Université de Provence (Aix-Marseille I)

Eléments d'algorithmique en Java

Solange COUPET-GRIMAL

Ce cours s'adresse à des débutants. Les notions essentielles de programmation y sont introduites de façon volontairement non exhaustive, dans le but de favoriser une acquisition correcte, rapide et intuitive des concepts fondamentaux nécessaires à la mise en œuvre en Java des algorithmes étudiés par la suite.

Table des matières

Première partie

Pour débuter rapidement en Java

Chapitre 1

Fondements

Un ordinateur est constitué de divers modules :
- **le processeur** qui *évalue des expressions* (effectue des calculs) et qui *exécute des instructions*
- **la mémoire** dans laquelle sont enregistrées les informations (les programmes, les données et les résultats de calculs effectués sur ces dernières)
- **les périphériques**, dont le *clavier*, *l'écran*, les imprimantes, le disque dur, les réseaux ...

Ces divers constituants communiquent entre eux au moyen de canaux appelés *bus*.

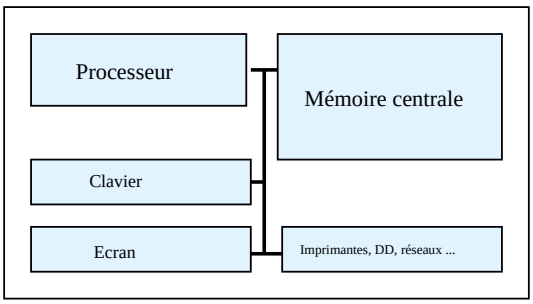


FIGURE 1.1 – structure d'un ordinateur

Instructions et expressions Les programmes sont amenés à modifier l'état de la mémoire (en y enregistrant des résultats par exemple) au moyen d'*instructions*. Dans la suite de ce document, on appellera *effet de bord* toute action du processeur provoquant un changement de l'état de la mémoire (on dira désormais *un changement d'état* sans plus de précision). Une *instruction* a donc, sauf exception, un effet de bord.
Ils font également effectuer par le processeur des calculs codés sous forme d'*expressions*. Par exemple, `2*3+2` et `5>8` sont des expressions que le processeur peut évaluer pour produire 8 et `false` respectivement. L'évaluation de ces deux expressions ne modifie pas l'état de mémoire. Elles sont sans effet de bord.

Les programmes prennent en compte des données pour produire des résultats. Données et résultats peuvent être de types simples (entiers, flottants, caractères, booléens) ou structurés (chaînes de caractères, tableaux, dictionnaires, fiches clients ...). Avant de s'intéresser à la façon dont les programmes sont exécutés par le processeur et à leurs effets, il convient donc de bien comprendre comment les données et les résultats sont codés dans la mémoire.

1.1 Représentation des données en machine

1.1.1 La mémoire

La mémoire d'un ordinateur est une suite ordonnée de *bits*. Un *bit* est un composant élémentaire, pouvant prendre deux états et deux seulement, communément notés **0** et **1**, tout comme un interrupteur électrique n'a que deux positions : **éteint** et **allumé**. Les bits de la mémoire sont regroupés par huit : un groupe de huit bits est un *octet* (*byte* en anglais).

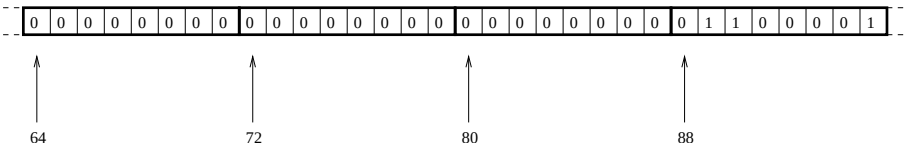


FIGURE 1.2 – Une zone mémoire de 4 octets

Enfin, bits et octets sont numérotés. Ce numéro d'ordre est appelé *adresse* (du bit ou de l'octet). La figure 1.2 représente une zone de la mémoire constituée des quatre octets d'adresses respectives 64, 72, 80, 88.

L'*état* de la mémoire est la suite des états (0 ou 1) des bits qui la composent. L'état de la zone mémoire représentée sur la figure 1.2 est donc :

00000000000000000000000001100001

ATTENTION ! Tout comme un interrupteur est toujours dans une des deux positions *"éteint"* ou *"allumé"*, une zone mémoire est **toujours** dans un certain état. Il est donc absurde de dire qu'*il n'y a rien dans une zone mémoire*.

1.1.2 Codage en mémoire d'une donnée : présentation informelle

Toute donnée, aussi complexe soit-elle, est donc représentée in fine par une suite binaire. Inversement, il faut être capable de décoder une suite binaire pour retrouver la donnée qu'elle représente. Examinons tout d'abord les données de type simple.

En ce qui concerne les entiers, une idée naturelle est de les représenter par leur développement en base 2. Pour prendre en compte le signe, on est amené à utiliser un bit supplémentaire, dit *bit de signe*, qui est 0 pour les entiers positifs et 1 pour les négatifs. Par exemple, sachant que $97 = 1 + 2^5 + 2^6$, on peut envisager de coder l'entier 97 par 01100001 et l'entier -97 par 11100001.

En ce qui concerne les caractères (symboles comme : a, A, 3, π, :, {, $\grave{a}$, ...), il existe un standard international, qui est une liste ordonnée de tous les caractères, appelée *unicode*. L'*unicode* est très complet. En informatique, on utilise souvent une sous-liste de l'*unicode* appelée code ASCII. Le code ASCII contient tous les caractères obtenus à partir d'un clavier QWERTY (y compris l'espace, le "retour chariot", la tabulation et ceux obtenus par combinaison de touches comme les lettres majuscules), mais ne contient ni les caractères accentués ni les caractères des alphabets autres que l'alphabet latin. Dans cette liste (ASCII ou *unicode*), chaque caractère a un numéro d'ordre bien déterminé et c'est donc l'écriture binaire de ce numéro d'ordre qui code le caractère. Voici ci-dessous, par curiosité, la table ASCII. Dans la pratique, il est inutile de connaître ces tables.

000	(nul)	016	(dle)	032	sp	048	0	064	@	080	P	096	‘	112	p
001	(soh)	017	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	(stx)	018	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	(etx)	019	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	(eot)	020	¶(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	(enq)	021	§(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	(ack)	022	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	(bel)	023	(etb)	039	’	055	7	071	G	087	W	103	g	119	w
008	(bs)	024	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009	(tab)	025	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	(vt)	027	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	(np)	028	(fs)	060	<	076	L	092	\	108	l	198	l	124	|
013	(cr)	029	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	(so)	030	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	(si)	031	(us)	047	/	063	?	079	O	095	_	111	o	127	

Table ASCII standard (codes de caractères de 0 à 127)

Ainsi, sachant que le code de la lettre *a* est 97, elle sera représentée par 01100001.

Il résulte de ce qui précède que 01100001 code à la fois l'entier 97 et le caractère *a*. Se pose donc le problème du décodage, en général au moment de l'affichage. Pour reconnaître la donnée représentée par 01100001, il est absolument nécessaire de connaître le *type* de la donnée que l'on cherche : en l'occurrence, un entier ou un caractère.

Il est donc essentiel de préciser le **type** d'une donnée au moment du codage et du décodage.

Une source fréquente de confusion provient du fait que si, dans les écrits courants, le nombre entier *quatre-vingt dix sept* est représenté à l'aide de deux symboles 9 et 7, le nombre entier *sept* est représenté par un unique symbole 7. Ainsi l'entier *sept* et le caractère 7 sont tous deux représentés par le même symbole. L'ambiguïté est alors levée par le contexte. Dans la phrase *7 plus 11 font 18*, il s'agit d'un nombre. Dans la phrase *le nom de code du célèbre agent secret Hubert Bonisseur de La Bath se termine par 7*, il s'agit du caractère. Une telle ambiguïté ne peut subsister dans les programmes qui sont destinés à être automatiquement exécutés sans prise en compte du contexte. Les langages de programmation utilisent donc des notations permettant de faire le distinguo. En Java :

`7` désigne l'entier et `'7'` le caractère
la lettre *a* est désignée par `'a'`

Plus généralement, tout caractère est désigné par un symbole écrit entre deux apostrophes. Ainsi `'7'`, dont le code ASCII (et également *unicode*) est 55 d'après la table ci-dessus, est codé par 010111 puisque $55 = 2^5 + 2^4 + 2^2 + 2 + 1$. Alors que 7 est codé par 0111.

Enfin, la mémoire d'un ordinateur étant de taille finie, on conçoit aisément que l'on ne peut représenter toutes les données. Par exemple, il est hors de question de représenter l'infinité des entiers relatifs. Et même en s'en tenant à une partie finie de cet ensemble, il n'est pas raisonnable de coder un entier tellement grand qu'il occuperait toute la mémoire, ce qui rendrait impossible toute autre opération.

Ainsi, on convient de coder toutes les données d'un même type, sur un nombre d'octets dépendant uniquement du type. Par exemple, en Java, les caractères sont codés sur 16 bits :

'7' est représenté en machine par 0000000000010111
'a' est représenté en machine par 0000000001100001

Dans tout codage binaire, le bit le plus à droite (bit des unités) est appelé *bit de poids faible*.

1.1.3 Types simples et chaînes de caractères

Type	Signification	Taille en bits	Constantes	Opérateurs
`char`	caractère	16	`'3','\n', '\''`	`+ - * / %`
`byte`	entiers	8	97 -97	`+ - * / %`
`short`		16		
`int`		32		
`long`		64		
`float`	flottants	32	`97. 0.97E2`	`+ - * /`
`double`		64	`9.7e1`	
`boolean`	booléens	8	`true false`	`&& || ! ?:`

Pour les types entiers, les opérateurs binaires `/` et `%` ont pour résultat respectivement le quotient et le reste de la division entière des opérandes. Par exemple : 7% 2 = 1 7/2 = 3 -7%2 = -1 -7/2 = -3. Lorsque les opérandes sont de type `char`, il s'agit de l'opération sur l'entier qui les code.

Les valeurs de type `float` sont évaluées avec une précision de 10^{-6}, celles de type `double` avec une précision de 10^{-15}. Les constantes (par exemple `0.97`) sont considérées comme étant de type `double`. Pour en faire un `float`, il faut écrire `0.97f`.

L'opérateur `!` désigne la négation et les opérateurs `&&` et `||` désignent respectivement la conjonction et la disjonction logiques. Il est bon de savoir que lors de l'évaluation d'une expression booléenne de la forme e_1`&&`e_2, l'expression e_2 n'est pas évaluée si e_1 est fausse. De même pour e_1`||`e_2 si e_1 est vraie.

Opérateurs de comparaison Les types simples supportent également les opérateurs suivants :

```
>   >=   <   <=   == !=
```

Ce sont des opérateurs binaires qui renvoient une valeur booléenne, résultat de la comparaison des deux opérandes.

Exemple 1.1 `x!=0 && 1/x>10E3` *est une expression booléenne. Son évaluation ne provoque jamais d'erreur puisque* `1/x` *n'est évalué que si l'expression* `x!=0` *a pour valeur* `true`.

Le type String C'est un type un peu particulier qui désigne les chaînes de caractères et qui n'est donc pas un type simple. Les chaînes de caractères sont notées entre guillemets : `"Ceci est une chaîne de caractères"`. Elles peuvent être concaténées au moyen de l'opérateur binaire `+` et affichées à l'écran au moyen de l'instruction :

```
System.out.print(<chaîne de caractères>)
```

ou encore

```
System.out.println(<chaîne de caractères>)
```

La deuxième instruction affiche à l'écran la chaîne de caractère suivie d'un retour chariot. Ainsi les trois instructions :

```
System.out.println("Il y a dans les bois\nDes arbres fous d'oiseaux")
System.out.print("Il y a dans les bois\nDes arbres fous d'oiseaux\n")
System.out.print("Il y a dans les bois\n" +"Des arbres fous d'oiseaux\n")
```

provoquent l'affichage suivant (on y a fait figurer la position du curseur) :

```
Il y a dans les bois
Des arbres fous d'oiseaux
```

Noter que les opérateurs de comparaison **ne s'appliquent pas** aux chaînes caractères.

L'opérateur ternaire :? Il permet de construire une expression à partir d'une expression booléenne et de deux expressions <expr$_1$ > et <expr$_2$ > comme suit :

<expr booléenne>? <expr$_1$ > : < expr$_2$ >

Cette expression a pour valeur celle de <expr$_1$ > si l'expression booléenne vaut `true` et celle de <expr$_2$ > sinon.

Exemple 1.2 `System.out.print(x==0? "x est nul":"x est non nul")`
a pour effet d'afficher à l'écran `x est nul` *ou* `x est non nul` *selon que la variable* `x` *est nulle ou pas.*

Affichage des valeurs de types simples Quand une telle valeur se trouve en lieu et place d'une chaîne de caractères, elle est automatiquement convertie par Java en sa représentation sous forme de `String`.

Exemple 1.3 *Supposons que* `a`, `b` *et* `d` *soient trois variables de type* `int` *et que l'on ait calculé dans* `d` *le plus grand diviseur commun à* `a` *et* `b`. *L'instruction :*

```
System.out.print("PGCD(" + a + ", " + b + ") = " + d)
```

affiche à l'écran : `PGCD(15, 12) = 3` *si les valeurs de* `a` *et* `b` *sont respectivement 15 et 12 au moment de l'exécution de l'instruction. Java a en particulier converti l'entier quinze codé en machine par :* `0000000000001111` *en la chaîne* `"15"`, *des caractères constituant son écriture dans le système décimal sans qu'il ait été besoin d'appeler pour cela une quelconque fonction de conversion.*

1.2 Les variables et l'affectation

Une variable est une zone mémoire dédiée à l'enregistrement d'une donnée. Elle est caractérisée par :

- **son adresse** : celle du premier octet qu'elle occupe
- **son type** : il définit sa taille et le codage/décodage
- **sa valeur** : c'est son état à un instant donné. Cet état peut évidemment varier, d'où le nom de *variable*.

1.2.1 Déclaration de variable

Toute variable utilisée dans un programme doit être déclarée au préalable comme suit.

<type> < nom de variable> ;

Exemple 1.4
```
int age;
char initiale;
```

Dans un programme, la déclaration d'une variable provoque l'*allocation* en mémoire de l'espace nécessaire pour coder les données du type indiqué. L'adresse de la zone allouée est associée au nom de la variable. A ce stade là, la valeur de la variable existe (bien évidemment, la variable se trouve dans un certain état !) mais est indéterminée : la mémoire allouée a pu être utilisée par un autre programme puis libérée en l'état. La variable est dite *non initialisée*. Plusieurs variables de même type peuvent être déclarées simultanément. Leurs noms sont alors séparés par des virgules.

Exemple 1.5 `float perimetre, surface ;`

1.2.2 Affectation

Une affectation consiste à enregistrer une valeur dans une variable. Il s'agit donc d'une instruction (effet de bord). La syntaxe des affectations est la suivante.

<nom de la variable> `=` <expression> `;`

Exemple 1.6
```
age = 97;
initiale = 'a';
```

Après exécution de ces deux instructions, l'état de la variable `age` est :

00000000000000000000000001100001

et celui de la variable `initiale` est : 0000000001100001.

1.2.3 Déclaration avec initialisation

Toute variable peut être initialisée au moment de sa déclaration selon la syntaxe suivante :

<type> <nom de la variable> `=` <expression> `;`

Ainsi, on peut écrire :

```
int age = 97;
char initiale = 'a';
float surface = 0.16,
      périmètre = 1.6;
```

au lieu de

```
int age;
char initiale;
float surface;
float perimetre;

age = 97;
initiale = 'a';
surface = 0.16;
perimetre = 1.6;
```

Attention : ne pas confondre l'affectation `age = 97` et l'expression booléenne `age == 97`.

1.2.4 Signification du nom de la variable

Le nom d'une variable désigne à la fois son adresse et sa valeur. Dans une affectation, l'ambiguïté est levée par le contexte. L'exécution de l'instruction :

```
age = 2*age +1;
```

a pour effet de rajouter 1 au double de la valeur de `age` et d'enregistrer le résultat à l'adresse `age`. Ainsi, à gauche du signe d'affectation, le nom `age` désigne l'adresse de la variable, tandis qu'à droite il désigne sa valeur.

1.2.5 Optimisations d'affectations

Affectations de la forme `x = x` < *op* > <expression> ;

Considérons l'instruction `x = x+2`. Son exécution requiert :

1. le calcul de l'adresse de `x` pour aller en mémoire récupérer sa valeur (*lecture*)
2. l'évaluation de l'expression à droite du signe d'affectation
3. à nouveau le calcul de l'adresse de `x` pour aller en mémoire enregistrer le résultat obtenu (*écriture*)

On peut optimiser ce type d'affectation à l'aide de la syntaxe suivante.

<nom> <op>= <expression> ;

Pour l'exemple précédent on obtient `x+=2`.
L'optimisation consiste à n'effectuer le calcul de l'adresse qu'une seule fois (suppression de l'étape 3). Ainsi,

```
n = n/2;                                          n /= 2;
x = x*x;          sont avantageusement remplacées par   x *= x;
r = r * x;                                        r *= x;
```

Cas particulier de l'incrémentation et de la décrémentation

Soit `i` une variable entière. On appelle *incrémentation* (respectivement *décrémentation*) de `i` le fait de rajouter 1 (respectivement retrancher 1) à sa valeur. Par exemple `i=i+1;` et `i=i-1;` sont une *incrémentation* et une *décrémentation* de `i`. D'après ce qui précède, on peut améliorer cette écriture en `i+=1;` et `i-=1;` respectivement.

Les incrémentations et décrémentations à répétition sont très fréquentes en programmation. Lorsqu'on parcourt une suite d'éléments indexée de 0 à `n`, on fait varier un indice `i` de 0 à `n` (respectivement de `n` à 0) en l'incrémentant (respectivement le décrémentant) à chaque itération. Il est donc important d'essayer d'optimiser ces instructions.

On remarque alors qu'incrémenter un entier peut se faire beaucoup plus simplement qu'une addition normale qui demande d'examiner tous les bits des deux opérandes. Par exemple, si le bit de poids faible est 0, il suffit de changer ce bit en 1. De même pour la décrémentation. Si l'on utilise les opérateurs `+` ou `-`, avec l'une ou l'autre des deux syntaxes ci-dessus, c'est bien l'algorithme général d'addition et de soustraction qui sera effectué. La syntaxe permettant un calcul optimisé de l'incrément et du décrément est la suivante :

Incrément : <nom de la variable> `++;` ou `++` <nom de la variable> `;`

Décrément : <nom de la variable> `--;` ou `--` <nom de la variable> `;`

Ainsi par exemple, `i++;` et `--i;` ont respectivement le même effet de bord que `i=i+1;` et `i=i-1;` mais sont beaucoup plus efficaces.

1.2.6 Retour sur les types simples : conversions

Conversions implicites Certains types peuvent être implicitement convertis en d'autres. Par exemple tout entier peut être converti implicitement en un flottant et tout caractère en un entier. On peut ainsi écrire :

```
float longueur = 2; short x = 'a';
```
au lieu de
```
float longueur = 2.0f; short x = 97;
```

Conversions explicites Certaines conversions peuvent être forcées (*cast* dans le jargon informatique). La syntaxe est la suivante :

(<type>) <expression>

Exemple 1.7
```
(int) (9.7*x + 2.3)
(short) 'a'
```

Dans le premier cas, on obtient la partie entière et dans le second, le code 97 du caractère `'a'`*, codé sur 16 bits.*

Les conversions explicites mal maîtrisées relèvent de l'acrobatie et sont fortement déconseillées aux programmeurs non avertis.

1.3 Instructions et expressions

Comme évoqué en introduction, on peut distinguer les *instructions* des *expressions*. Les instructions sont *exécutées* et leur exécution affecte l'état de la mémoire, c'est ce que l'on appelle l'*effet de bord*. Les expressions sont *évaluées* et leur évaluation par le processeur provoque un calcul et produit, sauf erreur, un résultat.
Ce qui suit va remettre en cause cette distinction. En Java, comme en C, certaines instructions sont des expressions, en ce sens qu'à la fois elles indiquent un calcul à effectuer et elles ont une valeur. De même certaines expressions ont un effet de bord.

1.3.1 L'affectation comme expression

Considérons l'expression `2*x+1`. A l'évaluation, le processeur effectue une multiplication et une addition. Le résultat obtenu est appelé valeur de l'expression. La mémoire n'est pas affectée par ce calcul. Il n'y a pas d'*effet de bord*. L'expression n'est donc pas une instruction.

En revanche, les affectations sont des instructions. Par exemple l'exécution de l'instruction `x = 2*x+1;` provoque :
- l'évaluation de l'expression `2*x+1`
- l'affectation de la valeur trouvée à `x` (effet de bord)

En Java (et également en C), une affectation a non seulement un effet de bord, mais également une valeur : celle de l'expression située à droite du signe d'affectation. En ce sens, c'est aussi une expression.

$$\texttt{x} = <\texttt{expr}> \begin{cases} \textbf{type} : \text{celui de } \texttt{x} \\ \textbf{valeur} : \text{la valeur v de } <\texttt{expr}> \\ \textbf{effet de bord} : \text{affectation de v à } \texttt{x} \end{cases}$$

En supposant par exemple que `x` et `y` sont deux variables de type `int` et de valeurs respectives 3 et 5, il est possible d'écrire : `x = y =2*(x+y)`. Ce code est interprété comme suit :

$$\texttt{x} \;=\; \texttt{y} \;=\; \underbrace{2*(\texttt{x}+\texttt{y});}_{\substack{\textit{effet de bord : aucun}\\ \textit{valeur : } v_1 = 16}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\substack{\textit{effet de bord : affectation de } v_1 \textit{ à } \texttt{y}\\ \textit{valeur : } v_2 = v_1}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\substack{\textit{effet de bord : affectation de } v_2 \textit{ à } \texttt{x}\\ \textit{valeur : } v_3 = v_2}}$$

Il en résulte que la valeur 16 est affectée à `y` puis à `x`.

1.3.2 Cas particulier : incrémentations et décrémentations

Examinons le cas des incrémentations et décrémentations efficaces d'une variable `i` de type entier. Les instructions `i++;` et `++i;` ont le même effet de bord : elles changent l'état de la variable `i` en rajoutant 1 à sa valeur. Toutefois, considérées comme des expressions, elles n'ont pas la même valeur. L'expression `i++` a la valeur de `i` **avant** incrémentation tandis que l'expression `++i` a la valeur de `i` **après** incrémentation.

Le tableau ci-dessous résume les effets de bords et les valeurs des quatre instructions concernées dans le cas où la valeur initiale de la variable est 5.

`i` avant exécution	expression instruction	`i` après *effet de bord*	valeur de l'expression
5	`i++`	6	5
5	`++i`	6	6
5	`i--`	4	5
5	`--i`	4	4

1.3.3 Expressions à effet de bord

On peut maintenant écrire des expressions à effet de bord, dont le tableau ci-dessous présente quelques exemples.

expression	effet de bord	valeur de l'expression
`i++ < n`	incrémente la variable `i`	`true` si (ancienne valeur de `i`) $<$ n `false` sinon
`--i == 0`	décrémente la variable `i`	`true` si (nouvelle valeur de `i`) $= 0$ `false` sinon
`(x /=2) != 1`	divise par 2 la variable `x`	`true` si (nouvelle valeur de `x`) $\neq 1$ `false` sinon

Le maniement de ces expressions à effet de bord est délicat. Il requiert une bonne connaissance de la sémantique des expressions et des instructions Java et une grande rigueur.

1.4 Blocs et boucles

Les seules instructions rencontrées jusqu'à présent sont des affectations, sous des formes diverses. Toutes se terminent par un point-virgule. Le but de cette section est d'introduire des instructions plus complexes.

1.4.1 Blocs

Les *blocs* permettent de composer séquentiellement une suite d'instructions en les écrivant entre accolades. Un bloc est considéré comme une unique instruction.

Exemple 1.8
```
{r = a%b;
 a = b;
 b = r;}
```

Un bloc peut aussi contenir des déclarations de variables. Ces variables sont *locales* au bloc. Elles sont créées au moment de l'exécution du bloc et restituées dès la fin de l'exécution. Il en résulte qu'elles sont bien évidemment inconnues à l'extérieur du bloc. Si une variable locale porte le même nom qu'un variable globale, cette dernière est masquée (donc inconnue) dans le bloc par la déclaration locale.

Exemple 1.9
```
{int aux;
 aux = a;
 a = b;
 b = aux; }
```

est un bloc qui utilise temporairement une variable auxiliaire `aux` permettent d'échanger les valeurs de deux variables `a` et `b` supposées précédemment déclarées de type `int`. L'espace occupé par `aux` est libéré après l'exécution de la dernière instruction `b = aux;`.

1.4.2 Les instructions de branchement

Le principe des instructions dites *de branchement* consiste à exécuter certaines instructions d'une séquence à partir d'un procédé d'aiguillage. Un cas particulièrement simple et usuel est celui de la conditionnelle.

L'instruction conditionnelle Elle est de la forme :

```
if (<expression booléenne>)
   <instruction_1 >
else
   <instruction_2 >
```

Elle est considérée comme une *unique* instruction dont l'exécution consiste tout d'abord à évaluer l'expression booléenne (dite *expression de contrôle*). Puis l'instruction 1 (ou première branche) ou l'instruction 2 (deuxième branche) est exécutée selon que la valeur trouvée est `true` ou `false`.

Exemple 1.10 *Supposons que* `n` *est une variable de type* `int` *et* `x` *et* `r` *des variables de type* `float`. *On peut écrire :*

```
if (n%2 == 0)
   {n /= 2; x *= x;}
else
   {n--; r *= x;}
```

Il en résulte que si `n` *est pair,* `n` *est remplacé par sa moitié et* `x` *par son carré. Sinon,* `n` *est décrémenté et la variable* `r` *est multipliée par* `x`. *Remarquer que l'instruction :*

```
if (n%2 == 0)
   x = n/2;
else
   x = n-1 ;
```

peut être remplacée par :

```
x = n%2==0? n/2 : n-1;
```

De même, en présence d'une variable booléenne `pair`*, l'instruction*

```
if (n%2 == 0)
   pair = true;
else
   pair = false ;
```

peut être remplacée par :

```
pair = n%2 == 0;
```

Remarque 1.1 La deuxième branche de l'instruction conditionnelle est facultative. Dans ce cas, si l'expression de contrôle a pour valeur `false`, l'instruction est équivalente à l'instruction *vide*, instruction virtuelle dont l'exécution consiste à ne rien faire. Par exemple, l'instruction

```
if(b<a) a = b;
```

est destinée à enregistrer dans la variable `a` le minimum de `a` et de `b`. La variable `a` n'est pas modifiée si elle contenait déjà le minimum.

L'instruction switch Cette instruction a pour but d'exécuter toutes les instructions d'une séquence, à partir de certains points repérés par la valeur d'une variable de contrôle de type entier (`char`, `byte`, `short`, `int` ou `long`). La syntaxe est la suivante.

```
switch (<variable entière>){
    case <valeur_1 > :   <instruction_1 >
    case <valeur_2 > :   <instruction_2 >
    . . .
    case <valeur_n > :   <instruction_n >
    default        :  <instruction_{n+1} >
    }
```

Il s'agit à nouveau d'une unique instruction. Les valeurs doivent être distinctes. Lorsque la variable prend la valeur $<\text{valeur}_i>$, toutes les $<\text{instruction}_j>$ pour $j \in \{i, \ldots (n+1)\}$ sont exécutées séquentiellement. Le mot `default` est un mot clef et remplace toute valeur ne figurant pas dans les `case` précédents. Cette dernière ligne est facultative. En son absence, pour toute autre valeur de la variable, l'instruction se comporte comme l'instruction *vide*. De plus, si une $<\text{instruction}_i>$ est absente, elle se comporte comme l'instruction vide. Enfin, si pour les valeurs $<\text{valeur}_j>$ où $j \in \{1, \ldots i\}$ on ne souhaite pas exécuter les instructions de rang supérieur à i, on utilise l'instruction `break` qui a pour effet de sortir de la boucle.

Exemple 1.11 *On illustre ici, non seulement le fonctionnement de l'instruction* `switch`*, mais également le codage des caractères et les conversions explicites (cast). Supposons que* `ch` *soit une variable de type* `char` *et* `nb` *une variable de type* `int` *préalablement initialisées. Considérons le bloc suivant.*

```
{boolean chiffre = false, minuscule = false;
 switch(ch){
   case'0': case'1': case'2': case'3':case'4':
   case'5': case'6': case'7':case'8':case'9':
      chiffre = true;break;
   case'a': case'b': case'c': case'd':case'e':
   case'f': case'g': case'h':case'i':case'j':
   case'k': case'l': case'm':case'n':case'o':
```

```
   case'p': case'q': case'r':case's':case't':
   case'u': case'v': case'w':case'x':case'y':
   case'z' : minuscule = true;break;
   default : System.out.println("Minuscule ou chiffre attendu");
 }
 if (chiffre){
   nb = 10*nb + ch-'0';
   System.out.println("nb = " + nb);
 }
 else if (minuscule){
   ch += 'A'-'a';
   System.out.println("ch a ete transforme en " + ch);
   System.out.println("Code de " + ch +": " + (short)ch);
 }
}
```

Le résultat de l'exécution de ce bloc est le suivant :

- Si `ch` contient initialement un caractère autre qu'un chiffre ou une minuscule, il s'affiche à l'écran :

  ```
  Minuscule ou chiffre attendu
  ```

- Si `ch` contient initialement le caractère `'t'`, il s'affiche à l'écran :

  ```
  ch a ete transforme en T
  Code de T: 84
  ```

- Si `ch` contient initialement le caractère `'3'` et `nb` la valeur `52`, il s'affiche à l'écran :

  ```
  nb = 523
  ```

Le deuxième cas montre comment transformer une minuscule en majuscule, en utilisant le fait que dans les codes standards, les minuscules, de même que les majuscules, sont consécutives et rangées par ordre alphabétique (voir la table ASCII section 1.1.2). On remarquera que `ch` s'affiche `T` alors que `(short)ch` s'affiche `84` qui est l'entier codant le caractère `'T'`. L'affichage (i.e. le décodage) s'est donc fait selon le type (`char` pour `ch` et `short` pour `(short)ch`).

Le troisième cas utilise l'expression `ch-'0'` (cf. opérations sur les caractères, section 1.1.3) qui, dans le cas où `ch` contient un chiffre (ici `'3'`), a pour valeur l'entier correspondant (ici `3`). Cela repose sur le fait que dans les codes standards, les chiffres sont consécutifs et rangés de `'0'` à `'9'`. On peut de cette façon transformer une suite de caractères, par exemple `'5'`, `'2'` et `'3'` en un entier, ici `523`. Il suffit pour cela d'initialiser `nb` à `0` et d'itérer l'instruction `nb = 10*nb + ch-'0';`.

1.4.3 Instructions while et do-while

Les boucles `while` et `do-while` permettent d'itérer une instruction tant qu'une expression booléenne a la valeur `true`. En général, cette expression dépend de variables dont la valeur est modifiée à chaque itération. Il convient de s'assurer que la valeur de l'expression booléenne converge vers `true` au bout d'un nombre fini d'itérations. Sans quoi, la boucle s'exécute indéfiniment. On dit que "l'on ne sort jamais de la boucle" et que le programme "boucle".

L'instruction while

Cette instruction est la boucle la plus générale du langage Java : toutes les autres ne sont que des particularisations destinées à gagner en efficacité et lisibilité. La syntaxe de l'instruction `while` est la suivante :

```
while (<expression booléenne>)
   <instruction>
```

Il s'agit d'une *unique* instruction, dont l'exécution consiste à évaluer tout d'abord l'expression booléenne (dite *de contrôle*). Si elle a la valeur `false`, l'instruction (corps de la boucle) n'est jamais exécutée : on "n'entre pas dans la boucle". La boucle `while` se comporte comme l'instruction vide et l'exécution du programme se poursuit par l'instruction suivante. Si elle a la valeur `true`, le corps de la boucle est exécuté et le processus *évaluation de l'expression booléenne - exécution du corps de la boucle* est réitéré tant que l'expression booléenne reste vraie.

Exemple 1.12 (L'algorime d'Euclide) *Cet algorithme permet de calculer le plus grand diviseur commun (*PGCD*) de deux entiers a et b. Il s'appuie sur le fait que :*

$$PGCD(a,\ b) = PGCD(b,\ a\%b) \tag{1.1}$$

Il consiste à effectuer des divisions successives, la valeur cherchée étant celle du dernier reste non nul. Soit trois variables `a`, `b` *et* `r`*, supposées de type* `int` *et* a_0 *et* b_0 *les valeurs initiales de* `a` *et* `b`*. Le code de l'algorithme est le suivant :*

```
while (b!=0){
    r = a%b;
    a = b;
    b = r;
}
```

On remarque que l'on ne calcule le reste de la division de `a` par `b` qu'après s'être assuré que `b` est non nul. De plus, à chaque itération, la valeur de `b` décroit strictement puisqu'elle est remplacée par le reste d'une division entière dont elle est le diviseur. Donc elle atteint 0 après un nombre fini d'itérations. Cette instruction ne "boucle" donc pas. On peut démontrer en utilisant la formule (1.1) qu'après chaque itération (sauf la dernière après laquelle `b` est nul) $PGCD($`a`$,\ $`b`$) = PGCD(a_0,\ b_0)$. Une telle relation, conservée par les itérations est appelée *invariant de boucle*. Il est alors facile de prouver qu'à la sortie de la boucle, `a` a pour valeur le plus grand diviseur commun de a_0 et b_0.

L'instruction do-while

L'instruction `do-while` est de la forme :

```
do
   <instruction>
while (<expression booléenne>);
```

et est équivalente à :

```
<instruction>
while (<expression booléenne>)
   <instruction>
```

Autrement dit, **le corps de la boucle `do-while` s'exécute au moins une fois** après quoi l'expression de contrôle est évaluée et le comportement est identique à celui de la boucle `while`. "On entre toujours dans la boucle".

1.4.4 Instruction for

Le contrôle de la boucle `for` se fait au moyen d'une unique variable que l'on appellera provisoirement *compteur*. On indique :

– la valeur initiale du compteur,
– une expression qu'il doit satisfaire pour que s'exécute le corps de la boucle,
– une instruction qui fait évoluer sa valeur à chaque itération.

Sa syntaxe est la suivante :

```
for (<initialisation> ; <expr. booléenne> ; <transformation > )
   <instruction>
```

Elle se comporte comme :

```
<initialisation>
while (<expr. booléenne>){
   <instruction>
   <transformation>
}
```

La déclaration de la variable *compteur* peut se faire en même temps que l'initialisation. Elle est alors locale à la boucle.

Exemple 1.13 (La suite de Fibonacci) *C'est une suite récurrente définie par les relations suivantes :*

$u_0 = 1$
$u_1 = 1$
$\forall n,\ u_{n+2} = u_{n+1} + u_n$

Il est facile de calculer à la main le début de cette suite : 1, 1, 2, 3, 5, 8, 13, 21, ... en partant des deux premiers termes et en ajoutant, à chaque étape, les deux derniers termes obtenus pour en calculer un nouveau. En supposant déclarée une variable `n` de type `int` dûment initialisée, on peut ainsi calculer le nième terme de la suite comme suit.

```
{int a=1, b=1, nouveau ;
 for (int i=1 ; i<n ; i++){
    nouveau = a+b;
    a = b;
    b = nouveau;
 }
}
```

Ici, la variable `i` est déclarée au moment de l'initialisation et est locale à la boucle. Il est facile de démontrer qu'à la sortie `b` contient le terme u_n cherché. En effet, ceci est vrai si `n` vaut 0 ou 1 car dans ce cas, l'expression de contrôle est fausse et on n'entre pas dans la boucle. Sinon, on prouve qu'après chaque itération, `a`$= u_{i-1}$ et `b`$= u_i$ (invariant de boucle). On obtient le résultat en remarquant qu'à la sortie `i`$= n$.
On remarque enfin que l'on peut utiliser la variable `n` comme compteur :

```
{int a=1, b=1, nouveau ;
 for ( ; n>1 ; n--){
    nouveau = a+b;
    a = b;
    b = nouveau;
 }
}
```

Dans ce cas, l'initialisation est "vide" puisque `n` est supposée connue. En notant n_0 la valeur initiale de `n`, on démontre que si $n_0 > 1$, après chaque itération, `a`$= u_{n_0-n}$ et `b`$= u_{n_0-n+1}$. Comme à la sortie `n` vaut 1, le résultat cherché est donc la valeur de `b`.

REMARQUE La boucle `while` utilisée dans l'exemple 1.12 pour calculer le PGCD de `a` et de `b` peut être remplacée par :

```
for ( ; b!=0 ; b=r){
    r = a%b;
    a = b;
}
```

Cette écriture n'est pas nécessairement plus lisible !

Chapitre 2

Java sans objet

2.1 Programme minimum

Un programme Java est structuré en modules appelés *classes*. Informellement, une classe est constituée de *déclarations* et de *méthodes*. Les méthodes sont des sous-programmes présentant une analogie avec les *fonctions* du langage C ou les *procédures* du langage Pascal.

Un programme susceptible d'être exécuté doit :

1. contenir au moins le texte suivant :

```
public class <nom> {

   public static void main (String[ ] args){
     < corps de la méthode main>
   }
}
```

2. être contenu dans un fichier portant le même nom `<nom>` que la classe avec l'extension `.java`.
3. être compilé par la commande : `javac <nom>.java`.
   En l'absence d'erreurs à la compilation, un fichier exécutable appelé `<nom>.class` est alors produit.
4. être exécuté par la commande : `java <nom>` `<arg`$_1$`>` ...`<arg`$_n$`>`
   `<arg`$_1$`>` ...`<arg`$_n$`>` sont séparés par au moins un espace et sont appelés *arguments* du programme. Ce sont des données nécessaires à son exécution. Si le programme n'attend pas de données, aucun argument n'est fourni à l'exécution.

Ces divers points sont détaillés dans les sections suivantes.

2.1.1 La méthode main

Le mot `main` est le nom d'une méthode. Cette méthode est très particulière. C'est la seule dont le nom soit imposé. De plus l'exécution d'un programme consiste à exécuter sa méthode `main`.

Voici un exemple de programme Java, supposé enregistré dans le fichier `LePrintemps.java`.

```
public class LePrintemps{
   public static void main (String[ ] args){
      System.out.println("Il y a dans les bois");
```

```
      System.out.println("Des arbres fous d'oiseaux");
   }
}
```

Ce programme ne requiert aucune donnée. On n'indiquera donc aucun argument à l'exécution. Voici une copie d'écran qui montre les diverses phases décrites ci-dessus.

```
$ javac LePrintemps.java
$ java LePrintemps
Il y a dans les bois
Des arbres fous d'oiseaux
$
```

2.1.2 Les arguments d'un programme

Les paramètres d'un programme sont des chaînes de caractères séparées par au moins un caractère d'espacement. Supposons que le programme soit appelé avec un nombre `n` (éventuellement nul) de paramètres.

L'exécution du programme débute par la création d'un tableau appelé `args` dans la méthode `main`, dont les éléments sont de type `String`, destiné à recevoir des chaînes de caractères. Le type "tableau de String" est noté en Java `String[]`, d'où la déclaration `String[] args` dans l'entête de la méthode. On dit que `args` est un *paramètre* de `main`. Le nombre d'éléments du tableau créé est exactement le nombre `n` d'arguments passés au moment de l'exécution. Les tableaux Java étant indexés à partir de 0, `args` est donc indexé de `0` à `n-1`. L'élément d'indice `i` du tableau est noté `args[i]`.

Voici un exemple de programme requérant deux arguments.

```
public class MonNom{
  public static void main (String[ ] args){
    System.out.println("Vous etes " + args[0] + " " +args[1]);
  }
}
```

Et voici une trace d'éxécution :

```
$java MonNom Georges Sand
Vous etes Georges Sand
```

L'utilisateur ayant indiqué deux paramètres, une zone mémoire est allouée, destinée à recevoir un tableau `args` à deux éléments, `args[0]` et `args[1]` qui eux-mêmes recoivent respectivement `"Georges"` et `"Sand"`.

Si l'utilisateur n'indique pas de paramètres lors de l'exécution du programme ci-dessus, le tableau `args` est vide. Or l'unique instruction de la méthode `main` fait appel à `args[0]` et `args[1]` , qui n'existent pas, puisqu'il n'y a pas eu d'allocation de mémoire pour ces deux éléments. Ceci provoque un message célèbre :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

Les erreurs détectées par Java à l'exécution sont appelées *exceptions*. Le message ci-dessus indique que dans la méthode `main` on a utilisé un indice (ici 0) hors des bornes du tableau.

Sachant que la longueur du tableau `args` est prédéfinie en Java par `args.length`, il est prudent pour éviter l'erreur provoquée par l'abscence d'arguments de modifier le programme ci-dessus de la façon suivante :

```
public class MonNom{

  public static void main (String[ ] args){
    if(args.length>=2)
      System.out.println("Vous etes " + args[0] + " " +args[1]);
    else
      System.out.println("Deux arguments attendus");
    }
}
```

Enfin, on peut rajouter souplesse et généralité de la façon suivante :

```
public class MonNom{

  public static void main (String[ ] args){
   if (args.length ==0)
    System.out.println("je ne sais qui vous etes");
   else{
    System.out.print("Vous etes");
      for(int i=0;i<args.length;i++)
        System.out.print(" " + args[i]);
      System.out.println("\n");
    }
   }
}
```

On obtient les exécutions suivantes :

```
$java MonNom
Je ne sais qui vous etes

$java MonNom Marie de France
Vous etes Marie de France

$java MonNom Marie Madeleine Pioche de La Vergne, Comtesse de La Fayette
Vous etes Marie Madeleine Pioche de La Vergne, Comtesse de La Fayette
```

2.2 Les méthodes de classes (ou statiques)

Les classes Java comportent en général plusieurs méthodes, chacune étant un sous programme dédié à une tâche particulière. Une méthode doit être relativement brève. Dans le cas contraire, il est préférable de la scinder en plusieurs méthodes plus courtes après avoir identifié les divers calculs qui la constituent.

2.2.1 Etude d'un exemple

Supposons que l'on souhaite écrire un programme calculant le PGCD de deux entiers par l'algorithme d'Euclide (cf. section 1.4.3). On peut écrire la méthode `main` de telle sorte qu'elle calcule le PGCD de deux entiers passés en paramètre et affiche à l'écran le résultat. Cependant il est préférable de définir une méthode uniquement dédiée au calcul du PGCD. On obtient ainsi :

```
/*1*/public class ProgPgcd{

/*2*/  public static int pgcd (int a, int b){
/*3*/    int r;
```

```
/*4*/    while (b!=0){
/*5*/        r=a%b;
/*6*/        a=b;
/*7*/        b=r;
/*8*/    }
/*9*/    return a;
/*10*/ }

/*11*/ public static void main(String[] args){
/*12*/   int x = Integer.parseInt(args[0]);
/*13*/   int y = Integer.parseInt(args[1]);
/*14*/   System.out.println("PGCD("+ x+"," +y+") = "+ pgcd(x,y));
/*15*/ }
/*16*/}
```

Dont voici une exécution :

```
$ java ProgPgcd 15 20
PGCD(15, 20) = 5
$
```

Cet exemple appelle divers commentaires et permet d'illustrer un certain nombre de notions.

2.2.2 Commentaires dans un programme

On peut commenter un programme en rajoutant du texte compris entre `/*` et */. Ce texte est ignoré à la compilation. Nous avons pu ainsi faire figurer en commentaire le numéro des lignes du programme sans changer l'exécutable.

La séquence des deux caractères `//` a pour effet de commenter tout le texte qui suit sur la même ligne.

2.2.3 Nombres en argument de programme

On a insisté dans la section 1.1.2 sur la distinction entre l'entier 97 et la chaîne de caractères `"97"`. Nous avons également signalé dans la section 1.1.3 que dès qu'un nombre se trouve à la place d'une chaîne de caractères, il est automatiquement converti en la chaîne qui le représente. En revanche, la conversion inverse ne se fait pas implicitement.

Les arguments d'un programme sont toujours de type `String` (cf. section 2.1.2). Dans l'exécution montrée ci-dessus, `args[0]` prend la valeur `"15"` et `args[1]` prend la valeur `"20"`. Il est nécessaire d'appeller une méthode prédéfinie en Java et nommée `Integer.parseInt` qui étant donnée une chaîne de caractères, renvoie l'entier qu'elle représente. Ainsi, `Integer.parseInt("15")` est l'entier `15`. Ceci explique les lignes `12` et `13` de l'exemple. Une écriture erronée du nombre provoque une exception.

Les méthodes `Byte.parseByte`, `Short.parseShort`, `Long.parseLong`, `Float.parseFloat`, `Double.parseDouble` sont les analogues pour les autres types numériques.

2.2.4 Radiographie de la méthode pgcd

- La ligne `2` est appelée *en-tête* de la méthode.
- Le mot `public` sera expliqué par la suite.
- Le mot `static` indique qu'il s'agit d'une méthode dite *statique*, ou encore *méthode de classe*. Cette notion ne prendra tout son sens qu'au chapitre suivant où seront introduites les méthodes non statiques.

- Le mot `int` indique le *type de retour* : toute exécution de cette méthode produit une valeur de ce type. Lorqu'une méthode ne renvoie aucune valeur, le type de retour est `void` (c'est le cas de `main`).
- Le mot `pgcd` est le *nom* de la méthode et est laissé au choix de l'utilisateur.
- Les entiers `a` et `b` sont appelés les *paramètres* de la méthode.
- A la ligne `3` on déclare une *variable locale* à la méthode appelée `r`.
- Les lignes `5` à `8` calculent le PGCD des valeurs initiales de `a` et `b`. A la sortie de la boucle, ce PGCD est dans `a` (cf. section 1.4.3).
- La ligne `9` indique que la valeur renvoyée est celle de `a`. Le type de cette valeur est bien le type de retour indiqué dans l'en-tête de la méthode.

2.2.5 Comment ça marche ?

Appel de méthode A la ligne `14` dans la méthode `main` figure l'expression `pgcd(x,y)`. On dit qu'à ce moment-là, `main` *appelle* la méthode `pgcd`. L'évaluation de cette expression **provoque l'exécution** de la méthode `pgcd`. Si le programme a été appelé avec les arguments `"15"` et `"20"`, au moment de l'appel les variables `x` et `y` ont pour valeur `15` et `20`. L'état de la mémoire est le suivant (le tableau `args` n'est pas représenté par souci de clarté) :

x	y
15	20

Voici le déroulement précis de l'exécution de la méthode `pgcd`, illustré par les états successifs de la mémoire.

1. Allocation de mémoire pour 3 nouvelles variables `a`, `b` et `r` de type `int`.

x	y	a	b	r
15	20	?	?	?

2. Initialisation automatique des *paramètres formels* `a` et `b` avec les *valeurs* des *paramètres effectifs* `x` et `y`.

x	y	a	b	r
15	20	15	20	?

3. Exécution de la boucle `while` qui se termine quand `b` vaut 0.

x	y	a	b	r
15	20	20	15	15
15	20	15	5	5
15	20	5	0	0

4. Communication à la méthode appelante (ici `main`) de l'entier `5`, valeur de la variable `a` et restitution de toute la mémoire allouée à l'étape 1 (i.e. `a`, `b` et `r`).

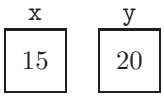
x	y
15	20

2.2.6 Passage des paramètres par valeur

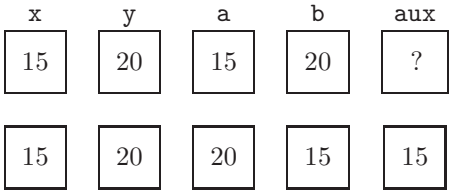
Il résulte de ce qui précède que paramètres et variables locales n'existent que pendant l'exécution de la méthode. Ceci explique qu'ils soient inconnus partout ailleurs. De plus après l'appel `pgcd(x,y)`, la méthode travaille sur des *copies* des variables `x` et `y` (étapes 1 et 2). Par suite, **ces variables sont protégées et en aucun cas affectées** par l'exécution de la méthode. Ceci est un avantage puisque le calcul d'une expression dépendant de variables ne doit pas modifier leurs valeurs, sauf cas particulier expressément indiqué. Ceci empêche toutefois d'utiliser des paramètres pour communiquer des valeurs à l'extérieur. Il est par exemple absurde d'écrire :

```
public static void echange(int a, int b){
  int aux;
  aux=a; a=b; b=aux;
}
```

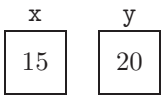
dans l'espoir d'échanger les valeurs des paramètres effectifs. Cette méthode n'échange que les valeurs de copies éphémères et est donc sans effet sur l'environnement. Si par exemple, deux variables `x` et `y` ont pour valeur `15` et `20` à l'appel de la méthode `echange(x, y)`, l'état de la mémoire est :



Pendant l'exécution de la méthode `echange`, la mémoire prend les états suivants :



Après exécution :



La méthode a effectivement procédé à un échange, mais sur les copies de `x` et `y` dans les variables locales perdues en fin d'exécution.

2.2.7 Coût d'un appel de méthode

Il ne faut jamais oublier qu'un appel de méthode provoque l'exécution d'un calcul et a donc un coût en temps d'exécution. Ainsi, si l'on doit affecter à une variable `carre` le carré du PGCD de `x` et `y`, on se gardera d'écrire étourdiment :

```
carre = pgcd(x, y) * pgcd(x, y);
```

mathématiquement correct mais qui provoque 2 fois le même calcul. On écrira plutôt :

```
int d = pgcd(x, y);
carre = d*d;
```

Ce genre de maladresse peut provoquer des explosions combinatoires dans les programmes récursifs. On en verra des exemples par la suite.

2.2.8 L'instruction return

Cette instruction peut être absente dans le cas où le type de retour est `void`. Cette instruction provoque l'arrêt immédiat de l'exécution de la méthode.

Considérons par exemple l'instruction :

```
if(x==0 || y==0)
 return(0);
else
 return(pgcd(x,y));
```

D'après la remarque précédente, le `else` y est inutile. On peut écrire de façon équivalente :

```
if(x==0 || y==0) return(0);
return(pgcd(x,y));
```

Si l'une des deux valeurs au moins de `x` et `y` est nulle, la méthode renvoie `0` et s'arrête. Les instructions suivantes ne sont pas examinées.

2.3 Modularité

Imaginons que l'on souhaite développer une bibliothèque pour l'arithmétique. Pour faire simple, supposons qu'elle fournisse un calcul du PGCD et du PPCM. Une telle bibliothèque est susceptible d'être utilisée par divers autres programmes. En conséquence, elle ne comportera pas elle-même de méthode `main`. On pourra définir une classe publique `Arithmetique`, dans un fichier `Arithmetique.java`, contenant ces deux méthodes.

```
public class Arithmetique{

  static int pgcd (int a, int b){
    int r=0;
    while (b!=0){
       r=a%b; a=b; b=r;
    }
    return a;
  }
  static int ppcm(int a, int b){
    int d = pgcd(a,b);
    return (a*b)/d ;
  }
}
```

Cette bibliothèque peut-être utilisée par des méthodes situées dans une autre classe. Pour cet exemple, on a ainsi défini une classe `ProgTestArith` dans un fichier `ProgTestArith.java` du même répertoire. Elle contient une méthode `main` qui peut appeler les méthodes `pgcd` et `ppcm`, à condition toutefois de préciser à quelle classe elles appartiennent. L'appel ne se fera plus simplement par `pgcd(x,y)` mais par `Arithmetique.pgcd(x,y)` qui signifie *la méthode* `pgcd` *de la classe* `Arithmetique`, d'où l'expression *méthode de classe*. Ceci permet l'existence de diverses méthodes de même nom `pgcd`, situées dans diverses classes $\texttt{C}_1$, ... $\texttt{C}_n$. On peut alors appeler la méthode de la classe $\texttt{C}_i$ de son choix par $\texttt{C}_i$`.pgcd`. Dans l'exemple qui nous intéresse, on peut écrire :

```
public class ProgTestArith{

  public  static void main(String[] args){
    int x = Integer.parseInt(args[0]);
```

```
    int y = Integer.parseInt(args[1]);
    int d = Arithmetique.pgcd(x,y);
    int m = Arithmetique.ppcm(x,y);

    System.out.println("PGCD("+ x +", " + y+ ") = " + d);
    System.out.println("PPCM("+ x +", " + y+ ") = " + m);
  }
}
```

REMARQUE Dans une même classe, plusieurs méthodes peuvent porter le même nom, à condition qu'elles puissent être différenciées par la liste de leurs arguments (en nombre ou en type).

Si des classes sont situées dans le même répertoire, elles ont accès aux méthodes les unes des autres, à condition que ces méthodes ne soient pas déclarées avec l'attribut `private` (auquel cas elles ne sont accessibles que de leur propre classe). L'organisation des classes dans divers répertoires repose sur un mécanisme dit de `package` que nous n'exposerons pas ici. Les attributs `public`, `private` ainsi que l'absence d'attributs sont relatifs aux permissions d'accès aux champs et aux méthodes d'une classe depuis une autre classe selon le `package` dans lequel elle se trouve. Notons que la compilation d'une classe provoque automatiquement la compilation de toutes celles qu'elle utilise.

2.4 Conclusion provisoire

On est loin d'avoir exposé toutes les caractéristiques du langage Java. On a vu pour l'instant qu'il conduit à l'organisation des programmes en divers modules : les classes, elles-mêmes regroupées en *packages* mis à disposition de certains utilisateurs au moyen d'un mécanisme de permission. Ceci permet de gagner en :

- **fiabilité** : le fait d'être conduit à structurer les programmes comme un ensemble de modules simples dont les diverses tâches ainsi que leur portée sont clairement identifiées, aide à l'analyse et à la conception de programmes corrects
- **clarté** : les divers modules sont courts et donc facilement compréhensibles, ce qui est un atout important pour l'utilisation correcte des bibliothèques et la maintenance des programmes.
- **généralité** : dans un processus de conception de programme, on est conduit à généraliser (souvent à peu de frais) certaines parties ad hoc. L'intérêt de généraliser une démarche scientifique est évident : meilleure compréhension de divers phénomènes ayant des points communs, abstraction permettant de se défaire des détails inutiles en se concentrant sur l'essentiel ainsi clairement dégagé et simplifié, réutilisabilité des résultats obtenus à des situations diverses.

