Algorithmique *(Agrégation)* [1]

Serge Haddad
Professeur de l'ENS Cachan
61, Avenue du Président Wilson
94235 Cachan cedex, France
adresse électronique : haddad@lsv.ens-cachan.fr
page personnelle : www.lsv.ens-cachan.fr/∼haddad/

2 avril 2014

1. Merci à Paul Gastin pour les transparents de son cours d'algorithmique en L3 et à Thomas Place pour ses sujets de TD.

Table des matières

Chapitre 1

Preuve de programmes

Livres recommandés

The Formal Semantics of Programming Languages. An Introduction
Glynn Winskel
Foundations of Computing Series, The MIT Press. 1993

Pour ce qui concerne la preuve de programmes, je recommande la lecture de ce livre jusqu'au chapitre 7.

1.1 Algorithmes : notations

Sauf mention du contraire, on considèrera que l'algorithme est défini par une fonction dont les entrées sont indiquées par le mot-clef **Input** et les sorties (il peut y en avoir plusieurs) sont indiqués par le mot-clef **Output**.

Les déclarations des variables locales de la fonction sont indiquées par le mot-clef **Data** dans le corps de la fonction après les entrées et les sorties et avant le bloc d'instructions de la fonction.

L'opérateur d'affectation est noté $\leftarrow$ à ne pas confondre avec le test d'égalité $=$ utilisé pour construire des expressions booléennes. On n'utilisera le « ; » que pour concaténer des instructions placées sur la même ligne. Les opérateurs de comparaison sont notés $=, \neq, <, >, \leq, \geq$, les opérateurs arithmétiques sont notés $+, *$ et les opérateurs logiques sont indifféremment notés **and** ou $\wedge$, **or** ou $\vee$, **not** ou $\neg$.

Lorsque nous introduirons un opérateur spécifique à un nouveau type de donnée, nous en préciserons la signification. Nous travaillerons avec des tableaux dont la dimension sera précisée lors de leur définition. Pour accéder à un élément du tableau, on utilise la notation $Tableau[indice]$.

Les constructeurs de programmes (avec leur interprétation usuelle) sont :

— **if** *condition* **then**
  instructions
else if *condition* **then** (optionnel)
  ...
else (optionnel)
  instructions
end

— **while** *condition* **do**
  instructions
  end
— **repeat**
  instructions
  until *condition*
— **for** *indice* $\leftarrow$ *valeur initiale* **to** *valeur finale* **do**
  instructions
  end

Lorsqu'un bloc correspondant à un constructeur tient sur une ligne, nous omettons le mot-clef **end**.

Pour renvoyer le résultat d'une fonction, on utilise le mot-clef **return** *résultats*. A l'inverse, l'appel d'une fonction se note `NomFonction`(*paramètres*).

L'algorithme 1, que nous étudierons plus loin dans ce chapitre, illustre ces conventions d'écriture.

Algorithme 1: Tri d'un tableau

`Trie`(T, deb, fin) : tableau d'entiers

Input : T un tableau, $deb \leq fin$ deux indices du tableau
Output : un tableau contenant les valeurs de $T[deb]$ à $T[fin]$ triées

Data : T' un tableau de $fin + 1 - deb$ entiers
Data : T_1 un tableau de $\lfloor (fin + 1 - deb)/2 \rfloor$ entiers
Data : T_2 un tableau de $\lceil (fin + 1 - deb)/2 \rceil$ entiers

if $deb = fin$ **then**
  $T'[1] \leftarrow T[deb]$
else
  $T_1 \leftarrow$ `Trie`$(T, deb, \lfloor (deb + fin - 1)/2 \rfloor)$
  $T_2 \leftarrow$ `Trie`$(T, \lfloor (deb + fin - 1)/2 \rfloor + 1, fin)$
  $T' \leftarrow$ `Fusion`$(T_1, \lfloor (fin + 1 - deb)/2 \rfloor, T_2, \lceil (fin + 1 - deb)/2 \rceil)$
end
return T'

1.2 Introduction

Les programmes que nous considérons ont des entrées (notées symboliquement) e et des sorties s. La spécification d'un programme à développer est donnée par :

— un domaine de définition E sur les entrées (autrement dit les garanties que doit fournir l'appelant).
— une fonction (totale) f de E dans F, le domaine des valeurs possibles des sorties.

La preuve (ou vérification) d'un programme `prog` consiste à démontrer que :

(1) pour toute entrée $e \in E$, le programme `prog` se termine. On appelle cette propriété *la terminaison*.

(2) lorsque `prog` se termine pour une entrée $e \in E$ alors $s = f(e)$. On appelle cette propriété *la correction partielle*.

Les techniques de preuve de ces deux propriétés sont radicalement différentes. En règle générale, il est plus intéressant de commencer par établir la correction partielle car celle-ci peut aider à démontrer la terminaison. Il y a cependant un point commun à l'établissement de ces deux propriétés : la difficulté principale réside dans le traitement des boucles `while` dans le cas de programmes « impératifs » et le traitement des fonctions récursives dans le cas de programmes « fonctionnels ».

1.3 La terminaison

Remarquons d'abord que si on écrit des programmes dont toutes les boucles sont des `for` (i.e. dont le nombre de tours est déterminé au début de l'exécution de la boucle) et sans appel récursif (direct ou indirect) alors la terminaison est garantie.

Malheureusement même si ces programmes appelés *récursifs primitifs* ont un grand pouvoir d'expression, certains problèmes difficiles ne peuvent être résolus qu'avec des programmes non primitifs récursifs. D'autre part, il est souvent plus simple de concevoir dans un premier temps un programme avec appels récursifs ou avec boucle `while`. L'étude de la terminaison et de la correction partielle permet ensuite dans la plupart des cas de transformer le programme en un programme récursif primitif.

Le concept essentiel sous-jacent à la terminaison d'un programme est celui de relation *bien fondée*.

Définition 1 *Une relation binaire $\prec$ sur un ensemble E est bien fondée s'il n'existe pas de suite infinie $\{e_i\}_{i\in\mathbb{N}}$ de E vérifiant $\forall i \in \mathbb{N}\ e_{i+1} \prec e_i$.*

Par abus de langage, on dit qu'un ensemble est bien fondé si la relation associée ne prête pas à confusion. Enonçons quelques propriétés élémentaires des relations bien fondées. On dira aussi qu'une suite qui vérifie la propriété de la définition 1 est une suite *invalidante*.

Proposition 1
Une relation bien fondée $\prec$ est irréflexive. Sa fermeture transitive $\prec^+$ est aussi bien fondée et sa fermeture transitive et réflexive $\prec^*$ est un ordre partiel.

Une relation $\prec$ est bien fondée ssi tout sous-ensemble non vide Fde E admet (au moins) un élement minimal *$m \in F$, i.e. $\forall x \in F\ x \not\prec m$. Par conséquent, $(\mathbb{N}, <)$ est un ensemble bien fondé.*

Preuve
Supposons qu'il existe un élément a tel que $a \prec a$, alors on fabrique la suite $\ldots a \prec a \prec a$. Supposons qu'on ait une suite infinie $\{e_i\}_{i\in\mathbb{N}}$ de E vérifiant $\forall i \in \mathbb{N}\ e_{i+1} \prec^+ e_i$ alors on déplie la suite en insérant les éléments intermédiaires. Puisque $\prec^+$ est irréflexive, $\prec^*$ est un ordre partiel.

Soit $\prec$ une relation bien fondée et $F \subset E$. Choisissons un élément arbitraire a_0 de F. S'il n'est pas minimal, on choisit a_1 tel que $a_1 \prec a_0$. On ne peut itérer

indéfiniment ce procédé car on construirait alors une suite infinie invalidante. Le dernier élément de la suite finie est donc un élément minimal de F.

Supposons qu'il existe une suite infinie $\{e_i\}_{i\in\mathbb{N}}$ invalidante. Alors l'ensemble des éléments de cette suite infinie n'admet pas d'élément minimal.

Soit $F \subset \mathbb{N}$ non vide, $\inf(F)$ existe puisque F est minorée et $\inf(F) \in F$ puisque la topologie sur $\mathbb{N}$ est discrète.

c.q.f.d. ◊◊◊

La proposition suivante permet de construire (inductivement) des relations bien fondées.

Proposition 2 *Si $(E, \prec_E)$ et $(F, \prec_F)$ sont des ensembles bien fondés alors $(E \times F, \prec)$ est bien fondé avec $\prec$ défini par $(m, n) \prec (m', n')$ ssi $m \prec_E m'$ ou $m = m'$ et $n \prec_F n'$.*

Preuve
Supposons qu'il existe une suite infinie de paires $\{(m_i, n_i)\}_{i\in\mathbb{N}}$ invalidante pour $\prec$. Les éléments de la suite m_i sont finis sinon en les prenant dans leur ordre d'apparition dans la suite, on construit une suite invalidante pour $(\prec_E)^+$. Soit m_{i_0} le dernier élément à apparaître. On a alors $\forall i \geq i_0\ m_i = m_{i_0}$: sinon $\exists i < i_0 < j$ avec $m_i = m_j$ et $m_i(\prec_E)^+ m_i$. La suite $\{n_i\}_{i\geq i_0}$ est invalidante pour $\prec_F$ ce qui aboutit à une contradiction.

c.q.f.d. ◊◊◊

La sortie d'une boucle `while` s'établit ainsi. On construit une fonction f des états du programme s qui satisfont la condition de boucle dans un ensemble bien fondé. Puis on démontre que si s est l'état du programme au début d'un tour et s' l'état au début du tour suivant, on a $f(s') \prec f(s)$. Dans les cas simples, cette fonction s'exprime syntaxiquement sous forme d'une expression bâtie à partir des variables et dans les cas les plus complexes elle représente un attribut d'un objet manipulé par le programme (e.g. la hauteur d'un arbre). Ce type de raisonnement est illustré par l'exercice relatif à l'algorithme 5.

Le cas des appels récursifs est similaire mais cette fois-ci la fonction f envoie la valeur des paramètres lors d'un appel dans un ensemble bien fondé. On démontre que si p est la valeur des paramètres d'une fonction et p' la valeur de ces paramètres lors d'un appel récursif, on a $f(p') \prec f(p)$. Ce type de raisonnement est illustré par l'exercice relatif à l'algorithme 7.

Nous concluons cette étude en montrant qu'il n'est pas possible dans le cas général de « mécaniser » le test de terminaison.

Proposition 3 (Terminaison d'un programme à paramètres) *La terminaison d'un programme* `prog`*, prenant en entrée un paramètre entier x est un problème indécidable.*

Preuve
Nous allons démontrer ce résultat par l'absurde. Supposons qu'il existe un programme `testarret` prenant en entrée deux paramètres entiers : une représentation (par un entier) d'un programme `prog` et une valeur d'entrée de ce programme. Le choix de la représentation du programme est ici sans importance ;

par exemple, on pourrait choisir comme représentation le nombre entier correspondant à la suite de bits du programme (en prenant soin d'ajouter un bit de poids fort à 1 pour éviter une ambiguïté au décodage). On notera $\overline{\texttt{prog}}$ cette représentation. `testarret` renvoie vrai si `prog` s'arrête avec la valeur fournie et sinon renvoie faux. Le comportement de `testarret` est indéterminé si le premier paramètre n'est pas la représentation d'un programme.
Nous construisons alors un programme `fou` à un paramètre entier qui fonctionne ainsi.

- `fou` appelle `testarret`(x, x). Autrement dit, il teste si le programme `prog` s'arrête en prenant comme entrée sa représentation.
- Si `testarret`(x, x) renvoie vrai, alors `fou` boucle sans fin sinon il s'arrête.

Examinons le comportement de `fou`$(\overline{\texttt{fou}})$.

- Si `fou`$(\overline{\texttt{fou}})$ s'arrête alors `testarret`$(\overline{\texttt{fou}}, \overline{\texttt{fou}})$ renvoie vrai et par conséquent `fou`$(\overline{\texttt{fou}})$ ne s'arrête pas ce qui est absurde.
- Dans le cas contraire, `testarret`$(\overline{\texttt{fou}}, \overline{\texttt{fou}})$ renvoie faux et par conséquent `fou`$(\overline{\texttt{fou}})$ s'arrête ce qui est absurde. Il n'existe donc pas de programme `testarret`.

c.q.f.d. ◊◊◊

Le fait que le programme ait un paramètre en entrée est tout à fait accessoire comme l'indique le corollaire suivant. Par ailleurs, celui-ci illustre le principe de réduction.

