ÉCOLE SUPÉRIEURE D'ÉLECTRICITÉ

Fondements de l'Informatique
Structures de Données et Algorithmes

Claude Bocage
Frédéric Boulanger

2012-06-25

Table des matières

1. Types abstraits algébriques

1.1 Introduction

Lors de la conception d'une application, on a souvent besoin de définir de nouveaux types de données (ou de nouvelles classes) correspondant aux objets spécifiques manipulés par cette application.

Il importe de définir ces nouveaux types de données par leur comportement (leurs fonctionnalités) et non par leur représentation en mémoire.

On définit donc un nouveau type en spécifiant les opérations que l'on souhaite effectuer sur les données de ce type.

Si on utilise un langage à objets pour écrire l'application, le type de données sera associé à une classe, et chaque opération à une méthode de la classe.

Si on utilise un langage ne possédant pas la notion d'objet, comme C ou PASCAL, le type de données sera associé à un type du langage, et chaque opération sera associée à une fonction travaillant sur des données de ce type.

Lors de la conception de l'application, le type ne sera connu dans un premier temps que par son nom et le nom des opérations associées.

Il faudra bien sûr aussi indiquer pour chaque opération sur quelles données elle travaille et le résultat qu'elle rend (par exemple, l'opération + sur les entiers reçoit deux entiers et rend un entier).

Mais il faut aussi préciser pour chaque opération ce qu'elle fait en indiquant quelles doivent être les propriétés vérifiées par le résultat qu'elle rend.

Dans un premier temps, on ne s'intéresse ni à la représentation des données en mémoire, ni à la manière de réaliser les opérations. On se situe donc à un niveau pratiquement indépendant du langage de programmation qu'on utilisera par la suite.

Spécifier les propriétés du résultat d'une opération peut se faire « en bon français » (par exemple, pour l'opération + sur les entiers, on peut indiquer que le résultat est la somme des paramètres).

Toutefois, une formulation en langage naturel est dans de nombreux cas une source d'ambiguïté. C'est pourquoi nous allons utiliser une méthode mathématique rigoureuse pour spécifier un type de données : les types abstraits algébriques.

Cette méthode nous fournira aussi des outils permettant de vérifier que l'implémentation est conforme à la spécification.

1.2 Écriture d'une spécification

1.2.1 Exemple des booléens

Nous allons présenter cette méthode de spécification sur un exemple simple : le type de données booléen.

```
spéc BOOL₀
  sorte Bool
  opérations
    vrai :           → Bool
    faux :           → Bool
    non  : Bool      → Bool
    et   : Bool Bool → Bool
    ou   : Bool Bool → Bool
  axiomes a, b, c : Bool
    (b₁) non(vrai) = faux
    (b₂) non(non(a)) = a
    (b₃) et(a,vrai) = a
    (b₄) et(a,faux) = faux
    (b₅) et(et(a,b),c) = et(a,et(b,c))
    (b₆) et(a,b) = et(b,a)
    (b₇) ou(a,b) = non(et(non(a),non(b)))
fspéc
```

Une spécification commence par le mot clé *spéc* suivi du nom que l'on souhaite donner à la spécification, et se termine par le mot clé *fspéc*.

Derrière le mot clé *sorte*, on donne la liste des types de données définis par cette spécification (nommés dans une telle spécification les sortes).

Si cela est possible, on essaiera de ne définir qu'une seule sorte dans une spécification. Une spécification multi-sortes est toujours plus compliquée, et ne doit être envisagée que si on ne peut faire autrement (par exemple dans le cas de types fortement imbriqués).

On trouvera ensuite, derrière le mot clé *opérations*, la liste des opérations que l'on souhaite pouvoir effectuer sur notre type de données, accompagnées de ce que l'on appelle leur profil : il s'agit de donner pour chaque opération le nombre et le type des opérandes, ainsi que le type de la valeur rendue (devant la flèche, la liste des opérandes, derrière la flèche la sorte du résultat).

Les opérations sans opérande (ici *vrai* et *faux*) sont nommées (à juste titre !) des constantes.

Les lignes qui suivent le mot clé *axiomes* donnent les propriétés que doivent vérifier les opérations.

Nous verrons par la suite la signification formelle de ces axiomes. De manière plus pragmatique, sur un exemple :

```
(b₃) et(a,vrai) = a
```

b_3 est simplement une étiquette (facultative) permettant de référencer par la suite cet axiome. On trouve ensuite, de part et d'autre du signe =, deux expressions, chacune d'entre elles pouvant utiliser des variables (celles déclarées derrière le mot clé *axiomes*). L'axiome indique que, quelles que soient les valeurs de ces variables, l'expression de gauche doit avoir même valeur que celle de droite.

Cet ensemble d'axiomes est l'ensemble des propriétés qui doivent être vérifiées pour que les opérateurs donnent le résultat escompté. L'ensemble d'axiomes définit le comportement des opérations. En vérifiant les axiomes pour une implémentation, on vérifie ainsi qu'elle a le comportement spécifié.

Pour des raisons de commodité d'usage, il est aussi possible d'utiliser une notation plus classique pour certains opérateurs que la notation fonctionnelle. Dans cette notation, dite in-fixée, les opérandes figurent de part et d'autre de l'opérateur. On prendra alors soin de placer dans le nom de l'opérateur (au niveau du profil) un certains nombres de caractères _, le nième caractère _ symbolisant la place du nième opérande. Par exemple :

```
_∧_ : Bool Bool → Bool
```

indique que le *et* de a et b s'écrit $a \wedge b$.

La spécification du type booléen devient alors :

```
spéc BOOL₁
  sorte Bool
  opérations
    vrai :           → Bool
    faux :           → Bool
    ¬_   : Bool      → Bool
    _∧_  : Bool Bool → Bool
    _∨_  : Bool Bool → Bool
  axiomes a, b, c : Bool
    (b₁) ¬vrai = faux
    (b₂) ¬¬a = a
    (b₃) a∧vrai = a
    (b₄) a∧faux = faux
    (b₅) (a∧b)∧c = a∧(b∧c)
    (b₆) a∧b = b∧a
    (b₇) a∨b = ¬((¬a)∧(¬b))
fspéc
```

1.2.2 Signature

Définition 1 : Signature

La *signature* d'une spécification est l'ensemble des sortes de cette spécification, associé à l'ensemble des noms d'opérations et de leur profil.

Il s'agit donc des paragraphes *sorte(s)* et *opérations* de la spécification.

1.2.3 Termes

Nous allons maintenant présenter de manière plus rigoureuse que nous l'avons fait dans le paragraphe précédent la manière de spécifier le résultat de chaque opération, autrement dit la manière d'écrire les axiomes.

De manière informelle, on appelle terme toute expression que l'on peut construire à partir des opérateurs et des variables en respectant le profil des opérations. Un terme sans variable est appelé un terme clos.

Définissons maintenant de manière plus formelle cette notion de terme.

Définition 2 : Alphabet

Un *alphabet* est un ensemble de symboles. Un *mot* construit sur un alphabet A est une suite finie d'éléments de A, éventuellement vide.

Soit une spécification $SPEC$ et un ensemble X de variables. On considère l'alphabet A constitué des opérateurs de $SPEC$, des variables de X, des parenthèses ouvrante et fermante et de la virgule.

Définition 3 : Termes

L'ensemble $T(X)_S$ des termes de sorte S de la spécification *SPEC* avec variables dans X est le plus petit ensemble de mots construits sur A contenant :
- Les symboles de constantes de *SPEC* de sorte S.
- Les variables de X de sorte S
- Les suites de symboles de la forme `f(t1,...,tn)` où `f` est un opérateur de profil `f : S1 ... Sn → S` et `t1 ... tn` des termes de $T(X)_{S_1} \dots T(X)_{S_n}$
- Les suites de symboles de la forme `t1 f t2` où `f` est un opérateur de profil `_f_ : S1 S2 → S` et `t1` un terme de sorte $T(X)_{S_1}$ et `t2` un terme de sorte $T(X)_{S_2}$.
- Les suites de symboles de la forme `f t1` où `f` est un opérateur de profil `f_ : S1 → S` et `t1` un terme de sorte $T(X)_{S_1}$.
- Les suites de symboles de la forme `t1 f` où `f` est un opérateur de profil `_f : S1 → S` et `t1` un terme de sorte $T(X)_{S_1}$.

L'ensemble $T(X)$ des termes de *SPEC* avec variables dans X est l'union de tous les $T(X)_S$ pour toutes les sortes S de *SPEC*.
L'ensemble $T(\emptyset)_S$ encore noté T_S est l'ensemble des termes clos de sorte S de *SPEC*.

1.2.4 Axiomes

Définition 4 : Axiome

Un axiome est une expression de la forme :
`t1=t2`
où `t1` et `t2` sont des termes de *SPEC* de la même sorte (termes avec ou sans variables).

Intuitivement, les axiomes permettent d'indiquer quels sont les termes clos qui doivent être « égaux », donnant ainsi les propriétés que doivent posséder les opérations pour qu'elles fassent ce que l'on attend d'elles. Ces termes clos sont obtenus en remplaçant les variables par des termes clos de la même sorte que les variables remplacées, une même variable devant bien sûr être remplacée partout par le même terme.

Toutes les autres propriétés que les résultats des opérations doivent posséder doivent pouvoir être démontrées à l'aide des seuls axiomes de la spécification (ou encore à l'aide de propriétés précédemment démontrées), en utilisant une méthode que nous verrons par la suite.

Cette notion « d'égalité » de termes clos peut être définie de manière plus formelle. Les axiomes induisent une relation $\mathcal{R}$ entre termes clos définie par :

Définition 5

Soient t_1 et t_2 deux termes clos. On a $t_1 \mathcal{R} t_2$ si et seulement si il existe un axiome tel qu'en remplaçant les variables par des termes clos on obtienne t_1 à gauche du signe égal et t_2 à droite.

Cette relation $\mathcal{R}$ induit une autre relation sur les termes clos notée $\cong$ et nommée *congruence* définie par les règles suivantes :
- $\forall (t_1, t_2), t_1 \mathcal{R} t_2 \Rightarrow t_1 \cong t_2$
- $\forall t, t \cong t$
- $\forall (t_1, t_2), t_1 \cong t_2 \Rightarrow t_2 \cong t_1$
- $\forall (t_1, t_2, t_3), t_1 \cong t_2$ et $t_2 \cong t_3 \Rightarrow t_1 \cong t_3$
- Pour tout opérateur f de profil $f : S_1 \dots S_n \to S$
  Pour tous termes $t_1 \dots t_n$ de sortes $S_1 \dots S_n$ et $t'_1 \dots t'_n$ de sortes $S_1 \dots S_n$
  $t_1 \cong t'_1, \dots, t_n \cong t'_n \Rightarrow f(t_1, \dots, t_n) \cong f(t'_1, \dots, t'_n)$

Cette relation est, par construction, une relation d'équivalence. Elle respecte de plus les opérateurs. On appelle une telle relation une congruence. Les termes que nous avons appelés « égaux » sont en fait des termes appartenant à la même classe d'équivalence (de congruence).

La relation $\cong$ des spécifications $BOOL_0$ et $BOOL_1$ possède deux classes de congruence, le terme *vrai* étant un représentant de l'une d'entre elle et le terme *faux* un représentant de l'autre (ceci restant bien sûr à démontrer).

1.3 Théorèmes

Un théorème est une propriété que l'on peut démontrer à l'aide des axiomes de la spécification et des théorèmes précédemment démontrés.

Un théorème possède la même forme qu'un axiome :

$t_1 = t_2$

où t_1 et t_2 sont des termes (avec ou sans variables) de la même sorte.

Un théorème possède la même signification intuitive qu'un axiome : il indique que si on remplace dans t_1 et t_2 les variables par des termes clos de la même sorte, les termes obtenus sont alors « égaux » (en fait ils sont congrus au sens de la relation $\cong$).

La démonstration de théorèmes doit permettre de confirmer que les résultats des opérations de la spécification sont ceux qui sont escomptés. Si un théorème jugé utile ne peut être démontré, il est alors légitime de se demander si le jeu d'axiomes de la spécification est suffisant, et s'il n'est pas opportun de le compléter. A l'opposé, si on arrive à démontrer une propriété non souhaitée (i.e. manifestement fausse), il est alors indispensable de vérifier les axiomes afin d'isoler celui ou ceux qui sont erronés.

A titre d'exemple, en considérant la spécification $BOOL_1$, on doit être capable de démontrer $vrai \wedge a = a$ et $a \vee vrai = vrai$.

Par contre, si on arrive à démontrer $vrai = faux$, cela indique clairement qu'un ou plusieurs axiomes sont erronés.

On disposera de deux méthodes pour démontrer un théorème, basées sur la définition de la relation $\cong$.

Définissons auparavant une notation :

Soit t un terme avec variables utilisant la variable x de sorte S. Soit u un terme de sorte S (avec ou sans variables). On note $t[u/x]$ le terme obtenu en remplaçant partout la variable x par le terme u.

1.3.1 Théorèmes déductifs

Soit à démontrer le théorème $t_1 = t_2$.

A partir d'un des deux termes (t_1 ou t_2), on applique une succession de transformations jusqu'à obtenir le deuxième.

Un terme t peut être remplacé par un terme t' si et seulement si $t = t'$ est un axiome ou un théorème.

Les cinq règles suivantes permettent de construire directement des théorèmes à partir des axiomes ou de théorèmes précédemment démontrés :

– R1 : réflexivité
  Pour tout terme t, $t = t$ est un théorème
– R2 : symétrie
  Pour tous termes t_1 et t_2 de même sorte, si $t_1 = t_2$ est un théorème ou un axiome, alors $t_2 = t_1$ est un théorème.
  Par exemple, selon $BOOL_1$, $a{=}a{\wedge}vrai$ est un théorème puisque $a{\wedge}vrai{=}a$ est un axiome.
– R3 : transitivité
  Pour tous termes t_1, t_2 et t_3 de même sorte, si $t_1 = t_2$ et $t_2 = t_3$ sont des théorèmes ou des axiomes, alors $t_1 = t_3$ est un théorème.
– R4 : substitution
  Pour tous termes t_1 et t_2 de même sorte avec variables dans X,
  pour tout $x \in X$,
  pour tout terme u de même sorte que x,
  si $t_1 = t_2$ est un théorème ou un axiome, alors $t_1[u/x] = t_2[u/x]$ est un théorème.
– R5 : congruence
  Pour tous termes $t_1, \ldots, t_n$ de sortes $S_1, \ldots, S_n$,
  Pour tous termes $t'_1, \ldots, t'_n$ de sortes $S_1, \ldots, S_n$,
  Pour toute opération de profil $f : S_1 \ldots S_n \to S$,
  Si $t_1 = t'_1, \ldots, t_n = t'_n$ sont des théorèmes ou des axiomes, alors $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ est un théorème.

Exemple 1

Avec la spécification $BOOL_1$, montrons le théorème suivant :

(th_1) $\neg faux = vrai$

On a $faux = \neg vrai$ par b_1 et $R2$

donc $\neg faux = \neg\neg vrai$ par $R5$

$\neg\neg vrai = vrai$ par b_2 et $R4$

donc $\neg faux = vrai$ par $R3$

Exemple 2

Montrons, avec la spécification $BOOL_1$, que $a \vee vrai = vrai$ est un théorème.

$a \vee vrai = \neg((\neg a) \wedge (\neg vrai))$ par b_7 et $R4$

$\neg((\neg a) \wedge (\neg vrai)) = \neg((\neg a) \wedge faux)$ par b_1 et $R5$

or $(\neg a) \wedge faux = faux$ par b_4 et $R4$

donc :

$\neg((\neg a) \wedge faux) = \neg(faux)$ par $R5$ et le théorème précédent

$\neg faux = vrai$ par le théorème de l'exemple 1

donc :

$a \vee vrai = vrai$ par $R3$

Exemple 3

Montrons, avec la spécification $BOOL_1$, que $a \wedge b = \neg((\neg a) \vee (\neg b))$ est un théorème.

$\neg((\neg a) \vee (\neg b)) = \neg(\neg(\neg(\neg a) \wedge \neg(\neg b)))$ par b_7 et $R5$

$\neg(\neg(\neg(\neg a) \wedge \neg(\neg b))) = \neg(\neg(a \wedge b))$ par b_2 et $R5$

$\neg(\neg(a \wedge b)) = a \wedge b$ par b_2 et $R4$

donc $\neg((\neg a) \vee (\neg b)) = a \wedge b$ par $R3$

et donc $a \wedge b = \neg((\neg a) \vee (\neg b))$ par $R2$

1.3.2 Induction structurelle

Soit une spécification *SPEC*, X un ensemble de variables et $T(X)$ l'ensemble des termes avec variables dans X.

Soit P un prédicat sur les éléments de $T(X)$, c'est-à-dire une propriété telle que pour tout élément t de $T(X)$, $P(t)$ soit vérifiée ou non.

La propriété P est vérifiée pour tout terme t de $T(X)$ si :

1. $P(t)$ est vérifiée pour toutes les constantes de *SPEC* et toutes les variables de X.
2. Pour tout terme t de la forme $f(t_1, \ldots, t_n)$, si $P(t_1), \ldots, P(t_n)$ sont vérifiées, alors $P(t)$ est vérifiée.

Exemple

Considérons la spécification $BOOL_1$, un ensemble X de variables vide (donc $T(X)$ est l'ensemble des termes clos de $BOOL_1$).

Montrons que tout terme clos t est congru à *vrai* ou à *faux* (selon la relation $\cong$). Cela revient à dire que pour tout terme clos t, t=*vrai* est un théorème ou t=*faux* est un théorème. Notons cette propriété P.

1. La propriété est vérifiée pour *vrai* et *faux* car *vrai*=*vrai* et *faux*=*faux* sont des théorèmes.
2. Les termes sont de trois formes différentes : $\neg t$, $t_1 \wedge t_2$ ou $t_1 \vee t_2$.
   (a) forme $\neg t$ avec *P(t)*
   - si t=*vrai*,
     $\neg t = \neg vrai$ par $R5$
     $\neg vrai = faux$ par b_1
     donc $\neg t = faux$ par $R3$
   - si t=*faux*,
     $\neg t = \neg faux$ par $R5$
     $\neg faux = vrai$ par le théorème th1 de l'exemple 1
     donc $\neg t = vrai$ par $R3$

   (b) forme $t_1 \wedge t_2$ avec $P(t_1)$ et $P(t_2)$
   - si t_1=*faux*
     $t_1 \wedge t_2 = faux \wedge t_2$ par $R5$
     $faux \wedge t_2 = t_2 \wedge faux$ par b_6
     $t_2 \wedge faux = faux$ par b_4 et $R4$
     donc $t_1 \wedge t_2 = faux$ par $R3$
   - si t_1=*vrai* et t_2=*vrai*,
     $t_1 \wedge t_2 = vrai \wedge vrai$ par $R5$
     $vrai \wedge vrai = vrai$ par b_3 et $R4$
     donc $t_1 \wedge t_2 = vrai$ par $R3$
   - si t_1=*vrai* et t_2=*faux*, $t_1 \wedge t_2 = vrai \wedge faux$ par $R5$
     $vrai \wedge faux = faux$ par b_4 et $R4$
     donc $t_1 \wedge t_2 = faux$ par $R3$

   (c) forme $t_1 \vee t_2$ avec $P(t_1)$ et $P(t_2)$
   - si t_1=*vrai*, $t_1 \vee t_2 = vrai \vee t_2$ par $R5$
     $vrai \vee t_2 = t_2 \vee vrai$ par un théorème à démontrer et $R4$
     $t_2 \vee vrai = vrai$ par le théorème de l'exemple 2 et $R4$
     donc $t_1 \vee t_2 = vrai$ par $R3$
   - si t_1=*faux* et t_2=*vrai*, $t_1 \vee t_2 = faux \vee vrai$ par $R5$
     $faux \vee vrai = vrai$ par le théorème de l'exemple 2 et $R4$
     donc $t_1 \vee t_2 = vrai$ par $R3$

– si $t_1 = faux$ et $t_2 = faux$, $t_1 \vee t_2 = faux \vee faux$ par $R5$
$faux \vee faux = faux$ par un théorème à démontrer et $R4$
donc $t_1 \vee t_2 = faux$ par $R3$

1.3.3 Théorèmes inductifs

Tous les théorèmes ne peuvent pas être démontrés de manière déductive, c'est-à-dire en appliquant uniquement une suite de transformations.

Par exemple, on ne saurait montrer de cette manière le théorème $t \wedge t = t$ avec la spécification $BOOL_1$, aucun axiome ne permettant de transformer le terme $t \wedge t$.

Il est toutefois possible de monter ce théorème par une autre méthode dite par induction.

Définition 6 : théorème inductif

$t_1 = t_2$ est un théorème inductif si toute équation obtenue à partir de $t_1 = t_2$ en remplaçant les variables par des termes clos de la même sorte que ces variables est un théorème déductif.

Il est possible de démontrer le théorème $t \wedge t = t$ de cette manière, en utilisant le résultat vu dans le paragraphe précédent qui permet d'affirmer que pour tout terme clos t, $t = vrai$ ou $t = faux$ est un théorème.

– si $t = vrai$ est un théorème, alors $t \wedge t = vrai \wedge vrai = vrai = t$
– si $t = faux$ est un théorème, alors $t \wedge t = faux \wedge faux = faux = t$

1.4 Extension d'une spécification

Le procédé d'extension de spécification permet de construire de nouvelles spécifications en incluant d'autres.

Donnons pour commencer la spécification d'un type « entier naturel » extrêmement simplifié.

```
spéc NAT0
  sorte Nat
  opérations
    0    :      → Nat
    succ : Nat → Nat
  axiomes
fspéc
```

Cette spécification est munie d'un opérateur 0 (constante), et d'un opérateur *succ* donnant le successeur d'un entier. Elle ne possède pas d'axiome.

Tout terme clos de NAT_0 possède donc la forme :
– soit de la constante 0
– soit d'une suite d'utilisations de *succ* sur cette même constante.
L'opérateur *succ* possède une propriété bien particulière : il permet de construire tous les termes clos à partir de la constante 0. Un tel opérateur, permettant de « fabriquer » tous les termes clos à partir des constantes est qualifié de *constructeur*.

Nous souhaitons maintenant ajouter des opérations arithmétiques classiques à ce type « entier naturel ». Nous pouvons le faire sans avoir à réécrire complètement la spécification, mais en en définissant une nouvelle qui « étend » NAT_0.

```
spéc NAT1
  étend NAT0
  opérations
    _+_ : Nat Nat → Nat
  axiomes n,m : Nat
    0 + n = n
    succ(m) + n = succ(m + n)
fspéc
```

On peut remarquer que cette spécification n'introduit pas de nouvelle sorte. Cela reste toutefois possible dans le cas général.

Lorsque l'on a réussi à isoler un constructeur (ici *succ*), la méthode à employer pour écrire les axiomes permettant de spécifier les autres opérateurs est, dans la plupart des cas, extrêmement simple et systématique : il suffit de donner des axiomes donnant le résultat de l'opération pour les constantes et pour les termes construits.

Une spécification peut en étendre plusieurs. Si on souhaite ajouter à NAT_1 un opérateur de comparaison noté ==, on peut le faire par une nouvelle spécification NAT_2 étendant à la fois $BOOL_1$ et NAT_1.

```
spéc NAT2
  étend NAT1,BOOL1
  opérations
    _==_ : Nat Nat → Bool
  axiomes n,m : Nat
    0 == 0 = vrai
    succ(n) == 0 = faux
    0 == succ(n) = faux
    succ(n) == succ(m) = n == m
fspéc
```

1.5 Termes conditionnels

Dans de nombreux cas, il est difficile d'exprimer le résultat d'une opération par un terme simple. Il est souvent utile de pouvoir dire que le résultat doit être un terme t_1 ou un autre terme t_2 en fonction d'une condition exprimée sur les paramètres.