Dans les aspects de Java que nous avons décrits jusqu'ici, les *classes* apparaîssent comme des modules regroupant un certain nombre de méthodes dites *de classes* ou *statiques* (leur déclaration comporte l'attribut `static`). On peut dire sommairement que ces classes sont des portions de programmes au sens classique du terme. Le chapitre suivant montre que la notion de classe en Java est bien plus riche que cela.

Chapitre 3

Les objets

3.1 Classes comme structures de données

3.1.1 Qu'est ce qu'un objet ?

Reprenons l'exemple de la classe `Arithmetique` du chapitre précédent. Le calcul du PPCM requérant celui du PGCD, on peut imaginer définir une méthode `PgcdPpcm` qui renvoie à la fois le PGCD et le PPCM de deux entiers passés en arguments. Mais une méthode ne peut renvoyer qu'un seul résultat. L'idée est alors de définir une unique entité composée, en l'espèce, d'un couple d'entiers. En Java de telles données structurées (sauf les tableaux abordés au chapitre 4) sont appelés *objets*.

Objet = Donnée structurée (autre qu'un tableau)

Comment donc définir le type de retour de la méthode `PgcdPpcm` ?

3.1.2 Définir les classes d'objets

On rajoute pour cela dans le fichier `Arithmetique.java` une classe, sans l'attribut `public` puisqu'elle ne porte pas le même nom que le fichier, destinée à définir les couples d'entiers. On choisit de la définir comme suit :

```
class coupleInt{
   /*1*/ int div, mul;

   /*2*/ coupleInt (int a, int b){
   /*3*/    this.div = a;
   /*4*/    this.mul = b;
   /*5*/ }
}
```

Cette classe n'est pas constituée comme celles du chapitre précédent : elle comporte une déclaration à la ligne 1 et elle ne comporte pas de méthodes. En bref, elle n'a de commun avec les classes précédentes que la déclaration à l'aide du mot `class`.

La classe `coupleInt` n'est pas un programme mais plutôt une *structure de données* (ou un *type*) : celle des couples d'entiers.

`this` et ses champs

Pour expliquer le propos et le mécanisme d'une telle classe, référons-nous au texte suivant :

Soit c=(x, y). Quand x vaut 2^3 *et y vaut* $4/2$*, c vaut* $(8, 2)$

Dans la partie *Soit c=(x, y)* qui n'est autre que l'écriture concise de :

Supposons que c soit un couple (x, y)

c désigne un couple virtuel, hypothétique, qui n'a pas d'existence propre, contrairement au couple (8, 2) dont il s'agit par la suite. De façon analogue, la déclaration dans la classe `coupleInt` :

```
int div, mul;
```

est sémantiquement équivalente à :

Soit this = (div, mul) où div et mul sont des entiers

Le mot `this` désigne donc un *objet* hypothétique, qui n'a pas d'existence réelle en mémoire. C'est l'objet virtuel dont il s'agit partout à l'intérieur de la classe, d'où le nom `this` : "celui-ci, que la classe a pour but de décrire".

Les deux composants de `this` sont appelés ses *champs* et notés respectivement `this.div` et `this.mul`.

Le constructeur

La déclaration de la ligne 1 ne permet pas à elle seule de construire des objets de la classe `coupleInt`. Il faut pour cela définir un *constructeur* (lignes 2 à 5). Il s'agit d'une méthode très particulière puisque :

1. elle n'est pas précédée du mot `static`.
2. elle n'a pas de nom
3. son type de retour est celui des objets de la classe, soit ici `coupleInt`.

Ce constructeur permet à toute méthode qui le souhaite, de construire des objets bien réels de la classe en précisant les deux valeurs prises par les champs `div` et `mul`.

En conclusion

Une classe, vue comme structure de données, est une description d'objets. Elle en indique les *champs* qui les constituent et fournit un ou plusieurs constructeurs pour permettre les créations ultérieures. Tout comme les méthodes, les divers constructeurs doivent se distinguer par la liste de leurs paramètres.

A RETENIR

1. Il s'agit d'une description de la façon dont sont constitués les objets de cette classe
2. `this` est un objet hypothétique sans existence réelle dont la classe est dédiée à la description.
3. Il faut un constructeur au moins pour créer **ultérieurement** de réels objets
4. **On n'a créé ici aucun objet** : il n'y a pas eu d'allocation de mémoire

Désormais, on dira comme il se doit en Java *objet de classe* `coupleInt` et non *objet de type* `coupleInt`.

3.1.3 Créer des objets

Le mécanisme de création d'objets est illustré par l'exemple de la méthode `PgcdPpcm`. Un fichier `ArithBis.java` contient la classe `coupleInt` de la section 3.1.2 suivie de la classe :

```
public class ArithBis{

   public static coupleInt PgcdPpcm(int a, int b){

      int r, a0 = a, b0 = b;
      coupleInt resultat;

      while(b!=0){
         r = a % b;
         a = b;
         b = r;
      }
      resultat = new coupleInt(a, (a0*b0)/a);
      return resultat;
   }
}
```

On remarque que l'on a sauvegardé les valeurs initiales de `a` et `b` dans `a0` et `b0` puisqu'elles sont utilisées pour le calcul du PPCM.

Les objets sont repérés par leurs adresses

Considérons la déclaration :

```
coupleInt resultat;
```

Contrairement à ce que pourrait laisser croire une telle déclaration, la variable `resultat` n'est pas destinée à recevoir un objet de la classe `coupleInt`, mais **l'adresse de** (ou de façon équivalente un **pointeur vers**, ou une **référence à**) un tel objet. Au moment de l'exécution, cette déclaration provoque l'allocation d'un espace mémoire destiné à recevoir une adresse qui prend automatiquement la valeur `null`, constante prédéfinie qui ne référence aucun objet. Soit :

Création

L'objet n'est réellement créé dans cette méthode qu'au moment de l'évaluation de l'expression : `new coupleInt(a, (a0*b0)/a)`. On utilise ici le constructeur `coupleInt`, avec les valeurs effectives des champs, soit `a` et `(a0*b0)/a`, le tout précédé par le mot clé `new`. Ceci a pour effet d'allouer une zone mémoire pour les deux champs, l'un de valeur `a`, l'autre de valeur `(a0*b0)/a`. De plus le résultat de l'évaluation est l'adresse de cette zone. Ceci est illustré par la figure 3.1 dans le cas où les valeurs initiales `a0` et `b0` sont 15 et 20.

Par suite, après l'affectation `resultat = new coupleInt(a, (a0*b0)/a);`, la variable `resultat`

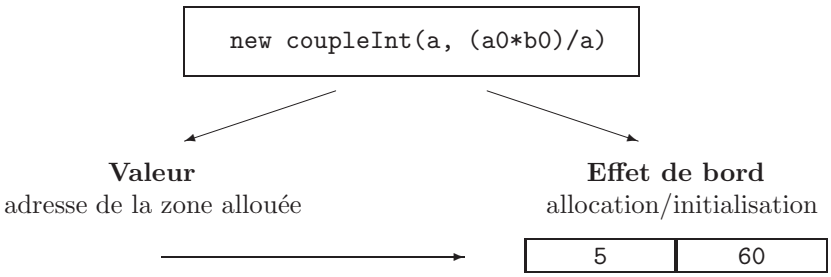
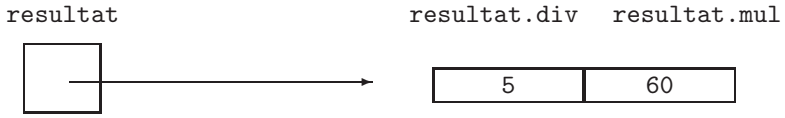


FIGURE 3.1 – Création d'un objet

a pour valeur l'adresse du couple d'entiers nouvellement créé. Ce couple d'entiers est appelé *objet de la classe* `coupleInt`, ou encore *instance* de cette classe. Les champs `div` et `mul` de cette instance particulière sont désignés respectivement par `resultat.div` et `resultat.mul`.



Il est clair qu'il est absurde d'accéder aux champs d'un objet qui n'a pas encore été créé. Ainsi la séquence : `coupleInt resultat;`
`resultat.div = 0;`

provoque une exception : `NullPointerException`, qui indique que le *pointeur* `resultat` vaut `null` et donc `resultat.div` et `resultat.mul` n'existent pas.

Toute expression de la forme <**nom**>.<**attribut**> provoque une exception quand le pointeur <nom> vaut `null`, i.e. quand l'instance prétendument référencée par <nom> n'a pas été créée.

De plus, il résulte de tout ce qui précède, qu'une simple affectation `c1 = c2` d'une variable de la classe `coupleInt` à une autre ne recopie pas l'objet mais uniquement son adresse. Pour dupliquer un objet, il faut le faire explicitement de la façon suivante :

```
c1 = new coupleInt(c2.div, c2.mul);
```

On peut remarquer maintenant que la méthode `PgcdPpcm` de la classe `ArithBis` peut s'écrire plus simplement

```
public static coupleInt PgcdPpcm(int a, int b){
   int r, a0 = a, b0 = b;
   while(b!=0){
      r = a % b;
      a = b;
      b = r;
   }
   return (new coupleInt(a, (a0*b0)/a));
}
```

Cette méthode est testée par le petit programme ci-dessous :

```
public class ProgTestArithBis{

  public  static void main(String[] args){
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    coupleInt divmul = ArithBis.PgcdPpcm(x, y);

    System.out.println("PGCD("+ x +"," + y+ ") = " + divmul.div);
    System.out.println("PPCM("+ x +"," + y+ ") = " + divmul.mul);
  }
}
```

dont voici un exemple d'exécution :

```
$ java ProgTestArithBis 21 15
PGCD(21, 15) = 3
PPCM(21, 15) = 105
```

3.2 Classes comme boîtes à outils sécurisées

Dans le chapitre 2, nous avons présenté des classes qui étaient des portions de programmes, structurées en méthodes dites de classes. Dans la section 3.1, nous avons présenté une classe qui ne faisait que définir une structure de données et permettre la création d'*objets* ou *instances* de cette classe. Ces deux cas particuliers assez différents au premier abord ne s'opposent en réalité pas. En effet, la notion de *classe* en Java est suffisamment générale pour offrir un cadre unique à ces deux situations.

3.2.1 Méthodes d'instance

Supposons que l'on souhaite développer une bibliothèque pour la géométrie plane. Les points du plan (supposé muni d'un repère) seront désignés par un couple de flottants, leur abscisse et leur ordonnée. De façon analogue à la classe `coupleInt` de la section précédente, on peut définir une classe `point` comme suit :

```
public class point{
    double abs, ord;

    public point (double x, double y){
        this.abs = x;
        this.ord = y;
    }
}
```

La structure de donnée étant définie par les champs et le constructeur, il est possible de rajouter des méthodes dans cette classe. Par exemple une méthode qui calcule le milieu de deux points :

```
public static point milieuStatique(point p1, point p2){
        return (new point((p1.abs+p2.abs)/2, (p1.ord+p2.ord)/2));
}
```

Cette méthode est une méthode statique de la classe `point`. Ses paramètres sont deux pointeurs `p1` et `p2` vers des points. Elle construit un nouveau point dont les coordonnées sont les demi-sommes des coordonnées des points référencés en paramètres et elle renvoie son adresse. A l'extérieur de la classe `point` et en supposant avoir créé deux points `a` et `b`, on appelle sans surprise cette méthode en indiquant ainsi qu'il s'agit de la méthode `milieuStatique` de la classe `point` :

```
point c = point.milieuStatique(a,b);
```

La classe `point` définit donc à la fois des objets (les points) et une méthode `milieuStatique`. En ce sens elle généralise comme annoncé, les deux notions de classes (programme ou structure de données).

Toutefois, dans cette situation, il est préférable de définir des méthodes (ici calculant des résultats en liaison avec la géométrie) qui s'appliquent à l'objet virtuel `this` que la classe définit, plutôt qu'à des points quelconques. De telles méthodes ne sont plus qualifiées par le mot `static`. Ainsi, à la méthode `milieuStatique`, on préfèrera :

```
public point milieu(point p){
     return (new point ((this.abs+p.abs)/2, (this.ord+p.ord)/2));
}
```

De toute évidence cette méthode crée un nouveau point, milieu de `this` et du point référencé par le paramètre `p`. A l'extérieur de la classe `point`, et en présence de deux points `a` et `b`, on écrira alors :

```
point m = a.milieu(b);
```

On indique ici qu'il s'agit de la méthode `milieu` pour laquelle le point virtuel `this` que définit la classe est instancié par le point bien réel `a`. D'où le nom de *méthode d'instance*. Si l'on écrit maintenant :

```
point quart = m.milieu(a);
```

on appelle la méthode `milieu` de l'instance `m` qui remplace `this` pendant l'exécution.

La classe `point` peut de la même manière être complétée par diverses méthodes d'instance relatives à la géométrie, comme celle qui calcule le translaté de `this` ou celle qui calcule son homothétique dans une homothétie de centre O et de rapport `k`, pour ne citer qu'elles.

3.2.2 Privatisation des champs

La classe `point` peut être vue comme une structure de données complétée d'une boîte à outils (les méthodes d'instance) pour manipuler correctement ces données. L'idée est de définir une bonne fois pour toutes et de façon sûre, le calcul du milieu, du translaté, de l'homothétique et de toutes les opérations intéressantes, en interdisant à l'utilisateur de manipuler les points autrement qu'avec ces méthodes. C'est la philosophie de la programmation objet.

Supposons qu'un utilisateur de la classe `point`, en présence d'un point `a`, souhaite le translater d'un vecteur de coordonnées (2, -2). Il pourrait étourdiment écrire : `a.abs += 2; a.ord -=2;` et perdre ainsi la valeur antérieure de `a`, ce qui n'est pas en général souhaité.

La méthode `translate` ci-après conserve le point auquel on s'intéresse (autrement dit `this`) et calcule son translaté :

```
public point translate (double x, double y){
   return (new point(this.abs+x, this.ord+y));
}
```

La façon correcte de translater alors un point `a` serait la suivante :

```
point b = a.translate(2, -2);
```

On évite ce type d'erreurs en contraignant les utilisateurs à ne manipuler les points qu'au moyen des méthodes que leur fournit la classe `point`. Pour ce faire, on *protège en écriture* les champs des objets en les affectant du qualificatif `private`. Il deviennent alors inaccessibles de l'extérieur de la classe : on ne peut donc en modifier la valeur. Mais on ne peut plus de ce fait avoir connaissance des coordonnées des points qu'on utilise, ce qui n'est pas souhaitable. On fournira donc deux méthodes qui renverront ces valeurs. Voici une classe `point` écrite dans cet esprit.

```
public class point{
    private  double abs,ord;

    public point (double x, double y){
        this.abs = x;
        this.ord = y;
    }
    public double getAbs(){
        return this.abs;
    }
    public double getOrd(){
        return this.ord;
    }
    public point milieu(point p){
      return (new point ((this.abs+p.abs)/2,(this.ord+p.ord)/2));
    }
    public point translate(double x, double y){
        return (new point(this.abs+x, this.ord+y));
    }
    public point homothetique(double k){
        return (new point(this.abs*k, this.ord*k));
    }
    public String toString(){
        return( "(" + this.abs + ", " + this.ord + ")");
    }
}
```

Si `a` est un point, on pourra écrire :

```
x = a.getAbs(); y = a.getOrd(); b = a.translate(1.2, 1.3);
```

Mais en aucun cas on ne pourra modifier directement les coordonnées de `a`. Pour ce faire, il faut créer un nouveau point et l'affecter à `a`.

```
a = new point (..., ...);
```

ce qui est sémantiquement correct puisque changer les coordonnées d'un point revient à considérer un nouveau point.

Cet exemple nous donne également l'occasion de présenter une façon d'afficher agréablement des objets. C'est le propos de la section suivante, qui décrit la dernière méthode de la classe `point`.

3.2.3 La méthode toString

Toute classe définissant des objets peut être pourvue d'une méthode dont le nom : `toString` et le type de retour : `String` sont imposés. Elle renvoie la chaîne de caractères que l'on choisit pour afficher l'objet `this`. Si l'on décide par exemple d'afficher un point comme le couple (x, y) de ses coordonnées, on écrira :

```
 public String toString(){
        return( "(" + this.abs + ", " + this.ord + ")");
}
```

Supposons, qu'à l'extérieur de la classe `point`, `a` soit le point de coordonnées 1.2 et 3.4. L'affichage à l'écran de :

```
a : (1.2, 3.4)
```

devrait logiquement se faire par la commande :

```
System.out.println("a : " + a.toString());
```

L'intérêt de cette méthode est que `a` sera automatiquement remplacé par `a.toString()` dès qu'il se situe dans un contexte où une chaîne de caractères est attendue. C'est ainsi que l'on obtient le même affichage que ci-dessus en écrivant simplement :

```
System.out.println("a : " + a);
```

Cela explique qu'il n'ait pas été nécessaire de convertir les nombres en chaînes de caractères pour les afficher : chaque type de nombres est doté d'une méthode `toString` que l'on a appelée implicitement sans le savoir.

3.3 Variables statiques et constantes

La classe `point` peut à son tour être utilisée pour définir une classe relative aux cercles comme suit :

```
public class cercle{

    private point centre;
    private double rayon;

    public cercle (point c, double r){
        this.centre = c;
        this.rayon = r;
    }
    public point getCentre(){
        return this.centre;
    }
    public double getRayon(){
        return this.rayon;
    }
    public double circonference(){
        return(2*Math.PI * this.rayon);
    }
    public double surface(){
        return(Math.PI * this.rayon * this.rayon);
    }

    public cercle translate(double x, double y){
        return (new cercle(this.centre.translate(x,y), this.rayon));
    }
    public cercle homothetique(double k){
        return(new cercle(this.centre.homothetique(k), rayon*k));
    }
    public String toString(){
        return ("Centre : " +  this.centre +"\nRayon: "+ this.rayon +"");
    }
}
```

La création de cercles se fait selon le schéma suivant :

`cercle C = new cercle(new point(x,y), r);`

où `x`, `y` et `r` sont supposés être des variables de type `double` préalablement initialisées.

La méthode `toString` a été écrite pour produire un affichage des cercles comme suit :

```
Centre : (3.0, 3.0)
Rayon: 2.0
```

On a de plus utilisé la constante `Math.PI`, prédéfinie dans la classe `Math`. Les constantes des bibliothèques Java sont écrites en majuscules et les programmeurs se plient en général à cette convention bien qu'elle ne soit pas imposée.

Imaginons que l'on ne souhaite pas une valeur de π d'une telle précision, et que l'on décide de définir sa propre constante $\pi = 3.14$ par exemple. La première idée est de définir une variable `pi` dans la classe `cercle` initialisée à cette valeur. On aurait ainsi :

```
public class cercle{

   double pi = 3.14;
   private point centre;
   private double rayon;
   ...
```

Cette solution n'est pas la bonne car `pi` devient un nouveau champ des cercles : un cercle est alors défini par son centre, son rayon et "son `pi`" !. Il y aura ainsi autant de copies de `pi` que d'instances de cercles, ce qui est absurde. On souhaite donc qu'il n'y ait qu'une variable `pi`, commune à tous les cercles. Cette variable ne devant dépendre que de la classe `cercle` et non pas des instances, on la déclare avec l'attribut `static`. On dit alors que `pi` est une variable *de classe* ou une variable *statique*. La classe `cercle` débute ainsi :

```
public class cercle{
   static double pi = 3.14;
   private point centre;
   private double rayon;
   ...
```

Le nom de cette variable à l'extérieur de la classe est `cercle.pi`.

A RETENIR Tout comme on distingue méthodes d'instance et méthodes statiques (ou de classe), on distingue les variables d'instance (ou champs) des variables statiques (ou de classe) qui n'existent qu'en un seul exemplaire, quel que soit le nombre d'instances en cours.

Enfin, dans le cas de cet exemple, plutôt que de définir une variable, il serait préférable de déclarer une constante. Le valeur de `pi` n'étant pas destinée à être modifiée dans le programme, il faut la protéger en écriture pour éviter des erreurs du genre :

```
pi*=2; return (pi*this.rayon);
```

dans le but de calculer le périmètre de `this`. Chaque calcul de périmètre doublerait la valeur de `pi` pour toute la suite du programme !

Pour déclarer la valeur de π sous forme non modifiable, on utilise l'attribut `final`. On obtient donc ici :

```
public class cercle{
   final static double PI = 3.14;
   private point centre;
   private double rayon;
   ...
```

Dans la classe `cercle`, cette constante a pour nom `PI` et à l'extérieur, elle se nomme `cercle.PI`.

3.3.1 `this` implicite

Il faut enfin signaler que le mot `this` peut être omis dans la classe qui le définit, dès que l'on fait appel à l'un de ses attributs. Par exemple :

```
public class point{
    double abs, ord;
    public point (double x, double y)
        { this.abs = x; this.ord = y; }

    public point translate (double x, double y)
        { return (new point(this.abs+x, this.ord+y)); }
}
```

peut être remplacé par :

```
public class point{
    double abs, ord;
    public point (double x, double y)
        { abs = x; ord = y; }

    public point translate (double x, double y)
        { return (new point(abs+x, ord+y)); }
}
```

Chapitre 4

Les tableaux

4.1 Tableaux à une dimension

Il s'agit d'enregistrer une suite finie d'éléments de même type dans des zones mémoires contigües. Par exemple, la suite d'entiers (10, 8, -2, 5, 4) sera enregistrée en mémoire comme suit :

···	10	8	-2	5	4	···

4.1.1 Déclarations

La déclaration des tableaux se fait selon la syntaxe suivante :

<type> `[ ]` <nom du tableau> ;

où <type> désigne le type des éléments du tableau.

Exemple 4.1
```
int [ ] T;
String [ ] args;
```

On a vu que la déclaration d'une variable de type simple provoque l'allocation d'une portion de mémoire de taille adaptée au type. Les déclarations de tableaux comme celles de l'exemple ne permettent pas l'allocation de mémoire pour un tableau, puisqu'à ce moment-là, le nombre d'éléments est inconnu. En conséquence, de façon analogue aux objets, **aucun tableau n'est créé au moment de la déclaration. Les variables créées (ici `T` et `args`) sont des adresses de tableaux dont la valeur initiale est `null`.**

4.1.2 Création

Comme pour les objets, la création d'un tableau se fait explicitement avec le mot clé `new`, selon la syntaxe :

<nom du tableau> `= new` <type> `[`<taille>`];`

Exemple 4.2 `T = new int [5];` *provoque une allocation de mémoire décrite par la figure 4.1.*

Cinq variables consécutives de la taille d'un entier, nommées `T[0]`, `T[1]`, `T[2]`, `T[3]` et `T[4]` ont été allouées. A ce stade-là, elles ne sont pas encore initialisées. La variable `T` a pour valeur l'adresse du premier élément du tableau. L'initialisation des éléments d'un tableau se fait par les affectations :

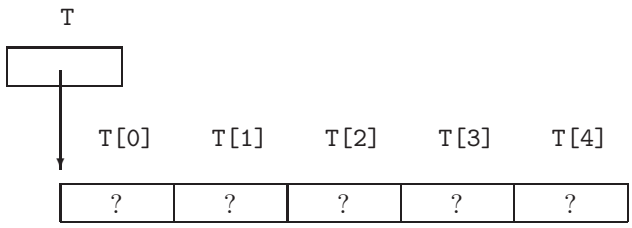


FIGURE 4.1 – Création d'un tableau d'entiers à 5 éléments

```
T[0] = 10; T[1] = 8; T[2] = -2 ; T[3] = 5; T[4] = 4;
```

L'état du tableau est alors décrit par la figure 4.2.

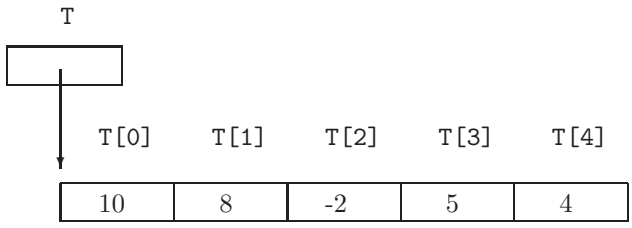


FIGURE 4.2 – Tableau d'entiers initialisé

Autre syntaxe Il est également possible de déclarer, créer et initialiser un tableau tout à la fois. Sur l'exemple ci-dessus, l'instruction :

```
int [ ] T = {10; 8; -2; 5; 4}
```

déclare la variable `T`, alloue un espace pour cinq entiers et leur affecte les valeurs indiquées. On se retrouve alors dans la situation de la figure 4.2.

4.1.3 Taille et indexation d'un tableau

Toute variable `T` de type tableau est munie d'une primitive prédéfinie nommée `T.length` dont la valeur est le nombre d'éléments de `T`. Elle est donc nulle avant la création.

Tout tableau `T` est indexé de 0 à `T.length-1`. Ces deux valeurs sont les bornes du tableau. Si `i`$\notin \{0, \ldots,$ `T.length-1`$\}$, toute tentative d'accès `T[i]` à l'élément d'indice `i` de `T` provoque l'exception :

`ArrayIndexOutOfBoundsException: i`

Le parcours de gauche à droite d'un tableau `T` peut ainsi se faire selon le schéma suivant :

```
for (int i=0; i < T.length; i++)
    <traitement de T[i]>
```

4.1.4 Coût des accès

L'adresse des éléments d'un tableau n'est pas directement accessible comme celles des variables de type simple. L'adresse de l'élément `T[i]` résulte d'un calcul faisant intervenir l'adresse `T` du début du tableau et l'indice `i`. Plus précisément :

$$\textbf{adresse de T[i]} = T + i \times \textit{(taille du type des éléments de T)}$$

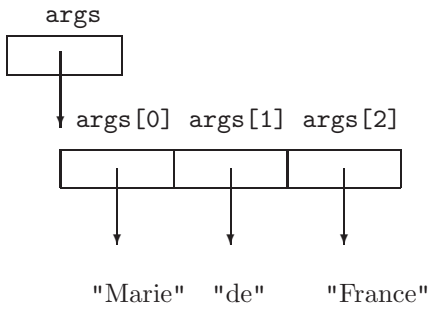
Ainsi, l'accès à un élément d'un tableau à une dimension requiert une addition et une multiplication et est donc plus coûteux que celui d'une variable simple.

4.1.5 Tableaux d'objets

Les éléments des tableaux de chaînes de caractères ou d'objets ne sont pas les chaînes ou les objets eux-mêmes, mais leurs adresses. Ainsi, après l'appel de programme (cf. section 2.1.2) :

```
java MonNom Marie de France
```

le tableau `args` est créé et initialisé comme suit :

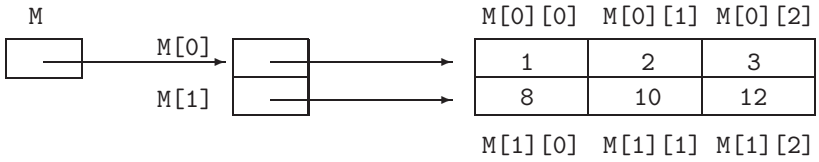


4.2 Tableaux à plusieurs dimensions

Les tableaux à `n` dimensions sont représentés comme des tableaux à une dimension de tableaux de dimension `n-1`. Par exemple, une matrice de format $n \times m$ sera codée par un tableau de n lignes, elles-mêmes codées comme des tableaux à m éléments. Ainsi l'enregistrement en mémoire de la matrice :

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 10 & 12 \end{bmatrix}$$

peut schématiquement être représenté comme suit :



4.2.1 Déclaration et création

Les tableaux à 2 dimensions se déclarent ainsi :

<type> [ ] [ ] <nom du tableau> ;

Plus généralement, les tableaux à `n` dimensions se déclarent avec `n` paires de crochets. Les créations et initialisations se font de manière analogue à celles des tableaux de dimension 1.

Exemple 4.3 *Déclaration :* `int[][] M;`
Création : `M = new int [2][3];`
Initialisation :
```
M[0][0] = 1; M[0][1] = 2; M[0][2] = 3;
M[1][0] = 8; M[1][1] = 10; M[2][2]= 12;
```
Création avec initialisation : `M = {{1,2,3},{8,10,12}};`

De plus, il est possible de définir des tableaux dont les lignes ont des longueurs variables. Au moment de la création, on indique le nombre de lignes. Puis chaque ligne est créée individuellement en fonction de sa longueur.

Exemple 4.4 *On se propose d'écrire une méthode qui crée et renvoie l'adresse d'un triangle de Pascal dont la dimension est passée en paramètre. Un triangle de Pascal est la matrice triangulaire inférieure P des coefficients binômiaux. Plus précisément : $\forall n, \forall p,\ t.q.\ 0 \leq p \leq n,\ P_{np} = C_n^p$. Les coefficients binômiaux se calculent à l'aide des relations :*

$$\begin{array}{l} C_n^n = C_n^0 = 1 \\ C_n^p = C_{n-1}^p \ + \ C_{n-1}^{p-1} \quad si\ 0 < p < n \end{array}$$

Voici un programme qui calcule et affiche un triangle de Pascal.

```
public class pascal{

   static int[][] Pascal (int n){

     int [][] P = new int[n+1][]; //création du tableau
                                  //des adresses des lignes

     for(int i=0; i<=n; i++) {  // calcul de la ligne i
       P[i] = new int[i+1];     // création de la ligne i
       P[i][0] = P[i][i] = 1;   // initialisation des extrémités
       for(int j=1; j<i;j++)    // calcul des autres coefficients
          P[i][j] = P[i-1][j] + P[i-1][j-1];
     }
     return P;
   }

   static void afficher(int[][]T){

        for(int i=0; i<T.length; i++){
            for(int j=0; j<T[i].length; j++)
               System.out.print( T[i][j]+"  ");
            System.out.println();
        }
    }

    public static void  main(String [ ] args){

        afficher(Pascal(Integer.parseInt(args[0])));
    }
}
```

Voici un exemple d'exécution :

```
$java pascal 5
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
```

4.2.2 Coût des accès

Le calcul de l'adresse de `M[i][j]` s'obtient en recherchant l'adresse `M[i]` de la ligne `i`, puis celle de son `j`ieme coefficient. On obtient ainsi :

adresse de M[i] = M + i × <taille des adresses>
adresse de M[i][j] = M[i] + j × <taille des éléments>

L'accès à un élément d'un tableau de dimension 2 requiert donc deux additions et deux multiplications. Il est bon en conséquence de limiter ces accès chaque fois que c'est possible.

Exemple 4.5 *Considérons deux matrices A de format $m \times n$ et B de format $n \times l$. Leur produit C a pour coefficients $C_{ij} = \Sigma_{k=0}^{n-1} A_{ik} \times B_{kj}$. Si l'on transcrit littéralement ces formules en un programme Java calculant le produit, on obtient :*

```
int m = A.length, l = B[0].length, n = B.length;

for(int i=0; i<m; i++)
  for(int j=0; j<l; j++){       // calcul de C[i][j]
    C[i][j] = 0;
    for(int k=0; k<n; k++)
      C[i][j] = C[i][j] + A[i][k]*B[k][j];
  }
```

Dans la boucle la plus interne, chacune des n itérations fait 2 accès à `C[i][j]` (nécessitant en tout $4n$ opérations). On remarque que l'écriture de l'affectation sous la forme :

```
C[i][j] += A[i][k]*B[k][j]
```

en supprime la moitié. Cependant, l'utilisation d'une variable auxiliaire comme ci-dessous permet de ne faire plus qu'un seul accès :

```
int m = A.length, l = B[0].length, n = B.length;
for(int i=0; i<m; i++)
  for(int j=0; j<l; j++){        // calcul de C[i][j]
    int aux = 0;
    for(int k=0; k<n; k++)
      aux += A[i][k]*B[k][j];
    C[i][j] = aux;
  }
```

Ceci illustre le fait qu'un programme ne s'écrit pas uniquement comme une formule mathématique pour laquelle seule importe la correction. Le codage d'une telle formule sous forme de programme induit des calculs en machine plus ou moins importants selon la forme que prend le codage. Même si le programme est valide, ces calculs peuvent nuire à son efficacité, voire provoquer une explosion combinatoire, ce qui signifie qu'il est en fait inutilisable.

4.3 Effet du codage sur l'occupation mémoire

Les programmes ne consomment pas uniquement du temps, mais également de l'espace. Ecrits sans précaution, ils peuvent rapidement saturer la mémoire. Le but de cette section est une étude de cas illustrant l'influence du codage sur l'occupation mémoire.

4.3.1 Puissance égyptienne

Considérons la méthode suivante, qui calcule la puissance $n$ième d'un flottant x en appliquant l'algorithme parfois nommé "puissance egyptienne" :

```
static float puiss(float x, int n){
   float p = 1;
   while(n!=0){
     if(n%2 ==1)
       {n--; p *= x;}
     else
       {n /= 2; x *= x;}
   return p;
}
```

Correction

Cette méthode renvoie la valeur de x^n. En effet, si on note x_0 et n_0 les valeurs initiales de x et n, on démontre que la relation $p \times x^n = x_0^{n_0}$ est un invariant de la boucle. En effet, p valant initialement 1, cette relation est vérifiée avant d'entrer dans la boucle. Elle est conservée par itération du fait que :

$p \times x^n = (p \times x) \times x^{n-1}$ si n est impair et
$p \times x^n = p \times (x \times x)^{n/2}$ si n est pair.

En conséquence, elle est vérifiée à la sortie quand n vaut 0 et donc, à ce moment là, $p = x_0^{n_0}$.

Complexité

Remarquons tout d'abord que les entiers n dont l'écriture binaire comporte $k+1$ chiffres en base deux sont ceux tels que $2^k \leq n < 2^{k+1}$. On en déduit successivement les relations :

$log_2(2^k) \leq log_2(n) < log_2(2^{k+1})$
$k \leq log_2(n) < k+1$
$k = \lfloor log_2(n) \rfloor$.

Supposons donc que l'écriture binaire de n soit constituée de $(k+1)$ symboles : chaque itération de la boucle `while` a pour effet, si n est impair, de transformer le chiffre 1 des unités en un 0, si n est pair, de supprimer le 0 final. Ainsi, le nombre d'itérations est minimal si $n = 2^k$: dans ce cas, son développement binaire est de la forme `10...0` (avec k symboles égaux à 0) et donc $k+1$ itérations.sont nécessaires avant d'obtenir 0. Le nombre d'itérations est maximal si le développement binaire est constitué de $k+1$ symboles égaux à 1 : dans ce cas il y a $2k+1$ itérations. Chaque itération ayant un coût borné, l'algorithme est logarithmique. Il est donc très supérieur à celui, linéaire, qui consiste à effectuer n multiplications de facteurs égaux à x. Cet exemple est repris en détail dans la section 5.4.

4.3.2 Puissance d'une matrice

L'idée est d'appliquer cette méthode au calcul de la puissance $n$ième d'une matrice carrée. L'algorithme s'appuie donc sur les formules :

$M^0 = I$
$M^n = (M \times M)^{n/2}$ si n est pair et >0
$M^n = M \times M^{n-1}$ si n est impair

On définit donc une classe `Puiss`, comportant un certain nombre de méthodes statiques destinées à traiter des matrices de flottants et à programmer cet algorithme. Cette classe possède notamment une méthode permettant d'afficher à l'écran une matrice donnée :

```
public class Puiss{

       public static void imprime(float[][]M){

        for(int i=0;i<M.length;i++){
            for(int j=0; j<M[i].length;j++)
                System.out.print(M[i][j] + " ");
            System.out.println();
        }
   }
          ...
```

Pour calculer la puissance d'une matrice, il est tout d'abord nécessaire d'écrire une méthode *mult* calculant le produit de deux matrices supposées de dimensions compatibles. La méthode crée une troisième matrice P de dimensions convenables pour recevoir le produit des deux matrices M et N passées en paramètre, en calcule les coefficients et en renvoie l'adresse.

```
       private static float[][] mult(float[][] M, float[][]N){

        int n = M.length-1, m = N.length-1, p=N[0].length-1;

        float[][] P = new float[n+1][p+1];
        for(int i=0;i<=n;i++)
            for(int k=0; k<=p; k++){
                int aux=0;
                for(int j=0; j<=m; j++)
                    aux+=M[i][j]*N[j][k];
                P[i][k]=aux;
            }
        return(P);
       }
```

La matrice unité est créée et renvoyée par la méthode suivante :

```
       private static float[][] unite(int dim){

        float[][] I = new float[dim][dim];

        for(int i=0; i<=dim-1; i++){
          for(int j=0; j<=dim-1; j++) I[i][j]=0;
          I[i][i]=1;
        }
        return I;
       }
```

On peut alors utiliser ces outils pour programmer l'algorithme de la puissance égyptienne :

```
       public static float[][] puiss_egypt(int n, float[][] M){

         float[][] P = unite(M.length);

         while (n>0)
```

```
    if (n%2==1){
     n--;
     P = mult(M, P);  // cree un espace memoire pour une nouvelle matrice
    }                 // un new a chaque nouvel appel de mult
    else {
      n=n/2;
      M = mult(M, M);  // cree un espace memoire pour une nouvelle matrice
    }                   // un new a chaque nouvel appel
   return(P);
}
```

Comme l'indiquent les commentaires, une nouvelle matrice de la taille de `M` est créée à chaque itération. L'espace occupé est de l'ordre de $2 \times dim^2 \times lg_2(n)$. Bien que les complexités logarithmiques soient de bonnes complexités, il n'en reste pas moins vrai que la taille de la mémoire occupée tend vers l'infini avec n et que le coefficient dim^2 peut être très gros.

On remarque que ce programme gaspille beaucoup de mémoire car les anciennes matrices `M` et `P` sont perdues à chaque itération mais occupent toujours de la place en mémoire. Ces espaces pourraient être réutilisés pour de nouveaux calculs quand les matrices qu'ils codent sont devenues inutiles. Pour ce faire, il faut pouvoir choisir l'espace dans lequel la méthode `mult` calcule son résultat : il suffit de l'indiquer en paramètre comme suit.

```
private static void mult(float[][] M, float[][] N, float[][] P){

  int n = M.length-1, m = N.length-1, p = N[0].length-1;

  for(int i=0;i<=n;i++)
    for(int k=0; k<=p; k++){
      int aux=0;
      for(int j = 0; j<=m; j++)
        aux+=M[i][j]*N[j][k];
      P[i][k]=aux;
  }
}
```

Une telle méthode calcule correctement le produit de `M` et de `N` dans le paramètre `P`. Ceci pourra paraître en contradiction avec la remarque de la section 2.2.6, où l'on a expliqué que les paramètres étant passés par valeur, la méthode travaille sur une copie locale de ces paramètres et donc ils ne peuvent en aucun cas être utilisés pour communiquer des résultats à l'environnement.

Toutefois, si la méthode `mult` crée (et travaille sur) sa propre copie de la variable `P`, cette dernière est une référence à un tableau (supposé précédemment créé). Ainsi, c'est une *adresse* qui est recopiée (et non la matrice elle-même) et donc le produit est bien calculé dans l'espace référencé. On pourrait alors être tenté de transposer la méthode `puiss_egypt` de la façon suivante :

```
public static float[][] puiss_egypt(int n, float[][] M){

  float[][] P = unite(M.length);

  while (n>0)
    if    (n%2==1) {n--; mult(M, P, P);}
    else           {n=n/2; mult(M, M, M);}
  return(P);
}
```

Mais ceci est totalement erroné puisque, par exemple, lors de l'exécution de `mult(M, M, M)` on écrit dans l'espace mémoire d'adresse `M` les coefficients du résultat, et ce faisant on écrase les anciennes valeurs pourtant nécessaires à la poursuite du calcul.

Ce problème se résoud en créant une matrice auxiliaire `Aux` dont l'adresse référencera, à chaque itération, un tableau dont la valeur est sans importance et peut donc être utilisée pour un nouveau calcul. Cette propriété de `Aux` est un invariant de la boucle. Ainsi, pour calculer le produit de `M` et de `M` dans `M`, on commencera par calculer le produit dans `Aux`, puis on échangera `Aux` et `M` : l'ancienne valeur de la matrice `M` se trouve maintenant à l'adresse `Aux` et peut être écrasée. Toutefois, cet échange ne consistera pas à faire trois recopies de matrices (très coûteuses en temps), mais à échanger simplement leurs adresses :

`{float[][] temp; temp = M; M = Aux; Aux = temp;}`.

On note aussi que la déclaration de la variable `temp` ne crée pas une matrice supplémentaire mais simplement une adresse qui sert à l'échange de deux adresses : peu de place, peu de temps.

Pour terminer, on note que dans la version coûteuse en mémoire, les valeurs successives de la matrice `M` (référencées dans une copie locale de l'adresse `M` passée en paramètre) sont créées dans de nouveaux espaces, et donc la matrice initiale dont l'adresse `M` est passée en paramètre à l'appel de la méthode est préservée. Ceci est indispensable : le calcul de la puissance nième de `M` ne doit pas affecter la matrice `M`. Dans la nouvelle version, ce n'est plus le cas. Le contenu des tableaux dont les adresses sont `M`, `Aux` et `P` évoluent à chaque itération : ainsi par exemple, l'appel `Puiss.puiss_egypt(A.length, A)` modifierait la matrice `A` ! On prend donc soin de travailler donc pas directement sur la matrice `M` mais sur une copie de celle-ci. On obtient ainsi :

```
private static float[][] copie (float[][]M){

 int dim = M.length;
 float[][] C = new float[dim][dim];

 for(int i=0; i<=dim-1; i++)
      for(int j=0; j<=M[0].length-1; j++)
        C[i][j] = M[i][j];
 return(C);
}

public static float[][] puiss_egypt(int n, float[][]M){

 int dim = M.length;
 float[][] P = unite(dim), M1 = copie(M), temp,
           Aux = new float[dim][dim];

 while (n>0)
   if (n%2==1){
     n--; mult(M1, P, Aux);
     temp = P; P = Aux; Aux = temp;
   }
   else {
     n=n/2; mult(M1, M1, Aux);
     temp = M1; M1 = Aux; Aux = temp;
   }
 return(P);
}
```

Deuxième partie

Eléments d'algorithmique

Chapitre 5

Principes généraux

5.1 Qu'est-ce qu'un algorithme ?

5.1.1 Définition informelle

Un cours d'algorithmique se devrait de débuter par la définition simple, claire et précise de la notion d'algorithme. Mais c'est là chose impossible. En effet, donner une réponse aux deux questions :

Qu'est-ce qu'un algorithme ?
Que peut-on calculer par des algorithmes ?

est l'objet d'une théorie profonde et difficile, située à la croisée des chemins mathématique, logique et informatique : il s'agit de la théorie de la *calculabilité*, qui est hors du propos de ce document.

Si ces deux questions fondamentales sont de façon évidente en amont de toute l'informatique, la notion d'algorithme est très antérieure à l'apparition des premiers calculateurs. Ethymologiquement, le mot *algorithme* provient du nom du mathématicien perse *al-Khowâ-Rismî* (IXième siècle), qui écrit un manuel d'arithmétique utilisant la numération arabe. Les algorithmes élémentaires de l'arithmétique sont connus de tous : il s'agit des règles opératoires pour le calcul d'une somme, du produit, du quotient, du plus grand diviseur commun (célèbre algorithme d'Euclide) etc. Dans la vie courante, recettes de cuisine, notices de montage, indications pour se rendre d'un point à un autre sont autant d'algorithmes. La variété des termes employés (recette, notice, marche à suivre, plan, instructions ...) reflète l'aspect assez flou du concept. On peut toutefois s'accorder sur le fait qu'un algorithme décrit un automatisme et qu'il doit pour cela satisfaire les conditions suivantes :

1. il opère sur une donnée pour produire un résultat
2. il possède un ensemble de définition (ensemble des données valides) bien déterminé
3. pour toute donnée valide, il calcule le résultat en un nombre fini d'étapes
4. pour une donnée non valide, il peut produire un résultat ou pas. Dans ce dernier cas, on dit qu'il boucle.
5. il produit le même résultat chaque fois qu'il opère sur la même donnée.
6. les données, les résultats et l'algorithme lui-même possèdent une description finie.

5.1.2 Les grandes questions de la *calculabilité*

Initialement, deux définitions formelles de la notion d'algorithme furent données indépendemment et simultanément par Church (Lambda-Calcul) et Turing (machines de Turing) dans les années 30. On peut considérer le Lambda-Calcul et les machines de Turing comme deux langages de programmation en ce sens qu'il s'agit de langages formels, répondant à une syntaxe précise et possédant une

sémantique bien spécifiée. Un algorithme est alors défini comme un *terme syntaxiquement correct* (intuitivement un programme syntaxiquement correct) du langage considéré. La différence essentielle entre ces langages et les véritables langages de programmation actuels (autrement dit entre les algorithmes appliqués par le mathématicien et les programmes susceptibles d'être exécutés par un ordinateur) est que les premiers ne font pas intervenir de limitation de mémoire. Antérieurs à l'apparition des premiers calculateurs, ils n'étaient pas exécutés automatiquement par une machine physique de capacité finie. La quantité de papier utilisée par le mathématicien pour dérouler un algorithme est virtuellement infinie. Si la description des données, des résultats et du déroulement des algorithmes se fait en écrivant un nombre fini symboles, ce nombre n'est pas majoré. Toutefois, ces langages abstraits servirent de paradigmes aux langages de programmation d'aujourd'hui : les langages de programmation fonctionnelle (Lisp, Scheme, ML, Caml, OCaml ...) sont directement inspirés du Lambda-Calcul et les langages impératifs (Pascal, C, C++, Java etc) des machines de Turing.

La question qui s'est alors immédiatement posée fut : *L'une de ces définitions est-elle plus générale que l'autre ?* Il fut prouvé (par Turing) qu'elles étaient équivalentes en ce sens que tout algorithme écrit dans l'un des deux langages peut être traduit dans l'autre. Qui plus est, cette traduction se fait elle-même par un algorithme du langage considéré.

La question suivante était alors : *Ces langages sont-ils suffisamment puissants pour exprimer tous les algorithmes, au sens intuitif du terme ?* La réponse à cette question ne peut être l'objet d'un théorème mathématique : en effet, comment démontrer rigoureusement l'équivalence entre une notion formelle d'algorithme et une notion intuitive ? L'équivalence entre ces deux concepts, l'un formel, l'autre intuitif, est ainsi appelée *Thèse de Church* ou *Thèse de Church-Turing*. Le fait que l'on ait pu démontrer que toutes les définitions formelles d'algorithmes introduites ultérieurement sont équivalentes aux deux premières, conforte cette *thèse* qui, par essence, ne peut être prouvée.

La notion d'algorithme étant alors rigoureusement définie et universellement admise, la troisième question était : *Toute fonction est-elle calculable par un algorithme ?* (on dit simplement *calculable*). La réponse est négative. En effet l'ensemble $\mathbb{N}^{\mathbb{N}}$ de toutes les fonctions sur les entiers naturels et à valeurs dans les entiers naturels, a pour cardinal la puissance du continu $\aleph_1$: il est équipotent à $\mathbb{R}$. Or on démontre (facilement en utilisant le fait que les algorithmes s'écrivent au moyen d'un alphabet fini) que l'ensemble des fonctions de $\mathbb{N}^{\mathbb{N}}$ qui sont calculables est dénombrable (équipotent à $\mathbb{N}$). Il y a donc *infiniment plus* de fonctions non calculables que de fonctions qui le sont.

Si les ouvrages d'algorithmique présentent une vaste collection de fonctions calculables, rencontrer une fonction qui ne l'est pas n'est pas si fréquent. Il se trouve que tout algorithme, quel que soit le problème qu'il est supposé résoudre, calcule une fonction de $\mathbb{N}^{\mathbb{N}}$. Une façon d'en prendre conscience est de remarquer que les données et le résultat d'un algorithme, quelle que soit leur apparente complexité, sont codés in fine dans un ordinateur par une suite binaire représentant un nombre entier. Un problème est dit *décidable* s'il possède une solution algorithmique (on dit aussi *effective*). Exhiber un problème non décidable revient donc à exhiber une fonction non calculable. C'est l'objet de la section suivante.

5.1.3 Le problème de l'arrêt

Le problème de l'arrêt est un problème non décidable et il est présenté ici de façon informelle.

Un programme est écrit dans un langage de programmation (code source). Ce programme source est enregistré en mémoire (donc sous forme d'un entier binaire). Puis il est en général transformé par un compilateur en un programme exécutable, également enregistré sous forme d'un entier binaire. Un compilateur est donc un programme (codant lui-même un algorithme), qui prend en entrée un programme source, qui l'analyse syntaxiquement, et qui en l'absence d'erreurs produit en résultat un exécutable.

On pourrait légitimement envisager d'améliorer les compilateurs en affinant leur analyse de telle sorte qu'ils détectent non seulement les erreurs de syntaxe, mais également d'éventuelles erreurs dans l'exécution des programmes syntaxiquement corrects. Ils pourraient en particulier signaler si l'exécution d'un programme `P` sur une entrée `n` s'arrête au bout d'un temps fini ou boucle : c'est ce que l'on appelle le *problème de l'arrêt*. On peut imaginer examiner toutes les boucles `while` de `P` pour vérifier que l'expression de contrôle prend la valeur `false` au bout d'un nombre fini d'itérations, en examinant comment évoluent, à chaque itération, les variables qu'elle fait intervenir. Un résultat important de la théorie de la calculabilité montre que ceci est impossible : on prouve en effet que le problème de l'arrêt est indécidable. Voici une esquisse du raisonnement.
Considérons les programmes source Java. Chacun d'eux est représenté, lorsqu'il est enregistré en mémoire, par un entier binaire `n`. On note $\mathtt{P}_n$ ce programme. Toutefois, le développement binaire d'un entier `n` quelconque ne code pas nécessairement un programme Java correct (c'est le cas par exemple de l'entier codant dans la machine la source LaTeX du présent document). On convient dans ce cas, de noter $\mathtt{P}_n$ le programme suivant (qui boucle indéfiniment) :

```
public class loop{
  public static void main(String[] args){
    while(true);
  }
}
```

La suite infinie (avec répétitions) de tous les programmes Java

$$\mathtt{P}_0, \mathtt{P}_1, \ldots, \mathtt{P}_n, \ldots$$

est donc parfaitement définie : tout programme Java dont le code en machine est l'entier binaire `n` se trouve au `n`ième rang dans la liste. Quant au programme `loop`, il se trouve à tous les rangs qui ne représentent pas un programme Java syntaxiquement correct.

On considère alors le tableau infini à deux dimensions suivant :

Argument → / Programme ↓	0	1	2	...	n	...
$\mathtt{P}_0$	1	0	1	...	1	...
$\mathtt{P}_1$	0	0	1	...	1	...
$\mathtt{P}_2$	1	0	1	...	1	...
...	...	...	...	...	...	...
$\mathtt{P}_n$	1	1	0	...	1	...
...	...	...	...	...	...	...

Le coefficient à l'intersection de la ligne $\mathtt{P}_n$ et de la colonne `i` indique, selon qu'il vaut 0 ou 1, que le programme $\mathtt{P}_n$ boucle ou pas sur la donnée `i`.

On utilise pour prouver le résultat, un argument de *diagonalisation* amenant à un *paradoxe*. Supposons qu'il existe une méthode Java `Arret` qui étant donné un entier `n` renvoie 0 si l'exécution de $\mathtt{P}_n$ boucle sur l'entrée `n` et 1 sinon. La méthode `Arret` est donc supposée calculer l'élément *arrêt* de $\mathbb{N}^{\mathbb{N}}$ défini par :

arrêt(n) = 0 si $\mathtt{P}_n$ boucle sur l'entrée `n`
arrêt(n) = 1 si $\mathtt{P}_n$ se termine sur l'entrée `n`

On se propose d'écrire un programme `P` différent de tous les programmes énumérés dans le tableau (*paradoxe* puisque le tableau est censé les contenir tous), en ce sens que $\forall n \in \mathbb{N}, \mathtt{P(n)} \neq \mathtt{P_n(n)}$

(nième élément de la *diagonale*). Cette condition est réalisée par le programme `P` ci-après :

```
public class P{
  public static void main(String[] args){
    int n = Integer.parseInt(args[0]);
    while(Arret(n) == 1);
    System.out.println(n);
  }
}
```

Pour tout entier `n`, si $\texttt{P}_n$ boucle sur la donnée `n`, `Arret(n)` renvoie 0 et donc `P` s'arrête sur la donnée `n` et renvoie `n`. Si $\texttt{P}_n$ s'arrête sur la donnée `n`, `Arret(n)` revoie 1 et donc `P` boucle sur la donnée `n`. Dans les deux cas, on peut affirmer que $\texttt{P} \neq \texttt{P}_n$ puisque ces deux programmes se comportent différemment sur la donnée `n`, et ce pour tout entier `n`. On en déduit que le programme `P` ne figure pas dans la liste $\texttt{P}_0, \texttt{P}_1, \ldots, \texttt{P}_n, \ldots$ de tous les programmes Java, ce qui est absurde.

La démonstration rigoureuse est analogue, si ce n'est qu'elle fait intervenir des algorithmes (termes du Lambda-Calcul ou machines de Turing) et non des programmes.

On peut ainsi conclure que le problème de l'arrêt est indécidable ou, de façon équivalente, que la fonction *arrêt* de $\mathbb{N}^{\mathbb{N}}$ n'est pas calculable.