Corollaire 1 (Terminaison d'un programme sans paramètre) *La terminaison d'un programme prog sans paramètre est un problème indécidable.*

Preuve
Montrons que le problème de la terminaison d'un programme à un paramètre est réductible au problème de la terminaison d'un programme sans paramètre. Nous supposons donc qu'il existe un programme `testarretbis` pour le problème du corollaire et nous décrivons comment construire un programme `testarret`. Soit `prog` un programme à un paramètre et x une valeur entière. Alors `testarret` fonctionne comme suit :

- `testarret` construit la représentation du programme $\texttt{prog}'$ sans paramètre qui consiste à appeler `prog`(x).
- Puis `testarret` appelle `testarretbis`$(\overline{\texttt{prog}'})$ et renvoie le résultat correspondant.

Donc `testarretbis` ne peut exister.

c.q.f.d. ◊◊◊

Nous en déduisons un deuxième corollaire.

Corollaire 2 *L'ensemble des (codes des) programmes sans paramètre qui ne se terminent pas est un ensemble non récursivement énumérable.*

Preuve
L'ensemble des (codes des) programmes sans paramètre qui se terminent est un ensemble récursivement énumérable. Le programme qui les énumère consiste en une boucle infinie. Au tour i de cette boucle, on exécute i pas élémentaires des i premiers programmes et on affiche ceux qui se terminent. Tout programme i qui

se termine est affiché au tour $\max(i, j)$ où j est le nombre de pas élémentaires du programme i jusqu'à la terminaison.

Par conséquent, si l'ensemble des (codes des) programmes sans paramètre qui ne se terminent pas était un ensemble récursivement énumérable, on construirait par un procédé classique une procédure de décision pour la terminaison des programmes. Pour savoir si un programme se termine, on exécute en parallèle les deux énumérations et on décide lorsque le programme apparaît sur l'une des listes (ce qui arrive nécessairement).

c.q.f.d. ◊◊◊

1.4 Correction partielle

Afin d'établir la correction partielle, il nous faut exprimer des assertions relatives à l'état du programme. Dans la plupart des cas, la logique du premier ordre enrichie avec les opérations des types des variables (e.g. incluant l'arithmétique sur les entiers) et incluant les variables du programme (à ne pas confondre avec les variables de la logique) est suffisante. Par exemple, pour exprimer que dans le tableau t de dimension n, la cellule $t[i]$ contient la valeur maximale on écrira l'assertion :

$$\forall x \ (x \geq 1 \wedge x \leq n) \Rightarrow t[x] \leq t[i]$$

ou de manière plus relachée :

$$\forall 1 \leq x \leq n \ t[x] \leq t[i]$$

L'objectif est alors d'établir des énoncés du type :

$$\{Pre\}\texttt{pg}\{Post\}$$

où Pre et $Post$ sont des assertions et `pg` est un programme. La sémantique d'un tel énoncé est (en langage naturel) la suivante : *« A partir d'un état s vérifiant Pre, si l'exécution de* `pg` *se termine alors l'état atteint s' vérifie $Post$ ».*

Afin de définir formellement la validité d'un énoncé, on introduit plusieurs notations :

- Une interprétation I est une valuation pour les variables de la logique.
- Soit une expression E, une assertion A, une interprétation I et un état s : $E^{I,s}$ (resp. $A^{I,s}$) désigne la valeur de E (resp. A) dans l'état s pour l'interprétation I. Puisque les variables libres de ces expressions sont soit les variables du programme soit les variables libres de l'expression, la valeur est déterminée de manière unique. On note $s \models_I A$ ssi $A^{I,s} =$ **true**.
- Soit A une assertion et I une interprétation, $[\![A]\!]_I$ désigne l'ensemble des états (de programme) qui vérifient l'assertion A pour l'interprétation I : $[\![A]\!]_I = \{s \mid s \models_I A\}$.
- Soit s un état et `pg` un programme, $s \cdot \texttt{pg}$ désigne l'état éventuel atteint par l'exécution de `pg` à partir de s. Dans le cas où le programme ne se termine pas, on note $s \cdot \texttt{pg} = \bot$.

On dit qu'un énoncé $\{Pre\}\texttt{pg}\{Post\}$ est valide ce qu'on note $\models \{Pre\}\texttt{pg}\{Post\}$ si pour toute interprétation I :

$$\forall s \in [\![Pre]\!]_I \ s \cdot \texttt{pg} \neq \bot \Rightarrow s \cdot \texttt{pg} \in [\![Post]\!]_I$$

On désire établir ces énoncés par une induction structurelle sur les programmes. Aussi on va se doter d'axiomes et de règles de déduction.

Le premier axiome est relatif au programme qui ne fait rien (excepté se terminer !) que l'on note `skip`.

$$\frac{}{\{A\}\texttt{skip}\{A\}}$$

Cet axiome semble inutile à première vue mais il est intéressant dans le cas d'une structure `if...then...else...` dégénérée où la deuxième branche de l'alternative est omise donc équivalente à `skip`.

Cet axiome est en fait un cas particulier d'un axiome plus général qui requiert *qu'aucune variable de programme intervenant dans A ne soit modifiée par* `pg`.

$$\frac{}{\{A\}\texttt{pg}\{A\}}$$

L'axiome suivant est relatif à l'affectation d'une variable.

$$\frac{}{\{A[E/X]\}X \leftarrow E\{A\}}$$

où $A[E/X]$ désigne l'assertion A dans laquelle on a substitué à toutes les occurrences de X l'expression E.

Fixons-nous une interprétation I. Soit s un état qui vérifie $A[E/X]$ et s' l'état obtenu après affectation de la variable, on démontre par induction sur la structure de A que $A[E/X]^{I,s} = A^{I,s'}$. Seuls les cas de base sont intéressants : $E^{I,s} = X^{I,s'}$ et $Y^{I,s} = Y^{I,s'}$ pour une variable Y différente de X. Cet axiome naturel appelle cependant un commentaire qui a son importance si on veut rendre cette approche effective. La précondition est construite à partir de la postcondition et il est difficile (voire impossible) d'imaginer un axiome qui procèderait dans l'autre direction.

Remarquons qu'il s'agit d'une instruction d'affectation « limitée » puisque la variable qui est affectée est une variable simple et non pas la cellule d'un tableau ni un élément accédé via un pointeur. Moyennant certaines précautions, cet axiome s'étend à ce type d'affectation et nous nous en servirons lors des exercices.

La première règle de déduction concerne la concaténation de programmes.

$$\frac{\{A\}\ \texttt{pg}\ \{B\} \qquad \{B\}\ \texttt{pg}'\{C\}}{\{A\}\ \texttt{pg}\ ;\ \texttt{pg}'\{C\}}$$

Cette règle ne nécessite pas d'explications complémentaires. En la combinant avec l'axiome d'affectation, on peut déjà prouver des propriétés intéressantes (voir l'exercice d'échange de variables).

La deuxième règle de déduction concerne l'instruction conditionnelle et correspond à une étude de cas.

$$\frac{\{A \wedge c\}\ \texttt{pg}\ \{B\} \qquad \{A \wedge \neg c\}\ \texttt{pg}'\{B\}}{\{A\}\ \texttt{if}\ c\ \texttt{then pg else pg}'\{B\}}$$

Notons qu'ici aussi les préconditions des hypothèses sont construites à partir du programme alors que la postcondition n'est pas modifiée.

La troisième règle de déduction concerne l'instruction `while`.

$$\frac{\{A \wedge c\}\ \texttt{pg}\ \{A\}}{\{A\}\ \texttt{while}\ c\ \texttt{do pg}\ \{A \wedge \neg c\}}$$

L'assertion A apparaît à la fois dans les préconditions et les postconditions. On appelle une telle assertion un *invariant de boucle*. Cette règle est la plus difficile à appliquer car il s'agit de deviner un invariant utile à l'établissement des propriétés du programme. L'exercice relatif à l'extraction de boules d'une urne en est une bonne illustration.

Les invariants de boucle sont obtenus généralement en renforçant les assertions qu'on cherche à établir. Aussi la *règle de conséquence* permet de retrouver les assertions visées.

$$\frac{\models (A \Rightarrow A') \qquad \{A'\}\ \texttt{pg}\ \{B'\} \qquad \models (B' \Rightarrow B)}{\{A\}\ \texttt{pg}\ \{B\}}$$

Dans cette règle $\models \phi$ signifie que pour tout état s et toute interprétation I, on a $s \models_I \phi$. On omet souvent cette règle lorsque les déductions $A \Rightarrow A'$ et $B \Rightarrow B'$ sont immédiates. Observons que la vérification de cette règle n'est pas nécessairement automatisable puisque $\models$ est un prédicat de nature sémantique.

La règle de conjonction permet de décomposer des preuves en sous-preuves indépendantes.

$$\frac{\{A\}\ \texttt{pg}\ \{B\} \qquad \{A'\}\ \texttt{pg}\ \{B'\}}{\{A \wedge A'\}\ \texttt{pg}\ \{B \wedge B'\}}$$

Exemple. Nous illustrons l'usage de ces règles sur un algorithme itératif du calcul de factorielle. Cet exemple appelle plusieurs commentaires. Plutôt que de représenter l'arbre de déduction, on procède par annotation du programme ce qui allège considérablement les notations. D'autre part, on procède implicitement à la règle de conséquence lorsque son emploi est évident comme dans $n - 1 \geq 0 \Leftrightarrow n > 0$.

Nous ne traitons qu'un cas d'appel de fonction, l'appel d'une fonction sans résultat avec un paramètre v passé par valeur et *non modifié* durant l'exécution de la fonction et un paramètre r passé par référence. Cette fonction a accès aux variables globales. Autrement dit, les variables sur lesquelles portent les assertions de la fonction (disons `f`) sont les variables globales et les paramètres formels. La règle impose que :

— X, la variable passée en référence n'apparaisse pas dans le corps de `f` (ni dans les appels imbriqués d'autres fonctions) ou corresponde à une variable locale de `f` (ce qui masque la variable d'appel). Par conséquent X n'apparaît pas non plus dans les formules A et B.

Algorithme 2: Calcul itératif de la factorielle

`Fact`(n) : un entier
Input : n un entier positif
Output : f un entier finalement égal à $n!$
$\{n \geq 0 \wedge n = v\}$ $f \leftarrow 1$ $\{n \geq 0 \wedge n! * f = v!\}$
while $n > 0$ **do**
  $\{n > 0 \wedge n! * f = v!\}$ $f \leftarrow f * n$
  $\{n > 0 \wedge (n-1)! * f = v!\}$ $n \leftarrow n - 1$
  $\{n \geq 0 \wedge n! * f = v!\}$
end
$\{n \geq 0 \wedge n \leq 0 \wedge n! * f = v!\} \equiv \{n = 0 \wedge f = v!\}$
return f

— E, l'expression passée par valeur ne contienne pas X et vérifie qu'aucune de ses variables ne soit modifiée lors de l'appel de `f`.

$$\frac{\{A\}\texttt{corps de f}(r, v)\{B\}}{\{A[X/r][E/v]\}\ \texttt{f}(X, E)\{B[X/r][E/v]\}}$$

Exemple. Soit la fonction

$$\texttt{Inc}(r, v)\{r \leftarrow r + v\}$$

qui ajoute à la variable r la valeur de v. D'après la règle d'affectation,

$$\{r = z\}\{r \leftarrow r + v\}\{r = z + v\}$$

En appliquant la règle des fonctions à l'appel, $\texttt{Inc}(X, 2 * Y + T)$, on obtient :

$$\{X = z\}\texttt{Inc}(X, 2 * Y + T)\{X = z + 2 * Y + T\}$$

La règle précédente ne permet pas a priori de construire une preuve de correction partielle d'une fonction récursive. Pour le cas des fonctions récursives on introduit l'axiome suivant valable uniquement si :

— la racine de l'arbre est étiquetée par $\{A\}\texttt{f}(r, v)\{B\}$;
— l'axiome ne s'applique pas à la racine de l'arbre ;
— les conditions de l'appel non récursif sont vérifiées.

$$\frac{}{\{A[X/r][E/v]\}\texttt{f}(X, E)\{B[X/r][E/v]\}}$$

De manière a priori surprenante, l'utilisation de cette règle pour les appels récursifs de `f` alors qu'on cherche à établir la validité de $\{A\}\texttt{f}(r, v)\{B\}$ est justifiée ! Ceci est dû au fait que l'on n'établit que la correction partielle. Autrement dit, si la fonction se termine alors la « preuve » des appels terminaux ne s'appuie pas sur cet axiome. Puis en remontant l'arbre des appels récursifs chaque preuve est justifiée par les preuves des appels des niveaux inférieurs.