A cet effet, toute spécification étendant $BOOL$ peut être munie, pour chaque sorte S, d'un opérateur que l'on peut noter $\boldsymbol{si}_S$:

```
si_S : Bool S S → S
```

défini par les 2 axiomes :

```
si_S(vrai,x,y) = x
si_S(faux,x,y) = y
```

Pour des raisons de commodité d'écriture, on introduit un opérateur *si_alors_sinon_fsi* de profil :

```
si_alors_sinon_fsi : Bool S S → S
```

répondant aux mêmes spécifications, mais valable pour toute sorte S. Cet opérateur est dit polymorphe.

```
si vrai alors x sinon y fsi = x
si faux alors x sinon y fsi = y
```

Cet opérateur est supposé implicitement défini pour toute spécification étendant $BOOL$

Exemple : Spécification d'un type ensemble d'entiers

```
spéc ENS
  étend NAT₁, BOOL₁
  sorte Ens
  opérations
    ∅    :          → Ens
    i    : Ens Nat → Ens          ajoute un élément à un ensemble
    _∈_  : Nat Ens →Bool
    card : Ens      → Nat
    s    : Ens Nat →Ens           supprime un élément d'un ensemble
  axiomes x,y:Nat;E:Ens
    x ∈ ∅ = faux
    x ∈ i(E,y) = x==y ∨ x∈E
    card(∅) = 0
    card(i(E,x)) = si x ∈ E alors card(E) sinon succ(card(E)) fsi
    s(∅,x) = ∅
    s(i(E,x),y) = si x∈E
                    alors s(E,y)
                    sinon
                      si x==y alors E sinon i(s(E,y),x) fsi
                  fsi
fspéc
```

Signalons que l'opérateur i est un constructeur du type ensemble.

1.6 Domaine de définition des opérateurs

Il est possible de restreindre le domaine de définition d'un ou plusieurs opérateurs, en indiquant une condition devant être remplie par ses opérandes pour que le terme résultant ait un sens (i.e. soit défini).

Par exemple, si on souhaite indiquer pour la spécification *ENS* que $i(E, x)$ n'a de sens que s'il n'y a pas déjà dans E un élément de même valeur que x, on écrira :

```
pré i(E,x) = ¬(x ∈ E)
```

De même, si $s(E, x)$ n'a de sens que s'il y a dans E un élément de même valeur que x, on écrira :

```
pré s(E,x) = x ∈ E
```

La spécification devient alors :

```
spéc ENS₁
  étend NAT₁,BOOL₁
  sorte Ens
  opérations
    ∅    :          → Ens
    i    : Ens Nat → Ens          ajoute un élément à un ensemble
    _∈_  : Nat Ens →Bool
    card : Ens      → Nat
    s    : Ens Nat →Ens           supprime un élément d'un ensemble
  préconditions
    pré i(E,x) = ¬(x ∈ E)
    pré s(E,x) = x ∈ E
  axiomes x,y:Nat;E:Ens
    x∈∅ = faux
    x∈i(E,y) = x==y ∨ x∈E
    card(∅) = 0
    card(i(E,x)) = succ(card(E))
    s(i(E,x),y) = si x==y alors E sinon i(s(E,y),x) fsi
fspéc
```

On peut remarquer que les axiomes ne prévoient pas les cas où on est en dehors du domaine de définition. Il n'est donc plus utile, pour $card(i(E, x))$, de tester si x appartient à E.

Pour toute spécification, on considérera que l'opérateur polymorphe Def est implicitement défini par :

- pour tout opérateur f, $Def(f(x_1, \ldots, x_n)) = Def(x_1) \wedge \ldots \wedge Def(x_n) \wedge \text{pré}f(x_1, \ldots, x_n)$
- pour les termes conditionnels, $Def(\text{si } x_1 \text{ alors } x_2 \text{ sinon } x_3 \text{ fsi}) = Def(x_1) \wedge ((x_1 \wedge Def(x_2)) \vee (\neg x_1 \wedge Def(x_3))$

Remarque

Un opérateur sans pré-condition est supposé implicitement muni d'une pré-condition toujours vraie.

1.7 Suffisante complétude et consistance

Une spécification est dite « suffisamment complète » si elle n'introduit pas de nouvelle classe de congruence dans les sortes prédéfinies. Par exemple, dans la spécification NAT_1, si on arrive à isoler des termes de sorte *Bool* qui ne sont congrus ni à *vrai*, ni à *faux*, la spécification n'est pas suffisamment complète.

Une spécification est dite « consistante » si elle ne réunit pas des classes de congruences de sorte prédéfinies. Par exemple, dans la spécification NAT_1, si on arrive à montrer que *vrai* est congru à *faux*, la spécification est inconsistante.

Malheureusement ces propriétés sont indécidables dans le cas général. Autrement dit, s'il est souvent possible de montrer qu'une spécification n'est pas suffisamment complète ou qu'elle est inconsistante en exhibant des exemples, il est souvent impossible de montrer qu'une spécification est consistante ou suffisamment complète.

2. Complexité des algorithmes

2.1 Objectifs

On a souvent le choix entre plusieurs algorithmes lorsque l'on cherche à résoudre un problème donné.

Considérons, à titre d'exemple, le problème consistant à classer les éléments d'un tableau en ordre croissant. Il existe de nombreux algorithmes permettant de le résoudre. On peut citer, parmi les plus connus :

– le tri par sélection,
– le tri par insertion,
– le tri par tas,
– le tri rapide,
– le tri par fusion.

Cette liste n'est bien sûr pas exhaustive.

Il convient donc, face à une situation donnée, de choisir le bon algorithme de classement. Les critères de choix sont principalement de deux natures :

– la rapidité d'exécution du programme résultant,
– l'encombrement en mémoire de ce programme (plus précisément la taille des données temporaires qu'il utilise).

Dans le cadre de ce document, nous nous intéresserons uniquement au critère de rapidité d'exécution des programmes.

Ce critère de rapidité mérite cependant d'être précisé. Il est en effet souvent impossible d'affirmer qu'un algorithme est meilleur qu'un autre dans toutes les situations. La qualité d'un algorithme dépend de paramètres propres au problème traité qu'il importe de préciser pour effectuer un choix judicieux. On peut citer :

– La taille des données manipulées par l'algorithme
  Dans le cas d'un algorithme de classement de tableau, il s'agit du nombre d'éléments du tableau. D'une manière générale, on peut dire que la différence de rapidité entre plusieurs algorithmes permettant de résoudre un problème n'est sensible que lorsque les données manipulées sont de taille importante.
– La nature et l'organisation des données
  Les données à manipuler dans un problème ne sont pas toujours quelconques. On peut, par exemple, être amené à choisir un algorithme pour classer un tableau que l'on sait parfaitement être peu désordonné (presque classé). Le meilleur algorithme n'est alors sans doute pas celui qui va le plus vite dans un cas « moyen ».
– Des contraintes de temps maximal d'exécution.
  Dans certaines applications (typiquement les applications de contrôle de processus physiques critiques dans le domaine des transports, de la santé ou du nucléaire), le temps d'exécution ne doit pas dépasser une limite donnée. On peut alors être amené à proscrire un algorithme très bon dans la majorité des cas, mais très mauvais dans certains cas particuliers.

Il importe donc de posséder des outils permettant d'évaluer le temps d'exécution d'un algorithme, avant même d'écrire le programme correspondant.

Cette évaluation, effectuée pendant la phase d'analyse du problème, peut permettre d'éviter de s'engager dans des impasses et de gagner un temps considérable pendant la phase de réalisation d'un projet.

2.2 Complexité en temps

2.2.1 Définitions et exemples

Nous allons donc tenter d'évaluer le temps d'exécution d'un programme utilisant un algorithme donné. Une évaluation exacte de ce temps est totalement illusoire, car elle dépend de paramètres trop nombreux et trop difficiles à maîtriser. Parmi ces paramètres, nous pouvons citer :

- le langage utilisé pour coder l'algorithme.
- le programme lui-même : on désire en effet évaluer un algorithme avant d'écrire le programme correspondant. Il n'est pas possible de connaître le temps exact d'exécution d'un programme avant qu'il soit écrit.
- le compilateur utilisé : le temps d'exécution exact d'un programme ne dépend pas que de ce programme, mais aussi de la capacité du compilateur (le programme qui traduit le texte du programme en des instructions compréhensibles par l'ordinateur) à générer du code efficace.
- l'ordinateur sur lequel le programme va s'exécuter (en particulier de sa rapidité).

On désire en fait obtenir une valeur caractéristique du temps d'exécution du programme qui ne dépende que de l'algorithme utilisé, et pas de l'implémentation de ce dernier.

L'objectif recherché dans cette approche est de pouvoir comparer plusieurs algorithmes permettant de résoudre un même problème, et ceci en particulier lorsque la taille ou le nombre de données manipulées devient grand.

En effet, pour un problème donné, bien des algorithmes se comportent de manière similaire pour des données de petite dimension. Le temps d'exécution du programme correspondant est suffisamment faible pour qu'il soit relativement inutile de se préoccuper de l'efficacité de l'algorithme utilisé (par exemple quelques millisecondes).

Par contre, lorsque la taille des données manipulées devient importante, il devient capital de choisir un algorithme qui peut donner un temps d'exécution de quelques secondes, plutôt qu'un autre qui le ferait passer à quelques heures ou même quelques jours.

Dans certains cas, on cherchera plutôt à évaluer le temps d'exécution en fonction de la valeur des données. Par exemple, pour le problème consistant à élever un nombre k à une puissance entière n, la représentation informatique des données manipulées est de taille fixe. La donnée intéressante dans ce problème est l'évolution du temps d'exécution du programme quand n varie. Dans la suite de ce chapitre, nous parlerons toujours de « paramètre reflétant la taille des données », sachant que dans certains cas particuliers ce paramètre sera plutôt associé à la valeur de la donnée.

Nous allons donc chercher, pour réaliser cette mesure, une fonction dépendant de la taille des données.

Pour réaliser l'analyse d'un algorithme, nous allons mettre en évidence :

- Une opération élémentaire, choisie de manière à ce que le temps d'exécution du programme soit toujours asymptotiquement proportionnel, quand la taille des données tend vers l'infini, au nombre de fois que l'on exécute cette opération.
- Un paramètre reflétant la taille des données.

Définition 7 : complexité d'un algorithme

Soit A un algorithme et d une donnée d'entrée de A.
On appelle complexité de A pour d, et on note $C_A(d)$ le nombre de fois que l'opération élémentaire est exécutée lors de l'exécution de A sur d.

Nous allons illustrer ces notions sur un exemple : le tri par sélection.

Les algorithmes de tri ont tous un objectif commun : étant donné un ensemble sur lequel on dispose d'une relation d'ordre total que l'on nommera par la suite $\geq$, classer un tableau dont les éléments appartiennent à cet ensemble en ordre croissant (ou décroissant).

La méthode employée par le tri par sélection se fait en *tailleTab* itérations, où *tailleTab* est le nombre d'éléments du tableau à classer. Lors de la $i^{ème}$ itération, on place, classés en ordre croissant, les i plus petits éléments du tableau dans les i premières positions de ce tableau. Au début de cette itération, les $i-1$ plus petits éléments du tableau sont donc classés et occupent les $i-1$ premières positions du tableau. Il reste donc à rechercher le plus petit élément des $tailleTab - i$ éléments restant, et à l'échanger avec le $i^{ème}$ élément.

L'algorithme de classement du tableau *tab* de taille *tailleTab* par sélection peut donc s'écrire, en supposant que les indices du tableau varient entre 1 et *tailleTab* :

```
1  pour i variant de 1 à tailleTab
2    indiceDuPlusPetit ←i
3    pour j variant de i+1 à tailleTab
4      si tab[j] < tab[indiceDuPlusPetit]
5        indiceDuPlusPetit←j
6      fin si
7    fin pour
8    echanger tab[i] et tab[indiceDuPlusPetit]
9  fin pour
```

Exemple 2.1 – Algorithme du tri par sélection

Le paramètre reflétant la taille des données est ici le nombre d'éléments du tableau, c'est à dire *tailleTab*.

D'une manière générale, ce paramètre doit être choisi de manière à représenter quelque chose de significatif pour le problème traité. Si ce dernier est un problème concernant des matrices carrées, on pourra alors choisir la dimension d'une matrice ou encore son nombre d'éléments en fonction de ce qui est effectivement significatif pour la sémantique du problème. Il est clair que la valeur de la complexité dépendra de ce choix. Il est donc essentiel de préciser clairement le paramètre choisi avant de donner une complexité. Il va de soi que l'on ne pourra comparer la complexité de deux algorithmes résolvant le même problème que si on a choisi le même paramètre dans les deux calculs de complexité.

Le choix de l'opération élémentaire est moins évident. On pourrait être tenté de choisir celle qui prend le plus de temps à s'exécuter, en pensant que c'est elle qui va prédominer sur le temps d'exécution total du programme. S'il est plus long d'échanger deux éléments du tableau que de les comparer, ce qui est probablement le cas, un individu peu attentif déciderait alors que cet échange est l'opération élémentaire.

Cependant, si on dénombre le nombre d'échanges et le nombre de comparaisons d'éléments du tableau effectuées pendant le tri, on obtient :

Nombre d'échanges : *tailleTab*

Nombre de comparaisons : $(tailleTab \times (tailleTab - 1))/2$

Si te est la durée d'une opération d'échange et tc celle d'une opération de comparaison, le temps passé à faire des comparaisons sera supérieur à celui passé à faire des échanges pour toute valeur de $tailleTab$ supérieure à $2(te/tc) + 1$.

A titre d'exemple, si $te = 10 \times tc$, l'opération de comparaison devient prédominante dès que *tailleTab* dépasse 21.

Ce simple exemple montre bien qu'il importe de choisir, non pas l'opération la plus longue, mais celle qui est effectuée le plus souvent. Dans le cas d'un algorithme itératif, ce sera souvent une opération située dans la boucle la plus interne.

La complexité de notre algorithme de tri par sélection, en ayant choisi *tailleTab* comme paramètre reflétant la taille des données et la comparaison comme opération élémentaire, est donc de $(tailleTab \times (tailleTab - 1))/2$. Il est ici possible d'exprimer cette complexité en fonction du paramètre reflétant la taille des données car elle est identique pour toutes les données de même taille. Dans ce calcul, nous avons négligé certaines opérations car elles s'exécutent moins souvent que la comparaison. Il aurait été possible de les prendre en compte, mais l'ordre de grandeur du résultat, qui est en fait le seul résultat intéressant, n'aurait pas été modifié.

Cependant, pour que le calcul de complexité possède un intérêt, il est impératif que la durée d'exécution de l'opération élémentaire choisie soit indépendante du paramètre reflétant la taille des données. C'est effectivement le cas des deux opérations envisagées pour l'algorithme précédent. Imaginons cependant un langage de programmation qui offre une instruction permettant de trouver l'indice du plus petit élément d'un tableau. Il ne faudrait pas alors choisir cette instruction comme opération élémentaire, et affirmer que la complexité de l'algorithme est égale à $tailleTab$, car le temps d'exécution de cette instruction ne peut que dépendre de la taille du tableau.

2.2.2 Complexité dans le cas moyen, le meilleur et le pire

Il nous a été possible d'exprimer la complexité de l'algorithme de tri par sélection uniquement en fonction de la taille des données, sans avoir à se préoccuper de la valeur de ces données, c'est à dire de la valeur des éléments contenus dans le tableau. Le temps d'exécution d'un programme de tri par sélection ne dépend effectivement que de la taille du tableau à classer.

Cependant, pour la majorité des algorithmes, le temps d'exécution dépend aussi de la valeur ou de l'organisation des données.

Définition 8 : complexité dans le pire des cas

Soit A un algorithme et $D(n)$ l'ensemble de toutes les données d'entrée possibles de A de taille n.
On appelle complexité de A dans le pire des cas et on note $CA_{pire}(n)$ la complexité de cet algorithme dans le cas le plus défavorable, autrement dit, la valeur maximale de cette complexité pour tous les éléments de $D(n)$:
$CA_{pire}(n) = \max_{d \in D(n)} CA(d)$

La complexité dans le pire des cas est particulièrement importante, car elle permet d'évaluer les performances d'un algorithme quand les circonstances sont défavorables. Dans certains types d'applications, par exemple pour la commande de dispositifs critiques (contrôle d'attitude d'un avion, freinage d'un train ou flux de neutrons dans une centrale nucléaire), il existe des contraintes impératives imposant qu'une action soit réalisée dans un délai maximum. On peut être alors amené à rejeter un algorithme qui serait très rapide dans la majorité des cas, mais très lent dans certains cas particuliers.

Définition 9 : complexité dans le meilleur des cas

Soit A un algorithme et $D(n)$ l'ensemble de toutes les données d'entrée possibles de A de taille n.
On appelle complexité de A dans le meilleur des cas et on note $CA_{meilleur}(n)$ la complexité de cet algorithme dans le cas le plus favorable, autrement dit, la valeur minimale de cette complexité pour tous les éléments de $D(n)$.
$CA_{meilleur}(n) = \min_{d \in D(n)} CA(d)$

La complexité dans le meilleur des cas nous permet d'isoler des situations dans lesquelles un algorithme est particulièrement rapide. Si, dans le problème à traiter, on se trouve dans ces situations, on peut alors être amené à choisir un algorithme qui n'est pas très rapide dans le cas moyen, mais excellent dans les situations auxquelles on est confronté.

Par exemple, l'algorithme de tri par insertion que nous allons étudier par la suite se comporte mieux que beaucoup d'autres algorithmes de classement lorsque le tableau à classer est déjà « presque » classé. Si on se trouve dans cette situation (et ce sera souvent le cas), on aura alors sans doute intérêt à choisir cet algorithme.

Définition 10 : complexité en moyenne

Soit A un algorithme et $D(n)$ l'ensemble de toutes les données d'entrée possibles de A de taille n. On appelle complexité de A en moyenne et on note $CA_{moyen}(n)$ la complexité de cet algorithme dans le cas moyen, autrement dit, la valeur moyenne de cette complexité pour tous les éléments de $D(n)$.
$CA_{moyen}(n) = \frac{\sum_{d \in D(n)} CA(d)}{card(D(n))}$

On remarquera que l'expression précédente n'est valable que si $D(n)$ est un ensemble fini. Un ordinateur étant un dispositif fini, ce sera toujours le cas dans les problèmes traités. La complexité en moyenne est l'outil de mesure qu'il faudra utiliser lorsque la configuration et la valeur des données peut être quelconque, et qu'il n'existe pas de contrainte sur le temps maximum d'exécution du programme.

Considérons à titre d'exemple l'algorithme de classement connu sous le nom de tri par insertion.

Le tri par insertion se fait lui aussi en *tailleTab* itérations, où *tailleTab* est le nombre d'éléments du tableau à classer. Au début de la $i^{ème}$ itération, les $i-1$ premiers éléments du tableau sont classés en ordre croissant, mais ce ne sont pas nécessairement les $i-1$ plus petits éléments du tableau. On place alors le $i^{ème}$ élément du tableau à sa place dans ce sous-tableau, afin d'obtenir un sous-tableau classé de i éléments. Pour ce faire, tous les éléments du sous-tableau plus grands que l'élément à placer sont décalés d'une position vers la fin du tableau, et ce dernier est mis dans le trou ainsi laissé. On constate donc qu'à chaque itération, les i premiers éléments ne sont pas les i plus petits du tableau original (comme c'est le cas pour le tri par sélection) mais ceux qui occupaient effectivement les i premières positions avant le début du classement.

Cette méthode est utilisée dans l'algorithme de l'exemple 2.2.

On peut ici choisir indifféremment la copie ou la comparaison d'éléments comme opération élémentaire, car les nombres de fois où elles sont exécutées ne diffèrent que d'une unité, ce qui ne change pas l'ordre de grandeur de la complexité. Nous verrons par la suite que seul cet ordre de grandeur présente en réalité un intérêt. Choisissons la copie d'éléments.

Le paramètre reflétant la taille des données est toujours le nombre d'éléments du tableau.

On constate ici que la complexité de cet algorithme dépend de l'ordre initial des éléments dans le tableau.

Si l'élément à déplacer est plus petit que tous ses prédécesseurs, la boucle *tant que* va s'exécuter $i-1$ fois, entraînant donc $i-1$ copies d'éléments. Si on ajoute la copie réalisée au début de la boucle *pour* et celle réalisée à la fin, cette itération de cette boucle réalise $i+1$ copies. Si pour toutes les itérations de la boucle *pour* l'élément à déplacer est plus petit que ceux qui le précèdent, le nombre de copies est donc égal à la somme des entiers compris entre 3 et $tailleTab+1$, c'est à dire $((tailleTab+1)(tailleTab+2))/2-3$. Il s'agit de la complexité dans le pire des cas de l'algorithme de tri par insertion. Cette situation se produit en particulier lorsque le tableau est initialement classé en ordre décroissant.

$C_{pire}(tailleTab) = ((tailleTab+1)(tailleTab+2))/2-3$

A l'opposé, si le tableau est initialement classé en ordre croissant, on n'exécutera jamais le corps de la boucle *tant que*, chaque itération de la boucle *pour* faisant donc 2 copies. La complexité dans le meilleur des cas de l'algorithme est donc de $(tailleTab-1)\times 2$

$C_{meilleur}(tailleTab) = (tailleTab-1)\times 2$

```
1  pour i variant de 2 à tailleTab
2    elementADeplacer ←tab[i]
3    j ←i - 1
4    tant que j ≥1 et que tab[j] > elementADeplacer
5      tab[j+1] ←tab[j]
6      j ←j-1
7    fin tant que
8    tab[j+1] ←elementADeplacer
9  fin pour
```

Exemple 2.2 – Algorithme du tri par insertion

Nous allons maintenant étudier la complexité en moyenne de cet algorithme.

A chaque itération de la boucle *pour*, on insère un élément dans le sous-tableau de *tab* composé des éléments dont les indices sont compris entre 1 et $i-1$. L'insertion se fait à une position k comprise entre 1 et i, entraînant $i-k$ copies d'éléments dans la boucle *tant que*. L'itération de la boucle pour réalise donc $i-k+2$ copies.

Intuitivement, on peut dire que le nombre de configurations du sous-tableau entraînant une insertion à l'indice k est identique pour toute valeur de k (cela peut se démontrer rigoureusement). Si on nomme N ce nombre de configurations, le nombre moyens de copies pour une itération de la boucle *pour* est donc égal à :

$$\frac{\sum_{k=1}^{i} N\times(i-k+2)}{N\times i} = \frac{\sum_{k=1}^{i} i-k+2}{i} = \frac{i+3}{2}$$

Le nombre de copies réalisées en moyenne pendant le tri, et donc sa complexité en moyenne, est de :

$$C_{moyen}(tailleTab) = \sum_{i=2}^{tailleTab} \frac{i+3}{2} = \frac{tailleTab^2}{4} + \frac{7\times tailleTab}{4} - 2$$

2.2.3 Ordres de grandeur

Un des objectifs du calcul de complexité d'un algorithme est de permettre la comparaison de plusieurs algorithmes résolvant le même problème. Nous avons pu constater que les calculs menés ne permettent pas de déterminer de manière exacte les temps d'exécution des programmes. Seule une opération élémentaire est prise en compte, et cette opération peut être différente pour deux algorithmes résolvant le même problème. Clairement, les

facteurs constants additifs ou même multiplicatifs ne doivent pas être pris en compte lors de la comparaison des performances de plusieurs algorithmes. Par contre, l'ordre de grandeur de la complexité va nous donner des indications très intéressantes.