5.1.4 Pour conclure

Avant de s'engager dans l'algorithmique et la programmation, il convient d'avoir clairement à l'esprit les questions-réponses suivantes :

Tout problème possède-t-il une solution effective ? La réponse est NON

L'exemple du problème de l'arrêt montre que l'on ne peut prétendre tout résoudre algorithmiquement, et donc les ordinateurs sont impuissants à produire la solution de quantité de problèmes.

Dans le cas favorable où l'on connaît une solution effective (i.e. algorithmique) d'un problème, est-t-on capable d'écrire un programme susceptible de s'exécuter sur un ordinateur pour produire un résultat ? Autrement dit :

Tout algorithme est-il programmable ? La réponse est : OUI

Les langages de programmation actuels, et Java en particulier, sont complets : ils permettent d'exprimer tout algorithme.

Tout programme correct est-il utilisable ? La réponse est : NON.

Nous verrons que certains programmes sont si coûteux en temps d'exécution, que même les ordinateurs les plus rapides ne peuvent obtenir le résultat dans des délais raisonnables. C'est le cas par exemple quand le nombre d'opérations à effectuer est exponentiel en la taille de la donnée. On dit alors qu'il se produit une *explosion combinatoire*. D'autres requièrent tant d'espace mémoire qu'ils sont également inutilisables.

Le processus de résolution informatique d'un problème consiste donc à :

1. Examiner s'il est réaliste de chercher une solution *effective* à un problème donné (ne pas se lancer inconsidérément dans la recherche d'une solution algorithmique d'un problème comme celui de l'arrêt).
2. Si oui, concevoir un algorithme.

3. Démontrer que l'algorithme est correct.
4. Etudier sa complexité en temps et en espace (se convaincre qu'il est utilisable)
5. *Last but not necessarily least*, programmer l'algorithme.

Les sections suivantes sont dédiées à ces divers points.

5.2 Conception et expression d'algorithmes

Cette section introduit sur un exemple les méthodes de conception et de preuve d'algorithme.

5.2.1 Spécification du problème

Il convient de décrire avec précision le problème à résoudre. Cette première phase n'est pas nécessairement triviale : le problème peut être complexe et n'est pas en général posé par la personne qui doit concevoir la solution. On s'intéresse dans cette section au problème suivant :

Etant donné un texte, calculer le nombre d'apparitions des mots du texte.

Cette formulation doit tout d'abord être affinée : on peut convenir qu'il s'agit de construire le lexique de tous les mots apparaissant dans le texte, accompagnés du nombre de leurs apparitions. On souhaitera également en général que les mots du lexique soient rangés par ordre alphabétique. La recherche d'une solution peut alors débuter.

5.2.2 Analyse descendante - Modularité

Procéder à une analyse descendante du problème consiste à l'appréhender dans sa globalité dans un premier temps, sans se préoccuper (et se perdre) dans les détails : on commence par en chercher une solution en supposant déjà résolus un certain nombre de problèmes plus simples. Chacun de ces sous-problèmes sera à son tour examiné selon la même approche pour être décomposé en nouvelles tâches, et ce jusqu'à ne plus obtenir que des problèmes dont la solution tient en quelques lignes. A chacune des étapes du processus, on est également amené à introduire de plus en plus de précision dans l'expression.

Dans le cas auquel on s'intéresse, débuter l'étude en se posant les questions :

Comment est stocké le texte ?
Comment les mots sont-ils délimités ?
Comment saisir un mot ?
Que fait-on des signes de ponctuation ?
Comment faire pour détecter que le texte est fini ?

est à proscrire puisqu'à l'opposé d'une démarche descendante. La façon correcte de traiter ce problème serait plutôt d'ébaucher une première solution de la façon suivante :

Calculer le nombre des apparitions des mots d'un texte c'est :
- saisir le mot courant
- le rechercher dans le lexique
- s'il y est déjà, incrémenter le nombre de ses apparitions
sinon
** entrer le mot dans le lexique*
** initialiser à 1 le nombre de ses apparitions*
- recommencer jusqu'à la fin du texte

Après cette première approche assez grossière, on passe à un niveau plus concret et plus précis par

une expression en français structuré (ou pseudo-code) faisant intervenir des boucles, des affectations ... , sur le modèle des langages de programmation.

```
nombre-des-apparitions

tant que le texte n'est pas fini faire
   - saisir le mot courant
   - rechercher le mot dans le lexique
   - si il y est
         incrémenter le nombre de ses apparitions
     sinon
         * entrer le mot dans le lexique
         * initialiser à 1 le nombre de ses apparitions
```

On peut alors introduire de la modularité, par décomposition en sous-problèmes comme indiqué précédemment. Le parcours de la totalité du texte en vue de construire le lexique répond au schéma suivant, qui exprime de façon concise le fonctionnement de l'algorithme au plus haut niveau :

```
nombre-des-apparitions

tant que le texte n'est pas fini faire
   - saisir le mot courant
   - traiter le mot courant
```

On a ainsi dégagé le mécanisme, simple et clair, de l'analyse du texte, sans se préoccuper de détails. Il convient alors de préciser ce que l'on entend par *traiter le mot courant*.

```
traiter un mot

   - rechercher le mot dans le lexique
   - si il y est
         incrémenter le nombre de ses apparitions
     sinon
         entrer le mot dans le lexique
```

L'ajout d'un nouveau mot dans le lexique demande à être précisé soigneusement :

```
entrer un mot dans le lexique

   - insérer le mot en préservant l'ordre alphabétique
   - initialiser à 1 le nombre de ses apparitions
```

Se posent alors les problèmes de la *recherche* et de l'*insertion* d'un mot dans un lexique supposé ordonné par l'ordre alphabétique. Les solutions sont diverses et dépendent essentiellement de la structure de donnée choisie pour représenter le lexique. L'étude des structures de données et des problèmes de recherche/insertion dans un ensemble trié sont des grands classiques de l'informatique et font l'objet de chapitres entiers dans les ouvrages d'algorithmique. C'est seulement lorsque toutes ces questions auront été résolues, que l'on abordera le problème plus technique, mais sans grande difficulté théorique, de la saisie des mots.

5.2.3 Correction de l'algorithme

Correction partielle et correction totale On dit qu'un algorithme est *partiellement correct* lorsque l'on a prouvé que s'il se termine, alors il produit le résultat attendu. On dit qu'il est *correct*, lorsqu'il est partiellement correct et que l'on a de plus établi sa terminaison. De façon synthétique :

correction = correction partielle + terminaison

Modularité Comme pour la conception, la preuve de correction d'un algorithme se fait de façon descendante et modulaire : on suppose dans un premier temps que les divers modules sont corrects pour démontrer la correction de l'algorithme global. Puis l'on prouve la correction des modules.

Invariant de boucle La correction totale d'un algorithme itératif se fait au moyen d'un *invariant de boucle*[1]. Un invariant de boucle est un prédicat (propriété) sur les variables intervenant dans le boucle qui est :
- satisfait avant l'entrée dans la boucle
- conservé par itération.

On en déduit immédiatement qu'un invariant est satisfait à la sortie de la boucle. Dans l'exemple considéré on peut choisir l'invariant I comme suit :

I(lexique) $\Leftrightarrow$ *le lexique contient exactement l'ensemble, ordonné par l'ordre alphabétique, des mots rencontrés depuis le début du texte jusqu'à l'instant présent, accompagnés du nombre de leurs apparitions*

Il faut donc prouver que l'invariant est vrai avant l'entrée dans la boucle de l'algorithme `nombre-des-apparitions`. Or, à ce moment là, aucun mot n'a encore été saisi, donc le lexique doit être vide. On s'aperçoit à cet instant que ce fait n'est nullement précisé : si le lexique n'est pas vide au moment où débute l'algorithme, il n'y a aucune chance qu'il contienne à la fin le résultat attendu. Le lexique n'a pas été *initialisé*. On vient ainsi de détecter une faute (volontairement !) commise pour illustrer le propos. On est donc amené à modifier l'algorithme comme suit :

`nombre-des-apparitions`

```
Initialiser le lexique à vide
tant que le texte n'est pas fini faire
   - saisir le mot courant
   - traiter le mot courant
```

La satisfaction de la propriété d'invariance est assurée avant l'entrée dans le boucle par une initialisation correcte des variables.

Le fait que l'invariant est conservé par itération est facile à établir. On suppose pour l'instant que tous les autres modules sont corrects. On suppose de plus (hypothèse H) que l'invariant est satisfait avant une itération. Celle-ci provoque la lecture du mot suivant et son traitement. Ce dernier consistant à incrémenter le nombre `n` des apparitions du mot s'il est dans le lexique (donc s'il a déjà été rencontré `n` fois d'après l'hypothèse H) ou sinon à le rajouter à la bonne place (grâce au module `insérer`) tout en initialisant `n` à 1, l'invariant est à nouveau satisfait par la nouvelle valeur du lexique après l'itération.

A la sortie de la boucle `tant que`, la condition de contrôle prend la valeur `false`. Le texte est donc terminé. Mais comme l'invariant est vrai à ce moment là, on en déduit que *le lexique contient exactement l'ensemble, ordonné par l'ordre alphabétique, des mots du texte, accompagnés du nombre de leurs apparitions.*

La correction partielle s'obtient en combinant invariant et condition de sortie.

1. On pourra également se référer aux deux exemples du calcul du PGCD dans la section 1.4.3 et de la puissance égyptienne dans la section 4.3.1, qui contribuent à illustrer ce paragraphe

Enfin, la preuve de terminaison repose sur la correction de la saisie d'un mot nouveau et celle de la détection de la fin du fichier. Si le texte est composé d'un nombre fini de mots (l'algorithme ne s'applique pas à l'ensemble, virtuellement infini, des mots de la toile !), le nombre d'itérations est égal au nombre de mots du texte, donc il est fini. CQFD.

5.3 Algorithmes récursifs

La section précédente a illustré comment décomposer un problème en problèmes plus simples. Une autre approche, appelée *diviser pour régner* (en anglais *divide and conquer*), consiste à résoudre le problème sur une donnée en supposant le *même* problème déjà résolu sur des données plus *simples*. Un algorithme A qui, pour calculer le résultat $A(x)$ sur une donnée x, utilise des résultats $A(y)$ supposés connus sur des données y de *taille* inférieure à x, est appelé algorithme *récursif*. Par exemple, si l'on suppose que l'on connait $(n-1)!$, alors la formule $n! = n \times (n-1)!$ permet de calculer $n!$. De même, si l'on suppose que l'on sait trier un tableau à $\frac{n}{2}$ éléments, on peut trier un tableau de n éléments en triant tout d'abord ses deux moitiés, et en *fusionnant* ensuite ces deux parties qui sont alors ordonnées. La suite de la section précise cette présentation informelle.

5.3.1 Introduction sur des exemples

Une fonction f est définie récursivement quand sa définition fait intervenir f elle-même.

Exemple 5.1 (La factorielle)
fact(0) = 1
fact(n) = n × fact(n-1) $\quad \forall n \geq 1$

Exemple 5.2 (La dérivée $n$ième)
$f^{(0)} = f$
$f^{(n)} = (f')^{(n-1)} \quad \forall n \geq 1$

Exemple 5.3 (La puissance égyptienne) $\forall x \in \mathbb{R}$
$x^0 = 1$
$x^n = (x^2)^{n/2} \quad \forall n > 0$, *n pair*
$x^n = x \times x^{n-1} \quad \forall n > 0$, *n impair.*

Exemple 5.4 (L'algorithme d'Euclide) $\forall a \in \mathbb{N}, \forall b \in \mathbb{N}^*$
$\Delta(a, b) = b$ *si b divise a,*
$\Delta(a, b) = \Delta(b, a \bmod b)$ *sinon.*

Exemple 5.5 (Le PGCD : autre méthode) $\forall a \in \mathbb{N}, \forall b \in \mathbb{N}^*$
$\delta(0, b) = b$
$\delta(a, b) = 2 \times \delta(a/2, b/2)$ *si a et b sont pairs et* $a > 0$
$\delta(a, b) = \delta(a/2, b)$ *si a est pair et b impair et* $a > 0$
$\delta(a, b) = \delta(a, b/2)$ *si a est impair et b pair*
$\delta(a, b) = \delta(a - b, b)$ *si a et b impairs et* $a \geq b$
$\delta(a, b) = \delta(a, b - a)$ *si a et b impairs et* $a < b$.

Etude détaillée de l'exemple 5.1

L'algorithme induit La définition récursive de la fonction *fact* exprime un algorithme récursif. Elle permet en effet à un automate de calculer *fact(n)* pour tout entier n. Si $n = 6$, un tel automate procèderait par étapes comme décrit par la figure 5.1. L'algorithme peut être programmé en Java en traduisant directement les deux conditions qui constituent la définition.

$$
\begin{aligned}
\text{fact}(6) &= 6 \times \mathbf{fact(5)} && (1)\\
&= 6 \times 5 \times \mathbf{fact(4} && (2)\\
&= 6 \times 5 \times 4 \times \mathbf{fact(3)} && (3)\\
&= 6 \times 5 \times 4 \times 3 \times \underline{\mathbf{fact(2)}} && (4)\\
&= 6 \times 5 \times 4 \times 3 \times \underline{2 \times \mathbf{fact(1)}} && (5)\\
&= 6 \times 5 \times 4 \times 3 \times \underline{2 \times 1 \times \mathbf{fact(0)}} && (6)\\
&= 6 \times 5 \times 4 \times 3 \times \underline{2 \times 1 \times \mathbf{1}} && (7)\\
&= 6 \times 5 \times 4 \times 3 \times \underline{2 \times \mathbf{1}} && (8)\\
&= 6 \times 5 \times 4 \times 3 \times \underline{\mathbf{2}} && (9)\\
&= 6 \times 5 \times 4 \times \mathbf{6} && (10)\\
&= 6 \times 5 \times \mathbf{24} && (11)\\
&= 6 \times \mathbf{120} && (12)\\
&= \mathbf{720} && (13)
\end{aligned}
$$

FIGURE 5.1 – Déroulement de l'algorithme `fact` pour n=6

```
static int fact (int n) {
  if (n==0)
    return 1;
  else
    return(n*fact(n-1));
}
```

ou encore

```
static int fact (int n) {
  if (n==0) return 1;
  return(n*fact(n-1));
}
```

Cette méthode est récursive puisque si n est strictement positif, elle se rappelle elle-même sur l'argument $(n-1)$. Le déroulement par étapes de l'algorithme détaillé figure 5.1 pour la valeur $n = 6$, suit très exactement l'exécution de la méthode. Chacune des six premières lignes comporte en gras un appel récursif. Les multiplications sont mises en attente tant que l'exécution de cet appel récursif n'est pas terminée. Par exemple la ligne (4) comporte l'appel `fact(2)`, dont les étapes de l'exécution sont soulignées et s'achèvent à la ligne (9).

Pile de récursivité La longueur des six premières lignes croît, tout comme l'espace occupé en mémoire, où la nouvelle valeur mise en attente est *empilée sur* celles qui le sont déjà. La notion de *pile* sera introduite en détail par la suite. Informellement, il s'agit d'une suite d'éléments gérée comme une pile au sens courant du terme (et mentalement représentée de façon verticale) : on rajoute un élément en l'empilant à une extrémité de la liste, on dépile en supprimant le dernier élément empilé. Sur la figure 5.1, la pile est représentée horizontalement, son sommet en étant l'extrémité droite. Cette pile, constituée des valeurs mémorisées en attente de la fin des appels récursifs dont le résultat est nécessaire à la poursuite des calculs, est appelée *pile de récursivité*. Dans l'exemple considéré, sa taille maximale est n fois la taille d'un entier, si n est la donnée. Cet espace mémoire, contrairement à ceux rencontrés jusqu'à présent et qui étaient explicitement alloués par une déclaration (types simples) ou une instruction `new` (objets et tableaux), est créé dynamiquement et automatiquement du fait des appels récursifs. Lors de l'évaluation de l'espace mémoire nécessaire à l'exécution d'un algorithme récursif, il convient donc de bien prendre en compte cette pile de récursivité dont la création et la gestion échappe au programmeur.

Cas terminal et terminaison Le dernier appel récursif (ligne (6)) aboutit au *cas terminal* (n=0) de la définition, pour lequel le calcul du résultat (qui vaut 1) ne requiert pas de nouvel appel à la fonction : c'est ce qui assure la terminaison de l'algorithme. Les sommets de pile sont alors dépilés un à un pour être multipliés à la valeur courante (en gras), et ce tant que la pile n'est pas vide.

Après cette présentation informelle, les deux sections qui suivent traitent respectivement de la terminaison et de la correction des algorithmes récursifs.

5.3.2 Terminaison des algorithmes récursifs

Un algorithme récursif peut boucler sur une donnée quand le calcul de celle-ci provoque une infinité d'appels récursifs, sans que jamais un cas terminal ne soit atteint. L'étude de la terminaison d'un algorithme récursif consiste à trouver l'ensemble des données pour lesquelles l'ensemble des appels récursif est fini.

Etude de deux exemples

La factorielle La terminaison de l'algorithme se prouve en remarquant que pour tout entier strictement positif n, $fact(n)$ appelle récursivement $fact(n-1)$. Les appels se font donc successivement sur les arguments :

$$n,\, n-1,\, n-2,\, n-3,\ \ldots$$

Cette suite atteint donc l'argument 0 après un nombre fini d'étapes (n exactement). Or le cas $n = 0$ est un cas terminal pour lequel le résultat est immédiat. Donc l'algorithme se termine pout tout entier $n \geq 0$.

L'algorithme d'Euclide Pour tout couple $(a,\, b)$ de l'ensemble de définition, si b n'est pas un diviseur de a, $\Delta(a,\, b)$ appelle récursivement $\Delta(b,\, r_0)$ où r_0 est le reste de la division de a par b. Les appels se font donc successivement sur :

$$(a,\, b),\, (b,\, r_0),\, (r_0,\, r_1),\, (r_1,\, r_2),\, (r_2,\, r_3),\ldots,\, (r_i,\, r_{i+1})\ \ldots$$

où, pour tout indice $i \geq 0$, r_{i+1} est le reste de la division euclidienne de r_{i-1} par r_i (avec la convention : $r_{-1} = b$). Par suite, le reste étant strictement inférieur au diviseur,

$$\forall i \geq -1,\ \ r_i > r_{i+1}.$$

L'algorithme calcule donc, à chaque appel récursif, un nouvel élément de la suite

$$b = r_{-1},\, r_0,\, r_1,\, r_2,\, \ldots,\, r_i,\, \ldots$$

qui est une suite strictement décroissante d'entiers positifs : elle est donc finie. Par suite, le nombre d'appels récursifs est fini, ce qui prouve que l'algorithme se termine.

Induction bien fondée

Plus généralement, pour prouver qu'un algorithme récursif se termine sur toute donnée $x_0 \in D$, il faut montrer que les suites des arguments

$$x_0,\, x_1,\, \ldots,\, x_i,\, x_{i+1},\, \ldots$$

des appels récursifs successifs sont finies. Une façon de procéder consiste à trouver une relation $>$ telle que

$$x_0 > x_1 > \cdots > x_i > x_{i+1} > \ldots$$

et pour laquelle on prouve qu'une telle suite est nessaicérement finie. On en déduit alors que la suite des appels récursifs est finie, ce dont découle la terminaison de l'algorithme.

Définition 5.1 (Relations bien fondées) *Soit* E *un ensemble et une relation binaire sur E notée $>$. Toute suite (x_n) d'éléments de* E *telle que :*

$$x_0 > x_1 > \cdots > x_i > x_{i+1} > \ldots$$

est appelée chaîne descendante pour la relation $>$. La relation $>$ est dite bien fondée *si toute chaîne descendante d'éléments de D est finie.*

Théorème 5.2 *Soit* D *un ensemble muni d'une relation bien fondée $>$ et un prédicat* P *sur* D. *On démontre le principe d'induction bien fondée suivant.*

$$\frac{\forall x \in D \quad (\forall y \in D,\ y < x \Rightarrow P(y)) \Rightarrow P(x)}{\forall x \in D \quad P(x)} \tag{5.1}$$

Autrement dit, pour démontrer que tout élément de D satisfait P, il suffit de prouver que tout x de D satisfait P, sous l'hypothèse que tout élément y inférieur à x satisfait P.

Ce principe d'induction est présenté dans la formule 5.1 sous forme de *séquent* : il s'agit d'une notation fractionnaire, le numérateur représentant l'hypothèse et le dénominateur la conclusion. La règle 5.1 signifie donc que pour prouver $\forall x \in D\ P(x)$, il suffit, pour tout x de D, de démontrer $P(x)$ sous l'hypothèse dite *d'induction* :

$$\forall y \in D,\ y < x \Rightarrow P(y)$$

Preuve Le théorème se démontre par l'absurde. Supposons que

$$\forall x \in D \quad (\forall y \in D,\ y < x \Rightarrow P(y)) \Rightarrow P(x) \qquad (H)$$

et qu'il existe un élément x_0 qui ne satisfait pas P. D'après l'hypothèse (H), on déduit que la proposition

$$\forall y \in D,\ y < x_0 \Rightarrow P(y)$$

est fausse et donc qu'il existe un élément $x_1 < x_0$ qui ne satisfait pas P. A partir de x_1 et avec les mêmes arguments, on déduit l'existence d'un élément $x_2 < x_1$ qui ne satisfait pas P. On peut ainsi de proche en proche, prouver l'existence d'une infinité d'éléments ne satisfaisant pas P et tels que :

$$x_0 > x_1 > \ldots > x_k > \ldots$$

Ceci est contraire au fait que la relation $>$ est bien fondée. □

Théorème 5.3 *Soit D un ensemble muni d'une relation bien fondée $>$. Soit A un algorithme récursif défini sur D. Si, pour toute donnée x de D, $A(x)$ appelle récursivement $A(y)$ sur des données y telles que $y < x$, alors l'algorithme se termine.*

Preuve On procède par induction bien fondée (théorème 5.2).
Soit $x \in D$. Supposons (hypothèse d'induction) que l'agorithme se termine sur toute donnée $y < x$. Alors il se termine sur x puisque les rappels récursifs ont lieu par hypothèse sur des arguments $y < x$.
Conclusion : l'algorithme se termine pour tout x de D. □

Corollaire 5.4 *Soit* D *un ensemble muni d'une fonction :*

$$\text{taille} : \mathrm{D} \to \mathrm{N}$$

Soit A *un algorithme récursif sur* D. *Si tout appel récursif se produit sur un argument de taille strictement inférieure à l'argument de départ, alors* A *se termine.*

En effet, la relation définie par

$$x > y \Leftrightarrow \mathit{taille}(x) > \mathit{taille}(y)$$

est bien fondée, ce qui permet de prouver la terminaison de l'algorithme.

Pour l'exemple de la factorielle, *taille* est l'identité et pour l'algorithme d'Euclide, il suffit de définir $\mathit{taille}(a, b) = b$.
L'existence d'une telle fonction *taille* est une condition suffisante, mais non nécessaire à la terminaison. La preuve de terminaison de la fonction d'*Ackermann* (cf. section 6.2.2) s'appuie sur l'existence d'une relation bien fondée qui ne peut être définie à partir d'une taille.

Il faut savoir que prouver la terminaison d'un algorithme récursif peut s'avérer extrêmement difficile. La terminaison de l'algorithme *syracuse* (exercice 5.3) est à ce jour un problème ouvert.

Exercice 5.1 *Démontrer la terminaison des algorithmes des exemples 5.3 et 5.5.*

5.3.3 Correction des algorithme récursifs

Comme pour les algorithmes itératifs, on peut synthétiser la définition de la correction d'un algorithme récursif par la formule :

correction = correction partielle + terminaison

La correction partielle consiste à démontrer que si l'algorithme se termine sur une donnée, alors il produit le résultat attendu. *La correction partielle d'un algorithme récursif s'établit par induction bien fondée.* En présence d'une fonction *taille* sur l'ensemble des données, à valeurs dans l'ensemble des entiers naturels, il suffit de démontrer que le résultat produit pour une donnée x est correct en supposant qu'il l'est pour toute donnée y de taille strictement inférieure à x. Dans les cas les plus simples, une récurrence sur la taille des données suffit.

Cas de la factorielle Etablir la correction de la méthode Java `fact`, autrement dit de l'algorithme induit par la définition de l'exemple 5.1, consiste à prouver que pour tout n, `fact(n)`=$n!$, où $n!$ est défini par :

$$0! \;=\; 1 \quad \text{et} \quad \forall n > 0,\; n! \;=\; \prod_{i=0}^{n} i. \qquad (Def)$$

Ceci est immédiat pour $n = 0$. Soit $n \in I\!N$ quelconque. Supposons que `fact(n)`=$n!$. Alors :

`fact(n+1)`	$= n \times$`fact(n)`	*par définition de l'algorithme* `fact`
	$= n \times n!$	*d'après l'hypothèse de récurrence*
	$= (n+1)!$	*d'après la définition de* (Def) *de la factorielle.*

Cas de l'algorithme d'Euclide La définition de la fonction Δ de l'exemple 5.4 induit la méthode Java :

```
public static int delta(int a, int b){
   int r = a%b;
   if(r == 0) return b;
   return (delta(b, r));
}
```

On montre que :
- PGCD$(a,\, b) = b$ si b divise a.
- PGCD$(a,\, b)$ = PGCD$(b,\, a \; mod \; b)$ sinon.

La première condition est évidente. La seconde s'obtient en remarquant que si r et q sont respectivement le reste et le quotient de la division euclidienne de a par b, du fait de la relation $a = bq + r$, les couples $(a,\, b)$ et $(b,\, r)$ ont même ensemble de diviseurs communs.

On peut conclure en remarquant que la méthode `delta` définissant une unique fonction sur $I\!N \times I\!N^*$, d'après la preuve de terminaison, toute fonction f définie sur $I\!N \times I\!N^*$ et qui satisfait les relations définissant l'algorithme est exactement la fonction calculée par la méthode. Et c'est le cas de la fonction PGCD. Ce type de démonstration s'applique également à la factorielle

Une autre démonstration, plus "procédurière", peut se faire par induction. Soit $(a,\, b) \in I\!N \times I\!N^*$, quelconque, fixé. Supposons que pour tout couple $(a',\, b') \in I\!N \times I\!N^*$, tel que $b' < b$, le résultat soit correct, c'est-à-dire : `delta(a',b')` = PGCD$(a',\, b')$. Montrons que `delta(a,b)` = PGCD$(a,\, b)$.
Le résultat est immédiat si b divise a.
Sinon, par définition de la méthode `delta`, on sait que `delta(a,b) = delta(b, a mod b)`, et d'après l'hypothèse d'induction `delta(b, a mod b)` = PGCD$(b,\, a \; mod \; b)$, enfin on a montré que PGCD$(b,\, a \; mod \; b)$= PGCD$(a,\, b)$. De ces trois égalités, on déduit le résultat.

Exercice 5.2 *Prouver la correction partielle des algorithmes des exemples 5.3 et 5.5.*

Exercice 5.3 *Prouver que l'algorithme défini sur $\mathbb{N}^*$ par :*
- *syracuse(1) = 1*
- *syracuse(n) = syracuse(n/2) si n est pair*
- *syracuse(n) = syracuse(3*n+1) si n est impair*

calcule, s'il se termine sur toute donnée $n \geq 1$, la fonction constante et égale à 1 sur $\mathbb{N}^*$.

Récursif ou itératif ?

On démontre que tout algorithme récursif possède une version itérative. Voici, par exemple, une version itérative de la méthode `fact` :

```
static int fact (int n) {
    int r = 1;
    for(int i=1; i<=n; i++) r *= i;
    return r;
}
```

Tout comme la méthode récursive, cette méthode effectue n multiplications. De même, l'exemple 5.4 est une version récursive de l'algorithme d'Euclide. Une version itérative de cet algorithme a été donnée dans l'exemple 1.12 de la section 1.4.3 et reprise dans la section 2.2.1. Il s'agit du même algorithme, exprimé différemment : les divisions euclidiennes successives effectuées par les deux formes de l'algorithme sont identiques.

Toutefois l'espace occupé dans l'un et l'autre cas diffère. Pour la version itérative du calcul de la factorielle, trois entiers sont nécessaires (`n`, `i` et `r`) quelle que soit la valeur de la donnée n : l'espace occupé est constant. La version récursive met en place une pile de récursivité de taille proportionnelle à n. Pour l'algorithme d'Euclide, la version itérative requiert également un espace mémoire constant (trois entiers pour `a`, `b` et `r`). Quant à la version récursive, elle empile les couples `(a,b)` arguments des appels récursifs, en attente du cas terminal : l'espace occupé croît avec la donnée initiale.

En conclusion, la récursivité est un outil précieux qui offre un style de programmation de haut niveau, souvent très proche de la formulation mathématique. Elle décharge le concepteur de tâches bureaucratiques comme la description d'itérations au moyen de structure de contrôles (boucles) et la fastidieuse gestion de pile. Elle permet de plus les approches du type *diviser pour régner* qui permettent des solutions claires, aisées à concevoir et à prouver. La contrepartie de ces avantages est que l'expression récursive est parfois tellement proche conceptuellement des mathématiques, que l'on peut en oublier qu'il s'agit en réalité d'algorithmes qui effectuent des itérations cachées et qui ont un coût en temps et en mémoire. Ces coûts masqués par la simplicité de la formulation sont souvent à l'origine d'explosions combinatoires. On en verra plusieurs exemples dans la suite de ce cours. Il convient donc de bien maîtriser l'évaluation de la complexité en temps et en mémoire d'un algorithme récursif avant de l'adopter.

On peut maintenant pressentir que les questions de coût en temps et en espace d'un algorithme sont essentielles : la section suivante leur est consacrée.

5.4 Complexité

Il y a deux notions distinctes de *complexité d'algorithmes* : la complexité en temps et la complexité en espace. Il s'agit respectivement d'une évaluation du comportement asymptotique (pour des données de taille arbitrairement grande) du temps et de l'espace mémoire nécessaires à leur

exécution.

On peut étendre cette notion à la *complexité des problèmes*, définie comme celle du meilleur des algorithmes qui les résout. La théorie de la complexité des problèmes et des algorithmes est une théorie difficile, hors du propos de ce cours. Elle comporte de nombreuses questions ouvertes. Par exemple, on ne sait pas toujours prouver qu'un algorithme est optimal (meilleure complexité) pour un problème donné. De plus, si l'on se propose de définir le *temps d'exécution* d'un algorithme en fonction de la *taille* des données, encore faut-il que de ces deux concepts aient été préalablement rigoureusement définis indépendamment de tout langage de programmation et de toute machine.

Dans cette section, on s'attache à donner une présentation pragmatique de la complexité en temps des programmes plutôt que des algorithmes, tout en raisonnant indépendemment du langage de programmation et de la machine sur laquelle ils s'exécutent. La définition de la taille d'un entier permet de faire la différence entre ces deux démarches. Lorsque l'on s'intéresse aux algorithmes dans une approche théorique de la complexité, la taille d'un entier est définie comme le nombre de chiffres de son développement binaire. L'algorithme de multiplication de deux entiers est alors d'autant plus coûteux que les opérandes sont grands. Lorsque, au contraire, on s'intéresse à des programmes, on considère en général que les entiers ont une taille fixe (puisque c'est le cas en machine), le coût d'une multiplication est donc constant. C'est ce dernier point de vue qui est adopté dans cette section.

5.4.1 Coût et complexité en temps d'un algorithme

Le *temps d'exécution* ou *coût* d'un algorithme est défini comme le nombre *d'opérations significatives (OS)* effectuées. Ce nombre est en général fonction de la *taille* de la donnée. La *complexité en temps* est définie comme le comportement asymptotique du coût.

Taille des données et opérations significatives (OS)

Les définitions de la taille de la donnée et des *OS* sont liées. Il s'agit en pratique de mettre en évidence un paramètre sur les données avec lequel croît le temps d'exécution. Voici, sur trois exemples, comment se détermine en pratique le choix de la taille des données et des *OS*.

Recherche d'un élément dans un ensemble L'algorithme consiste à parcourir l'ensemble en comparant chaque nouvel élément à celui qui est recherché. Le temps d'exécution est fonction du nombre de comparaisons effectuées, et celui-ci pourra être d'autant plus grand qu'il y a d'éléments dans l'ensemble. Les *OS* sont donc ici les **comparaisons** et la taille de la donnée est le **cardinal** de l'ensemble.

Produit de deux matrices Le calcul du produit consiste à effectuer des sommes de produits de coefficients. Il est d'autant plus élevé que les matrices sont grosses. Les *OS* sont donc les **sommes et produits de flottants**, et c'est le **format** des matrices qui est ici pris en compte pour la définition de la taille.

Calcul de la puissance *n*ième d'une matrice Un calcul de la puissance *n*ième d'une matrice carrée de dimension *dim* consiste à effectuer un certain nombre de produits de telles matrices. Le coût de chacun de ces produits ne dépend que de *dim*. Optimiser l'algorithme revient donc à en minimiser le nombre (cf. section 4.3), puisque l'on ne peut en aucune manière influer sur leur coût. La démarche intéressante est ici de choisir **le produit de deux matrices carrées de format** *dim* pour *OS* et puisqu'alors le nombre d'*OS* croît avec **l'exposant** n, c'est n qui est choisi comme taille.

On verra par la suite que si un doute subsiste entre plusieurs choix d'*OS* paraissant cohérents, ceux-ci n'influent pas sur le comportement asymptotique du coût.

Coût en moyenne et dans le pire des cas

Le nombre d'*OS* peut dépendre non seulement de la taille des données, mais également de leur configuration. Lorsque l'on recherche un élément x dans une liste de n éléments, la recherche s'arrête immédiatement si x est le premier élément de la liste : c'est le meilleur des cas. En revanche il faut faire n comparaisons, s'il est en fin de liste ou s'il n'y figure pas. On dit que la fonction C définie par $C(n) = n$ est le coût de l'algorithme dans le *pire des cas*. Lorsqu'il y a une telle distorsion entre le meilleur et le pire des cas, il est pertinent de rechercher le nombre d'*OS* effectuées *en moyenne*. En supposant qu'il y ait équiprobabilité des $n+1$ situations possibles (l'élément cherché n'est pas dans la liste, ou il est au $i$ème rang pour $i \in \{1, \ldots, n\}$), le nombre moyen d'*OS* est :

$$C_{moy}(n) = \frac{1}{n+1}(\sum_{i=1}^{n} i \ + \ n) \ = \ \frac{n}{2} + \frac{n}{n+1}.$$

Techniques d'évaluation du coût

Le nombre d'*OS* s'évalue selon les règles suivantes :

Suite d'instructions Le coût est la somme de celui de chaque instruction.
Conditionnelle Le coût de l'instruction `if(C) I1 else I2` est la somme du coût de l'évaluation de la condition `C` et du maximum des coûts des l'instructions $\texttt{I}_1$ et $\texttt{I}_2$.
Boucles Le coût d'une boucle (`while`, `for`, ...) est la somme des coûts de chaque itération et de celui l'évaluation de la condition de contrôle.
Appel de méthode S'il n'y a pas de récursivité, c'est le coût de la méthode. S'il y a récursivité, on obtient des relations de récurrence qu'il conviendra de résoudre.

On considère par exemple la méthode calculant itérativement *la puissance égyptienne* d'un flottant (cf. section 4.3.1).

```
static float puiss(float x, int n){
   float p = 1;
   while(n!=0){
     if(n%2 ==0)
       {x *= x; n /= 2;}
     else
       {p *= x; n--;}
   return p;
}
```

On a vu dans la section 4.3.1 que le nombre d'itérations de la boucle `while` est compris entre $\lfloor lg_2(n) \rfloor + 1$ à $2\lfloor lg_2(n) \rfloor + 1$. Si les *OS* sont les divisions, les multiplications, les décrémentations et les comparaisons, chaque itération effectue cinq *OS*. Le coût $C(n)$ de l'algorithme est donc tel que

$$5\lfloor lg_2(n) \rfloor + 5 \leq C(n) \ \leq 10\lfloor lg_2(n) \rfloor + 5$$

Une version récursive, directement traduite de la définition de l'exemple 5.3, peut-être programmée comme suit :

```
static float puiss(float x, int n){
  if (n==0) return 1;
  if (n%2 ==0) return(puiss(x*x, n/2));
  return(x*puiss(x, n-1));
}
```

Si $n \neq 0$, le calcul de x^n requiert une comparaison de n à 0 , un calcul du reste de la division de n par 2, et :

- soit, si n est pair, une multiplication (de x par lui-même), une division de n par 2 et un calcul de puissance $\frac{n}{2}$ ième.

- soit, dans le cas contraire, une décrémentation, une multiplication par x et un calcul de puissance $(n-1)$ième.

On en déduit, en notant *coût(n)* le coût de la version récursive pour un exposant n, que la suite $(\text{coût(n)})_{n\in I\!N}$ est définie par les formules récurrentes suivantes :

coût(0) = 0
coût(n) = coût(n/2)+5 si n> 0 et pair
coût(n) = coût(n-1)+5 si n> 0 et impair

Une façon de résoudre ces équations consiste, comme pour le cas itératif, à considérer le développement binaire de n. Si u désigne une suite binaire, on note $u.0$ ou $u.1$ la suite de symboles obtenue en concaténant à la suite u respectivement le symbole 0 ou 1. Ainsi, les équations précédentes peuvent s'écrire :

coût(0) = 0 (1)
coût(u.0) = coût(u)+5 (2)
coût(u.1) = coût(u.0)+5 (3)

Par exemple si n=19, il a pour développement binaire 10011. On obtient la suite d'égalités suivantes :

$coût(n) = coût(10011) = cout(10010) + 5 = coût(1001) + 2 \times 5 =$
$coût(1000) + 3 \times 5 = coût(100) + 4 \times 5 = coût(10) + 5 \times 5 =$
$coût(1) + 6 \times 5 = coût(0) + 7 \times 5 = 35.$

Plus généralement, notons $b_k b_{k-1} \dots b_1 b_0$ le développement binaire de n. On a montré (cf. section 4.3.1) qu'alors $k = \lfloor log_2(n) \rfloor$. On sait de plus que :

$$\text{coût(n)} = \text{coût}(b_k b_{k-1} \dots b_1 b_0).$$

Le calcul de cette valeur à partir de la définition récursive de *coût* aboutira à l'équation (1) après avoir utilisé l'équation (3) pour tous les indices $i \in \{0, \dots k\}$ tel que $b_i = 1$, donc un nombre de fois compris entre 1 et $k+1$, et l'équation (2) exactement k fois. Chaque utilisation de l'équation (2) ou (3) rajoutant 5 au résultat, on en déduit que le $5(1+k) \leq \text{coût(n)} \leq 5(2k+1)$, c'est-à-dire :

$$5\lfloor log_2(n) \rfloor + 5 \leq \text{coût(n)} \leq 10\lfloor log_2(n) \rfloor + 5$$

Le coût est identique pour les deux versions (itérative et récursive) de l'algorithme.

Complexité en temps

La complexité en temps est définie comme le comportement asymptotique du coût de l'algorithme. Comme l'on a distingué le coût en moyenne et dans le pire des cas, on définit la complexité en temps *en moyenne* et dans *le pire des cas*. La notion de comportement asymptotique est précisée dans la section 5.4.2. Il s'agit de comparer le comportement en $+\infty$ de la fonction de coût à celui des fonctions classiques (logarithmes, polynômes, exponentielles ...).

Ainsi, les complexités dans le pire des cas et en moyenne de la recherche séquentielle d'un élément dans une liste sont toutes deux *linéaires* en ce sens qu'elles se comportent comme des polynômes de degré 1 en $+\infty$: dans le pire des cas le coût est le polynôme $C(n) = n$ et en moyenne, il est équivalent à $\frac{n}{2}$. Pourtant, le second est approximativement deux fois plus faible que le premier.

Quant à l'algorithme de la puissance égyptienne, il est logarithmique en la taille de l'exposant.

La section suivante a pour objet de préciser la notion de comportement asymptotique.

5.4.2 Asymptotique

L'idée est de comparer le comportement des algorithmes sur des données de *grande* taille. On considère par exemple deux algorithmes A et B, solutions du même problème, dont les coûts sont respectivement définis par $C_A(n) = kn$ et $C_B(n) = k'n^2$, avec $k > 0,\ k' > 0$. Le premier est dit *linéaire* et le second *quadratique*. En termes de complexité, A est meilleur que B effet :

$$\exists N_0 \in I\!N \quad \forall n \geq N_0 \quad C_A(n) < C_B(n)$$

Par suite le coût de A sera moindre que celui de B pour toute donnée dont la taille est supérieure ou égale à N_0 [2].

Relations de comparaison

Définition 5.5 *Soient* f *et* g *deux fonctions de l'ensemble des entiers naturels dans l'ensemble des nombres réels positifs. On dit que* f est dominée asymptotiquement par g*, ou que* l'ordre de grandeur asymptotique de *f* est inférieur ou égal à celui de *g, et on note* f=O(g) *ou encore* f∈O(g) *si et seulement si :*

$$\exists k > 0 \quad \exists N_0 \in I\!N \quad \forall n \geq N_0 \quad f(n) \leq k.g(n)$$

Définition 5.6 *Soient* f *et* g *deux fonctions de l'ensemble des entiers naturels dans l'ensemble des nombres réels positifs. On dit que que* f *et* g *ont* même ordre de grandeur asymptotique*, et on note* f=Θ(g) *ou encore* f∈Θ(g) *si et seulement si l'une des trois propositions équivalentes suivantes est satisfaite :*

1. f=$O(g)$ *et* g=$O(f)$
2. $\exists k_1 > 0 \quad \exists k_2 > 0 \quad \exists N_0 \in I\!N \quad \forall n \geq N_0 \quad k_1.g(n) \leq f(n) \leq k_2.g(n)$
3. $\exists k_1 > 0 \quad \exists k_2 > 0 \quad \exists N_0 \in I\!N \quad \forall n \geq N_0 \quad k_1.f(n) \leq g(n) \leq k_2.f(n)$

L'équivalence des trois propositions est facile à établir et laissée au lecteur. On remarque les relations O et Θ sont définies à une constante multiplicative strictement positive près (cf. la règle 3 de la proposition 5.7 ci-dessous).

Proposition 5.7 *Soient* f, f_1, f_2, g, g_1, g_2 *des fonctions de l'ensemble des entiers naturels dans l'ensemble des nombres réels positifs. On démontre les règles suivantes.*

1. f=$O(f)$
2. f=$O(g)$ *et* g=$O(h)$ $\Rightarrow$ f=$O(h)$
3. f=$O(g)$$\Rightarrow$ $\forall a > 0,\ a.f$=$O(g)$
4. f_1=$O(g_1)$ *et* f_2=$O(g_2)$ $\Rightarrow$ f_1+f_2 = $O(max(g_1, g_2))$
5. f_1=$O(g_1)$ *et* f_2=$O(g_2)$ $\Rightarrow$ f_1-f_2 = $O(g_1)$
6. f_1=$O(g_1)$ *et* f_2=$O(g_2)$ $\Rightarrow$ $f_1.f_2$ = $O(g_1.g_2)$
7. $f \leq g \Rightarrow$ $O(f+g) = O(f)$
8. $\lim\limits_{+\infty} \dfrac{f}{g} = a > 0 \Rightarrow f = \Theta(g)$
9. $\lim\limits_{+\infty} \dfrac{f}{g} = 0 \Rightarrow f = O(g)$ *et* $g \neq O(f)$
10. $\lim\limits_{+\infty} \dfrac{f}{g} = +\infty \Rightarrow g = O(f)$ *et* $f \neq O(g)$

2. Il n'en reste pas moins vrai que si $k' = 1$ et $k = 10^9$, B est plus efficace que A pour toute donnée dont la taille est inférieure à cent millions.

Ces règles sont aisées à établir. La règle 8, par exemple, provient du fait que si $\lim_{+\infty} \frac{f}{g} = a > 0$,

$$\forall \epsilon > 0 \quad \exists N_0 \quad \forall n \geq N_0 \quad (a - \epsilon).g(n) < f(n) < (a + \epsilon).g(n).$$

En choisissant $\epsilon < a$, on obtient la propriété 2 de la définition 5.6 avec $k_1 = a - \epsilon$ et $k_2 = a + \epsilon$.

Il découle en particulier de ces règles que tout polynôme $P(n) = a_k.n^k + \ldots + a_1.n + a_0$ est en $\Theta(n^k)$ si $a_k > 0$.

Echelle de comparaison

Les fonctions de comparaison utilisées pour évaluer le comportement asymptotique des fonctions de coût sont, par ordre croissant, les suivantes :

1, log(n), n, nlog(n), n^α ($\alpha > 1$), a^n (a>1), n!, n^n

Chacune des fonctions de cette liste est dominée asymptotiquement par la suivante. La première est constante et égale à 1. $\Theta(1)$ est d'après la définition 5.6 l'ensemble des fonctions bornées en $+\infty$. Il s'agit du coût des algorithmes dits en *temps constant* (en fait, on devrait dire en temps borné). Il en est ainsi par exemple d'un algorithme qui renvoie le premier élément d'une liste : son coût est indépendant du nombre d'éléments de la liste puisque le parcours s'arrête sur le premier. On remarque de plus que la base des logarithmes n'est pas précisée. En effet d'après la formule :

$$log_a(n) = \frac{log_b(n)}{log_b(a)}$$

les deux logarithmes $log_a(n)$ et $log_b(n)$ diffèrent d'une constante multiplicative strictement positive dès que les bases a et b sont strictement supérieures à 1. On en déduit que :

$$\forall a > 1 \quad \forall b > 1 \quad \Theta(log_a(n)) = \Theta(log_b(n))$$

Il s'agit de l'ordre de grandeur asymptotique commun à la complexité de tous les algorithmes dits logarithmiques.
Enfin d'après la formule de Stirling, au voisinage de $+\infty$:

$$n! \sim \sqrt{2\pi n}.(\frac{n}{e})^n$$

Par suite, si $a > 0$:

$$\frac{n!}{a^n} \sim \sqrt{2\pi n}.(\frac{n}{e.a})^n$$

et donc :

$$\forall a > 0, \quad \lim_{+\infty} \frac{n!}{a^n} = +\infty$$

De même,

$$\frac{n!}{n^n} \sim \sqrt{2\pi n}.(\frac{1}{e})^n$$

et donc

$$\lim_{+\infty} \frac{n!}{n^n} = 0$$

Ceci explique que $n!$ domine asymptotiquement toutes les exponentielles et soit dominé par n^n.

Quelques nombres

Imaginons que l'ordinateur effectue 10^8 OS par seconde. Le tableau ci-dessous donne une évaluation du temps de calcul en fonction de la taille n de la donnée, pour des algorithmes dont les coûts respectifs sont indiqués en première ligne. On utilise les notations suivantes :

ns : nanosecondes	μs : microsecondes	ms : millisecondes
" : secondes	' : minutes	j : jours
M millions d'années	MM : milliards d'années	∞ : plus de 3.10^{285} milliards d'années

Les résultats ont été établis sachant que $log_2(10) = 3.321928094887362$.

Coût → n ↓	1	$log_2(n)$	n	$nlog_2(n)$	n^2	n^3	2^n
10^2	10 ns	66 ns	1 μs	6,6 μs	0,1 ms	10 ms	4.10^5 MM
10^3	10 ns	99 ns	10 μs	99 μs	10 ms	10"	∞
10^4	10 ns	0,13 μs	0.1 ms	1,3 ms	1"	2h46'	∞
10^5	10 ns	0,17 μs	1 ms	16,6 ms	1,7'	3 mois 25j	∞
10^6	10 ns	0,2 μs	10 ms	0,2"	2h46'	317 ans	∞
10^7	10 ns	0,23 μs	0.1"	2,33"	11,5 j	317.000 ans	∞
10^8	10 ns	0,26 μs	1"	26,6"	3 ans 2 mois	317 M	∞
10^9	10 ns	0,3 μs	10"	5 '	317 ans	317 MM	∞

Pour bien comprendre la portée de ces chiffres, il faut savoir que l'âge de l'univers est estimé à 13,7 milliards d'années.

On portera attention à l'évolution du temps de calcul de l'algorithme quadratique sur des données de taille 10^k pour $k \geq 5$. On remarquera également la différence de temps de calcul de l'algorithme en n^3 sur les données $n = 1000$, $n = 10000$, et $n = 100000$. Enfin la dernière colonne, entre autres, illustre ce qu'est *une explosion combinatoire* et montre qu'il est illusoire de fonder des espoirs sur les progrès techniques pour traiter informatiquement certains problèmes. Multiplier par mille la vitesse des ordinateurs serait une belle performance, mais resterait d'un piètre secours dans le cas de certaines complexités.

L'exemple suivant permet à nouveau de mesurer ce qu'est une explosion combinatoire. Supposons qu'un algorithme sur les matrices ait un coût $C(n) = 2^n$ où n est le nombre de coefficients de la matrice. Supposons maintenant que cet algorithme traite en une heure une matrice de format 10×10 (donc pour laquelle $n = 100$). Le coût du traitement d'une matrice de format 11×11 est :

$$C(121) = 2^{121} = 2^{100} \times 2^{21} = 2^{21}\, C(100)$$

Il faut ainsi 2^{21} fois plus de temps pour traiter la matrice 11×11 que pour traiter la matrice 10×10. Par suite, il faut 2^{21} heures = 231 ans pour traiter la matrice de format 11×11.

Quelques exemples

Exemple 5.6 (Les tours de Hanoï) *Ce problème a été posé par Edouard Lucas, mathématicien du XIXème siècle. On considère trois pieux et n disques percés en leur centre et de diamètres tous différents. Ces disques sont enfilés sur l'un des pieux, appelé* pieu de départ*, par ordre de diamètre décroissant. Le but est de les déplacer sur un autre pieu, appelé* pieu d'arrivée*, en utilisant si besoin est le troisième pieu, dit* pieu auxiliaire*. Les déplacements doivent obéir aux règles suivantes :*

- *on ne peut déplacer qu'un disque à la fois*
- *un disque ne peut être déplacé que sur un disque de diamètre supérieur au sien ou sur un emplacement vide.*

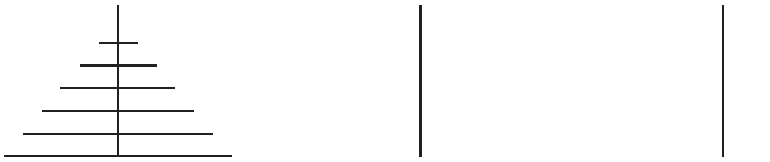
Il s'agit d'un problème qui se résoud aisément de façon récursive. Considérons l'algorithme `hanoï` possédant quatre paramètres : le nombre `n` de disques et les trois pieux appelés respectivement `départ`, `auxiliaire` et `arrivée`. Cet algorithme est défini comme suit :

```
hanoï(n, départ, arrivée, auxiliaire)
si n>0
   /*1*/ hanoï(n-1, départ, auxiliaire, arrivée)
   /*2*/ déplacer(départ, arrivée)
   /*3*/ hanoï(n-1, auxiliaire, arrivée, départ)
```

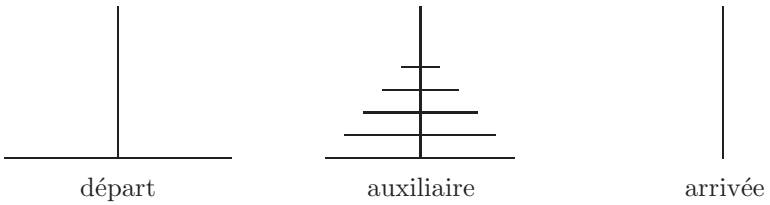
Si $n > 0$, cet algorithme se rappelle récursivement deux fois. Les cas terminaux sont ceux pour lesquels $n = 0$: l'algorithme ne fait rien dans ces cas-là[3].

Terminaison On choisit ici comme taille des données, le paramètre n. On montre par récurrence sur la taille des données que l'algorithme se termine toujours.
Si $n = 0$, l'algorithme ne fait rien, donc il se termine.
Soit $n \geq 0$, quelconque, fixé. Supposons que l'algorithme se termine sur toute donnée de taille n. Pour une donnée de taille $n + 1$ les deux appels récursifs se font sur des données de taille n, et donc se terminent par hypothèse de récurrence. En conséquence, l'algorithme se termine pour toute donnée de taille $n + 1$.

Correction Si $n = 0$, il n'y a rien à faire, donc l'algorithme est correct. Soit $n > 0$ et supposons que l'algorithme est correct pour $n - 1$ disques. La situation de départ est la suivante :

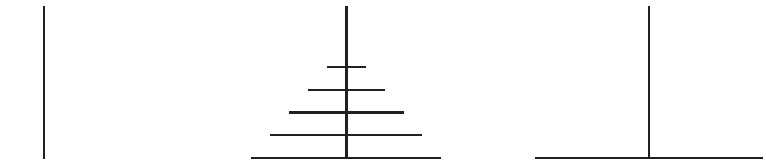


Après le premier appel récursif (instruction /*1*/), l'algorithme étant supposé correct pour $n - 1$ disques et le plus grand des disques n'influant pas sur les règles de déplacement des disques de diamètre inférieur, on se retrouve dans la situation suivante :

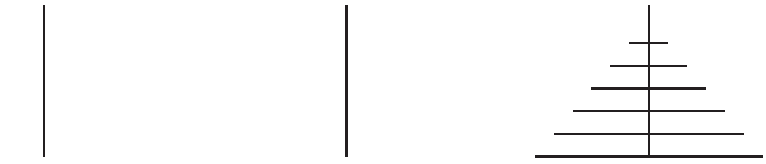


L'instruction /*2*/ consiste à déplacer le disque au sommet du pieu de départ (dans ce cas il n'y en n'a plus qu'un) vers le pieu d'arrivée, ce qui peut se faire sans enfreindre les règles. On est alors dans la situation suivante :

3. La seconde branche de la conditionnelle serait `"sinon rien"` et est ici sous-entendue (cf. section 1.1)



L'instruction /*3*/, que l'on a supposée correcte, déplace les $n-1$ disques du pieu auxiliaire sur le pieu d'arrivée, en utilisant le pieu de départ comme auxiliaire. Le disque préalablement enfilé sur le pieu d'arrivée ne gène en rien, son diamètre étant supérieur à tous les autres, tout se passe comme s'il n'existait pas. On obtient ainsi :



L'algorithme est donc correct.

