Table des matières

Chapitre A

La stratégie «diviser-pour-régner» pour la conception d'algorithmes récursifs

A.1 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 A.1.6).

```
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( ref int A[*],
                         int inf, int sup, int v )
              returns int pos
{
  if (inf == sup) {
    pos = (A[inf] == v) ? inf : 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 A.1: Fouille binaire sur un tableau ordonné d'éléments.

A.1.1 Fouille binaire (dichotomique)

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

– Analyse de l'algorithme :

- Opération barométrique : comparaison de l'élément `v` avec un élément du tableau `A`.
- 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.
- 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 $\Theta(\lg n)$.

A.1.2 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.

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

```
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 {
    # Decomposition... triviale!
    int mid = (inf + sup) / 2

    # Solution (recursive) des sous-problemes.
    trierRec ( A, inf,   mid )
    trierRec ( A, mid+1, sup )

    # Combinaison des sous-solutions.
    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++]
    } else {
      C[i++] = A[i2++]
    }
  }

  # 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."
  for [i = inf to sup] { A[i] = C[i] }
}
```

Algorithme A.2: Tri fusion.

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

- Opération barométrique : affectation de `A[-]` vers `C[-]`.
- Taille du problème : nombre total d'items à fusionner dans les deux sous-listes, donc `sup-inf+1`.
- Type d'analyse : pire 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 :

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

– 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 : pire cas.
- 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.).

A.1.3 L'approche diviser-pour-régner en général

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).

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

```
PROCEDURE resoudreDiviserPourRegner( p: Probleme ):
                                                  Solution
  SI p est un problème simple ALORS
    sol <- on résout le problème simple p
  SINON
    (p_1, ..., p_k) <- on décompose le problème p
                       en k sous-problèmes
     POUR i <- 1 A k FAIRE
       sol_i <- resoudreDiviserPourRegner( p_i )
     FIN
     sol <- on combine les solutions sol_1, ..., sol_k
                       des sous-problèmes
  FIN
  RETOURNER( sol )
FIN
```

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

A.1.4 Tri rapide (*Quicksort*)

– Algorithme célèbre développé par C.A.R. 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.

```
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] # Echange les deux elements.
    }
  }

  # On deplace le pivot a sa bonne position.
  A[posPivot] :=: A[inf] # Echange les deux elements.
}
```

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

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

- Algorithme pour tri rapide : Algorithme A.4.

- Analyse du pire cas pour tri rapide :

```
procedure trierRec( ref int A[*], int inf, int sup )
{
  if (inf >= sup) {
    # Tableau vide ou de taille 1: rien a faire
  } else {
    int posPivot
    # Decomposition en deux sous-problemes.
    partitionner( A, inf, sup, posPivot )

    # Solution (recursive) des deux sous-problemes.
    trierRec( A, inf, posPivot-1 )
    trierRec( A, posPivot+1, sup )

    # Combinaison des sous-solutions.
    # Rien a faire!
  }
}


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

Algorithme A.4: Tri *quicksort*.

– 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.

Donc, le pire cas sera $\Theta(n^2)$:

$$\begin{aligned}
T(n) &= T(n-1)+n-1 \\
&= [T(n-2)+n-2]+n-1 \\
&= T(n-2)+(n-2)+(n-1) \\
&= T(n-3)+(n-3)+(n-2)+(n-1) \\
&= \ldots \\
&= T(n-i)+(n-i)+(n-i+1)+\ldots+(n-2)+(n-1) \\
&= \ldots \\
&= T(1)+1+2+\ldots+(n-2)+(n-1) \\
&= 0+1+2+\ldots+(n-2)+(n-1) \\
&= \sum_{i=0}^{n-1} i \\
&= \frac{n(n-1)}{2}
\end{aligned}$$

– 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* (cf. le cours INF4100). Ceci peut aussi être montré, de façon informelle, tel qu'illustré dans les paragraphes qui suivent.

Variante améliorée du tri rapide

Dans l'algorithme A.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 $\Theta(n)$, 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 (grâce au choix de la médiane comme pivot), on obtiendrait alors une décomposition *équilibrée* (générant deux sous-problèmes de tailles presqu'identiques) semblable à celle du tri par fusion.

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 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 :

[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.

- 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 :

Algorithme (coût pour *chaque* appel)	Coût de la décomposition en sous-problèmes	Coût de l'assemblage des sous-solutions	**Coût pour la partie non-récursive**	**Complexité totale**
Tri par fusion	$O(1)$	$O(n)$	$O(n)$	$O(n\ lg\ n)$
Tri rapide pire cas (avec mauvais pivot)	$O(n)$	$O(1)$	$O(n)$	$O(n^2)$
Tri rapide (avec **médiane** comme pivot)	$O(n)$	$O(1)$	$O(n)$	$O(n\ lg\ n)$

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).

A.1.5 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 A.5 (en supposant une décomposition *dichotomique*, c'est-à-dire en deux sous-problèmes) :

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

Algorithme A.5: 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 système d'exploitation 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.

A.1.6 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 exponentielle ou factorielle, 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 (modèle producteurs-consommateurs avec canaux de communication).