Considérons à nouveau les problèmes de classement de tableau. Les deux algorithmes étudiés précédemment possèdent une complexité en moyenne de la forme $an^2 + bn + c$, où n est le nombre d'éléments du tableau à classer. Nous verrons par ailleurs un algorithme, connu sous le nom de tri rapide, dont la complexité en moyenne est de la forme $a'n\log(n) + b'$.

On constate que, quelles que soient les valeurs de a, b, c, a' et b', il existe une valeur de n pour laquelle la première expression devient supérieure à celle de la deuxième. Lorsque l'on travaille sur des tableaux de grande dimension, on aura donc intérêt à choisir un algorithme de la deuxième catégorie. On dira que le premier type d'algorithme est en n^2 et le deuxième en $n\log(n)$.

2.2.3.1 Notations en O et Θ

Définition 11 : domination asymptotique

Une fonction f de $\mathbb{N}^*$ dans $\mathbb{R}$ est dominée asymptotiquement par une fonction g, elle aussi de $\mathbb{N}^*$ dans $\mathbb{R}$ si, et seulement si :
$\exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^*$ tels que $\forall n \in \mathbb{N}^*, n > n_0 \Rightarrow f(n) \leq c \times g(n)$
On utilisera alors la notation $f = \mathrm{O}(g)$, et dira que « f est en grand O de g ».

Les complexités dans le meilleur des cas, le pire des cas et en moyenne du tri par sélection sont donc en $\mathrm{O}(n^2)$. Les complexités dans le pire des cas et en moyenne du tri par insertion sont en $\mathrm{O}(n^2)$, et la complexité dans le meilleur des cas est en $\mathrm{O}(n)$.

Cependant, ces complexités sont toutes dominées asymptotiquement par des fonctions d'ordre de grandeur supérieur, et on peut donc en particulier aussi dire qu'elle sont toutes en $\mathrm{O}(n^3)$ et en $\mathrm{O}(e^n)$.

Définition 12 : ordre de grandeur

Deux fonctions f et g de $\mathbb{N}^*$ dans $\mathbb{R}$ ont même ordre de grandeur asymptotique si, et seulement si :
$\exists c \in \mathbb{R}^{+*}, \exists d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}^*$ tels que $\forall n \in \mathbb{N}^*, n > n_0 \Rightarrow d \times g(n) \leq f(n) \leq c \times g(n)$
On utilisera alors la notation $f = \Theta(g)$, et on dira que « f est en grand thêta de g ».

Les complexités dans le meilleur des cas, le pire des cas et en moyenne du tri par sélection sont donc en $\Theta(n^2)$.

Les complexités dans le pire des cas et en moyenne du tri par insertion sont en $\Theta(n^2)$), et la complexité dans le meilleur des cas est en $\Theta(n)$).

2.2.3.2 Échelle de comparaison

Nous allons envisager ici un ensemble de fonctions représentant les ordres de grandeur de complexité les plus classiques, formant ainsi une échelle de comparaison pour les algorithmes. Pour toutes ces fonctions, n est le paramètre reflétant la taille des données. Chaque fonction envisagée est d'un ordre de grandeur supérieur à la précédente (elle la domine asymptotiquement).

– $f(n) = 1$
  On parle dans ce cas de complexité constante. Toute fonction constante est en $\Theta(1)$.

- $f(n) = \log(n)$
  On parle dans ce cas de complexité logarithmique. En particulier, toute fonction f telle que $f(n) = a\log(n) + b$, où a et b sont des constantes est en $\Theta(\log(n))$. La propriété $1 = \mathrm{O}(\log(n))$ est vérifiée.
- $f(n) = n$
  On parle dans ce cas de complexité linéaire. En particulier, toute fonction f telle que $f(n) = an + b$, où a et b sont des constantes est en $\Theta(n)$. La propriété $\log(n) = \mathrm{O}(n)$ est vérifiée.
- $f(n) = n\log(n)$
  En particulier, toute fonction f telle que $f(n) = an\log(n) + bn + c$, où a, b et c sont des constantes est en $\Theta(n\log(n))$. La propriété $n = \mathrm{O}(n\log(n))$ est vérifiée.
- $f(n) = n^2$
  En particulier, toute fonction f telle que $f(n) = an^2 + bn + c$, où a, b et c sont des constantes est en $\Theta(n^2)$. La propriété $n\log(n) = O(n^2)$ est vérifiée.
- $f(n) = n^k$
  On parle dans ce cas de complexité polynomiale. Pour toute valeur de k supérieure à 2, cette fonction domine asymptotiquement toutes les précédentes.
- $f(n) = 2^n$
  On parle dans ce cas de complexité exponentielle. Cette fonction domine asymptotiquement toutes les précédentes.

A titre d'exemple, et afin mettre en évidence toute l'importance de ces calculs de complexité, nous allons donner les évolutions du temps d'exécution d'un programme utilisant l'algorithme de tri par insertion et d'un autre utilisant l'algorithme du tri rapide (qui est étudié par ailleurs) en fonction de la taille du tableau à classer. Nous ajouterons à ces programmes un hypothétique programme utilisant un algorithme exponentiel, afin de monter à quel point un tel algorithme devient vite inutilisable. Les temps indiqués sont ceux observés sur un ordinateur exécutant environ cent millions d'instructions par secondes, c'est-à-dire un ordinateur dont les performances sont très raisonnables au jour où ce chapitre est écrit.

	Complexité	Taille du tableau		
		100	10000	100000
Tri par insertion dans cas le meilleur	$\Theta(n)$	0,09 ms	9 ms	90 ms
Tri par insertion dans le cas moyen	$\Theta(n^2)$	1,75 ms	17 s	29 mn
Tri rapide dans le cas moyen	$\Theta(n\log(n))$	0,5 ms	100 ms	1,2 s
Tri rapide dans le cas le pire	$\Theta(n^2)$	2 ms	20 s	33 mn
Algorithme exponentiel	$\Theta(2^n)$	1 ms	5×10^{2969} jours	énorme!!!

On constate de manière évidente qu'un algorithme exponentiel devient très vite inutilisable lorsque la taille des données croît. L'amélioration des performances des ordinateurs ne saurait représenter une solution à ce problème (une multiplication par 1024 de la vitesse de l'ordinateur ne permettrait que d'ajouter 10 au nombre de données traitées dans le même temps). Il convient donc d'éviter de tels algorithmes dès que le problème posé le permet (ce qui n'est pas toujours le cas).

On remarquera que l'algorithme de tri par insertion est loin d'être ridicule devant un tri rapide lorsque la configuration des données est appropriée.

Une dernière remarque concerne le tri rapide qui, s'il est probablement le meilleur algorithme de classement connu à ce jour pour le cas moyen, devient assez mauvais si par malheur les données sont dans une configuration défavorable. On lui préférera alors un autre algorithme, comme le tri pas tas (heap sort), dont la complexité est en $\Theta(n\log(n))$ dans le cas moyen comme dans le cas le pire.

3. Algorithmes à essais successifs - Back tracking

3.1 Problèmes concernés - Modélisation

Nous allons présenter le problème sur un exemple, puis nous le traiterons ensuite dans toute sa généralité.

Considérons le problème connu sous le nom de problème des huit reines.

On dispose d'un échiquier (de dimension 8 sur 8), et on désire placer 8 reines sur cet échiquier, de telle sorte qu'aucune ne soit en situation de prise.

On rappelle qu'une reine peut prendre sur sa colonne, sur sa ligne et sur ses deux diagonales. Il s'agit donc de placer les reines afin qu'il n'y ait jamais deux reines sur la même ligne, la même colonne ou la même diagonale.

On va placer les reines une par une, en vérifiant à chaque étape que la reine placée n'est pas en situation de prise, et en revenant en arrière à chaque fois que l'on se trouve bloqué (c'est-à-dire en enlevant des reines posées précédemment).

Ce style d'algorithme est appelé algorithme à essais successifs, ou encore algorithme de back tracking. Remarquons dès à présent que la complexité de ce genre d'algorithme est de nature exponentielle, et qu'ils ne doivent donc être utilisés que si l'on n'en connaît pas de meilleur.

D'une manière générale, un algorithme à essais successifs consiste donc à construire progressivement une solution à un problème donné, en essayant toutes les possibilités à chaque étape de la construction.

Les problèmes concernés sont ceux où :
- On part d'une situation initiale S_0, pour aller à une situation finale S_f qui répond au problème. Plusieurs situations peuvent éventuellement répondre au problème.
- La situation finale est atteinte par le passage successif par les situations S_1 à $S_f - 1$, qui doivent être toutes différentes.
- Le passage de la situation S_i à la situation S_{i+1} se fait par l'application d'une action.
- Pour chaque situation S_i, on peut avoir le choix entre plusieurs actions menant à des situations éventuellement différentes

Pour modéliser ce genre de problème par un algorithme à essais successifs, on doit disposer de :

Une fonction *succès*, qui permet de savoir si une situation répond ou non au problème.
- Une fonction *ensembleChoix*, qui permet d'obtenir, à partir d'une situation donnée, l'ensemble des actions possibles.
- Une fonction *successeur*, qui donne la situation obtenue par l'application de l'action A à la situation S.

On appelle *chemin* une suite d'actions $a_1, a_2, \ldots, a_n$.

Le problème consiste à trouver un chemin qui mène de la situation initiale S_0 à une situation finale S_f. Diverses variantes du problème existent ; citons, par exemple, celle qui consiste à trouver des chemins qui mènent à toutes les situations finales, ou encore celle qui consiste à trouver un chemin vers la meilleure situation finale (dans le cas où on peut donner une valeur à une situation).

Un chemin menant à une situation finale peut être vide, auquel cas la situation initiale répond au problème.

On introduit de plus une constante de type *Chemin* nommée ERR_{Chemin}, représentant un chemin ne menant nulle part .

Le principe d'un algorithme à essais successifs est assez simple. Il est cependant toujours indispensable de faire une analyse assez fine du problème afin d'éviter les erreurs potentielles principales :

- Il ne faut éliminer aucune solution potentielle
- Il faut prendre soin de ne jamais introduire de boucle dans la recherche. Cela signifie qu'il ne faut jamais, dans un même chemin, passer plusieurs fois par la même situation (par contre, bien que cela nuise considérablement à l'efficacité, il est possible de passer par la même situation dans deux chemins différents).

3.2 Algorithmes

L'algorithme à essais successifs d'une fonction permettant de trouver un chemin menant à une situation finale, et rendant ce chemin et cette situation peut être :

```
1  aes(s:Situation) résultat : (Chemin,Situation)
2    si succès(s) alors
3      résultat←(vide,s)
4    sinon
5      choix←ensembleChoix(s)
6      chemin←erreurChemin
7      tant que choix n'est pas vide et chemin est égal à erreurChemin faire
8        action←un élément de choix
9        choix←choix - {action}
10       (chemin,sf)←aes(successeur(s,action))
11     fin tant que
12     si chemin est égal à erreurChemin alors
13       résultat←(erreurChemin,erreurSituation)
14     sinon
15       résultat←(ajouter(action, chemin),sf)
16     fin si
17   fin si
18   retour
19 fin aes
```

Exemple 3.1 – Algorithme à essais successifs

Cette fonction reçoit une situation en paramètre, et cherche un chemin partant de celle ci et menant à une situation finale.

On remarque que le retour en arrière évoqué précédemment est géré de façon élégante par la récursivité. La pile d'exécution contient, entre autres informations, la pile des situations menant à la situation courante.

Ce style d'algorithme peut cependant être implémenté à l'aide d'un langage ne supportant pas la récursivité, en gérant explicitement une pile de situations, ou encore en construisant un arbre ayant pour racine la situation initiale, et dans lequel chaque nœud possède pour enfants les situations pouvant être atteintes à partir de la situation contenue dans ce nœud (par application des différentes actions rendues par *ensembleChoix*). La méthode récursive ne réalise rien d'autre que l'exploration de cet arbre !

L'opération *ajouter* utilisée dans cet algorithme permet d'ajouter un élément en tête d'une liste. On remarque que la construction du chemin se fait en retour de récursivité. Si la liste des actions allant de la situation s_0 à la situation s_f est $a_1, a_2, \ldots, a_f$, on place donc les actions dans le chemin en commençant par a_f et en terminant par a_1. Les actions sont donc dans la liste dans l'ordre où elles doivent être effectuées pour aller de s_0 à s_f (a_1 est en tête de liste).

La fonction*successeur* ne doit pas modifier la situation qu'elle reçoit en paramètre. Les diverses actions obtenues par la fonction *ensembleChoix* doivent en effet être appliquées à la situation reçue en paramètre par *aes*, afin de pouvoir explorer toutes les branches partant de cette situation. Si, pour une raison quelconque (par exemple, d'efficacité de l'implémentation), la fonction *successeur* modifie la situation reçue, il faut alors disposer d'une fonction ou d'une procédure permettant d'annuler les effets d'une action sur une situation (et donc de faire un retour en arrière, d'où le nom de back-tracking donné à ces algorithmes).

La ligne :

```
1  (chemin,sf)←aes(successeur(s,action))
```

doit alors être remplacée par :

```
1  successeur(s,action)
2  (chemin,sf)←aes(s)
3  predecesseur(s,action)
```

où la fonction *successeur* a été transformée en procédure.

Afin de réduire le temps d'exécution de cette fonction, qui est par nature très grand, il importe le soigner particulièrement la façon de déterminer les actions possibles à partir d'une situation donnée (fonction *ensembleChoix*).

Tout d'abord, il est recommandé d'essayer les différentes actions possibles dans une situation donnée en commençant par celles qui ont le plus de « chances » de mener à une situation finale, ce qui permettra d'arrêter la boucle plus tôt, et donc de diminuer considérablement le temps de traitement.

D'autre part, s'il est possible de déterminer qu'une action ne peut pas mener à une situation finale, on doit alors l'éliminer de la liste rendue par la fonction *ensembleChoix*. Ce sera le cas dans les problèmes pour lesquels il faut respecter une contrainte, et où il est donc inutile de choisir une solution qui mène à une situation pour laquelle cette contrainte n'est pas respectée, et ne peut plus l'être quelle que soient les actions envisagées par la suite.

Pour le problème des huit reines, si on choisit de partir d'un échiquier vide, et si on décide que la seule action possible consiste à poser une reine sur une case (déplacement de reine interdit), il est alors inutile d'envisager les actions qui conduisent à un échiquier sur lequel une reine est en position de prise.

Pour trouver un chemin menant à une situation finale on fera donc :

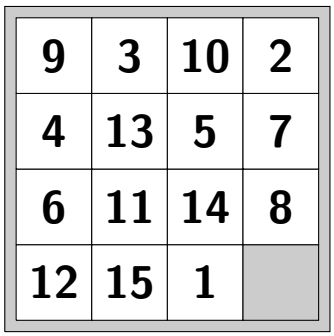
```
1  (chemin,sf)←aes(s0)
```

Les algorithmes permettant de résoudre les problèmes où seul le chemin menant à une situation finale est utile, et ceux où seule la situation finale importe, se déduisent aisément du précédent.

3.2.1 Exemple du jeu de Taquin

Nous allons illustrer les notions qui viennent d'être présentées sur un problème classique pouvant être traité grâce à un algorithme à essais successifs : le jeu de « taquin ».

On dispose d'un plateau de seize cases, sur lequel sont disposées quinze pièces carrées, numérotées de 1 à 15.



Une pièce peut être déplacée s'il est possible de la faire glisser à l'emplacement de la case vide. Dans l'exemple ci-dessus, il est possible de déplacer les pièces 1 et 8.

Le but du jeu est de réorganiser les pièces par glissements successifs, jusqu'à ce qu'elles soient classées par ordre de numéro.

Dans ce problème, une situation est un plateau de jeu sur lequel chacune des 15 pièces occupe une position donnée. Une situation finale est une situation pour laquelle les pièces sont ordonnées. Il n'existe donc ici qu'une seule situation finale. La fonction succès doit donc rendre *vrai* pour cette situation, et *faux* pour toute autre.

Une action consiste à déplacer une des pièces voisines de la case libre d'une position. Le successeur d'une situation est donc le plateau de jeu obtenu après glissement de la pièce choisie.

Un chemin est donc une suite de déplacements de pièces, chaque déplacement pouvant être représenté par les coordonnées de la pièce bougée, et la direction du déplacement.

On se trouve ici typiquement dans le cas où seul le chemin menant à une situation finale nous intéresse, la situation finale elle même étant unique et parfaitement connue dès le départ. On modifie donc l'algorithme précédant en imposant à la fonction *aes* de ne rendre que le chemin menant de la situation reçue à la situation finale.

On se trouve cependant ici confronté à un problème particulier. La fonction *aes* ne s'exécutera en un temps fini que si on peut garantir que, pour toute situation reçue, on effectue un nombre fini de récursions. Autrement dit, il faut être sur qu'au bout d'un nombre fini de récursions on tombe sur une situation gagnante ou sur une situation pour laquelle aucune action n'est possible. Ce ne sera le cas que si l'on prend soin d'imposer, comme cela a été exposé au paragraphe précédant, de ne jamais passer par deux situations identiques.

Dans de nombreux problèmes, tel celui des huit reines ou encore celui du cavalier d'échec qui sera exposé dans un paragraphe suivant, les différentes situations rencontrées lors du parcours d'un chemin ne peuvent être que différentes. Dans le problème du taquin, une solution simple, mais extrêmement coûteuse, consiste à mémoriser toutes les situations rencontrées, et à imposer à la fonction *ensembleChoix* de ne pas rendre d'action menant à ces situations. Cette solution ne doit, bien sûr, être choisie qui si on ne possède pas d'autre moyen de garantir l'absence de bouclage.

3.2.2 Recherche de tous les chemins

Un algorithme à essais successifs permettant de trouver tous les chemins menant à une situation finale peut être :

```
1  aes(s:Situation, chemin:Chemin, listeSituations: Liste de situations,
2     listeChemins:Liste de chemins)
3     résultat :(Liste de situations,Liste de chemins)
```

```
 4
 5    si succès(s) alors
 6      résultat←(ajouter(s,listeSituations), ajouter(chemin, listeChemins))
 7    sinon
 8      choix←ensembleChoix(s)
 9      tant que choix n'est pas vide faire
10        action←un élément de choix
11        choix←choix - {action}
12        chemin←ajouter(action,chemin)
13        (listeSituations, listeChemins)←aes(successeur(s,action), chemin,
14          listeSituations, listeChemins))
15        chemin←supprimerPremier(chemin)
16      fin tant que
17      résultat←(listeSituations, listeChemins)
18    fin si
19    retour
20  fin aes
```

On remarque que la construction du chemin se fait à l'aller de la récursivité. Si la liste des actions allant de la situation s_0 à la situation s_f est $a_1, a_2, \ldots, a_f$, on place donc les actions dans le chemin en commençant par a_1 et en terminant par a_f. Les actions sont donc dans la liste dans l'ordre inverse où elles doivent être effectuées pour aller de s_0 à s_f (a_f est en tête de liste).

L'ordre d'exploitation des actions rendues par *ensembleChoix* peut ici être quelconque, car toutes les actions seront étudiées. Il importe cependant toujours d'éliminer dès que possible les actions ne pouvant mener à une situation finale.

Pour trouver tous les chemins menant à une situation finale on fera donc :

```
1   (listeChemins,listeSituations)←aes(s0, chemin vide,
2     liste vide de situations, liste vide de chemins))
```

Les algorithmes permettant de résoudre les problèmes où seul le chemin menant à une situation finale est utile, et ceux où seule la situation finale importe, se déduisent aisément du précédent.

3.2.3 Recherche du meilleur chemin

On peut aussi concevoir un algorithme permettant de trouver le meilleur chemin conduisant à une situation finale (ou la meilleure situation), si on sait donner une valeur à un chemin (ou à une situation).

Par exemple, un algorithme permettant de trouver la meilleure situation (en réalité une situation de valeur maximale) peut être :

```
 1  aes(s:Situation, meilleure:Situation) : Situation
 2    si succès(s) et valeur(s) > valeur(meilleure) alors
 3      meilleure←s
 4    sinon
 5      choix←ensembleChoix(s)
 6      tant que choix n'est pas vide faire
 7        action←un élément de choix
 8        choix←choix - {action}
 9        successeur(s,action)
10        meilleure←aes(s,meilleure)
11        predecesseur(s,action)
12      fin tant que
13    fin si
14    resultat←meilleure
15    retour
16  fin aes
```

Pour trouver la meilleure situation on fera donc :

```
1  meilleure←mauvais
2  s←aes(s0,meilleure)
```

On a utilisé dans cet algorithme une opération *valeur*, supposée donner une valeur nulle pour une situation non finale, et une valeur strictement positive pour une situation finale. La constante *mauvais* est une situation a priori non finale, et donc de valeur 0.

On peut utiliser cet algorithme pour trouver le plus court chemin allant d'un sommet à un autre dans un graphe valué, pour lequel chaque arête (ou arc) possède une valeur positive (ce n'est qu'un exemple, cette méthode étant particulièrement inefficace pour résoudre ce problème).

Une optimisation très intéressante est possible si le successeur d'une situation est toujours de valeur inférieure ou égale à la valeur de cette situation. Il suffit alors de mémoriser la valeur de la meilleure situation finale obtenue (valeur réactualisée à chaque fois que l'on rencontre une situation finale), et d'imposer à *ensembleChoix* de rendre un ensemble vide si elle reçoit une situation de valeur inférieure ou égale à ce maximum. Cela ne peut s'appliquer qu'aux problèmes où il est possible de donner une valeur aux situations intermédiaires. C'est le cas du problème du plus court chemin entre deux sommets d'un graphe.

Cet algorithme devra cependant être modifié plus profondément si une situation répondant au problème peut posséder des successeurs, et donc mener à d'autres situations répondant au problème.

Un cas particulier intéressant est celui ou toute situation répond au problème, et pour lequel la seule condition d'arrêt de la récursivité est l'impossibilité de trouver une action à appliquer à la situation reçue. C'est le cas du problème dans lequel on dispose de N nombres positifs, et où on cherche un sous-ensemble de ces N nombres dont la somme est inférieure ou égale à un nombre K donné, mais le plus proche possible de K. L'algorithme précédant devient :

```
 1  aes(s:Situation,meilleure:Situation) résultat:Situation
 2    si valeur(s) > valeur(meilleure) alors meilleure ←s
 3    choix←ensembleChoix(s)
 4    tant que choix n'est pas vide faire
 5      action←un élément de choix
 6      choix←choix - {action}
 7      successeur(s,action)
 8      aes(s,meilleure)
 9      predecesseur(s,action)
10    fin tant que
11    retour
12  fin aes
```

Remarque :

Ces algorithmes (et leur dérivés) peuvent être simplifiés en fonction du problème traité. En particulier, la constitution (par la fonction *ensembleChoix*) de l'ensemble des actions possibles, et son exploration élément par élément, peuvent souvent être remplacées par une boucle où on envisage toutes les actions possibles sans en créer l'ensemble précédemment.

3.3 Un exemple : Le problème du cavalier d'échec

Ce problème consiste à faire parcourir à un cavalier les 64 cases d'un échiquier sans passer deux fois par la même case. La position de départ peut être quelconque. L'échiquier est considéré comme étant « circulaire » : la colonne de droite est voisine de la colonne de gauche, et la ligne du haut est voisine de la ligne du bas (au niveau des déplacements du cavalier). On va écrire un algorithme permettant de trouver un chemin menant à une situation finale.

Pour ce problème, un chemin est une liste de positions sur l'échiquier.

Une situation est un échiquier où l'on trouvera, entre autres informations, les cases par lesquelles le cavalier est déjà passé. Diverses opérations devront être possibles sur une

situation. Celles-ci seront mises en évidence lors de l'écriture de l'algorithme. On en déduira alors la représentation mémoire de la situation conduisant à la réalisation la plus efficace possible de ces opérations.