Complexité Les OS sont ici les déplacements des disques et la taille n du problème est le nombre de disques. Soit donc $C(n)$ le nombre de déplacements nécessaires à l'algorithme `hanoï` pour traiter un problème de taille n. Les relations suivantes découlent immédiatemment de l'algorithme :

$C(0) = 0$
$C(n) = 2C(n-1) + 1 \qquad \forall n > 0$

On déduit de cette définition récurrente le calcul suivant pour $n > 0$:

$$\begin{array}{lll} & \mathrm{C(n)} & = 2\mathrm{C(n\text{-}1)}+1 \\ 2 \times | & \mathrm{C(n\text{-}1)} & = 2\mathrm{C(n\text{-}2)}+1 \\ 2^2 \times | & \mathrm{C(n\text{-}2)} & = 2\mathrm{C(n\text{-}3)}+1 \\ & \dots & \\ 2^{n-1} \times | & \mathrm{C(1)} & = 2\mathrm{C(0)}+1 \\ 2^n \times | & \mathrm{C(0)} & = 0 \\ \hline & \mathrm{C(n)} & = 1+2+2^2+ \ \dots \ +2^{n-1} = 2^n - 1 = \Theta(2^n) \end{array}$$

Exemple 5.7 (La suite de Fibonacci) *Elle est définie par récurrence comme suit :*

$\mathcal{F}_0 = 1$
$\mathcal{F}_1 = 1$
$\mathcal{F}_n = \mathcal{F}_{n-1} + \mathcal{F}_{n-2} \quad si \ \ n > 1\,;$

On en déduit immédiatement une méthode récursive calculant cette suite :

```
int fib(int n){
   if (n==0 || n==1)
```

```
    return(1);
  return(fib(n-1)+ fib(n-2));
}
```

Les OS sont ici les additions d'entiers et la taille du problème est le rang n du terme que l'on calcule. Le coût C de l'algorithme ainsi programmé satisfait les relations :

$C(0) = C(1) = 0 \qquad (I)$
$C(n) = C(n-1) + C(n-2) + 1 \quad \forall n > 1 \qquad (E)$

Le problème revient à résoudre l'équation du second ordre(E) avec les conditions initiales (I).

La première étape de la résolution consiste à résoudre l'équation homogène associée, à savoir :

$$C(n) \; - \; C(n-1) \; - \; C(n-2) \; = \; 0 \quad (H)$$

L'ensemble des suites solutions de cette équation est un espace vectoriel de dimension 2, donc de la forme :

$$\{(\lambda\varphi_n + \mu\varphi'_n)_{n\in I\!N} \; / \; (\lambda, \, \mu) \in I\!R^2\}$$

où $(\varphi_n)_n$ et $(\varphi'_n)_n$ sont deux solutions non nulles. On cherche ces solutions sous la forme de suites géométriques $(r^n)_{n\in I\!N}$, avec $r \neq 0$. Ces suites devant satisfaire (H), on obtient la condition :

$$\forall n > 1, \; r^n \; - \; r^{n-1} \; - \; r^{n-2} \; = \; 0$$

qui, puisque l'on suppose que r n'est pas nul, est équivalente à :

$$r^2 - r - 1 = 0$$

Cette équation a deux solutions :

$$\varphi = \frac{1+\sqrt{5}}{2} \quad (nombre \; d'or) \qquad \varphi' = \frac{1-\sqrt{5}}{2}$$

Les solutions de l'équation homogène (H) sont donc toutes les suites de terme général :

$$\lambda.\varphi^n \; + \; \mu.\varphi'^n \qquad (\lambda, \, \mu) \in I\!R^2$$

La deuxième étape de la résolution consiste à trouver une solution particulière de l'équation (E). La suite constante $(-1)_n$ convient. On en déduit que les solutions de l'équation (E) sont exactement les suites de terme général :

$$\lambda.\varphi^n \; + \; \mu.\varphi'^n - 1 \qquad (\lambda, \, \mu) \in I\!R^2$$

Sachant que l'on recherche parmi ces solutions, celles qui satisfont les conditions initiales (I), on en déduit qu'il faut et il suffit que : $\lambda \; + \; \mu \; = \; 1$
$\lambda\varphi \; + \; \mu\varphi' \; = \; 1$

On trouve ainsi :

$$\lambda \; = \; \frac{1-\varphi'}{\varphi-\varphi'} \; = \; \frac{\varphi}{\varphi-\varphi'} \; = \; \frac{5+\sqrt{5}}{10} \qquad et \qquad \mu \; = 1 - \lambda \; = \; \frac{5-\sqrt{5}}{10}$$

On obtient alors le résultat cherché :

$$C(n) = \frac{5+\sqrt{5}}{10}\varphi^n \; + \; \frac{5-\sqrt{5}}{10}\varphi'^n \; - \; 1$$

Puisque $-1 < \varphi' < 0$, on en déduit que

$$C(n) \; = \; \Theta(\varphi^n) \; = \; \Theta(\frac{1+\sqrt{5}}{2})^n \; \approx \; \Theta(1,618^n)$$

Il s'agit donc d'un algorithme exponentiel. Cette inefficacité s'explique par le fait que l'algorithme effectue un grand nombre de fois le même calcul. La méthode `fib(n)` fait deux appels récursifs sur $(n-1)$ et $(n-2)$. Chacun de ces appels fait à son tour deux nouveaux appels récursifs, etc. On peut schématiser ces appels par un *arbre de récursion*, partiellement décrit par la figure 5.2 dans le cas $n = 11$. De chaque nœud n de l'arbre sont issus deux autres nœuds, arguments des deux appels récursifs effectués par `fib(n)`. On peut noter que `fib(7)` par exemple est appelé cinq fois. Chacun de ces cinq appels fait lui-même des calculs redondants. C'est ce défaut de conception qui provoque l'explosion combinatoire.

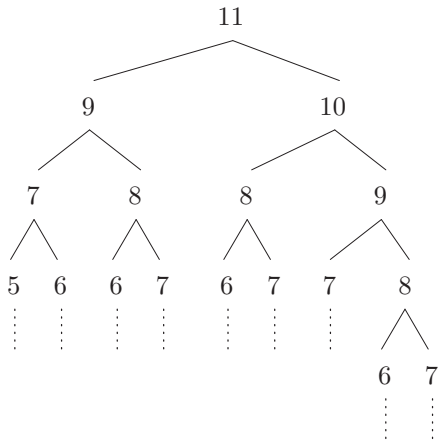


FIGURE 5.2 – L'arbre de récursion de `fib(11)`

Il existe toutefois une solution itérative bien plus efficace pour ce problème (cf. section 1.4.4, exemple 1.13) rappelée ci-après.

```
int fib(int n){
  int a=1, b=1, nouveau;
  for (int i=1 ; i<n ; i++){
    nouveau = a+b;
    a = b;
    b = nouveau;
 }
 return b;
}
```

Chaque itération fait un nombre constant d'opérations. Puisqu'il y a $(n-1)$ itérations, on en déduit que cet algorithme est linéaire et donc infiniment plus efficace que la version précédente.

Exemple 5.8 (Complexité de l'algorithme d'Euclide) *Cet algorithme de recherche du* PGCD *se programme récursivement (cf. exemple 5.4) et itérativement (cf. section 2.2.1) comme suit :*

```
static int delta(int a, int b){
  int r = a%b;
  if(r == 0) return b;
  return (delta(b, r));
}
```

```
static int pgcd (int a, int b){
   int r;
   while (b!=0)
    {r=a%b; a=b;b=r;}
   return a;
}
```

Le coût de la version récursive est proportionnel au nombre d'appels récursifs, celui de la version itérative est proportionnel au nombre d'itérations. Clairement, ces deux nombres sont identiques, puisque chaque appel récursif, de même que chaque itération, remplace le couple courant (a, b) par le couple $(b, a \ mod \ b)$ et la condition d'arrêt est identique. L'évaluation du coût de l'algorithme d'Euclide, quelle qu'en soit la version, utilise la suite de Fibonacci $(\mathcal{F}_n)_{n\in\mathbb{N}}$.

Lemme 5.8 *Soient a et b deux entiers tels que $a > b > 0$. Si l'algorithme d'Euclide fait $n \geq 1$ itérations (ou appels récursifs) sur (a, b), alors $a \geq \mathcal{F}_{n+2}$ et $b \geq \mathcal{F}_{n+1}$.*

La preuve se fait par récurrence sur n.
Si $n = 1$, b ne divise pas a, donc $b \geq 2 = \mathcal{F}_2$ et puisque $a > b$, nécessairement $a \geq 3 = \mathcal{F}_3$.
Soit $n \geq 1$, quelconque fixé. Supposons que la propriété à démontrer est vraie pour n et considérons un couple (a, b) pour lequel l'algorithme fait $(n+1)$ appels récursifs. On en conclut que le premier appel récursif, qui se fait sur le couple $(b, a \ mod \ b)$, rappelle récursivement l'algorithme n fois. On déduit alors de l'hypothèse de récurrence que : $b \geq \mathcal{F}_{n+2}$ et $a \ mod \ b \geq \mathcal{F}_{n+1}$.
Or, en posant $r = a \ mod \ b$, on sait que a est de la forme : $a = bq + r$, et que le quotient q est supérieur ou égal à 1, puisque l'on a supposé que $a > b$. On en déduit que :

$$a \geq b + r \geq \mathcal{F}_{n+2} + \mathcal{F}_{n+1} = \mathcal{F}_{n+3}$$

□

Théorème 5.9 (Théorème de Lamé) *Soient a et b deux entiers tels que $a > b > 0$. Soit $n \geq 0$ un entier tel que $b < \mathcal{F}_{n+2}$, où $\mathcal{F}_{n+2}$ désigne le terme de rang $(n + 2)$ de la suite de Fibonacci. L'algorithme d'Euclide fait au plus* n *itérations (ou appels récursifs) sur un tel couple (a, b). De plus, ce nombre est atteint pour certains couples (a, b) satisfaisant ces conditions.*

Ce théorème découle immédiatement du lemme précédent, puisque s'il y avait $(n + 1)$ appels récursifs ou plus, on aurait $b \geq \mathcal{F}_{n+2}$. De plus, on montre par récurrence sur n, qu'il y a exactement n appels récursifs pour le couple $(a,\ b) = (\mathcal{F}_{n+2},\ \mathcal{F}_{n+1})$.
Si n=0, $(\mathcal{F}_{n+2},\ \mathcal{F}_{n+1}) = (2,\ 1)$. Il n'y a dans ce cas aucun rappel récursif.
Soit $n \geq 0$. Supposons qu'il y ait exactement n appels récursifs pour le couple $(\mathcal{F}_{n+2},\ \mathcal{F}_{n+1})$. Sur le couple $(\mathcal{F}_{n+3},\ \mathcal{F}_{n+2})$, l'algorithme s'appelle récursivement une première fois avec $(\mathcal{F}_{n+2},\ r)$ où r est le reste de la division de $\mathcal{F}_{n+3}$ par $\mathcal{F}_{n+2}$. Or, puisque $\mathcal{F}_{n+3} = \mathcal{F}_{n+2} + \mathcal{F}_{n+1}$ et que $\mathcal{F}_{n+2} > \mathcal{F}_{n+1}$, on en déduit que $r = \mathcal{F}_{n+1}$.
Finalement, sur le couple $(\mathcal{F}_{n+3},\ \mathcal{F}_{n+2})$, l'algorithme fait un premier rappel récursif sur $(\mathcal{F}_{n+2},\ \mathcal{F}_{n+1})$, qui d'après l'hypothèse de récurrence effectue lui-même n appels récursifs. Donc il y a au total $(n + 1)$ appels récursifs. □

On peut maintenant évaluer la complexité de l'algorithme pour tous les couples (a, b) tels que $a > b > 0$. Rappelons que la fonction *taille* a été définie par $taille(a, b) = b$. La fonction de coût est donc une fonction de b. De tout ce qui précède, on peut déduire qu'il existe une constante strictement positive k telle que, dans le pire des cas pour $\mathcal{F}_{n+1} \leq b < \mathcal{F}_{n+2}$, *coût(b)=k.n.* Il reste donc à évaluer n en fonction de la taille b.

Le calcul de $\mathcal{F}_n$ se fait de façon analogue à celui de la fonction C de l'exemple précédent. L'équation générale est ici homogène et a donc pour solutions les suites de terme général :

$$\lambda.\varphi^n + \mu.\varphi'^n \qquad (\lambda,\ \mu) \in \mathbb{R}^2$$

avec $\varphi = \dfrac{1+\sqrt{5}}{2}$ $\qquad$ $\varphi' = \dfrac{1-\sqrt{5}}{2}$.

Des conditions initiales (les termes de rang 0 et 1 sont égaux à 1), on déduit : $\begin{array}{l} \lambda + \mu = 1 \\ \lambda\varphi + \mu\varphi' = 1 \end{array}$

On trouve ainsi :

$$\mathcal{F}_n = \lambda\varphi^n \ + \ \mu\varphi'^n \qquad avec \qquad \lambda \ = \ \frac{5+\sqrt{5}}{10} \quad et \quad \mu \ = \ \frac{5-\sqrt{5}}{10}$$

Par suite, la condition $\mathcal{F}_{n+1} \leq b < \mathcal{F}_{n+2}$ équivaut à :

$$\lambda.\varphi^{n+1} \ + \ \mu.\varphi'^{n+1} \leq b < \lambda.\varphi^{n+2} \ + \ \mu.\varphi'^{n+3}$$

Puisque $-1 < \varphi' < 0$, et que n tend vers l'infini quand b tend vers l'infini, à partir d'une certaine valeur b_0 de b :

$$\frac{\lambda}{2}.\varphi^{n+1} < \lambda.\varphi^{n+1} \ - \ \frac{\lambda}{2} \leq b < \lambda.\varphi^{n+2} \ + \ \frac{\lambda}{2} < \frac{3\lambda}{2}.\varphi^{n+2}$$

De l'encadrement :

$$\frac{\lambda}{2}.\varphi^{n+1} < b < \frac{3\lambda}{2}.\varphi^{n+2}$$

on déduit qu'il existe deux constantes k_1 et k_2 telles que pour tout $b > b_0$

$$\alpha.n + k_1 < log(b) < \alpha.n + k_2$$

où $\alpha \ = \ log(\varphi) > 0$. Donc $n = \Theta(log(b))$ et par suite *coût(b)*=$\Theta(log(b))$.

Ce calcul a été fait pour des données (a, b) telles que $a > b$. Le cas où $a = b$ est un cas terminal et ne pose donc pas de problème. Enfin, si $a < b$, le premier reste calculé est a et donc le premier rappel récursif se fait sur le couple (b, a) qui remplit les conditions du théorème. On ne fait dans ce cas qu'un rappel récursif supplémentaire, ce qui ne change pas la complexité.

En ce qui concerne l'espace occupé, la version itérative requiert trois variables quelle que soit la donnée. La version récursive empile les couples arguments des appels récursifs jusqu'à ce qu'un cas terminal soit atteint. La taille de la pile de récursivité est donc proportionnelle au nombre d'appels récursifs.

On peut donc conclure par la proposition suivante.

Proposition 5.10 *La complexité en temps, dans le pire des cas, de l'algorithme d'Euclide pour le calcul du* PGCD *d'un couple* $(a, b) \in I\!N \times I\!N^*$ *est en* $\Theta(log(b))$.
La complexité en espace est en $\Theta(1)$ *pour la version itérative et en* $\Theta(log(b))$ *pour la version récursive.*

Chapitre 6

Structures séquentielles

Ce chapitre et les deux suivants, traitent de la structuration des données complexes et de leur traitement algorithmique. Chacun est consacré à l'une des trois grandes classes de structures : les structures séquentielles (ou listes), les structures arborescentes (ou arbres) et les structures relationnelles (ou graphes).

Toute structure sera définie à deux niveaux : l'un abstrait, indépendant de tout langage de programmation, l'autre consistant en une ou plusieurs mises en œuvre en Java. La définition abstraite d'une structure de donnée sera composée non seulement d'une description des données elles-mêmes mais également d'un jeu de *primitives* destiné à leur manipulation, le tout constituant en quelque sorte une *boîte à outils* prête à l'emploi. Au niveau concret, il s'agira d'une classe Java, dont les champs représentent la donnée et les méthodes d'instance les primitives. Le programmeur peut ainsi disposer d'outils bien pensés, sans avoir à les redéfinir : on gagne alors en efficacité et en fiabilité.

On désigne sous le nom de *structures séquentielles*, les structures de données qui représentent des suites finies d'éléments d'un même type. Parmi celles-ci, les listes sont les plus générales.

6.1 Les listes

6.1.1 Description abstraite

Donnée

Il s'agit d'une suite finie, éventuellement vide, $l = (l_0, \ \ldots, l_{n-1})$ dont les éléments sont tous d'un même type T. On considère également un type *position*, isomorphe à $\mathbb{N}$. A chaque élément de la liste est affectée une position qui, informellement, fait référence à l'endroit où il se trouve dans la liste et sera définie précisément dans les mises en œuvre.

Primitives

Soient x un élément de type T et p une *position*. On définit les primitives suivantes.

fin(l) : la position qui suit immédiatement celle du dernier élément de la liste. Intuitivement, $fin(l)$ est la position d'un nouvel élément que l'on rajouterait à la fin de la liste. Si l est vide, c'est une position choisie par convention qui sera celle de l'unique élément de la liste obtenue en rajoutant un élément à la liste vide.

vide(l) : renvoie une valeur booléenne indiquant si l est vide ou non.

début(l) : renvoie la position du premier élément de la liste l si elle n'est pas vide. Si l est vide, renvoie $fin(l)$.

suivant(p, l) : n'est défini que si p est la position d'un élément l_i de la liste. Renvoie alors la position de l_{i+1} si cet élément existe, et $fin(l)$ sinon.

précédent(p, l) : n'est défini que si p est la position d'un élément l_i de la liste autre que le premier. Renvoie alors la position de l_{i-1}.

listeVide() : renvoie une liste vide.

position(x, l) : renvoie la position de la première occurrence de x dans la liste si elle existe, et $fin(l)$ sinon.

élément(p, l) : n'est défini que si p est la position d'un élément de la liste. Renvoie cet élément.

rajouter(x, p, l) : n'est défini que si p est la position d'un élément l_i de la liste ou $fin(l)$. Dans le premier cas, l est transformée en $l = (l_0, \ldots, l_{i-1}, x, l_i, \ldots, l_{n-1})$, et dans le second en $l = (l_0, , \ldots, l_{n-1}, x)$

supprimer(p, l) : n'est défini que si p est la position d'un élément l_i de la liste. La liste est transformée en $l = (l_0, \ldots, l_{i-1}, l_{i+1}, \ldots, l_{n-1})$ ou en $l = (l_0, \ldots, l_{n-2})$ selon que l_i n'est pas ou est le dernier élément de la liste. La valeur de l'élément supprimé est renvoyée.

On peut désormais utiliser ces outils élémentaires pour concevoir des algorithmes sur les listes. Le parcours d'une liste `l` se fait par exemple sur le schéma suivant :

```
pour(p ←début(l); p ≠ fin(l); p ← suivant(p,l)) ...
```

Voici, en illustration, un algorithme `purger` qui supprime d'une liste toutes les répétitions. Il consiste à parcourir itérativement la liste depuis le début. On suppose qu'à chaque itération, la partie de la liste comprise entre le premier élément et celui précédant immédiatement l'élément courant est déjà purgée, c'est-à-dire que la liste ne contient plus aucune répétition des éléments de ce segment initial.

La nouvelle itération consiste alors à supprimer les éventuelles répétitions de l'élément courant (de position `posCour`) dans la suite de la liste.

```
purger(l)
   p, posCour : position
   pour(posCour ← début(l); posCour ≠ fin(l); posCour ← suivant(posCour,l))
      p ← suivant(posCour,l)
      tant que (p ≠ fin(l)) faire
         si identique(élément(p,l), élément(posCour,l))
            supprimer(p, l)
         sinon
            p ← suivant(p,l)
```

La fonction *identique* dépendra des implémentations. On remarquera, et c'est le point délicat de l'algorithme, qu'il ne faut pas modifier `p` si l'on vient de supprimer l'élément `x` de position `p`. En effet, `p` devient alors la position de l'élément suivant, qui doit être à son tour examiné.

6.1.2 Mise en œuvre par des tableaux

On décide de mémoriser une liste dans un tableau de dimension 1. La *position* d'un élément est ici son indice. Le tableau n'est pas nécessairement totalement rempli, la liste pouvant n'en occuper que le début. Aussi, la liste est-elle définie non seulement par le tableau, mais aussi par

l'indice `fin` de la première case non occupée, ce qui est conforme à la spécification de la primitive abstraite **fin**.

`fin` = indice de la première case non occupée

On est ainsi amené à définir la classe Java suivante :

```
public class Liste{

    private int[]T;
    private int fin;

    Liste(int n){
        T = new int[n];
        fin = 0;
    }
```

Pour éviter une certaine lourdeur dans l'écriture, on utilise la notation avec `this` implicite (cf. section 3.3.1). Les deux champs sont déclarés avec l'attribut `private`, selon la méthodologie indiquée dans la section 3.2.2. La primitive **créerVide** est mise en œuvre par le constructeur `Liste` : on crée un tableau `T` destiné à recevoir la future liste. L'initialisation à 0 du champ `fin` assure que la liste est vide.

Certaines primitives sont évidentes et ne demandent pas l'écriture d'une méthode. Ainsi, `debut(l)` est simplement 0 et `suivant(i,l)` est `i+1`. Quant aux primitives `element`, `fin` et `vide` elles se programment aisément comme suit.

```
    int element(int p) {return(T[p]);}
    boolean vide() {return(fin == 0);}
    int fin(){return fin;}
```

La recherche de la position d'un élément `x` dans la liste a été ici programmée à l'aide d'une sentinelle. Cela suppose que la liste n'occupe pas tout le tableau. Le principe consiste à mémoriser `x` dans la première case non occupée (instruction /*1*/), qui porte alors le nom de *sentinelle*. Ceci ne revient en aucun cas à modifier la liste, puisque la *sentinelle* n'y appartient pas. Elle assure cependant la sortie de la boucle `while` (ligne /*2*/) même si `x` ne figure pas dans la liste. Dans ce cas, la méthode renvoie `i = fin`, ce qui est conforme à la spécification de la primitive **position**.

```
    int position(int x){
        int i = 0;

        /*1*/ T[fin] = x;
        /*2*/ while(T[i]!=x) i++;
        /*3*/ return(i);
    }
```

Si l'on n'utilise pas de sentinelle et que `x` n'est pas dans la liste, l'exécution de l'instruction

```
while(T[i]!=x) i++;
```

provoque des comparaisons erronées puisque `i` est incrémenté jusqu'à ce qu'il prenne la valeur `fin`. A partir de cette valeur, `x` est comparé à des éléments qui ne sont pas dans la liste. L'algorithme est donc faux. Mais de plus, si `x` ne figure pas dans le tableau, `i` finit par prendre la valeur `T.length`. La comparaison `T[i]!=x` provoque alors un *débordement de tableau*, puisque `i` est strictement supérieur à l'indice de la dernière case du tableau, et l'affichage de l'exception :

```
ArrayIndexOutOfBoundsException: i
```

On est donc contraint de comparer la valeur de `i` à `fin` *avant* de comparer `T[i]` à `x`.

```
int position (int x){
    int i = 0;

    while(i!=fin && T[i]!=x) i++;
    return i;
}
```

Il faut souligner que la version sans sentinelle n'est pas uniquement plus délicate à mettre en œuvre, elle est également moins efficace puisqu'elle demande deux fois plus de comparaisons que l'autre.

Pour rajouter un élément `x` à l'indice `p`, il convient de décaler d'un rang vers la droite tous les éléments de rang `p`$\leq$`i< fin`. On remarque qu'il faut procéder de droite à gauche, sous peine de recopier l'élément d'indice `p` dans toutes les variables `T[i]`, pour `p<i<fin` en écrasant tous les autres. Enfin, le rajout n'est possible que si le tableau `T` n'est pas plein, ce qui est implicitement supposé dans la version ci-après.

```
void rajouter (int x, int p){
    for(int i = fin-1; i>=p ; i--)
      T[i+1] = T[i];
    fin++;
    T[p]=x;
}
```

La suppression d'un élément d'indice `p` revient, après avoir sauvegardé la valeur supprimée pour la renvoyer ultérieurement, à décaler d'un rang vers la gauche tous les éléments d'indices `p` <`i`<`fin` et à décrémenter la variable `fin`.

```
int supprimer(int p){
    int aux= T[p];
    for(int i = p+1; i!=fin;i++)
      T[i-1]=T[i];
    fin--;
    return(aux);
}
```

On peut rajouter, une méthode permettant de visualiser les résultats.

```
void imprimer(){
    for(int i=0; i<fin(); i++)  System.out.print(T[i]+" ");
    System.out.println();
}
```

Ces primitives dûment implémentées, la méthode qui purge une liste suit exactement l'algorithme précédemment décrit, et c'est ce qui fait l'intérêt de cette approche. Si elle est déclarée à l'intérieur de la classe `Liste`, c'est une méthode d'instance, qui purge donc `this`. On a accédé ici directement aux éléments du tableau, évitant ainsi un appel à la méthode `element`, qui ne fait rien d'autre que de renvoyer l'élément de `T` dont l'indice est indiqué en paramètre. De même, on évite un accès à la méthode `fin`, qui se borne à renvoyer la valeur de la variable, accessible à l'intérieur de la classe, de même nom.

```
void purger(){
    int  p;

    for(int posCour=0; posCour!=fin; posCour++){
```

```
        p= posCour+1;
        while(p!=fin)
            if (T[posCour] == T[p])  supprimer(p);
            else   p++;
    }
}
```

Si la méthode est programmée dans une autre classe, il s'agit d'une méthode statique, qui purge une liste passée en paramètre. On n'a plus directement accès aux éléments du tableau `T`, ni à la variable `fin` puisqu'il s'agit de champs privés de la méthode `Liste`. On est contraint d'appeler les méthodes `element` et `fin`.

```
static void purger(Liste l){
    int  p;

    for(int posCour=0; posCour !=l.fin(); posCour ++){
        p= posCour +1;
        while(p!=l.fin())
            if (l.element(posCour) == l.element(p))  l.supprimer(p);
            else  p++;
    }
}
```

En supposant que la méthode `purger` ait été définie comme une méthode d'instance de la classe `Liste`, on peut la tester au moyen du programme suivant.

```
import java.util.*;

public class ProgListes{

    public static void main(String [] args){
        Scanner sc = new Scanner(System.in);
        int n =args.length, x, p;
        Liste l = new Liste(n);

        for(int i=0;i<n; i++)
            l.rajouter(Integer.parseInt(args[i]), l.fin());
        l.purger();
        l.imprimer();
        try{
            while(true){
                System.out.println("Elt a rechercher?");
                x = sc.nextInt();
                p=l.position(x);
                if(p==l.fin())
                    System.out.println(x + " est absent");
                else
                    System.out.println("Position de " + x + " : "+ p);
            }
        }
        catch(NoSuchElementException e){
        System.out.println("fin");
        }
    }
}
```

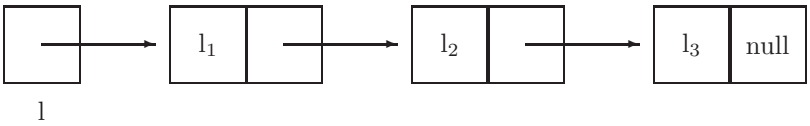
dont voici une trace d'exécution :

```
$ java ProgListes 3 67 1 90 8 7 5 67 78 90 67 67 1 1 1
3 67 1 90 8 7 5 78
Elt a rechercher?
98
98 est absent
Elt a rechercher?
90
Position de 90 : 3
Elt a rechercher?
0
0 est absent
Elt a rechercher?
78
Position de 78 : 7
Elt a rechercher?
fin
```

La fin de l'exécution a été détectée par un `Ctrl D`, tapé par l'utilisateur.

6.1.3 Mise en œuvre par des listes chaînées

Une liste chaînée est composée d'objets appelés *maillons*, chacun étant constitué de deux champs : l'un portant un élément de la liste, l'autre l'adresse de (i.e. un pointeur vers) le maillon suivant. La représentation en machine de la suite $l = (l_1, l_2, l_3)$ peut être alors schématisée de la façon suivante :



Toutefois, une liste est souvent construite par ajouts successifs d'éléments à partir de la liste vide (cf. le programme `ProgListe` de la section précédente). La représentation de la liste vide est donc essentielle. Comment la choisir si, comme indiqué dans le schéma, la variable `l` pointe vers le maillon portant le premier élément de la liste ? Que faire s'il n'y a aucun d'élément ? Il serait tentant de donner à `l` la valeur `null` dans ce cas-là : pas d'élément, donc pas de maillon. Mais quand aucune liste n'a encore été créée, `l` a également la valeur `null`. Dès lors, on ne peut plus faire de différence entre l'absence de liste et l'existence d'une liste vide.

Par analogie, si sous le système Unix et dans un répertoire vide, on tape la commande `touch fic`, un fichier vide nommé `fic` apparaît dans le répertoire. Le répertoire n'est plus vide, il contient un fichier, qui a un nom, une date de création, un jeu de permissions, etc. Ce fichier est vide mais pourra par la suite faire l'objet d'ajouts. Il en est de même avec les listes : il faut pouvoir faire la différence entre l'absence de liste et l'existence d'une liste vide, réel objet créé à l'aide d'un `new`, comme c'était le cas dans la représentation par des tableaux. La solution consiste à faire débuter toutes les listes par un maillon, appelé *en-tête*, qui ne porte aucun élément de la liste.

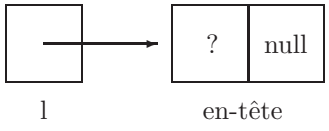


FIGURE 6.1 – Représentation de la liste vide : $l = ()$

La liste vide est alors la liste réduite à son en-tête, comme illustré par la figure 6.1. Dans le cas général d'une liste a n éléments, la liste chaînée possède $n + 1$ maillons. Ainsi, la figure 6.2 représente une suite à trois éléments.

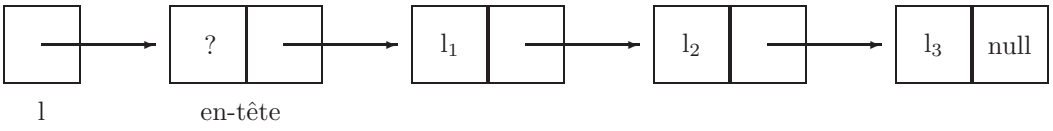


FIGURE 6.2 – Représentation de la suite $l = (l_1, l_2, l_3)$

Avec ce type de représentation, on peut parcourir la liste du premier au dernier élément, en passant d'un maillon au maillon suivant. Il est en revanche impossible de revenir en arrière, puisqu'il n'y a pas d'accès au maillon précédent.
De plus, en cas d'ajout à un certain endroit de la liste, cet *endroit* est nécessairement précisé par l'adresse du maillon *après* lequel on rajoute l'élément, donc par l'adresse du maillon qui *précèdera* l'élément rajouté. En cas de suppression, il conviendra de modifier le maillon *précédant* le maillon à supprimer. C'est ainsi que l'on convient de définir la *position* d'un élément par *l'adresse du maillon précédent*.

position d'un élément = adresse du maillon précédent

Il en résulte immédiatement que la position du premier élément l_1 d'une liste l est l'adresse de l'en-tête, c'est-à-dire l lui-même.

début(l) = l

La classe `Liste` débute par la déclaration de deux champs, l'un `elt` de type entier puisque l'on s'intéresse ici à des listes d'entiers, l'autre qui porte l'adresse du maillon suivant, donc lui aussi de type `Liste`. Ils sont protégés en écriture par l'attribut `private`. Par abus de langage, on dira par la suite *le maillon l* pour *le maillon d'adresse l*.

```
public class Liste {

    private int elt;
    private Liste suiv;

    Liste() {this.suiv = null;}
    Liste(int n, Liste s) {this.elt = n; this.suiv = s;}

    boolean vide() {return (this.suiv == null);}

    int element() {return this.suiv.elt;}
    Liste suivant() {return this.suiv;}
```

Le premier constructeur initialise une liste à vide ; en stipulant que le suivant du maillon est `null`, on indique qu'il n'y a qu'un seul maillon : `this`. On ne précise pas la valeur du champ `elt` puisqu'elle est non significative dans l'en-tête. Le second constructeur permet de créer un maillon dont les valeurs des deux champs sont indiquées en paramètres. Quant à la primitive `vide`, elle renvoie la valeur `true` si et seulement si la liste est vide, donc si et seulement si elle est réduite au maillon d'en-tête, donc si et seulement si son champ `suiv` est `null`.

Les champs étant protégés en écriture, on y accède en lecture à l'extérieur de la classe par les

méthodes `element` et `suivant` : l'élément de position `this` est `this.suiv.elt` d'après les conventions choisies pour définir les positions, et le suivant est simplement `this.suiv`.

Le parcours de la liste `this` se fait alors sur le schéma suivant :

```
for (p = this; !p.vide(); p=p.suiv)
```

La primitive **fin(l)** ayant été définie comme la position suivant celle du dernier élément, la position du dernier élément étant l'adresse du maillon qui le précède, la fin de la liste est donc l'adresse du dernier maillon ! La fin de la liste `this` se calcule donc de la façon suivante :

```
Liste fin (){
   Liste p;
   for (p = this; !p.vide(); p=p.suiv);
   return (p);
}
```

Contrairement à la représentation avec tableau où l'accès à la fin se fait en temps constant, ce calcul est ici coûteux puisqu'il nécessite le parcours de toute la liste. On remarque toutefois que la comparaison d'une *position* `p` à la fin de la liste peut se faire par `p.vide()` (temps constant) plutôt que par `p==this.fin()` (complexité linéaire).

La fin de la liste est la *position p* telle que *p.vide()*

La recherche de la position d'un élément `x` consiste à renvoyer le maillon `p` tel que `p.element() == x`. Comme pour les tableaux, il faut pouvoir s'arrêter à la fin de la liste si `x` n'a pas été trouvé. Ainsi, *avant* d'examiner l'élément de position `p`, c'est-à-dire avant de faire un appel à `p.element()`, il convient de s'assurer qu'un tel élément existe bien, autrement dit que `p` n'est pas vide, faute de quoi il se produit l'erreur `NullPointerException` (cf. section 3.1.3).

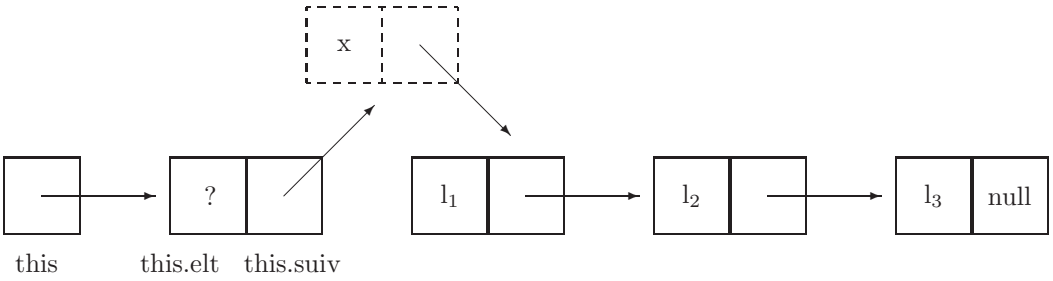
```
 Liste position (int x){
        Liste p = this;
        while (!p.vide() && p.element() != x) p=p.suiv;
        return p;
}
```

On remarque que l'évaluation de l'expression

```
!p.vide() && p.element() != x
```

ne provoque pas d'exception. En effet, l'opérateur `&&` est évalué séquentiellement (cf. section 1.1.3) : si `!p.vide()` prend la valeur `false`, l'expression globale est évaluée à `false` sans que la seconde partie de l'expression ne soit évaluée. Ainsi, on ne tente pas d'accès à l'élément de position `p` si `p` est la fin de la liste.
On remarque enfin que cette méthode renvoie bien, comme spécifié, la fin de la liste si `x` n'y figure pas.

On se propose maintenant de rajouter un élément `x` en première position dans la liste `this`. Autrement dit, si `this` pointe vers la représentation de la liste (l_1, l_2, l_3), après insertion de `x`, il doit pointer vers (x, l_1, l_2, l_3). Il convient donc de rajouter comme suivant de l'en-tête, c'est-à-dire comme suivant de `this`, un nouveau maillon, créé à l'aide du second constructeur. Celui-ci est alors appelé avec `x` comme premier paramètre, et l'adresse du maillon qui suivait `this` comme second.



Ceci se programme simplement comme suit.

```
void rajouter(int x){
   this.suiv = new Liste(x, this.suiv);
}
```

L'ajout d'un élément `x` en position `p`, c'est-à-dire juste après le maillon d'adresse `p`, se fait en appelant la méthode précédente avec `p` à la place de `this`, soit `p.rajouter(x)`.

La suppression du premier élément de la liste consiste, après avoir sauvegardé sa valeur pour la renvoyer ultérieurement, à modifier le maillon suivant `this`.

```
int supprimer(){
   int aux = this.element();
   this.suiv = this.suiv.suiv;
   return(aux);
}
```

Cette méthode provoque une erreur si la liste est vide, puisque dans ce cas, `this.suiv` vaut `null` et donc `this.suiv.suiv` n'existe pas. Comme pour l'ajout, la suppression du maillon suivant celui d'adresse `p` se fait en appelant cette méthode sur l'instance `p` c'est-à-dire : `p.supprimer()`.

La méthode `imprimer` fait un parcours classique et permet de visualiser les résultats.

```
void imprimer(){
   for (Liste p = this; !p.vide(); p = p.suiv)
       System.out.print(p.element() + " ");
   System.out.println();
}
```

La méthode `purger`, comme c'était le cas dans la mise en œuvre par les tableaux, se transcrit directement de l'algorithme à l'aide des méthodes déjà définies. Plutôt que de ne pas renvoyer de résultat, on a choisi ici de renvoyer `this` ce qui ne change pas grand-chose au problème. Voici la version à rajouter dans la classe `Liste` comme méthode d'instance.

```
Liste  purger(){
   Liste  p;

   for(Liste posCour =this; !posCour.vide(); posCour=posCour.suiv){
       p=posCour.suiv;
       while(!p.vide())
           if(p.element() == posCour.element())  p.supprimer();
           else  p=p.suiv;
   }
   return(this);
}
```

Si l'on devait programmer cette méthode à l'extérieur de la classe `Liste`, il s'agirait d'une méthode statique purgeant la liste passée en paramètre. Noter que l'on n'a plus alors accès au champ `l.suiv` qui est privé à la classe `Liste`. D'où l'appel obligé à la méthode `suivant`.

```
static Liste  purger(Liste l){
   Liste  p;

   for(Liste posCour = l; !posCour.vide(); posCour=posCour.suivant()){
       p=posCour.suivant();
       while(!p.vide())
           if(p.element() == posCour.element())
                p.supprimer();
           else
               p=p.suivant();
   }
   return(l);
}
```

On remarque que cette dernière version de la méthode `purger` est à nouveau calquée sur l'algorithme, à ceci près que la condition `posCour != l.fin()` à été remplacée par `!posCour.vide()` pour des raisons d'efficacité.

On peut maintenant clore la classe `Liste` et tester quelques unes de ses méthodes par le petit programme suivant, que l'on a fait suivre d'une trace d'exécution.

```
public class ProgListe{

  public static void main (String [] args) throws IOException {
     int i;
     Liste l = new Liste();

     for(i=0; i<args.length; i++)
         l.rajouter(Integer.parseInt(args[i]));
     System.out.println("Liste saisie"); l.imprimer();
     l.purger();
     System.out.println("Liste purgee"); l.imprimer();
  }
}
```

```
$ java ProgListe 6 8 9 10 9 7 4 6 8 4 8 3 9 1 6 10 6 5
Liste saisie
5 6 10 6 1 9 3 8 4 8 6 4 7 9 10 9 8 6
Liste purgée
5 6 10 1 9 3 8 4 7
```

On notera que l'on a rajouté en tête de liste, les uns après les autres, les éléments passés en arguments du programme : la liste construite est donc dans l'ordre inverse de la donnée. La construction de la liste dans l'ordre des données est l'objet de l'exercice 6.1.

6.1.4 Cas des listes ordonnées

La plupart du temps, le type de base des éléments de la liste est muni d'une relation d'ordre $\leq$, ce qui permet d'ordonner les listes en vue de faciliter leur traitement. On suppose ainsi, dans cette section, que les listes sont ordonnées par ordre croissant. La spécification de la primitive **position** est modifiée de telle sorte que la position renvoyée est celle de l'élément `x` passé en argument si

celui-ci figure dans la liste et, s'il n'y figure pas, celle à laquelle il devrait être rajouté pour que la liste reste ordonnée.

Tableaux et recherche dichotomique

Dans le cas d'une représentation par tableaux, on peut mettre en œuvre une recherche dite *séquentielle*, semblable à celle du cas non ordonné : on débute au premier élément de la liste et l'on s'arrête dès l'obtention d'un élément supérieur ou égal à `x` ou, si `x` est supérieur à tous les éléments de la liste, à la fin de celle-ci. Comme précédemment, cette recherche se programme avec ou sans sentinelle. La version avec sentinelle est la suivante.

```
int position(int x){
        int i;

        T[fin]=x;
        for(i=0; T[i]<x; i++);
        return(i);
}
```

A la sortie de la boucle, soit `T[i] = x` et `i` est bien la position de `x` dans le tableau, soit `i` est le plus petit indice tel que `T[i]>x` et c'est bien la position à laquelle il faudrait l'insérer pour que le tableau reste rangé. Cette méthode est linéaire dans le pire des cas.

La représentation permet toutefois de programmer cette méthode de façon beaucoup plus efficace, grâce à une recherche dite *dichotomique*. Grossièrement, on compare l'élément recherché à celui qui est au milieu de la liste. S'il est inférieur, on le recherche dans la moitié gauche, sinon, on le recherche dans la moitié droite. L'algorithme de recherche dichotomique est ici directement programmé en Java. La méthode renvoie un couple constitué d'une valeur booléenne `ilyEst` indiquant si l'élément a ou non été trouvé, et d'un entier `rang` indiquant :

- soit le rang où l'on a trouvé l'élément,
- soit, si l'élément ne figure pas dans la liste, celui auquel on doit le rajouter pour qu'elle reste ordonnée.

A cette fin, on rajoute au préalable dans la classe `Liste` une classe `boolInt` définissant ce type de couples (cf. section 3.1.2).

```
class boolInt{
   boolean ilyEst; int rang;
   boolInt(boolean b, int r)
       {ilyEst = b; rang = r;}
}
```

La méthode `dichotomie` est alors la suivante :

```
public boolInt dichotomie(int x){
      int premier=0, dernier= fin-1, milieu;

      while(premier<=dernier){
          milieu = (premier+dernier)/2;
          if(x<=T[milieu]) dernier = milieu -1;
          if(x>=T[milieu]) premier = milieu + 1;
      }
      return new boolInt(premier==dernier+2,  dernier+1);
}
```

Cet algorithme, et notamment les deux instructions `if` sans branche `else` ainsi que la condition de contrôle de la boucle `while` (qui provoque des itérations jusqu'à faire se croiser les variables `premier` et `dernier`), peut paraître surprenant. C'est cependant l'une des meilleures versions de la recherche dichotomique.

Correction

On démontre (cf. exercice 6.2) qu'à la sortie de la boucle :
- soit `premier == dernier+2`, dans ce cas `x` est dans la liste à l'indice `dernier+1` du tableau.
- soit `premier == dernier+1`, dans ce cas `x` ne figure pas dans la liste et l'indice auquel on doit le rajouter pour qu'elle reste ordonnée est `dernier+1`.

Ceci explique le résultat renvoyé par la méthode.

Complexité

Le coût de l'algorithme est proportionnel au nombre d'itérations de la boucle `while`. Chacune de ces itérations diminue le nombre d'éléments de la liste dans laquelle s'effectue la recherche. Supposons qu'avant une itération, la recherche se fasse parmi n éléments. Après l'itération, si les valeurs des variables `premier` et `dernier` ne se sont pas croisées, la recherche se fera, si n est pair :

- soit parmi les $\frac{n}{2}$ éléments dont l'indice est dans {`milieu+1, ..., dernier`}
- soit parmi les $\frac{n}{2} - 1$ éléments dont l'indice est dans {`premier, ..., milieu -1`}

Si n est impair, une nouvelle recherche a lieu sur les $\frac{n-1}{2}$ éléments dont l'indice est soit dans {`premier, ..., milieu-1`}, soit dans {`milieu+1, ..., dernier`}.

En conséquence, toute itération sur une liste à n éléments restreint, dans le pire des cas, la recherche à une liste à $\frac{n}{2}$ éléments.

Par suite, si les variables `premier` et `dernier` ne se sont pas croisées, après la $i$ème itération, celle-ci a restreint l'espace de recherche à une liste qui a au plus $\frac{n}{2^i}$ éléments. Il en résulte que la condition de contrôle de la boucle reste vraie pendant au plus `k` itérations avec

$$\frac{n}{2^{k+1}} < 1 \leq \frac{n}{2^k}$$

A ce moment là, il reste un unique élément dans l'espace de recherche et la boucle se termine après une dernière itération. Des inégalités ci-dessus, on déduit successivement les relations suivantes :

$$n < 2^{k+1} \leq 2n, \quad log_2(n) < k+1 \leq 1 + log_2(n), \quad k \leq log_2(n) < k+1, \quad k = \lfloor log_2(n) \rfloor.$$

Le nombre d'itérations est donc borné par $\lfloor log_2(n) \rfloor + 1$. Cette borne est effectivement atteinte quand l'élément est supérieur à tous ceux de la liste, et que $n = 2^k$. Dans ce cas, chaque itération divisera exactement par 2 l'espace de recherche en le restreignant aux éléments dont les indices sont dans {`milieu+1,..., dernier`}.

Proposition 6.1 *La complexité en temps dans le pire des cas de la recherche dichotomique d'un élément parmi* n*, est en* $\Theta(log(n))$.

Application

Le programme suivant utilise (et donc teste) cette méthode. A partir d'une liste d'entiers non ordonnée, avec d'éventuelles répétitions, donnée en argument, il construit une liste croissante sans répétition. La méthode consiste, après avoir créé une liste vide, à procéder à une recherche dichotomique de chaque entier en argument, et à un rajout à l'endroit correct de cet entier s'il n'y est pas déjà.

```
public class ProgListes{

    public static void main(String [] args){
```

```
        int n=args.length;
        Liste l = new Liste(n);

        for(int i=0; i<n; i++){
            int e = Integer.parseInt(args[i]);
            Liste.boolInt resultat = l.dichotomie(e);
            if(!resultat.ilyEst) l.rajouter(e, resultat.rang);
        }
        l.imprimer();
    }
}
```

En voici une trace d'exécution :

```
$java ProgListesTableaux 3 67 1 90 8 7 5 67 78 90 67 67 1 1 1
1 3 5 7 8 67 78 90
```

En supposant que les éléments sont tous distincts, les recherches dans la liste des éléments successifs sont donc, dans le pire des cas, proportionnelles à :

$$0, \log(2), \quad \log(3), \ldots, \ \log(\text{n-1})$$

Le coût total des recherches est donc en $\Theta(\sum_{i=2}^{n-1} log(i)) \ = \ \Theta(nlog(n))$ (cf. REFERENCES).

En ce qui concerne les rajouts, le pire des cas est celui où la liste passée en argument est strictement décroissante. Chaque nouvel entier sera en effet inséré dans la première case du tableau et donc le rajout du $i$ème élément nécessitera $i-1$ décalages vers la droite. Le coût total des insertions sera donc en $\Theta(\sum_{i=1}^{n-1} i) \ = \ \Theta(\frac{n(n-1)}{2}) \ = \ \Theta(n^2)$.

Le coût des recherches est donc négligeable devant le coût des insertions.

Listes chaînées ordonnées

Dans le cas des listes chaînées, faute d'accès direct aux éléments à partir de la seule adresse de la liste, on ne peut utiliser de méthode dichotomique. On met en œuvre, comme dans le cas non ordonné, une recherche dite *séquentielle*.

```
 Liste position (int x){
        Liste p = this;
        while (!p.vide() && p.element()<x) p=p.suiv;
        return p;
}
```

Le pire des cas est ici celui où `x` est supérieur à tous les éléments de la liste : il faut alors parcourir un à un tous les éléments. La recherche reste linéaire en temps.

Voici un programme, suivi d'une trace d'exécution, qui construit la liste ordonnée et sans répétition des entiers donnés en arguments.

```
public class ProgListe{

  public static void main (String [] args) throws IOException {

        int i;
        Liste l = new Liste();
```

```
        for(i=0; i<args.length; i++){
            int n = Integer.parseInt(args[i]);
            Liste p = l.position(n);
            if(p.vide() ||  p.element()!= n)
                p.rajouter(n);
        }
         l.imprimer();
  }
}
```

```
$ java ProgListe 6 8 9 10 9 7 4 6 8 4 8 3 9 1 6 10 6 5
1 3 4 5 6 7 8 9 10
```

Si `p.vide()` prend la valeur `true`, la seconde partie de l'expression de contrôle de l'instruction `if` n'est pas évaluée. Il n'y a donc pas de tentative d'accès à l'élément de position `p` si `p` est la fin de la liste.

6.1.5 Tableaux ou listes chaînées ?

Après la présentation de ces deux types de mises en œuvre, il faut se demander laquelle offrira la meilleure efficacité en temps et le moindre coût en mémoire. La réponse n'est pas simple et dépend en général du problème traité. On peut synthétiser les avantages et les inconvénients de chaque représentation dans le tableau suivant.

	Tableaux	Listes chaînées
Avantages	Accès aux éléments en $\Theta(1)$ Prédécesseur en $\Theta(1)$ Recherche en $\Theta(log(n))$ (si ordonné)	Insertion en $\Theta(1)$ Suppression en $\Theta(1)$ Pas de cellules inutiles
Inconvénients	Insertion en $\Theta(n)$ Suppression en $\Theta(n)$ Encombrement maximal	Recherche en $\Theta(n)$ Fin et prédécesseur en $\Theta(n)$ Taille des maillons

Le point fort des tableaux est la possibilité, en cas d'ensemble ordonné, d'une recherche dichotomique extrêmement rapide. Le point faible en est la lenteur des insertions et des suppressions. De plus, il peut être nécessaire de vérifier que le tableau n'est pas plein avant chaque insertion, ce qui ralentit encore le processus. C'est exactement l'inverse qui se produit avec les listes chaînées qui, faute d'accès direct aux éléments, ne permettent que des recherches séquentielles (dont exponentiellement plus coûteuses que les recherches dichotomiques). En revanche, insertions et suppressions se font en temps constant et la place dont on dispose pour de nouvelles insertions est virtuellement infinie (c'est la mémoire de l'ordinateur). On privilégiera donc l'une ou l'autre des représentations, selon que l'on traite de listes relativement stables dans lesquelles on fait essentiellement des recherches, ou au contraire de listes fluctuantes dans lesquelles on fait de nombreuses modifications.

En ce qui concerne l'espace occupé par la représentation par tableau, il faut remarquer qu'elle est proportionnelle à la taille de la liste maximale. En effet, la suppression d'un élément ne libère pas de mémoire. Ceci n'est pas le cas avec les listes chaînées puisque si un élément est supprimé, le maillon correspondant devient inaccessible et est donc susceptible d'être libéré par le récupérateur de mémoire Java (cf. section REFERENCE), ou explicitement par le programmeur dans d'autres langages. Ceci conforte l'idée de consacrer les tableaux aux listes stables. Il faut toutefois nuancer cette comparaison par le fait que la taille des maillons est supérieure à celle des cases de tableau puisque chacun porte, en plus de l'information, l'adresse du maillon suivant.

6.2 Les piles

6.2.1 Description abstraite

Une *pile* (*stack* en anglais), ou structure FIFO (de l'anglais First In First Out), est une suite gérée de façon particulière. On s'interdit de rajouter ou de supprimer un élément n'importe où dans la liste. Les ajouts et les suppressions se font toujours à la même extrémité, appelée *sommet* de la pile. Il en résulte que le premier élément supprimé (on dit *dépilé*) est le plus récent. Le principe est celui d'une pile d'assiettes : on ne peut rajouter d'assiette ou en prélèver une, qu'au sommet de la pile.

Soient x un élément et p une pile. Les primitives sont les suivantes :

pileVide() : renvoie la pile vide.

dépiler(p) : n'est défini que si la pile p n'est pas vide. Supprime un élément de la pile et renvoie sa valeur.

empiler(x, p) : empile x au sommet de la pile p.

vide(p) : renvoie une valeur booléenne indiquant si la pile p est vide ou pas.

6.2.2 Application à la fonction d'Ackermann

On peut illustrer une utilisation des piles sur l'exemple de la fonction *d'Ackermann*. L'intérêt de cette fonction est purement théorique. Elle a été introduite comme exemple de fonction dont la croissance est supérieure à celle de n'importe quelle fonction *primitive récursive*, c'est-à-dire pouvant être programmée en n'utilisant que des boucles de la forme :

`for(int i=0; i<=n;i++)` ou `for(int i=n; i>=0;i--)` avec $n \in \mathbb{N}$

(on parle alors d'itérations bornées).

