seuil pour terminer la récursion

Le choix du seuil (*threshold*) à partir duquel 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 s'exécute.

En d'autres mots, l'analyse *théorique* de l'algorithme *ne suffit plus*. On doit plutôt utiliser diverses techniques et outils pratiques et empiriques, par exemple, on pourrait exécuter le programme avec différentes données et mesurer son temps d'exécution, ou bien utiliser l'outil `gprof` (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-mêmes 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 $c \geq 2$).

Dans l'un ou l'autre de ces cas, le temps d'exécution résultant est alors de complexité exponentiel ou factoriel, ce qui rend donc l'algorithme tout à fait inefficace et inutilisable, sauf pour de (très) petites valeurs de n. (Pour s'en convaincre, il suffit d'identifier et d'analyser les équations de récurrence associées à des telles décompositions récursives.)

Et peut-on utiliser diviser-pour-régner... sans récursivité?

L'approche diviser-pour-régner *avec récursivité* ne peut être utilisée que si les sous-problèmes qui sont générés lors de la décomposition du problème initial *sont du même type* que le problème initial (par ex., le tri d'un tableau se fait en triant ses sous-tableaux). Toutefois, il existe de nombreux problèmes où on peut considérer qu'une approche diviser-pour-régner est utilisée, *et ce même si aucune récursivité n'est nécessaire ou utile.*

Par exemple, supposons qu'on ait le problème suivant :

- Entrée : 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
  ...
  ```

- Sortie : On désire obtenir le nombre de noms *distincts* qui sont présents dans le fichier `commentaires.txt`.

On peut dire que l'algorithme non récursif suivant utilise une stratégie diviser-pour-régner, puisqu'il y a décomposition du problème initial en sous-problèmes *plus simples*, lesquels sous-problèmes sont résolus de façon indépendante :

```
DEBUT
  noms <- obtenir la liste des noms contenus dans le fichier commentaires.txt
  noms_tries_et_uniques <- trier (de façon unique) les noms
  nb <- nombre d'éléments dans noms_tries_et_uniques
  RETOURNER( nb )
FIN
```

En fait, on peut dire que la stratégie diviser-pour-régner, sans récursivité, est la base même de la décomposition fonctionnelle.

Finalement, il est intéressant de noter que cet algorithme peut être mis en oeuvre de façon *très* simple sur une machine Unix/Linux, et ce au niveau même du *shell* :

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

On reviendra ultérieurement sur des exemples de ce style de décomposition lors de l'études des stratégies de base de programmation parallèle.

A Procédures/fonctions récursives et équations de récurrence

De façon générale, l'analyse de la complexité d'un algorithme récursif génère des équations de récurrence dont la structure ressemble à celle du flot de récursion de l'algorithme. Ces équations de récurrence donnent une définition *implicite* de la complexité. Il faut ensuite *résoudre* ces équations de récurrence (voir annexe B pour plus de détails) pour obtenir une solution nous donnant, de façon explicite, la complexité de l'algorithme.

Décomposition dichotomique

Soit l'algorithme 7 (diviser-pour-régner avec décomposition dichotomique, donc en deux sous-problèmes), où le temps pour chaque partie de l'algorithme est indiqué en commentaire.

```
PROCEDURE proc( p: Probleme ): Solution
DEBUT
  SI problème simple(p) ALORS // Θ(t1)
    sol <- Solution pour résoudre problème simple(p)  // Θ(t2)
  SINON
    (p1, p2) <- Décomposer problème(p) // Θ(t3)
    sol1 <- proc(p1)
    sol2 <- proc(p2)
    sol <- Combiner solutions(sol1, sol2) // Θ(t4)
  FIN
  RETOURNER( sol )
FIN
```

Algorithme 7: Algorithme générique diviser-pour-régner avec décomposition dichotomique

Supposons ensuite que les diverses conditions caractérisant cet algorithme sont les suivantes :

1. la taille du problème est n ;
2. un problème est considéré *simple* si sa taille est inférieure ou égale à c ;
3. le temps pour déterminer si un problème est simple est $\Theta(t_1)$ et le temps pour résoudre un problème simple est $\Theta(t_2)$;
4. le temps pour décomposer le problème initial en deux sous-problèmes est $\Theta(t_3)$;
5. la taille de chacun des sous-problèmes est $n/2$ (décomposition dichotomique *équilibrée*) ;
6. le temps pour combiner les deux solutions des sous-problèmes et obtenir la solution globale est $\Theta(t_4)$;

Sous ces conditions, cet algorithme donnerait alors lieu aux équations de récurrence suivantes :

$$\begin{array}{lcll} T(n) & = & \Theta(t_1) + \Theta(t_2) & n \leq c \\ T(n) & = & \Theta(t_1) + \Theta(t_3) + 2T(n/2) + \Theta(t_4) & n > c \end{array}$$

Simplification des équations de récurrence

La forme précédente est une forme générale ... qui, si possible, *devrait être simplifiée* pour obtenir la forme finale des équations . Il est particulièrement important de faire les simplifications qui s'imposent *avant* de tenter de résoudre de telles équations (y compris si on utilise l'un des théorèmes généraux présentés à l'annexe B).

Par exemple, supposons que la forme générale de l'équation pour le cas récursif soit la suivante :

$$T(n) \quad = \quad \Theta(1) + \Theta(n^2) + 2T(n/2) + \Theta(1) \quad \text{pour } n > 1$$

En utilisant la règle des maximums pour les sommes, on devrait alors simplifier cette équation pour obtenir la forme finale suivante :

$$T(n) \quad = \quad 2T(n/2) + \Theta(n^2) \quad \text{pour } n > 1$$

Cas typique de la décomposition jusqu'au cas de base trivial

Notons que dans le cas typique d'une décomposition récursive jusqu'au cas de base trivial (problème de taille 1), le coût pour déterminer si un problème est simple et résoudre un tel problème simple est généralement $\Theta(1)$. Les équations de récurrence associées seraient alors les suivantes :

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= \Theta(t_3) + 2T(n/2) + \Theta(t_4) \quad n > 1 \end{aligned}$$

Décomposition avec récursion linéaire

Une autre forme typique de récursion est celle qui génère une structure *linéaire* de récursion, c'est-à-dire qu'à partir d'un problème initial de taille n, on génère un sous-problème plus simple de taille $n-1$.

Soit l'algorithme 8 (diviser-pour-régner avec récursion linéaire), où le temps pour certaintes parties de l'algorithme est indiqué en commentaire.