Exemple. Nous illustrons ce point en analysant un calcul récursif de la factorielle obtenu par l'algorithme 3. L'énoncé à démontrer est $\{n \geq 0\}\texttt{Fact}(n)\{m = n!\}$ car n est passé par valeur.

Algorithme 3: Calcul récursif de la factorielle

Data : m un entier finalement égal à $n!$
`Fact`(n)
Input : n un entier positif
$\{n \geq 0\}$
if $n = 0$ **then**
  $\{n = 0\}$
  $m \leftarrow 1$ $\{n = 0 \wedge m = n!\}$
else
  $\{n - 1 \geq 0\}$ `Fact`$(n - 1)$
  $\{m = n - 1!\}$ $m \leftarrow n * m$ $\{m = n!\}$
end

Il n'y a pas ici de variable passée par référence et la variable locale n n'est pas modifiée lors d'un appel récursif (elle est même masquée par la variable locale de l'appel récursif).

Lorsqu'à l'aide du système de déduction, on établit une preuve de l'énoncé $\{Pre\}\texttt{pg}\{Post\}$, on le note $\vdash \{Pre\}\texttt{pg}\{Post\}$. Nous avons établi (plus ou moins formellement) que si $\vdash \{Pre\}\texttt{pg}\{Post\}$ alors $\models \{Pre\}\texttt{pg}\{Post\}$. Autrement dit, le système de déduction est correct.

1.5 Complétude du système de déduction

Intéressons-nous maintenant à la complétude du système de déduction. Supposons que l'énoncé $\{A\}\texttt{pg}\{B\}$ soit correct, existe-il une preuve dans notre système de déduction de cet énoncé ? Autrement dit :

$$\models \{Pre\}\texttt{pg}\{Post\} \Rightarrow \vdash \{Pre\}\texttt{pg}\{Post\} \ ?$$

Nous nous limitons ici à un modèle simple de programmes :
— uniquement des variables entières ;
— des expressions arithmétiques formées à partir l'addition, la soustraction entière et la multiplication ;
— l'affectation et `skip` comme intructions élémentaires ;
— la concaténation, `if` et `while` comme structures de contrôle ;
— pas d'appels de fonctions.

Ce modèle est suffisamment expressif pour illustrer notre propos. Du point de vue théorique, il est équivalent aux machines de Turing ce qui est suffisant et du point de vue pratique, il permet de coder des algorithmes typiques. Il ne serait pas difficile d'étendre les résultats de cette section si le modèle incluait les tableaux d'entiers.

Première étape. La première étape de la démonstration de complétude consiste à montrer qu'étant donnée une post-condition B, un programme $\texttt{pg}$, il existe une plus faible précondition $wp(B, \texttt{pg})$ pour laquelle l'énoncé $\{wp(B, \texttt{pg})\}\texttt{pg}\{B\}$ est correct.

L'assertion $wp(B, \texttt{pg})$ doit alors vérifier pour toute interprétation I :

$$[\![wp(B, \texttt{pg})]\!]_I = \{s \mid s \cdot \texttt{pg} \neq \bot \Rightarrow s \cdot \texttt{pg} \models_I B\}$$

Puisque la définition de la plus faible précondition est sémantique, il n'y a pas unicité de cette assertion. Cependant, cette même définition assure que si A et A' sont deux plus faibles préconditions, on a $\models A \Leftrightarrow A'$. Observons qu'un état entrainant la non terminaison de pg satisfait la plus faible précondition de n'importe quelle assertion via pg. De manière plus précise, $[\![wp(\mathbf{false}, \mathtt{pg})]\!]_I$ est exactement l'ensemble des états pour lesquels pg ne se termine pas.

L'existence d'une plus faible précondition s'établit par induction structurelle. Nous laissons au lecteur le soin de prouver les affirmations suivantes :

— $wp(B, \mathtt{skip}) \Leftrightarrow B$
— $wp(B, X \leftarrow E) \Leftrightarrow B[E/X]$
— $wp(B, \mathtt{pg}; \mathtt{pg}') \Leftrightarrow wp(wp(B, \mathtt{pg}'), \mathtt{pg})$
— $wp(B,\ \mathtt{if}\ c\ \mathtt{then\ pg\ else\ pg}') \Leftrightarrow (wp(B, \mathtt{pg}) \wedge c) \vee (wp(B, \mathtt{pg}') \wedge \neg c)$

La partie difficile est l'existence d'une plus faible précondition pour le constructeur while. Nous allons procéder en décrivant tout d'abord la sémantique de cette précondition par une formule qui n'est pas une assertion puis en opérant des transformations justifiées afin d'obtenir une assertion équivalente. Dans ce développement on fixe une interprétation I. Nous démontrons simultanément que les seules variables libres (de la logique) de la plus faible précondition d'une assertion B sont incluses dans les variables libres de B.

La première formulation en français ne nécessite pas d'explications complémentaires.

s appartient à $[\![wp(B,\ \mathtt{while}\ c\ \mathtt{do\ pg})]\!]_I$ ssi
pour tout k, si partant de s le programme a pu exécuter $k-1$ tours
alors après le $k$ième tour soit B soit c est vérifié.

La deuxième formulation rend explicite les états rencontrés.

$$\forall k\ \forall s_0, \ldots, s_k\ (s = s_0 \wedge \forall 0 \leq i < k\ s_i \models_I c \wedge s_i \cdot \mathtt{pg} = s_{i+1}) \Rightarrow s_k \models_I c \vee B$$

Afin de parvenir à la troisième formulation, on remarque que la partie significative d'un état vis à vis du programme while c do pg et de l'assertion B est uniquement l'état des variables apparaissant dans c, pg et B. Notons $\overline{X} = X_1, \ldots, X_l$ ces variables et $\overline{v} = v_1, \ldots, v_l$ des valeurs possibles de ces variables. La substitution des valeurs aux variables dans une assertion A est notée $A[\overline{v}/\overline{X}]$.

Observons qu'étant donné un état s dont l'état des variables $\overline{X}$ est égal à $\overline{v}$ et un vecteur de valeurs $\overline{v}'$, on a l'équivalence suivante :

$$\exists s'\ s \cdot\ \mathtt{pg}\ = s' \wedge \overline{X}^{s'} = \overline{v}' \text{ ssi } \models \big(wp(\overline{X} = \overline{v}', \mathtt{pg}) \wedge \neg wp(\mathbf{false}, \mathtt{pg})\big)\ [\overline{v}/\overline{X}]$$

La satisfaction de la formule à droite ne dépend pas de l'interprétation I ni de l'état puisqu'elle n'a pas de variable libre et que les variables de programme ont été substituées par des valeurs. Les égalités vectorielles ci-dessus doivent être comprises comme des conjonctions d'égalités scalaires. Cette équivalence est justifiée car :

— le premier composant du terme droit de l'équivalence signifie que si partant d'un état s vérifiant $\overline{X} = \overline{v}$ le programme pg se termine alors l'état atteint vérifie $\overline{X} = \overline{v}'$;

— et le deuxième composant du terme droit de l'équivalence signifie que partant d'un état s vérifiant $\overline{X} = \overline{v}$ le programme pg se termine (car les seuls états qui peuvent établir **false** sont ceux pour lesquels pg ne se termine pas).

D'où la troisième formulation :

$$\forall k\ \forall \overline{v}_0, \ldots, \overline{v}_k$$
$$\big(\overline{X} = \overline{v}_0 \wedge \forall 0 \leq i < k\ c[\overline{v}_i/\overline{X}] \wedge (wp(\overline{X} = \overline{v}_{i+1}, \mathtt{pg}) \wedge \neg wp(\mathbf{false}, \mathtt{pg}))[\overline{v}_i]/\overline{X}\big)$$
$$\Rightarrow (c \vee B)[\overline{v}_k/\overline{X}]$$

Cette troisième formulation est très proche d'une assertion à l'exception de la quantification sur la suite $\overline{v}_0, \ldots, \overline{v}_k$.

Afin d'éliminer la référence explicite à cette suite, nous nous appuyons sur le lemme suivant.

Lemme 1 *Soit $\beta(a, b, i, x)$ le prédicat sur les entiers défini par :*

$$\beta(a, b, i, x) \stackrel{def}{=} x = a \mod (1 + (1 + i)b)$$

Alors pour toute séquence $n_0, \ldots, n_k$ d'entiers, il existe deux nombres n et m tels que pour tout $0 \leq j \leq k$ et tout x, on ait :

$$\beta(n, m, j, x) \Leftrightarrow x = n_j$$

Preuve

Posons $m = \max\{k, n_0, \ldots, n_k\}!$ et $p_i = 1 + (1 + i)m$ pour $0 \leq i \leq k$.

Pour $i < j$, on a $pgcd(1+(1+i)m, 1+(1+j)m) = pgcd(1+(1+i)m, (j-i)m) = 1$. En effet si p est un nombre premier qui divise $(j - i)m$ alors soit p divise m, soit p divise $j - i$ donc m. Donc p ne divise pas $1 + (1 + i)m$.

Puisque $\prod_{j \neq i} p_j$ et p_i sont premiers entre eux, il existe u_i et v_i tels que $u_i \prod_{j \neq i} p_j + v_i p_i = 1$. Soit u'_i défini par $u'_i = u_i \prod_{j \neq i} p_j$, $u'_i \mod p_i = 1$ et $u'_i \mod p_j = 0$ pour $j \neq i$. Posons $n = \sum_{i=0}^{k} u'_i n_i$. Par construction $n_i = n_i \mod p_i = n \mod p_i$ $(n_i < p_i)$.

c.q.f.d. ◊◊◊

Par conséquent, toute séquence finie est codée par deux entiers. Observons aussi que pour a, b, i fixés $\beta(a, b, i, x)$ est vraie pour exactement une valeur de x. Ceci nous conduit à l'assertion recherchée. Nous l'indiquons d'abord dans le cas où $\overline{X}$ est une unique variable X.

$$\forall k\ \forall m\ \forall n$$
$$(\beta(n, m, 0, X) \wedge \forall 0 \leq i < k\ \forall x\ \beta(n, m, i, x) \Rightarrow c[x/X]$$
$$\wedge \forall x\ \forall y\ (\beta(n, m, i, x) \wedge \beta(n, m, i + 1, y) \Rightarrow (wp(X = y, \mathtt{pg}) \wedge \neg wp(\mathbf{false}, \mathtt{pg}))[x/X]))$$
$$\Rightarrow (\forall x\ \beta(n, m, k, x) \Rightarrow (c \vee B)[x/X])$$

Le cas général consiste à considérer une séquence de $l(k + 1)$ nombres et d'associer $\beta(n, m, il + t, x)$ à la valeur de la variable X_t dans l'état s_i. Ceci ne

présente pas de difficultés majeures mais produit une formule très longue et peu lisible.

Deuxième étape. La deuxième étape de la démonstration de complétude consiste à établir l'existence d'une preuve pour un énoncé $\{A\}\texttt{pg}\{B\}$ tel que $\models \{A\}\texttt{pg}\{B\}$. On établit l'existence d'une preuve par induction sur la structure du programme. Pour chaque cas, il suffit de considérer $\{wp(B,\texttt{pg})\}\texttt{pg}\{B\}$ car le cas général s'obtient par application de la règle de conséquence à partir du cas particulier. En effet, supposons que $\models \{A\}\texttt{pg}\{B\}$. Par définition de la plus faible précondition, on a $\models A \Rightarrow wp(B,\texttt{pg})$. On complète donc la preuve $\{wp(B,\texttt{pg})\}\texttt{pg}\{B\}$ par la règle de conséquence.

$$\frac{\models (A \Rightarrow wp(B,\texttt{pg})) \qquad \{wp(B,\texttt{pg})\}\ \texttt{pg}\ \{B\} \qquad \models (B \Rightarrow B)}{\{A\}\ \texttt{pg}\ \{B\}}$$

Nous ne traitons que le cas le plus difficile : le constructeur `while`. Soit une assertion B, soit le programme `while` c `do pg` et soit $A \Leftrightarrow wp(B, \texttt{while}\ c\ \texttt{do pg})$. Montrons que :

1. $\models \{A \wedge c\}\texttt{pg}\{A\}$
2. $\models A \wedge \neg c \Rightarrow B$

1. On note que (par dépliage) :

$$\texttt{while}\ c\ \texttt{do pg} \equiv \texttt{if}\ c\ \texttt{then while}\ c\ \texttt{do pg else skip}$$

Soit un état s et une interprétation I, tels que $s \models_I A \wedge c$.
Si $s \cdot \texttt{pg} = \bot$, le résultat est immédiat. On suppose donc $s \cdot \texttt{pg} \neq \bot$.
Puisque $s \models_I c$, $s \cdot \texttt{while}\ c\ \texttt{do pg} = (s \cdot \texttt{pg}) \cdot \texttt{while}\ c\ \texttt{do pg}$.
Puisque $s \models_I A = wp(B, \texttt{while}\ c\ \texttt{do pg})$,

- Soit $s \cdot \texttt{while}\ c\ \texttt{do pg} = \bot$. Puisque $(s \cdot \texttt{pg}) \cdot \texttt{while}\ c\ \texttt{do pg} = s \cdot \texttt{while}\ c\ \texttt{do pg} = \bot$, on a $s \cdot \texttt{pg} \models A$ car tout état qui conduit à la non terminaison d'un programme satisfait la plus faible précondition via ce programme de toute postcondition.
- Soit $s \cdot \texttt{while}\ c\ \texttt{do pg} \neq \bot$. Puisque $s \cdot \texttt{while}\ c\ \texttt{do pg} \models_I B$, on a également $(s \cdot \texttt{pg}) \cdot \texttt{while}\ c\ \texttt{do pg} = s \cdot \texttt{while}\ c\ \texttt{do pg} \models_I B$.
  D'où $s \cdot \texttt{pg} \models_I wp(B, \texttt{while}\ c\ \texttt{do pg}) = A$.