La fonction d'Ackermann est la fonction A définie récursivement de la façon suivante :

$A(0,\ n) = n+1 \quad \forall n \in \mathbb{N}$
$A(m,\ 0) = A(m-1,\ 1) \quad \forall m \in \mathbb{N}^*$
$A(m,\ n) = A(m-1,\ A(m,\ n-1)) \quad \forall m \in \mathbb{N}^*, \forall n \in \mathbb{N}^*$

On se propose de trouver un algorithme itératif calculant cette fonction. Pour cela, on débute un calcul à la main sur un exemple. C'est ce qui est fait sur la partie gauche de la figure 6.3, où figurent les premières expressions obtenues en appliquant la définition récursive ci-dessus au calcul de $A(3,\ 4)$.

Algorithme récursif	Algorithme itératif	
	Pile p	n
A(3, **4**) =	3	4
A(2, A(3, **3**)) =	2 3	3
A(2, A(2, A(3, **2**))) =	2 2 3	2
A(2, A(2, A(2, A(3, **1**)))) =	2 2 2 3	1
A(2, A(2, A(2, A(2, A(3, **0**))))) =	2 2 2 2 3	0
A(2, A(2, A(2, A(2, A(2, **1**))))) = ...	2 2 2 2 2	1

FIGURE 6.3 – Déroulement de l'algorithme sur le couple (3, 4)

On s'aperçoit qu'à chaque étape, on est en présence :

– de la liste des premiers arguments en attente. Cette liste a été reproduite sur chaque ligne. Il s'agit en fait d'une pile p, présentée ici horizontalement, dont le sommet est l'élément le plus à droite de la ligne.

– du deuxième argument du dernier appel en date à la fonction A (en gras dans les expressions). Cet entier n figure dans le dernière colonne.

Propriété d'invariance Dans le cas général du calcul de $A(m_0, n_0)$, chaque itération se fera donc en présence d'une pile $p = (p_0, \ldots, p_s)$ (p_s désigne ici le sommet de la pile) et d'une valeur n qui devront satisfaire (et ce sera l'*invariant de la boucle*) la relation :

$$A(m_0, n_0) = A(p_0, A(p_1, \ldots, A(p_s, n)\ldots)) \qquad (I)$$

L'étape suivante détermine de nouvelles valeurs de la pile et de n, satisfaisant à nouveau l'invariant (I). Puisqu'elle consiste à débuter le calcul de $A(p_s, n)$, elle se fait en examinant le sommet p_s de la pile ainsi que la valeur de n. Elle est réalisée d'après les trois conditions de la définition, amenant aux trois cas suivants :

– si $p_s = 0$, $A(p_s, n)$ doit être remplacé par $n + 1$, ce qui revient à supprimer p_s de la pile et incrémenter n.
– si $p_s \neq 0$ et $n = 0$, $A(p_s, n)$ doit être remplacé par $A(p_s - 1, 1)$, ce qui conduit à dépiler p_s, empiler $p_s - 1$ et donner à n la valeur 1.
– si ni p_s ni n ne sont nuls, $A(p_s, n)$ doit être remplacé par $A(p_s - 1, A(p_s, n - 1))$, ce qui conduit à dépiler p_s, empiler successivement $p_s - 1$ et p_s et décrémenter n.

On est alors assuré que l'invariant est conservé.

Initialisations Il faut aussi faire en sorte que l'invariant soit satisfait avant la première itération. Il suffit pour cela d'initialiser n à n_0 et la pile à celle dont l'unique élément est m_0.

Conditions d'arrêt Enfin, il reste à déterminer les conditions d'arrêt de l'algorithme. Le calcul se termine lorsque la pile est vide : cela n'a pu se produire que si à l'étape précédente la pile était constituée d'un unique élément de valeur nulle. D'après la propriété d'invariance, à ce moment là : $A(m_0, n_0) = A(0, n)$. La pile a ensuite été vidée et n incrémenté. La valeur de n est alors le résultat recherché et doit donc être renvoyée.

On en déduit l'algorithme suivant :

```
Ackermann(m, n)
   p ← pileVide()
   empiler(m, p)
   faire
      m ← depiler(p)
      si (m=0)
         n ← n+1
      sinon
         si (n=0)
            empiler(m-1, p)
            n ← 1
         sinon
            empiler(m-1, p)
            empiler(m, p)
            n ← n-1
   tant que non vide(p)
   renvoyer(n)
```

Terminaison Si l'on démontre que l'algorithme résursif induit par la définition de la fonction d'Ackermann se termine, on pourra en déduire que l'algorithme itératif se termine aussi. En effet, il y a une correspondance bijective entre les appels récursifs en cours à une étape donnée de

l'exécution de l'algorithme récursif et l'état de la pile p et de la variable n, à la même étape de l'algorithme itératif, comme l'indique la figure 6.3.

En examinant l'algorithme récursif, on remarque que le calcul de $A(m, n)$ provoque des appels récursifs de la forme
- soit $A(m-1,\ 1)$ si $n = 0$
- soit $A(m-1,\ x)$ et $A(m,\ n-1)$ sinon.

Un appel sur un couple $(m,\ n)$ fait donc un ou deux rappels récursifs sur des couples $(m',\ n')$ tels que :
- soit $m > m'$
- soit ($m = m'$ et $n > n'$).

On est ainsi amené à introduire la définition suivante.

Définition 6.2 *On appelle ordre lexicographique sur $\mathbb{N} \times \mathbb{N}$ la relation $>_{lex}$ définie par :*

$$\forall (m,\ n), (m',\ n') \in \mathbb{N} \times \mathbb{N} \quad (m,\ n) >_{lex} (m',\ n') \Leftrightarrow (m > m') \text{ ou } (m = m' \text{ et } n > n')$$

Proposition 6.3 *La relation lexicographique est une relation transitive* bien fondée.

Démonstration La démonstration de la transitivité est laissée au lecteur.
Montrons que la relation est bien fondée. D'après la définition 5.1, il suffit de montrer qu'il n'existe pas de suite infinie strictement décroissante. On raisonne par l'absurde en supposant qu'il existe une suite infinie strictement décroissante de couples d'entiers naturels tels que :

$$c_0 \ >_{lex} \ c_1 \ >_{lex} \ \ldots \ >_{lex} \ c_k \ \ldots$$

avec $c_i = (m_i,\ n_i)$. D'après la définition de la relation $>_{lex}$, la suite des premières coordonnées de ces couples est nécessairement telle que :

$$m_0 \ \geq \ m_1 \geq \ \ldots \ \geq m_k \ \ldots$$

Puisqu'il n'existe pas de suite infinie strictement décroissante d'entiers naturels, il existe donc une infinité de termes de cette suite tous égaux. Appelons a leur valeur commune. On peut donc extraire une sous-suite $(m_{\varphi(k)})_{k \in \mathbb{N}}$ constante et égale à a :

$$a = \ m_{\varphi(0)} = \ m_{\varphi(1)} \ = \ \ldots \ = m_{\varphi(k)} = \ \ldots$$

Mais alors, par transitivité, la suite extraite $(c_{\varphi(k)})_{k \in \mathbb{N}}$ est une suite infinie de couples strictement décroissante :

$$c_{\varphi(0)} >_{lex} c_{\varphi(1)} >_{lex} \ldots >_{lex} c_{\varphi(k)} = \ \ldots$$

Puisque ces couples ont tous pour première composante a, on en déduit que :

$$n_{\varphi(0)} > \ n_{\varphi(1)} \ > \ \ldots \ n_{\varphi(k)} > \ \ldots$$

On a ainsi obtenu une suite strictement décroissante d'entiers naturels, ce qui est absurde. □

La terminaison de l'algorithme `Ackermann` découle alors immédiatement de cette proposition et du théorème 5.2.

6.2.3 Mise en œuvre par des tableaux

Dans ce type de représentation, les piles sont caractérisées par un tableau et l'indice du dernier élément. Le dernier élément est ici le sommet de la pile, et son indice a été appelé `top`.

Dans l'implantation ci-dessous, la taille maximale de la pile peut être définie par l'utilisateur comme paramètre d'un constructeur. Elle peut également être déterminée par défaut au moyen d'une constante `MAX` fixée par le concepteur de la classe `Pile`. On obtient ainsi :

```
public class Pile{
    public static final MAX=300000;
    private int top;
    private int[]T;

    public Pile(int n){
        top=-1;
        T = new int[n];
    }

    public Pile(){
        top=-1;
        T = new int[MAX];
    }

    public void empile(int x) {T[++top] = x;}
    public int depile() {return(T[top--]);}
    public boolean vide() {return(top == -1);}
    public boolean pleine() {return(top == T.length-1);}
}
```

On remarquera l'opportunité d'une primitive **pleine** dans ce type de représentation, puisque la taille dont on dispose étant limitée, l'utilisateur pourra s'assurer que la pile n'est pas pleine avant d'empiler un nouvel élément.

6.2.4 Mise en œuvre par des listes chaînées

La mise en œuvre est un cas particulier de la représentation des listes quelconques par des listes chaînées. Pour des raisons d'efficacité, rajouts et suppressions se font en tête de la liste. Les méthodes `empiler` et `depiler` sont donc identiques aux méthodes `rajouter` et `supprimer` de la classe implémentant les listes chaînées.

```
public class Pile{

    private int elt;
    private Pile suiv;

    public Pile(){suiv=null;}

    public Pile(int n, Pile s){
        elt = n;
        suiv = s;
    }

    public boolean vide(){return(suiv==null); }

    public void empile(int n){suiv = new Pile(n,suiv);}

    public int depile(){
        int aux = suiv.elt;
        suiv = suiv.suiv;
        return(aux);
    }
}
```

6.2.5 Application à la fonction d'Ackermann

La fonction d'Ackermann peut maintenant être programmée itérativement en utilisant indifféremment l'une ou l'autre des classes définissant les piles. Il faut remarquer que le programme ci-dessous est indépendant de la mise en œuvre choisie.

```
public class Ackermann{

    public static int Ack(int m, int n){
       Pile P = new Pile();

       P.empile(m);
       do{
          m=P.depile();
          if (m==0)
            n++;
          else
             if (n==0){
               P.empile(m-1);
               n=1;
             }
             else{
               P.empile(m-1);
               P.empile(m);
               n--;
             }
        }
        while (!(P.vide()));
        return(n);
    }

    public static void main(String[] args){

       if (args.length!=2)
          System.out.println("La fonction attend 2 entiers en arguments");
       else{
          System.out.println("A(" + args[0] + ", " + args[1]+") = " +
             Ack(Integer.parseInt(args[0]), Integer.parseInt(args[1])));
          }
    }
}
```

La difficulté est ici de prévoir la taille de la pile dans le cas d'une représentation par tableau. De toutes façons, la croissance de la fonction calculée est telle, que l'on dépasse rapidement les limites raisonnables en temps, et les limites physiques en mémoire et dans le codage des entiers.

6.3 Les files d'attente

6.3.1 Description abstraite

Tout comme les piles, les files d'attente (*queue* en anglais) sont des listes gérées de façon particulière. Insertions et suppressions se font chacunes aux deux extrémités opposées. Les insertions se font à la fin et c'est le premier élément qui est supprimé. En conséquence, c'est l'élément le plus ancien qui est sorti de la file à chaque suppression, d'où le nom de structure FIFO (de l'anglais

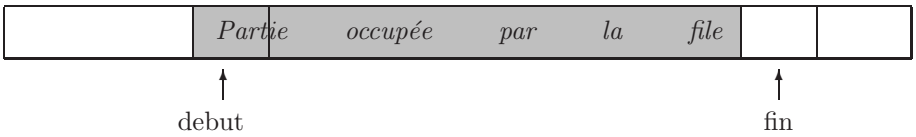
First In First Out). C'est exactement le fonctionnement des files d'attente au sens commun du terme. Les primitives du type abstrait sont les suivantes :

debut(f) : renvoie la position du premier élément de la file
fin(f) : renvoie la position de la fin de la liste
fileVide() : renvoie une file vide
vide(f) : renvoie une valeur booléenne indiquant si la file f est vide ou pas.
enfiler(x, f) : rajoute un élément à la file.
defiler(f) : défini si f n'est pas vide. Supprime l'élément le plus ancien et renvoie sa valeur.

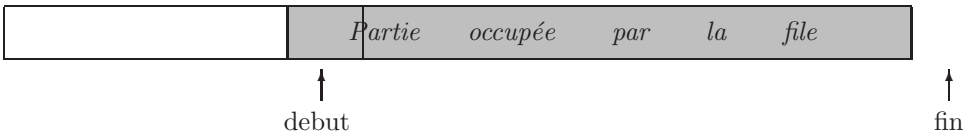
6.3.2 Mise en œuvre par des tableaux

Comme toujours dans ce type de représentation, le tableau destiné à recevoir la file n'est pas nécessairement entièrement occupé, puisque la taille de la file est supposée varier avec les insertions et suppressions. Si l'on commence par insérer les éléments à partir de la première case du tableau, on se retrouvera après un certain nombre d'insertions dans la situation suivante :

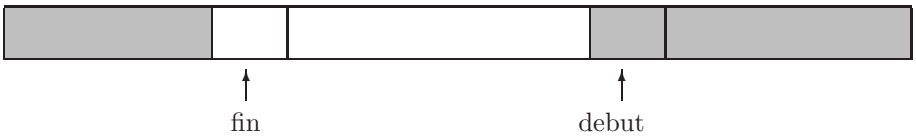
Les suppressions se faisant au début, on pourrait être tenté de décaler pour ce faire, tous les éléments restants vers la gauche : ces opérations se feraient alors en $\Theta(n)$. On peut gagner en efficacité et obtenir des suppressions en $\Theta(1)$ en remarquant qu'il suffit d'incrémenter l'indice *debut* pour ôter un élément de la file. Ce faisant, la file se déplace avec le temps vers la droite du tableau.



Avec un certain nombre d'insertions et de suppressions, le dernier élément de la file finit par se retrouver à la fin du tableau, ce qui rend impossible tout nouvel ajout en position *fin*.



On pourrait envisager à ce moment là de décaler toute la file pour la refaire débuter à la première case du tableau. Cette opération coûteuse peut à son tour être évitée en poursuivant les insertions dans les cases libres situées au début du tableau. Ceci revient à chaque insertion et chaque suppression à incrémenter les indices *fin* et *debut* modulo le nombre d'éléments du tableau. On peut alors ainsi se retrouver dans la situation suivante :



De ce fait, le tableau peut être mentalement représenté comme un anneau sur lequel la file se déplace vers la droite. C'est pour cela que les files d'attente ainsi codées sont appelées *files circu-*

laires.
Le dernier point délicat est la détection d'une file *pleine* ou d'une file *vide*. Une file pleine occupe tout le tableau. Si c'est le cas, la fin rejoint le premier de la file et une nouvelle insertion écraserait l'élément le plus ancien. Mais si la file est vide, les suppressions antérieures des éléments de rang *debut* ont successivement incrémenté *debut* qui a ainsi rejoint la fin. L'égalité entre la fin et le debut de la file pouvant survenir à la fois si la file est pleine et si la file est vide, cette condition ne peut caractériser aucune de ces situations à elle seule. Une solution consiste à introduire un champ de type entier dans lequel est enregistré le nombre d'éléments de la file et qui est mis à jour à chaque modification de la file. On obtient ainsi :

```
public class Files {

    public static final int MAX=100;

    private int[] T;
    private int debut, fin, nombre;

    public Files (){
        T = new int[MAX];
        fin=nombre=debut=0;
    }

    public Files (int n){
        T = new int[n];
        nombre=debut=fin=0;
    }

    public boolean vide() {return(nombre==0);}
    public boolean pleine() {return(nombre==T.length);}
    public int taille(){return nombre;}

    public int defile(){
        int aux;

        aux=T[debut];
        debut = (debut+1)% MAX;
        nombre--;
        return(aux);
    }

    public void enfile(int x){
        nombre++;
        T[fin] = x;
        fin = (fin+1)%MAX;
    }
}
```

La méthode imprime permet de visualiser le contenu de la file

```
    public void imprime(){
        int i = debut;

        for(int t= nombre; t!=0;t--){
            System.out.print(T[i]+" ");
            i= (i+1)%MAX;
```

```
        }
        System.out.println();
    }
```

Voici un exemple d'utilisation de cette classe.

```
import java.util.*;
public class ProgFile{

    public static void main(String[] args){
        Files f = new Files();
        int i;
        Scanner sc  = new Scanner(System.in);

        if (args.length > Files.MAX)
             System.out.println("Dépassement de capacité");
        else{
             for(i=0; i<args.length;i++)
                 f.enfile(Integer.parseInt(args[i]));
             System.out.println(" J'ai lu :");
             f.imprime();

             System.out.println("Combien d'elements voulez-vous supprimer?");
             i= sc.nextInt();
             if(i>f.taille()){
                 System.out.print("On ne peut supprimer plus de ");
                 System.out.println(f.taille()+ " elements");
             }
             else{
                 for(; i>0; i--) f.defile();
                 f.imprime();
             }
         }
    }
}
```

On remarquera comment les tests `f.pleine()` avant chaque ajout et `f.vide()` avant chaque suppression ont ici pu être évités.

6.3.3 Mise en œuvre par des listes chaînées

Dans toute liste chaînée, l'accès au premier élément se fait en temps constant, tandis que l'accès au dernier se fait en temps linéaire. C'est ainsi que pour les piles, on a choisi de faire figurer le sommet en premier élément de la structure chaînée : insertions et suppressions se font alors en temps constant.
En revanche, pour les files, il est nécessaire de travailler à la fois sur les deux extrémités : au début pour les suppressions, à la fin pour les insertions, ou l'inverse. Pour éviter de recalculer la fin de la liste à chaque insertion par exemple, il convient de la mémoriser dans une variable mise à jour à chaque ajout. Toutefois, une déclaration analogue à celle de la classe `liste`, avec le rajout d'un champ supplémentaire `fin` comme suit :

```
public class Files{
  private int element;
  private Files suiv, fin;
}
```

est particulièrement maladroite : en effet chaque maillon est alors constitué de trois champs : celui portant l'information, celui portant l'adresse du maillon suivant et celui portant l'adresse de la fin ; la fin serait donc dupliquée autant de fois qu'il y a de maillons dans la liste ! Ce n'est assurément pas la configuration souhaitée. La solution consiste à définir une classe `maillon` privée à la classe `Files` permettant de définir des maillons à deux champs, puis de définir une file comme un couple de maillons : celui d'en-tête et celui de fin. On obtient ainsi :

```
public class Files{

    private class maillon{
        private int element;
        private maillon suiv;

        maillon (int n, maillon s){element=n; suiv=s;}
        maillon(){suiv=null;}
    }

    private maillon debut;
    private maillon fin;
```

Les constructeurs de la classe `maillon` sont identiques à ceux de la classe `Liste`. Comme c'est le cas pour les suites représentées par des listes chaînées, une file vide est constituée d'un seul maillon, l'*en-tête*, qui dans ce cas là désigne à la fois le début et la fin. D'où le constructeur créant une file vide :

```
    public Files(){
        fin = new maillon();
        fin = debut;
    }
```

Le fait que le début et la fin soient confondus caractérisent ainsi les files vides.

```
    public boolean vide(){return(fin==debut);}
```

La suppression est analoque à la suppression du premier élément d'une liste, à ceci près qu'il convient de mettre à jour la variable `fin` si l'élément supprimé était le dernier, c'est-à-dire si la file ne comportait plus qu'un seul élément.

```
    public int defile(){
        int aux = debut.suiv.element;

        if(fin==debut.suiv) fin=debut;
        debut.suiv = debut.suiv.suiv;
        return(aux);
    }
```

Le rajout d'un élément se fait comme annoncé à la fin et nécessite donc une mise à jour de la variable `fin`.

```
    public void enfile(int n){

        fin.suiv= new maillon(n, fin.suiv);
        fin=fin.suiv;
    }
```

Enfin, la méthode `imprime` ci-après permet d'illustrer un parcours de la file.

```
    public void imprime(){
        maillon m;

        for(m=debut; m!=fin; m=m.suiv)
            System.out.print(m.suiv.element+" ");
        System.out.println();
    }
}
```

Le programme ci-dessous illustre le fonctionnement de cette classe, très similaire à celui de la classe de la section précédente.

```
import java.util.*;
public class ProgFile{

    public static void main(String[] args){
        Files f = new Files();
        int i;
        Scanner sc  = new Scanner(System.in);

        for(i=0; i<args.length;i++)
            f.enfile(Integer.parseInt(args[i]));
        System.out.println(" J'ai lu :");
        f.imprime();

        System.out.println("Combien d'elements voulez-vous supprimer?");
        i= sc.nextInt();
        for(; i>0; i--)
           if(!f.vide()) f.defile();
        f.imprime();
    }
}
```

On remarque qu'il est inutile de s'assurer que la file n'est pas pleine avant les insertions. En ce qui concerne les suppressions, on aurait pu dans la cas présent éviter les appels à la méthode `vide` avant chaque suppression en comparant i et $args.length$ avant la dernière boucle `for`.

6.4 Exercices

Exercice 6.1 *Modifier le programme* `ProgListes` *de la section 6.1.3, de façon à préserver l'ordre des arguments, en évitant toutefois de reparcourir toute la liste à chaque nouvel ajout.*

Exercice 6.2

1. *Faire tourner l'algorithme de recherche dichotomique sur plusieurs exemples, notamment dans les cas où* `x` *est dans la liste (à une extrémité ou pas), où* `x` *est entre deux éléments de la liste, où* `x` *est supérieur ou inférieur à tous les éléments de la liste.*
2. *On peut sans changer le problème, supposer que le tableau commence et se termine par deux cases fictives ; la première,* `T[-1]` *contiendrait un élément strictement inférieur à tous les éléments de E, noté par exemple* $-\infty$ *; l'autre* `T[fin]` *contiendrait un élément strictement supérieur à tous les éléments de E, noté* $+\infty$*. Montrer que la proposition* I_1 *ou* I_2*, avec :*

   I_1 *: T[premier-1]* $< x <$ *T[dernier +1]*
   I_2 *: (x = T[premier-1]=T[dernier+1]) et (premier = dernier + 2)*

   est un invariant de la boucle.

3. En déduire que l'algorithme satisfait sa spécification.

Exercice 6.3 *Modifier le programme calculant la fonction d'Ackermann, pour faire afficher, à chaque itération, le contenu de la pile. On traitera les deux implémentations des piles.*

Exercice 6.4 *Donner une mise en œuvre des listes par des structures doublement chaînées, c'est-à-dire pour lesquelles chaque maillon porte à la fois l'adresse du suivant et du précédent. On fera en sorte que l'accès à la fin se fasse en temps constant.*

Chapitre 7

Structures arborescentes

7.1 Présentation informelle

Un exemple bien connu Les arbres sont des structures bi-dimensionnelles, dont l'exemple le plus connu est celui des arbres généalogiques, comme illustré par la figure 7.1. Les arbres sont représentés la racine en haut et les feuilles en bas.

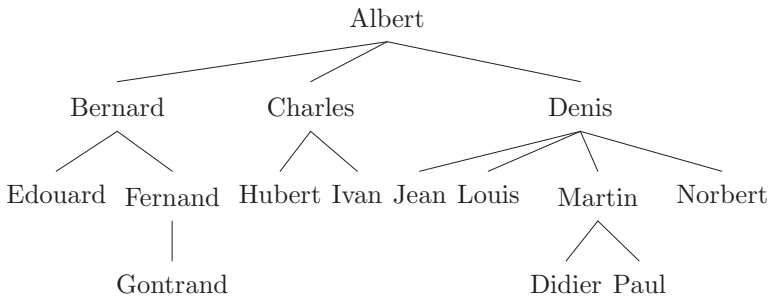


FIGURE 7.1 – Un exemple d'arbre : l'arbre généalogique

Les arbres sont destinés à représenter un ensemble de données, et donc, comme pour les listes, on convient de définir un `arbre vide` qui représentera l'ensemble vide. La terminologie est en partie empruntée au domaine de la généalogie. Tout arbre non vide a possède des *nœuds*. Ce sont ici les individus. La racine est un nœud particulier (ici *Albert*), qui n'a pas de parent. Chaque nœud est à son tour racine d'un arbre qui est appelé *sous-arbre* de l'arbre global. Un sous-arbre de a autre que a est appelé sous-arbre *propre* de a. Les racines des sous-arbres issus d'un nœud sont appelés les *fils* du nœud. Tout nœud est le *père* de ses fils. Une *feuille* est un nœud qui n'a pas de fils. Un *chemin* est une suite de nœuds chacun étant le fils du précédent. Une *branche* est un chemin dont le premier élément est la racine, et le dernier une feuille. La *taille* d'un arbre est le nombre de ses nœuds. La *hauteur* d'un arbre est le nombre d'éléments de la plus longue branche, diminué de 1. Ainsi, un arbre de hauteur 0 est réduit à un élément.

Des structures essentiellement récursives Si les listes se traitent de façon séquentielle, la récursivité est particulièrement bien adaptée au traitement des arbres. Ainsi, les arbres peuvent-ils être aisément caractérisés par une définition récursive comme suit.
Un arbre sur un ensemble D est :

- *soit* vide

– *soit un couple constitué d'un élément de* D *(la* racine*) et d'une suite d'arbres (ses* sous-arbres immédiats*, ou par abus de langage ses* fils *).*

La plupart des algorithmes sur les arbres se rappellent récursivement sur les fils, le cas terminal étant celui de l'arbre vide. La terminaison des algorithmes est assurée par le fait que les rappels récursifs ont lieu sur des arbres de taille moindre. Ainsi, les relations suivantes permettent de calculer récursivement la taille d'un arbre a :

– *si* a *est vide, sa taille est nulle*
– *si* a *n'est pas vide sa taille est la somme des tailles de ses fils, augmentée de 1.*

Des structures exprimant des hiérarchies L'aspect bi-dimensionnel des arbres permet de représenter des hiérarchies qui ne peuvent être prises en compte par des structures à une dimension. Un exemple classique est celui des expressions arithmétiques, comme $(3y + x)(4 + (2 - 3x)/y)$ représentée par l'arbre de la figure 7.2. On remarquera que l'emploi de parenthèses, nécessaire pour rendre compte des priorités des opérations dans l'expression écrite selon l'usage comme une suite de symboles, est inutile dans l'écriture sous forme d'arbre. Les priorités sont exprimées par la structure de l'arbre : les deux opérandes sont déterminés sans ambiguïté respectivement par les fils gauches et les fils droits des opérateurs.

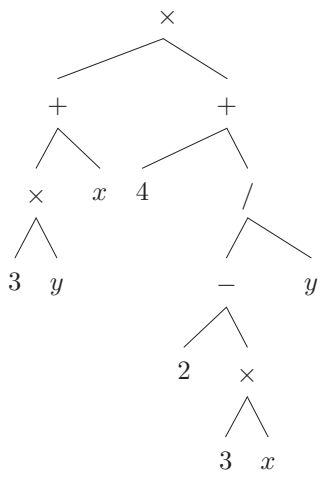


FIGURE 7.2 – Structure de l'expression $(3y + x)(4 + (2 - 3x)/y)$

On remarque que les nœuds de cet arbre :

– soit n'ont pas de fils,
– soit en ont deux.

On dit que l'arbre est un arbre binaire. Il s'agit là d'un cas particulier important de structures arborescentes, auquel est consacrée la section qui suit.

7.2 Les arbres binaires

7.2.1 Description abstraite

Definition 1 (Arbre binaire) *Un arbre binaire sur un ensemble* D *est :*

– *soit un élément particulier appelé* arbre vide
– *soit un triplet* (r, sag, sad) *où* r *est un élément de D appelé* racine *de l'arbre et* sag *et* sad *sont des arbres binaires sur D appelés respectivement sous-arbre gauche et sous-arbre droit.*

On peut alors introduire les définitions élémentaires suivantes.

Definition 2 (Feuille) *Un arbre binaire dont les sous-arbres gauche et droit sont vides est appelé feuille.*

Definition 3 (Nœuds) *Si* a *est un arbre non vide, un nœud de* a *est :*
- *soit sa racine*
- *soit un nœud de son sous-arbre gauche*
- *soit un nœud de son sous-arbre droit*

Definition 4 (Sous-arbres) *Si* a *est un arbre non vide, un sous-arbre de* a *est :*
- *soit* a *lui-même*
- *soit, si* a *n'est pas vide, un sous-arbre de son sous-arbre gauche ou de son sous-arbre droit*

Les primitives

Soit D un ensemble. Dans la définition qui suit, a, g et d désignent des arbres binaires sur D, et r un élément de D.

ab(r, g, d) : renvoie l'arbre binaire de racine r, de sous-arbre gauche g et de sous-arbre droit d.
arbreVide : désigne l'arbre vide.
vide(a) : renvoie une valeur booléenne indiquant si a est l'arbre vide ou non.
racine(a) : n'est défini que si a n'est pas vide. Renvoie la racine de a.
sag(a) : n'est défini que si a n'est pas vide. Renvoie le sous-arbre gauche de a.
sad(a) : n'est défini que si a n'est pas vide. Renvoie le sous-arbre droit de a.
feuille(a) : renvoie vrai si a est une feuille.

7.2.2 Les divers parcours d'arbres binaires

Parcourir un arbre binaire consiste à visiter chacun de ses nœuds. On donne ici le schéma le plus général d'un parcours gauche-droite en profondeur d'abord. Cela signifie que lorsque l'on arrive sur un nœud de l'arbre pour la première fois, on commence par visiter tous les nœuds de son sous-arbre gauche, avant de visiter tous ceux de sous-arbre droit. Lors d'un tel parcours, comme illustré par la figure 7.3, chaque nœud est visité trois fois :

1. en arrivant sur le nœud pour la première fois, en descente, juste après avoir visité son père,
2. quand on y arrive en remontant après avoir parcouru son sous-arbre gauche et avant de redescendre à droite,
3. quand on y remonte pour la dernière fois après avoir parcouru son sous-arbre droit.

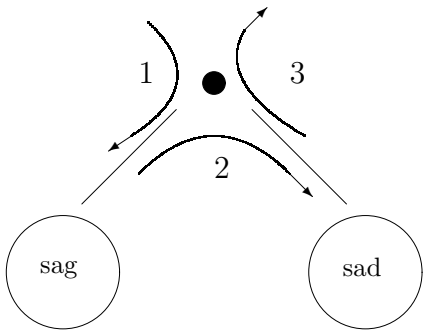


FIGURE 7.3 – Les visites d'un nœud lors d'un parcours gauche-droite en profondeur

A chacune de ces trois visites, le nœud peut faire l'objet de traitements qui dépendent du problème envisagé, comme mis en lumière dans les exemples introduits par la suite. Sachant que parcourir un arbre vide consiste à ne rien faire, on obtient l'algorithme du parcours gauche-doite en profondeur de la figure 7.4.

```
parcours(a)
si (non vide(a))
    traitement 1
    parcours(sag(a))
    traitement 2
    parcours(sad(a))
    traitement 3
```

FIGURE 7.4 – Parcours général d'un arbre binaire

On remarque que le schéma de la figure 7.3 reste valable même si le nœud est une feuille : on "remonte" immédiatement après la descente dans chaque fils, après avoir constaté qu'ils sont vides. Pour une feuille, les trois traitements sont donc consécutifs.

Ce schéma de parcours est le plus général. On distingue cependant trois modes de parcours particuliers :

L'ordre préfixe ou préordre : seul le traitement 1 est effectué. Un nœud est traité avant tous ses descendants.

L'ordre infixe ou inordre : seul le traitement 2 est effectué. Un nœud est traité après tous ceux de son sous-arbre gauche et avant tous ceux de son sous-arbre droit.

L'ordre postfixe ou postordre : seul le traitement 3 est effectué. Un nœud est traité près tous ses descendants.

Si le traitement consiste à imprimer le nœud, on obtient pour l'arbre de la figure 7.2 les résultats suivants :

Préordre : $\times\ +\ \times\ 3\ y\ x\ +\ 4\ /\ -\ 2\ \times\ 3\ x\ y$.
Inordre : $3\ \times\ y\ +\ x\ \times\ 4\ +\ 2\ -\ 3\ \times\ x\ /\ y$
Postordre : $3\ y\ \times\ x\ +\ 4\ 2\ 3\ x\ \times\ -\ y\ /\ +\ \times$

Les résultats obtenus par les parcours en préordre et postordre de l'expression arithmétique sont appelés respectivement notation polonaise et notation polonaise inverse de l'expression. On peut se convaincre que ces notations, qui consistent à écrire l'opérateur soit *avant* soit *après* ses deux opérandes évitent l'emploi des parenthèses : on peut en effet reconstruire l'expression habituelle à partir de l'une ou l'autre des notations qui n'induisent aucune ambiguïté sur l'ordre des opérations. La notation polonaise inverse fut jadis utilisée par certaines calculatrices.

```
prefixe(a)                  infixe(a)                   postfixe(a)
si (non vide(a))            si (non vide(a))            si (non vide(a))
    traitement                  infixe(sag(a))              postfixe(sag(a))
    prefixe(sag(a))             traitement                  postfixe(sad(a))
    prefixe(sad(a))             infixe(sad(a))              traitement
```

FIGURE 7.5 – Les trois ordres de parcours des arbres binaires

Si l'on souhaite afficher l'expression arithmétique représentée sur la figure 7.2 sous la forme habi-

tuelle, un parcours en inordre est insuffisant, puisqu'il n'exprime pas la hiérarchie des calculs. On est donc conduit à réintroduire un parenthésage : chaque sous-arbre représentant une expression, on affichera une parenthèse ouvrante avant le parcours de ce sous-arbre, et une parenthèse fermante juste après. On est ainsi assuré d'un parenthésage correct. Toutefois, lorsque le sous-arbre est réduit à une feuille, le parenthésage, bien que correct, est lourd et superflu.

Toutes ces considérations amènent à une aménagement du parcours général (fig.7.4), le traitement 1 consistant à ouvrir une parenthèse, le traitement 2 à afficher le nœud, et le traitement 3 à fermer une parenthèse, sauf dans le cas des feuilles pour lesquelles les traitements 1 et 3 sont supprimés. On obtient ainsi l'algorithme récursif suivant :

```
BienParenthéser(a)
si (non vide(a))
   si (feuille(a))
      afficher(racine(a)))
   sinon
      afficher("(")
      BienParenthéser(sag(a))
      afficher(racine(a))
      BienParenthéser(sad(a))
      afficher(")")
```

On obtient ainsi, pour l'exemple de la figure 7.2 : $(((3 \times y) + x) \times (4 + (2 - ((3 \times x) / y))))$. Bien sûr, des analyses plus fines, prenant en compte les priorités des divers opérateurs, permettraient d'alléger encore cette écriture.

Enfin, si la récursivité permet une expression aisée, claire, sûre et synthétique des divers parcours, il est toutefois possible d'en donner une forme itérative. L'idée est la suivante. Lorsque, lors d'un parcours, on visite un nœud trois actions sont possibles : soit remonter, soit descendre à gauche, soit descendre à droite. On se trouve alors face à deux problèmes :

1. Comment "remonter", puisqu'à partir d'un sous-arbre il est impossible de connaître son père ?
2. Quelle action choisir parmi les trois possibles ?

Le premier problème se résoud selon le principe du *fil d'Ariane* : lorsque l'on descend sur une branche de l'arbre à partir de la racine, on mémorise les sous-arbres sur lesquels on passe, pour pouvoir y remonter ultérieurement, après les avoir quittés pour s'enfoncer plus profondément dans l'arbre. Le fil d'Ariane n'est autre qu'une pile, qui contient exactement tous les ancêtres du sous-arbre courant dans l'ordre dans lequel on les a rencontrés, le sommet de la pile étant le dernier ancêtre en date, c'est-à-dire le père. Il faut donc empiler en descente, et dépiler pour remonter.

Le deuxième problème se résoud en définissant un ensemble de trois *états*, caractérisant les trois visites d'un même sous-arbre décrites sur la figure 7.3. Soit donc l'ensemble

$$Etat = \{D,\ RG,\ RD\}$$

où D signifie *descente*, RG signifie *remontée gauche*, et RD *remontée droite*. Par suite, au lieu d'empiler simplement les sous-arbres sur lesquels on passe, on les accompagne d'un état qui indiquera, lors de la prochaine visite, s'il s'agit d'une remontée gauche ou d'une remontée droite.

Par suite, à chaque itération, on est en présence du sous-arbre courant a, d'un état e, et d'une pile constituée de couples de la forme : *(arbre, état)*. Toutes ces données permettent alors de décider de l'action à mener.

Il reste enfin à déterminer la condition de sortie de la boucle. Le fait que la pile soit vide n'est pas

suffisante, puisque cette propriété signifie que le nœud courant n'a pas d'ancêtre, donc que c'est la racine. Cette condition est satisfaite trois fois, à chaque visite de la racine de l'arbre. En fait, le parcours est terminé quand la pile et vide (on est sur la racine) et l'état est *RD* (on y est pour la troisième et dernière fois), et l'on doit alors effectuer le traitement 3. On obtient ainsi l'algorithme itératif suivant :

```
ParcoursIteratif(a)
e: état, p: pile de couples (arbre, état)

si (non vide(a))
   p ← pileVide()
   e ← D
   faire
      cas e :
         D :  traitement 1
              si (non vide(sag(a))
                 empiler((a, RG), p)
                 a ← sag(a)
              sinon
                 e ← RG
         RG : traitement 2
              si (non vide(sad(a))
                 empiler((a, RD), p)
                 a ← sad(a)
                 e ← D
              sinon
                 e ← RD
         RD : traitement 3
              (a,e)← dépiler(p)
   tant que (non (vide(p) et e = RG))
   traitement 3
```

7.2.3 Mise en œuvre en Java

La classe `AB` définissant en Java les arbres binaires comporte trois champs, chacun correspondant à un élément du triplet constituant un arbre binaire non vide. Contrairement aux listes, le problème de la représentation de l'arbre vide n'a pas ici de solution simple. C'est donc par `null` que l'on convient de le représenter, ce qui revient à confondre les notions d'abscence d'arbre et d'arbre vide. La primitive **vide(a)** ne peut faire l'objet d'une méthode, le test à vide étant nécessairement extérieur à la classe `AB`. On obtient ainsi :

```
public class AB{
    private int r;
    private AB g, d;

    AB  (int e, AB gg, AB dd){
        r = e;
        g = gg;
        d = dd;
    }

  int racine() {return r;}
  AB sag() {return g;}
  AB sad() {return d;}
  boolean feuille() {return (g==null) && (d==null);}
```

Le parcours récursif d'un arbre s'achevant sur l'arbre vide ne peut donc être programmé comme méthode d'instance puisque l'arbre vide n'est pas une instance. L'affichage en in-ordre d'un arbre binaire est ainsi programmé par une méthode de classe comme suit :

```
private static void affiche_inordre(AB a){

    if(a!=null){
        affiche_inordre(a.g);
        System.out.print(a.r + " ");
        affiche_inordre(a.d);
    }
}
```

On peut toutefois en déduire une méthode d'instance comme suit :

```
void affiche_inordre(){
    affiche_inordre(this);
}
```

Toutes les méthodes nécessitant un parcours peuvent se programmer selon ce schéma, où la méthode d'instance utilise une méthode de classe, comme par exemple la méthode ci-après, calculant la taille de l'instance.

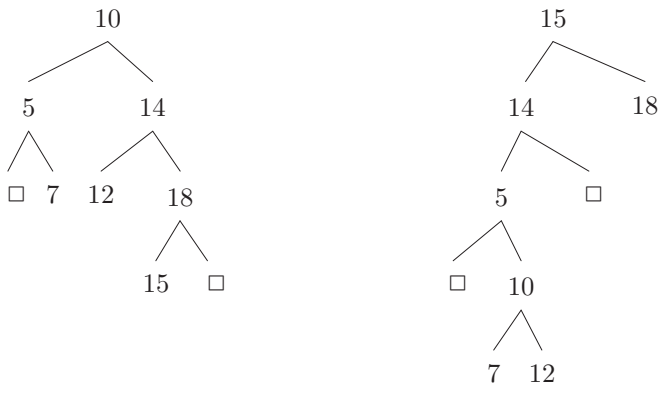
```
private static int size(AB a){
        if(a==null)return 0;
        return(size(a.g)+size(a.d)+1);
    }

int size(){return(size(this)); }
```

7.3 Arbres binaires de recherche (ABR)

Dans cette section, on considère des arbres binaires sur un ensemble totalement ordonné.

Définition 7.1 *Un arbre* a *est un arbre binaire de recherche (ABR) :*
- *soit si* a *est l'arbre vide*
- *soit si, pour chacun des sous-arbres* b *de* a*, la racine de* b *est supérieure à tous les nœuds du sous-arbre gauche de* b *et inférieure à tous les nœuds du sous-arbre droit de* b.

Les deux arbres ci-dessous, dans lesquels le symbole □ figure l'arbre vide, sont des arbres binaires de recherche sur les entiers naturels, représentant tous deux l'ensemble {5, 7, 10, 12, 14, 15, 18}.



Les arbres binaires de recherche permettent de représenter des dictionnaires dans lesquels les recherches peuvent se faire efficacement. En effet, pour rechercher un élément x dans un ABR, il suffit de le comparer à la racine : s'il ne lui est pas égal, on le recherche soit dans le sous-arbre gauche soit dans le sous-arbre droit selon qu'il lui est inférieur ou supérieur. On conçoit que si l'arbre est relativement équilibré, c'est-à-dire tel que les sous-arbres gauche et droit de chaque nœud sont à peu près de même taille, on divise par deux l'espace de recherche après chaque comparaison, ce qui conduit, comme pour la recherche dichotomique, à une complexité en temps logarithmique.

La gestion de tels dictionnaires consiste donc en trois algorithmes, recherche, suppression, insertion, les deux dernières opérations devant préserver la structure d'ABR.

7.3.1 Recherche

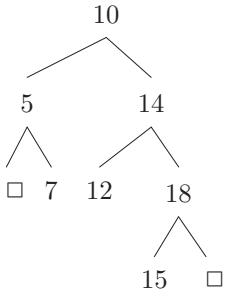
L'algorithme suivant recherche l'élément `x` dans l'arbre `a`. Il renvoie soit le sous-arbre dont `x` est racine si l'élément figure dans l'arbre, soit, dans le cas contraire, l'arbre vide.

`adresse(x,a)`

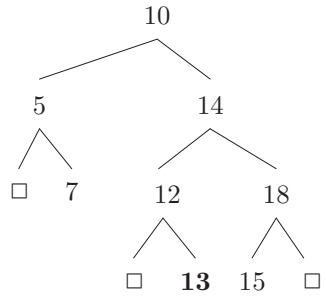
```
si (vide(a))
   renvoyer(arbreVide)
sinon
   si (x = racine(a))
      renvoyer(a)
   sinon
      si(x < racine(a))
         renvoyer(adresse(x,sag(a))
      sinon
         renvoyer(adresse(x,sad(a))
```

7.3.2 Adjonction aux feuilles

L'adjonction aux feuilles consiste à rajouter un nouvel élément comme feuille d'un ABR de façon à ce que celui-ci demeure un ABR. Ainsi le rajout de l'élément 13 comme feuille de l'arbre :



ne peut se faire que comme sous-arbre droit du sous-arbre de racine 12. L'analyse du processus qui permet d'aboutir à cette constatation est la suivante : puisque 13>10, il doit être rajouté dans le sous-arbre droit sur lequel on est descend ; de façon analogue, puisque 13<14 on descend à gauche et puisque 13>12, il doit être rajouté au sous-arbre droit. Ce dernier étant vide, il est remplacé par une feuille portant 13.



L'algorithme d'adjonction aux feuilles prend en paramètre l'élément x à rajouter et l'ABR a. Il renvoie l'ABR dûment modifié. Il est conçu récursivement : si a est vide, a est transformé en une feuille portant x. Sinon, si x est inférieur à la racine de a, le sous-arbre gauche de a est remplacé par l'arbre obtenu en y rajoutant x (rappel récursif). On procède de façon analogue dans le cas contraire.

```
ajoutAuxFeuilles(x,a)

si (vide(a))
   a ← ab(x, arbreVide, arbreVide)
sinon
   si (x < racine(a))
      sag(a) ← ajoutAuxFeuilles(x,sag(a))
   sinon
      sad(a) ← ajoutAuxFeuilles(x,sad(a))
renvoyer(a)
```

Si l'ABR est construit à partir de l'arbre vide par adjonctions aux feuilles successives, les éléments les plus récents sont alors les plus profonds dans l'arbre. Imaginons un cas de figure dans lesquels les éléments les plus récents sont également les plus instables : par exemple, dans une entreprise, les employés les plus récemment recrutés feront certainement l'objet de mises à jour plus fréquentes : titularisation, augmentation de salaire, changements d'affectation, changement du statut familial, etc. Dans ce cas, pour minimiser le coût des accès aux éléments les plus récents, il est préférable de les rajouter le plus haut possible dans l'arbre. C'est ce qui motive le paragraphe suivant.

7.3.3 Adjonction à la racine

Le but est donc de rajouter un nouvel élément x à un ABR a, de façon à obtenir un nouvel ABR dont il est la racine. Il faut donc *couper* l'ABR initial en deux ABR, l'un, G, comportant tous les éléments de a inférieurs à x, l'autre D portant tous ceux qui lui sont supérieurs. L'ABR cherché sera donc le triplet *(x, G, D)*. On se propose donc d'écrire un algorithme *coupure*, tel que si a est un ABR et x un élément :

coupure(x, a) = (G, D) avec : G, l'ABR des éléments e de a tels que $e < x$
D, l'ABR des éléments e de a tels que $e \geq x$

Soit par exemple l'élément 11 à ajouter à la racine dans l'arbre a de la figure 7.6.

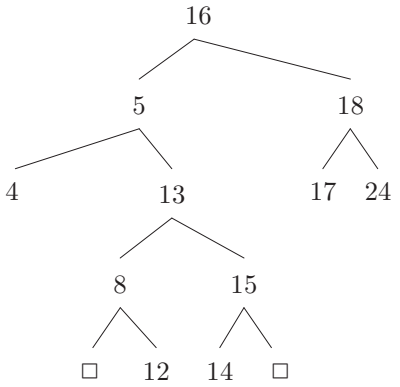
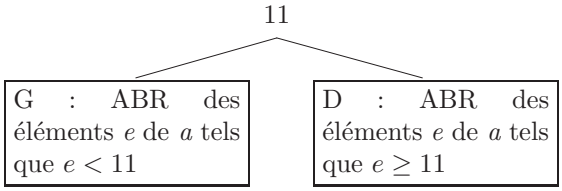


FIGURE 7.6 – L'ABR a objet d'une adjonction à la racine

Il faut donc construire un arbre de la forme :

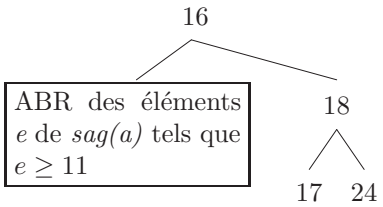


Puisque 11 est strictement inférieur à la racine 16 de a, le sous-arbre D comportera donc exactement :

1. la racine et tous les éléments du sous-arbre droit de a
2. les éléments du sous-arbre gauche de a qui sont supérieurs ou égaux à 11.

Clairement, tous les éléments de l'alinéa 2 sont inférieurs à tous ceux de l'alinéa 1, du fait qu'ils proviennent respectivement du sous-arbre gauche de a et de son sous-arbre droit (ou de sa racine).

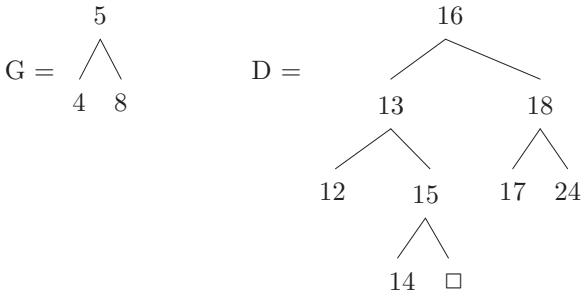
On en déduit que D est de la forme :



Par suite, le sous-arbre gauche D' de D est l'arbre tel que :

$$coupure(11,\ sag(a)) = (G',\ D')$$

Quant à l'arbre G, il est égal à G'. Le cas où l'élément à rajouter est supérieur à la racine se traite de façon symétrique. On obtient, en réitérant à la main comme ci-dessus les calculs de coupures sous les sous-arbres de a, les deux ABR : :



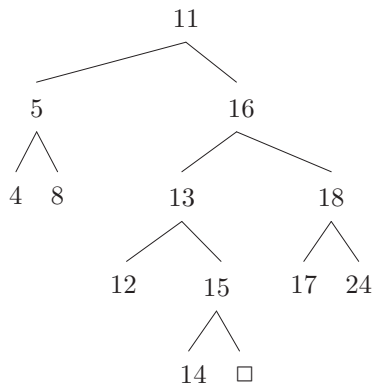
On peut maintenant mettre en forme l'algorithme

```
coupure(x, a)

G, D : arbres

si (vide(a))
   renvoyer(arbreVide, arbreVide)
sinon
   si (x< racine(a))
      D ← a
      (G, sag(D)) ← coupure(x, sag(a))
   sinon
      G ← a
      (sad(G), D) ← coupure(x, sad(a))
renvoyer(G,D)
```

L'algorithme d'adjonction à la racine s'en déduit immédiatement :

```
ajoutALaRacine(x, a)

G, D : arbres

(G,D) ← coupure(x, a)
renvoyer(ab(x, G, D))
```

Le résultat de l'ajout de 11 à la racine dans l'arbre a de la figure 7.6 est ainsi :



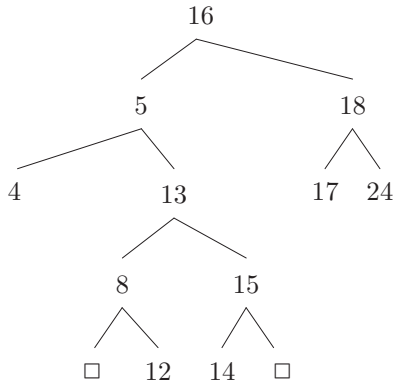
7.3.4 Suppression

Il s'agit ici de supprimer un élément x d'un ABR a tout en préservant la structure d'ABR. Soit a' le sous-arbre de racine x. Soient g et d respectivement le sous-arbre gauche et le sous-arbre droit de a'. Si g (resp. d) est vide, il suffit de remplacer a' par d (resp. g). Sinon, l'idée consiste, à remplacer la racine de a' par le plus petit élément m de d. On est ainsi assuré que la structure d'ABR est préservée puisque la nouvelle racine m :

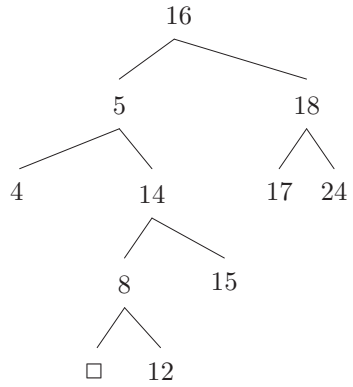
- étant issue de d, est supérieure à tous les éléments de g
- étant le minimum de d, est inférieure à tous les éléments du nouveau sous-arbre droit, à savoir d privé de m.

Ainsi, la suppression de 13 dans l'arbre de la figure 7.6, consiste à le remplacer par l'élément 14, préalablement ôté du sous-arbre droit. On obtient ainsi

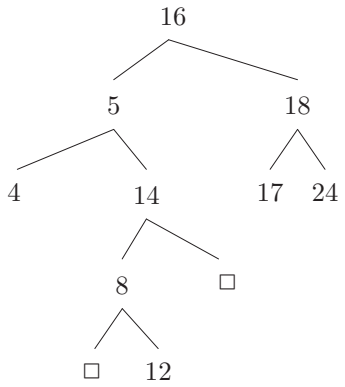
à partir de



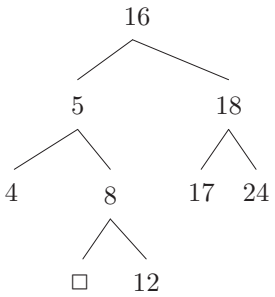
le nouvel arbre



Ceci n'est correct que si d n'est pas vide. Si ça n'est pas le cas, on a déjà remarqué qu'il suffit alors simplement de remplacer a' par g. Par exemple, en supprimant l'élément 14 de l'arbre :



on obtient



L'algorithme `supprimer` fait donc appel à un algorithme `supprimerMin` qui ôte le minimum d'un ABR et renvoie sa valeur.

Le minimum d'un ABR non vide est soit le minimum de son sous-arbre gauche si celui-ci n'est pas vide, soit, dans le cas contraire, sa racine. L'algorithme supprimant le minimum m d'un ABR non vide a renvoie un couple constitué de m et de l'arbre obtenu en ôtant m à l'arbre a.

`supprimerMin(a)`

```
si (vide(sag(a)))
   renvoyer(racine(a), sad(a))
sinon
   (x,sag(a)) ← supprimerMin(sag(a))
   renvoyer(x, a)
```

L'algorithme suivant supprime l'élément x de l'arbre a et renvoie l'arbre modifié :

`supprimer(x, a)`

```
si (non vide(a))
   si (x<racine(a))
      sag(a) ← supprimer(x, sag(a))
   sinon
      si (x>racine(a))
         sad(a) ← supprimer(x, sad(a))
      sinon          /* x = racine(a) */
```

```
            si (vide(sag(a)))
               a ← sad(a)
            sinon
               si (vide(sad(a)))
                  a ← sag(a)
               sinon
                  (racine(a), sad(a)) ← supprimerMin(sad(a))
   renvoyer(a)
```

On remarque que si l'élément x n'est pas dans l'arbre, l'algorithme de suppression ne fait que parcourir une branche jusqu'à arriver à l'arbre vide et s'arrête. Ainsi, l'arbre a n'est pas modifié, ce qui est conforme à la sémantique attendue.

7.4 Complexité des opérations sur les ABR

Il s'agit d'évaluer la comportement asymptotique des coûts des recherches, des insertions et des suppressions dans un ABR en fonction de la taille de l'arbre. Ce développement théorique nécessite l'introduction de quelques notions sur les arbres binaires.