L'algorithme peut être :

```
 1  cavalier(s:Situation) résultat:Chemin
 2   si les 64 cases de s sont explorées alors
 3    résultat←vide
 4   sinon
 5    (x,y)←position du cavalier sur s
 6    chemin←erreur
 7    i←0
 8    tant que i < 8 et chemin = erreur faire
 9     (xs, ys)←ième case possible pour un déplac. à partir de (x,y)
10     si la case (xs,ys) de s est libre alors
11      déplacer le cavalier de s en (xs,ys) et marquer cette case prise
12      chemin←cavalier(s)
13      reculer le cavalier sur s et marquer (xs,ys) libre
14     fin si
15     i←i + 1
16    fin tant que
17    si chemin = erreur alors
18     résultat←erreur
19    sinon
20     résultat←ajouter((xs,ys), chemin)
21    fin si
22   fin si
23   retour
24  fin cavalier
```

Le programme principal étant :

```
1  s←créer
2  chemin←cavalier(s)
```

On met donc en évidence dans cet algorithme les opérations suivantes devant être effectuées sur une situation :

`créer`	`:` → `Echiquier`	*Crée un échiquier vide*
`plein`	`: Echiquier` → `booléen`	*Rend vrai si les 64 cases de l'échiquier ont été parcourues*
`dernierx`	`: Echiquier` → `Abscisse`	*Rendre l'abscisse de la case où se trouve le cavalier*
`derniery`	`: Echiquier` → `Ordonnee`	*Rendre l'ordonnée de la case où se trouve le cavalier*
`caseLibre`	`: Echiquier,Abscisse,Ordonnee` → `booléen`	*Indique si une case donnée de l'échiquier est libre*
`avancer`	`: Echiquier,Abscisse,Ordonnee` → `Echiquier`	*Déplace le cavalier vers une position donnée*
`reculer`	`: Echiquier,Abscisse,Ordonnee` → `Echiquier`	*Annule le dernier mouvement du cavalier*

Afin que ces opérations puissent être réalisées de façon efficace, une situation pourra être représentée en mémoire par une structure contenant :
- le nombre de cases par lesquelles le cavalier est passé
- la position actuelle du cavalier
- un tableau de dimension 8 par 8 de booléens, indiquant si une case est libre ou non.
- Une pile de coordonnées, si on désire introduire une opération permettant d'annuler le dernier déplacement effectué.

On supposera spécifié et implémenté par ailleurs le type liste de positions, utilisé pour modéliser les chemins.

On pourra se contenter d'envisager, dans les équations spécifiant les opérations, les seules exceptions pouvant se produire dans l'algorithme.

4. Le type liste

4.1 Définition

Définition 13 : Liste (définition « itérative »)

Une liste est une collection totalement ordonnée de données. La relation d'ordre ne porte pas sur la valeur des données, mais sur leur position dans la liste.

Il faut donc voir une liste comme une collection de « boîtes » (on dit plutôt de conteneurs), chaque boite possédant un rang et contenant une donnée quelconque.

Cette définition est de nature « itérative », et incitera à définir sur le type abstrait algébrique associé des opérations traduisant ce caractère itératif :

- insérer un nouvel élément à une position donnée
- obtenir l'élément situé à une position donnée
- supprimer l'élément situé à une position donnée
- connaître le nombre d'éléments contenus dans la liste

Cette énumération n'est nullement limitative, et toute opération jugée utile pourra être ajoutée en étendant le type abstrait algébrique.

On peut toutefois donner une autre définition de cette notion de liste, possédant un caractère plutôt récursif, ressemblant fort à ce que l'on trouve dans des langages comme LISP.

Définition 14 : Liste (définition « récursive »)

Une liste est :
- soit vide,
- soit composée d'un élément suivi d'une liste.

Cette définition incite alors plutôt à introduire des opérations de type *car* ou *cdr*. Toutefois, toute autre opération jugée utile pourra ici aussi être ajoutée en étendant le type abstrait algébrique.

4.2 Spécifications algébriques

4.2.1 Définition itérative

Nous allons donner ici une spécification comportant un nombre minimal d'opérateurs basés sur la définition itérative de la liste. Ces opérateurs devront suffire pour permettre de définir tout autre opérateur jugé utile par la suite en utilisant les opérateurs de cette spécification minimale.

Nous allons prendre soin, comme dans toute spécification, de définir une ou plusieurs constantes et au moins un constructeur.

Ici, la seule constante utile est la liste vide, que nous allons matérialiser par un opérateur de nom *listevide*.

La définition itérative de la liste nous conduit à introduire un constructeur qui a pour effet d'ajouter un élément à une liste à une position donnée.

Cet opérateur est bien un constructeur puisqu'il est possible de construire toute liste à partir d'une liste vide en y ajoutant successivement tous ses éléments. Nous nommons cet opérateur *insérer*.

Nous ajoutons de plus les trois opérations suivantes :
– *supprimer*, permettant de supprimer d'une liste l'élément situé à une position donnée,
– *ième*, permettant d'obtenir l'élément situé à une position donnée,
– *longueur*, donnant le nombre d'éléments de la liste.
On suppose de plus définis par ailleurs un type *Position* (type entier représentant une position dans la liste), un type *Taille* (type entier représentant un nombre d'éléments), et un type *Elément* (représentant ce que l'on met dans la liste).

La signature de cette spécification est donc :

```
spéc LISTE_ITER
  étend ELEMENT, POSITION, TAILLE
  sorte Liste
  opérations
    listevide :                        →Liste
    longueur  :Liste                   →Taille
    insérer   :Liste,Position,Elément →Liste
    supprimer :Liste,Position          →Liste
    ième      :Liste,Position          →Elément
```

Certaines opérations n'étant pas définies pour toute position, il faut penser à préciser leur domaine de définition en ajoutant des pré-conditions.

```
  préconditions
    pré insérer(l,p,e) = p ≥ 1 ∧ p ≤ longueur(l) + 1
    pré supprimer(l,p) = p ≥ 1 ∧ p ≤ longueur(l)
    pré ième(l,p) = p ≥ 1 ∧ p ≤ longueur(l)
```

Nous avons choisi de numéroter les éléments d'une liste l entre *1* et *longueur(l)*. De plus, une insertion en position p doit insérer l'élément devant celui situé actuellement en position p. Par convention, une insertion en position *longueur(l)* $+ 1$ ajoute le nouvel élément en fin de liste.

Il nous reste à donner les axiomes définissant les propriétés des opérateurs. Normalement, il suffit de donner, pour tout opérateur qui n'est ni une constante ni un constructeur, un axiome donnant le résultat de cette opération pour chaque constante, et un axiome donnant le résultat de l'opération pour les termes construits. On a alors défini le fonctionnement de l'opération pour tout terme de la sorte en cours de définition.

Cette méthode s'applique bien lorsque tout terme clos peut se mettre sous une forme unique d'utilisations successives du constructeur. Par exemple, si l'on reprend la spécification du type *Nat*, tout entier naturel peut s'exprimer une forme unique d'applications successives de l'opérateur *succ* à la constante 0.

Ici, notre constructeur est l'opération `insérer`. S'il est effectivement possible de représenter toute liste sous la forme d'une succession d'insertion sur une liste au départ vide, cette forme n'est bien évidemment pas unique.

L'axiome li_1 a pour but d'indiquer quels sont les termes composés d'une succession d'appels à `insérer` à une liste vide qui sont congrus entre eux. Il indique simplement que, à partir d'une liste `l`, on peut obtenir la même liste en insérant d'abord l'élément e_1 puis l'élément e_2, ou d'abord l'élément e_2 puis l'élément e_1.

Cet axiome permet de démontrer par induction que tout terme de sorte *Liste* peut se mettre sous la forme :

– soit de la liste vide
– soit `insérer(l,p,e)` où `p` peut prendre une valeur quelconque entre 1 et la longueur de la liste.

Ainsi, pour spécifier le fonctionnement d'un opérateur, il suffit de donner un axiome pour `listevide` (si c'est dans le domaine de définition de l'opérateur), et pour le terme `insérer(l,p,e)` où `p` peut prendre la valeur qui nous arrange.

```
   axiomes l:Liste;p,p1,p2:Position;e,e1,e2:Element
     (li1) insérer(insérer(l,p1,e1),p2,e2)= si p1 < p2
                   alors insérer(insérer(l,p2-1,e2), p1,e1)
                   sinon insérer(insérer(l,p2,e2),p1+1, e1) fsi
     (li2) supprimer(insérer(l,p,e),p) = l
     (li3) longueur(listevide) = 0
     (li4) longueur(insérer(l,1,e)) = longueur(l)+1
     (li5) ième(insérer(l,p,e),p) = e
 fspéc
```

L'axiome li_2 doit en principe exprimer quel doit être le résultat de l'opération *supprimer* sur une liste construite quelconque à une position p quelconque. Une liste construite quelconque est de la forme `insérer(l,p1,e1)`. L'axiome li_1 nous permet d'affirmer que cette liste peut aussi se mettre sous la forme `insérer(l,p,e)`. Le choix de cette forme nous permet d'écrire simplement l'axiome, alors que cela aurait été très difficile si l'insertion et la suppression se faisait à deux positions différentes. On a utilisé le même principe pour les axiomes li_4 et li_5.

4.2.2 Définition récursive

Cette spécification possède cinq opérations :
– `listevide`
– `ajouterDébut` permet d'ajouter un élément en tête d'une liste. Cette opération, similaire à la fonction *cons* de LISP, doit être un constructeur.
– `enleverPremier` retire son premier élément de la liste reçue en paramètre. Elle est similaire à la fonction *cdr* de LISP.
– `premier` rend le premier élément de la liste reçue en paramètre. Elle est similaire à la fonction *car* de LISP.
– `estvide` rend vrai si la liste reçue en paramètre est vide, faux sinon

La spécification algébrique est donc :

```
spéc LISTE_REC
  étend ELEMENT, BOOL
  sorte Liste
  opérations
    listevide :                      →Liste
    ajouterDébut   : Liste, Elément →Liste
    enleverPremier : Liste          →Liste
    premier        : Liste          →Elément
    estvide        : Liste          →bool
  préconditions
    pré premier(l) = ¬estvide(l)
    pré enleverPremier(l) = ¬estvide(l)
  axiomes l:Liste;e:Element
    (lr1) estvide(listevide) = vrai
    (lr2) estvide(ajouterDebut(l,e)) = faux
    (lr3) premier(ajouterDebut(l,e)) = e
    (lr4) enleverPremier(ajouterDebut(l,e)) = l
fspéc
```

Nous n'avons pas rencontré les mêmes problèmes qu'avec la liste itérative car ici toute liste se met sous une forme unique d'appels à l'opération `ajouterDébut`.

Il est bien sûr possible d'étendre cette spécification en lui ajoutant toute opération jugée utile. En particulier, si on souhaite ajouter les opérations définies pour la liste itérative, on pourra écrire :

```
spéc LISTE_REC2
  étend LISTE_REC, TAILLE, POSITION
  opérations
    longueur  :Liste                  →Taille
    insérer   :Liste,Position,Elément →Liste
    supprimer :Liste,Position         →Liste
    ième      :Liste,Position         →Elément
  préconditions
    pré insérer(l,p,e) = p ≥ 1 ∧ p ≤ longueur(l) + 1
    pré supprimer(l,p) = p ≥ 1 ∧ p ≤ longueur(l)
    pré ième(l,p) = p ≥ 1 ∧ p ≤ longueur(l)
  axiomes l:Liste;e:Element
    (lr2₁) longueur(listevide) = 0
    (lr2₂) longueur(ajouterDébut(l,e)) = longueur(l) + 1
    (lr2₃) insérer(listevide, 1, e) = ajouterDébut(listevide, e)
    (lr2₄) insérer(ajouterDébut(l,e₁),p,e₂) =
              si p == 1 alors ajouterDébut(ajouterDébut(l,e₁),e₂)
                        sinon ajouterDébut(insérer(l, p-1,e₂), e₁)
              fsi
    (lr2₅) supprimer(ajouterDébut(l,e),p) =
              si p == 1 alors l
                        sinon ajouterDébut(supprimer(l, p-1), e)
              fsi
    (lr2₆) ième(ajouterDébut(l,e), p) =
              si p == 1 alors e sinon ième(l, p-1) fsi
fspéc
```

Il a suffit ici de définir le fonctionnement de ces quatre nouvelles opérations pour les constantes et pour les termes construits, tout en restant dans le domaine de définition de ces nouvelles opérations.

Ces axiomes permettraient de montrer par induction que tout terme **défini** de la forme `insérer(l,p,e)` est congru à un terme de la forme `ajouterDébut(l',e')`.

5. Le type pile

5.1 Définition

Définition 15 : Pile

Une pile est une liste sur laquelle on se définit un nombre restreint d'opérateurs, offrant des fonctionnalités similaires à une pile d'objets. On doit pouvoir :
- ajouter un objet au dessus de la pile. Cet opérateur sera nommé *empiler*
- examiner l'objet situé au dessus de la pile. Cet opérateur sera nommé *sommet*
- enlever l'objet situé au dessus de la pile. Cet opérateur sera nommé *dépiler*
- savoir si la pile est vide. Cet opérateur sera nommé *estvide*

On pourrait définir d'autres opérations, mais elles sont rarement utiles et de plus peu conformes avec la vision habituelle d'une pile.

5.2 Spécification abstraite algébrique

```
spéc PILE
  étend ELEMENT,BOOL
  sorte Pile
  opérations
    pilevide :              →Pile
    empiler  :Pile,Elément →Pile
    dépiler  :Pile          →Pile
    sommet   :Pile          →Elément
    estvide  :Pile          →Bool
  préconditions p:Pile
    pré dépiler(p) = non estvide(p)
    pré sommet(p) =non estvide(p)
  axiomes p:Pile;e:Element
    (pi1) dépiler(empiler(p,e)) = p
    (pi2) sommet(empiler(p,e)) = e
    (pi3) estvide(pilevide) = vrai
    (pi4) estvide(empiler(p,e)) = faux
fspéc
```

On reconnaît, au nom des opérations près, la spécification de la liste récursive. On obtiendra donc des implémentations identiques en omettant toutefois d'introduire la gestion des curseurs devenus inutiles.

5.3 Réalisation avec les classes des paquetages de java

La sous-classe *Stack* de la classe *Vector* implémente ces opérations. Citons les méthodes suivantes de cette classe.

- `boolean empty()`
  implémentation de l'opération *estvide*
- `Object peek()`
  implémentation de l'opération *sommet*
- `Object pop()`
  implémentation de l'opération *dépiler*. Cette méthode rend l'objet qui a été dépilé.
- `Object push(Object item)`
  implémentation de l'opération *empiler*

6. Le type file

6.1 Définition

Définition 16 : File

Une file est une liste sur laquelle on se définit un nombre restreint d'opérateurs, offrant des fonctionnalités similaires à celles d'une file d'attente. On doit pouvoir :
– ajouter un objet après le dernier élément de la file. Cet opérateur sera nommé *ajouter*
– examiner examiner le premier élément de la file. Cet opérateur sera nommé *premier*
– enlever le premier élément de la file. Cet opérateur sera nommé *enlever*
– savoir si la file est vide. Cet opérateur sera nommé *estvide*

6.2 Spécification abstraite algébrique

```
spéc FILE
  étend ELEMENT,BOOL
  sorte File
  opérations
    filevide :              →File
    ajouter  :File,Elément →File
    enlever  :File          →File
    premier  :File          →Elément
    estvide  :File          →Bool
  préconditions f:File
    pré enlever(f) = non estvide(f)
    pré premier(f) =non estvide(f)
  axiomes f:File;e,e1,e2:Element
    (fi1) enlever(ajouter(filevide,e)) = filevide
    (fi2) enlever(ajouter(ajouter(f,e1),e2)) =
             ajouter(enlever(ajouter(f,e1)),e2)
    (fi3) premier(ajouter(filevide,e)) = e
    (fi4) premier(ajouter(ajouter(f,e1),e2)) = premier(ajouter(f,e1))
    (fi5) estvide(filevide) = vrai
    (fi6) estvide(ajouter(f,e)) = faux
fspéc
```

Comme d'habitude, l'ensemble des axiomes doit permettre d'indiquer le résultat de chaque opération qui n'est ni une constante ni un constructeur pour des termes constants et des termes construits, et ceci en restant dans le domaine de définition de l'opération. Jusqu'à présent, il avait suffit de donner un axiome par constante, et un pour un terme construit générique.

La file pose un problème particulier, car on ne peut pas indiquer comment transformer `enlever(ajouter(f,e))` et `premier(ajouter(f,e))` sans en savoir plus sur `f`. Le problème est ici simplement résolu en divisant les termes construits en deux catégories : les files contenant un seul élément, représentées par `ajouter(filevide,e)`, et les files contenant plus d'un élément, représentées par `ajouter(ajouter(f,e1),e2)`.

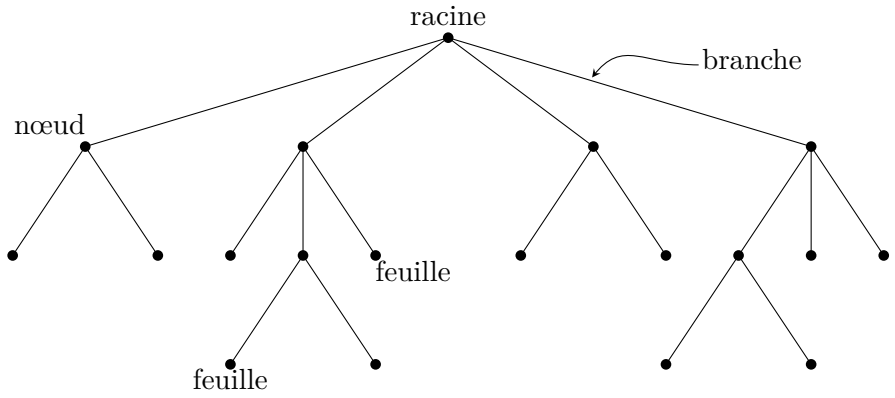
7. Arbres et Forêts

7.1 Définition du type *Arbre*

La structure d'arbre est plus générale que la structure de liste, dans la mesure où elle permet de représenter des relations hiérarchiques, alors que la structure de liste ne permettait que de représenter des relations de succession. Elle permet notamment de modéliser :

- des arbres de jeu : chaque nœud représente une configuration possible du jeu ; les fils d'un nœud représentent les configurations qu'il est possible d'atteindre par un coup
- des expressions arithmétiques : chaque nœud représente un opérateur ; les feuilles sont les opérandes. L'opérateur situé à un nœud intermédiaire porte sur l'expression représentée par ses fils gauche et droit.
- des systèmes de classification
- des arbres de décision
- la structure syntaxique d'une phrase
- etc.

La représentation graphique associée à un arbre est la suivante :



On distinguera dans cet arbre :

- des nœuds, parmi lesquels la racine, à laquelle n'arrive aucun lien, et les feuilles, dont ne partent aucun lien.
- des branches, matérialisant la relation hiérarchique. Implicitement, une branche est orientée d'un nœud père vers un nœud fils, plus éloigné de la racine. Une seule branche aboutit à chaque nœud.

On appellera descendance d'un nœud de l'arbre l'ensemble des nœuds accessibles à partir de ce nœud en suivant une suite de branches s'éloignant de la racine.

Il est par ailleurs intéressant d'introduire la notion de sous-arbre, dans la mesure où elle met bien en évidence la nature récursive des arbres. On appellera sous-arbre de racine X d'un arbre de racine R la structure formée par les nœuds appartenant à la descendance de X et les branches qui les relient. Un sous-arbre est, de toute évidence un arbre. Il apparaît alors que, pour appliquer un traitement à un arbre, il suffira de l'appliquer au nœud racine, et récursivement, à tous les sous-arbres dont la racine est un des nœuds fils de l'arbre de départ.

7.2 Arbres binaires

7.2.1 Définition

Un arbre binaire présente la particularité que chacun de ses nœuds possède au plus deux fils : on parle de fils gauche et de fils droit. Il est toutefois possible que certains nœuds possèdent un fils gauche sans posséder de fils droit, ou encore un fils droit sans posséder de fils gauche (ou même aucun fils).

On peut définir récursivement un arbre binaire :

Un arbre binaire est :
– soit vide,
– soit composé d'un élément, d'un arbre fils gauche et d'un arbre fils droit.
Chaque fils peut éventuellement être un arbre vide, permettant ainsi de modéliser l'absence de ce fils.

7.2.2 Spécification abstraite algébrique

```
spéc ARBRE
  étend ELEMENT, BOOL
  sorte Arbre
  opérations
    arbrevide  :                       → Arbre
    construire : Arbre,Elément,Arbre → Arbre
    filsGauche : Arbre                 → Arbre
    filsDroit  : Arbre                 → Arbre
    racine     : Arbre                 → Elément
    estvide    : Arbre                 → bool
  préconditions
    pré racine(a) = ¬estvide(a)
  axiomes g,d:Arbre;e:Elément
    (ar1) filsGauche(arbrevide) = arbrevide
    (ar2) filsGauche(construire(g,e,d)) = g
    (ar3) filsDroit(arbrevide) = arbrevide
    (ar4) filsDroit(construire(g,e,d)) = d
    (ar5) racine(construire(g,e,d)) = e
    (ar6) estvide(arbrevide) = vrai
    (ar7) estvide(construire(g,e,d)) = faux
fspéc
```

– *construire*(g, e, d) donne un arbre possédant e pour racine, g pour sous-arbre gauche et d pour sous-arbre droit. Si g (resp. d) est un arbre vide, l'arbre résultant n'a pas de fils gauche (resp. droit). Cette opération doit être un constructeur : il doit être possible de construire tout arbre à partir d'un arbre vide et d'une succession d'application de cette opération.
– *filsGauche*(a) donne le sous-arbre gauche de l'arbre a. Si l'arbre est vide ou n'a pas de fils gauche, l'arbre résultant est un arbre vide.
– *filsDroit*(a) donne le sous-arbre droit de l'arbre a. Si l'arbre est vide ou n'a pas de fils droit, l'arbre résultant est un arbre vide.
– *racine*(a) donne l'élément contenu dans la racine de l'arbre a. Cet opération n'est pas définie pour un arbre vide.
– *estvide*(a) rend $VRAI$ si l'arbre a est vide, $FAUX$ sinon.

7.3 Arbres planaires (n-aires) - Forêts

7.3.1 Définition

Un arbre planaire (on parle aussi d'arbre n-aire) présente la particularité que chacun de ses nœuds peut avoir un nombre quelconque de fils. On peut définir ce type de données

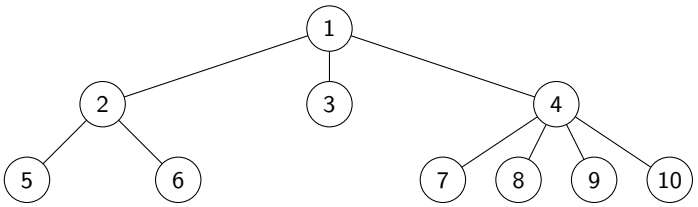
de manière récursive :

Un arbre planaire est :
– soit vide,
– soit composé d'un élément et d'une liste d'arbres planaires.
La liste d'arbres est la liste des fils du nœud racine.