2. Soit un état s et une interprétation I, tels que $s \models_I A \wedge \neg c$.
Puisque $s \models_I \neg c$, $s \cdot \texttt{while}\ c\ \texttt{do pg} = s$.
Puisque $s \models_I A$, $s = s \cdot \texttt{while}\ c\ \texttt{do pg} \models_I B$.

La première propriété permet d'établir une preuve de :

$$\{A\}\texttt{while}\ c\ \texttt{do pg}\{A \wedge \neg c\}$$

en utilisant l'hypothèse inductive et la règle du `while`. La deuxième propriété permet d'établir une preuve de :

$$\{A\}\texttt{while}\ c\ \texttt{do pg}\{B\}$$

en utilisant la règle de conséquence.

On obtient comme corollaire une forme faible du premier théorème de Gödel. Notez que ce résultat ne dépend que de la calculabilité de la plus faible précondition et pas de la complétude du système de déduction.

Théorème 1 *L'ensemble des assertions du premier ordre sur l'arithmétique dans* $\mathbb{N}$ *(avec l'addition et la multiplication comme opérateurs) n'est ni récursivement énumérable, ni co-récursivement énumérable.*

Preuve
Supposons par l'absurde que cet ensemble soit récursivement énumérable et donc récursif puisque soit une assertion est valide, soit sa négation l'est.

Soit maintenant pg un programme, on fabrique l'assertion $wp(\mathbf{false}, \mathtt{pg})$. Cette assertion fait intervenir les variables du programme mais on s'intéresse uniquement à l'exécution du programme lorsque les variables sont initialement nulles. On fabrique donc $wp(\mathbf{false}, \mathtt{pg})[\overline{0}/\overline{X}]$. Cette formule est une assertion du premier ordre sur l'arithmétique et elle est valide ssi pg ne se termine pas lorsqu'il démarre avec des variables nulles. On a donc une procédure de test de terminaison des programmes contrairement au corollaire 1.

c.q.f.d. $\Diamond\Diamond\Diamond$

1.6 Exercices

Exercice 1. Une urne contient initialement n boules noires et b boules blanches. Tant que l'urne contient au moins deux boules, on tire deux boules dans l'urne et :

— Si elles sont de la même couleur, on les jette.

— Sinon on jette la boule noire et on remet la boule blanche.

Si l'urne est vide, on remet une boule noire.
Quelle est la couleur de la boule restante dans l'urne ?

Exercice 2. Montrez que la séquence d'instructions suivantes échange le contenu des variables x et y : $x \leftarrow x + y; y \leftarrow x - y; x \leftarrow x - y;$

Exercice 3. Montrez que l'algorithme 4 trie le tableau S.

Algorithme 4: Tri d'un tableau

```
Tri(S, n) : un tableau
Input : S un tableau, n la dimension de S
Output : T, le tableau trié
Data : i, j deux indices
T ← S
for i from 2 to n do
    temp ← T[i] ; j ← i
    while j > 1 and then T[j − 1] > temp do T[j] ← T[j − 1] ; j ← j − 1
    T[j] ← temp
end
return T
```

Exercice 4.1 On considère un tableau T d'entiers naturels de dimension n. On modifie T par le procédé itératif suivant. Si on peut trouver deux indices $1 \leq i < j \leq n$ tels que $T[i] > T[j]$, alors on échange $T[i]$ et $T[j]$ et on essaye à nouveau sinon on stoppe. Remarquons que ce procédé est non déterministe puisque dans le cas où existent plusieurs paires (i, j) « telles que ... » on peut choisir l'une quelconque de ces paires. Que fait ce procédé ?

Exercice 4.2 On modifie la procédure d'échange. Soient deux indices $i < j$ tels que $T[i] > T[j]$. L'échange consiste maintenant à choisir deux nouvelles valeurs a_i et a_j telles que $T[i] \geq a_j \geq a_i \geq T[j]$ et à effectuer $T[j] \leftarrow a_j; T[i] \leftarrow a_i$. Autrement dit, on peut rapprocher les valeurs des éléments échangés. Par exemple, on pourra passer de $(0, 30, 20, 40)$ à $(0, 22, 28, 40)$). Que peut-on dire de ce nouveau procédé ?

Exercice 4.3 On modifie une dernière fois la procédure d'échange. Soient deux indices $i < j$ tels que $T[i] > T[j]$. L'échange consiste maintenant à choisir deux nouvelles valeurs a_i et a_j telles que $T[i] \geq a_j \geq a_i \geq T[j]$ **et** $a_j > T[j]$ puis à effectuer $T[j] \leftarrow a_j; T[i] \leftarrow a_i$. Autrement dit, $T[j]$ doit forcément croître après l'opération. Que peut-on dire de ce dernier procédé ?

Exercice 5. Que fait l'algorithme 5 avec $n \geq 0$ pour entrée ?

Exercice 6. Que fait l'algorithme 6 avec n un entier pour entrée ?

Exercice 7. Montrez que l'algorithme 7 se termine lorsqu'il a pour entrée des entiers naturels.

Exercice 8. Que fait l'algorithme 8 avec a, b des entiers naturels ?

Algorithme 5: Arithmétique dans les entiers

$a \leftarrow 0\,;\ s \leftarrow 1\,;\ t \leftarrow 1$
while $s \leq n$ **do**
  $a \leftarrow a + 1$
  $s \leftarrow s + t + 2$
  $t \leftarrow t + 2$
end

Algorithme 6: La fonction 91 de McCarthy

Data : r, une variable globale
$\texttt{F}(n)$
Data : x, une variable locale
if $n > 100$ **then** $r \leftarrow n - 10$
else $\texttt{F}(n + 11)\,;\ x \leftarrow r\,;\ \texttt{F}(x)$

Algorithme 7: La fonction d'Ackermann

$\texttt{Ack}(m, n)$: un entier
if $m = 0$ **then return** $n + 1$
if $n = 0$ **then return** $\texttt{Ack}(m - 1, 1)$
return $\texttt{Ack}(m - 1, \texttt{Ack}(m, n - 1))$

Algorithme 8: Une opération arithmétique

$\texttt{E}(a, b)$: un entier
$x \leftarrow a\,;\ y \leftarrow b\,;\ z \leftarrow 1$
while $y > 0$ **do**
  if y **mod** $2 = 1$ **then** $z \leftarrow z * x\,;\ y \leftarrow y - 1$
  $x \leftarrow x * x\,;\ y \leftarrow y/2$
end
return z

Chapitre 2

Complexité

Livres recommandés

Introduction à l'algorithmique
Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest, Clifford Stein
Deuxième édition, Sciences Sup, Dunod. 2004

Pour ce qui concerne la complexité, je recommande la lecture de la partie 1 de ce livre (chapitres 1 à 5) et le chapitre 17.

The Design and Analysis of Computer Algorithms
Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman
Addison-Wesley, Reading, Massachusetts, 1974

Pour ce qui concerne la complexité, je recommande la lecture du chapitre 1.

Elements d'algorithmique
Danièle Beauquier, Jean Berstel, Philippe Chrétienne
Masson, 1992

Ce livre est maintenant épuisé mais il est accessible sur l'internet à l'url `http://www-igm.univ-mlv.fr/~berstel/`. Pour ce qui concerne la complexité, je recommande la lecture des chapitres 1 et 2.

2.1 Problèmes et instances

2.1.1 Exemples

Planarité d'un graphe

Une instance du problème. Imaginons que dans le cadre de l'aménagement du territoire, on doive choisir le lieu d'implantation de trois usines et le lieu des sources de distribution du gaz, d'électricité et d'eau.

Le problème consiste à établir s'il est possible que le cablage et la tuyauterie allant des sources aux entreprises soit fait au même niveau du sous-sol et dans l'affirmative de produire un plan comprenant les usines, les sources et les liens.

La figure 2.1 décrit un plan qui conduit à un croisement et ne répond pas aux exigences énoncées.

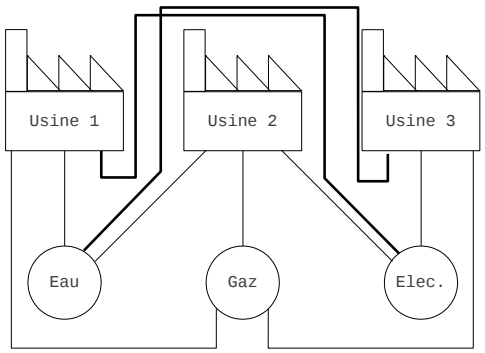

FIGURE 2.1: Un plan de cablage

	Orange	Banane	Pomme	Apport requis
Vitamine A	30 mg	40 mg	120 mg	15 mg
Vitamine B	14 mg	14 mg	10 mg	10 mg
Vitamine C	250 mg	8 mg	44 mg	75 mg
Prix au kg	4 €	2 €	4.5 €	

TABLE 2.1: Informations nécessaires au régime

Le problème. On se donne n lieux à placer sur le plan et m couples de lieux qui doivent être liés. Le problème à résoudre consiste à déterminer, s'il est possible de placer les lieux et les liens sur le plan de telle sorte qu'il n'y ait aucun croisement et dans l'affirmative de produire un tel plan.

Programmation linéaire

Une instance du problème. Supposons qu'une personne doive suivre un régime nutritif en fruits qui lui garantit un apport quotidien suffisant en vitamines A, B et C.

Cette personne ne mange que trois fruits : des oranges, des bananes et des pommes. Pour chaque fruit, elle connaît son prix et la quantité de vitamines par kilo comme indiqué dans le tableau 2.1.

Le problème à résoudre consiste à déterminer une quantité (éventuellement fractionnaire) de chaque fruit de telle sorte que l'apport requis en vitamines soit atteint et que le prix du régime soit *minimal*.

De manière plus formelle, il s'agit de trouver parmi les triplets (xo, xb, xp) qui vérifient :

$$30 \cdot xo + 40 \cdot xb + 120 \cdot xp \geq 15$$

$$14 \cdot xo + 14 \cdot xb + 10 \cdot xp \geq 10$$

$$250 \cdot xo + 8 \cdot xb + 44 \cdot xp \geq 75$$

l'un de ceux qui minimisent $4 \cdot xo + 2 \cdot xb + 4.5 \cdot xp$.

Le problème. On se donne n éléments composés $\{comp_1, \ldots, comp_n\}$ et m éléments de base $\{base_1, \ldots, base_m\}$. La composition d'un élément $comp_i$ est donnée par $\{a_{j,i}\}_{1 \leq j \leq m}$ et son coût est donné par c_i. La quantité d'élément de base $base_j$ à se procurer est b_j. Le problème s'énonce ainsi. Trouver parmi les tuples $(x_1, \ldots, x_n)$ qui vérifient :

$$\forall 1 \leq j \leq m, \sum_{i=1}^{n} a_{j,i} \cdot x_i \geq b_j$$

l'un de ceux qui minimisent $\sum_{i=1}^{n} c_i \cdot x_i$.

Recherche de seuils

Une instance du problème. Supposons que le département de ressources humaines d'une entreprise maintienne un fichier des employés (déclaré à la CNIL). Ce fichier contient entre autres le salaire de chaque employé. Le département souhaite établir quel est le seuil correspondant au 10% des employés les mieux payés. Si cette entreprise a 500 employés, cela revient à trouver quel est le 50ème plus gros salaire.