```
PROCEDURE proc( p: Items[*], n: Nat ): Solution
DEBUT
  SI n == 1 ALORS
    sol <- Solution pour p[1]
  SINON
    sol' <- proc(p[1:n-1]', n-1)
    sol <- Combiner solution sol' avec p[n] // Θ(t1)
  FIN
  RETOURNER( sol )
FIN
```

Algorithme 8: Algorithme générique diviser-pour-régner avec récursion linéaire

Cet algorithme donnerait alors lieu aux équations de récurrence suivantes :

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(n-1) + \Theta(t_1) \quad n > 1 \end{aligned}$$

B Procédures récursives MPD avec tranches de tableaux

Dans l'approche diviser-pour-régner, les sous-problèmes qui sont générés par la décomposition du problème initial doivent être *semblables* au problème initial, et ce de façon à pouvoir utiliser la récursion.

Dans plusieurs des exemples vus précédemment (sections 2.1, 2.2 et 2.4), les procédures récursives qui ont été définies de façon à obtenir des sous-problèmes semblables appropriés l'ont été par le biais *d'arguments supplémentaires*. Typiquement, ces arguments (`inf` et `sup`) indiquaient les bornes inférieure et supérieure représentant le sous-problème.

En MPD, il est aussi possible, comme nous le verrons dans les exemples qui suivent, de définir directement des procédure récursives manipulant des tableaux *sans nécessairement introduire de tels arguments supplémentaires*. Ceci peut se faire en utilisant des *tranches de tableaux*.

B.1 Tranches de tableaux

Soit `A` un tableau d'entiers introduit par la déclaration "`int A[inf:sup]`".[3] Les bornes inférieure et supérieure de `A` sont alors respectivement `inf` et `sup`. Une *tranche* du tableau `A` (en anglais, *a slice*) est alors dénotée par une expression de la forme "`A[i:j]`", où `i` et `j` sont compris (inclusivement) entre `inf` et `sup`. Cette tranche représente alors le sous-tableau (suite d'éléments de `A`) dénoté par l'intervalle indiqué. Une telle tranche peut alors être affectée à un autre tableau (ou tranche de tableau) de même taille, passée en paramètre (en mode `val`, ou `var`, mais pas en mode `ref`), etc.

B.2 Fouille binaire (dichotomique)

L'algorithme 9 présente une version récursive de la fouille binaire utilisant des tranches de tableau. Signalons, dans la branche `else`, l'ajout de la valeur `n/2` au résultat retourné par la fonction, pour tenir compte du fait que la recherche s'effectue bien dans la partie droite du tableau.

Signalons aussi que l'argument `A` étant déclaré avec "`int A[*]`", les bornes inférieure et supérieure de la copie locale de `A` sont toujours, respectivement, de `1` et `n`.

B.3 Tri par fusion

L'algorithme 10 présente une version récursive du tri par fusion utilisant des tranches de tableau. Soulignons que puisqu'il n'y a aucun traitement à faire dans le cas trivial (tableau de taille 1), la branche correspondante a été éliminée.

[3] Évidemment, les mêmes notions s'appliquent à des tableaux d'éléments de types arbitraires.

```
procedure trouverPosition( int A[*], int n, int v ) returns int pos
# PRECONDITION
#   SOME( k >= 0 :: n = 2^k ),
#   ALL( 1 <= i < n :: A[i] <= A[i+1] )
# POSTCONDITION
#   SOME( 1 <= i <= n :: A[i]  = v ) => (1 <= pos <= n) & A[pos] = v,
#   ALL ( 1 <= i <= n :: A[i] ~= v ) => pos = 0
{
  if (n == 1) {
    if (A[1] == v) {
      pos = 1;
    } else {
      pos = 0;
    }
  } else {
    if (v <= A[n/2]) {
      pos = trouverPosition(A[1:n/2],   n/2, v);
    } else {
      pos = trouverPosition(A[n/2+1:n], n/2, v);
      if (pos != 0) { pos += n/2; }
    }
  }
}
```

Algorithme 9: Fouille binaire sur un tableau ordonné d'éléments

```
procedure fusionner( ref int A[*], int inf, int mid, int sup )
{
  ...
  # Aucun changement.
  ...
}

procedure trier( var int A[*], int n )
{
  if (n > 1) {
    trier( A[1:n/2],   n/2 );
    trier( A[n/2+1:n], n/2 );
    fusionner( A, 1, n/2, n );
  }
}
```

Algorithme 10: Tri par fusion