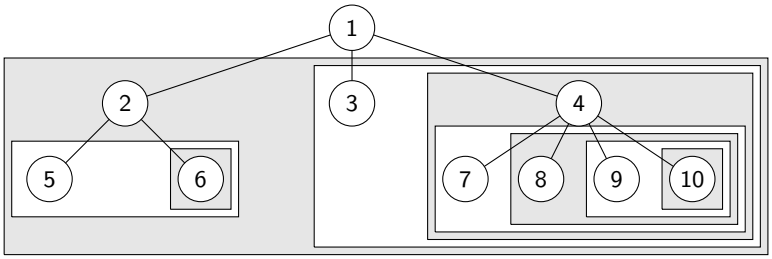
Cette définition, pour exacte qu'elle soit, n'est pas facile à modéliser et à implémenter car elle introduit deux types de données : le type *arbre* et le type *liste d'arbres*. En fait, il est tout à fait possible de se passer du type *arbre* et de n'envisager que le type *liste d'arbres* que nous nommerons par la suite *forêt*, un arbre n'étant rien d'autre qu'une forêt possédant 0 (pour l'arbre vide) ou 1 élément.

Le type forêt peut être défini récursivement par :

Une forêt est :
– soit vide,
– soit composée d'un élément, appelé racine de la forêt, d'une forêt « sœur » et d'une forêt « fille ».
A titre d'exemple, considérons l'arbre suivant, contenant comme éléments les entiers de 1 à 10 :



Il peut être vu comme la forêt suivante :



– La forêt principale est composée de l'élément 1, d'une forêt sœur vide et d'une forêt fille contenant les autres éléments et ayant 2 pour racine.
– La forêt ayant 2 pour racine possède comme forêt sœur la forêt ayant 3 pour racine et comme forêt fille celle ayant 5 pour racine
– La forêt ayant 3 pour racine possède comme forêt sœur la forêt ayant 4 pour racine et une forêt fille vide
– La forêt ayant 4 pour racine possède une forêt sœur vide et comme forêt fille celle ayant 7 pour racine
– La forêt ayant 5 pour racine possède comme forêt sœur la forêt ayant 6 pour racine et une forêt fille vide.
– La forêt ayant 6 pour racine possède une forêt sœur et une forêt fille vides
– La forêt ayant 7 pour racine possède comme forêt sœur la forêt ayant 8 pour racine et une forêt fille vide.
– La forêt ayant 8 pour racine possède comme forêt sœur la forêt ayant 9 pour racine et une forêt fille vide.

– La forêt ayant 9 pour racine possède comme forêt sœur la forêt ayant 10 pour racine et une forêt fille vide.
– La forêt ayant 10 pour racine possède une forêt sœur et une forêt fille vides

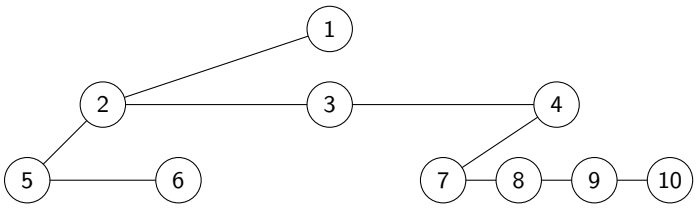
7.3.2 Spécification abstraite algébrique

```
spéc FORET
  étend ELEMENT, BOOL
  sorte Forêt
  opérations
    forêtvide  :                      → Forêt
    construire : Forêt,Elément,Forêt  → Forêt
    fille      : Forêt                → Forêt
    soeur      : Forêt                → Forêt
    racine     : Forêt                → Elément
    estvide    : Forêt                → bool
  préconditions
    pré racine(f) = ¬estvide(f)
  axiomes f,s:Forêt;e:Elément
    (f1) fille(forêtvide) = forêtvide
    (f2) fille(construire(f,e,s)) = f
    (f3) soeur(forêtvide) = forêtvide
    (f4) soeur(construire(f,e,s)) = s
    (f5) racine(construire(f,e,s)) = e
    (f6) estvide(forêtvide) = vrai
    (f7) estvide(construire(f,e,s)) = faux
fspéc
```

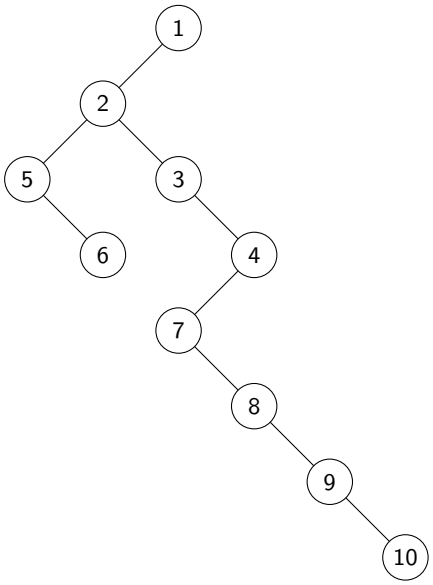
– *construire*(f, e, s) donne une forêt possédant e pour racine, f pour forêt fille et s pour forêt sœur. Si f (resp. s) est une forêt vide, la forêt résultante n'a pas de forêt fille (resp. sœur). Cette opération doit être un constructeur : il doit être possible de construire toute forêt à partir d'une forêt vide et d'une succession d'application de cette opération.
– *fille*(f) donne la forêt fille de la forêt f. Si f est vide ou n'a pas de forêt fille, la forêt résultante est une forêt vide.
– *sœur*(f) donne la forêt sœur de la forêt f. Si f est vide ou n'a pas de forêt sœur, la forêt résultante est une forêt vide
– *racine*(f) donne l'élément contenu dans la racine de la forêt f. Cet opération n'est pas définie pour une forêt vide.
– *estvide*(f) rend $VRAI$ si la forêt f est vide, $FAUX$ sinon.

On peut constater la grande similitude entre ces axiomes et ceux définissant le fonctionnement des opérations du type « arbre binaire » (paragraphe 7.2.2 p :38).

En fait, il s'agit exactement du même type de données, seuls les noms de certaines opérations ayant changé. Une forêt peut en effet très bien être représentée par un arbre binaire. La forêt de l'exemple précédent peut se représenter en matérialisant les liens fille-sœur :



Cette forêt est équivalente à l'arbre binaire :


1
2
5
3
6
4
7
8
9
10

8. Le type graphe

8.1 Introduction

Un graphe est constitué d'un ensemble de sommets, reliés entre eux par des liaisons, orientées ou non, valuées ou non.

Si les liaisons sont orientées, ces dernières sont nommées des arcs. On parle alors de graphe orienté. Dans le cas contraire, les liaisons prennent le nom d'arêtes, et le graphe est dit non orienté.

Si une valeur est associée à chaque liaison, on parle alors de graphe valué (ou pondéré).

Les graphes interviennent dans un grand nombre de problèmes, et constituent une modélisation très puissante dans la mesure où les algorithmes généraux développés sur les graphes s'appliquent, indépendamment de la signification associée aux sommets ou liaisons. Ils seront utilisés pour traiter par exemple des problèmes de transport, des jeux, des problèmes d'ordonnancement, etc.

Il faut noter que les graphes présentent une structure assez voisine de celle des arbres, mais qu'ils s'en distinguent par une caractéristique importante : un sommet (analogue à un nœud) peut être atteint à partir de plusieurs chemins, donc n'a pas de père unique. De plus, il n'est pas toujours possible de distinguer un nœud racine.

Ceci est illustré par le graphe orienté ci-dessous.

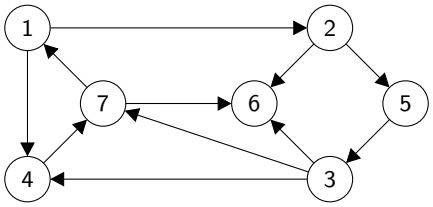


FIGURE 8.1 – Graphe orienté non valué

Si le graphe n'est pas orienté, on omet alors les flèches sur les liaisons. Si le graphe est valué, on place alors les valeurs des liaisons au niveau de chacune d'entre elles, comme dans le schéma ci dessous :

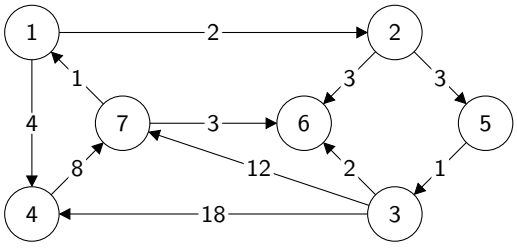


FIGURE 8.2 – Graphe orienté valué

8.2 Définitions

Nous allons définir de manière plus rigoureuse les quatre types de graphes évoqués ci-dessus.

Définition 17 : Graphe orienté

Un graphe orienté est un triplet $G = (S, A, \sigma)$ où :
- S est un ensemble dont les éléments sont appelés les sommets du graphe.
- A est un ensemble dont les éléments sont appelés les arcs du graphe.
- σ est une fonction de A dans S^2. A chaque élément a de A est associé un unique couple $\sigma(a) = (s_1, s_2)$. On dit alors que s_1 est l'extrémité initiale de a et s_2 son extrémité terminale.

Remarque :
A ne peut être assimilé à une partie de S^2, car il peut y avoir dans un graphe plusieurs arcs possédant les mêmes extrémités initiales et terminales.

Définition 18 : Graphe orienté valué

Un graphe orienté est un quadruplet $G = (S, A, \sigma, \omega)$ où :
- S est un ensemble dont les éléments sont appelés les sommets du graphe.
- A est un ensemble dont les éléments sont appelés les arcs du graphe.
- σ est une fonction de A dans S^2. A chaque élément a de A est associé un unique couple $\sigma(a) = (s_1, s_2)$. On dit alors que s_1 est l'extrémité initiale de a et s_2 son extrémité terminale.
- ω est une fonction de A dans un ensemble habituellement numérique $(\mathbb{N}, \mathbb{R}, \mathbb{Z}, \ldots)$. $\omega(a)$ est appelé le coût de l'arc a.

Définition 19 : Graphe non orienté

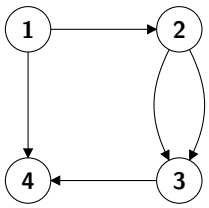
Un graphe non orienté est un triplet $G = (S, A, \sigma)$ où :
- S est un ensemble dont les éléments sont appelés les sommets du graphe.
- A est un ensemble dont les éléments sont appelés les arêtes du graphe.
- σ est une fonction de A dans $P_2(S)$, où $P_2(S)$ est l'ensemble des parties de S à 2 éléments. A chaque élément a de A est associé un unique sous-ensemble $\sigma(a) = \{s_1, s_2\}$. On dit alors que s_1 et s_2 sont les extrémités de a.

Définition 20 : Graphe non orienté valué

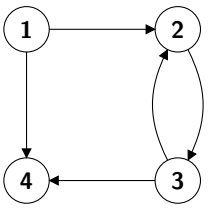
Un graphe non orienté est un quadruplet $G = (S, A, \sigma, \omega)$ où :
- S est un ensemble dont les éléments sont appelés les sommets du graphe.
- A est un ensemble dont les éléments sont appelés les arêtes du graphe.
- σ est une fonction de A dans $P_2(S)$, où $P_2(S)$ est l'ensemble des parties de S à 2 éléments. A chaque élément a de A est associé un unique sous-ensemble $\sigma(a) = \{s_1, s_2\}$. On dit alors que s_1 et s_2 sont les extrémités de a.
- ω est une fonction de A dans un ensemble habituellement numérique $(\mathbb{N}, \mathbb{R}, \mathbb{Z}, \ldots)$. $\omega(a)$ est appelé le coût de l'arête a.

8.3 Terminologie

- *p-graphe*
  Un p-graphe est un graphe orienté $G = (S, A, \sigma)$ tel que $\forall (s_1, s_2) \in S^2$, il existe au plus p arcs ayant s_1 comme extrémité initiale et s_2 comme extrémité terminale.

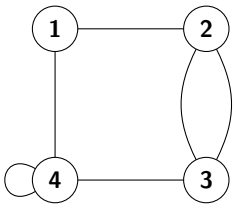


Un 2-graphe

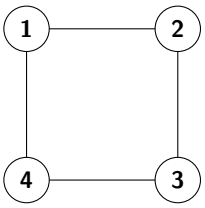


Un 1-graphe

– *multigraphe*
  Un multigraphe est un graphe non orienté tel qu'il existe aux moins deux arêtes ayant mêmes extrémités.
– *boucle*
  Une boucle est un arc ou une arête ayant le même sommet à ses deux extrémités.
– *graphe simple*
  Un graphe simple est un graphe non orienté sans boucles et tel qu'il n'existe pas plusieurs arêtes reliant les deux mêmes sommets.



Un multigraphe



Un graphe simple

– *successeur*
  Un sommet s_1 d'un graphe orienté est dit successeur d'un sommet s_2 du même graphe si il existe dans ce graphe un arc ayant s_1 comme extrémité terminale et s_2 comme extrémité initiale.
– *prédécesseur*
  Un sommet s_1 d'un graphe orienté est dit prédécesseur d'un sommet s_2 du même graphe si il existe dans ce graphe un arc ayant s_1 comme extrémité initiale et s_2 comme extrémité terminale.
– *arcs et arêtes adjacents*
  Deux arcs (ou deux arêtes) sont dits adjacents s'ils possèdent au moins une extrémité commune.
– *demi-degré extérieur*
  Le demi-degré extérieur d'un sommet d'un graphe orienté est le nombre d'arcs ayant ce sommet comme extrémité initiale.
– *demi-degré intérieur*
  Le demi-degré intérieur d'un sommet d'un graphe orienté est le nombre d'arcs ayant ce sommet comme extrémité terminale.
– *degré*
  Le degré d'un sommet d'un graphe orienté (resp. non orienté) est le nombre d'arcs (resp. d'arêtes) ayant ce sommet pour extrémité.
– *clique d'un graphe orienté*
  Une clique d'un graphe orienté $G = (S, A, \sigma)$ est un sous-ensemble $C \subseteq S$ tel que pour tout couple (s_1, s_2) d'éléments différents de C il existe au moins un arc ayant s_1 comme extrémité initiale et s_2 comme extrémité terminale, ou un arc ayant s_2 comme extrémité initiale et s_1 comme extrémité terminale.
– *clique d'un graphe non orienté*
  Une clique d'un graphe non orienté $G = (S, A, \sigma)$ est un sous-ensemble $C \subseteq S$ tel que pour tout couple (s_1, s_2) d'éléments différents de C il existe au moins une arête ayant s_1 et s_2 comme extrémités.

- *chaîne*
  Une chaîne de longueur n d'un graphe $G = (S, A, \sigma)$ est un n-uplet $(a_1, a_2, \ldots, a_n) \in A^n$ tel que pour tout $i \in [1, n[$, a_i est adjacent à a_{i+1}. Cette définition vaut aussi bien pour les graphes orientés que les graphes non orientés.
- *cycle*
  Un cycle est une chaîne fermée.
- *chemin*
  Un chemin d'un graphe orienté est une chaîne de ce graphe dont tous les arcs sont orientés dans le même sens.
- *circuit*
  Un circuit d'un graphe orienté est un cycle de ce graphe dont tous les arcs sont orientés dans le même sens.
- *graphe connexe*
  Un graphe connexe est un graphe (orienté ou non) tel que pour toute paire de sommets différents de ce graphe, il existe un chaîne ayant ces deux sommets pour extrémités.
- *sous-graphe*
  Un sous-graphe d'un graphe $G = (S, A, \sigma)$ est un graphe $G' = (S', A', \sigma)$ tel que $S' \subseteq S$ et $A' \subseteq A$.
- *composante connexe d'un graphe*
  Une composante connexe d'un graphe G est un sous-graphe de G connexe.

8.4 Recherche du plus court chemin : algorithme de Dijkstra

Il existe de nombreux algorithmes permettant de rechercher un chemin de longueur minimale entre deux sommets d'un graphe. Ils permettent de traiter de nombreux problèmes pratiques, comme par exemple la recherche d'un itinéraire optimal dans un réseau routier ou ferré. Ce problème n'a bien sûr de solution que s'il existe au moins un chemin reliant les deux sommets concernés.

Nous allons présenter ici l'algorithme de *Dijkstra*, qui permet de trouver un chemin de longueur minimale dans un graphe orienté valué par valeurs *positives* entre un sommet et tous les autres.

Soit donc un graphe $G = (S, A, \sigma, \omega)$. On recherche un chemin de longueur minimale entre les sommets *départ* et *arrivée*. La méthode consiste à construire progressivement un sous-ensemble E de sommets de S tel que pour tout s appartenant à E, on connaît dans G un chemin de distance minimale reliant *départ* à s.

Au début, E est vide, et le chemin de longueur minimale reliant *départ* à *départ* est bien sûr vide. L'algorithme s'arrête lorsque l'on ajoute *arrivée* à E, ou lorsque l'on ne peut plus ajouter aucun sommet à E (auquel cas, cela signifie qu'il n'y a pas de chemin entre *départ* et *arrivée*).

Il reste à déterminer quel sommet ajouter à E à chaque étape.

Considérons l'ensemble F des sommets de G n'appartenant pas à E, mais pouvant être atteints directement à partir d'un sommet de E par un seul arc.

Pour chaque sommet s de F, considérons tous les chemins reliant *départ* à s composés d'une première partie de longueur minimale reliant *départ* à un sommet de E et d'un arc reliant ce sommet à s (par définition de E et par construction de F, on est sûr qu'il existe au moins un tel chemin). Parmi tous ces chemins, retenons en un de longueur minimale. On associe donc ainsi à chaque sommet de F un chemin $cheminMin(s)$ reliant *départ* à s.

Choisissons un sommet *min* de F tel que la longueur de $cheminMin(min)$ soit minimale.

$cheminMin(min)$ est alors un chemin de longueur minimale reliant *départ* à *min*. En effet, tout chemin reliant *départ* à *min* est composé d'une première partie reliant *départ* à un sommet e de E, d'un arc reliant e à un sommet f de F et d'une dernière partie reliant f à *min*. La partie de ce chemin reliant *départ* à f est de longueur supérieure ou égale à la longueur de $cheminMin(min)$ (de par le choix de f). Tous les arcs de G étant à valeur positive, le chemin complet est donc de longueur supérieure ou égale à la longueur de $cheminMin(min)$.

8.5 Un algorithme approché de coloriage d'un graphe

On considère un graphe $G = (S, A, \sigma)$ non orienté. Colorier ce graphe consiste à attribuer une « couleur » à chaque sommet du graphe en faisant en sorte que deux sommets reliés par une arête ne soient jamais coloriés avec la même couleur. Parmi les problèmes pratiques pouvant être traités par un coloriage de graphe, on peut citer certains problèmes d'emploi du temps.

Le problème consistant à colorier un graphe avec un nombre minimal de couleurs fait partie de la classe des problèmes dits « NP-complets ». Sans entrer en détail dans la théorie de la complexité des problèmes, cela signifie que personne n'a jamais trouvé d'algorithme de complexité meilleure qu'exponentielle pour résoudre ce genre de problème.

Pour tous ces problèmes d'optimisation NP-complets, on peut, dans de nombreux cas, se contenter de trouver une « bonne » solution et non pas la meilleure. Il est alors souvent possible de trouver un algorithme de complexité bien meilleure permettant de résoudre ce problème ainsi modifié. On parlera alors d'algorithme approché résolvant un problème d'optimisation.

Nous allons étudier ici un algorithme « glouton » permettant de trouver une solution approchée au problème du coloriage d'un graphe en un nombre minimal de couleurs. Cet algorithme permet de colorier un graphe non orienté avec un nombre de couleurs proche de l'optimum.

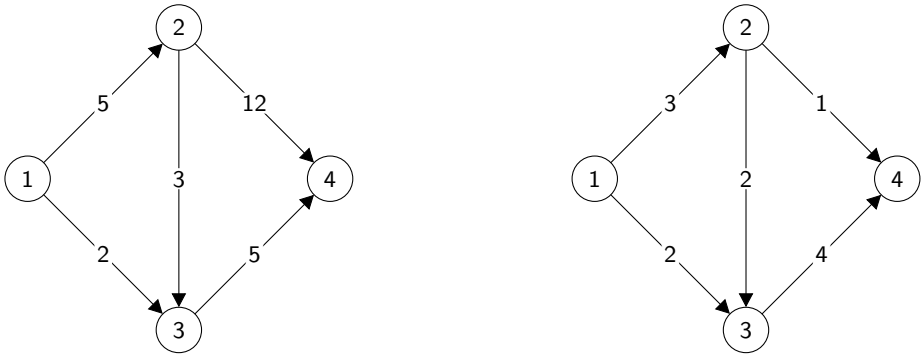
Un algorithme glouton est basé sur des principes assez proches de ceux utilisés pour les algorithmes à essais successifs. De même que pour les algorithmes à essais successifs, un algorithme glouton construit une solution à un problème étape par étape. A chaque étape, on a le choix entre plusieurs actions pour progresser. Alors qu'un algorithme à essais successifs décide d'essayer toutes les actions les une après les autres, un algorithme glouton décide de n'en essayer qu'une seule : celle qui lui semble la meilleure au moment où le choix doit être effectué. Ainsi, au lieu de parcourir la totalité de l'arbre de choix comme le fait une algorithme à essais successifs, un algorithme glouton n'en explore qu'une seule branche. La complexité de l'algorithme est alors en *log* de celle de l'algorithme à essais successifs correspondant.

L'algorithme glouton permettant de colorier un graphe est le suivant :

Nous allons colorier le graphe sommet par sommet, en ne changeant de couleur que lorsque la couleur courante ne peut plus être attribuée à aucun sommet. A chaque étape, nous avons donc la possibilité d'attribuer la couleur courante à un certain nombre de sommets. Le choix glouton qui va être effectué consiste à attribuer la couleur au sommet qui semble, dans l'état actuel des choses, le plus difficile à colorier (parmi les sommets qui peuvent être coloriés avec cette couleur). Une fois effectué, ce choix ne sera plus remis en cause. On décide de choisir le sommet qui possède le plus de voisins non coloriés.

8.6 Recherche d'un flot maximum dans un graphe orienté

Un graphe orienté valué à valeurs positives peut servir à modéliser la circulation de fluides dans un circuit. Ce terme de fluide doit être pris dans son sens le plus général. Il peut représenter de l'électricité, un liquide, mais aussi un flot de véhicules, etc. D'une manière générale, un arc représente un canal de communication, et la valeur d'un arc la capacité maximale de ce canal pour le fluide concerné.



Graphe pour calculs de flots Flot entre les sommets 1 et 4

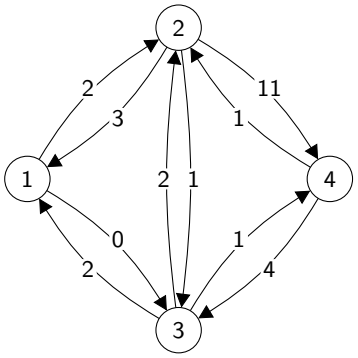
FIGURE 8.3 – Graphe pour calculs de flots et exemple de flot

Établir un flot dans un tel graphe entre un sommet d'entrée et un sommet de sortie consiste à attribuer une nouvelle valeur à chaque arc, positive mais inférieure ou égale à la valeur d'origine de l'arc, en faisant en sorte que pour tout sommet s autre que le sommet d'entrée et de sortie, la somme des valeurs des arcs ayant s comme extrémité terminale soit égale à la sommes des valeurs des arcs ayant s comme extrémité initiale.

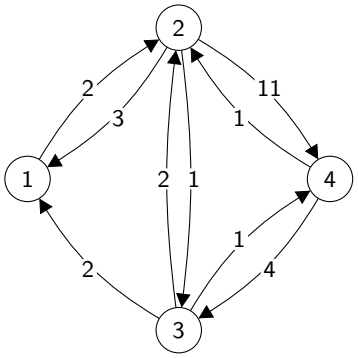
De manière plus pratique, il s'agit d'établir une circulation du fluide dans le réseau en faisant en sorte que tout ce qui entre à chaque nœud en ressorte. Ce qui entre par le nœud d'entrée doit ressortir par le nœud de sortie. La valeur du flot est égale à ce qui entre par le noeud d'entrée, c'est-à-dire à la somme des valeurs des arcs ayant le sommet d'entrée comme extrémité terminale, diminuée de la somme des valeurs des arcs ayant le sommet d'entrée comme extrémité initiale.