Le problème. Soit T un tableau de valeurs numériques (avec répétitions éventuelles) et k un entier inférieur ou égal à la taille de T. Le problème consiste à déterminer la $k$ième plus grande valeur du tableau et un indice du tableau contenant cette valeur.

Déduction automatique

Une instance du problème. Supposons connues les affirmations suivantes :

- *Tous les chats sont blancs ou noirs.*
- *Platon est un chat.*
- *Platon n'est pas noir.*
- *Aristote est blanc.*

On souhaiterait savoir si chacune des deux phrases suivantes est une conséquence des affirmations précédentes.

- *Platon est blanc.*
- *Aristote est un chat.*

Le problème. Etant donnée une logique (*e.g.*, logique propositionnelle ou logique du premier ordre), un ensemble de formules $\{\varphi_i\}_{1 \leq i \leq n}$ de cette logique et une formule φ, le problème consiste à déterminer si φ se déduit (à l'aide des axiomes et des règles de déduction de cette logique) de $\{\varphi_i\}_{1 \leq i \leq n}$.

Terminaison d'un programme

Une instance du problème. Supposons que nous ayons écrit un programme correspondant à l'algorithme 9 et qu'avant de l'exécuter, nous souhaitions savoir si son exécution se terminera.

Le problème. Etant donné le texte d'un programme écrit par exemple en JAVA, le problème consiste à déterminer si ce programme se termine. Plusieurs variantes sont possibles. Ainsi si le programme comprend une entrée, le problème

Algorithme 9: Se termine-t-il ?

```
x ← 1
while x ≠ 0 do
    if x%3 = 0 then
        x ← x + 5
    else
        x ← x − 2
    end
end
```

pourrait consister à déterminer si le programme se termine pour toutes les entrées possibles.

2.1.2 Taille d'une instance

L'algorithmique consiste à résoudre des problèmes de manière efficace. Il est donc nécessaire de définir une mesure de cette efficacité. Du point de vue de l'utilisateur, un algorithme est efficace si :

1. il met peu de temps à s'exécuter ;
2. il occupe peu de place en mémoire principale.

Cependant ces mesures dépendent de la taille de l'instance du problème à traiter. Il convient donc de définir la taille d'une instance. Plusieurs définitions sont possibles dans la mesure où une même instance peut s'énoncer de différentes manières. En toute rigueur, l'efficacité d'un algorithme devrait prendre en compte non pas l'instance mais sa représentation fournie en entrée de l'algorithme. Cependant la plupart des représentations *raisonnables* d'une instance conduisent à des tailles similaires. Plutôt que de formaliser cette notion, nous la préciserons pour chaque problème traité.

Si on prend comme unité de mesure le bit, nous commençons par faire quelques hypothèses.

- Le nombre de bits nécessaires pour représenter un caractère est constant (en réalité, il dépend de la taille de l'alphabet mais celle-ci ne change pas de manière significative). On le notera B_c.
- Le nombre de bits nécessaires pour représenter un entier est constant. Cette hypothèse n'est valable que si, d'une part on connaît *a priori* une borne supérieure de la taille d'un entier intervenant dans une instance et si, d'autre part les opérations effectuées sur les entiers par l'algorithme ne conduisent pas à un dépassement de cette borne [1]. On le notera B_e. Lorsqu'on ne peut appliquer cette hypothèse, la taille des entiers du problème constitue alors un paramètre du problème.
- Dans la plupart des systèmes d'information, les informations stockées dans des fichiers sont accessibles *via* des *identifiants* qui peuvent être des valeurs numériques (*e.g.* le n° de sécurité sociale) ou des chaînes de caractères (*e.g.* la concaténation du nom et du prénom). On supposera

1. pour certains problèmes, comme ceux liés à la cryptographie, cette hypothèse doit être levée.

aussi que le nombre de bits nécessaires pour représenter un identifiant est constant et on le notera B_i.

Illustrons maintenant la taille d'une instance à l'aide des exemples précédents.

Planarité d'un graphe. Il suffit de représenter les liaisons, *i.e.* des paires d'identifiants. On obtient donc $2m \cdot B_i$. Cette représentation appelle deux remarques.

D'un point de vue technique, le programme implémentant l'algorithme doit savoir où se termine la représentation : soit par la valeur m précédant la liste des paires soit par un identifiant spécial (différent des identifiants possibles) qui suit la liste. Dans les deux cas, on a ajouté un nombre constant de bits.

D'un point de vue conceptuel, un lieu qui n'apparaît dans aucune liaison, est absent de la représentation. Vis à vis du problème traité, cela n'est pas significatif car si le plan a pu être établi, il suffit d'ajouter ces lieux hors de l'espace occupé par le plan.

Programmation linéaire. On précise d'abord m et n, puis les éléments $a_{i,j}$ (ligne par ligne ou colonne par colonne), les élements b_j et finalement les éléments c_i. Ces nombres ne sont pas nécessairement des entiers. En faisant l'hypothèse que ce sont des rationnels on peut les représenter sous forme de fractions d'entier, *i.e.* par deux entiers [2]. On obtient comme taille du problème $(2 + 2m + 2n + 2m \cdot n) \cdot B_e$.

Recherche de seuils. On précise d'abord n la taille du tableau et k le seuil puis les cellules du tableau, ce qui nous donne (en supposant que les salaires soient des entiers) $(2 + n) \cdot B_e$.

Déduction automatique. On indique le nombre d'hypothèses puis les hypothèses suivi de la conclusion. *A priori*, les formules peuvent être de taille quelconque. En notant $|\varphi|$ le nombre de caractères d'une formule ϕ, la taille du problème est alors $B_e + (\sum_{i=1}^{n} |\varphi_i| + |\varphi|) \cdot B_c$.

Terminaison d'un programme. La taille de l'instance est $n \cdot B_c$ où n est le nombre de caractères du texte du programme.

Remarque importante. Nous nous intéressons au comportement des algorithmes sur des instances de grande taille. De manière encore plus précise, nous souhaitons estimer la nature de la variation du temps d'exécution de l'algorithme (ou de l'espace occupé) lorsque la taille de l'instance augmente. Aussi il est raisonnable de :

— conserver le terme prédominant de la taille d'une instance ;

— dans ce terme, remplacer les constantes multiplicatives par 1.

Ainsi pour le cas de la programmation linéaire, le terme prédominant est $2m \cdot n \cdot B_e$. Par conséquent, en oubliant les constantes, on prendra pour la taille d'une instance $m \cdot n$.

2. Ceci ne préjuge pas de la représentation utilisée par le programme.

2.2 Complexité d'un algorithme

2.2.1 Temps d'exécution des instructions

On considèrera qu'une instruction d'affectation est exécutée en une unité de temps à condition que l'expression à évaluer ne comporte pas d'appel de fonctions et que la variable à affecter soit d'un type élémentaire. Cette hypothèse n'est plus valable dans le cas d'opérations arithmétiques où la taille des nombres est un paramètre du problème. Il faut alors considérer les opérations bit à bit comme des opérations en temps constant et en déduire en fonction de la nature de l'opération le temps d'exécution de cette opération.

Dans le cas d'un appel de fonction, il faut ajouter le temps d'exécution des appels de fonction. Dans le cas d'un type complexe, il faut tenir compte du type. Ainsi si on affecte un tableau de taille t, le temps d'exécution sera t.

On considère aussi que l'évaluation d'un test intervenant dans un constructeur s'effectue aussi en une unité de temps avec les mêmes restrictions que pour l'affectation.

Enfin le temps d'exécution du renvoi du résultat d'une fonction est aussi une unité de temps sans préjuger du temps de son évaluation dans l'expression appelante.

2.2.2 Espace occupé par un programme

Pour la représentation des données, on adopte les mêmes règles que celles appliquées à la représentation d'une instance de problème.

L'espace du programme est la somme de trois composants :

— l'espace occupé par les données statiques : il s'agit d'une simple addition.

— l'espace occupé par le tas (données allouées et désallouées explicitement). Il s'agit alors d'estimer la taille maximum du tas lors de l'exécution du programme.

— l'espace occupé par la pile (données allouées et désallouées implicitement lors de l'appel ou du retour de fonctions). Il s'agit alors d'estimer pour chaque profondeur maximale d'appels, la taille cumulée des données locales des appels en cours d'exécution.

2.2.3 Définitions

Dans un premier temps, nous nous limitons à la complexité temporelle de l'algorithme. Nous recherchons une garantie de performances ; aussi nous nous intéressons au pire cas d'exécution d'un programme.

Soit $\mathcal{I}$ une instance du problème traité par l'algorithme $\mathcal{A}$, notons $size(\mathcal{I})$ la taille de $\mathcal{I}$ et $Time(\mathcal{A}, \mathcal{I})$ le temps d'exécution de $\mathcal{A}$ lorsqu'il s'applique à $\mathcal{I}$.

Etant donnée une taille maximale n de problème, le pire cas d'exécution de $\mathcal{A}$ sur les instances de taille inférieure ou égale à n, noté $Time(\mathcal{A})(n)$ est défini par $Time(\mathcal{A})(n) \equiv \max(Time(\mathcal{A}, \mathcal{I}) \mid size(\mathcal{I}) \leq n)$.

Autrement dit, $Time(\mathcal{A})$ est une fonction croissante de $\mathbb{N}$ dans $\mathbb{N}$. Ce qui nous intéresse, c'est l'efficacité de $\mathcal{A}$ sur les problèmes de grande taille, c'est à dire le comportement asymptotique de $Time(\mathcal{A})(n)$ lorsque n tend vers ∞. A cette fin, nous introduisons des notations asymptotiques.

Définition 2 *Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Alors :*

— $O(g) = \{f \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 \in N, f(n) \leq c \cdot g(n)\}$
On note par abus de langage $f = O(g)$ ssi $f(n) \in O(g)$;

— $\Omega(g) = \{f \mid \exists c \in \mathbb{N}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0 \in N, f(n) \geq c \cdot g(n)\}$
On note par abus de langage $f = \Omega(g)$ ssi $f \in \Omega(g)$;

— *On note $f = \Theta(g)$ ssi $f(n) = O(g)$ et $f(n) = \Omega(g)$.*

La constante c de la définition précédente tient compte du fait que si on exécute un programme sur une machine dont le processeur est c fois « plus rapide », alors le temps d'exécution sera divisé par c.

On dira que la complexité d'un algorithme est en $O(g)$ (resp. $\Omega(g)$, $\Theta(g)$) si $Time(\mathcal{A}) = O(g)$ (resp. $Time(\mathcal{A}) = \Omega(g)$, $Time(\mathcal{A}) = \Theta(g)$). Nous établirons que la complexité de la plupart des algorithmes que nous verrons en cours, est dans $O(n^k)$ ou dans $O(n^k \cdot \log^{k'}(n))$.

Notations. Lorsque l'algorithme $\mathcal{A}$ sera déterminé sans ambiguïté, nous désignerons plus simplement la fonction $Time(\mathcal{A})$ par $Time$.

2.2.4 Illustration

Algorithme 10: Fusion de deux tableaux triés.

`Fusion`(T_1, n_1, T_2, n_2) : tableau d'entiers

Input : $\forall i \in \{1, 2\}$ T_i un tableau trié, n_i sa dimension
Output : La fusion triée des deux tableaux

Data : T tableau de dimension $n_1 + n_2$
Data : i, i_1, i_2 entiers

```
i1 ← 1 ; i2 ← 1
for i ← 1 to n1 + n2 do
    if i1 > n1 then
        T[i] ← T2[i2]
        i2 ← i2 + 1
    else if i2 > n2 then
        T[i] ← T1[i1]
        i1 ← i1 + 1
    else if T2[i2] ≤ T1[i1] then
        T[i] ← T2[i2]
        i2 ← i2 + 1
    else
        T[i] ← T1[i1]
        i1 ← i1 + 1
    end
end
return T
```

Fusion de tableaux triés Un tableau T de valeurs numériques (ou plus généralement de valeurs prises dans un domaine totalement ordonné) est dit *trié* ssi $\forall i < j$ $T[i] \leq T[j]$.

La fusion de deux tableaux T_1 et T_2 triés consiste à produire un tableau T trié composé des valeurs des deux tableaux en tenant compte des répétitions. L'algorithme 10 résout ce problème. Indiquons son principe :

— Il maintient un indice i_k par tableau T_k pointant sur la prochaine valeur à insérer dans T ou au delà du plus grand indice si toutes les valeurs ont été insérées. L'indice de la boucle i sert d'indice d'insertion dans le tableau T puisqu'à chaque tour de boucle, on insère un élément.