Notation *Dans toute la suite, si* a *est un arbre binaire, on notera respectivement* g_a *et* d_a *les sous-arbres gauche et droit de* a.

7.4.1 Instruments de mesure sur les arbres

Définition 7.2 (Profondeur) *On appelle profondeur d'un nœud x d'un arbre le nombre $p(x)$ de ses ancêtres. Autrement dit, il s'agit du nombre d'arêtes du chemin de la racine à x.*

Ainsi, la profondeur de la racine d'un arbre est 0, celle de ses fils est 1.

Définition 7.3 (Hauteur) *On appelle hauteur d'un arbre non vide, la profondeur maximale de ses nœuds.*

La hauteur de l'arbre représenté sur la figure 7.6 est 4.

Proposition 7.4 *La taille* n *et la hauteur* h *d'un arbre binaire non vide satisfont la relation*

$$log_2(n) < h + 1 \leq n$$

Preuve Soit a un arbre binaire de taille maximale parmi tous ceux de hauteur h, donc ayant $h+1$ niveaux. Le premier niveau porte la racine et n'a donc qu'un seul élément. Tout autre niveau possède deux fois plus d'éléments que le niveau immédiatement au dessus. On en déduit la taille de l'arbre a :

$$taille(a) \ = \ \sum_{k=0}^{h} 2^k \ = \ 2^{h+1} - 1$$

De plus, un arbre de taille minimale parmi tous ceux de profondeur h a pour taille $h+1$. Il s'agit des arbres binaires dont tout sous-arbre qui n'est pas une feuille a un unique fils. De tels arbres binaires, isomorphes à une liste, sont dits dégénérés. Par suite, tout arbre binaire de hauteur h a pour taille un entier n tel que $h + 1 \leq n < 2^{h+1}$. De la deuxième inégalité on déduit que $log_2(n) < h + 1$ ce qui achève la démonstration. □

Définition 7.5 (Longueur de cheminement, profondeur moyenne) *On appelle longueur de cheminement d'un arbre non vide* a*, et on note* lc(a)*, la somme des profondeurs de ses nœuds. La* profondeur moyenne des nœuds *de l'arbre est donc :*

$$P(a) \ = \ \frac{lc(a)}{taille(a)}$$

Par convention, si a *est vide, on pose* $lc(a) = 0$ *et* $P(a) = -1$.

Sur l'exemple de l'arbre a de la figure 7.6, on obtient :

$$lc(a) \;=\; 2 \times 1 \,+\, 4 \times 2 \,+\, 2 \times 3 \,+\, 2 \times 4 \;=\; 24$$

$$P(a) \;=\; \frac{24}{11} \;=\; 2.18$$

Proposition 7.6 *Soit un arbre binaire de taille* n>0 *et* i *la taille de son sous-arbre gauche. La fonction* P *de profondeur moyenne des nœuds satisfait la relation :*

$$P(a) \;=\; \frac{1}{n} \times (iP(g_a) \,+\, (n-i-1)P(d_a) \,+\, (n-1))$$

Preuve Si $i \in \{0, \ldots, (n-1)\}$ est la taille de g_a, alors la taille de d_a est nécessairement $(n-1-i)$. Par définition de la longueur de cheminement :

$$lc(a) \;=\; \sum_{x \in g_a} p(x) \,+\, \sum_{x \in d_a} p(x)$$

puisque la profondeur de la racine est nulle. Notons p_g et p_d respectivement les profondeurs des nœuds dans les arbres g_a et d_a. Pour tout nœud x de g_a, sa profondeur dans a est supérieure de 1 à sa profondeur dans g_a. En procédant de façon analogue pour le sous-arbre droit, on obtient : $\forall x \in g_a,\ p(x) \;=\; p_g(x) + 1$ et $\forall x \in d_a,\ p(x) \;=\; p_d(x) + 1$. Par suite :

$$lc(a) \;=\; \sum_{x \in g_a} (p_g(x) \,+ 1\,) \,+\, \sum_{x \in d_a} (p_d(x) \,+\, 1) \;=\; \sum_{x \in g_a} p_g(x) \,+\, i \,+\, \sum_{x \in d_a} p_d(x) \,+\, (n-1-i) \;=$$

$lc(g_a) \,+\, lc(d_a) \,+\, (n-1)$.

Par définition de la fonction P, on en déduit que :

$$P(a) \;=\; \frac{1}{n} lc(a) \;=\; \frac{1}{n}(i \times P(g_a) \,+\, (n-i-1) \times P(d_a) \,+\, (n-1))$$

□

Définition 7.7 (Longueur de cheminement externe) *On appelle longueur de cheminement externe d'un arbre non vide* a*, et on note* lce(a)*, la somme des profondeurs de ses feuilles. Par convention, la longueur de cheminement externe de l'arbre vide est nulle.*

Sur l'exemple de l'arbre a de la figure 7.6, on obtient :

$$lce(a) \;=\; 2 \times 3 \,+\, 4 \times 2 \;=\; 14$$

Proposition 7.8 *Soit* a *un arbre binaire de taille* n>0 *et* f(a) *le nombre de ses feuilles. La longueur de cheminement externe* lce(a) *de* a *satisfait la relation :*

$$lce(a) \;=\; lce(g_a) + lce(d_a) + f(a)$$

Preuve Le résultat provient du fait que les feuilles de a sont exactement celles de g_a et d_a et que la profondeur d'une feuille de a est supérieure de 1 à sa profondeur dans le sous-arbre gauche ou droit dans lequel elle apparaît. □

7.4.2 Arbres aléatoires de recherche ($\mathbf{ABR}_n$)

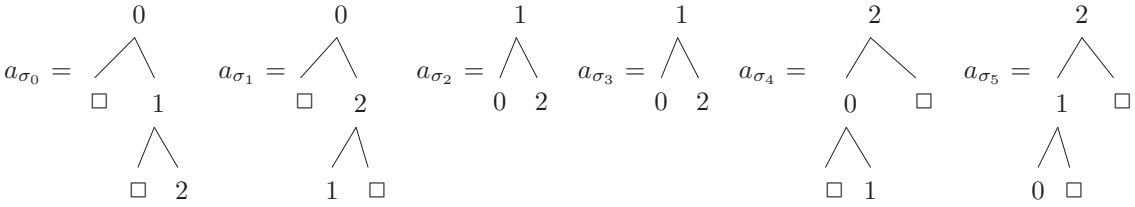
Motivations

La *recherche* d'un élément consiste à descendre le long d'une branche jusqu'à rencontrer l'élément ou atteindre une feuille. Le coût de l'algorithme est du même ordre de grandeur que la profondeur du nœud auquel on aboutit.
Pour l'*insertion aux feuilles*, il faut descendre jusqu'à une feuille. L'*insertion à la racine* requiert une coupure de l'arbre effectuée par un algorithme dont chaque rappel récursif descend d'un niveau jusqu'à arriver à une feuille. Une fois la coupure effectuée, l'adjonction à la racine ne requiert plus que des opérations qui se font en temps constant. Ces deux algorithmes ont donc un coût du même ordre de grandeur que la profondeur de la feuille à laquelle il aboutit.
Enfin, l'algorithme de *suppression* consiste à descendre sur une branche jusqu'à l'élément à supprimer puis à rechercher le minimum d'un sous-arbre. L'algorithme a un coût proportionnel à la profondeur du nœud auquel il aboutit.

Il en résulte que tous ces algorithmes ont un coût du même ordre de grandeur que la profondeur soit d'une feuille, soit d'un nœud quelconque. D'après la proposition 7.4, il peut donc varier dans le pire des cas et selon la hauteur de l'arbre, entre une complexité logarithmique et une complexité linéaire selon la configuration de l'ABR. On est ainsi conduit à faire une étude en moyenne, sur l'ensemble de toutes les configurations possibles.

Définition des ABR_n

Considérons un ensemble à n éléments que l'on souhaite organiser en ABR. Les $n!$ permutations de ces éléments donnent autant de configurations des données, toutes considérées comme équiprobables. Chacune de ces configurations permet de construire un ABR par adjonctions successives aux feuilles. Sur l'exemple de l'ensemble $\{0, 1, 2\}$, chacune des 3! permutations :
$\sigma_0 = (0,1,2)$, $\sigma_1 = (0,2,1)$, $\sigma_2 = (1,0,2)$, $\sigma_3 = (1,2,0)$, $\sigma_4 = (2,0,1)$, $\sigma_5 = (2,1,0)$
produit les ABR suivants :



On constate ici que deux permutations [1] distinctes donnent le même arbre ($a_{\sigma_2} = a_{\sigma_3}$). Par suite si les permutations sont équiprobables, les ABR en résultant ne le sont plus. Pour tourner la difficulté, on est naturellement conduit à travailler avec des multi-ensembles, c'est-à-dire informellement des ensembles *avec répétitions*. En d'autres termes, deux éléments identiques, comme a_{σ_2} et a_{σ_3}, sont considérés comme distincts s'ils proviennent de deux configurations (ou permutations) distinctes. Chaque éléments d'un multi-ensemble se voit ainsi affecté d'un *mulyipkicité*, qui est le nombre de fois qu'il y figure. Le cardinal du multi-ensemble est alors le nombre de ses éléments ainsi distingués par les configurations dont ils sont issus. Les multi-ensembles se notant entre doubles accolades.

Exemple On écrit :

$$ABR_3 = \{\{a_{\sigma_0}, a_{\sigma_1},\ a_{\sigma_2},\ a_{\sigma_3},\ a_{\sigma_4},\ a_{\sigma_5}\}\}$$

Son cardinal est noté $\#ABR_3$ et est égal à 6. La multiplicité de l'élément $a = a_{\sigma_2} = a_{\sigma_3}$ vaut deux, celle des autres éléments est égale à 1.

1. On emploie ici indifféremment les termes *configuration* et *permutation*.

Enfin, on notera que seul l'ordre sur des nœuds de l'arbre (et non pas la valeur des nœuds elles-mêmes) est significatif dans cette étude. A partir d'un ensemble ordonné $\{x_0,\ x_1,\ x_2\}$ avec $x_0 < x_1 < x_2$ on aurait obtenu un multi-ensemble isomorphe au précédent, en confondant tout nœud x_i avec son indice i. Ainsi, et sans perdre en généralité, on sera amené à considérer des ABR_n sur l'ensemble

$$I_n = \{0,\ \ldots,\ (n-1)\}$$

où chaque entier i doit être compris comme le $i$ème élément n_i d'un ensemble ordonné à n éléments [2].

Notation *On notera $\mathcal{S}_n$ l'ensemble des permutations de I_n.*

Définition 7.9 (ABR aléatoires) *Soit n un entier naturel et σ un élément de $\mathcal{S}_n$. On note a_σ l'ABR produit à partir de l'arbre vide par adjonctions successives aux feuilles de chacun des éléments, pris dans l'ordre : $\sigma(0), \ldots, \sigma(n-1)$.*
On appelle arbre binaire de recherche aléatoire de taille n *chacun des $n!$ éléments a_σ du multi-ensemble ABR_n obtenu à partir de chacune des $n!$ permutations σ de $\mathcal{S}_n$. Chacun de ces arbres est défini à un isomorphisme près sur l'ensemble des nœuds et affecté de la probabilité $\dfrac{1}{n!}$.*

On dira, par abus de langage, que a est un ABR_n au lieu de a est un *arbre binaire de recherche aléatoire de taille n*, confondant ainsi le nom du multi-ensemble et le terme générique désignant ses éléments.

Propriétés des ABR_n

Dans le but de décrire la conformation des ABR_n, on introduit la définition suivante permettant de prendre en compte la taille des sous-arbres gauche et droit.

Proposition et définition 7.10 *Soit* i$\in I_n$. *On note A_i^n le multi-ensemble des* ABR$_n$ *de racine* i. *Les propriétés suivantes sont équivalentes*

1. *A_i^n est le multi-ensemble des* ABR$_n$ *dont la racine est* i.
2. *A_i^n est le multi-ensemble des* ABR$_n$ *dont le sous-arbre gauche est de taille* i.
3. *A_i^n est le multi-ensemble des* ABR$_n$ *dont le sous-arbre droit est de taille* n-i-1.
4. *A_i^n est le multi-ensemble des éléments a_σ de* ABR$_n$ *tels que $\sigma(0) =$* i.

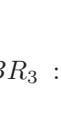
De plus, A_i^n a pour cardinal $(n-1)!$.
On notera simplement A$_i$ *au lieu de A_i^n quand il n'y a pas d'ambiguïté sur* n.

La démonstration de l'équivalence des quatre propriétés est évidente. Le cardinal de A_i^n est celui de l'ensemble des permutations σ de $\mathcal{S}_n$ telle que $\sigma(0) = i$, c'est donc le cardinal de l'ensemble des permutations des $(n-1)$ éléments restants, soit $(n-1)!$.

Les sous-arbres gauches des éléments de A_i sont des arbres binaires de recherche aléatoires de taille i. Toutefois, des éléments distincts de A_i peuvent avoir des sous-arbres gauches correspondant à la même permutation de $\mathcal{S}_i$. Par exemple, si $n = 5$ et $i = 3$, les permutations :

$$\sigma_1 = (3,\ 1,\ 0,\ 2,\ 4), \quad \sigma_2 = (3,\ 1,\ 0,\ 4,\ 2), \quad \sigma_3 = (3,\ 1,\ 4,\ 0,\ 2), \quad \sigma_4 = (3,\ 4,\ 1,\ 0,\ 2)$$

2. Plus formellement, si $X = \{x_0,\ \ldots,\ x_{n-1}\}$ et $Y = \{y_0,\ \ldots,\ y_{n-1}\}$ sont deux ensembles ordonnés à n éléments tels que $\forall i, j \in \{0,\ \ldots,\ n-1\}\ i < j \Rightarrow (x_i < x_j$ *et* $y_i < y_j)$, il existe un unique isomorphisme ϕ de X vers Y, celui défini par $\forall i,\ \phi(x_i) = y_i$. I_n n'est autre qu'un représentant canonique de tous ces ensembles isomorphes.

produiront toutes, comme sous-arbre gauche de la racine, le même ABR_3 : a_ρ =  avec $\rho = (1, 0, 2)$ puisque 1, 0 et 2, apparaissent dans cet ordre dans les quatre configurations ci-dessus. La proposition suivante permet de dénombrer les permutations σ de $\mathcal{S}_n$ qui produisent comme sous-arbre gauche de la racine le même ABR_i, c'est-à-dire les sous-arbres gauches issus de la même permutation ρ de $\mathcal{S}_i$.

Proposition 7.11 *Soient* i$\in I_n$ *fixé,* ρ *une permutation de* $\mathcal{S}_i$*,* a_ρ *l'*ABR$_i$ *correspondant,* ρ' *une permutation de* $\mathcal{S}_{n-i-1}$ *et* $a_{\rho'}$ *l'*ABR$_{n-i-1}$ *correspondant.*
Pour toute permutation σ *de* $\mathcal{S}_n$ *telle que* $\sigma(0) = i$*, on désigne respectivement par* g_σ *et* d_σ *les sous-arbres gauche et droit de l'*ABR$_n$ a_σ *(dont la racine est i puisque* $\sigma(0) = i$*). On démontre alors que :*

$$\#\{\sigma \in \mathcal{S}_n \ / \ g_\sigma = a_\rho\} = \frac{(n-1)!}{i!}$$

$$\#\{\sigma \in \mathcal{S}_n \ / \ d_\sigma = a_{\rho'}\} = \frac{(n-1)!}{(n-i-1)!}$$

Preuve Il s'agit dans un premier temps dénombrer toutes les permutations

$$\sigma = (i, \sigma(1), \ \ldots, \ \sigma(n-1))$$

de $\mathcal{S}_n$ telles que $g_\sigma = a_\rho$. Il faut et il suffit pour cela que les entiers $\rho(0), \ldots, \rho(i-1)$ apparaissent dans cet ordre dans le $(n-1)$-uple $(\sigma(1), \ \ldots, \ \sigma(n-1))$.
Il y a C_{n-1}^i façons de choisir les i rangs de $\rho(0), \ldots, \rho(i-1)$ dans le $(n-1)$-uplet. Ces rangs étant déterminés, il y $(n-1-i)!$ façons de configurer les éléments restants dans les emplacements restés vacants. Soit au total $(n-1-i)! \times C_{n-1}^i = \dfrac{(n-1)!}{i!}$ permutations.
La démonstration pour les sous-arbres droits est analogue, à ceci près que l'on considère des permutations de l'ensemble $\{(i+1), \ldots, (n-1)\}$ et non de l'ensemble $\{0, \ldots, (n-i-2)\}$, mais on a vu tous les ensembles totalement ordonnés de même cardinal fini se traitent de façon équivalente. □

Corollaire 7.12 *Soient* n *un entier naturel et* α *une fonction numérique sur les ABR. On démontre l'égalité suivante :*

$$\sum_{a \in ABR_n} \alpha(g_a) = \sum_{a \in ABR_n} \alpha(d_a)$$

Preuve Il est clair que $\displaystyle\sum_{a \in ABR_n} \alpha(g_a) = \sum_{i=0}^{n-1} \sum_{a \in A_i} \alpha(g_a)$. Par suite, d'après la proposition 7.11, on en déduit que :

$$\sum_{a \in ABR_n} \alpha(g_a) = \sum_{i=0}^{n-1} \frac{(n-1)!}{i!} \sum_{a \in ABR_i} \alpha(a)$$

Or,

$$\sum_{i=0}^{n-1} \frac{(n-1)!}{i!} \sum_{a \in ABR_i} \alpha(a) = \sum_{i=0}^{n-1} \frac{(n-1)!}{(n-1-i)!} \sum_{a \in ABR_{n-1-i}} \alpha(a)$$

Ces deux sommes ont en effet exactement les mêmes termes, écrits toutefois en ordre inverse. Enfin, en appliquant à nouveau la proposition 7.11, on obtient :

$$\sum_{i=0}^{n-1} \frac{(n-1)!}{(n-1-i)!} \sum_{a \in ABR_{n-1-i}} \alpha(a) = \sum_{i=0}^{n-1} \sum_{a \in A_i} \alpha(d_a) = \sum_{a \in ABR_n} \alpha(d_a)$$ □

7.4.3 Profondeur moyenne des nœuds dans ABR_n

Théorème 7.13 *La profondeur moyenne P_n des nœuds des ABR_n satisfait la double inégalité suivante :*

$$\forall n > 0 \quad ln(n) - \frac{5}{2} \leq P_n \leq 4ln(n)$$

La démonstration utilise le lemme suivant :

Lemme 7.14 *Pour tout entier strictement positif* n*, P_n satisfait la relation de récurrence suivante :*

$$P_n \ = \ \frac{(n-1)}{n} \ + \ \frac{2}{n^2} \sum_{i=1}^{n-1} iP_i$$

Preuve du théorème Supposons pour l'instant démontré le lemme.

1. La relation $\forall n \geq 1 \ P_n \leq 4ln(n)$ se démontre par récurrence sur n.
   Soit $n > 0$ fixé. Supposons que $\forall k, 0 < k < n, \ P_k \leq 4ln(k)$. On déduit du lemme précédent que :
   $$P_n \leq \frac{(n-1)}{n} \ + \ \frac{8}{n^2} \sum_{i=1}^{n-1} iln(i)$$
   Or
   $$\sum_{i=1}^{n-1} iln(i) \leq \int_1^n xln(x)dx \ = \ \frac{n^2}{2}ln(n) - \frac{n^2}{4} + \frac{1}{4}$$
   Donc
   $$P_n \leq \frac{(n-1)}{n} \ + 4ln(n) - 2 + \frac{2}{n^2} \ = \ 4ln(n) + \frac{2 - n(n+1)}{n^2} \leq 4ln(n)$$
2. Soit $n \geq 2$ fixé. Les ABR de taille n pour lesquels la profondeur moyenne des nœuds est minimale sont ceux dont la hauteur h est minimale (noter que puisque $n \geq 2$ nécessairement $h \geq 1$). Il s'agit des arbres dont chaque niveau i, sauf peut-être le dernier, comporte le maximum de nœuds, soit 2^i. Pour un tel arbre a, la profondeur moyenne P_a des nœuds est supérieure ou égale à la profondeur moyenne des nœuds qui ne sont pas situés sur le dernier niveau h. On en déduit que :

   $$P_n \geq P_a \geq \frac{1}{\sum_{i=0}^{h-1} 2^i} \sum_{i=0}^{h-1} i.2^i \geq \frac{1}{2^h} \sum_{i=1}^{h-1} i.2^i \geq \frac{1}{2^h} \int_0^{h-1} x.2^x dx.$$

   Sachant que $2^x = e^{x.ln(2)}$, on en déduit que la dérivée $(2^x)' = ln(2).2^x$ et par suite que $\frac{1}{ln(2)}2^x$ est une primitive de 2^x.

   Une intégration par parties montre alors que $\frac{1}{ln(2)^2}\left(ln(2)x.2^x - 2^x\right)$ est une primitive de $x.2^x$. Par suite :

   $$P_n \geq \frac{1}{2^h} \int_1^{h-1} x.2^x dx \ = \ \frac{1}{2^h ln(2)^2} \left(ln(2)(h-1).2^{h-1} - 2^{h-1} + 1\right) \ =$$

   $$\frac{1}{ln(2)^2} \left(\frac{ln(2)(h-1)}{2} - \frac{1}{2} + \frac{1}{2^h}\right) \geq \frac{1}{ln(2)^2} \left(\frac{ln(2)(log_2(n)-2)}{2} - \frac{1}{2} + \frac{1}{2^h}\right) \geq$$

   (d'après la proposition 7.4)

   $$\frac{1}{2ln(2)}.log_2(n) - \frac{1}{ln(2)^2}.\left(ln(2) + \frac{1}{2}\right) \ = \ \frac{1}{2ln(2)^2}.ln(n) - \frac{1}{ln(2)^2}.\left(ln(2) + \frac{1}{2}\right) \geq$$

$ln(n) - \frac{5}{2}$.

En effet : $\frac{1}{2ln(2)^2} \approx 1,041$ et $\frac{1}{ln(2)^2}.\left(ln(2) + \frac{1}{2}\right) \approx 2,49$.

Cette inégalité a été prouvée pour $n \geq 2$. Elle est évidemment vérifiée pour $n = 1$.

□

Preuve du lemme Soit $n > 0$. Sachant que P_n est par définition la somme des profondeurs de tous les nœuds des arbres a de ABR_n, qu'il y a $(n-1)!$ éléments dans ABR_n, chacun comportant n nœuds, on obtient :

$$P_n = \frac{1}{n! \times n} \sum_{a \in ABR_n} \sum_{x \in a} p(x) = \frac{1}{n!} \sum_{a \in ABR_n} (\frac{1}{n} \sum_{x \in a} p(x)) = \frac{1}{n!} \sum_{a \in ABR_n} P(a) = \frac{1}{n!} \sum_{i=0}^{n-1} \sum_{a \in A_i} P(a) =$$

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{(n-1)!} \sum_{a \in A_i} P(a).$$

Or, d'après la proposition 7.6 :

$$\sum_{a \in A_i} P(a) \ = \ \frac{1}{n} \sum_{a \in A_i} (iP(g_a) \ + (n-1-i)P(d_a) \ + \ (n-1))$$

Enfin, on a vu que la cardinal de A_i est $(n-1)!$, il y a donc $(n-1)!$ termes dans la somme ci-dessus. Par suite :

$$\frac{1}{(n-1)!} \sum_{a \in A_i} P(a) \ = \ \frac{1}{n} \left((n-1) \ + i \frac{1}{(n-1)!} \sum_{a \in A_i} P(g_a) \ + \ (n-1-i) \frac{1}{(n-1)!} \sum_{a \in A_i} P(d_a) \right)$$

De plus, en appliquant la proposition 7.11, on en déduit que :

$$\sum_{a \in A_i} P(g_a) \ = \ \frac{(n-1)!}{i!} \sum_{a \in ABR_i} P(a) \ = \ (n-1)! \, P_i$$

et que

$$\sum_{a \in A_i} P(d_a) \ = \ \frac{(n-1)!}{(n-1-i)!} \sum_{a \in ABR_{n-1-i}} P(a) \ = \ (n-1)! \, P_{n-1-i}$$

Par suite :

$$\frac{1}{(n-1)!} \sum_{a \in A_i} P(a) \ = \ \frac{1}{n} \left((n-1) \ + \ iP_i \ + \ (n-1-i)P_{n-1-i} \right)$$

et donc :

$$P_n \ = \ \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{(n-1)!} \sum_{a \in A_i} P(a) \ = \ \frac{1}{n^2} \sum_{i=0}^{n-1} ((n-1) \ + \ iP_i \ + \ (n-1-i)P_{n-1-i})$$

Or, il se trouve que

$$\sum_{i=0}^{n-1} iP_i \ = \ \sum_{i=0}^{n-1} (n-1-i)P_{n-1-i}$$

puisque les deux sommes ont exactement les mêmes termes, énumérés en ordre inverse. On obtient donc :

$$P_n \ = \ \frac{(n-1)}{n} \ + \ \frac{2}{n^2} \sum_{i=0}^{n-1} iP_i \ = \ \frac{(n-1)}{n} \ + \ \frac{2}{n^2} \sum_{i=1}^{n-1} iP_i$$

□

7.4.4 Profondeur moyenne des feuilles des ABR_n

Théorème 7.15 *Pour tout entier strictement positif* n*, la profondeur moyenne F_n des feuilles des ABR_n satisfait la double inégalité :*

$$ln(n) \leq F_n \leq 10.ln(n)$$

La démonstration du théorème requiert le calcul préalable du nombre moyen de feuilles dans les ABR_n.

Notations On désigne respectivement par :

$f(a)$	le nombre de feuilles d'un arbre a.
$lce(a)$	la longueur de cheminement externe d'un arbre a.
$\forall n \in IN,\ f_n = \sum_{a \in ABR_n} f(a),$	le nombre des feuilles de tous les ABR_n.
$\overline{f_n} = \frac{1}{n!} f_n,$	le nombre moyen de feuilles dans ABR_n.
$F_n = \frac{1}{f_n} \sum_{a \in ABR_n} lce(a), \quad \forall n > 0$ $F_0 = 0$	la profondeur moyenne des feuilles de tous les ABR_n.

Théorème 7.16 (Nombre moyen de feuilles dans les ABR_n) *Pour tout entier naturel n, le nombre moyen $\overline{f_n}$ de feuilles des ABR_n satisfait les relations suivantes :*

$$\overline{f_0} = 0, \quad \overline{f_1} = 1 \quad et \quad \forall n \geq 2,\ \overline{f_n} = \frac{n+1}{3}$$

Preuve Si n vaut 0 ou 1, le résultat est immédiat.
Soit donc $n \geq 2$. La racine des éléments de ABR_n n'est pas une feuille. Par suite, les feuilles d'un tel ABR_n sont exactement celles de son sous-arbre gauche et celles de son sous-arbre droit. On obtient donc le calcul suivant :

$$f_n = \sum_{a \in ABR_n} f(a) = \sum_{a \in ABR_n} f(g_a) + \sum_{a \in ABR_n} f(d_a) =$$

$$2 \sum_{a \in ABR_n} f(g_a) = \qquad \textit{(corollaire 7.12)}$$

$$2 \sum_{i=0}^{n-1} \sum_{a \in A_i} f(g_a) = 2 \sum_{i=0}^{n-1} \frac{(n-1)!}{i!} \sum_{a \in ABR_i} f(a) = \qquad \textit{(proposition 7.11)}$$

$$2(n-1)! \sum_{i=0}^{n-1} \frac{1}{i!} \sum_{a \in ABR_i} f(a) = 2(n-1)! \sum_{i=0}^{n-1} \overline{f_i} \qquad \textit{(définition des } \overline{f_i}\textit{)}.$$

Par suite : $\forall n \geq 2,\ \overline{f_n} = \frac{1}{n!} f_n = \frac{2}{n} \sum_{i=0}^{n-1} \overline{f_i},$

et donc : $\overline{f_{n+1}} = \frac{2}{n+1} \left(\sum_{i=0}^{n-1} \overline{f_i} + \overline{f_n} \right) = \frac{2}{n+1} \left(\frac{n}{2} \overline{f_n} + \overline{f_n} \right) = \frac{n+2}{n+1} \overline{f_n}.$

Une récurrence simple, s'appuyant sur ce résultat et sur le fait que $\overline{f_2}$ vaut 1, permet de conclure.
□

La démonstration du théorème 7.15 s'appuie sur le lemme suivant.

Lemme 7.17 *La profondeur moyenne des feuilles des ABR_n satisfait les relations de récurrence suivantes :*

$$F_1 = 0$$
$$F_n = 1 + \frac{2}{n(n+1)} \sum_{i=1}^{n-1} (i+1) F_i \quad \forall n \geq 2$$

Preuve Le résultat est immédiat pour $n = 1$. Soit n un entier quelconque, supérieur ou égal à 2. En appliquant la définition de F_n on obtient le calcul suivant :

$$F_n = \frac{1}{f_n} \sum_{a \in ABR_n} lce(a) = \frac{1}{f_n} \sum_{a \in ABR_n} (lce(g_a) + lce(d_a) + f(a)) = \quad \textit{(proposition 7.8)}$$
$$1 + \frac{1}{f_n} \left(\sum_{a \in ABR_n} lce(g_a) + \sum_{a \in ABR_n} lce(d_a) \right) = 1 + \frac{2}{f_n} \sum_{a \in ABR_n} lce(g_a) = \quad \textit{(corollaire 7.12)}$$
$$1 + \frac{2}{f_n} \sum_{i=1}^{n-1} \sum_{a \in A_i} lce(g_a) = 1 + \frac{2(n-1)!}{f_n} \sum_{i=1}^{n-1} \frac{1}{i!} \sum_{a \in ABR_i} lce(a) = \quad \textit{(proposition 7.11)}$$
$$1 + \frac{2(n-1)!}{n! \overline{f_n}} \sum_{i=1}^{n-1} \frac{f_i}{i!} F_i = \quad \textit{(déf. de } \overline{f_n} \textit{ et } F_i\textit{)}$$
$$1 + \frac{2}{n \overline{f_n}} \sum_{i=1}^{n-1} \overline{f_i} F_i = 1 + \frac{2 \times 3}{n(n+1)} \sum_{i=1}^{n-1} \frac{i+1}{3} F_i = \quad \textit{(théorème 7.16)}$$
$$1 + \frac{2}{n(n+1)} \sum_{i=1}^{n-1} (i+1) F_i$$

□

Preuve du théorème 7.15
1.**Il s'agit dans un premier temps, de trouver une constante mumérique $b > 0$ telle que :**

$$\forall n \geq 1 \quad F_n \leq b.ln(n)$$

On procède par récurrence sur n. Si $n = 1$, $F_1 = 0 \leq b.ln(1) \quad \forall b > 0$.
Soit $n \geq 2$ un entier quelconque fixé et supposons la propriété vraie pour tout entier naturel $1 \leq i < n$. On obtient alors :

$$F_n = 1 + \frac{2}{n(n+1)} \sum_{i=1}^{n-1} (i+1) F_i \leq 1 + \frac{2b}{n(n+1)} \sum_{i=1}^{n-1} (i+1) ln(i)$$

On est ainsi conduit à étudier l'expression :

$$\sum_{i=1}^{n-1} (i+1) ln(i) = \sum_{i=2}^{n-1} i \times ln(i) + \sum_{i=2}^{n-1} ln(i) \leq \int_2^n x ln(x) dx + \int_2^n ln(x) dx$$

Or :

$$\int_2^n x ln(x) dx = \left[\frac{x^2}{2} ln(x) - \frac{x^2}{4} \right]_2^n = \frac{n^2}{2} ln(n) - \frac{n^2}{4} - 2ln(2) + 1$$

et

$$\int_2^n ln(x) dx = [x ln(x) - x]_2^n = n \times ln(n) - n - 2ln(2) + 2$$

Par suite,

$$\sum_{i=1}^{n-1} (i+1) ln(i) \leq \frac{n(n+2)}{2} ln(n) - \frac{n(n+4)}{4} + 3 - 4ln(2) \leq \frac{n(n+2)}{2} ln(n) - \frac{n(n+4)}{4} + \frac{1}{4}$$

En effet, $3 - 4ln(2) \approx 0.227411278$

On en déduit que

$$F_n \leq 1 + \frac{2b}{n(n+1)}\left(\frac{n(n+2)}{2}ln(n) - \frac{n^2+4n-1}{4}\right) = 1 + b \times \frac{(n+2)}{(n+1)}ln(n) - b \times \frac{n^2+4n-1}{2n(n+1)} =$$
$$b \times ln(n) + b \times \frac{1}{(n+1)}ln(n) - b \times \frac{n^2+4n-1}{2n(n+1)} + 1$$

Pour montrer que $F_n \leq b.ln(n)$, il suffit donc de trouver b tel que :

$$\forall n \geq 2,\ b \times \frac{ln(n)}{(n+1)} - b \times \frac{n^2+4n-1}{2n(n+1)} + 1 < 0$$

c'est-à-dire tel que :

$$\forall n \geq 2,\ 0 < \frac{1}{b} \leq \frac{n^2+4n-1}{2n(n+1)} - \frac{ln(n)}{(n+1)}$$

Il suffit donc, puisque tel que $\frac{ln(n)}{n} \geq \frac{ln(n)}{(n+1)}$ de trouver b tel que que :

$$\forall n \geq 2, \quad 0 < \frac{1}{b} \leq \frac{n^2+4n-1}{2n(n+1)} - \frac{ln(n)}{n} \tag{7.1}$$

En posant $f(n) = \frac{n^2+4n-1}{2n(n+1)}$ et $g(n) = \frac{ln(n)}{n}$, on obtient les fonctions dérivées :

$f'(n) = -\frac{3n^2-2n-1}{2\left(n(n+1)\right)^2} = -\frac{(3n+1)(n-1)}{2\left(n(n+1)\right)^2}$ et $g'(n) = \frac{1-ln(n)}{n^2}$.

On en déduit leurs tableaux de variation sur $[1 \quad +\infty[$.

n	1		$+\infty$
f'(n)	0	−	
f(n)	1	↘	$\frac{1}{2}$

n	1		e		$+\infty$
g'(n)		+	0	−	
g(n)	0	↗	$\frac{1}{e}$	↘	0

Par suite, $\forall n \geq 1,\ f(n) - g(n) \geq \frac{1}{2} - \frac{1}{e} > \frac{1}{10}$.

En effet, $\frac{1}{2} - \frac{1}{e} \approx 0,132120559$.

Il suffit donc de choisir $b = 10$ pour satisfaire la condition 7.1, d'où le théorème.

2. **Il s'agit maintenant de trouver une constante $a > 0$ telle que** $\forall n \geq 1 \quad F_n \geq a.ln(n)$.
De façon analogue à celle employée pour la majoration, on procède par récurrence sur n.
Si $n = 1$, $F_1 = 0 \geq a.ln(1) \quad \forall a > 0$.
Soit $n \geq 2$ un entier quelconque fixé et supposons la propriété vraie pour tout entier naturel $1 \leq i < n$. On obtient alors :

$$F_n = 1 + \frac{2}{n(n+1)}\sum_{i=1}^{n-1}(i+1)F_i \geq 1 + \frac{2a}{n(n+1)}\sum_{i=1}^{n-1}(i+1)ln(i) \geq 1 + \frac{2a}{n(n+1)}\sum_{i=2}^{n-1} i \times ln(i)$$

Or l'expression :

$$\sum_{i=1}^{n-1} iln(i) \geq \int_1^{n-1} xln(x)dx \;=\; \left[\frac{x^2}{2}ln(x) - \frac{x^2}{4}\right]_1^{n-1} = \frac{(n-1)^2}{2}ln(n-1) - \frac{(n-1)^2}{4} + \frac{1}{4}$$

et

$$\sum_{i=1}^{n-1} ln(i) \geq \int_1^{n-1} ln(x)dx \;=\; [xln(x) - x]_1^{n-1} \;=\; (n-1)ln(n-1) - (n-1) + 1$$

Par suite,

$$F_n \geq 1 + \frac{2a}{n(n+1)}\left((n-1)ln(n-1)(\frac{n-1}{2}+1) - \frac{(n-1)^2 + 4(n-1) - 5}{4}\right) =$$
$$1 + \frac{2a}{n(n+1)}\left(\frac{(n+1)(n-1)}{2}l_n(n-1) - \frac{(n-2)(n+4)}{4}\right) =$$
$$1 + a\left(\frac{n-1}{n}ln(n-1) - \frac{(n-2)(n+4)}{2n(n+1)}\right).$$

On en déduit que $F_n - a.ln(n) \geq 1 - a.\alpha(n)$ avec :

$$\forall n > 1, \quad \alpha(n) = ln(n) - \frac{n-1}{n}ln(n-1) + \frac{(n-2)(n+4)}{2n(n+1)} \quad \geq \quad ln(n) - \frac{n-1}{n}ln(n-1)$$

La fonction $\beta(n) = ln(n) - \dfrac{n-1}{n}ln(n-1)$ a pour dérivée $\beta'(n) = -\dfrac{ln(n-1)}{n^2}$ qui est strictement négative sur $[2 \quad +\infty[$. Par suite, β est décroissante sur cet intervalle et donc :

$$\forall n > 1 \quad \alpha(n) \leq \beta(n) \leq \beta(2) = ln(2)$$

On en déduit que

$$0 < a < \frac{1}{ln(2)} \quad \Rightarrow \quad F_n - a.ln(n) \;\geq\; 1 - a.\alpha(n) \;\geq\; 1 - a.ln(2) > 0$$

Il suffit de choisir $a = 1$ puisque $\dfrac{1}{ln(2)} \approx 1,45$
□

7.4.5 Conclusion

On a vu dans la section 7.4.2 que le coût moyen de la recherche d'un élément dans un ABR_n est du même ordre de grandeur asymptotique que la profondeur moyenne des nœuds des ABR_n. On a également vu que le coût moyen d'une suppression, d'une adjonction aux feuilles et d'une adjonction à la racine dans un ABR_n est du même ordre de grandeur asymptotique que la profondeur moyenne des feuilles des ABR_n.
On peut donc énoncer le théorème suivant.

Théorème 7.18 (Coût moyen des opérations élémentaires sur les ABR) *On considère un ensemble $E = \{a_0, \ \ldots \ a_{n-1}\}$ totalement ordonné à n éléments. Soit σ une permutation des éléments de E. On suppose que E est configuré en l'arbre binaire de recherche obtenu, à partir de l'arbre vide, par adjonctions successives des éléments $\sigma(a_0) \ \ldots \ , \ \sigma(a_{n-1})$. On suppose également que les n! permutations de E sont équiprobables.*
Les complexités en moyenne des recherches, des adjonctions aux feuilles ou à la racine, et des suppressions dans ces ABR sont logarithmiques en la taille n *de l'arbre.*

7.5 Arbres binaires parfaits partiellement ordonnés (Tas)

7.5.1 Codage d'un arbre binaire parfait

Définition Un arbre binaire est dit *parfait* lorsque tous ses niveaux sont complètement remplis, sauf éventuellement le dernier niveau et dans ce cas les nœuds (feuilles) du dernier niveau sont groupés le plus à gauche possible.

Ceci est illustré par la figure ci-dessous.

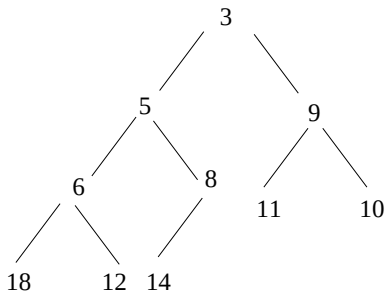


FIGURE 7.7 – Un arbre binaire parfait

Codage Un arbre binaire parfait de taille n peut se représenter de façon très compacte dans un tableau T de taille au moins égale à n. La représentation est la suivante :

La racine est en T[1] et si un nœud est en T[i], son fils gauche est en T[2i] et son fils droit en T[2i+1].

0	1	2	3	4	5	6	7	8	9	10	
?	3	5	9	6	8	11	10	18	12	14	...

FIGURE 7.8 – Codage de l'arbre de la figure 7.7

L'arbre et chacun de ses sous-arbres sont repérés par leur adresse i, $1 \leq i \leq n$, dans le tableau. Il est donc parfaitement déterminé par la donnée du tableau T et de sa taille n qui est aussi l'indice de la dernière case du tableau occupée par la représentation.
Il résulte de cette représentation qu'étant donné un entier $1 \leq i \leq n$,
- si $2i + 1 \leq n$, l'arbre d'adresse i a deux fils d'adresses respectives $2i$ et $2i + 1$,
- si $2i = n$ l'arbre d'adresse i a un fils gauche et pas de fils droit,
- si $2i > n$ n'a pas de fils : c'est une feuille.

Les primitives relatives à l'ensemble des sous-arbres $\{1, \ldots, n\}$ de l'arbre parfait de taille n peuvent ainsi être définies de la façon suivante :

racine(i) : T[i] si $i > 0$
sag(i) : si $2i \leq n$ alors $2i$
sad(i) : si $2i + 1 \leq n$ alors $2i + 1$.
pere(i) : $i/2$.

On remarque que le père de l'arbre global (d'adresse 1) a pour adresse 0, qui correspond ici au pointeur *null* quand les arbres sont représentés par des objets.

7.5.2 Tas

Un arbre binaire parfait (étiqueté sur un ensemble ordonné) est dit partiellement ordonné si l'étiquette de tout nœud est inférieure à celle de ses fils. Un arbre parfait partiellement ordonné est aussi appelé *tas*. Un arbre parfait non vide est donc un tas si et seulement si chacun de ses sous-arbres gauche et droit est un tas et si sa racine est inférieure à celles de ses fils. C'est le cas de l'exemple de la figure 7.7.
L'intérêt de cette structure de donnée est l'accès facile au minimum du tas, qui se trouve à la racine. Par exemple, l'algorithme de Dijkstra calculant le plus court chemin entre deux sommets d'un graphe valué (cf. section 8.10) procède par suppressions successives du minimum d'un ensemble jusqu'à épuisement de ce dernier. L'utilisation d'une structure de tas est donc essentielle à une mise en œuvre efficace de ce célèbre algorithme de la théorie des graphes.

7.5.3 Suppression du minimum d'un tas

L'idée est de supprimer la dernière feuille du dernier niveau après avoir recopié son étiquette à la racine, puis, pour respecter la structure de tas, de faire redescendre cette valeur en la comparant au contenu de ses fils. Si l'un au moins d'entre eux est supérieur à la racine, celle-ci doit être échangée avec leur minimum. Une fois déplacée, on réitère le processus en examinant à nouveau ses fils, et ceci jusqu'à ce qu'elle soit inférieure à ses fils, s'ils existent. Le fait de faire redescendre un nœud dans l'arbre pour respecter la structure de tas est appelé *tamisage* (ou *percolation*) vers le bas. La figure ci-après illustre ce processus dans le cas de l'exemple.

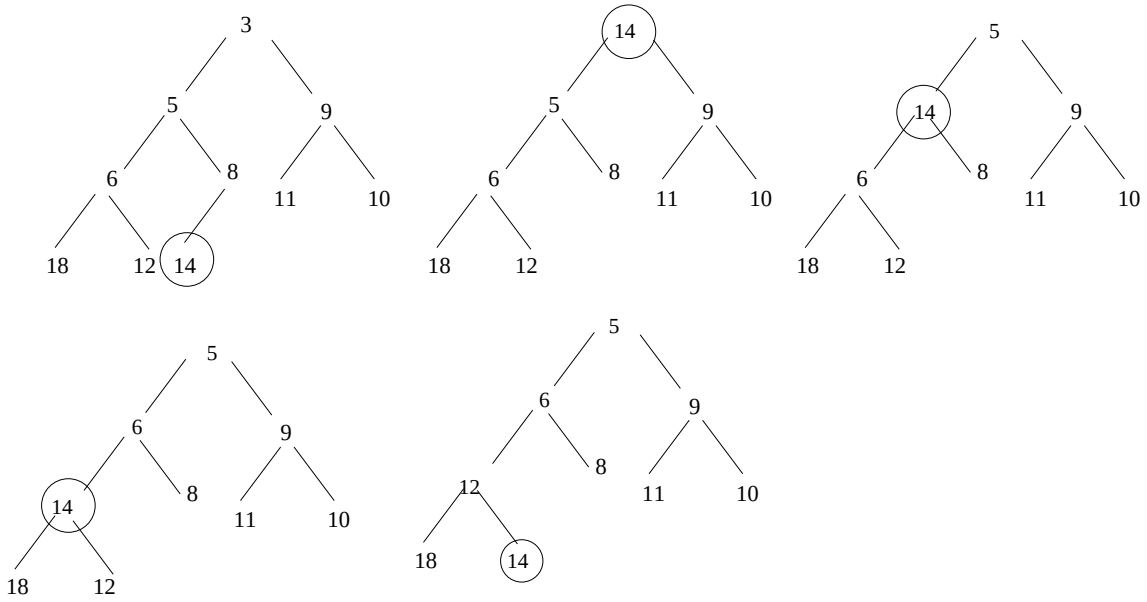


FIGURE 7.9 – Suppression du minimum d'un tas

On peut concevoir l'algorithme *tamiserVersLeBas* d'un nœud d'adresse i dans un tas T comme suit.

```
tamiserVersLeBas(T,i)
x←racine(i)
f←filsMin(T,i)
tant que f≤ n et x>racine(f)
    racine(i)←racine(f)
    racine(f)←x
    i←f
    f←filsMin(T,i)
```

Dans cet algorithme, x désigne l'élément qui descend par tamisage dans l'arbre (14 dans l'exemple de la figure 7.9). On remarque que $x = racine(i)$ est un invariant de la boucle. La variable f désigne celui des fils de i, s'il existe, dont la racine est minimale. L'échange entre les racines de f et i ne nécessite que deux affectations puisque la valeur de $racine(i)$ est toujours préservée dans la variable x. Enfin, si x est supérieure à tous les descendant de l'arbre i passé en paramètre, elle descendra dans l'arbre jusqu'à devenir une feuille. Il est donc nécessaire dans la condition de la boucle, de s'assurer que le fils f retenu existe bien avant d'examiner sa racine en vue de la comparer à x. Cela se fait en comparant f à la taille n du tas.

La recherche du fils de i racine minimale peut se faire comme suit :

```
filsMin(T,i)
f=2*i
si (f<n et racine(f)>racine(f+1))
    f ← f+1
```

Mise en œuvre en Java

Il est maintenant possible de définir une classe Java appelée Tas, permettant de manipuler cette structure de données. Elle comprend deux champs : un tableau d'entiers T, zone mémoire mise à disposition pour mémoriser l'arbre et un entier *taille* destiné à recevoir la taille effective de l'arbre (le tableau n'est pas nécessairement complètement utilisé). Elle comprend un constructeur créant un arbre vide, le champ T étant initialisé par un paramètre du constructeur. Un autre constructeur crée un arbre dont la taille est indiquée en paramètre.

```
public class Tas{

    private int taille;
    private int [] T;

    Tas(int [] tab){
        taille = 0;
        T = tab;
    }

    Tas(int [] tab, int n){
        taille = n;;
        T = tab;
    }

    boolean vide() {return(taille==0);}
    int getTaille() {return  taille;}
}
```

On peut alors définir la méthode *tamiserVersLeBas* et en déduire immédiatement celle qui extrait le minimum du tas.

```
void tamiserVersLeBas(int i){
   int x=T[i],f=2*i;

   if(f<taille && T[f]>T[f+1]) f++;
   while (f<=taille && x>T[f]){
      T[i] = T[f]; T[f] = x;
      i = f; f=2*i;
      if(f<taille && T[f]>T[f+1]) f++;
   }
 }

int extraireMin(){
   int r = T[1];
   T[1]=T[taille--];
   tamiserVersLeBas(1);
   return r;
}
```

Clairement, les algorithmes d'infiltration et d'extraction du minimum sont en $\Theta(log(n))$ dans le pire des cas, puisque le nombre d'échanges est au plus égal à la profondeur de l'arbre.

7.5.4 Transformation d'une liste en tas

Une liste contenue dans la partie $T[1..n]$ d'un tableau peut être transformée en tas de la façon suivante. On remarque tout d'abord que si les deux fils d'un arbre binaire sont déjà organisés en tas, transformer l'arbre en tas se réduit à placer correctement sa racine par un tamisage vers le bas. L'idée est alors de transformer successivement tous les sous-arbres en tas en partant des plus profonds. Plus précisément, on parcourt de droite à gauche tous les niveaux, depuis celui qui est juste au dessus des feuilles, jusqu'à celui portant la racine. A chaque itération, le nœud courant est la racine d'un sous-arbre dont les fils sont déjà des tas. Il est alors lui-même transformé en tas par tamisage vers le bas de sa racine. Le premier sous-arbre considéré est le père de la feuille la plus à droite, soit celui d'indice $i = taille/2$. Puis i est décrémenté jusqu'à 1, ce qui assure le parcours des niveaux comme indiqué.

```
void mettreEnTas(){
   for(int i=taille/2; i>0; i--)
      tamiserVersLeBas(i);
}
```

Complexité de la transformation

Si l'arbre transformé en tas a pour profondeur k, il possède $k+1$ niveaux, numérotés de 0 à k. Pour tout $i \in \{0, \ldots k-1\}$, le niveau i possède 2^i nœuds, le dernier niveau en possédant au moins 1 et au plus 2^k. La taille n de l'arbre est donc telle que

$$1 + \sum_{i=0}^{k-1} 2^i \ \leq \ n \ \leq \ \sum_{i=0}^{k} 2^i \qquad \text{c'est-à-dire} \qquad 2^k \ \leq \ n \ \leq \ 2^{k+1} - 1$$

Pour tout $i \in \{0, \ldots k-1\}$, chacun des 2^i nœuds du niveau i subit un tamisage vers le bas qui requiert au plus $k - i$ échanges. On en déduit que dans le pire des cas, le nombre d'échanges effectués pour transformer une liste de taille n en tas est :

$$C(n) \ = \ \sum_{i=0}^{k-1} (k-i)2^i \ = \ \sum_{i=1}^{k} i2^{k-i} \ = \ 2^k \sum_{i=1}^{k} i(\frac{1}{2})^i$$

On rappelle que :

$$\forall x \neq 1,\ \sum_{i=0}^{k} x^i \ = \ \frac{1-x^{k+1}}{1-x}$$

On obtient alors par dérivation :

$$\sum_{i=1}^{k} i x^{i-1} \ = \ \frac{kx^{k+1}-(1+k)x^k+1}{(1-x)^2} \ < \ \frac{1}{(x-1)^2} \qquad \text{si } x<1$$

et par suite, en multipliant par x : $\forall x<1,\ \sum_{i=1}^{k} i x^i \ < \frac{x}{(x-1)^2}$ et donc $\sum_{i=1}^{k} i(\frac{1}{2})^i < 2$. Comme de plus $2^k \leq n$, on obtient $C(n) \leq 2n$.

Il est par ailleurs facile de s'apercevoir que :

$$C(n) \ = \ 2^k \sum_{i=1}^{k} i(\frac{1}{2})^i \geq 2^k > \frac{n}{2} \qquad \text{dès que } k \geq 2$$

On déduit de cette démonstration que $C(n) \in \Theta(n)$.

Proposition 7.19 *La complexité en temps de l'algorithme de transformation d'une liste en tas est linéaire en la taille de la liste.*

7.5.5 Adjonction d'un élément à un tas

L'adjonction d'un élément à un tas est duale de l'extraction du minimum. Elle consiste à placer l'élément à rajouter comme dernière feuille du tas, puis à le faire remonter par un procédé de tamisage, pour le placer de telle sorte à respecter la structure de tas, comme illustré sur la figure 7.10.

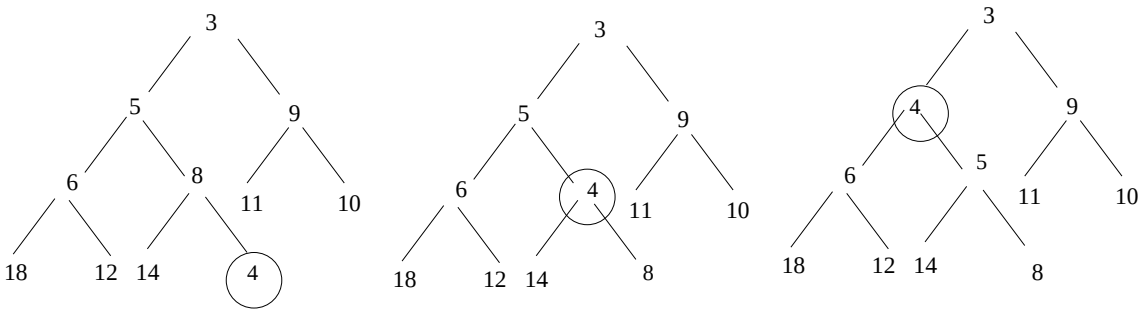


FIGURE 7.10 – Adjonction de l'élément 4 au tas

L'algorithme proposé ci-dessous consiste à tamiser vers le haut l'élément d'adresse i d'un tas. Comme pour le tamisage vers le bas, on fait en sorte que $racine(i) = x$ soit un invariant de la boucle. Ceci permet l'échange de $racine(i)$ et $racine(pere)$ en deux affectations seulement.

```
tamiserVersLeHaut(T,i)
x←racine(i)
pere←i/2
tant que (pere>0 et x<racine(pere))
    racine(i)←racine(pere)
    racine(pere)←x
    i←pere
    pere←pere/2
```

Si l'élément x que l'on tamise vers le haut est le minimum du tas, il finit par arriver à la racine et la variable *pere* prend alors la valeur 0. Il faut dans ce cas sortir de la boucle sous peine de débordement. Ceci explique la nécessité de vérifier que $pere > 0$ dans la condition de la boucle. Toutefois, on remarque que l'on peut utiliser la case d'indice 0 du tableau pour y placer x en sentinelle. Ceci assure dans tous les cas la sortie de la boucle sans avoir à tester la variable *pere* à chaque itération. On obtient ainsi la méthode Java $tamiserVersLeHaut$ utilisée en suite par la méthode *adjonction*.

```
void tamiserVersLeHaut(int i){
   int  x= T[i];
   T[0]=x;
   for (int p=i/2; x < T[p]; p/=2){
      T[i] = T[p]; T[p] = x; i = p;
   }
}

void adjonction(int x){
   T[++taille] = x;
   tamiserVersLeHaut(taille);
}
```

L'algorithme d'adjonction est évidemment en $\Theta(log_2(n))$, où n est la taille du tas.