Une manière agréable de représenter à la fois le flot et les capacités de chaque arc consiste à construire ce que l'on appelle un graphe d'écart. Il est construit à partir du graphe d'origine. A tout arc de ce graphe reliant un sommet s_1 à un sommet s_2, on associe deux arcs dans le graphe d'écart. Le premier possède s_1 comme extrémité initiale, et s_2 comme extrémité terminale. On lui donne pour valeur la capacité restante du canal de communication que l'arc représente. Le second possède s_2 comme extrémité initiale, et s_1 comme extrémité terminale. On lui donne pour valeur la capacité utilisée du canal de communication que l'arc représente.

Les deux schémas de la figure précédente peuvent donc se résumer par le graphe d'écart suivant :



Graphe d'écart avec tous les arcs



Graphe d'écart sans les arcs de valeur nulle

FIGURE 8.4 – Graphe d'écart

Pour pouvoir mettre en œuvre l'algorithme de Ford-Fulkerson que nous allons étudier par la suite, on décide de supprimer du graphe d'écart tous les arcs de valeur nulle. Cela donne alors :

Un des problèmes que l'on peut avoir à traiter consiste à trouver dans un graphe un flot de valeur maximale. L'algorithme de Ford-Fulkerson permet de résoudre ce problème. Il est basé sur l'utilisation de graphes d'écarts.

On commence par construire le graphe d'écarts associé à un flot de valeur nulle, ayant un débit 0 pour tout arc. Cela donne un graphe identique au graphe de gauche de la figure 8.3.

Nous allons ensuite transformer ce flot en prenant soin d'augmenter sa valeur à chaque étape. S'il existe un chemin dans le graphe d'écart entre le sommet d'entrée et le sommet de sortie, cela signifie que la valeur du flot peut être augmentée de la valeur du plus petit des arcs de ce chemin (plus petit étant entendu au sens du coût des arcs). Soit α cette valeur. Pour tout sommet s de ce chemin différent du sommet d'entrée et de sortie, considérons l'arc a_1 arrivant sur ce sommet et l'arc a_2 partant de ce sommet. Soient c_1 le coût de a_1 et c_2 le coût de a_2. On peut alors envisager 4 possibilités :

- a_1 et a_2 sont des arcs du graphe d'origine
  Cela signifie que le canal de communication associé à $a1$ peut encore recevoir un débit de c_1 (en direction de s), et que celui associé à a_2 un débit de c_2 (en provenance de s). Comme α est plus petit que ces deux valeurs, cela signifie que l'on peut faire arriver sur s un débit supplémentaire de α que l'on peut faire sortir par a_2. La propriété du flot sur s reste donc respectée si l'on fait cette transformation (tout ce qui entre ressort).
- a_1 est un arc d'origine mais pas a_2
  Cela signifie que le canal de communication associé à $a1$ peut encore recevoir un débit de c_1 (en direction de s), et que celui associé à a_2 reçoit un débit de c_2 (en direction de s). On peut donc conserver l'équilibre au sommet s en augmentant le débit de a_1 de α et en diminuant celui de a_2 de α. En d'autres termes, a_1 et a_2 étant des canaux qui font arriver du fluide sur s, la quantité de fluide arrivant sur s est identique si on augmente celle qui arrive par a_1 de la même valeur que l'on diminue celle qui arrive par a_2.
- a_2 est un arc d'origine mais pas a_1
  Par un raisonnement similaire, cela indique que a_1 et a_2 sont tous deux des canaux faisant sortir du fluide de s. On peut donc en faire sortir plus par a_2 si on en fait sortir moins par a_1.
- a_1 et a_2 ne sont pas des arcs du graphe d'origine
  Cela signifie ici que a_1 est un arc faisant sortir du fluide de s et a_2 un arc faisant entrer du fluide dans s. On peut donc augmenter de α la quantité de fluide entrant par a_2, et de la même valeur celle sortant par a_1.

Dans tous les cas, pour actualiser le flot, il suffit de retirer α à tout arc du chemin, et d'augmenter de α l'arc inverse associé. Si l'arc est un arc du graphe d'origine, on augmente le débit sur le canal et on diminue la capacité restante. Si ce n'est pas un arc du graphe d'origine, on diminue le débit et on augmente la capacité restante.

Remarquons qu'il était important de supprimer du graphe d'écart les arcs de coût nul, car cela garantit que α est toujours strictement positif, et donc que l'algorithme converge.

Annexes

Les chapitres suivants ont été placés en annexe car ils ne sont pas traités en cours :
- la preuve de programme est traitée dans le cours de génie logiciel ;
- seuls certains algorithmes de classement sont abordés en cours et en travaux dirigés, notamment pour l'étude de la complexité des algorithmes ;

A. Preuve de programmes

Nous allons présenter dans ce chapitre une méthode permettant de prouver de manière formelle qu'un programme (ou une partie de programme) fait correctement son travail : la logique de *Hoare*.

A.1 Définition du système

A.1.1 Syntaxe du langage de programmation

Nous allons utiliser une langage de programmation très simplifié, dérivé de langage comme C, C++ ou encore java, limité aux cinq éléments suivants :

- affectation

  Si x est une variable et *expr* une expression de même type que x,

  ```
  1  x = expr;
  ```

  est une instruction.
- groupement

  Si $I1, \ldots, In$ sont des instructions,

  ```
  1  {I1 ... In}
  ```

  est une instruction.
- alternative

  Si $I1$ et $I2$ sont des instructions, et C est une condition,

  ```
  1  if(C) I1 else I2
  ```

  est une instruction.
- conditionnelle

  Si I est une instruction,et C est une condition,

  ```
  1  if(C) I
  ```

  est une instruction.
- répétition

  Si I est une instruction,et C est une condition,

  ```
  1  while(C) I
  ```

  est une instruction.

On appellera par la suite toute suite d'instructions un programme.

A.1.2 Le système formel

Une formule du système de preuve est de la forme :

```
1  /*φ*/ P /*ψ*/
```

où P est une instruction ou une suite d'instructions, φ et ψ des formules logiques. Les formules logiques peuvent faire intervenir des variables de P.

Une telle formule indique que, si φ est vérifiée avant l'exécution de P, alors ψ est vérifiée après l'exécution de P.

φ est appelée *précondition* et ψ *postcondition*.

Exemple

```
1  /*y == 10*/ x = y; /*x == 10*/
```

La démonstration de formules se fait en utilisant des règles bien précises, et uniquement celles-ci.

A.1.2.1 Axiome d'affectation

Soit φ une formule logique utilisant la variable x. On note par $\varphi[expr/x]$ la formule obtenue en remplaçant dans φ la variable x par l'expression *expr*.

```
1  /*φ[expr/x]*/ x = expr; /*φ*/
```

est une formule vraie.

Cet axiome indique que si la condition φ est vérifiée après exécution de l'instruction d'affectation, alors, la même condition dans laquelle on a remplacé x par *expr* était vérifiée avant exécution de cette affectation.

Cette façon de faire, consistant à exprimer la pré-condition en fonction de la postcondition peut sembler un peu étrange. On aurait plutôt envie d'exprimer la postcondition en fonction de la pré-condition.

En fait cela simplifie beaucoup les choses pour deux raisons :
- La postcondition est ce que l'on cherche à démontrer. Il est donc logique de partir de ce qui doit être vérifié après exécution du programme pour arriver à ce qui doit l'être avant.
- Il n'est pas toujours facile d'exprimer la postcondition à partir de la pré-condition. Par exemple, avec la pré-condition est $x^2+8x+3 > 5$ et l'instruction `x = x**2 + 5*x + 12`, on ne sait guère que choisir comme postcondition. Par contre, si la postcondition est $x < 3$, l'axiome nous permet d'écrire de manière immédiate la pré-condition $x^2+5x+12 < 3$. De plus, la pré-condition ainsi obtenue est la plus générale pour que la postcondition soit vraie.

A.1.2.2 Première règle de conséquence

Soient φ, ψ, χ des formules logiques et P un programme.

Si $\chi \Rightarrow \varphi$ et $\boxed{/*\varphi*/\ \texttt{P}\ /*\psi*/}$ alors $\boxed{/*\chi*/\ \texttt{P}\ /*\psi*/}$

Cette règle indique simplement que l'on peut remplacer une pré-condition par une condition « plus forte » qu'elle.

Par exemple, on a $\boxed{/*10 == 10*/\ \texttt{x = 10;}\ /*x == 10*/}$ et $vrai \Rightarrow 10 == 10$,

donc $\boxed{/*vrai*/\ \texttt{x = 10;}\ /*x == 10*/}$

A.1.2.3 Deuxième règle de conséquence

Soient φ, ψ, χ des formules logiques et P un programme.

Si $\psi \Rightarrow \chi$ et $\boxed{/*\varphi*/\ \texttt{P}\ /*\psi*/}$ alors $\boxed{/*\varphi*/\ \texttt{P}\ /*\chi*/}$

Cette règle indique simplement que l'on peut remplacer une postcondition par une condition « plus faible » qu'elle.

Par exemple, on a $\boxed{/*10 == 10*/\ \texttt{x = 10;}\ /*x == 10*/}$ et $x == 10 \Rightarrow x >= 10$,

donc $\boxed{/*10 == 10*/\ \texttt{x = 10;}\ /*x >= 10*/}$

A.1.2.4 Règle de séquencement

Soient φ, ψ, χ des formules logiques et $P1$ et $P2$ des programmes.

Si $\boxed{/*\varphi*/\ \texttt{P1}\ /*\psi*/}$ et $\boxed{/*\psi*/\ \texttt{P2}\ /*\chi*/}$ alors $\boxed{/*\varphi*/\ \texttt{P1 P2}\ /*\chi*/}$

A.1.2.5 Règle d'alternative

Soient φ et ψ des formules logiques, $I1$ et $I2$ des instructions, C une condition.

Si $\boxed{/*\varphi \wedge C*/\ \texttt{I1}\ /*\psi*/}$ et $\boxed{/*\varphi \wedge \neg C*/\ \texttt{I2}\ /*\psi*/}$

alors $\boxed{/*\varphi*/\ \texttt{if(C) I1 else I2}\ /*\psi*/}$

Exemple

Montrons que :

$\boxed{/*vrai*/\ \texttt{if(x \% 2 == 0) y = x; else y = x-1;}\ /*y\%2 == 0*/}$

Il faut montrer :

$\boxed{/*vrai \wedge x\%2 == 0*/\ \texttt{y=x;}\ /*y\%2 == 0*/}$ et

$\boxed{/*vrai \wedge \neg(x\%2 == 0)*/\ \texttt{y=x-1;}\ /*y\%2 == 0*/}$

L'axiome d'affectation donne $\boxed{/*x\%2 == 0*/\ \texttt{y=x;}\ /*y\%2 == 0*/}$.

Or $vrai \wedge x\%2 == 0 \Rightarrow x\%2 == 0$. La première règle de conséquence nous donne donc la première propriété.

L'axiome d'affectation donne $\boxed{/*(x-1)\%2 == 0*/\ \texttt{y=x-1;}\ /*y\%2 == 0*/}$.

Or $vrai \wedge \neg(x\%2 == 0) \Rightarrow vrai \wedge (x-1)\%2 == 0 \Rightarrow (x-1)\%2 == 0$. La première règle de conséquence donne donc la deuxième propriété.

A.1.2.6 Règle de conditionnelle

Soient φ et ψ des formules logiques, I une instruction, C une condition.

Si $\boxed{/*\varphi \wedge C*/\ \texttt{I}\ /*\psi*/}$ et $\varphi \wedge \neg C \Rightarrow \psi$ alors $\boxed{/*\varphi*/\ \texttt{if(C) I}\ /*\psi*/}$

Exemple

Montrons que :

$\boxed{/*vrai*/\ \texttt{if(x < 0) x = -x;}\ /*x \geq 0*/}$

Il faut montrer :

$\boxed{/*vrai \wedge x < 0*/\ \texttt{x = -x;}\ /*x \geq 0*/}$ et

$vrai \wedge x \geq 0 \Rightarrow x \geq 0$

L'axiome d'affectation donne $\boxed{/*-x \geq 0*/\ \texttt{x = -x;}\ /*x \geq 0*/}$.

Or $vrai \wedge x < 0 \Rightarrow vrai \wedge -x \geq 0$. La première règle de conséquence nous donne donc la première propriété.

La deuxième propriété est directement vérifiée par les règles classiques de la logique.

A.1.2.7 Règle de répétition

Soient φ une formule logique, I une instruction, C une condition.

Si $\boxed{/*\varphi \wedge C*/\ \texttt{I}\ /*\varphi*/}$ alors $\boxed{/*\varphi*/\ \texttt{while(C) I}\ /*\varphi \wedge \neg C*/}$

La propriété φ est appelée un invariant de boucle. Tout l'art consiste donc à trouver un invariant permettant de prouver que la boucle fait correctement son travail.

Exemple

Considérons le programme P suivant calculant la somme des n premiers nombres entiers.

```
1  i = 0;
2  s = 0;
3  while( i = n ) i = i + 1;s = s + i;
```

Un invariant de boucle permettant de montrer la propriété qui nous intéresse peut être : $s == \sum_{k=0}^{i} k$.

En effet :

$\boxed{/*s == \sum_{k=0}^{i} k \wedge i \neq n*/\ \texttt{i = i + 1;s = s + i;}\ /*s == \sum_{k=0}^{i} k*/}(P1)$

car :

$\boxed{/*s + i == \sum_{k=0}^{i} k*/\ \texttt{s = s + i;}\ /*s == \sum_{k=0}^{i} k*/}$ et

$\boxed{/*s + i + 1 == \sum_{k=0}^{i+1} k*/\ \texttt{i = i + 1}\ /*s + i == \sum_{k=0}^{i} k*/}$

or

$s == \sum_{k=0}^{i} k \wedge i \neq n \Rightarrow s == \sum_{k=0}^{i} k$,

$s == \sum_{k=0}^{i} k \Rightarrow s + i + 1 == \sum_{k=0}^{i} k + i + 1$,

$s + i + 1 == \sum_{k=0}^{i} k + i + 1 \Rightarrow s + i + 1 == \sum_{k=0}^{i+1} k$

donc, par la première règle de conséquence :

$\boxed{/*s == \sum_{k=0}^{i} k \wedge i \neq n*/\ \texttt{i = i + 1;}\ /*s + i == \sum_{k=0}^{i} k*/}$

La règle de séquencement nous donne donc $(P1)$.

La règle de répétition donne donc :

```
1  //s == ∑_{k=0}^{i} k
2  while( i = n)) i = i + 1;s = s + i;//s == ∑_{k=0}^{i} k ∧ i == n
```

Or :

$\boxed{/*0 == \sum_{k=0}^{i} k*/\ \texttt{s = 0;}\ /*s == \sum_{k=0}^{i} k*/}$,

$\boxed{/*0 == \sum_{k=0}^{0} k*/\ \texttt{i = 0;}\ /*0 == \sum_{k=0}^{i} k*/}$,

$vrai \Rightarrow 0 == 0$ et

$0 == 0 \Rightarrow 0 == \sum_{k=0}^{0} k$

Donc :

$\boxed{/*vrai*/\ \texttt{P}\ /*s == \sum_{k=0}^{i} k \wedge i == n)*/}$

Comme $s == \sum_{k=0}^{i} k \wedge i == n) \Rightarrow s == \sum_{k=0}^{n} k$

$\boxed{/*vrai*/\ \texttt{P}\ /*s == \sum_{k=0}^{n} k*/}$

A.2 Annotation d'un programme

Afin d'améliorer la clarté d'une preuve de programme, on la présentera en encadrant chaque instruction de sa pré-condition et de sa postcondition, la postcondition d'une instruction étant souvent la pré-condition de la suivante. Ces conditions seront mises en commentaires. Si deux conditions se suivent sans être séparées par une instruction, cela signifiera simplement que la première implique la seconde.

Exemple 1

Reprenons le programme P suivant calculant la somme des n premiers nombres entiers. On écrit alors

```
1  //vrai
2  //0 == 0
3  //∑_{k=0}^{0} k == 0
4  i = 0;
5  //∑_{k=0}^{i} k == 0
6  s = 0;
7  //∑_{k=0}^{i} k == s
8  while (i = n) //∑_{k=0}^{i} k == s ∧ (i ≠ n)//∑_{k=0}^{i} k == s//∑_{k=0}^{i+1} k == s + i + 1i = i +
   1;//∑_{k=0}^{i} k == s + is = s + i;//∑_{k=0}^{i} k == s//∑_{k=0}^{i} k == s ∧ i == n//∑_{k=0}^{n} k == s
```

Exemple 2

Modifions le programme précédent en remplaçant

```
while (i != n)
```

par

```
while (i < n).
```

Si on ne prend pas de précaution, la postcondition de cette boucle devient alors :
$\sum_{k=0}^{i} k == s \wedge i >= n$,
ce qui empêche alors de conclure puisque rien ne permet d'affirmer dans ce cas que $i == n$.

Il faut en fait choisir un invariant de boucle plus précis permettant de conclure. Si on choisit $//\sum_{k=0}^{i} k == s \wedge i \leq n$, la postcondition de la boucle sera alors :
$\sum_{k=0}^{i} k == s \wedge i \leq n \wedge i \geq n)$,
ce qui permettra alors de dire que $i == n$.

Le programme annoté devient alors :

```
1  //0 ≤ n
2  //0 == 0 ∧ 0 ≤ n
3  //∑_{k=0}^{0} k == 0 ∧ 0 ≤ n
4  i = 0;
5  //∑_{k=0}^{i} k == 0 ∧ i ≤ n
6  s = 0;
7  //∑_{k=0}^{i} k == s ∧ i ≤ n)
8  while (i < n) {
9    //∑_{k=1}^{i-1} k == s ∧ i ≤ n) ∧ i < n
10   //∑_{k=0}^{i} k == s ∧ i < n
11   //∑_{k=0}^{i+1} k == s + i + 1 ∧ (i + 1) ≤ n
12   i = i + 1;
13   //∑_{k=0}^{i} k == s + i ∧ i ≤ n
14   s = s + i;
15   //∑_{k=0}^{i} k == s ∧ i ≤ n
16 }
17 //∑_{k=0}^{i} k == s ∧ i ≤ n ∧ i ≥ n)
18 //∑_{k=0}^{i} k == s ∧ i == n
19 //∑_{k=0}^{n} k == s
```

A.3 Problème posé par les tableaux

L'affectation d'éléments d'un tableau pose des problèmes particuliers. En effet, si on écrit une instruction de la forme :

```
tab[i] = expr;
```

il sera alors difficile d'écrire la pré-condition induite par une postcondition faisant intervenir des éléments de ce tableau autres que `tab[i]`. En effet, si la postcondition est $tab[j] = 23$, on pourrait être tenté de considérer que $tab[i]$ et $tab[j]$ sont des variables comme les autres et d'appliquer l'axiome d'affectation. Cela donnerait alors :

$\boxed{/*tab[j] = 23*/ \texttt{ tab[i] = expr; } /*tab[j] = 23*/}$

ce qui n'est bien sûr vrai que si i est différent à j.

On peut résoudre ce problème en ajoutant un axiome supplémentaire, permettant d'exprimer des conditions portant non plus sur un seul élément d'un tableau, mais sur la totalité de celui-ci.

Notons $tab_{expr/i}$ le tableau obtenu à partir de tab en remplaçant l'élément de rang i par $expr$. Ce nouvel axiome est :

$\boxed{/*\varphi[tab_{expr/i}/tab]*/ \texttt{ tab[i]=expr; } /*\varphi*/}$

Cet axiome indique que l'on obtient la pré-condition en remplaçant dans la postcondition le tableau tab par le tableau $tab_{expr/i}$. Il sera toutefois assez délicat à manipuler lors de la démonstration de preuves complexes.

Signalons que des problèmes similaires à celui rencontré sur les tableaux existent pour tout objet lors de l'utilisation d'une méthode modifiant son état. Ce sera en particulier le cas de tous les objets de type « collection ». Les solutions à ces problèmes ne seront pas envisagées dans le cadre de ce cours.

A.4 Terminaison d'un programme

La méthode de preuve de programme vue dans les paragraphes précédents permet de montrer que si la pré-condition est vérifiée, alors, après la fin de l'exécution du programme, la postcondition l'est aussi. Il reste encore à montrer que le programme a une fin ! On dit encore qu'il faut montrer que le programme *termine*.

Malheureusement, il n'existe pas de méthode générale permettant de prouver qu'un programme termine (on peut même démontrer qu'il ne peut en exister). Le problème de « non terminaison » se pose uniquement dans les boucles et dans les fonctions récursives.

Nous allons étudier ici uniquement le cas des boucles.

Si il existe une expression entière utilisant les variables du programme, strictement décroissante entre le début et la fin de toute itération et toujours positive ou nulle, alors la boucle termine.

Plus formellement :

Soit **`while`**`(C) P` une boucle, φ une formule logique, F une fonction des variables du programme. Si :

$\varphi \wedge C \Rightarrow F \geq 0$ et $\boxed{/*\varphi \wedge C \wedge F = F_0*/ \texttt{ P } /*\varphi \wedge F < F0*/}$

alors la boucle termine et $\boxed{/*\varphi*/ \textbf{\texttt{while}}\texttt{(C) P } /*\varphi \wedge \neg C*/}$.

Reprenons le programme précédent. La fonction F peut être $F = n - i$. Il s'agit bien d'une expression strictement décroissante à chaque itération et toujours positive ou nulle.

On a bien $(\sum_{k=0}^{i} k == s) \wedge (i \leq n) \wedge (i < n) \Rightarrow n - i \geq 0$

De plus

1 $//(\sum_{k=0}^{i} k = s) \wedge (i \leq n) \wedge (i < n) \wedge ((n - i) = F_0)$
2 $//\sum_{k=0}^{i} k == s \wedge (i + 1) \leq n) \wedge (n - i - 1) < F_0$
3 $//\sum_{k=0}^{i+1} k == s + i + 1 \wedge (i + 1) \leq n) \wedge (n - i - 1) < F_0$

```
4    i = i + 1;
5    //∑_{k=0}^{i} k == s + i ∧ i ≤ n) ∧ (n − i) < F_0
6    s = s + i;
7    //∑_{k=0}^{i} k == s ∧ i ≤ n) ∧ (n − i) < F_0
```

Donc la boucle *while* termine.

B. Algorithmes de classement

Les algorithmes de classement, aussi appelés *tris* bien que cette appellation soit impropre, permettent de classer les éléments dans un certain ordre.

Ces algorithmes sont particulièrement importants car ils sont utilisés pour améliorer l'efficacité de la recherche d'un élément dans une liste et la comparaison de deux listes. En effet, lorsqu'une liste est classée, on peut arrêter une recherche séquentielle dès qu'on rencontre un élément plus grand que celui qu'on recherche. On peut aussi utiliser une méthode de recherche dichotomique.

Le classement des éléments d'une liste s'appuie sur une propriété de ces éléments. Par exemple, une liste de personnes pourra être classée par ordre alphabétique croissant des noms des personnes. La propriété des éléments utilisée pour le classement est appelée « clef » du classement. Deux éléments ayant la même clef ne sont pas forcément identiques, ce qui fait que la relation utilisée pour le classement n'est pas anti-symétrique : $(a \leq b) \wedge (b \leq a) \not\Rightarrow (a == b)$. Cette relation n'est donc pas une relation d'ordre, mais un *pré-ordre*. Deux éléments distincts qui ne peuvent pas être distingués dans la relation de pré-ordre sont dits *équivalents*.