— Si les deux indices pointent encore dans leur tableau respectif, on insère la valeur pointée la plus petite et on incrémente l'indice correspondant. Sinon, on insère la seule valeur disponible en incrémentant également l'indice.

Calculons sa complexité. La taille de la représentation de deux tableaux de taille n_1 et n_2 est $(n_1 + n_2) \cdot B_e$. Comme nous en avons déjà discuté, les calculs de complexité se font à une constante près. Aussi on considérera que la taille de l'entrée est $n \equiv n_1 + n_2$. L'algorithme exécute deux instructions élémentaires puis n tours de boucle puis renvoie la valeur. Au cours d'un tour, il exécute au moins 4 instructions (en incluant le test d'entrée dans le tour) et au plus 6 instructions. Donc $3 + 4n \leq Time(n) \leq 3 + 6n$. Ce qui signifie que $Time(n) = \Theta(n)$.

Notons que le résultat est aussi vrai pour l'algorithme 11 (une légère variante de l'algorithme précédent) mais qu'il est un peu plus difficile à prouver.

Tri de tableaux par fusion. Le tri d'un tableau T consiste à produire un tableau T' avec les mêmes valeurs (et les mêmes répétitions) que T mais triées par ordre croissant. Pour ce faire, l'algorithme 1 procède ainsi :

1. Il partage le tableau en deux sous-tableaux (de dimension approximativement égales) qu'il trie (en s'appelant récursivement).
2. Puis il fusionne les deux tableaux à l'aide de l'algorithme 10.

La correction de cet algorithme est évidente. Intéressons-nous à sa complexité et notons n la taille du tableau à trier qu'on condidère comme la taille de l'entrée. Notons d'abord que $Time(1) = 2$ et que :

$$\forall n > 1, Time(n) \leq 1 + n + c \cdot n + Time(\lfloor n/2 \rfloor) + Time(\lceil n/2 \rceil)$$

avec c la constante de l'algorithme précédent.

Posons $c' \equiv c + 2$ et montrons d'abord par récurrence que :
$\forall k \geq 1\ Time(2^k) \leq c' \cdot k \cdot 2^k \cdot Time(1)$.
Nous laissons le soin au lecteur de le vérifier pour $k = 1$.

Appliquons l'hypothèse de récurrence :

$$Time(2^{k+1}) \leq c' \cdot 2^{k+1} + 2 \cdot Time(2^k)$$

$$\leq (c' \cdot 2^{k+1} + 2c' \cdot k \cdot 2^k) \cdot Time(1) = (c' \cdot (k+1) \cdot 2^{k+1}) \cdot Time(1)$$

Soit maintenant n quelconque et l'unique k tel que $n \leq 2^k < 2n$.

$$Time(n) \leq Time(2^k) \leq (c' \cdot k \cdot 2^k) \cdot Time(1)$$

$$\leq (c' \cdot \log_2(2n) \cdot 2n) \cdot Time(1) = (2c' \cdot (\log_2(n) + 1) \cdot n) \cdot Time(1)$$

D'où $Time(n) = O(n \log_2(n))$. En réalité, on a plus précisément $Time(n) = \Theta(n \log_2(n))$ *(faites-en la preuve).*

Algorithme 11: Autre fusion de deux tableaux triés.

`Fusion`(T_1, n_1, T_2, n_2) : tableau d'entiers

Input : $\forall i \in \{1, 2\}$ T_i un tableau trié, n_i sa dimension
Output : La fusion triée des deux tableaux

Data : T tableau de dimension $n_1 + n_2$
Data : i, i_1, i_2 entiers

$i_1 \leftarrow 1$; $i_2 \leftarrow 1$; $i \leftarrow 1$
while $i_1 \leq n_1$ **and** $i_2 \leq n_2$ **do**
    if $T_2[i_2] \leq T_1[i_1]$ **then**
        $T[i] \leftarrow T_2[i_2]$
        $i_2 \leftarrow i_2 + 1$
    else
        $T[i] \leftarrow T_1[i_1]$
        $i_1 \leftarrow i_1 + 1$
    end
    $i \leftarrow i + 1$
end
while $i_1 \leq n_1$ **do**
    $T[i] \leftarrow T_1[i_1]$
    $i_1 \leftarrow i_1 + 1$
    $i \leftarrow i + 1$
end
while $i_2 \leq n_2$ **do**
    $T[i] \leftarrow T_2[i_2]$
    $i_2 \leftarrow i_2 + 1$
    $i \leftarrow i + 1$
end
return T

2.3 Equations de récurrence

Nous fournissons un certain nombre de résultats généraux qui couvrent la plupart des complexités qu'on rencontre pour des algorithmes définis de manière récursive.

Proposition 4 *Soit $t : \mathbb{N} \mapsto \mathbb{R}$, une fonction croissante à partir d'un certain rang telle qu'il existe des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $k \geq 0$, $a > 0$, $c > 0$ et $d > 0$ pour lesquels :*

$$t(n_0) = d$$

$$t(n) = at(n/b) + cn^k \text{ pour } n = n_0 b^p \text{ avec } p \geq 1$$

Alors :

$$t(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \Leftrightarrow \log_b(a) < k \\ \Theta(n^k \log_b(n)) & \text{si } a = b^k \Leftrightarrow \log_b(a) = k \\ \Theta(n^{\log_b(a)}) & \text{si } a > b^k \Leftrightarrow \log_b(a) > k \end{cases}$$

Preuve
Posons $n = n_0 b^p$. Alors par récurrence, on obtient :

$$t(n) = da^p + \sum_{i=0}^{p-1} ca^i (n/b^i)^k = da^p + cn^k \sum_{i=0}^{p-1} (a/b^k)^i$$

Or $da^p = d(b^{\log_b(a)})^p = d(b^p)^{\log_b(a)} = \Theta(n^{\log_b(a)})$. Posons $\gamma(n) = \sum_{i=0}^{p-1} (a/b^k)^i$. On a :

- $\gamma(n) \sim \frac{1}{1-a/b^k}$ si $a/b^k < 1$
- $\gamma(n) = p \sim \log_b(n)$ si $a/b^k = 1$
- $\gamma(n) \sim \alpha \frac{a^p}{b^{kp}}$ avec $\alpha = (a/b^k - 1)^{-1}$ si $a/b^k > 1$

Les deux premières formules de la proposition s'en déduisent immédiatement. Pour la troisième, on observe que : $cn^k \gamma(n) \sim \alpha c n_0^k a^p = \Theta(n^{\log_b(a)})$

Soit maintenant n assez grand et soit p tel que $n_0 b^p < n \leq n_0 b^{p+1}$. On a $t(n_0 b^p) \leq t(n) \leq t(n_0 b^{p+1})$. D'autre part pour les trois fonctions de complexité de la proposition (disons f), on a $f(bn) = \Theta(f(n))$. Par conséquent, pour n quelconque on a $t(n) = \Theta(f(n))$.

c.q.f.d. ◊◊◊

Cette proposition se généralise de manière évidente en substituant à n^k selon le cas $O(n^k), \Theta(n^k), \Omega(n^k)$. La condition supplémentaire du premier cas est une condition sur un comportement « similaire » à celui du polynôme n^k utilisé dans la preuve précédente.

Proposition 5 *Soit $t : \mathbb{N} \mapsto \mathbb{R}$, une fonction croissante à partir d'un certain rang telle qu'il existe une fonction $f : \mathbb{N} \mapsto \mathbb{R}$,des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $k \geq 0$, $a > 0$ et $d > 0$ pour lesquels :*

$$t(n_0) = d$$

$$t(n) = at(n/b) + f(n) \text{ pour } n = n_0 b^p \text{ avec } p > 1$$

Alors :

$$t(n) = \begin{cases} \Theta(f(n)) & si\ a < b^k \wedge f(n) = \Omega(n^k) \\ & \wedge\ af(n/b) \leq cf(n)\ \ avec\ 0 < c < 1 \\ \Theta(n^{\log_b(a)} \log_b(n)) & si\ f(n) = \Theta(n^{\log_b(a)}) \\ \Theta(n^{\log_b(a)}) & si\ a > b^k \wedge f(n) = O(n^k) \end{cases}$$

Nous complétons maintenant les propositions précédentes pour traiter des cas moins fréquents mais rencontrés de manière significative.

Proposition 6 *Soit $t : \mathbb{N} \mapsto \mathbb{R}$, une fonction croissante à partir d'un certain rang telle qu'il existe une fonction $f : \mathbb{N} \mapsto \mathbb{R}$, des entiers $n_0 \geq 1$, $b \geq 2$ et des réels $k \geq 0$, $a > 0$ et $d > 0$ pour lesquels :*

$$t(n_0) = d$$

$$t(n) = at(n/b) + f(n)\ \ pour\ n = n_0 b^p\ \ avec\ p > 1$$

Supposons de plus que $f(n) = cn^k(log_b(n))^q$ pour des réels $c > 0$, $k \geq 0$ et q. Alors :

$$t(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \wedge q = 0 \\ \Theta(n^k \log_b(n)^{1+q}) & si\ a = b^k \wedge q > -1 \\ \Theta(n^k \log_b(\log_b(n))) & si\ a = b^k \wedge q = -1 \\ \Theta(n^{\log_b(a)}) & si\ a = b^k \wedge q < -1 \\ \Theta(n^{\log_b(a)}) & si\ a > b^k \end{cases}$$

Preuve
Le premier cas a déjà été traité. Nous traitons maintenant les autres cas.

$$t(n) = da^p + \sum_{i=0}^{p-1} a^i f(n/b^i) = da^p + \sum_{i=0}^{p-1} a^i f(n_0 b^{p-i}) = da^p + \sum_{i=1}^{p} a^{p-i} f(n_0 b^i)$$

$$= da^p + c\sum_{i=1}^{p} a^{p-i}(n_0 b^i)^k(\log_b(n_0 b^i))^q = da^p + cn_0^k a^p \sum_{i=1}^{p}(b^k/a)^i(\log_b(n_0) + i)^q$$

Par conséquent,

$$t(n) = \Theta(n^{\log_b(a)}\gamma(n)) \text{ avec } \gamma(n) = \sum_{i=1}^{p} h^i(r+i)^q$$

avec $h = b^k/a$ et $r = \log_b(n_0)$. Si $h = 1$ alors

$$\gamma(n) = \begin{cases} \Theta(p^{1+q}) = \Theta(\log_b(n)^{1+q}) & \text{si } q > -1 \\ \Theta(\log_b(p)) = \Theta(\log_b(\log_b(n))) & \text{si } q = -1 \\ \Theta(1) & \text{si } q < -1 \end{cases}$$

Enfin, si $h < -1$ alors $\gamma(n) = \Theta(1)$.

c.q.f.d. ◊◊◊

Cette dernière proposition couvre les cas d'appels récursifs dont les tailles des entrées sont différentes mais dont la somme est inférieure à une fraction fixe de l'entrée de l'appelant.

Proposition 7 *Soient* $\alpha_1, \ldots, \alpha_k : \mathbb{N} \mapsto \mathbb{N}$, *des fonctions telles qu'il existe un nombre réel* $0 < K < 1$ *et* n_0 *un entier avec :*

$$\forall n > n_0 \ \alpha_1(n) + \ldots + \alpha_k(n) \leq Kn$$

Si une fonction $t : \mathbb{N} \mapsto \mathbb{R}$ *vérifie avec* $b > 0$ *un réel :*

$$\forall n \leq n_0 \ t(n) \leq a_0$$

$$\forall n > n_0 \ t(n) \leq t(\alpha_1(n)) + \ldots + t(\alpha_k(n)) + bn$$

Alors : $t(n) = O(n)$

Preuve
Démontrons par récurrence que $t(n) \leq Dn + a_0$ avec $D = (b + (k-1)a_0)/(1-K)$. Cette inégalité est vérifiée pour $n \leq n_0$. Si $n > n_0$ alors :

$$t(n) \leq t(\alpha_1(n)) + \ldots + t(\alpha_k(n)) + bn \leq ka_0 + D(\alpha_1(n) + \ldots + \alpha_k(n)) + bn \leq ka_0 + (DK + b)n$$

Or $ka_0 + (DK + b)n \leq Dn + a_0$ ssi $(k-1)a_0 + bn \leq D(1-K)n$, inégalité vérifiée d'après notre définition de D.

c.q.f.d. ◊◊◊

2.4 Complexité en moyenne

La complexité en moyenne d'un algorithme suppose un choix aléatoire et une distribution de probabilité. Cette distribution peut porter sur l'entrée de l'algorithme ou sur le résultat d'un tirage aléatoire effectué par un algorithme probabiliste. Généralement la variable aléatoire dont on calcule l'espérance est le temps d'exécution. Il se comprend ainsi :

- Dans le cas d'un algorithme déterministe, il s'agit d'un temps d'exécution moyen sur une donnée tirée aléatoirement.
- Dans le cas d'un algorithme probabiliste, il s'agit d'un temps d'exécution moyen selon les tirages effectués par l'algorithme à partir d'une donnée arbitraire.

