Partie 2
Chapitre 8
Trier

1. Introduction

Ce chapitre présente le tri de données comparables. On considère une collection C de données de même type **`T`**, quelconque mais **`COMPARABLE`**. On veut obtenir une représentation ordonnée des éléments de la collection. Les données seront organisées le plus souvent en tableau. On ne cherche donc pas ici des tris de qualité, évalués selon leurs performances, mais plutôt un prétexte à la construction raisonnée d'algorithmes.

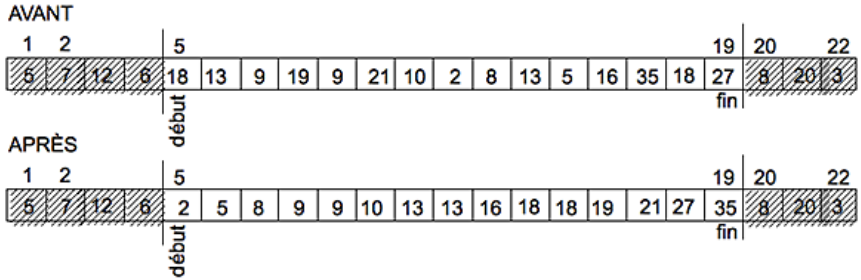
La section « Spécifier un algorithme de tri » commence par poser le problème. C'est la partie difficile du chapitre et elle peut être ignorée en première lecture. En particulier, la spécification algorithmique complète d'un algorithme de tri impose la définition des multi-ensembles invariables et de certaines opérations applicables difficiles à spécifier. La section suivante étudie quelques algorithmes usuellement définis dans une initiation à l'algorithmique. La section « Fusionner deux tableaux triés » montre comment les fusionner pour obtenir un nouveau tableau trié. La section « Exercices », enfin, propose des exercices plus difficiles.

2. Spécifier un algorithme de tri

Cette section est faite de deux parties. La première présente le problème du tri de données comparables. La seconde étudie la postcondition d'un tri. C'est la partie vraiment difficile du chapitre et elle peut être ignorée en première lecture.

2.1 Présentation du problème du tri

Soit `t` un tableau dont les composantes sont d'un type **`T`** dérivé de **`COMPARABLE`**. On veut un algorithme qui replace dans `t` ses composantes en ordre, par exemple, croissant. La figure suivante représente un tableau d'entiers avant et après le tri.



Comme d'habitude, l'algorithme à définir intervient sur une partie précisée du tableau, entre les cases de numéros `début` et `fin`. Les données sont replacées en ordre croissant dans le même tableau, qui se trouve donc modifié par l'algorithme. Par conséquent, il s'agit d'écrire une procédure. Les données en entrée sont le tableau `t` et les numéros des composantes extrêmes `début` et `fin`. La signature de la procédure et sa précondition peuvent donc être précisées :

```
Algorithme tri_ascendant
    # Trier `t[début .. fin]' en ordre croissant.

Entrée
    t : TABLEAU[T → COMPARABLE] # Le tableau à trier
    début, fin : ENTIER # Numéros des composantes extrêmes à trier
```

```
précondition
    # `début' et `fin' sont des index valides de `t'
    début ≤ fin => index_valide(t, début) et index_valide(t, fin)

    # `t[début .. fin]' a été initialisé
    est_défini(t, début, fin)
```

L'algorithme **`est_défini`** est un prédicat qui retourne `VRAI` si et seulement si le sous-tableau `t[début .. fin]` a été initialisé.

Algorithme 1 : Spécifications de ***`est_défini`***

```
Algorithme est_défini
    # `t[début .. fin]' a-t-il été initialisé ?

Entrée
    t : TABLEAU[T → COMPARABLE] # Le tableau à explorer
    début, fin : ENTIER # Numéros composantes extrêmes à vérifier

Résultat : BOOLÉEN

précondition
    # `début' et `fin' sont des index valides de `t'.
    début ≤ fin => index_valide(t, début) et index_valide(t, fin)

postcondition
    # VRAI lorsque les indices ne sont pas en ordre
    début > fin => Résultat = VRAI

    # VRAI si et seulement si les composantes de `t[début .. fin]'
    # sont non NUL i.e. sont initialisées :
    # Résultat = VRAI ou sinon (∃k ∈ ℕ) (début ≤ k ≤ fin => t[k] = NUL)
    début = fin => Résultat = (t[début] ≠ NUL)
    début < fin => Résultat =
        (
            (t[début] ≠ NUL)
          et alors
            est_défini(t, début+1, fin)
        )

fin est_défini
```

La postcondition du tri est plus difficile à obtenir. C'est elle qui est étudiée dans la section suivante.