7.5.6 Application : Tri par Tas

On a déjà évoqué une application des structures de *tas* à l'algorithme de Dijkstra, recherchant un plus court chemin entre deux sommets d'un graphe valué.

Une autre application très connue est celle d'un algorithme efficace de tri. La liste à trier est une liste d'entiers de taille n mémorisée, à partir de la case 1, dans un tableau T de taille $n + 1$. L'algorithme consiste :

- dans un premier temps, à transformer la liste en tas,
- puis à supprimer le minimum du tas (celui-ci n'occupe plus alors que $T[1 .. n - 1]$), recopier ce minimum en $T[n]$ et recommencer l'opération sur $T[1 .. n - 2]$, $T[1 .. n - 3]$, ...

La liste se trouve ainsi triée par ordre décroissant. On obtient ainsi le programme Java :

```
class triParTas{

   static void  tri(int [] T){
      Tas H = new Tas(T, T.length-1);
      H.mettreEnTas();
      for (int i = T.length-1; i>=1; i--)
         T[i] = H.extraireMin();
    }
}
```

Il est bien sûr possible de réaliser un tri par ordre croissant, il suffit alors de prendre l'ordre inverse de celui considéré (tout nœud du tas est ainsi supérieur à ses fils).

Complexité

L'algorithme du tri par tas commence par la mise en tas dont on a montré qu'elle est linéaire en la taille de la liste. Puis il effectue n extractions successives sur des tas dont la taille i varie de n à 2. On a vu que chacune de ces extractions est en $log(i)$ dans le pire des cas. Par suite le coût de la boucle `for` est

$$\sum_{i=2}^{n} log(i) \ \in \Theta(nlog(n))$$

La mise en tas est donc négligeable devant la somme des extractions successives.

Proposition 7.20 *Le tri par tas est en $nlog(n)$.*

7.6 Les arbres quelconques

Un arbre est dit *quelconque* si le nombre des fils de ses nœuds n'est pas borné a priori : c'est par exemple le cas des arbres généalogiques, comme illustré par la figure 7.11 ci-dessous.

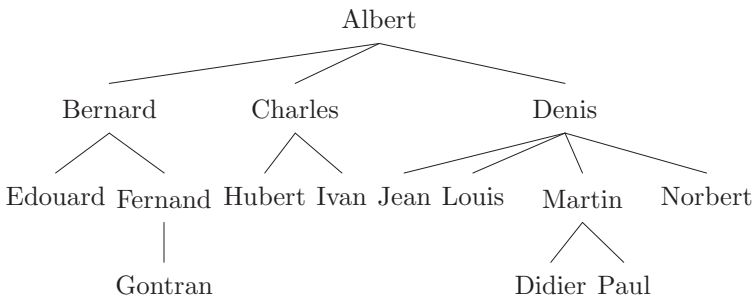


FIGURE 7.11 – Un exemple d'arbre quelconque : l'arbre généalogique

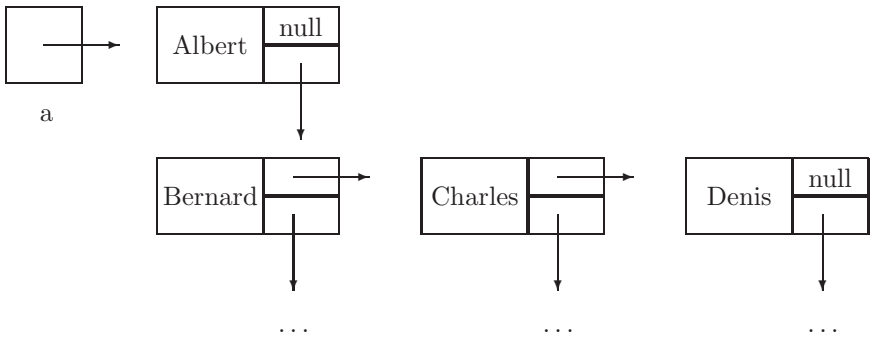
Le nombre des fils d'un nœud étant indéterminé, ceux-ci sont représentés sous forme de liste. Une telle liste d'arbres est appelée *forêt*.

Dans la pratique, arbres et forêts sont codés en machine par un même type d'objets, un arbre étant considéré comme une forêt réduite à un unique élément. Arbres et forêts, s'il ne sont pas vides, seront ainsi représentés par un triplet constitué de la racine du premier élément de la forêt, de la forêt de ses fils et de la forêt de ses frères suivants.

Ainsi, l'arbre de la figure 7.11 peut être codé comme suit :

7.6.1 Description abstraite

Definition 5 *Un* arbre quelconque *sur un ensemble* D *est :*
- *soit un élément particulier appelé* arbre vide
- *soit un couple* a = (r, fa, fs) *où*
  - r *est un élément de D appelé* racine *de l'arbre*
  - fa *est un arbre quelconque appelé, s'il n'est pas vide,* fils aîné *de* a



  - fs *est un arbre quelconque appelé, s'il n'est pas vide,* frère suivant *de* a

- *Si* fs *n'est pas vide, un* frère *de* a *est soit* fs*, soit l'un des frères de ce dernier.*
- *Si* fa *est vide, on dit que l'arbre* a *est une* feuille*. Dans le cas contraire, un* fils *de* a *est soit son fils aîné, soit l'un des frères de ce dernier.*

- *Si* a *n'est pas vide, on appelle* nœud *de* a
  - *soit sa racine*
  - *soit un* nœud *de l'un de ses fils.*

- *Un* sous-arbre *de* a *est :*
  - *soit* a *lui-même*
  - *soit un* sous-arbre *de l'un de ses fils, s'il en existe.*

Les primitives

Soit D un ensemble. Dans la définition qui suit, *a, fa* et *fs* désignent des arbres quelconques sur D, et r un élément de D.

nouvel-arbre(r, fa, fs) : renvoie l'arbre racine r, de fils aîné *fa* et de frère suivant *fs*.
arbreVide : renvoie l'arbre vide.
vide(a) : renvoie une valeur booléenne indiquant si a est l'arbre vide ou non.
racine(a) : n'est défini que si a n'est pas vide. Renvoie la racine de a.
fa(a) : n'est défini que si a n'est pas vide. Renvoie le fils aîné de a.
fs(a) : n'est défini que si a n'est pas vide. Renvoie le frère suivant de a.
feuille(a) : renvoie vrai si a est une feuille.

7.6.2 Parcours d'arbres quelconques

On considère deux type de parcours :
- le parcours d'un arbre proprement dit, qui consiste à visiter chacun de ses nœuds
- le parcours d'une forêt, qui consiste parcourir l'arbre ainsi que chacun de ses frères.

Parcours d'un arbre

Les parcours des arbres quelconques généralisent ceux des arbres binaires. On donne ici le schéma le plus général d'un parcours gauche-droite en profondeur d'abord, qui consiste, comme illustré par la figure 7.12, après avoir atteint un nœud pour la première fois, à visiter les nœuds de chacun de ses fils, pris de gauche à droite, avant d'y revenir une seconde fois en remontant. Chacun des deux passages sur le nœud peut donner lieu à un traitement sur ce nœud. On obtient ainsi une première version du parcours :

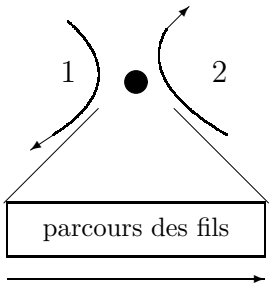


FIGURE 7.12 – Les visites d'un nœud lors d'un parcours gauche-droite en profondeur

```
parcours_arbre(a)
si non vide(a)
    traitement 1
    a ← fa(a)                          // descente sur le fils aîné
    tant que non vide(a) faire         // parcours des éléments de la liste des fils
       parcours_arbre(a)
       a ← fs(a)
    traitement 2
```

On remarque que le test `vide(a)` se fait deux fois sur tout sous-arbre propre de l'arbre initial : une première fois dans l'évaluation de la condition de la boucle **tant que** puis au début du parcours. Pour éviter cette redondance, on peut décider de supprimer le test en début de parcours, sachant que celui-ci ne s'appliquera qu'à des arbres non vides. On obtient ainsi l'algorithme de la figure 7.13.

```
parcours_arbre(a)
traitement 1
a ← fa(a)
tant que non vide(a) faire
    parcours_arbre(a)
    a ← fs(a)
traitement 2
```

FIGURE 7.13 – Parcours récursif d'un arbre non vide

Le parcours de l'arbre est dit en *pré-ordre* ou en *ordre préfixe* lorsque tout nœud est traité avant ses descendants, c'est-à-dire s'il ne subit que le traitement 1. Il est dit en *post-ordre* ou en *ordre postfixe* lorsque tout nœud est traité près ses descendants, c'est-à-dire s'il ne subit que le traitement 2.

Si le traitement consiste à afficher la racine de l'arbre, les deux parcours de l'arbre de la figure 7.11 donneront les résultats suivants :

Ordre préfixe : `Albert Bernard Edouard Fernand Gontran Charles Hubert Ivan Denis Jean Louis Martin Olivier Paul Norbert`

Ordre postfixe : `Edouard Gontran Fernand Bernard Hubert Ivan Charles Jean Louis Olivier Paul Martin Norbert Denis Albert`

```
prefixe_arbre(a)
traitement
a ← fa(a)
tant que non vide(a) faire
    prefixe_arbre(a)
    a ← fs(a)
```

```
postfixe_arbre(a)

a ← fa(a)
tant que non vide(a) faire
    postfixe_arbre(a)
    a ← fs(a)
traitement
```

FIGURE 7.14 – Les deux ordres de parcours des arbres quelconques non vides

Parcours de forêt

Une autre solution, proposée figure 7.15, consiste à parcourir toute une forêt : après un premier traitement de l'arbre courant, on rappelle récursivement le parcours sur la forêt des fils ; à la remontée, l'arbre subit un second traitement avant parcours de la forêt de ses frères suivants.

```
parcours_forêt(a)
si non vide(a)
    traitement 1
    parcours_forêt(fa(a))
    traitement 2
    parcours_forêt(fs(a))
```

FIGURE 7.15 – Parcours récursif d'une forêt

Comme c'est la cas pour les arbres, les forêts peuvent être parcourues *pré-ordre* ou en *post-ordre*, selon les schémas indiqués sur la figure 7.16

```
prefixe_forêt(a)
si non vide(a)
    traitement
    prefixe_forêt(fa(a))
    prefixe_forêt(fs(a))
```

```
postfixe_forêt(a)
si non vide(a)
    postfixe_forêt(fa(a))
    traitement
    postfixe_forêt(fs(a))
```

FIGURE 7.16 – Les deux ordres de parcours des forêts

Cette solution peut s'utiliser pour parcourir un seul arbre, si celui-ci n'a pas de frère suivant : le parcours des frères est alors l'instruction vide. Ainsi, ces algorithmes peuvent s'appliquer à l'arbre de la figure 7.11, et donnent les mêmes résultats que la première version des parcours.

Application : calcul de permutations et forêt virtuelle

On se propose d'afficher, à raison de une par ligne, chacune des $n!$ permutations des éléments de l'ensemble $I_n = \{0, \ldots, n-1\}$. Ce problème peut se résoudre en parcourant une forêt composée

de n arbres n-aires de profondeur $n-1$, dont les racines respectives sont les éléments de I_n. Tout nœud qui n'est pas une feuille possède à son tour n fils, rangés par ordre croissant, chacun étant un élément de I_n. Les permutations des éléments de I_n sont les chemins sans répétition d'une racine à une feuille. L'idée est donc de parcourir cette forêt en profondeur, en abandonnant la descente dès que l'on trouve une répétition. On mémorise dans un tableau T de longueur n les nœuds sur lesquels on descend, à condition qu'ils soient des candidats acceptables pour obtenir une permutation, c'est-à-dire qu'ils ne soient pas égaux à un de leur ancêtres. On sait que l'on est sur une feuille si le tableau est plein, et dans ce cas-là on l'imprime. On remarque que si un nœud est étiqueté par a et n'est pas une feuille, son fils aîné est 0. De plus, tout nœud $a < n-1$ a pour frère suivant est $a+1$.
Ainsi, la forêt que l'on parcourt est ici purement *virtuelle* en ce sens qu'elle n'est pas une structure de donnée du programme : on est toujours en mesure de connaître le fils ou le frère d'un nœud à partir de la valeur de celui-ci et de l'indice du dernier élément mémorisé dans le tableau T. Nul n'est besoin d'implanter réellement la forêt.

```
public class perm{
   static int []T;
   static int n;

   static void afficher(){
      for(int i=0; i<n; i++)
         System.out.print(T[i]+" ");
      System.out.println();
   }

   static boolean acceptable(int a, int i){
      for(int k=0; k<i; k++)
         if(a==T[k]) return false;
      return true;
   }

   static void permutation(int nb){
      n = nb;
      T= new int[n];
      parcours_foret(0,0);
   }

   static void parcours_foret(int a, int i){
      if(a<n){   //T[1..(i-1)] contient les ancêtres de a
         if(acceptable(a, i)){
            T[i]=a;
            if(i==n-1)
               afficher();
            else
               parcours_foret(0, i+1);  //Descente sur le fils aîné
         }
         parcours_foret(a+1, i);          //Passage au frère suivant
      }
   }

   public static void main(String[] args){
      permutation(Integer.parseInt(args[0]));
   }
}
```

On obtient alors la trace suivante :

```
$ java perm 3
0 1 2
0 2 1
1 0 2
1 2 0
2 0 1
2 1 0
```

On remarque enfin que l'on peut se passer du paramètre a en mémorisant le nœud courant dans $T[i]$. On modifie ainsi le programme de la façon suivante :

```
static boolean acceptable(int i){ //Le noeud courant est en T[i]
   for(int k=0; k<i; k++)
      if(T[i]==T[k])
         return false;
   return true;
}

static void permutation(int nb){
   n = nb;
   T= new int[n];
   T[0]=0;  //Le noeud courant est en T[0]
   parcours_foret(0);
}

static void parcours_foret(int i){ //Le noeud courant est en T[i]
   if(T[i]<n){
      if(acceptable(i)){
         if(i==n-1)
            afficher();
         else{
            T[i+1]=0;  //Le fils aine de T[i] en T[i+1]
            parcours_foret(i+1);
         }
      }
      T[i]++; //Passage au frere suivant
      parcours_foret(i);
   }
}
```

7.6.3 Mise en œuvre en Java de la structure d'arbre quelconque

Une classe Java mettant en œuvre des arbres quelconques (nécessairement non vides) dont les nœuds portent des chaînes de caractères, comportera donc trois champs. L'un, de type `String` contient l'information portée par la racine, les deux autres sont à leur tour des arbres quelconques et représentent respectivement le fils aîné et le frère suivant de `this`.

```
public class AQ{

  String nom;
  AQ fa, fs;

  AQ (String n,AQ a, AQ s){
```

```
    nom = n; fa = a; fs = s;
  }
```

Les primitives définies dans la description abstraite sont alors mises en œuvre de la façon suivante.

nouvel-arbre(r, fa, fs)	:	`new AQ(r, fa, fs)`
arbreVide	:	`null`
vide(a)	:	`a==null`
racine(a)	:	`a.nom`
fa(a)	:	`a.fa`
fs(a)	:	`a.fs`
feuille(a)	:	Si *a* n'est pas vide, `a.fa==null`

On peut alors calculer la *taille* d'un arbre ou d'une forêt en mettant en œuvre les algorithmes de parcours du paragraphe précédent.

```
int taille_arbre(){
  int nb = 1;
  AQ a = fa;
  while(a!=null){
    nb+=a.taille_arbre();
    a=a.fs;
  }
  return(nb);
}

static int taille_foret(AQ a){
  if(a==null) return 0;;
  return(taille_foret(a.fa)+taille_foret(a.fs)+1);
}

int taille_foret(){
  return(taille_foret(this));
}
```

Exercice 7.1 *Compléter la classe* `AQ` *en répondant aux questions suivantes.*

1. *Donner les méthodes renvoyant la profondeur d'un arbre et d'une forêt.*
2. *Donner les méthodes renvoyant le minimum d'un arbre et d'une forêt.*
3. *Donner les méthodes qui impriment l'arbre et la forêt en traduisant la hiérarchie par une indentation.*
4. *Implanter en Java les files d'attente d'arbres quelconques.*
5. *Utiliser la question précédente pour écrire une méthode permettant d'imprimer l'arbre par niveaux.*
6. *Ecrire une méthode* position *qui étant donnée une chaîne de caractères s, renvoie le sous-arbre de racine s s'il existe.*
7. *Ecrire une méthode qui rajoute une chaîne de caractères* s1 *comme nouveau fils de* s *si* s *figure dans l'arbre.*
8. *Ecrire un programme permettant de tester les méthodes précédentes.*

Chapitre 8

Les graphes

8.1 Présentation informelle

Les *graphes*, ou *structures relationnelles*, modélisent des relations sur des objets. Ces objets sont les sommets du graphe et le fait qu'un objet s soit en relation avec un objet t se traduit par l'existence d'une *arête* s — t ou d'un *arc* $s \to t$, selon que la relation est symétrique ou cas.

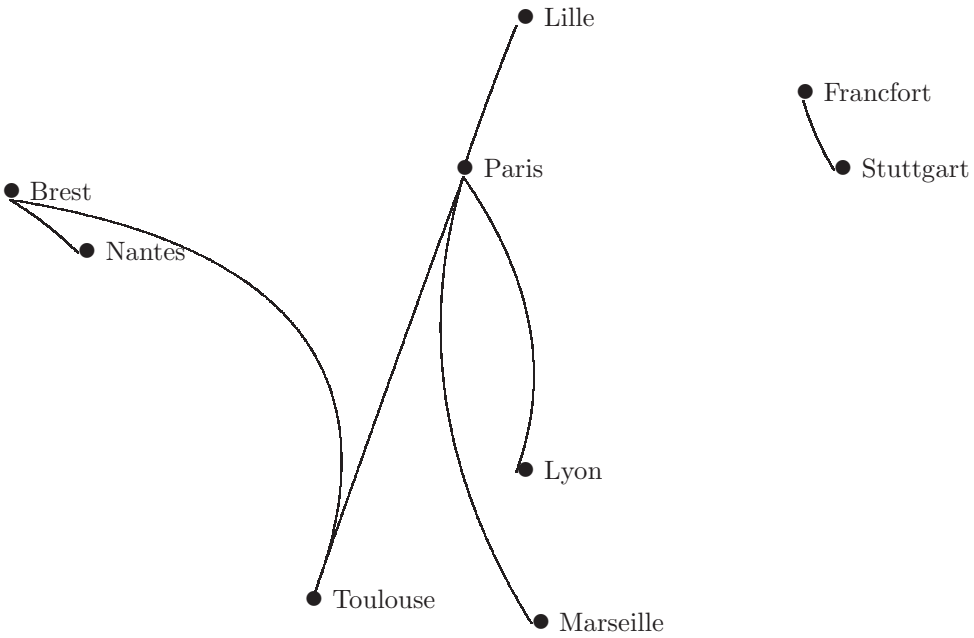


FIGURE 8.1 – Graphe non orienté : carte de liaisons aériennes

La figure 8.1 représente un graphe non orienté de liaisons aériennes : s'il existe une liaison d'un point s vers un point t, il existe une possibilité de retour de t vers s. Il s'agit donc d'une relation symétrique. Les sommets du graphe sont les villes. La présence d'une arête s—t traduit l'existence d'une liaison aérienne directe entre les villes s et t.
On peut alors se poser un certain nombre de questions comme :

- existe-il un moyen d'aller de la ville s à la ville t ?
- quel trajet minimise-t-il le nombre d'escales entre deux villes ?
- peut-on visiter chaque ville une et une seule fois (*problème du voyageur de commerce*)

Un autre problème classique est celui de l'ordonnancement des tâches à effectuer dans un processus (industriel, militaire ...) complexe. Le problème est modélisé par un graphe orienté dont les sommets sont les tâches. La présence d'un arc $s \to t$ indique que l'exécution de s doit être terminée avant que ne débute celle de t.

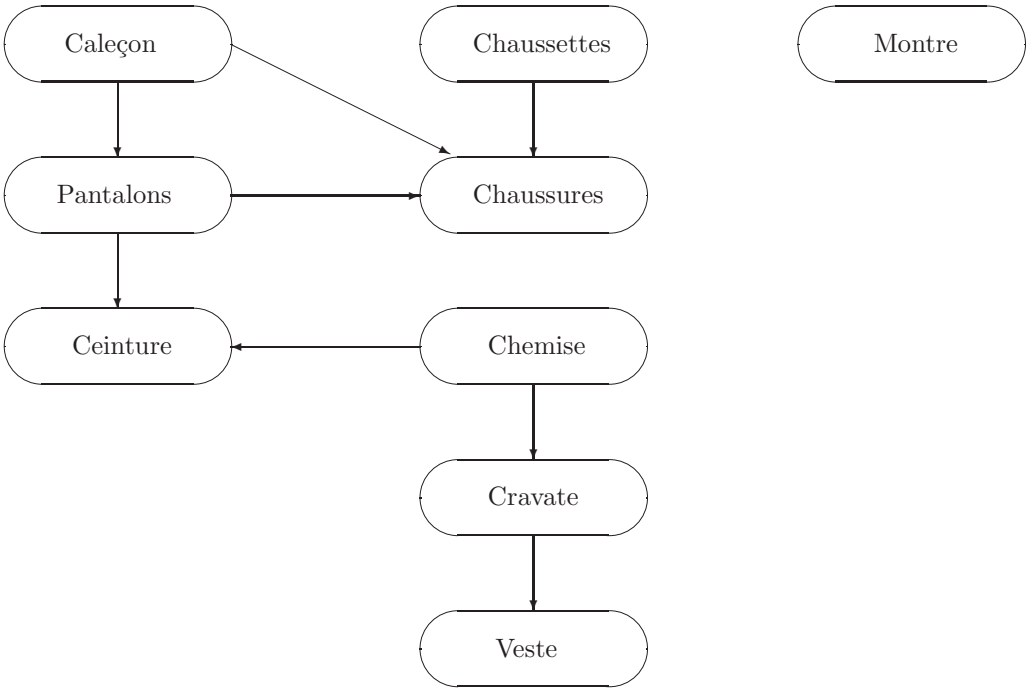


FIGURE 8.2 – Graphe orienté : ordonnancement des tâches pour vêtir le Pr. Tournesol

Par exemple, le graphe de la figure 8.2 représente la dépendance des tâches que doit effectuer le professeur Tournesol pour se vêtir. Le traitement d'un tel graphe consiste à trouver une indexation des tâches telles que, si t_i doit être exécutée avant t_j, alors $i < j$.

Considérons enfin un ensemble de villes reliées par des routes. Ces villes sont modélisées par les sommets d'un graphe, les routes entre deux villes sont des arcs ou des arêtes selon qu'elles sont à sens unique ou pas. A chaque route peut-être associée sa longueur, ce qui revient à associer à chaque arc ou arête du graphe un nombre appelé *valuation*. On dit alors que le graphe (orienté ou pas) est valué. Un problème classique consiste à rechercher un *plus court chemin* entre deux sommets d'un graphe valué.

8.2 Terminologie

Cette section présente la terminologie usuelle de la théorie des graphes.

Graphe non orienté : c'est un couple *(S,A)* où
- S est un ensemble fini d'éléments appelés sommets
- A est un ensemble fini de paires de sommets appelées *arêtes*.

L'arête $\{s, t\}$ est notée s —t. Les sommets s et t sont les extrémités de l'arête.

Graphe orienté : c'est un couple *(S,A)* où
- S est un ensemble fini d'éléments appelés sommets
- A est un ensemble fini de couples de sommets *distincts* appelées *arcs*.

L'arc *(s,t)* est noté $s \to t$. Le sommet s est l'extrémité initiale (*tail* en anglais) et t est l'extrémité terminale (*head* en anglais). On dit aussi que t est un *successeur* de s et que s est un prédécesseur de t.

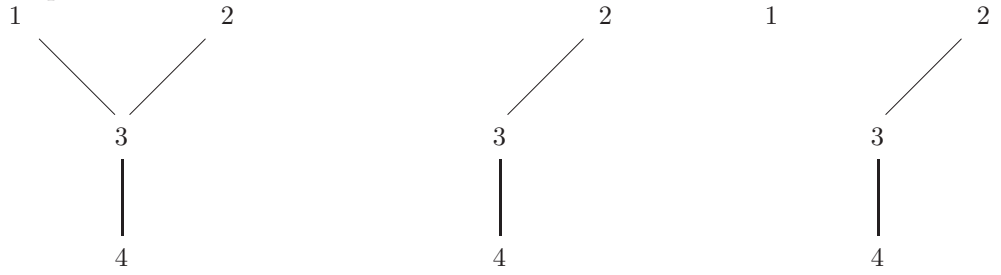
Graphe valué (on dit encore *pondéré* ou *étiqueté*). C'est un couple constitué
- d'un graphe *(S,A)* orienté ou pas
- d'une fonction $C : A \to \mathbb{R}$ appelée *valuation* ou *fonction de coût* ou encore *fonction de poids*.

On considère désormais un graphe *G= (S,A)*, orienté ou pas, valué ou pas.

Sous-graphe de G engendré par un sous-ensemble de sommets *S'* : c'est le graphe dont les sommets sont les éléments de *S'* et dont les arcs ou arêtes sont ceux dont les extrémités sont dans *S'*.

Graphe partiel de G engendré par un sous-ensemble *A'* de A : c'est le graphe *G'=(S, A')*.

Exemple 8.1

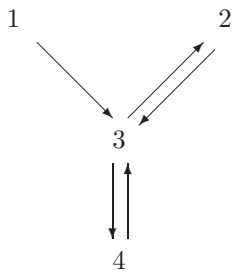


Graphe Sous-graphe Graphe partiel

Adjacence, incidence
- Deux arcs ou arêtes sont dits *adjacents* s'ils ont une extrémité commune.
- Dans un graphe non orienté, deux sommets sont dits *adjacents* s'ils sont reliés par une arête. On dit alors que cette arête est *incidente* à ces sommets.
- Dans un graphe orienté, un sommet s est dit *adjacent* à un sommet t, s'il existe un arc $s \to t$. Cet arc est dit *incident à s vers l'extérieur* et *incident à t vers l'intérieur*.

Degrés, demi-degrés
- Dans un graphe non orienté, le *degré* d'un sommet s est le nombre d'arêtes d'extrémités s.
- Dans un graphe orienté, on définit le demi-degré intérieur $d^{\circ -}(s)$ (respectivement extérieur $d^{\circ +}(s)$) comme le nombre d'arcs incidents à s vers l'intérieur (respectivement vers l'extérieur). Le degré de s est défini par $d^{\circ}(s) = d^{\circ +}(s) + d^{\circ -}(s)$.

Exemple 8.2



$$\begin{aligned} d^{\circ +}(3) &= 2 \\ d^{\circ -}(3) &= 3 \\ d^{\circ}(3) &= 5 \end{aligned}$$

Chaînes et chemins

- Un *chemin* (respectivement une *chaîne*) de longueur $k \geq 0$ est une suite de sommets $s_0, s_1, \ldots s_k$ telle que pour tout i, $0 \leq i \leq k-1$, il existe un arc $s_i \rightarrow s_{i+1}$ (respectivement une arête s_i— s_{i+1}).
- Un chemin (ou une chaîne) de longueur nulle est donc réduit à un seul élément.
- Si $s_0 = s_k$ et $k \geq 2$, le chemin (respectivement la chaîne) est appelé *circuit* (respectivement *cycle*).
- Un chemin (ou une chaîne) est dit *élémentaire* s'il ne contient pas plusieurs fois le même sommet, mis à part le premier qui peut être égal au dernier

Descendants et ascendants

- L'ensemble des *descendants* d'un sommet s est l'ensemble des sommets t tels qu'il existe un chemin (ou une chaîne) de s à t.
- L'ensemble des *ascendants* d'un sommet s est l'ensemble des sommets t tels qu'il existe un chemin (ou une chaîne) de t à s.

Dans le cas non orienté, les deux ensembles sont confondus.

Connexités

- Un graphe non orienté est dit *connexe* si pour toute paire de sommets, il existe une chaîne qui les relie.
- On appelle *composante connexe* d'un graphe tout sous-graphe connexe maximal (c'est-à-dire qui ne soit pas sous-graphe d'un autre sous-graphe connexe).
- Un graphe orienté est dit *fortement connexe* si pour tout couple de sommets, il existe un chemin qui les relie.
- On appelle *composante fortement connexe* d'un graphe orienté tout sous-graphe fortement connexe maximal (c'est-à-dire qui ne soit pas sous-graphe d'un autre sous-graphe fortement connexe).

Sur l'exemple 8.1, le graphe et le sous-graphe sont connexes, le graphe partiel ne l'est pas et comporte deux composantes connexes, sous-graphes respectifs engendrés par $\{1\}$ et $\{2, 3, 4\}$.
Le graphe de l'exemple 8.2 n'est pas fortement connexe. Il comporte deux composantes fortement connexes, sous-graphes respectifs engendrés par $\{1\}$, $\{2, 3, 4\}$.

8.3 Parcours de graphes

On peut en général envisager deux types de parcours de graphe.

Le parcours en profondeur d'abord On considère, dans un premier temps, un algorithme d'*exploration en profondeur* (depth first search). Il consiste, à partir d'un sommet donné, à suivre un chemin (ou chaîne) le plus loin possible, jusqu'à atteindre un sommet dont tous les successeurs, s'il y en a, ont déjà été visités, puis à faire des retours en arrière pour emprunter des chemins ignorés jusqu'alors. Cet algorithme est naturellement récursif.
On remarque de plus qu'il est possible que l'algorithme n'atteigne pas tous les sommets (ceci est évident quand il y a plusieurs composantes connexes). Pour parcourir l'intégralité du graphe, il convient donc de rappeler cet algorithme tant qu'il reste des sommets non visités.

Le parcours en largeur Il consiste à parcourir le graphe par niveaux : étant donné un sommet, on visite d'abord tous les sommets qui lui sont adjacents, puis les successeurs à ces derniers, etc. Généralisant le parcours par niveaux des arbres quelconques (cf. exercice 7.1, section 7.6.3), ce parcours utilise des files d'attente et est intrinsèquement itératif.

8.3.1 Parcours en profondeur

Le problème est de pouvoir décider, en cours de parcours, si un sommet a déjà été visité. Pour cela, on associe à chaque sommet une *marque* booléenne. Initialement toutes les marques ont la valeur *faux*.

```
parcours_profondeur(G)
pour tout sommet s de G
    marque(s) ← faux
pour tout sommet s de G
    si non marque(s)
       dfs(s)
```

```
dfs(s)

marque(s) ← vrai
pour tout sommet t adjacent à s
    si non marque(t)
       dfs(t)
```

FIGURE 8.3 – Parcours d'un graphe en profondeur d'abord

Comme indiqué sur la figure 8.3, on définit un premier algorithme d'exploration en profondeur *dfs* qui, à partir d'un sommet non marqué s, visite tous les descendants de s qui ne sont pas encore marqués.
Le parcours du graphe consiste à appeler l'algorithme *dfs* à partir de sommets non marqués et ce, tant qu'il en existe.

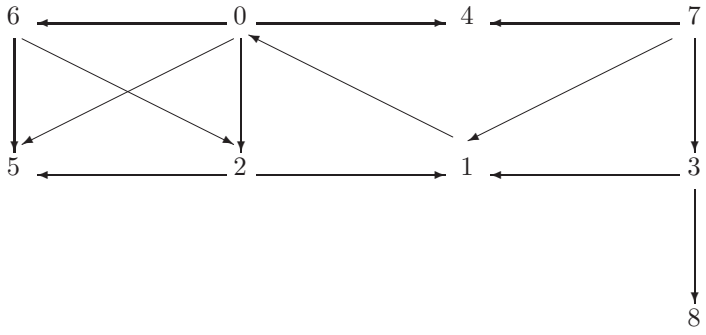


FIGURE 8.4 – Exemple de graphe orienté

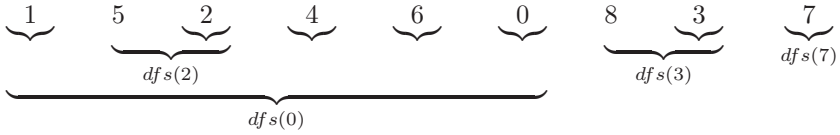
Supposons que l'on parcoure les sommets du graphe de la figure 8.4 par ordre croissant. On visite les sommets dans l'ordre suivant :

$$\underbrace{0\ \underbrace{2\ \underbrace{1}\ \underbrace{5}}_{dfs(2)}\ \underbrace{4}\ \underbrace{6}}_{dfs(0)}\ \underbrace{3\ \underbrace{8}}_{dfs(3)}\ \underbrace{7}_{dfs(7)} \tag{8.1}$$

On a fait figurer sur le schéma les divers appels à l'algorithme *dfs* et les sommets visités par chacun d'eux.

Parcours en pré-ordre et en post-ordre Les parcours préfixe et postfixe des arbres se généralisent aux graphes, selon que les sommets visités par le parcours en profondeur sont traités avant ou après leur descendants (cf. figure 8.5).

En admettant que le traitement d'un sommet consiste à l'afficher, le parcours en pré-ordre aura pour résultat l'affichage (8.3). En revanche l'affichage en post-ordre produit la ligne ci-dessous.



(8.2)

```
dfs_prefixe(s)

marque(s) ← vrai
traiter(s)
pour tout sommet t adjacent à s
    si non marque(t)
       dfs_prefixe(t)
```

```
dfs_postfixe(s)

marque(s) ← vrai
pour tout sommet t adjacent à s
    si non marque(t)
       dfs_postfixe(t)
traiter(s)
```

FIGURE 8.5 – Explorations en profondeur préfixe et postfixe d'un graphe

8.3.2 Parcours en largeur

L'algorithme de base est une *exploration en largeur* (*breadth first search* en anglais) dont le principe consiste à visiter tous les successeurs non marqués du sommet courant s avant de visiter les autres descendants. L'algorithme utilise une file d'attente dans laquelle sont mémorisés les sommets adjacents au sommet courant pendant que l'on visite les sommets de même niveau. Lorsque la visite de ces sommets est terminée (on ne peut aller plus avant en largeur), on retire un sommet en attente pour visiter ses successeurs non encore marqués.
Cet algorithme ne marque en général pas tous les sommets du graphe et doit être rappelé itérativement par l'algorithme de parcours en largeur tant qu'il subsiste des sommets non marqués.

```
parcours_largeur(G)
pour tout sommet s de G
    marque(s) ← faux
pour tout sommet s de G
    si non marque(s)
       bfs(s)
```

```
bfs(s)
f← creerVide()
enfiler(s,f)
marque(s) ←vrai
repeter
    s←defiler(f)
    pour tout sommet t adjacent à s
       si non marque(t)
          marque(t)←vrai
          enfiler(t,f)
tant que non vide(f)
```

FIGURE 8.6 – Parcours d'un graphe en largeur

L'algorithme décrit sur la figure 8.6 fait intervenir les primitives sur les files, introduites dans la section 6.3.

Sur l'exemple de la figure 8.4, on obtient l'ordre de parcours suivant :

$$\underbrace{0 \quad 2 \quad 4 \quad 5 \quad 6 \quad 1}_{bfs(0)} \quad \underbrace{3 \quad 8}_{bfs(3)} \quad \underbrace{7}_{bfs(7)} \tag{8.3}$$

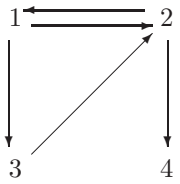
8.4 Représentations des graphes

Il existe deux représentations classiques des graphes, selon que l'on privilégie les sommets ou les arcs (ou arêtes).

8.4.1 Les matrices d'adjacence

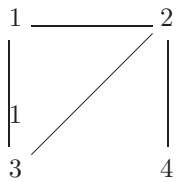
Dans ce type de représentation, ce sont les sommets qui sont privilégiés. Le graphe est représenté par une matrice carrée indexée sur les sommets. Les coefficients sont des booléens qui indiquent l'existence d'un arc ou d'une arête reliant les sommets en indice. La matrice est évidemment symétrique quand le graphe n'est pas orienté.

Exemple 8.3



est représenté par la matrice

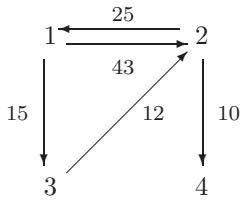
	1	2	3	4
1	F	V	V	F
2	V	F	F	V
3	F	V	F	F
4	F	F	F	F



est représenté par la matrice

	1	2	3	4
1	F	V	V	F
2	V	F	V	V
3	V	V	F	F
4	F	V	F	F

Dans le cas d'un graphe valué, les coefficients de la matrice sont les poids des arcs ou des arêtes. Il faut alors choisir une valeur particulière qui exprimera par convention l'absence de connexion entre deux sommets. Par exemple :



peut être représenté par la matrice

	1	2	3	4
1	-1	25	15	-1
2	43	-1	-1	10
3	-1	12	-1	-1
4	-1	-1	-1	-1

Cette représentation présente deux avantages :

1. l'accès aux arcs se fait en temps constant
2. l'accès aux prédécesseurs d'un sommet s est aisé : il suffit de parcourir la colonne indexée par s.

Elle présente deux inconvénients :

1. l'encombrement de la représentation est maximal. Il est en $\Theta(n^2)$ quel que soit le nombre d'arcs. Si celui-ci est faible par rapport au nombre de sommets, la matrice d'adjacence est creuse : la plupart des coefficients sont égaux à F.
2. le parcours des sommets adjacents à un sommet donné est en $\Theta(n)$ même si ce sommet possède peu de sommets adjacents. Il consiste en effet, si l'ensemble des sommets est $\{1, \ldots, n\}$ et si G désigne la matrice, à effectuer la boucle

```
Pour t variant de 1 à n
   si G[s,t]
      traiter(t)
```

Coût des parcours Les parcours visitent chacun des n sommets une et une seule fois. Chaque fois qu'un sommet s est visité, l'algorithme effectue un parcours de tous les autres sommets pour examiner s'ils lui sont adjacents et examiner leur marque. Les parcours sont donc en $\Theta(n^2)$.

Mise en oeuvre en Java

Voici une mise en oeuvre des graphes à l'aide de matrices d'adjacence comportant une méthode qui affiche les sommets en ordre préfixe. Le constructeur prend en paramètre le nombre n de sommets et initialise tous les coefficients de la matrice d'adjacence à *faux*. Cette dernière sera ensuite mise à jour par appels successifs à la méthode `ajoute_arc`.

```
class Graph{

  boolean [][] arc;
  private int  ns;
  private boolean [] marque;

 Graph (int n){
      ns=n;
      arc = new boolean[ns][ns];
      for(int s=0; s<ns; s++)
         for(int t=0; t<ns; t++)
            arc[s][t]=false;
      marque = new boolean[ns];
  }

 void ajoute_arc(int s, int t)
      arc[s][t]=true;
 }

 void dfs_prefixe (int s){
    marque[s] = true;
    System.out.print(s + " ");
    for(int t=0; t<ns; t++)
       if (arc[s][t])
          if (!marque[t])
             dfs_prefixe(t);
  }

  void affichage_prefixe (){
      int s;

      for(s=0; s<ns;s++) marque[s]=false;
```

```
        for(s=0; s<ns;s++)
            if (!marque[s])
                dfs_prefixe(s);
    }
}
```

Voici un exemple de programme permettant de tester l'affichage. Le premier argument du programme est le nombre de sommets. Chaque arc ou arête de s vers t est indiqué en donnant les entiers s et t en arguments consécutif du programme.

```
public class ProgGraph{

   public static void  main (String [] args){
      if(args.length>=1){
         Graph g = new Graph(Integer.parseInt(args[0]));
         for(int i=1; i<args.length; i+=2)
         g.ajoute_arc(Integer.parseInt(args[i]), Integer.parseInt(args[i+1]));
         g.affichage_prefixe();
   }
}
```

Avec la commande

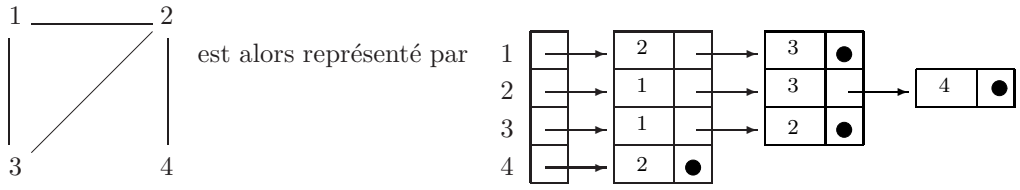
```
java ProgGraph 9 0 6 0 5 0 2 1 0 2 5 3 8 3 1 6 5 6 2 7 4 7 3 7 1
```

On obtient : `0 2 5 6 1 3 8 4 7`

Exercice 8.1 *Programmer le parcours en largeur avec cette représentation.*

8.4.2 Les listes d'adjacence

L'autre alternative consiste à associer à chaque sommet la liste des sommets qui lui sont adjacents. Par exemple,



Dans la pratique, la représentation est constituée d'un tableau indexé sur les sommets du graphe, dont chaque cellule porte la liste des sommets adjacents au sommet qui l'indexe.

Les avantages de cette représentation sont les suivants :

1. l'encombrement est en $\Theta(n+p)$ où n est le nombre de sommets et p le nombre d'arcs (intéressant si le nombre d'arcs ou d'arêtes est en $O(n^2)$).
2. le parcours des sommets adjacents à un sommet s est en $d^{\circ+}(s) \leq n$. Si G est le tableau représentant le graghe, il se fait comme suit :

```
a ← debut((G[s]))
tant que (a≠ fin(G[s]))
```

```
            traiter(element(a))
            a ← suivant(a)
```

En revanche

1. il n'y a pas d'accès direct aux arcs : pour savoir si un sommet t est adjacent à un sommet s, il faut effectuer un parcours de la liste d'adjacence de s, qui est en $\Theta(n)$ dans le pire des cas.
2. la recherche des prédécesseurs d'un sommet nécessite le parcours de toute la structure [1].

Coût du parcours en profondeur Le parcours en profondeur visite chacun des n sommets une et une seule fois. Chaque fois qu'un sommet s est visité, l'algorithme effectue un parcours de ses successeurs (au nombre de $d^{\circ+}(s)$) pour examiner leur marque. Il y a donc n marquages et $p = \sum_{s \in S} d^{\circ+}(s)$ examens de marques. Le parcours en profondeur est donc en $\Theta(n+p)$.

Mise en oeuvre en Java

Les listes d'adjacence sont mises en oeuvre dans une classe `List` partiellement décrite ici. On utilise ici des listes circulaires, dont le dernier maillon pointe sur l'en-tête.

```
class List{
    int sommet;
    List suiv;

    List (int s, List l){
       sommet = s;
       suiv=l;
    }

    List(){
       suiv = this;
    }

    void rajoute(int s){
       suiv=new List(s, suiv);
    }
}
```

Les graphes peuvent alors implantés par la classe `Graph` ci-dessous. Le constructeur prend en paramètre le nombre n de sommets et initialise les listes d'adjacence à vide. Ces dernières seront ensuite mises à jour par appels successifs à la méthode `ajoute_arc`.

```
class Graph{

    List [] succ;
    int ns;
    private boolean [] marque;

    Graph (int n){
       ns = n;
       succ = new List [ns];
       for(int i=0; i<ns;i++)
          succ[i]= new List();
```

1. Il est possible de faire également figurer dans la représentation les listes des prédécesseurs : les algorithmes sont moins coûteux en temps mais requièrent davantage d'espace

```
    marque = new boolean [ns];
}

void ajoute_arc(int s, int t){
   succ[s].rajoute(t);
}

void dfs_prefixe (int s){

   marque[s] = true;
   System.out.print(s + " ");
   for(List a =succ[s].suiv; a!=succ[s]; a = a.suiv)
      if (!marque[a.sommet])
      dfs_prefixe(a.sommet);
}

void affichage_prefixe (){
   int s;

for(s=0; s<ns;s++) marque[s]=false;
   for(s=0; s<ns;s++)
      if (!marque[s]) dfs_prefixe(s);
}
```

Le programme utilisé dans la section précédente pour tester la mise en œuvre des graphes avec les matrices d'adjacence, peut être repris à l'identique pour cette représention.

Exercice 8.2 *Programmer le parcours en largeur avec cette représentation.*

8.5 Graphes orientés acycliques (dag)

Dans toute la section, *G* désigne un graphe orienté *(S,A)*.

Definition 6 (Dag) G *est dit* acyclique *s'il ne possède pas de circuit. On dit souvent* dag *(de l'anglais* directed acyclic graph*) pour* graphe orienté acyclique.

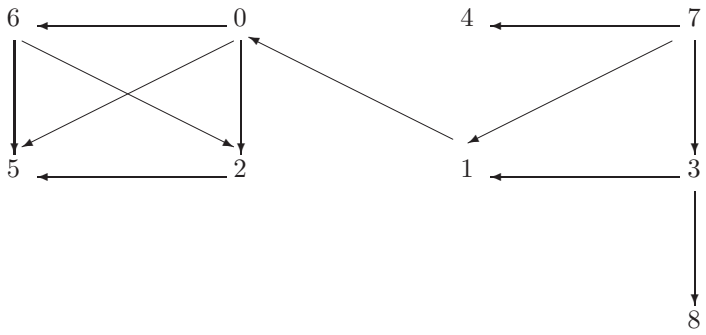


FIGURE 8.7 – Exemple de graphe orienté acyclique

Definition 7 *On appelle* source *de* G *tout sommet de demi-degré intérieur nul. On appelle* puits *de* G *tout sommet de demi-degré extérieur nul.*

Autrement dit, une source de G est un sommet sans prédécesseur, un puits est un sommet sans successeur. Par exemple, le graphe de la figure 8.7 a une seule source : le sommet 7. Les sommets 4, 5 et 8 sont des puits.

Notations On notera n le nombre des sommets de G et p le nombre de ses arcs. De plus, pour tout sous-ensemble *S'* de l'ensemble S des sommets, on notera *G(S')* le sous-graphe de G engendré par *S'*. Enfin, *Sources(G)* désignera l'ensemble de ses sources.

Proposition 8.1 *Si* G *est* acyclique*, il possède au moins une source et puits.*

Preuve Le graphe étant *acyclique*, les chemins ont une longueur au plus égale à *(n-1)*. Tout chemin de longueur maximale débute nécessairement sur une source et se termine sur un puits. □

8.5.1 Tri topologique

Le graphe G est maintenant supposé modéliser un problème d'ordonnancement des tâches décrit dans l'introduction de ce chapitre. On se propose de concevoir un algorithme qui, lorsque c'est possible, renvoie un tableau *num* indexé sur l'ensemble des sommets du graphe représentant une numérotation compatible des sommets, c'est-à-dire telle que s'il existe un arc $s \to t$ entre deux sommets, alors $num[s] < num[t]$.

Proposition 8.2 G *est acyclique si et seulement si il existe une numérotation compatible des sommets.*

Preuve Il est clair que la présence d'un circuit exclut toute numérotation compatible.
Supposons maintenant que le graphe soit acyclique, et considérons une suite de sous-graphes de G $G_0 = G, G_1 \ldots, G_i, G_{i+1} \ldots, G_{n-1}$, chacun d'eux étant obtenu à partir du précédent en ôtant l'une de ses sources (il en existe toujours au moins une puisque tous ces sous-graphes sont acycliques). Si l'on note s_i la source ôtée de G_i pour obtenir G_{i+1}, on obtient une numérotation compatible. En effet, considérons deux sommets s_i et s_j ainsi numérotés et supposons qu'il existe dans le graphe un arc $s_i \to s_j$. Puisque s_j est une source de G_j, s_i ne peut en être un sommet. Donc s_i est la source d'un sous-graphe G_i avec $i < j$. □

```
Tri_topologique(G)
deg←degreInt(G)
nb←-1
sources←sources(G)
pour tout sommet s de G
    num[s]← -1
tant que non vide(sources)
    s←supprimer(sources)
    nb←nb+1
    num[s]← nb
    pour tout sommet t adjacent à s
       deg[t]←deg[t] -1
       si deg(t)=0
          rajoute(t, sources)
    circuit ←nb+1<n
renvoyer(circuit,num)
```

```
degreInt(G)
pour tout sommet s de G
    deg[s]← 0
pour tout sommet s de G
    pour tout sommet t adjacent à s
       deg(t) ← deg(t) +1
renvoyer(deg)

sources(G)
deg←degreInt(G)
créerVide(S)
pour tout sommet s de G
   si deg(s) =0
      rajoute(s, S)
renvoyer(S)
```

FIGURE 8.8 – Tri topologique

On déduit de cette démonstration un algorithme pour trouver une numérotation compatible des sommets, s'il en existe. On suppose disposer d'un algorithme `degreInt` qui calcule le tableau des demi-degrés intérieurs des sommets du graphe et d'un algorithme `sources` qui calcule la liste des sources du graphe. L'algorithme donné sur la figure 8.8 utilise également les primitives usuelles sur les listes (les suppressions et rajouts se font à une position quelconque, par exemple en tête). Si le graphe contient un cycle, on obtiendra au bout d'un certain nombre d'itérations un sous-graphe sans sources, et donc certains sommets ne seront pas numérotés. On détectera l'éventuelle existence de circuits au fait que le nombre *nb+1* de sommets numérotés est inférieur au nombre n de sommets.

Exercice 8.3 *Programmer le tri topologique pour chacune des deux représentations des graphes.*

8.5.2 Décomposition par niveaux

Definition 8 *On appelle décomposition par niveaux de* G *l'ensemble des parties* non vides S_1, S_2, ..., S_k *de* S *définies par :*

- $S_1 = Sources(G)$
- $\forall i,\ 1 < i \leq k+1 \quad S_i = Sources(G(S - \bigcup_{1 \leq j < i} S_j))$
- $\forall i,\ 1 < i \leq k \quad S_i \neq \emptyset$
- $S_{k+1} = \emptyset$

Par exemple, la décomposition par niveaux du graphe de la figure 8.4 est la suivante :
$S_1 = \{7\}$ $S_2 = \{4, 3\}$ $S_3 = \{8, 1\}$ $S_4 = \{0\}$ $S_5 = \{6\}$ $S_6 = \{2\}$ $S_6 = \{5\}$.

Proposition 8.3 *Soit* S_1, ..., S_k *une décomposition par niveaux de* G. *Les parties* S_1, ..., S_k *sont disjointes. De plus, si* s *et* t *sont deux sommets de* G *tel que* t *est adjacent à* s *et s'il existe deux indices* a *et* b *tels que* $s \in S_a$ *et* $t \in S_b$, *alors* $a < b$.

Preuve Par construction, les éléments de la décomposition par niveaux sont des parties disjointes et non vides de l'ensemble des sommets. Considérons un sommet s et un sommet t qui lui est adjacent. Supposons qu'il existe un entier a tel que $s \in S_a$ et un entier b tel que $t \in S_b$. Le sommet t étant adjacent à s et t étant une source du sous-graphe G' engendré par $S - \bigcup_{1 \leq j < b} S_j$, s ne peut être un sommet de G'. Donc s appartient nécessairement à $\bigcup_{1 \leq j < b} S_j$. Donc il existe $j < b$ tel que $s \in S_j$ et $j = a$ puisque les ensembles de la décomposition sont disjoints deux à deux. □

Proposition 8.4 G *est acyclique si et seulement si la décomposition par niveaux de* G *est une partition de l'ensemble* S *des sommets.*

Preuve Les ensembles $S_1, S_2, \ldots, S_k$ étant disjoints deux à deux, pour qu'ils constituent une partition de S, il faut et il suffit que tout sommet du graphe soit dans l'un des ensembles S_i.
Partie directe Si G est acyclique, tout sous-graphe de G l'est aussi, et donc il possède des sources. Dans ce cas, $S - S_1$ est strictement inclus dans S puisque G possède au moins une source. De même, si $S - S_1$ n'est pas vide, $G(S - S_1)$ possède des sources et donc l'ensemble $S - S_1 - S_2$ est strictement inclus dans $S - S_1$ etc. Ainsi, si $S - \bigcup_{1 \leq j < i} S_j$ est non vide, on peut construire un ensemble $S - \bigcup_{1 \leq j \leq i} S_j$ strictement inclus dans le précédent. L'ensemble des sommets étant fini, il existe nécessairement un indice k tel que $S - \bigcup_{1 \leq j \leq k} S_j = \emptyset$ et donc tel que $S = \bigcup_{1 \leq j \leq k} S_j$.
Réciproque Supposons que la décomposition par niveaux constitue une partition du graphe G. Considérons un chemin $s_0, \ldots, s_l$ du graphe G. Il existe des éléments $S_{i_0}, \ldots, S_{i_l}$ de la partition qui contiennent respectivement chacun des sommets $s_0, \ldots, s_l$. D'après la proposition 8.3,

$$i_0 < i_1 < \cdots < i_l$$

Ceci exclut la possibilité que $s_0 = s_l$. □

```
DecompositionParNiveaux(G)

nb←0, niveaux← ∅
deg←degreInt(G)
sources←sources(G)

tant que non vide(sources)
    niveaux ← niveaux ∪ {sources}
    nouveau ← ∅                          //Calcul des nouvelles sources
    pour tout sommet s de sources
       nb←nb+1
       pour tout successeur t de s
           deg(t)←deg(t)-1
           si deg(t)=0
              rajoute(t, nouveau)
    sources←nouveau

acyclique ←nb=n
renvoyer(acyclique, niveaux)
```

FIGURE 8.9 – décomposition par niveaux

Dans l'algorithme de la figure 8.9, la variable `nb` sert à calculer le nombre de sommets affectés à un niveau et donc à détecter la présence éventuelle de circuits dans le graphe. Cet algorithme renvoie une variable booléenne `acyclique` indiquant si le graphe est acyclique ou pas et l'ensemble `niveaux` des niveaux résultant de la décomposition.