On note que la deuxième mesure est plus crédible car l'hypothèse probabiliste sur les données n'est peut-être pas vérifiée expérimentalement.

Un autre type d'analyse probabiliste consiste en une garantie probabiliste de complexité : avec une grande probabilité (tendant vers 1 quand la taille de l'entrée grandit) le temps d'exécution est borné par une fonction de la taille de l'entrée. Ce type d'analyse complète de manière significative l'analyse en moyenne.

Nous illustrons ces différents concepts sur un problème de gestion de données. Il nous faut définir une structure de données qui permette d'ajouter, de rechercher et de supprimer des enregistrements. Chaque enregistrement est déterminé de manière unique par un identifiant (appelé aussi clé).

2.4.1 Gestion d'un annuaire par un arbre binaire

Principe d'un arbre binaire

La recherche d'un enregistrement dans un tableau de n enregistrements où les enregistrements sont triés par identifiant s'effectue en $O(\log(n))$. Le principe des arbres binaires de recherche est de maintenir une structure dynamique « triée » accessible par son « milieu » d'où il est possible d'accèder soit à la « moitié » inférieure des enregistrements, soit à la « moitié » supérieure des enregistrements. De la même façon, ces « moitiés » sont aussi accessibles par leur « milieu ».

Ceci nous conduit naturellement à la structure suivante :
struct *enregistrement* {
id : identifiant
$info$: information
$suivG, suivD$: référence
}

L'enregistrement contient l'identifiant et les informations de l'enregistrement ainsi que deux références $suivG$ et $suivD$ sur des enregistrements. Le point clef est que tous les enregistrements accessibles par $suivG$ (resp. $suivD$) ont un identifiant plus petit (resp. plus grand) que l'identifiant de l'enregistrement.

La figure 2.2 présente un arbre binaire de recherche. L'axe horizontal sur lequel nous avons reporté à la verticale les identifiants montre l'intuition qui se cache derrière cette structure : les identifiants sont rangés « de gauche à droite » par ordre croissant.

Expliquons comment s'effectue la recherche d'un identifiant dans un arbre binaire de recherche. Celle-ci se déroule de manière similaire à la recherche dans un tableau trié. Si l'arbre est vide alors la recherche est infructueuse. Sinon, on compare l'identifiant de l'enregistrement à l'identifiant de la racine. Si les deux identifiants sont égaux alors on renvoie une référence sur l'enregistrement de la racine. Si l'identifiant recherché est plus petit (resp. plus grand) on se déplace en suivant la référence vers le *sous-arbre* gauche (droit) et on procède ainsi itérativement.

Le cas de l'ajout est traité de manière analogue. Il s'agit de savoir où l'enregistrement doit être inséré. L'algorithme utilise deux variables p et $newp$, p est le *père* de $newp$, excepté initialement lorsque $newp$ référence la racine (qui n'a pas de père). Dans ce cas, p est égal à `NULL`. On effectue le même parcours de l'arbre que pour une recherche et on n'insère l'enregistrement que si la recherche a été infructueuse. Dans ce cas, $newp$ est égal à `NULL` et p référence le futur père du nouvel enregistrement.

Il convient d'examiner trois cas. p est `NULL` ; par conséquent, l'arbre est vide et l'enregistrement est inséré à la *racine* de l'arbre. L'identifiant de p est plus grand (resp. plus petit) que l'identifiant de l'enregistrement à insérer ; par conséquent, l'enregistrement est inséré comme *fils gauche* (resp. *fils droit*) de l'enregistrement référencé par p. Puisque la recherche a été infructueuse, on est assuré que le sous-arbre gauche (resp. droit) de l'enregistrement référencé par p est vide. La recherche et l'ajout sont décrits par l'algorithme 12.

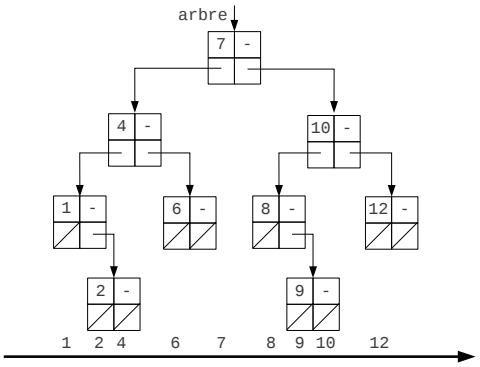

FIGURE 2.2: Un arbre binaire de recherche

La structure décrite est appelée arbre car en retournant la figure représentant cette structure, elle présente une analogie avec un arbre. La racine se trouve au niveau le plus bas et le tronc se subdivise (éventuellement) pour former des sous-arbres. Les enregistrements qui n'ont ni sous-arbre gauche, ni sous arbre droit sont appelés des *feuilles*. Les enregistrements sont aussi appelés *noeuds* car ils sont à l'origine des embranchements.

Si la figure n'est pas retournée alors elle ressemble à un arbre généalogique (sexiste) où seuls les hommes et les fils (au plus deux) sont présentés. Chaque enregistrement sauf la racine a un père et chaque enregistrement a au plus deux fils. Notons qu'être fils gauche ou droit n'est pas équivalent.

Affichage trié d'un arbre binaire

Comme nous l'avons déjà indiqué, un arbre est implicitement trié. Nous le vérifions avec l'algorithme 13 qui affiche les enregistrements dans l'ordre croissant des identifiants. L'algorithme procède ainsi : il affiche par un appel récursif les enregistrements du sous-arbre gauche, puis la racine et enfin, par un deuxième appel récursif, les enregistrements du sous-arbre droit.

Analysons sa complexité. Pour chaque enregistrement, l'algorithme exécute 5 instructions auquel on ajoute (éventuellement) les 2 instructions lors d'un appel infructueux pour un sous arbre vide, soit dans le pire des cas 9 instructions par enregistrement (et dans le meilleur des cas 5 instructions). Par conséquent, si $n > 0$ est le nombre d'enregistrements alors l'algorithme exécute au plus $9n$ instructions (et au moins $5n$ instructions). Sa complexité est en donc en $\Theta(n)$ comme l'affichage (trié) d'un tableau trié.

Suppression dans un arbre binaire

La suppression d'un enregistrement d'un arbre binaire est nettement plus difficile que dans le cas des structures précédentes. Afin de mettre en oeuvre cette suppression, nous décomposons le problème en quatre cas, du plus simple au plus complexe.

Algorithme 12: Gestion d'un annuaire par un arbre

Type : **struct** *enregistrement*
{ id : identifiant ; $info$: information ; $suivG, suivD$: référence }
Data : *arbre* une référence pointant sur la racine de l'annuaire, initialisée à `NULL`

`Chercher`($unid$) : référence d'un enregistrement
Input : $unid$ l'identifiant d'un enregistrement
Output : une référence de cet enregistrement ou `NULL` s'il est absent de l'annuaire
Data : p une variable référence

$p \leftarrow arbre$
while $p \neq$ `NULL` **do**
  if $p\rightarrow id = unid$ **then return** p
  else if $p\rightarrow id > unid$ **then** $p \leftarrow p\rightarrow suivG$
  else $p \leftarrow p\rightarrow suivD$
end
return `NULL`

`Ajouter`($unid, uneinfo$) : booléen
Input : $unid$ l'identifiant d'un enregistrement à ajouter, $uneinfo$ ses informations
Output : un booléen indiquant si l'ajout s'est déroulé correctement
Data : $p, newp$ deux variables références
$newp \leftarrow arbre$; $p \leftarrow$ `NULL`
while $newp \neq$ `NULL` **do**
  $p \leftarrow newp$
  if $p\rightarrow id = unid$ **then return false**
  else if $p\rightarrow id > unid$ **then** $newp \leftarrow p\rightarrow suivG$
  else $newp \leftarrow p\rightarrow suivD$
end
$newp \leftarrow$ **new**($enregistrement$)
$newp\rightarrow id \leftarrow unid$; $newp\rightarrow info \leftarrow uneinfo$
$newp\rightarrow suivG \leftarrow$ `NULL` ; $newp\rightarrow suivD \leftarrow$ `NULL`
if $p =$ `NULL` **then** $arbre \leftarrow newp$
else if $p\rightarrow id > unid$ **then** $p\rightarrow suivG \leftarrow newp$
else $p\rightarrow suivD \leftarrow newp$
return true

Algorithme 13: Affichage d'un arbre

`Afficher`(p)
Input : p une référence d'arbre
if $p \neq$ `NULL` **then**
  `Afficher` ($p\rightarrow suivG$)
  print $p\rightarrow id, p\rightarrow info$
  `Afficher` ($p\rightarrow suivD$)
end
return

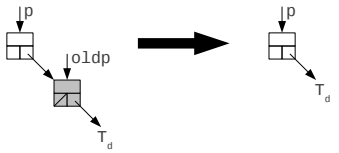

FIGURE 2.3: Premier cas de suppression

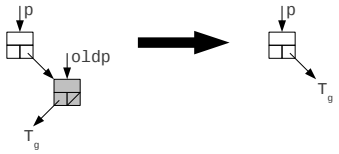

FIGURE 2.4: Deuxième cas de suppression

Cas n° 1 (voir la figure 2.3) L'enregistrement (désigné par *oldp* dans l'algorithme 14) a un sous-arbre gauche vide. Il suffit alors de rattacher le sous-arbre droit au père de *oldp*, appelé *p* dans l'algorithme 14, en lieu et place de celui-ci.

Cas n° 2 (voir la figure 2.4) Ce deuxième cas est très similaire au précédent. L'enregistrement (désigné par *oldp* dans l'algorithme 14) a un sous-arbre droit vide. Il suffit alors de rattacher le sous-arbre gauche au père de *oldp*, appelé *p* dans l'algorithme 14, en lieu et place de celui-ci.

Les deux cas restants nécessitent l'introduction d'une fonction auxiliaire. Cette fonction, appelée `PluspetitD` dans l'algorithme 14, prend en entrée un enregistrement dont le sous-arbre droit est non vide et renvoie le père du plus petit enregistrement de ce sous-arbre. Expliquons son fonctionnement. L'enregistrement le plus petit d'un arbre se trouve en suivant les fils gauches jusqu'à ce que l'un d'eux soit `NULL`. C'est ce que fait cette fonction en conservant dans *p* le père du plus petit enregistrement désigné par *newp*.

Si on n'entre pas dans les deux cas précédents, l'enregistrement à supprimer a ses deux sous-arbres non vides. On appelle alors la fonction `PluspetitD` avec pour entrée l'enregistrement à supprimer.

Cas n° 3 (voir la figure 2.5) Si le résultat de l'appel à la fonction (désigné par *pere* dans l'algorithme 14) est l'enregistrement à supprimer (*oldp*), cela signifie que son fils droit a un sous-arbre gauche vide. Dans ce cas il suffit de rattacher le sous-arbre gauche de *oldp* à son fils droit (appelé *newp*) puis de rattacher *newp* à *p* le père de l'enregistrement à supprimer, en lieu et place de celui-ci.

Cas n° 4 (voir la figure 2.6) Le dernier cas nécessite deux opérations conjointes. *newp*, étant le petit élément du sous-arbre droit de *oldp*, va être détaché de son père (*pere*) pour être attaché à *p* en lieu et place de *oldp* en « héritant » des sous-arbres de *oldp*. Il ne faut cependant pas oublier que *newp* a peut-être un sous-arbre droit non vide (mais un sous-arbre gauche vide). Aussi ce sous-arbre

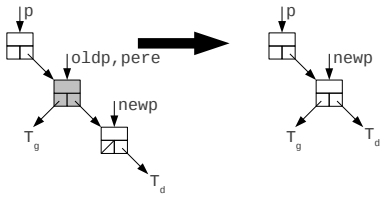

FIGURE 2.5: Troisième cas de suppression

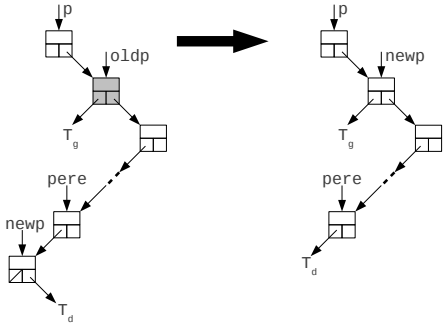

FIGURE 2.6: Quatrième cas de suppression

droit sera rattaché à son père en lieu et place de *newp*.