Un tri (ou algorithme de classement) est dit *stable* s'il conserve l'ordre dans la liste des éléments équivalents. Dans le cas contraire, le tri est dit *instable*.

Soit une liste $l = (e_i)_{1 \leq i \leq n}$, et $(c_i)_{1 \leq i \leq n}$ les clefs associées aux éléments de l. Soit $\leq$ une relation d'ordre total sur les clefs, et $\preceq$ le pré-ordre total induit par $\leq$ sur les éléments de l. Classer la liste l selon $\preceq$ consiste à déterminer une permutation σ de $[1, n]$ telle que :

$$\forall i \in [1, n-1], c_{\sigma(i)} \leq c_{\sigma(i+1)}$$

Le classement est stable si de plus :

$$\forall i, j \in [1, n], (i < j) \wedge (c_i = c_j) \Rightarrow \sigma(i) < \sigma(j)$$

B.1 Tris par sélection

Les tris par sélection consistent à rechercher un élément minimal de la liste, à le placer dans la liste classée, puis à réitérer l'opération sur la liste initiale privée de cet élément minimal.

Pour simplifier, on distingue ici la liste à classer de la liste classée. Dans la pratique, il est assez fréquent de déplacer les éléments dans la liste au lieu de les placer dans une nouvelle liste résultat. Lorsque c'est le cas, on parle de tri *sur place*.

B.1.1 Sélection ordinaire

Le tri par sélection le plus simple consiste à parcourir la liste du début à la fin pour rechercher un élément minimal. On échange alors cet élément minimal avec l'élément situé en tête de la liste, et on recommence avec la liste privée de son premier élément.

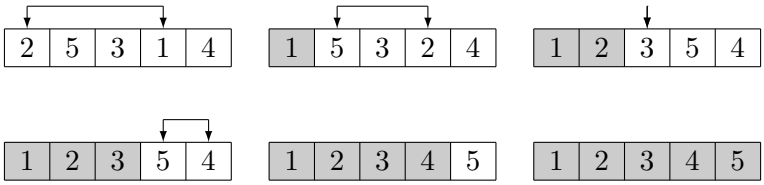
En utilisant un tableau pour représenter la liste à classer, le tri par sélection ordinaire peut être codé comme suit :

```
 1  static void classer(int[] tab) {
 2    // On effectue un classement "sur place", en considérant des
 3    // tableaux de plus en plus courts.
 4    for (int premier = 0; premier < tab.length; premier++) {
 5      // On recherche un élément minimal
 6      int indice_mini = premier;
 7      for (int i = premier + 1; i < tab.length; i++) {
 8        if (tab[i] < tab[indice_mini]) {
 9          indice_mini = i;
10        }
11      }
12      // Si l'élément minimal n'est pas le premier élément du
13      // sous-tableau considéré, on l'échange avec le premier.
14      if (indice_mini != premier) {
15        int tmp = tab[premier];
16        tab[premier] = tab[indice_mini];
17        tab[indice_mini] = tmp;
18      }
19    }
20  }
```

La boucle qui commence à la ligne 4 applique l'algorithme à la liste complète, puis à la liste privée de son premier élément etc.

L'algorithme consiste à rechercher un élément minimal grâce à la boucle qui débute à la ligne 7, puis à permuter cet élément minimal avec le premier élément de la liste (lignes 15 à 17) dans le cas où l'élément minimal n'est pas en tête de liste.

La figure suivante montre l'évolution d'une liste de 5 éléments au cours de ce classement, la zone grisée indiquant la partie classée de la liste, et les flèches indiquant la permutation de l'élément minimal avec le premier élément de la partie non-classée de la liste :



Ce tri est stable puisque, si plusieurs éléments équivalents se trouvent dans la liste, ils seront traités dans leur ordre d'apparition dans la liste. Le premier sera placé en tête de liste avant que le second soit placé en tête de la liste privée du premier etc.

B.1.1.1 Complexité

La boucle qui débute à la ligne 4 est effectuée n fois si n est la longueur de la liste. Chaque exécution du corps de cette boucle s'applique à une sous-liste de longueur k. Dans la boucle qui débute à la ligne 7, on compare le premier élément de la liste à tous les autres, ce qui fait $k-1$ comparaisons. L'algorithme complet effectue donc $\sum_{k=n}^{1}(k-1)$, soit $\sum_{k=n-1}^{0} k = \frac{n(n-1)}{2}$ comparaisons.

Dans tous les cas, la complexité en nombre de comparaisons du tri par sélection est donc en $\mathrm{O}(n^2)$.

En ce qui concerne le nombre de permutations, tout dépend de la disposition des éléments dans la liste :
- si la liste est déjà classée, il n'y aura aucune permutation ;
- si la liste est presque classée, mais que son plus grand élément est en tête, il y aura une permutation à chaque tour, soit $n-1$ permutations.

La complexité en nombre de permutations est donc au pire en $\mathrm{O}(n)$, alors que la complexité en nombre de comparaisons est toujours en $\mathrm{O}(n^2)$. Asymptotiquement, c'est la donc la complexité en nombre de comparaisons qui prime.

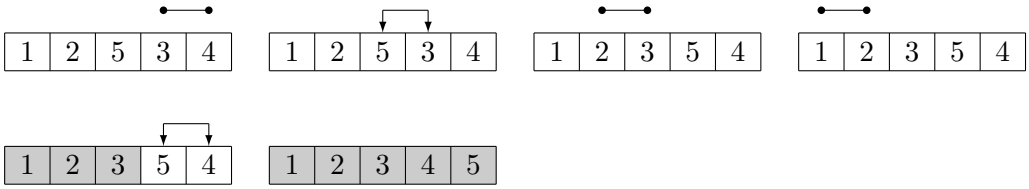
B.1.2 Sélection par transposition, ou « tri à bulles »

Cet algorithme consiste à effectuer les permutations en même temps que la recherche d'un élément minimal : en partant de la fin de la liste à classer, on permute un élément avec le précédent s'ils ne sont pas dans l'ordre. L'élément minimal se comporte donc comme une « bulle » qui remonte vers le début de la liste.

```
1  static void classer(int[] tab) {
2    int premier = 0; // indice du premier élément de la liste
3    int dernier = tab.length - 1; // indice du dernier élément
4
5    // On effectue un classement "sur place", en considérant des
6    // tableaux de plus en plus courts.
7    while (premier < dernier) {
8      int derniere_perm = dernier; // rang de la dernière permutation
9      for (int i = dernier; i > premier; i--) {
10       if (tab[i] < tab[i-1]) {
11         derniere_perm = i;
12         int tmp = tab[i];
13         tab[i] = tab[i-1];
14         tab[i-1] = tmp;
15       }
16     }
17     premier = derniere_perm;
18   }
19 }
```

Tous les éléments situés avant les deux derniers éléments permutés sont correctement classés, il est donc inutile de les comparer de nouveau. On mémorise donc le rang de la dernière permutation dans la variable `derniere_perm`, et on l'utilise pour mettre à jour `premier` afin de ne pas faire de comparaisons inutiles par la suite.

La figure suivante montre l'évolution d'une liste de 5 éléments au cours de ce classement, la zone grisée indiquant la partie que l'on sait être classée puisqu'elle se trouve avant la dernière permutation effectuée. Les flèches indiquent la comparaison suivie de la permutation de deux éléments, alors que les points indiquent une comparaison sans permutation :



La dernière permutation se fait entre les 3e et 4e éléments lors de la sélection du premier élément minimal. Au tour suivant, on sait donc que les 3 premiers éléments sont bien classés, il ne reste donc que les deux derniers à comparer.

B.1.2.1 Complexité

Comme pour le tri par sélection ordinaire, on doit au pire faire remonter $n-1$ fois l'élément minimal en début d'une liste de longueur k. La complexité au pire de cet algorithme en nombre de comparaisons est donc aussi en $\mathrm{O}(n^2)$.

Dans le pire des cas, on est aussi amené à permuter deux éléments à chaque tour de boucle interne, ce qui donne une complexité au pire en nombre de permutations en $\mathrm{O}(n^2)$. On peut montrer que la complexité en moyenne est du même ordre de grandeur.

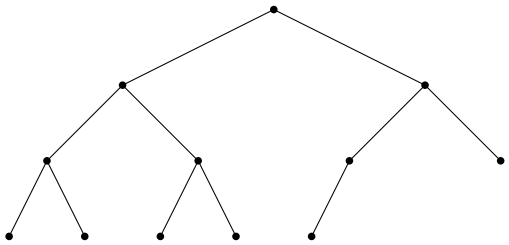
Dans le cas général, le tri à bulle n'est donc pas intéressant comparé au tri par sélection ordinaire. Il n'est intéressant que lorsque les listes sont déjà classées à quelques permutations d'éléments consécutifs près.

Ce tri est stable puisque deux éléments équivalents ne sont jamais permutés.

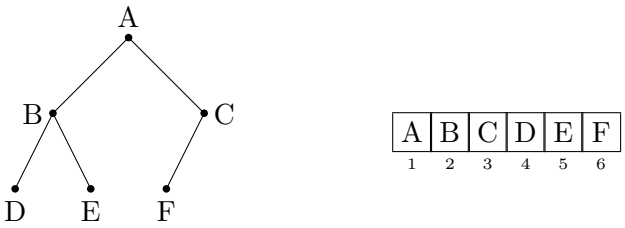
B.1.3 Le tri par tas

Comme son nom l'indique, cet algorithme utilise un *tas* pour sélectionner un élément minimal de la liste. Un tas est la représentation sous forme de liste d'un arbre binaire parfait partiellement ordonné.

Un arbre binaire est un arbre dont les nœuds ont au plus deux fils. Un arbre binaire de est dit *parfait* lorsque tous ses niveaux sauf le dernier sont pleins, et que le dernier niveau est rempli par la gauche :



Un tel arbre se représente facilement sous la forme d'une liste, en plaçant la racine en tête de liste, suivie des ses fils (le gauche, puis le droit), suivis de leurs fils etc. Le fils gauche de l'élément i est l'élément $2i$, son fils droit étant l'élément $2i+1$. Le père de l'élément i est l'élément $\lfloor \frac{i}{2} \rfloor$, comme indiqué ci-dessous :



Lorsque les étiquettes des nœuds de l'arbre sont munies d'un pré-ordre total, on dit que l'arbre parfait est *partiellement ordonné* lorsque tout nœud a une étiquette supérieure ou égale à celle de son père s'il existe. La liste représentant un tel arbre est alors un tas.

La racine d'un arbre parfait partiellement ordonné étant un élément minimal de l'arbre, la tête du tas correspondant est un élément minimal de ce tas. C'est cette propriété qui est utilisée pour sélectionner un élément minimal dans l'algorithme de classement par tas.

On commence donc par construire un tas à partir de la liste à classer. On parcourt pour cela les nœuds internes de l'arbre (ceux qui ne sont pas des feuilles) en partant de celui qui se trouve le plus bas à droite (le nœud C dans l'exemple précédent). Pour chaque nœud, si un des fils a une étiquette supérieure à celle de son père, on permute le nœud avec celui des fils qui a l'étiquette la plus petite, et on recommence jusqu'à ce que le sous-arbre considéré soit partiellement ordonné ou qu'on atteigne une feuille.

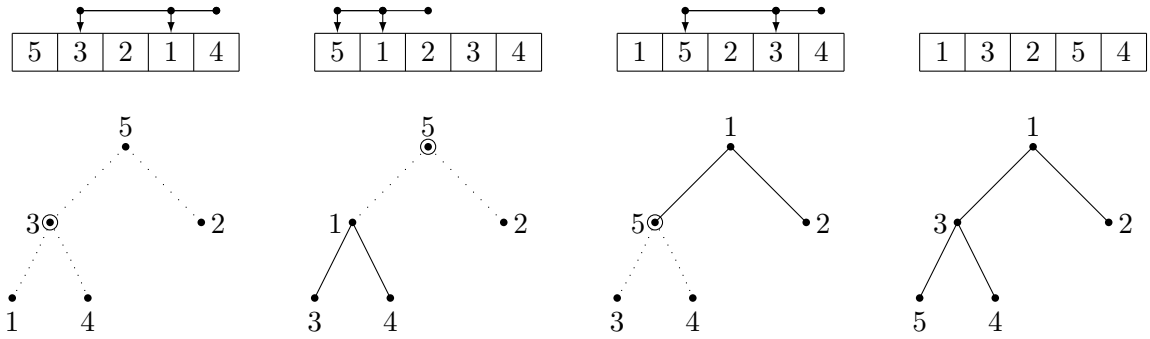
Il est facile de trouver le dernier nœud interne : c'est le père de la dernière feuille, donc l'élément $\lfloor \frac{n}{2} \rfloor$ du tas si n est la taille de la liste à classer. Lorsqu'on utilise un tableau pour représenter la liste, les éléments sont indicés à partir de 0. Le dernier nœud interne a donc pour indice $\lfloor \frac{n}{2} \rfloor - 1$, le fils gauche de l'élément d'indice i a pour indice $2i+1$ et son fils droit $2i+2$. L'algorithme de construction du tas devient :

```
1  static int construireTas(int[] tab) {
2    int courant = (tab.length / 2) - 1;
3    while (courant ≥0) {
4      inserer(tab, courant, tab.length);
5      courant--;
6    }
7    return tab.length;
```

```
8  }
9
10 static void inserer(int[] tab, int courant, int taille) {
11   int gauche = 2 * courant + 1;
12   int droit = 2 * courant + 2;
13   if (gauche ≥taille) {
14     return; // le noeud courant est une feuille
15   }
16   int filsmin = gauche;
17   if ((droit < taille) && (tab[droit] < tab[gauche])) {
18     filsmin = droit;
19   }
20   if (tab[courant] > tab[filsmin]) {
21     echanger(tab, courant, filsmin);
22     inserer(tab, filsmin, taille);
23   }
24 }
```

La méthode `inserer` réorganise l'arbre de façon à ce que le sous-arbre de racine l'élément d'indice `courant` soit partiellement ordonné. La méthode `construireTas` applique `inserer` à chacun des nœuds internes, ce qui rend l'arbre global partiellement ordonné et transforme donc la liste en tas.

La méthode `inserer` commence par rechercher le fils du nœud courant qui a l'étiquette la plus petite, puis, si cette étiquette est supérieure à celle du nœud courant, elle permute les deux nœuds et se rappelle pour former un tas avec le sous-arbre ayant pour racine le fils permuté. La figure suivante montre le déroulement de la construction du tas pour la liste `{5, 3, 2, 1, 4}`. Le nœud inséré est indiqué par un cercle. Les traits en pointillé indiquent les filiations pour lesquelles l'ordre n'est pas forcément respecté, alors que les traits pleins indiquent le respect de l'ordre :

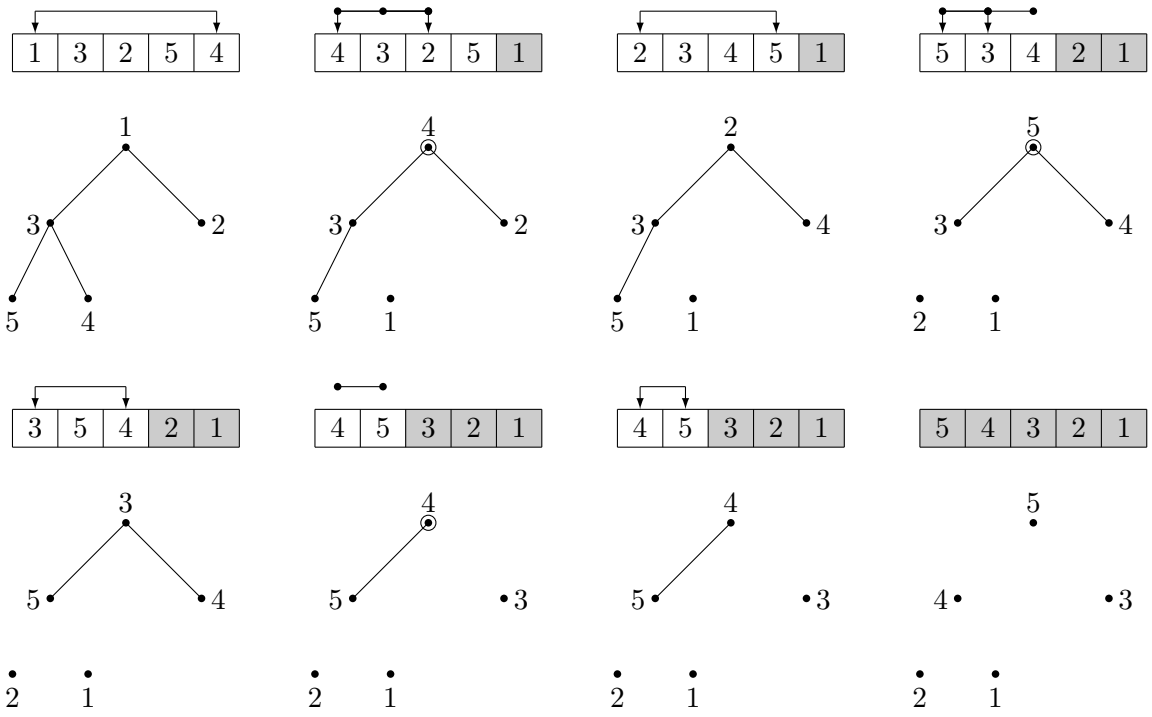


Une fois le tas construit, la sélection d'un élément minimal consiste à prendre l'élément en tête du tas, qui est la racine de l'arbre parfait partiellement ordonné.

Il faut ensuite appliquer de nouveau l'algorithme au tas privé de son premier élément. Mais il serait stupide de reconstruire complètement un tas à partir de la liste privée de son premier élément. On va donc conserver la structure de notre tas en permutant son premier et son dernier élément. La taille du tas va être réduite d'une unité afin que l'élément minimal ainsi placé en fin de liste ne fasse plus partie du tas. L'élément placé en tête de la liste (donc à la racine de l'arbre parfait) sera ensuite inséré pour restaurer la structure de tas. La méthode `oterMin` implémente cette opération :

```
1 static int oterMin(int[] tab, int taille) {
2   // on permute l'élément minimal avec la dernière feuille
3   echanger(tab, 0, taille - 1)
4   taille--; // la taille est réduite
5   inserer(tab, 0, taille); // reconstitution d'un tas
6   return taille;
7 }
```

La figure suivante indique l'évolution du tas lorsque l'on retire ses éléments un à un, en insérant la nouvelle racine pour restaurer la structure de tas. La zone grisée correspond à la partie de la liste qui ne fait plus partie du tas :



Finalement, l'algorithme de classement par tas se code :

```
1  static void classer(int[] tab) {
2    int taille = construireTas(tab);
3    while (taille > 0) {
4      taille = oterMin(tab, taille);
5    }
6  }
```

Le tri par tas classe les éléments de la liste en ordre inverse de l'ordre sur les clefs puisque l'élément minimal est toujours placé en fin de liste. Pour obtenir un classement usuel, il suffit d'utiliser la relation d'ordre inverse lors de la construction du tas, ou d'inverser la liste finale.

Le tri par tas n'est pas stable. Une liste contenant deux éléments équivalents est un contre-exemple de sa stabilité.

B.1.3.1 Complexité

Au pire, `inserer` effectue 2 comparaisons et une permutation avant de se rappeler sur un des fils du nœud courant. Ce fils est l'élément $2i$ ou l'élément $2i+1$ du tas, donc la position du nœud courant est au moins doublée à chaque appel. Comme cette position est limitée à la taille n du tas, il y a au plus $\log_2(\frac{n}{i}) = \log_2(n) - \log_2(i)$ appels récursifs pour insérer l'élément i dans un tas de taille n. La complexité au pire de `inserer` est donc en $\mathrm{O}(log_2(n))$, aussi bien en nombre de comparaisons qu'en nombre de permutations.

Pour construire le tas, on appelle `inserer` pour chacun des $\lfloor \frac{n}{2} \rfloor$ nœuds internes de l'arbre. Considérons k tel que $2^{k-1} \leq i < 2^k$. k est le niveau du nœud i dans l'arbre, la racine étant au niveau 1 et les feuilles les plus basses au niveau h tel que $h-1 \leq \log_2(n) < h$. Le nombre de nœuds situés au niveau k est $2^k - 2^{k-1} = 2^{k-1}$. Pour ces nœuds, la complexité au pire de `inserer` est inférieure à $\log_2(n) - k - 1$. Donc pour chaque niveau k de l'arbre, la complexité au pire de la construction du tas est $2^{k-1}(\log_2(n) - k - 1)$. Le nombre de niveaux est $h \leq \log_2(n) + 1$, mais comme on ne traite que les nœuds internes, seuls les $h-1$ premiers niveaux sont à considérer. La complexité au pire de la construction du tas est donc :

$$C = \sum_{k=1}^{h-1} 2^{k-1}(\log_2(n) - k - 1) < \sum_{k=1}^{h-1} 2^{k-1}(h - k - 1)$$

En posant $i = h - k - 1$, il vient :

$$C < \sum_{i=h}^{2} 2^{h-i} i = \sum_{i=2}^{h} 2^{h-1} 2^{1-i} i = \sum_{i=2}^{h} 2^{h-1} (\frac{1}{2})^{i-1} i = 2^{h-1} \sum_{i=2}^{h} i(\frac{1}{2})^{i-1} < 2^{h-1} \sum_{i=1}^{\infty} i(\frac{1}{2})^{i-1}$$

Or $\sum_{i=1}^{\infty} i x^{i-1}$ est la dérivée par rapport à x de $\sum_{i=1}^{\infty} x^i = \frac{1}{1-x} - 1$ quand $|x| < 1$. La dérivée par rapport à x de $\frac{1}{1-x} - 1$ étant $\frac{1}{(1-x)^2}$, on a donc :

$$C < 2^{h-1} \frac{1}{(1 - \frac{1}{2})^2} = 2^{h-1} \times 4 \leq 4n$$

La complexité au pire en nombre de comparaisons ou de permutations de la construction du tas est donc en $\mathrm{O}(n)$.

Pour classer la liste, on appelle ensuite n fois `otermin` sur des tas dont la taille varie de n à 1. La complexité de `otermin` étant celle de `inserer` plus une permutation, la complexité du classement proprement dit est donc :

$$\sum_{i=n}^{1} \log_2(i) = \log_2(\prod_{i=1}^{n} i) = \log_2 n!$$

D'après la formule de Stirling, $n!$ se comporte comme $n^n e^{-n} \sqrt{2\pi n}$ quand $n \to \infty$. Asymptotiquement, la complexité du classement est donc en $\mathrm{O}(n \log_2(n))$. La complexité du tri par tas est donc en $\mathrm{O}(n \log_2(n))$.

B.2 Classements par insertion

Ces algorithmes de classement consistent à insérer un nouvel élément à sa place dans une liste déjà classée.

B.2.1 Insertion séquentielle

La recherche de la position à laquelle le nouvel élément doit être inséré se fait en parcourant la liste de la fin au début, et en décalant vers la droite les éléments supérieurs à l'élément à insérer :

```
1  static void classer(int[] tab) {
2    for (int n = 1; n < tab.length; n++) {
3      int elem = tab[n]; // élément à insérer
4      int i = n;
5      while ((i > 0) && (tab[i-1] > elem)) {
6        tab[i] = tab[i-1]; // décalage vers la droite
7        i--;
8      }
9      if (i != n) {
10       tab[i] = elem; // insertion.
11     }
12   }
13 }
```

Cet algorithme de classement est stable car lors de la recherche de la position d'insertion, on s'arrête au premier élément inférieur ou équivalent à l'élément à insérer, et on insère après (on compare avec l'élément $i - 1$, mais on insère en i).