2.2 Étude de la postcondition du tri

Cette section est difficile et peut être ignorée en première lecture. Dans ce cas, on peut passer directement à la section « Quelques algorithmes simples ».

La postcondition précise d'abord que le sous-tableau `t[début .. fin]` du tableau `t` est triée, ici en ordre croissant. La clause qui exprime cet état a déjà été écrite au chapitre « Itération ». Elle utilise le prédicat **`est_trié_en_ordre_ascendant`** dont le chapitre « Itération » a étudié une version itérative.

On obtient donc :

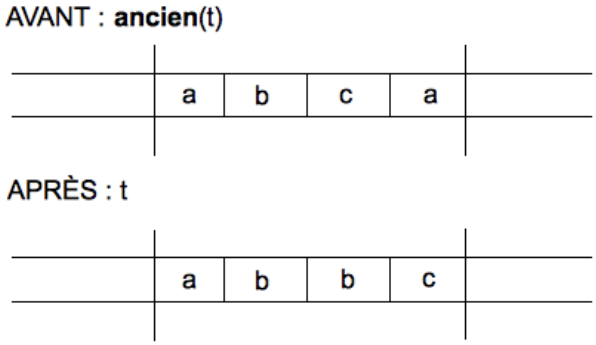
```
...
postcondition
    # `début' et `fin' ne sont pas modifiés
    ancien(début) = début
    ancien(fin)   = fin

    # t[début .. fin] est trié en ordre croissant
    est_trié_en_ordre_ascendant(t, début, fin)
...
```

Cependant, nous devons exprimer que la partie `t[début .. fin]` a certes été modifiée, mais que les mêmes éléments restent présents. Ainsi, les tableaux `t` et **`ancien`**`(t)` ont les mêmes éléments, dans les mêmes sous-tableaux, entre les cases de numéros `début` et `fin`, mais qu'ils ne sont pas nécessairement identiques puisque certaines composantes ont peut-être été déplacées. Nous ne pouvons pas écrire la condition (c1) :

```
(∀k ∈ ℕ)(début ≤ k ≤ fin)(∃i ∈ ℕ)(début ≤ i ≤ fin)
(t[k] = ancien(t)[i])
```

car il peut exister des composantes de `t` en plusieurs exemplaires entre `début` et `fin`.

Considérons, par exemple, les tableaux de la figure suivante :



Le résultat `t` est trié en ordre croissant et il passe le test exprimé par la condition (c1). Pourtant, le tri réalisé n'est pas correct puisqu'on ne retrouve pas dans `t[début .. fin]` tous les éléments de **`ancien`**`(t)[début .. fin]`. Le second exemplaire de l'élément `a` a été remplacé par un nouvel exemplaire de l'élément `b`. Nous devons pouvoir nous assurer que `t` et **`ancien`**`(t)` possèdent exactement le même nombre d'exemplaires des mêmes composantes. Pour définir un prédicat qui permettra de les comparer, on peut « éliminer », à chaque étape de la comparaison, les composantes trouvées dans les deux tableaux. Pour simplifier les notations, écrivons les composantes des tableaux en utilisant la notation ensembliste. Pour les tableaux de la figure ci-dessus, on a, initialement :

```
E = {a, b, c, a}
F = {a, b, b, c}
```

`E` et `F` sont appelés des multi-ensembles.

Définition

On appelle **multi-ensemble** un ensemble dans lequel les éléments peuvent apparaître en plusieurs exemplaires.

Pour vérifier que les deux multi-ensembles sont égaux, on compare les deux premiers éléments et on les trouve égaux. Ils sont éliminés et on obtient alors les deux multi-ensembles : {b, c, a} et {b, b, c}. Une nouvelle comparaison permet de constater que les premiers éléments de chacun d'eux sont égaux. Après élimination, on obtient {c, a} et {b, c}. Le premier élément du premier multi-ensemble est alors égal au deuxième élément du second multi-ensemble et on obtient {a} et {b}. La dernière comparaison établit que les deux multi-ensembles initiaux n'étaient pas égaux. Comme pour deux ensembles quelconques, l'égalité de deux multi-ensembles est définie à l'aide de l'inclusion :

Définition

Soient E et F deux multi-ensembles. On a : (E = F) ⇔ (E ⊑ F **et** F ⊑ E)