A.2 Diviser-pour-régner générique... 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 A.3 (p. 8).

```
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
```

Code Haskell 1: Diviser-pour-régner générique en Haskell.

– 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 `Solution`s 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 A.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`.

```
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
```

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

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 `n`ième nombre de Fibonnaci.

A.3 Diviser-pour-régner générique... en Java

Le Programme Java A.1 présente une classe abstraite `ProblemeDPR` définissant un patron générique (*template pattern*) pour la résolution d'un problème à l'aide de l'approche diviser-pour-régner.

Le Programme Java A.2 présente une classe concrète `Factoriel`, sous-classe de la classe `ProblemeDPR`, permettant de calculer la factoriel d'un nombre entier, et ce en instantiant le patron générique (*template pattern*) de l'approche diviser-pour-régner.

Le Programme Java A.3–A.4 présente une classe concrète `TriFusion`, sous-classe de la classe `ProblemeDPR`, permettant d'effectuer un tri fusion, et ce en instantiant le patron générique (*template pattern*) de l'approche diviser-pour-régner.

Programme Java A.1 Classe abstraite Java pour patron (*template pattern*) diviser-par-régner générique.

```
//
// Une classe Java abstraite et generique pour representer des problemes
// a resoudre par une approche diviser-pour-regner.
//

abstract class ProblemeDPR<T> {
   private T solution;

   public T solution() { return(solution); }


   public abstract boolean estSimple();

   public abstract T resoudreSimple();


   public abstract ProblemeDPR<T>[] decomposer();

   public abstract T combiner( ProblemeDPR<T>[] s );

   public void resoudre()
   {
     if ( estSimple() ) {
       solution = resoudreSimple();
     } else {
       ProblemeDPR<T>[] problemes = decomposer();
       for ( int i = 0; i < problemes.length; i++ ) {
         problemes[i].resoudre();
       }
       solution = combiner( problemes );
     }
   }

}
```

Programme Java A.2 Classe concrète Java pour calcul de factoriel.

```
//
// Calcul de factoriel vu comme une instance de la classe generique.
//
class Factoriel extends ProblemeDPR<Integer> {
    int inf, sup;

    Factoriel( int inf, int sup ) {
      this.inf = inf;
      this.sup = sup;
    }

    public boolean estSimple() {
      return( inf == sup );
    }

    public Integer resoudreSimple() {
      return( inf );
    }

    public Factoriel[] decomposer() {
      int mid = (inf + sup)/2;
      return( new Factoriel[] {
                  new Factoriel( inf, mid ),
                  new Factoriel( mid+1, sup ) } );
    }

    public Integer combiner( ProblemeDPR<Integer>[] s ) {
      return( s[0].solution() * s[1].solution() );
    }


    public static void main(String[] args) {
      if (args.length == 0) {
        System.out.println( "Usage:" );
        System.out.println( "  java Factoriel n" );
        System.exit(1);
      }

      Factoriel f
          = new Factoriel( 1, Integer.parseInt(args[0]) );
      f.resoudre();
      System.out.println( f.solution() );
    }
}
```

Programme Java A.3 Classe concrète Java pour tri par fusion (première partie).

```
//
// Tri par fusion vu comme une instance de la classe generique.
//
class TriFusion extends ProblemeDPR<Integer[]> {
  Integer[] elems;

  TriFusion( Integer[] aTrier ) {
    elems = aTrier;
  }

  public boolean estSimple() {
    return( elems.length == 1 );
  }

  public Integer[] resoudreSimple() {
    return( elems );
  }

  public TriFusion[] decomposer() {
    assert elems.length % 2 == 0;

    int mid = elems.length/2;

    Integer[] gauche = new Integer[mid];
    for ( int i = 0; i < mid; i++ ) {
       gauche[i] = elems[i];
    }
    Integer[] droite = new Integer[mid];
    for ( int i = 0; i < mid; i++ ) {
       droite[i] = elems[i+mid];
    }
    return( new TriFusion[]{ new TriFusion(gauche), new TriFusion(droite) } );
  }

  public Integer[] combiner( Integer[] s ) {
    Integer[] res = new Integer[solution().length + s.length];
    int i1 = 0, i2 = 0, i = 0;
    while ( i1 < solution().length && i2 < s.length ) {
      if ( solution()[i1] < s[i2] ) {
        res[i++] = solution()[i1++];
      } else {
        res[i++] = s[i2++];
      }
    }
    for (int j = i1; j < solution().length; j++ ) {
      res[i++] = solution()[j];
    }
    for (int j = i2; j < s.length; j++ ) {
      res[i++] = s[j];
    }
```

Programme Java A.4 Classe concrète Java pour tri par fusion (deuxième partie).

```
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println( "Usage:" );
      System.out.println( "  java TriFusion n" );
      System.exit(1);
    }

    int nbElems = Integer.parseInt(args[0]);
    Integer[] vals = new Integer[nbElems];
    for ( int i = 0; i < nbElems; i++ ) {
      vals[i] = 100 - i;
    }
    TriFusion f = new TriFusion( vals );
    f.resoudre();
    for ( int i = 0; i < nbElems; i++ ) {
      System.out.println( f.solution()[i] );
    }
  }
}
```

Références