B.2.1.1 Complexité

Au pire, la recherche de la position d'insertion dans une liste de longueur k se fait en k comparaisons et k permutations si tous les éléments sont supérieurs à celui que l'on insère.

Le classement consiste à insérer un élément dans des sous-listes de longueur 1 à $n-1$ de la liste à classer. La complexité au pire, tant en nombre de comparaisons qu'en nombre de permutations est donc :

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

Lorsqu'on insère un élément supérieur ou équivalent à tous les éléments de la liste, l'insertion se fait en une seule comparaison. Cet algorithme est donc très efficace pour des listes presque classées.

B.2.2 Insertion dichotomique

Dans cette version du classement par insertion, on utilise une recherche dichotomique pour trouver la position d'insertion : on compare l'élément à insérer avec un élément situé vers le milieu de la liste. La liste étant classée, si l'élément à insérer est plus grand que l'élément milieu, la position d'insertion se trouve dans la deuxième partie de la liste, sinon, elle se trouve dans la première partie.

```
 1  static void classer(int[] tab) {
 2   for (int n = 1; n < tab.length; n++) {
 3     int elem = tab[n]; // élément à insérer
 4     int bas = 0;       // bas de l'intervalle de recherche
 5     int haut = n;      // haut de l'intervalle de recherche
 6     while (bas ≤haut) {
 7       int milieu = bas + (haut - bas)/2;
 8       if (tab[milieu] ≤elem) {
 9         bas = milieu + 1;
10       } else {
11         haut = milieu - 1;
12       }
13     }
14     if (bas < n) {    // si l'élément à insérer n'est pas en place
15       for (int i = n; i > bas; i--) {
16         tab[i] = tab[i-1]; // décaler les éléments vers la droite
17       }
18       tab[bas] = elem; // et insérer.
19     }
20   }
21 }
```

Les éléments situés à gauche de `bas` sont inférieurs ou équivalents à l'élément à insérer. Ceux situés à droite de `haut` lui sont strictement supérieurs. À la fin de la boucle **`while`**, `haut` est situé juste avant `bas`, donc `haut` indique un élément maximal inférieur ou équivalent à l'élément à insérer, et `bas` indique un élément minimal strictement supérieur à l'élément à insérer, si ces éléments existent. L'élément à insérer doit donc l'être entre `haut` et `bas`. Cet algorithme de classement est stable puisqu'un élément est inséré après les éléments équivalents déjà présents dans la liste.

B.2.2.1 Complexité

La recherche dichotomique divise l'intervalle de recherche en 2 à chaque étape jusqu'à ce qu'il soit vide. L'intervalle initial étant de longueur n, le nombre d'étapes est de l'ordre de $\log_2(n)$. Chaque étape compare l'élément à insérer à l'élément milieu, la recherche est donc en $\log_2(n)$ comparaisons. Cette recherche est faite dans les sous-listes de longueur 1

à $n-1$ de la liste à classer. La complexité du classement en nombre de comparaisons est donc :

$$\sum_{k=1}^{n-1} \log_2(k) = \log_2(\prod_{k=1}^{n-1} k) = \log_2((n-1)!) \underset{\text{(formule de Stirling)}}{\approx} \log_2((n-1)^{n-1}) = \mathrm{O}(n \log_2(n))$$

La complexité en nombre de transferts est en $\mathrm{O}(n^2)$, comme pour l'insertion séquentielle car le nombre d'éléments à décaler est le même.

B.3 Le tri rapide

Les tris par insertion insèrent un élément à sa place dans une sous-liste déjà classée de la liste complète, et répètent cette opération sur des sous-listes classées de longueur croissante jusqu'à ce que toute la liste soit classée.

L'algorithme du tri rapide consiste à placer un élément p à sa place dans la liste complète. Il faut pour cela déterminer quels sont les élément de la liste inférieurs à p et ceux supérieurs à p. On en profite pour placer les éléments supérieurs à droite de p et les éléments inférieurs à gauche de p.

Lorsque l'élément p est bien placé dans la liste, les sous-listes situées à gauche et à droite de p peuvent donc être classées indépendamment l'une de l'autre en appliquant le même algorithme.

La phase cruciale du tri rapide est donc le placement de l'élément p — appelé *pivot* — dans la liste, et la constitution des sous-listes gauche et droite contenant respectivement les éléments inférieurs au pivot et les élément supérieurs au pivot. C'est ce qu'on appelle la *partition* de la liste.

Le principe de `partition` est, partant de chaque extrémité de la liste, de rechercher le dernier élément inférieur ou équivalent au pivot et le premier élément supérieur au pivot, puis de les échanger. L'opération est répétée jusqu'à ce qu'il n'y ait plus d'éléments de la liste entre ces deux éléments.

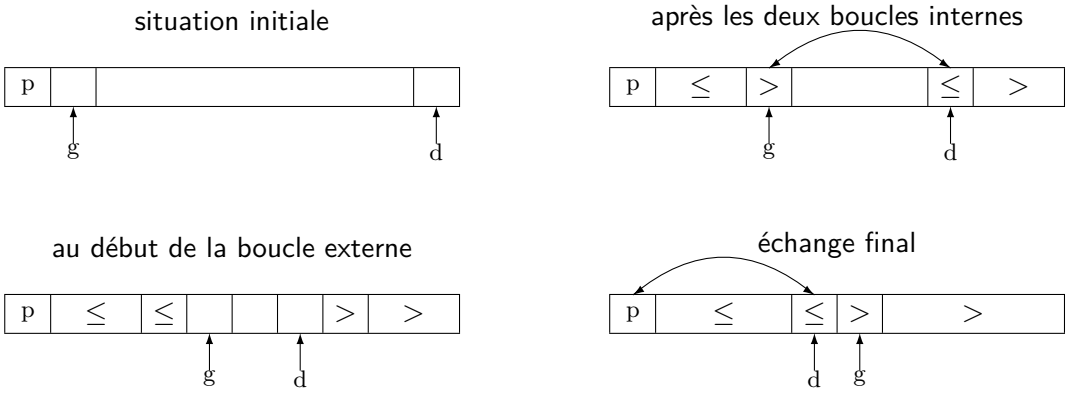
Le dernier élément inférieur ou équivalent au pivot sont alors échangés de façon à ce que tous les éléments situés à gauche du pivot lui soient inférieurs ou égaux, et tous ceux situés à sa droite lui soient supérieurs.

On choisit en général le premier élément de la liste pour pivot, ce qui conduit à l'implémentation suivante de `partition` :

```
1  static int partition(int[] tab, int bas, int haut) {
2    int p = tab[bas]; // on prend le 1er élément pour pivot
3    int g = bas + 1;  // les éléments à gauche de g sont ≤p
4    int d = haut;     // les éléments à droite de d sont > p
5
6    while (g ≤d) {   // tant qu'il y a des éléments indéterminés
7      while (tab[d] > p) { // si l'élément d est > p ...
8        d--;           //    il est à droite de d.
9      }
10     while ((g ≤d) && (tab[g] ≤p)) {
11       g++;           // tab[g] ≤p, donc à gauche de g
12     }
13     if (g < d) {     // tab[g] > p et tab[d] ≤p
14       int tmp = tab[g]; // on les échange
15       tab[g] = tab[d];
16       tab[d] = tmp;
17       g++;           // et on met à jour g et d
18       d--;
19     }
20   }
21   if (bas != d) {   // tab[d] est le dernier élément ≤p
```

```
22      tab[bas] = tab[d]; // donc on l'échange avec p qui est
23      tab[d] = p;        // maintenant à sa place.
24    }
25    return d;
26  }
```

Le schéma suivant montre le déroulement de `partition`, du début où les indices `g` et `d` sont placés à chaque extrémité de la liste, à la fin où `d` indique la position du pivot. À la fin de la première ligne du schéma, les deux boucles internes ont été exécutées, et les éléments indiqués par `g` et `d` vont être permutés. Au début de la deuxième ligne, ils ont été permutés, et `g` et `d` ont été mis à jour. La boucle externe s'exécute jusqu'à se que `g` et `d` se croisent. On arrive alors à la situation finale, et il ne reste qu'à permuter le pivot avec l'élément indiqué par `d`, et à rendre `d` :



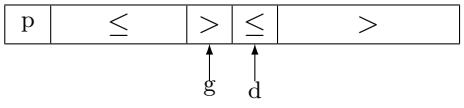
L'algorithme du tri rapide consiste à classer de la même façon les sous-listes créées par `partition`. On a donc :

```
1  static void triRapide(int[]tab, int bas, int haut) {
2    if (bas < haut) {
3      int pivot;
4      pivot = partition(tab, bas, haut);
5      triRapide(tab, bas, pivot - 1);
6      triRapide(tab, pivot + 1, haut);
7    }
8  }
```

B.3.1 Complexité

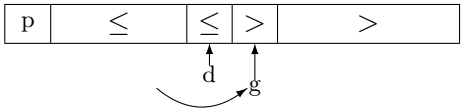
L'algorithme de partition s'arrête lorsque `g` et `d` se croisent. Ceci peut se produire de trois façons :

– les boucles internes s'arrêtent avec $d = g+1$, puis les éléments sont permutés et les deux indices sont mis à jour, rendant `d` inférieur à `g` :



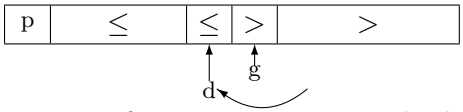
On a alors comparé au pivot les éléments compris entre le pivot et `g` inclus, soit $g - 1$ comparaisons. On a aussi comparé au pivot les éléments compris entre `d` inclus et la fin de la liste. Si n est la longueur de la liste, cela fait $n - d + 1$ comparaisons. Comme $d = g + 1$, cela fait un total de $n - 1$ comparaisons.

– les boucles internes s'arrêtent après que `g` a dépassé `d` vers la droite :



L'élément indiqué par `g` n'a pas été comparé au pivot puisque `g` < `d`. Il y a donc eu $(n - \texttt{d} + 1) + (\texttt{g} - 2)$, avec $\texttt{g} = \texttt{d} + 1$, ce qui donne n comparaisons.

– enfin, les boucles internes peuvent s'arrêter après que `d` a dépassé `g` vers la gauche :



Dans ce cas, la boucle interne qui fait progresser `g` vers la droite n'est pas exécutée. `g` se trouve donc à l'endroit où il a été placé lors de la dernière permutation, et l'élément indiqué par `g` n'a pas été comparé au pivot. Le nombre de comparaisons est donc le même que dans le cas précédent, soit n.

La complexité de `partition` en nombre de comparaisons est donc de l'ordre de n dans tous les cas.

En ce qui concerne le nombre de permutations d'éléments, dans le meilleur des cas (la liste est classée), il n'y en a aucune. Dans le pire des cas, il peut y avoir une permutation à chaque fois qu'un des indices `g` ou `d` progresse. Après une permutation, `g` est incrémenté et `d` est décrémenté, le nombre d'éléments restant à comparer est donc diminué de 2. Comme il y a au départ $n-1$ éléments à comparer au pivot, on aura au maximum $\lfloor \frac{n-1}{2} \rfloor = \lfloor \frac{n}{2} \rfloor - 1$ permutations.

La complexité de `partition` en nombre de permutations d'éléments est donc de l'ordre de n dans le pire des cas.

`triRapide` se rappelle récursivement sur chacune des sous-listes déterminées par `partition`. À chaque niveau k de la récursion, il y a au plus $n-k$ éléments à classer puisqu'on élimine le pivot. Les deux appels à `triRapide` travaillent donc sur des sous-listes dont la somme des longueurs est au plus $n-k$. La complexité de `partition` à ce niveau de la récursion est donc de l'ordre de $n-k$. Comme la récursion s'arrête quand la sous-liste est de longueur inférieure ou égale à 1, il y a au plus $n-1$ niveaux de récursion. La complexité au pire du tri rapide est donc :

$$\sum_{k=0}^{n-1} n - k = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \mathrm{O}(n^2)$$

Au mieux, le niveau de récursion k détermine 2^k pivots : l'appel principal, au niveau 0, détermine un pivot et appelle `triRapide` sur deux sous-listes de plus d'un élément, ce qui permet de déterminer 2 pivots au niveau 1 etc. Le nombre d'éléments à comparer au niveau $k > 0$ est donc au mieux $n - 2^{k-1}$ (on retire à chaque fois les pivots placés au niveau précédent). La récursion s'arrêtant lorsque la sous-liste est de longueur inférieure ou égale à 1, on aura au plus $1 + \log_2(n-1)$ niveaux de récursion. À chaque niveau de récursion k, `partition` est appliquée à des sous-listes dont la somme des longueurs vaut $n - 2^{k-1}$ pour $k > 0$ (la liste étant de longueur n au niveau 0). La complexité du tri rapide dans le meilleur des cas est donc :

$$\begin{aligned}
& n + \sum_{k=1}^{1+\log_2(n-1)} n - 2^{k-1} \\
= \;& n + (1 + \log_2(n-1))n - \sum_{k=1}^{1+\log_2(n-1)} 2^{k-1} \\
= \;& n(2 + \log_2(n-1)) - \sum_{i=2}^{\log_2(n-1)} 2^k \\
= \;& n(2 + \log_2(n-1)) - (2^{1+\log_2(n-1)} - 2^2) \\
= \;& n(2 + \log_2(n-1)) - (2(n-1) - 4) = n\log_2(n-1) + 5 \\
= \;& \mathrm{O}(n \log_2(n))
\end{aligned}$$

La complexité en moyenne du tri rapide pour un tableau de n éléments est la complexité en moyenne du placement pour n élément plus la complexité en moyenne des tris rapides des deux partitions du tableau. Le pivot peut se trouver à n'importe quelle position dans le tableau, on a donc n cas possibles. La complexité en moyenne est donc la moyenne de la complexité de tous ces cas ($n > 1$). Le sous-tableau de droite comporte les éléments d'indice 1 à $p-1$, soit $p-1-1+1 = p-1$ éléments. Le sous-tableau de gauche comporte les éléments d'indice $p+1$ à n, soit $n-(p+1)+1 = n-p-1+1 = n-p$ éléments, d'où la formule suivante :

$$M(n) = M_{\text{placement}}(n) + \frac{1}{n}\sum_{p=1}^{n} M(p-1) + M(n-p)$$

Comme la complexité au pire du placement pour n éléments est n, sa complexité en moyenne est $\leq n$. On a donc, en notant $M(n)$ la complexité en moyenne du tri rapide pour un tableau de n éléments :

$$M(n) \leq n + \frac{1}{n}\sum_{p=1}^{n} M(p-1) + M(n-p)$$

On a de plus $M(0) = M(1) = 0$.

On pose $S_n = n + \frac{1}{n}\sum_{p=1}^{n}(S_{p-1} + S_{n-p})$ pour $n \geq 2$, avec $S_0 = S_1 = 0$.

On a alors :

$$\begin{aligned}
S_n &= n + \frac{1}{n}\left(\sum_{k=0}^{n-1} S_k + \sum_{l=n-1}^{0} S_l\right) \quad \text{avec} \quad k = p-1 \quad \text{et} \quad l = n-p \\
&= n + \frac{1}{n}\left(\sum_{k=0}^{n-1} S_k + \sum_{l=0}^{n-1} S_l\right) \\
&= n + \frac{2}{n}\sum_{k=0}^{n-1} S_k \quad \text{pour} \quad n \geq 2 \\
S_{n-1} &= n - 1 + \frac{2}{n-1}\sum_{k=0}^{n-2} S_k \quad \text{pour} \quad n \geq 3
\end{aligned}$$

On veut soustraire les deux séries pour faire apparaître S_{n-1}.

On va pour cela supprimer le facteur $\frac{2}{n}$ devant la somme :

$$\begin{aligned}
nS_n &= n^2 + 2\sum_{k=0}^{n-1} S_k \quad \text{pour} \quad n \geq 2 \\
(n-1)S_{n-1} &= (n-1)^2 + 2\sum_{k=0}^{n-2} S_k \quad \text{pour} \quad n \geq 3
\end{aligned}$$

d'où :

$$\begin{aligned}
nS_n - (n-1)S_{n-1} &= n^2 - (n-1)^2 + 2\left(\sum_{k=0}^{n-1} S_k - \sum_{k=0}^{n-2} S_k\right) \quad \text{pour} \quad n \geq 3 \\
&= n^2 - (n-1)^2 + 2S_{n-1} \\
nS_n &= n^2 - (n^2 - 2n + 1) + (n+1)S_{n-1} \\
&= 2n - 1 + (n+1)S_{n-1} \\
S_n &= 2 - \frac{1}{n} + (n+1)\frac{S_{n-1}}{n} \\
\frac{S_n}{n+1} &= \frac{2}{n+1} - \frac{1}{n(n+1)} + \frac{S_{n-1}}{n} \\
\text{or} \quad & \quad \frac{1}{n(n+1)} = \frac{1}{n} - \frac{1}{n+1} \\
\frac{S_n}{n+1} &= \frac{2}{n+1} - \frac{1}{n} + \frac{1}{n+1} + \frac{S_{n-1}}{n} \\
\frac{S_n - 1}{n+1} &= \frac{2}{n+1} + \frac{S_{n-1} - 1}{n} \quad \text{pour} \quad n \geq 3
\end{aligned}$$

On pose alors $C_n = \frac{S_n - 1}{n+1}$ et on obtient :

$$\begin{aligned}
C_n &= C_{n-1} + \frac{2}{n+1} \quad \text{pour} \quad n \geq 3 \\
C_0 &= \frac{S_0 - 1}{1} = 0 - 1 = -1 \\
C_1 &= \frac{S_1 - 1}{2} = \frac{0-1}{2} = -\frac{1}{2} \\
C_2 &= \frac{S_2 - 1}{3} = \frac{2-1}{3} = \frac{1}{3}
\end{aligned}$$

$$\begin{aligned}
C_n &= C_2 + \sum_{k=3}^{n} \frac{2}{k+1} \\
&= \frac{1}{3} + 2\sum_{l=4}^{n+1} \frac{1}{l} \\
&= \frac{1}{3} + 2\left(\sum_{l=1}^{n+1} \frac{1}{l} - \sum_{k=1}^{3} \frac{1}{k}\right) \\
&= \frac{1}{3} + 2\left(\sum_{l=1}^{n+1} \frac{1}{l} - (1 + \frac{1}{2} + \frac{1}{3})\right) \\
&= \frac{1}{3} - 2\frac{11}{6} + 2\sum_{l=1}^{n+1} \frac{1}{l} \\
C_n &= 2\mathcal{H}_{n+1} - \frac{10}{3} \\
\text{or} \quad & \quad S_n = (n+1)C_n + 1 \\
S_n &= (n+1)\left(2\mathcal{H}_{n+1} - \frac{10}{3}\right) + 1 \\
&= 2(n+1)\mathcal{H}_{n+1} - \frac{10n}{3} - \frac{7}{3} \\
S_n &= \Theta(n\log(n))
\end{aligned}$$

où $\mathcal{H}_n$ désigne la série harmonique $\sum_{k=1}^{n} \frac{1}{n}$ qui se comporte comme $\mathcal{C} + \log(n)$ quand $n \to \infty$, $\mathcal{C} = 0,577215$ étant la constante d'Euler.

De tous les algorithmes de classement connus, le tri rapide est le plus efficace en moyenne lorsque la taille des listes est suffisamment grande. Pour de petites listes, un classement par insertion sera plus efficace.

Le principal inconvénient du tri rapide est que sa complexité au pire est en $O(n^2)$. S'il est nécessaire d'avoir un classement en $O(n \log(n))$ dans tous les cas, on devra utiliser un tri par tas. Bien que la complexité en moyenne du tri par tas et celle du tri rapide soient toutes deux en $O(n \log(n))$, le tri par tas est plus lent.

B.4 Borne inférieure de la complexité d'un tri

Le tri par tas et le tri rapide ayant une complexité en $O(n \log(n))$, on peut se demander s'il est possible de faire mieux : existe-t-il un algorithme permettant de classer une liste avec une complexité d'un ordre de grandeur inférieur à $O(n \log(n))$?

Nous avons vu au début de ce chapitre qu'effectuer un classement revient à déterminer une permutation des positions des éléments de la liste. Pour une liste de longueur n, le nombre de permutations possibles est $n!$. Si on s'autorise pour seule opération la comparaison de deux éléments l_i et l_j de la liste, le résultat de chaque opération ne peut prendre que deux valeurs : $l_i \leq l_j$ ou $l_i \not\leq l_j$. Si on numérote les permutations de 0 à $n! - 1$, chaque comparaison de deux éléments détermine au mieux un bit de la représentation binaire du numéro de la permutation recherchée. Sur k bits, on peut représenter les entiers de l'intervalle $[0..2^k - 1]$, donc pour représenter les numéros des permutations, il faut un nombre de bits k tel que $2^k - 1 \geq n! - 1$, soit $k \geq \log_2(n!)$. D'après la formule de Stirling, $n! \approx n^n e^{-n} \sqrt{2\pi n}$ quand $n \to \infty$, donc k est au mieux de l'ordre de $n \log(n)$.

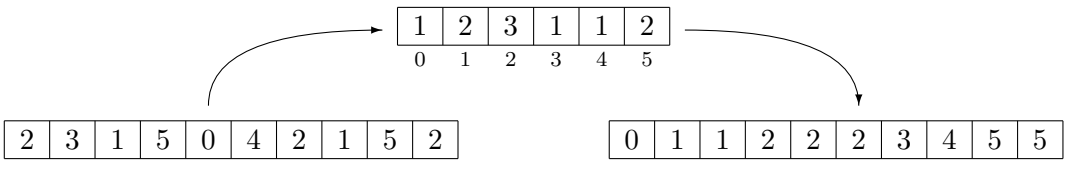
Il est donc impossible de classer une liste avec une complexité d'un ordre inférieur à $n \log(n)$ si on n'utilise que des comparaisons deux à deux d'éléments de la liste.

B.5 Tri par compteurs

Cet algorithme est un exemple de tri qui atteint une complexité inférieure à $n \log(n)$ en exploitant une information supplémentaire sur les éléments à classer.

Si tous les éléments de la liste sont des entiers de l'intervalle $[0..m-1]$, il suffit de compter le nombre d'occurrences de chaque élément dans la liste pour pour les classer : on construit la liste classée en lui ajoutant chaque élément autant de fois que l'indique le compteur correspondant.

La figure suivante montre le classement d'une liste d'entiers de l'intervalle [0..5] à l'aide d'un tableau de 6 compteurs :



Un tel classement s'effectue en n comparaisons (pour calculer les compteurs) et en n déplacements d'éléments (pour remplir la liste classée). Le classement peut se faire en place, mais il nécessite m compteurs. Sa complexité en espace peut donc devenir très importante : pour classer des entiers codés sur 32 bits, il faudrait un tableau de 2^{32} compteurs !

Le tri par compteur fait partie de la famille des tris basés. Ces tris permettent de classer des éléments d'après une clef appartenant à une base finie. Pour des éléments quelconques,

les compteurs sont remplacés par des listes d'éléments équivalents, le résultat étant obtenu par concaténation de ces listes.

B.6 Conclusion

Cet éventail d'algorithmes de classement montre qu'il n'y a pas d'algorithme idéal adapté à toutes les situations. Il ne faut pas oublier que les ordres de grandeur de complexité indiquent un comportement asymptotique pour des tailles de données très grandes. Il ne faut donc pas hésiter à utiliser un simple tri par insertion si on a souvent à classer des listes de quelques dizaines d'éléments, ou si on ajoute des éléments à une liste déjà classée, au fur et à mesure qu'ils sont disponibles. Par contre, ne pas utiliser le tri rapide ou le tri par tas lorsque c'est nécessaire rendra un programme inutilisable.

Composé par pdfTEX 1.4012 le 25 juin 2012