Exercice 8.4 *Programmer l'algorithme en Java, pour les deux représentations possibles des graphes.*

8.6 Recherche de plus courts chemins dans un graphe valué

On considère ici un graphe valué orienté ou pas, dont l'ensemble des sommets est

$$S = \{0, \ldots, (n-1)\}$$

On suppose de plus que toutes les valuations w_{st} des arcs (ou arêtes) entre deux sommets s et t sont *strictement positives*. Etant donné un sommet particulier, appelé ici *source*, on se propose, pour tout sommet s de S, de trouver, s'il existe, un plus court chemin de la source à s.

8.6.1 L'algorithme de Dijkstra

L'algorithme de Dijkstra produit deux tableaux *dist* et *pred*, indexés sur l'ensemble des sommets. Pour tout sommet s, *dist[s]* et *pred[s]* contiennent respectivement la longueur d'un plus court chemin de la source à s (distance) et le prédécesseur de s sur ce chemin. Le tableau *pred* permet alors de retrouver ce plus court chemin en remontant de prédécesseur en prédécesseur à partir de s. Par convention, si un tel chemin n'existe pas, *dist[s]*=$+\infty$ et *pred[s]*=*-1*.

L'algorithme est décrit sur la figure 8.10. Initialement, le fait qu'aucun plus court chemin n'ait été calculé se traduit par une initialisation de tous les prédécesseurs à -1 et de toutes les distances à

$+\infty$, sauf celle de la source, qui prend la valeur nulle.
L'algorithme examine alors itérativement chaque sommet. La variable Q désigne l'ensemble des sommets qui restent à examiner. Initialement Q est l'ensemble de tous les sommets, et l'algorithme se termine quand $Q = \emptyset$. Chaque itération consiste à prélever dans Q un sommet dont la distance est minimale (c'est l'objet de l'algorithme EXTRAIREMIN) et à mettre à jour les attributs (distance et prédécesseur) de tous ses sommets adjacents.

```
Dijkstra

pour tout sommet i        //Initialisation
    dist(i) ← +∞
    pred(i) ← -1
dist(source) ← 0
Q ← {0, ..., n − 1}

tant que Q≠ ∅
    s ← EXTRAIREMIN(Q)
    pour tout sommet t adjacent à s    //Mise à jour des attributs de t
       d ← dist(s)+ W_st
       si (dist(t)> d)
          dist(t) ← d
          pred(t) ← s

renvoyer(dist, pred)
```

FIGURE 8.10 – L'algorithme de Dijkstra

8.6.2 Correction de l'algorithme

Notons V l'ensemble des sommets déjà vus, autrement dit tous ceux qui ne sont pas dans Q. La correction peut être établie en exhibant deux invariants de la boucle qui constitue le coeur de l'algorithme.

Lemme 8.5 (Invariant I1) *A chaque itération, la variable* Q *satisfait la propriété suivante :*

$$\forall t \in V \quad \forall t' \in Q \quad dist(t) \leq dist(t')$$

Preuve Avant la première itération, V est vide, donc la propriété est évidemment satisfaite.
Supposons que la propriété est satisfaite avant une itération et soit s le nouveau sommet prélevé dans Q. On sait donc que

$$\forall t \in V \quad dist(t) \leq dist(s)$$

De plus, s est choisi de telle sorte que

$$\forall t' \in Q - \{s\} \quad dist(s) \leq dist(t')$$

On en déduit que

$$\forall t \in V \quad \forall t' \in Q - \{s\} \quad dist(t) \leq dist(s) \leq min(dist(t'), dist(s) + w_{st'})$$

Par suite, après mise à jour des valeurs $dist(t')$ pour les sommets t' de $Q - \{s\}$, on obtient

$$\forall t \in V \cup \{s\} \quad \forall t' \in Q - \{s\} \quad dist(t) \leq dist(t')$$

□

Lemme 8.6 (Invariant I2) *A chaque itération, pour tout sommet* s, dist(s) *est la longueur d'un plus court chemin de la source à* s, *ne contenant, mis à part* s, *que des sommets de l'ensemble* V *et* pred(s) *contient le prédécesseur de s sur ce chemin. Par convention, cette longueur vaut* $+\infty$ *et le prédécesseur vaut -1 si un tel chemin n'existe pas.*

Preuve Notons, pour tout sommet s, $sp_V(s)$ un plus court chemin de la source à s ne passant (mis à part s) que par des sommets de V, et $lg(sp_V(s))$ sa longueur. Il faut montrer que la propriété : $\forall s, dist(s) = lg(sp_V(s))$ et pred(s) est le prédécesseur de s sur $sp_V(s)$ est un invariant de l'algorithme.
Avant la première itération, $V = \emptyset$ et l'initialisation de *dist* et de *pred* assure alors que la proposition est satisfaite.
Supposons qu'elle soit satisfaite avant une itération et soit s l'élément que l'on ôte de Q et dont la distance *dist(s)* est minimale parmi toutes celles des sommets de Q.
Soit t un sommet quelconque. Trois cas se présentent.

1er cas : t=s Dans ce cas, $dist(s)$ et $pred(s)$ sont inchangés et de façon évidente, $sp_{V\cup\{s\}}(s) = sp_V(s)$.

2ème cas : t ∈Q-{s} Dans ce cas, si un plus court chemin jusqu'à t passant par des sommets de $V \cup \{s\}$ contient s, t est nécessairement adjacent à s. En effet, si cela n'était, t serait adjacent à un sommet u de V et s serait situé entre la source et u sur ce chemin, ce qui est impossible puisque $dist(u) \leq dist(s)$ d'après le lemme 8.5. Le sommet t ne peut donc être qu'adjacent à s et le chemin $sp_{V\cup\{s\}}(t)$ a alors pour longueur $dist(s) + w_{st}$. Si c'est un plus court chemin, cette longueur est nécessairement à inférieure ou égale à la valeur courante de $dist(t)$ (qui est la longueur d'un plus court chemin par des sommets de V). La mise à jour de cette dernière assure alors que $dist(t) = lg(sp_{V\cup\{s\}}(s))$. De plus s est le prédécesseur de t dans ce plus court chemin et devient la nouvelle valeur de $pred(t)$ après mise à jour.

3ème cas : t∈V Dans ce cas, même si t est adjacent à s, $dist(t)$ et $pred(t)$ restent inchangés du fait que $dist(t) \leq dist(s)$ d'après le lemme 8.5. Par hypothèse, $dist(s) = lg(sp_V(s))$ et $dist(t) = lg(sp_V(t))$. On en déduit que

$$lg(sp_V(t)) \leq lg(sp_V(s)) \qquad (1)$$

Un plus court chemin de la source à t ne contenant que des sommets de $V \cup \{s\}$, s'il passait pas s, aurait une longueur nécessairement supérieure à $lg(sp_V(s))$, ce qui est impossible d'après l'inégalité (1). On en déduit que le plus court chemin $sp_{V\cup\{s\}}(t)$ ne passe pas par s et donc $sp_{V\cup\{s\}}(t) = sp_V(t)$. L'invariant est donc encore satisfait.
□

Remarque Cette preuve montre en particulier qu'à partir du moment où un sommet s est ôté de Q, $dist(s)$ et $pred(s)$ ne sont plus modifiés. Par suite la ligne

```
pour tout sommet t adjacent à s
```

peut être remplacée par :

```
pour tout sommet t de Q adjacent à s
```

Proposition 8.7 *L'algorithme de Dijkstra est correct.*

Ceci découle immédiatement du lemme 8.6 et du fait qu'à la sortie de la boucle, l'ensemble Q est vide.

8.6.3 Complexité

La compexité de l'algorithme dépend en particulier du coût, à chaque itération, de l'extraction du sommet s de Q dont la distance est minimale. Pour optimiser le coût de l'extraction d'un élément minimal d'un ensemble, on peut imaginer organiser l'ensemble Q en tas (cf. section 7.5), l'ordre sur les sommets étant alors donné par le tableau *dist*. L'extraction de s est dans ce cas en $\Theta(lg(k))$ (cf. section 7.5), où k est la taille courante du tas. Toutefois, après mise à jour d'un sommet t de Q adjacent à s, l'ordre sur les éléments de Q est modifié du fait de la modification de $dist(t)$. Il convient alors de déplacer t dans le tas, en remarquant que puisque $dist(t)$ n'a pu que diminuer, t doit être tamisé vers le haut du tas. Cette opération est également en $\Theta(lg(k))$.

```
Dijkstra

pour tout sommet i        //Initialisation
    dist(i) ← +∞
    pred(i) ← -1
dist(source) ← 0
Q ← NouveauTas(n, dist);

tant que non vide(Q)
    s ← extraireMin(Q)
    pour tout sommet t (de Q) adjacent à s    //Mise à jour des attributs de t
       d ← dist(s)+ W_st
       si (dist(t)>d)
          dist(t) ← d
          pred(t) ← s
          tamiserVersLeHaut(t, Q)

renvoyer(dist, pred)
```

FIGURE 8.11 – Algorithme de Dijkstra utilisant une structure de tas

La figure 8.11 décrit une version raffinée de l'algorihme de Dijkstra, utilisant une organisation en tas de l'ensemble Q.
La primitive NouveauTas(n, dist) construit un tas dont les éléments sont les n sommets du graphe, l'ordre étant celui défini par le tableau *dist*. La solution pour transformer une liste en un tas, donnée dans la section 7.5 est linéaire en la taille de la liste.
La primitive extraireMin(t,Q) est logarithmique en la taille du tas. L'exécution des n extractions sera donc en $\sum_{k=1}^{n} lg(k) \in \Theta(nlg(n))$.
La primitive tamiserVersLeHaut(t,Q), déplace l'élément t vers le haut dans le tas Q en prenant en compte l'éventuelle modification de la valeur de $dist(t)$ de façon à préserver la structure de tas et est logarithmique en la taille du tas. Elle est également en $lg(k)$ où k est la taille du tas. Si p est le nombre d'arcs ou d'arêtes, il y a au plus p exécutions de cette primitive, dont le coût est toujours majoré par $lg(n)$.
On en déduit que la complexité de l'algorithme est en $O((p+n)lg(n))$.

8.6.4 Une mise en œuvre de l'algorithme en Java

Mise en œuvre du tas

Comme décrit dans la section 7.5, le tas est représenté par un objet constitué d'un tableau T contenant les éléments du tas (ici des sommets) virtuellement organisés en arbre binaire et d'un entier indiquant le nombre d'éléments du tas.
La relation d'ordre sur les éléments du tas est ici variable puisqu'elle sera induite par les distances. Elle est donc représentée par un tableau *dist* d'entiers.
Le constructeur affecte à la variable *dist* l'adresse d'un tableau passé en paramètre, crée le tableau T dans lequel sont mémorisés les sommets à partir du rang 1, de façon que la case de rang 0 puisse être utilisée comme sentinelle. et le transforme immédiatement en tas.
Les méthodes *tamiserVersLeBas*, *mettreEntas*, *extraireMin tamiserVersLeHaut*, sont identiques à celles de la section 7.5, si ce n'est que l'ordre est maintenant induit par les valeurs du tableau *dist*.

```
public class tas{
   private int taille;
   private int [] T;
   private int[] dist;

   tas(int n, int[] d){
      taille = n;
      T = new int[n+1];
      for(int i=0;i<n; i++)
      T[i+1]=i;
      dist=d;
      mettreEnTas();
   }

   boolean vide(){ return(taille==0);}

   int element(int i){return(T[i]);}

   int getTaille(){return  taille;}

   void tamiserVersLeBas(int i){
      int x=T[i],f=2*i,dx=dist[x];

      if(f<taille && T[f]>T[f+1]) f++;
      while (f<=taille && dx>dist[T[f]]){          // Invariant: T[i] = x
         T[i] = T[f]; T[f] = x;                   // Echange
         i = f;
         f=2*i;                                  // Choix du nouveau fils
         if(f<taille && dist[T[f]]>dist[T[f+1]]) f++;
      }
   }

   int extraireMin(){
      int r = T[1];
      T[1]=T[taille--];
      tamiserVersLeBas(1);
      return r;
   }
```

```
   void mettreEnTas(){
      for(int i=taille/2; i>0; i--)
         tamiserVersLeBas(i);
   }

   void tamiserVersLeHaut(int i){
      int  x= T[i], dx=dist[x];

      T[0]=x;
      for(int p=i/2; dx<dist[T[p]]; p/=2){      // Invariant: T[i] = x
         T[i] = T[p]; T[p] = x; i = p;
      }
   }
}
```

Programmation de l'algorithme

Le graphe est ici représenté avec des listes d'adjacence, décrites dans une classe *List*, locale à la classe *Graph*. Elles sont composées de maillons, chacun portant non seulement un sommet mais aussi un poids. Les ajouts se font en tête.

```
public class Graph{

    class List{
       int sommet;
       int poids;
       List suiv;

       List(){
          suiv = null;
       }

       List(int s, int p, List suivant){
          sommet = s;
          poids = p;
          suiv = suivant;
       }

       void ajoute(int s, int p){
          suiv = new List(s, p, suiv);
       }
    }
```

Un graphe est donc un objet constitué d'un tableau *succ* de listes d'adjacence et d'un entier représentant le nombre de sommets. Le constructeur prend en paramètre le nombre n de sommets, crée le tableau *succ* de taille n et initialise chacun de ses éléments à la liste vide. Les listes d'adjacence seront effectivement construites par appels successifs à la méthode *ajoute_arc* qui prend en paramètre les extrémités de l'arc et son poids.
Deux champs supplémentaires *dist* et *pred* sont deux tableaux dans lesquels sont mémorisées les distances et les prédécesseurs.

```
    List [] succ;
    int ns;
    int[]dist, pred;
```

```
Graph(int n){
    ns = n;
    succ =  new List[ns];
    for(int i=0;i<ns;i++)
        succ[i]=new List();
}

void ajoute_arc (int i, int j, int p){
    succ[i].ajoute(j,p);
}
```

L'algorithme de Dijkstra est programmé ci-après. Après extraction du minimum du s du tas Q, la mise à jour se fait sur tous les sommets adjacents à s (et non seulement sur ceux du tas) pour des raisons d'efficacité. Enfin, si le graphe n'est pas connexe, l'un des sommets s extraits de Q sera tel que $dits[s] = Integer.MAX\_VALUE$ et il en sera ainsi de tous les sommets restants dans Q. On sort alors de la boucle *while* par une instruction `break`, faute de quoi l'instruction `aux = dist[s]+ l.poids` provoquerait un débordement de capacité.

```
void Dijkstra(int source){
   int s, t, d;

   dist= new int[ns];
   pred = new int[ns];
   for(int i=0; i<ns; i++){
      dist[i] = Integer.MAX_VALUE;
      pred[i]=-1;
   }
   dist[source]=0;
   tas Q = new tas(ns, dist);

   while (Q.getTaille()!=0){
      s = Q.extraireMin();
      if(dist[s]<Integer.MAX_VALUE)
         for(List l=succ[s].suiv; l!=null;l=l.suiv){
            t=l.sommet;
            d = dist[s]+ l.poids;
            if (dist[t]>d){  // t est necessairement dans Q
               dist[t] = d;
               pred[t] = s;
               Q.tamiserVersLeHaut(t);
            }
         }
       else break;
   }
}
```

Affichage des résultats

Il est maintenant possible, à l'aide d'un couple de tableaux résultat de l'algorithme de Dijkstra, d'afficher pour tout sommet s du graphe un plus court chemin de la source à ce sommet, s'il existe, ainsi que sa longueur.

```
void affichePlusCourtsChemins(int source){
   Dijkstra(source);
   for(int s = 0; s<ns; s++){
```

```
        AfficheChemin(s);
        System.out.println();
    }
  }
```

La méthode *afficheChemin(s* affiche, à partir du résultat c de la méthode $Dijkstra$, la distance de la source au sommet s, suivie du plus court chemin, s'il existe. Pour afficher celui-ci, il suffit de le parcourir en sens inverse, chaque sommet à partir de s étant obtenu à partir du précédent à l'aide du tableau des prédécesseurs. La source est le sommet dont le prédécesseur vaut -1. Puisque le chemin est parcouru en ordre inverse de celui souhaité pour l'affichage, ce parcours est un parcours récursif en post-ordre. On obtient ainsi :

```
void AfficheChemin(int s){
   int d = dist[s];
   if(d==Integer.MAX_VALUE)
      System.out.print(s+ ":\tPas de chemin vers ce sommet");
   else{
      System.out.print(s + ":\tDistance = " + d + "\tChemin : ");
      AffChem(s);
   }
 }

private void AffChem (int s){
   if (s != -1){
      AffChem(pred[s]);
      System.out.print( " " + s  +" ");
   }
 }
```

Chapitre 9

Algorithmique du tri

La donnée du problème est ici un tableau A dont les éléments appartiennent à un ensemble totalement ordonné. On se propose de trier par ordre croissant la partie $A[d\,..\,f]$ des éléments dont les indices sont compris entre d et f. Par convention $A[d\,..\,f] = \emptyset$ quand $f < d$.
Le coût des algorithmes de tris présentés dans ce chapitre est le nombre d'opérations significatives qu'ils effectuent en fonction de la taille $n = (f - d + 1)$ de la donnée. Les opérations prises en compte sont ici les déplacements des éléments du tableau et leur comparaisons : ces éléments étant virtuellement complexes, ce sont les opérations les plus coûteuses.

9.1 Tris élémentaires

Les algorithmes de tri par sélection et de tri par insertion, sont deux algorithmes itératifs élémentaires.

9.1.1 Le tri par sélection

Le principe

L'algorithme consiste à parcourir le tableau, à l'aide d'un indice j, incrémenté à chaque itération et de valeur initiale d. L'invariant de cet algorithme est le suivant : après chaque itération, $A[d\,..\,j]$ est déjà triée et contient les $j - d + 1$ plus petits éléments de $A[d\,..\,f]$. Pour que cette propriété soit préservée par itération, il suffit, après incrémentation de j, d'échanger $A[j]$ et le minimum de la partie $A[j\,..\,f]$. Cet invariant assure qu'après l'itération pour laquelle $j = f - 1$, le tableau est trié par ordre croissant.

```
TriParSelection(A, d, f)

pour j=d à (f-1) faire
    min=Minimum(A,j,f)
    si min≠j
       echanger(A[min], A[j])
```

```
Minimum(A, d, f)
indmin ← d
min ← A[d]
pour i = d+1 à f faire
    si A[i]<min
       min ← A[i]
       indmin ← i
renvoyer(i)
```

FIGURE 9.1 – Tri par sélection

Complexité

Le coût de la recherche du minimum dans un tableau de taille n requiert $(n-1)$ comparaisons. Le tri par sélection effectue donc $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparaisons et au plus $(n-1)$ échanges. Il est donc en $\Theta(n^2)$, quelle que soit la configuration des données.

9.1.2 Le tri par insertion

Le principe

L'algorithme consiste à parcourir le tableau, à l'aide d'un indice j, incrémenté à chaque itération et de valeur initiale $d+1$. L'invariant de cet algorithme est le suivant : après chaque itération, $A[d\,..\,j]$ est rangé par ordre croissant. Pour que cette propriété soit préservée par itération, il suffit, après incrémentation de j, de placer la valeur x de $A[j]$ dans $A[1\,..\,(j-1)]$ de façon à préserver l'ordre des éléments. Pour cela, tous les éléments de $A[1\,..\,(j-1)]$ à partir de celui d'indice $(j-1)$ et strictement plus grands que x sont décalés vers la droite, puis x est recopié à la place du dernier élément déplacé. Cet invariant assure qu'après l'itération pour laquelle $j = f$, le tableau est trié par ordre croissant.

```
TriParInsertion(A, d, f)

pour j=d+1 à f faire
    x ← A[j]
    i ← j-1
    tant que (i≥d) et (A[i]>x) faire
       A[i+1]←A[i]
       i← i-1
   A[i+1] ← x
```

FIGURE 9.2 – Tri par insertion

Complexité

Chaque itération de la boucle la plus externe fait le même nombre (à 1 près) de comparaisons et de déplacements des éléments du tableau. Ce nombre vaut 1 dans le meilleur des cas, $(j-d)$ dans le pire des cas, et $(j-d)/2$ en moyenne.
Le meilleur des cas est celui pour lequel le tableau est déjà trié. L'algorithme effectue alors $(n-1)$ comparaisons et aucun déplacement. Il est donc dans ce cas linéaire.
Dans le pire des cas et en moyenne, le coût est proportionnel à $\sum_{i=1}^{n-1} i = \Theta(n^2)$. L'algorithme est donc quadratique en moyenne et dans le pire des cas.

9.2 Tri rapide

Le principe

Le tri rapide consiste, dans un premier temps, à partitionner le tableau en deux sous-tableaux de taille strictement inférieures, dans lesquels ont été placées les valeurs respectivement inférieures ou égales et supérieures ou égales à un certain élément, appelé *pivot*, choisi arbitrairement dans le tableau initial. Tout élément du premier sous-tableau étant alors inférieur ou égal à tout élément

du second, il suffit d'appeler ensuite récursivement le tri sur chaque sous-tableau. La terminaison est assurée par le fait que ces deux sous-tableaux sont de taille strictement inférieure à celle du tableau initial. Au bout d'un nombre fini d'appels récursifs, on obtiendra des tableaux de taille 1 qui constituent les cas terminaux de la récursion.

Nous présentons ici l'une des multiples variantes du tri rapide, qui dépendent essentiellement de l'algorithme de partition choisi (et en particulier du pivot).

On suppose donc disposer d'un algorithme de partition, satisfaisant la spécification suivante :

`Partition(A, d, f)` *(On suppose que $d < f$)*

- choisit pour pivot $A[d]$
- classe les éléments de $A[d\,..\,f]$ par rapport au pivot
- renvoie un indice j tel que $d \leq j < f$ et tel que :
  - tous les éléments de $A[d\,..\,j]$ sont inférieurs ou égaux au pivot
  - tous ceux de $A[(j+1)\,..\,f]$ sont supérieurs ou égaux au pivot.

Le fait que $d \leq j < f$ assure que les deux sous-tableaux sont de taille strictement inférieure à celle du tableau initial. Il suffit alors, pour trier $A[d\,..\,f]$, de rappeler récursivement le tri sur chacun des sous-tableaux $A[d\,..\,j]$ et $A[(j+1)\,..\,f]$.

```
TriRapide(A, d, f)

si d<f
   j←Partition(A, d, f)
   TriRapide(A, d, j)
   TriRapide(A, j+1, f)
```

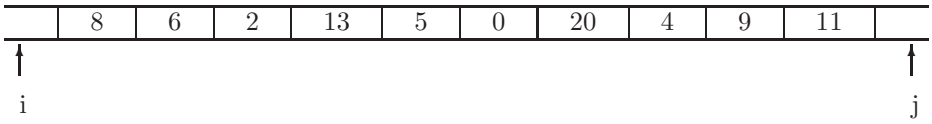
FIGURE 9.3 – Tri rapide

Cet algorithme est correct, sous réserve que l'algorithme de partition donné figure 9.4 le soit.

```
Partition(A, d, f)

pivot ← A[d]
i ← d-1
j ← f+1
(*)répéter
     (a) répéter j ← j-1 tant que A[j]> pivot
     (b) répéter i ← i+1 tant que A[i]< pivot
         si i<j
            echanger(A[i], A[j])
tant que (i<j)
renvoyer(j)
```

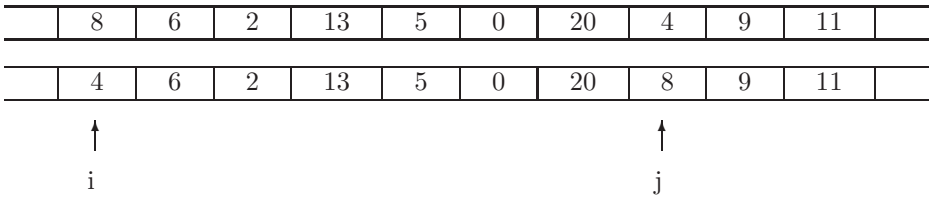
FIGURE 9.4 – L'algorithme de partition

Considérons par exemple la configuration initiale

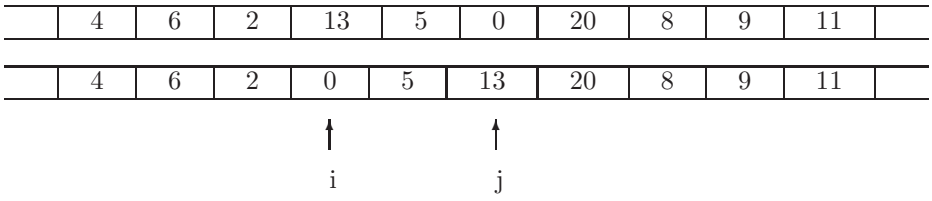


Le pivot choisi est prend ici la valeur 8. Voici les configurations obtenues à chaque itération de l'algorithme de partition.

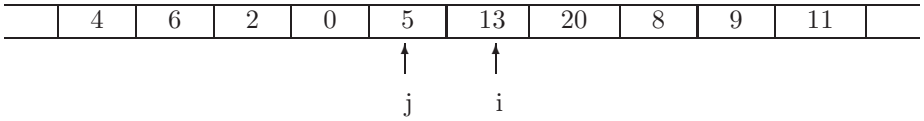
Première itération



Seconde itération



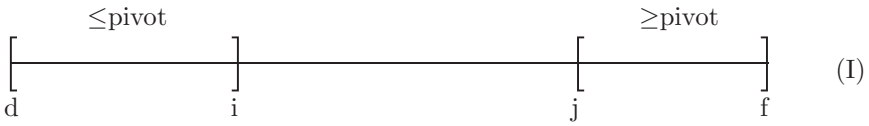
Dernière itération



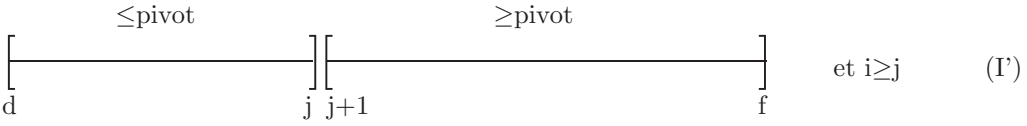
Correction de la partition

On suppose que $d < f$. On remarque qu'il se produit au moins une itération de la boucle (*).

On montre qu'à chaque itération, ni (a) ni (b) ne bouclent, que les indices des éléments du tableau examinés sont toujours compris entre d et f et qu'après chaque itération, on est dans l'une des situations suivantes :



(I)



Chacun des intervalles délimités par les crochets est vide au départ. Si i et j sont les valeurs des indices avant une itération, on note i' et j' leur nouvelle valeur après l'itération. La boucle (a) s'arrête dès que l'on trouve un élément d'indice j' inférieur ou égal au pivot. On sort donc toujours de cette boucle avec une valeur j' telle que $d \leq j' < j$ pour la première itération ou en $i \leq j' < j$ pour les autres. De même on sort de la boucle (b) avec une nouvelle valeur i' telle que $i < i' \leq (j' + 1)$, puisque $A[j' + 1] \geq pivot$. On sait donc que :

(i) tous les éléments d'indices $> j'$ sont supérieurs ou égaux au pivot
(ii) tous les éléments d'indices $< i'$ sont inférieurs ou égaux au pivot
(iii) $A[i'] \geq pivot$
(iv) $A[j'] \leq pivot$
(v) $i' \leq (j' + 1)$

Deux cas sont possibles :

1. Soit $i' < j'$: l'échange de $A[i']$ et $A[j']$ et les conditions (i) et (ii) assurent que l'invariant (I) est satisfait par les nouvelles valeurs i' et j'.
2. Soit $i' \geq j'$. D'après (v) on en déduit que seuls deux cas sont possibles. Soit $i' = j'$ et, d'après (iii) et (iv), $A[i'] = A[j'] = pivot$. Soit $i' = j' + 1$. Dans les deux cas, (i) et (ii) assurent que la condition (I') est satisfaite.

Seule la situation (I') satisfait la condition d'arrêt et dans ce cas-là, la valeur de j renvoyée satisfait bien la spécification.

Coût de la partition

On rappelle que n désigne la taille de $A[d\,..\,f]$. On a vu qu'à la sortie de la boucle (*), soit $i = j$ soit $i = j+1$. Le nombre total d'incrémentations de i et de décrémentations de j est donc $n+1$ ou $n+2$. Pour chacune de ces opérations on fait une comparaison entre le pivot et un élément du tableau, soit $n+2$ comparaisons dans le pire des cas. Le nombre d'échanges est le nombre d'itérations de la boucle la plus externe, soit dans le pire des cas $\lfloor \frac{n+2}{2} \rfloor$. Le nombre d'échanges et de comparaisons requis par l'algorithme de partition d'un tableau de taille n est donc, dans le pire des cas :

$$f(n) = \frac{3}{2}(n+2)$$

Coût du tri rapide dans le pire des cas

Intuitivement, il semble que le pire des cas est celui où chaque sous-tableau de taille k est partitionné en un sous-tableau de taille 1 et un sous-tableau de taille $k-1$. Si l'on note $C(n)$ le coût du tri rapide d'un tableau de taille n dans ce type de configuration et sachant que $C(1) = 0$, on obtient les relations de récurrence suivantes :
C(n)= C(n-1) + f(n)
C(n-1) = C(n-2) + f(n-1)
...
C(3) = C(2) + f(3)
C(2) = 0 + f(2)

En ajoutant membre à membre ces égalités, on obtient :

$$\text{C(n)}= \sum_{i=2}^{n} f(i) \;=\; \frac{3}{2}\left(\sum_{i=2}^{n} i + \sum_{i=2}^{n} 2\right) \;=\; \frac{3}{2}\left(\frac{n(n+1)}{2} - 1 + 2(n-1)\right) \;=\; \frac{3}{4}(n^2+5n-6).$$

Pour montrer qu'il s'agit effectivement du pire des cas, on va prouver par récurrence que, quelle que soit la façon dont sont partitionnés les sous-tableaux :

$$\forall n \geq 1 \qquad C(n) \leq \frac{3}{4}(n^2+5n-6)$$

Pour $n = 1 : C(1) = 0 = \frac{3}{4}(1^2+5-6)$.
Soit $n \geq 2$, quelconque, fixé. Supposons (hypothèse de récurrence) que :

$$\forall p, \quad 0 < p < n, \qquad C(p) \leq \frac{3}{4}(p^2+5p-6)$$

Dans l'hypothèse où l'intervalle est partitionné en deux tableaux de tailles respectives p et $n-p$, $C(n) = C(p) + C(n-p) + f(n)$. On en déduit que :

$$C(n) \leq \max_{0<p<n}(C(p)+C(n-p)) + f(n) \;\leq\; \frac{3}{4}\max_{0<p<n} g(p) + f(n)$$

avec $g(p) = (p^2+5p-6)+((n-p)^2+5(n-p)-6)$. Puisque $g'(p) = 4p-2n$, on obtient le tableau de variation ci-dessous, dans lequel :

$max = g(1) = g(n-1) = n^2+3n-10$
$min = g(\frac{n}{2}) = 2\left((\frac{n}{2})^2+5\frac{n}{2}-6\right) = \frac{1}{2}(n^2+10n-24) = \frac{1}{2}(n+12)(n-2)$

n	1		$\frac{n}{2}$		$n-1$
g'(p)		$-$	0	$+$	
g(p)	max	↘	min	↗	max

On en déduit que $C(n) \leq \frac{3}{4}(n^2+3n-10) + \frac{3}{2}(n+2) = \frac{3}{4}(n^2+5n-6)$. □

Théorème 9.1 *Le tri rapide est en $\Theta(n^2)$ dans le pire des cas.*

Analyse en moyenne du tri rapide

Il semblerait qu'un cas très favorable soit celui où chaque partition coupe les tableaux en deux sous-tableaux de tailles égales (à une unité près si les éléments sont en nombre impair). La fonction de coût satisfait alors l'équation :

$$C(n) = 2C(\frac{n}{2}) + f(n) \tag{9.1}$$

En supposant que $n = 2^k$ et donc que $k = lg_2(n)$, on obtient le calcul suivant :

$$\begin{array}{lll} & \text{C(n)} & = 2\ \text{C(n/2)+f(n)} \\ 2 \times | & \text{C(n/2)} & = 2\ \text{C(n/}2^2\text{)+f(n/2)} \\ 2^2 \times | & \text{C(n/}2^2\text{)} & = 2\ \text{C(n/}2^3\text{)+f(n/}2^2\text{)} \\ & & \cdots \end{array}$$

$2^{k-1}\times$ | C(n/2^{k-1}) = 2C(1)+ f(n/2^{k-1})

$$\text{C(n)} = \sum_{i=0}^{k-1} 2^i\, f(n/2^i) = 3\sum_{i=0}^{k-1}(2^i + \frac{n}{2}) = 3(\frac{k}{2}n + 2^k - 1) = \frac{3}{2}\,(nlg_2(n) + n - 1)$$

et donc $C(n) \in \Theta(nlg(n))$

On obtient un résultat analogue dans le cas général en encadrant n par deux puissances consécutives de 2. C'est la meilleure des situations observées jusqu'ici. Se pose alors la question de la complexité en moyenne : peut-elle également être meilleure que quadratique ? Peut-on prouver qu'elle est également en $nlg(n)$?

Pour conduire une analyse en moyenne du tri rapide, on suppose que les éléments du tableau sont tous distincts et on les numérote par ordre croissant. On obtient ainsi les n éléments :

$$a_1 < a_2 < \ldots < a_n$$

On suppose que la probabilité pour que chacun d'eux soit le pivot de la partition est égale à $1/n$. Si le pivot est a_1, la partition de l'ensemble des éléments du tableau ne peut être que $\{a_1\}$ et $\{a_2, \ldots, a_n\}$.
Si le pivot est dans $\{a_2, \ldots, a_n\}$, après la première itération de la boucle (*) de la partition, on a $i = d < j$. On échange alors $A[d]$ et $A[j]$ et on réitère, ce qui a pour effet de faire décroître j strictement. Par suite l'indice du pivot sera strictement supérieur à la valeur finale de j, donc le pivot se trouve dans la partie droite de la partition. On en déduit que le tableau suivant :

pivot	partition	$C(n)$
a_1	$\{a_1\}$ $\{a_2, \ldots a_n\}$	$C(1) + C(n-1) + f(n)$
a_2	$\{a_1\}$ $\{a_2, \ldots a_n\}$	$C(1) + C(n-1) + f(n)$
a_3	$\{a_1, a_2\}$ $\{a_3, \ldots a_n\}$	$C(2) + C(n-2) + f(n)$
$\ldots$	$\ldots$	$\ldots$
a_{k+1}	$\{a_1, \ldots, a_k\}$ $\{a_{k+1}, \ldots a_n\}$	$C(k) + C(n-k) + f(n)$
$\ldots$	$\ldots$	$\ldots$
a_n	$\{a_1, \ldots, a_{n-1}\}$ $\{a_n\}$	$C(n-1) + C(1) + f(n)$

On en déduit le coût en moyenne :

$$C(n) = \frac{1}{n}(C(1) + C(n-1) + \sum_{k=1}^{n-1}(C(k) + C(n-k)) + f(n)$$

D'après l'analyse dans le pire des cas,

$$C(n-1) \leq \frac{3}{4}((n-1)^2 + 5(n-1) - 6) = \frac{3}{4}(n^2 + 3n - 10)$$

. De plus, $C(1) = 0$. Par suite

$$\frac{1}{n}(C(1) + C(n-1)) + f(n) \leq \frac{3}{4n}(n^2 + 3n - 10) + \frac{3}{2}(n+2) = \frac{3}{4}(3n + 7 - \frac{10}{n})$$

En remarquant enfin que $\sum_{k=1}^{n-1} C(k) = \sum_{k=1}^{n-1} C(n-k)$, on en déduit que :

$$C(n) \leq \frac{2}{n}\sum_{k=1}^{n-1} C(k) + \frac{3}{4}(3n + 7 - \frac{10}{n}) \tag{9.2}$$

On se propose de montrer qu'il existe une constante a strictement positive telle que :

$$\forall n \geq 1,\ C(n) \leq a\, nln(n) \tag{9.3}$$

On procède par récurrence sur n.
Si $n = 1$, $C(n) = 0$. La propriété est satisfaite dès que l'on choisit $a \geq 0$.
Soit $n \geq 2$ et supposons que $\forall k < n,\ C(k) \leq a\, nln(k)$. On déduit de la relation (9.2) que :

$$C(n) \leq \frac{2}{n}\sum_{k=1}^{n-1} a\, kln(k) + \frac{3}{4}(n + 7 - \frac{10}{n}) = \frac{2}{n}(a\sum_{k=1}^{n-1} kln(k)) + \frac{3}{4}(3n + 7 - \frac{10}{n}) \tag{9.4}$$

Comme la fonction $x \to xln(x)$ est croissante :

$$\sum_{k=1}^{n-1} kln(k) = \sum_{k=2}^{n-1} kln(k) \leq \int_2^n xln(x)dx = \left[\frac{x^2}{2}ln(x)\right]_2^n - \int_2^n \frac{x}{2}dx = \left[\frac{x^2}{2}ln(x) - \frac{x^2}{4}\right]_2^n =$$

$$\frac{n^2}{2}(ln(n) - \frac{1}{2}) + (1 - ln(4)) \leq \frac{n^2}{2}(ln(n) - \frac{1}{2}).$$

Par suite, d'après la relation (9.4), on obtient :

$$C(n) \leq an(ln(n) - \frac{1}{2}) + \frac{3}{4}(3n + 7 - \frac{10}{n}) \leq a\, nln(n) - \frac{a}{2}n + \frac{3}{4}(3n + 7)$$

Et donc la condition

$$C(n) \leq a\, nln(n)$$

est satisfaite dès que

$$\frac{3}{4}(3n + 7) - \frac{a}{2}n \leq 0$$

c'est-à-dire dès que

$$21 \leq n(2a - 9)$$

et ce, pour tout $n \geq 1$. Il suffit pour cela que $21 \leq 2a - 9$, et donc de choisir par exemple $a = 15$.

Théorème 9.2 *Le tri rapide est en $\Theta(nlg(n))$ en moyenne.*

9.3 Tri fusion

Comme indiqué sur la figure 9.5, le tri fusion consiste à trier récursivement les deux moitiés du tableau, puis à *fusionner* les deux sous-tableaux ainsi ordonnés. La *fusion* de deux tableaux rangés par ordre croissant consiste à les parcourir simultanément de gauche à droite et à recopier dans un troisième tableau le minimum des éléments courants de chacun d'eux.

Soit $f(n)$ le coût de la fusion de deux tableaux totalisant n éléments. La fonction C de coût du tri fusion d'un tableau de taille n satisfait l'équation :

$$C(n) = 2C(\frac{n}{2}) + f(n) \qquad f(n) \in \Theta(n)$$

```
TriFusion(A, d, f)

si d<f
   m ← ⌊(d + f)/2⌋
   TriFusion(A, d, m)
   TriFusion(A, (m+1), f)
   Fusion(A, d, m, f)
```

FIGURE 9.5 – Tri fusion

Cette équation est analogue à l'équation (9.1) étudiée dans l'analyse dans le pire des cas du tri rapide. On a montré que sa solution est en $\Theta(nlg(n))$.

Théorème 9.3 *Le tri fusion est en* $\Theta(nlg(n))$.

On montrera dans la section suivante que ce tri est optimal. En revanche, il ne s'agit pas d'un tri *en place* en ce sens que la fusion nécessite des recopies dans un tableau auxiliaire (cf. l'exercice 9.3). En ce sens, il est moins performant *en moyenne* que le tri rapide qui ne présente pas cet inconvénient. Ce dernier peut être toutefois quadratique dans le pire des cas.

Pour une mise en œuvre et une analyse détaillée du tri fusion, on pourra se reporter à l'exercice 9.3 de la section 9.7.

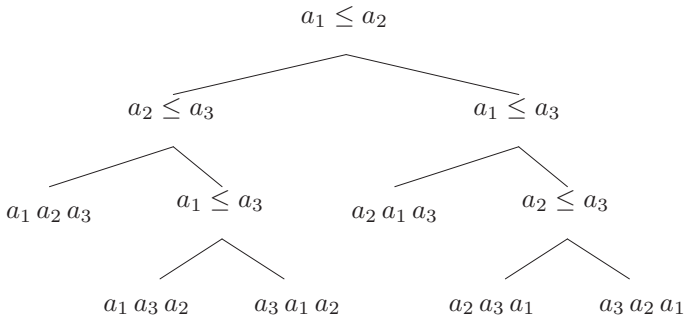
9.4 Tri par tas

Ce tri a été traité de façon exhaustive dans la section 7.5.

9.5 Complexité du problème du tri

Théorème 9.4 *Le problème du tri est en* $\Theta(nlg(n))$ *dans le pire des cas.*

Nous allons démontrer que, dans le pire des cas, il faut $lg_2(n!)$ comparaisons pour trier n éléments, et ce *quel que soit l'algorithme choisi*. Plaçons nous tout d'abord dans le cas d'un ensemble à 3 éléments. Soit a_1, a_2 et a_3 les éléments à trier. L'arbre de décision suivant exprime les comparaisons à effectuer pour ranger les éléments par ordre croissant. Une descente à gauche signifie que le résultat de la comparaison est vrai, une descente à droite qu'il est faux.



Dans le cas général d'un ensemble à n éléments, cet arbre est un arbre binaire à $n!$ feuilles (autant de feuilles que de permutations d'un ensemble de cardinal n). Le nombre de comparaisons à faire

pour trier les éléments est égal, dans le pire des cas à la hauteur h de l'arbre. Or un arbre binaire de hauteur h a au plus 2^h feuilles (ce nombre est atteint quand tous les niveaux de l'arbre sont remplis). On en déduit donc que $n! \leq 2^h$ et donc que $lg_2(n!) \leq h$.
Par suite, tout tri requiert au moins $lg_2(n!)$ comparaisons dans le pire des cas. Or $lg_2(n!) = \sum_{i=2}^{n} lg_2(i) \in \Theta(nlg(n))$

Du point de vue asymptotique, le tri fusion et le tri par tas sont donc optimaux et le tri rapide est optimal en moyenne.

9.6 Complexité des algorithmes *diviser pour régner*

Les algorithmes de type *diviser pour régner* (*divide and conquer*) sont ceux qui divisent le problème à résoudre en un certain nombre de problèmes identiques, dont les tailles sont strictement inférieures au problème initial et sur lesquels ils se rappellent récursivement. Le tri fusion et le tri rapide en sont deux exemples. Sur un tableau de taille n, le premier se rappelle récursivement sur deux problèmes de taille $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, puis combine les deux résultats par une fusion. Le second partitionne l'ensemble des éléments en deux tableaux de tailles k et $n-k$ (avec $0 < k < (n-1)$) puis se rappelle récursivement sur ces deux tableaux. Un autre exemple est celui des *tours de Hanoï* (cf. section 5.4.2) où l'on résoud un problème de taille n par deux rappels récursifs sur des problèmes de taille $n-1$.

En ce qui concerne l'évaluation de la complexité en temps de ce type d'algorithmes, un cas intéressant est celui où le problème initial de taille n, est divisé en a sous-problèmes identiques et *tous de même taille* n/b, avec $b > 1$. Si l'on note $C(n)$ le coût de l'algorithme sur une donnée de taille n et $f(n)$ le coût des opérations nécessaires à la scission du problème en sous-problèmes et de la combinaison des résultats des sous-problèmes pour obtenir le résultat final, on obtient la formule :

$$C(n) = aC(\frac{n}{b}) + f(n)$$

Si n n'est pas un multiple de b, $\frac{n}{b}$ doit être compris comme $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Supposons dans un premier temps que $n = b^l$ et donc $l = lg_b(n)$. On obtient le calcul suivant :

		C(n)	=	a C(n/b)+f(n)
a	× |	C(n/b)	=	a C(n/b^2)+f(n/b)
a^2	× |	C(n/b^2)	=	a C(n/b^3)+f(n/b^2)
			...	
a^{l-1}	× |	C(n/b^{l-1})	=	a C(1)+ f(n/b^{l-1})
a^l	× |	C(1)	=	f(1)

$$\text{C(n)} = \sum_{i=0}^{lg_b(n)} a^i f(n/b^i) = a^{lg_b(n)} f(1) + \sum_{i=1}^{lg_b(n)-1} a^i f(n/b^i) + f(n)$$

Sachant que $a^{lg_b(n)} = n^{lg_b(a)}$, on obtient :

$$C(n) = n^{lg_b(a)} f(1) + \sum_{i=1}^{lg_b(n)-1} a^i f(n/b^i) + f(n)$$

Le comportement asymptotique de $C(n)$ dépend donc du terme dominant de la somme ci-dessus. Le premier terme représente la contribution des $n^{lg_b(a)}$ cas terminaux (de taille 1). Le second celle des sous-problèmes de taille intermédiaire entre le problème initial et les cas terminaux. Le dernier représente le coût supplémentaire pour obtenir les a sous-problèmes et agencer leur résultats pour

obtenir le résultat final. On admettra que cette formule est valable même dans les cas où n n'est pas une puissance de b. Le théorème suivant fournit immédiatement le comportement asymptotique de la fonction C dans un grand nombre de cas.

Théorème 9.5 ("Master theorem") *Soit $C : \mathbb{N} \to \mathbb{R}^+$ une fonction définie par la relation de récurrence :*

$$C(n) = aC(\frac{n}{b}) + f(n)$$

où $a \geq 1$ et $b > 1$ sont deux constantes réelles et $f : \mathbb{N} \to \mathbb{R}^+$ une fonction. Trois cas sont envisageables, synthétisés dans le tableau ci-dessous :

Hypothèses	*Conclusions*	
$\exists \epsilon > 0,\ \mathbf{f(n)} = \mathbf{O(n^{\log_b(a)-\epsilon})}$	$\mathbf{C(n) = \Theta(n^{\log_b a})}$	*premier terme dominant*
$\exists \mathbf{k} \geq 0,\ \mathbf{f(n)} = \mathbf{\Theta(n^{\log_b a} \lg(n)^k)}$	$\mathbf{C(n) = \Theta(n^{\log_b a} \lg(n)^{(k+1)})}$	*tous les termes comptent*
$\exists \epsilon > 0,\ \mathbf{f(n)} = \mathbf{\Omega(n^{\log_b(a)+\epsilon})}$ $\exists \mathbf{c} < \mathbf{1},\ \mathbf{a.f(n/b)} \leq \mathbf{c.f(n)}$ *pour n suffisament grand*	$\mathbf{C(n) = \Theta(f(n))}$	*dernier terme dominant*

On a vu que le coût du tri fusion est donné par : $C(n) = 2C(n/2) + f(n)$, où $f(n) = \Theta(n)$. Ici $a = b = 2$, donc $lg_b(a) = 1$ et par suite $f(n) = \Theta(n^{lg_b(a)})$. C'est la deuxième ligne du tableau qui s'applique ici : on retrouve le fait que le tri fusion est en $\Theta(nlg(n))$. Il en est de même pour le pire des cas de l'algorithme du tri rapide.

9.7 Exercices

Exercice 9.1 *Trouver le comportement asymptotique des fonctions définies par les relations suivantes :*

1. $C(n) = 9C(n/3) + n$
2. $C(n) = C(2n/3) + 1$
3. $C(n) = 3C(n/4) + nlg(n)$
4. $C(n) = 2C(n/2) + nlg(n)$

Exercice 9.2 *Considérons deux matrices carrées* A *et* B *d'ordre* n. *L'algorithme ci-dessous calcule leur produit* C.

```
Pour i =1 à n faire
   Pour j =1 à n faire
      s ← 0
      Pour k =1 à n faire
         s ← s + A_ik * B_kj
      C_ij ← s
```

1- Quelle est la complexité de cet algorithme (la taille des données étant le format n *des matrices, les opérations significatives étant les additions et les multiplications de réèls qui sont supposées faites en temps constant) ?*

On suppose maintenant que $n = 2^k$, *où* k *est un entier positif. On peut décomposer la matrice carrée d'ordre* n *en quatre matrices carrées d'ordre* $n/2$.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad C = A * B = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

On obtient ainsi un algorithme récursif fondé sur les égalités suivantes :

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21} \qquad C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$
$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21} \qquad C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

2- Donner une relation de récurrence sur le nombre d'opérations effectuées par cet algorithme en fonction du format n *des matrices et en déduire le comportement asymptotique de la complexité en temps.*

Une seconde propriété permet également de réaliser le produit de matrices. Elle est fondée sur la même décomposition des matrices carrées d'ordre n *en quatre matrices d'ordre* $n/2$. *En posant :*

$$P_1 = (A_{11} + A_{22}) * (B_{11} + B_{22}) \qquad P_2 = (A_{21} + A_{22}) * B_{11}$$
$$P_3 = A_{11} * (B_{12} - B_{22}) \qquad P_4 = A_{22} * (-B_{11} + B_{21})$$
$$P_5 = (A_{11} + A_{12}) * B_{22} \qquad P_6 = (-A_{11} + A_{21}) * (B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$

on en déduit un algorithme de type diviser pour régner *pour calculerla matrice produit* C. *Il repose sur les relations suivantes :*

$$C_{11} = P_1 + P_4 - P_5 + P_7 \qquad C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4 \qquad C_{22} = P_1 - P_2 + P_3 + P_6$$

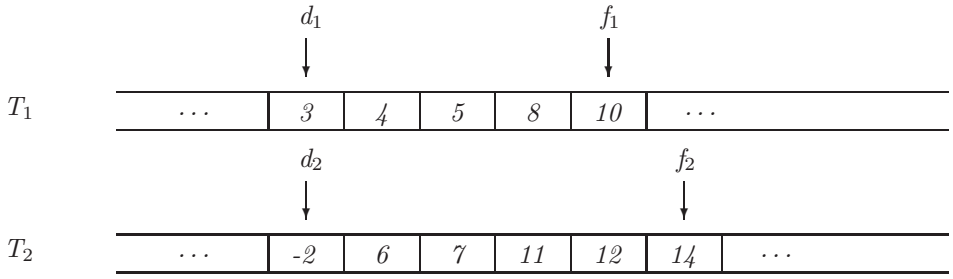
3- Trouver la complexité de cet algorithme.

Exercice 9.3 *On se propose de programmer en Java le tri fusion sur des tableaux d'entiers. On utilisera pour ce faire la classe* Tris *ci-après.*

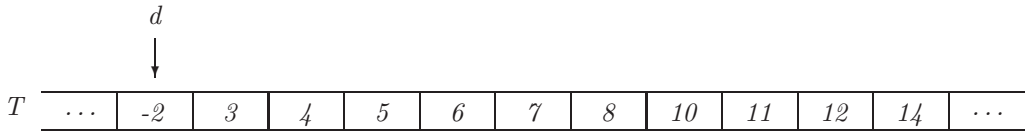
```
public class Tris{

   static void affiche(int[]t){
        int i=0;

        while(i<t.length)
            System.out.print(t[i++]+ "  ");
        System.out.println();
    }

    public static void main (String [] args){

        int[] t = new int[args.length];
        int i;

        for(i=0;i<args.length;i++)
            t[i]=Integer.parseInt(args[i]);

        affiche(t);
        Tri_Fusion.tri(t);
        affiche(t);
    }
}
```

Il s'agit donc de concevoir la classe Tri_Fusion*, comportant une méthode* tri *effectuant un tri fusion sur le tableau en argument.*

Considérons tout d'abord le problème de la fusion. Etant données deux portions de tableaux, supposées rangées par ordre croissant, on se propose de les fusionner dans un troisième tableau T*, à partir d'un certain indice* d*. Ainsi sur les deux portions de tableau décrites ci-dessous :*



on doit obtenir le résultat suivant :



1- Ecrire une méthode fusionne *qui prend en paramètres les tableaux* T_1*,* T_2 *et* T *ainsi que les entiers* d_1*,* f_1*,* d_2*,* f_2 *et* d *et qui fusionne les parties concernées de* T_1 *et* T_2 *dans* T *comme indiqué sur l'exemple.*

Considérons maintenant un tableau t *passé en variable privée de la classe* Tri_Fusion*. Soit* (début, fin) *un couple d'entiers. En supposant que les deux moitiés de la portion de* t *débutant à l'indice* début *et s'achevant l'indice* fin *ont été triées, il s'agira alors de les fusionner* en place*. L'expression* en place *signifie que le résultat de la fusion doit se retrouver dans la même partie de* t *(celle qui débute à l'indice* début*).*

2- Démontrer rigoureusement qu'il suffit pour cela de recopier la première moitié de la portion concernée de t *dans un tableau auxiliaire* aux*, puis de fusionner le résultat de cette copie avec la deuxième moitié dans le tableau* t *dans* t*.*

3- Ecrire alors une méthode fusion *qui effectue cette fusion en place pour deux entiers* début *et* fin *passés en paramètre. Les tableaux* t *et* aux *sont supposés construits par ailleurs.*

4- Ecrire une méthode tri_fusion *qui applique le tri fusion à un segment du tableau* t *dont les indices de début et de fin sont passés en paramètres.*

5- Terminer l'écriture de la classe Tri_Fusion *en donnant une méthode* tri *qui prend en paramètre un tableau* A *et lui applique le tri fusion.*