Autrement dit, comme pour deux ensembles quelconques, deux multi-ensembles sont égaux lorsque l'un est inclus dans l'autre et réciproquement. Soit alors **`est_égal_à`** le prédicat qui établit l'égalité de deux tableaux considérés comme des multi-ensembles. Sa spécification est :

Algorithme 2 : Spécification de l'égalité de deux multi-ensembles

```
Algorithme est_égal_à
    # `t[début_t .. fin_t]' et `u[début_u .. fin_u]' sont-ils égaux ?

Entrée
    t, u : TABLEAU[T → COMPARABLE]
    début_t, fin_t, début_u, fin_u : ENTIER

Résultat : BOOLÉEN

précondition
    index_valide(t, début_t)
    index_valide(t, fin_t)
    est_défini(t, début_t, fin_t)
    index_valide(u, début_u)
    index_valide(u, fin_u)
    est_défini(u, début_u, fin_u)

postcondition
    Résultat =
        (
```

```
            est_inclus_dans(t, début_t, fin_t, u, début_u, fin_u)
          et alors
            est_inclus_dans(u, début_u, fin_u, t, début_t, fin_t)
        )

fin est_égal_à
```

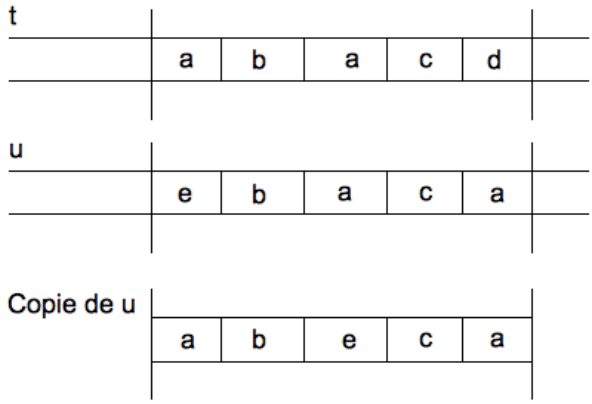
Il reste à définir le prédicat **est_inclus_dans**. C'est une autre difficulté de cette spécification. Comme pour les ensembles, l'inclusion sera définie à l'aide de l'appartenance. Soit **appartient** le prédicat qui rend VRAI si et seulement si un élément est une composante d'un tableau.

*Algorithme 3 : Spécification de **appartient** dans un tableau.*

```
Algorithme appartient, infixe ∈
    # `e' appartient-il à `t[début .. fin]' ?

Entrée
    t : TABLEAU[T]
    début, fin : ENTIER
    e : T

Résultat : BOOLÉEN

précondition
    index_valide(t, début)
    index_valide(t, fin)
    est_défini(t, début, fin)

postcondition
    # Résultat = [(∃k ∈ ℕ)(début ≤ k ≤ fin => t[k] = e)]
    Résultat = (rang(t, début, fin, e) ≠ ABSENT)

fin appartient
```

Pour s'assurer qu'un élément donné appartient au tableau, on calcule son rang, c'est-à-dire le numéro de la composante de cet élément dans le tableau. Ce rang doit être différent de ABSENT.

Pour vérifier qu'un multi-ensemble `E` est inclus dans le multi-ensemble `F`, nous devons construire un algorithme capable de vérifier l'appartenance de chacun des exemplaires d'un même élément de `E` à `F`. Pour cela, lorsqu'un élément de `E` a été trouvé dans `F`, on construit une copie de `F` dans laquelle l'élément commun occupe la première composante. Mais cette copie laisse `F` invariable. La figure ci-dessous illustre les opérations à réaliser lorsque le premier élément `a` du tableau `t` a été reconnu comme un élément du tableau `u`.



Comme le premier élément de `t` et de la copie de `u` sont les mêmes, on peut poursuivre la même comparaison sur les mêmes ensembles, mais privés de leur premier élément. Autrement dit, il suffit de réaliser la même comparaison sur les tableaux `t` dans `[début_t+1 .. fin_t]` et copie de `u` dans `[début_u+1 .. fin_u]`. La copie de `u` est réalisée par la fonction **`copie`**, dont les spécifications sont :

*Algorithme 4 : Spécification de la fonction **`copie`** d'un tableau*

```
Algorithme copie
    # Réalise une copie de `t[début .. fin]'.

Entrée
    t : TABLEAU[T]
    début, fin : ENTIER

Résultat : TABLEAU[T][début]
```