Quelque soit le cas étudié précédemment, au moment du rattachement de *newp* à *p*, il y a trois possibilités :

- *p* est `NULL`, ce qui signifie que l'enregistrement supprimé était la racine. *newp* devient la nouvelle racine.
- *p* référence un enregistrement plus petit que celui référencé par *newp*. *newp* est rattaché à « droite » de *p* (cas représenté dans les figures).
- *p* référence un enregistrement plus grand que celui référencé par *newp*. *newp* est rattaché à « gauche » de *p*.

A titre d'exemple, l'arbre résultant de la suppression de l'enregistrement d'identifiant 7 dans l'arbre de la figure 2.2 est présenté dans la figure 2.7.

Complexité des opérations

Afin de mesurer plus précisément la complexité des opérations sur les arbres binaires de recherche, nous introduisons une quantité entière relative à un arbre et à ses noeuds, la *hauteur*.

Définition 3 *La hauteur d'un noeud dans un arbre est définie inductivement par :*

Algorithme 14: Suppression dans un arbre

`Pluspetitd`($uneref$) : référence
Input : $uneref$ une référence d'enregistrement qui a un suivant droit
Output : le « père » du plus petit enregistrement à droite de $uneref$
Data : $p, newp$ deux variables références
$p \leftarrow uneref$; $newp \leftarrow p\rightarrow suivD$
while $newp\rightarrow suivG \neq$ `NULL` **do** $p \leftarrow newp$; $newp \leftarrow p\rightarrow suivG$
return p

`Supprimer`($unid$) : booléen
Input : $unid$ l'identifiant d'un enregistrement à supprimer
Output : un booléen indiquant le statut de l'opération
Data : $oldp, p, newp, pere$ variables références
$newp \leftarrow arbre$; $p \leftarrow$ `NULL`
while $newp \neq$ `NULL` **do**
  if $newp\rightarrow id = unid$ **then**
    $oldp \leftarrow newp$
    if $oldp\rightarrow suivG =$ `NULL` **then** $\boxed{1}$
      $newp \leftarrow oldp\rightarrow suivD$
    else if $oldp\rightarrow suivD =$ `NULL` **then** $\boxed{2}$
      $newp \leftarrow oldp\rightarrow suivG$
    else
      $pere \leftarrow$ `Pluspetitd`($oldp$)
      if $pere = oldp$ **then** $\boxed{3}$
        $newp \leftarrow pere\rightarrow suivD$
        $newp\rightarrow suivG \leftarrow oldp\rightarrow suivG$
      else $\boxed{4}$
        $newp \leftarrow pere\rightarrow suivG$
        $pere\rightarrow suivG \leftarrow newp\rightarrow suivD$
        $newp\rightarrow suivG \leftarrow oldp\rightarrow suivG$
        $newp\rightarrow suivD \leftarrow oldp\rightarrow suivD$
      end
    end
    if $p =$ `NULL` **then** $arbre \leftarrow newp$
    else if $p\rightarrow id > unid$ **then** $p\rightarrow suivG \leftarrow newp$
    else $p\rightarrow suivD \leftarrow newp$
    delete($oldp$) ; **return true**
  end
  $p \leftarrow newp$
  if $p\rightarrow id > unid$ **then** $newp \leftarrow p\rightarrow suivG$
  else $newp \leftarrow p\rightarrow suivD$
end
return false

— *La hauteur de la racine est 1.*
— *La hauteur d'un autre noeud est égal à la hauteur de ce noeud dans le sous-arbre de la racine où il est présent incrémentée de 1.*

La hauteur d'un arbre (notée h) est définie par :
— *La hauteur d'un arbre vide est 0.*
— *La hauteur d'un arbre non vide est le maximum de la hauteur de ses noeuds (*i.e.*, le maximum de la hauteur de ses deux sous-arbres incrémenté de 1).*

En fait, il existe une variante de cette définition avec la hauteur de la racine égale à 0. Ces deux définitions sont éqivalentes du point de vue de l'étude de la complexité et on choisit celle qui se prête le plus aux manipulations algébriques qu'on a en vue.

Analyse au pire des cas
Analysons la recherche d'un arbre dans le cas d'une recherche infructueuse (le pire cas). Elle exécute au plus 4 opérations relatives à un enregistrement avant de continuer la recherche dans un sous-arbre. Autrement dit sa complexité est inférieure ou égale à $4h + 2$ (en comptant la première et la dernière instruction) soit une complexité en $O(h)$. La complexité de l'ajout est similaire à la recherche au plus $5h + 10$ soit aussi une complexité en $O(h)$.

La suppression qui est conceptuellement plus difficile est aussi en $O(h)$ car on « descend » jusqu'à l'élément à supprimer puis (au pire des cas) le long de son sous-arbre droit.

Supposons que l'arbre comporte n enregistrements, la plus grande valeur possible pour h est alors n. En effet, en imaginant une insertion d'enregistrements par ordre croissant, l'arbre est analogue à une liste chaînée par le champ $suivD$.

Analyse en moyenne
Pour apprécier l'intérêt des arbres, il faut procéder à une analyse probabiliste de la hauteur des noeuds vues comme des variables aléatoires. L'hypothèse probabiliste que nous faisons est la suivante.

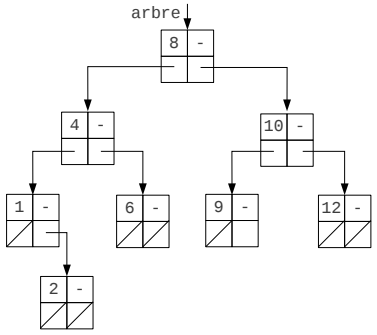

FIGURE 2.7: Une suppression dans un arbre (cas n°4)

L'arbre est obtenu par ajout de n enregistrements où l'enregistrement à ajouter est choisi de manière équiprobable parmi les enregistrements restants.

Dans la suite nous notons les identifiants des n enregistrements par $id_1 < \ldots < id_n$. Préalablement à nos résultats, nous caractérisons le fait que deux enregistrements seront comparés lors de la construction de l'arbre.

Lemme 2 *Soient $1 \leq i < j \leq n$, les enregistrements d'identifiants id_i et id_j seront comparés si et seulement si l'un des deux identifiants est le premier à être inséré dans l'annuaire parmi l'ensemble des identifiants compris entre id_i et id_j,* i.e. $ID(i,j) = \{id_i, id_{i+1}, \ldots, id_{j-1}, id_j\}$.

Preuve
Tant qu'aucun enregistrement d'identifiant appartenant à $ID(i,j)$ n'est inséré dans l'arbre, la recherche de tels identifiants conduit au même chemin puisque les tests conduisent aux mêmes résultats. Quand le premier enregistrement d'identifiant de $ID(i,j)$ est inséré, la branche qui y conduit sera ensuite parcourue par tous les autres enregistrements de $ID(i,j)$.

Par conséquent si cet enregistrement a pour identifiant id_i (resp. id_j), alors l'enregistrement d'identifiant id_j (resp. id_j) sera comparé à lui lors de son insertion.

Si à l'inverse cet enregistrement est différent de id_i et de id_j, alors id_i et id_j seront insérés respectivement dans son sous-arbre gauche et droit et ne seront pas comparés.

c.q.f.d. ◊◊◊

Nous introduisons maintenant quelques variables aléatoires nécessaires à notre raisonnement.

Définition 4 *Soient $1 \leq i < j \leq n$,*

- *$X_{i,j}$ désigne la variable aléatoire qui vaut 1 si id_i et id_j ont été comparés et 0 sinon (par conséquent $\mathbf{E}(X_{i,j}) = \mathbf{Pr}(X_{i,j} = 1)$).*
- *$h(i)$ désigne la variable aléatoire correspondant à la hauteur de l'enregistrement d'identifiant id_i*
- *$h_m = \frac{1}{n}\sum_{i=1}^{n} h(i)$ désigne la variable aléatoire correspondant à la hauteur moyenne d'un enregistrement dans un arbre aléatoire.*
- *$h_{max} = \max(\{h(i) \mid 1 \leq i \leq n\})$ désigne la variable aléatoire correspondant à la hauteur d'un arbre aléatoire.*

h_m s'exprime en fonction des $X_{i,j}$ ainsi que l'établit le lemme suivant.

Lemme 3 $h_m = 1 + \frac{1}{n}\sum_{i<j} X_{i,j}$

Preuve
La hauteur d'un noeud est égale à 1 plus le nombre de comparaisons avec les autres noeuds lors de son insertion. Par conséquent la somme des hauteurs des noeuds est égale à $n + \sum_{i<j} X_{i,j}$. Par conséquent, la hauteur moyenne d'un noeud $h_m = 1 + \frac{1}{n}\sum_{i<j} X_{i,j}$.

c.q.f.d. ◊◊◊

Avant d'entamer nos raisonnements probabilistes, nous effectuons un bref rappel d'analyse.

Définition 5 *Un fonction f de domaine inclus dans $\mathbb{R}$ et à valeurs dans $\mathbb{R}$ est convexe si :*
$\forall x, y, \forall 0 \leq p_x, p_y$ *t.q.* $p_x + p_y = 1, f(p_x x + p_y y) \leq p_x f(x) + p_y f(y)$

La notion duale de la convexité est la concavité avec l'inégalité inverse.

Lemme 4 *Soit f une fonction de domaine inclus dans $\mathbb{R}$ et à valeurs dans $\mathbb{R}$*
- *Si f est dérivable de dérivée croissante (respectivement décroissante), alors f est convexe (respectivement concave).*
- *Si f est convexe alors :*
  $\forall x_1, \ldots, x_n, \forall 0 \leq p_1, \ldots, p_n$ *t.q.* $\sum p_i = 1, f(\sum p_i x_i) \leq \sum p_i f(x_i)$.
- *Si f est convexe (resp. concave) et X une v.a. (ici prenant un nombre fini de valeurs) alors $f(\mathbf{E}(X)) \leq \mathbf{E}(f(X))$ (resp. $f(\mathbf{E}(X)) \geq \mathbf{E}(f(X))$).*

Preuve
Supposons $x < y$.
$(f(p_x x + p_y y) - f(x))/(p_x x + p_y y - x) = f'(a)$ pour un $a \in [x, p_x x + p_y y]$.
$(f(y) - f(p_x x + p_y y))/(y - p_x x - p_y y) = f'(b)$ pour un $b \in [p_x x + p_y y, y]$.
Puisque f' est croissante :
$(f(p_x x + p_y y) - f(x))/(p_x x + p_y y - x) \leq (f(y) - f(p_x x + p_y y))/(y - p_x x - p_y y)$
Soit $(f(p_x x + p_y y) - f(x))(y - p_x x - p_y y) \leq (f(y) - f(p_x x + p_y y))(p_x x + p_y y - x)$.
Après simplifications :
$f(p_x x + p_y y)y - f(x)(y - p_x x - (1 - p_x)y) \leq f(y)((1 - p_y)x + p_y y - x) + f(p_x x + p_y y)x$
$f(p_x x + p_y y)(y - x) \leq f(x)(p_x(y - x)) + f(y)(p_y(y - x))$
D'où le résultat en divisant par $y - x$.

L'inégalité se démontre par récurrence. Le cas $n = 2$ est la définition de la convexité. Notons $s = \sum_{i=1}^{n-1} p_i$.
$f(\sum_{i=1}^{n} p_i x_i) = f(s \sum_{i=1}^{n-1} (p_i/s) x_i + p_n x_n)$
$\leq s f(\sum_{i=1}^{n-1} (p_i/s) x_i) + p_n f(x_n)$ *(convexité de f)*
$\leq s \sum_{i=1}^{n-1} (p_i/s) f(x_i) + p_n f(x_n)$ *(hypothèse de récurrence)*
$= \sum_{i=1}^{n} p_i f(x_i)$
La dernière inégalité est une conséquence directe de la précédente en remplaçant les espérances par leur définition.

c.q.f.d. ◊◊◊

Proposition 8 $2(1 + \frac{1}{n}) \log(n + 1) - 3 \leq \mathbf{E}(h_m) \leq 2(1 + \frac{1}{n})(1 + \log(n)) - 3$

Preuve
D'après le lemme 2, la probabilité que les enregistrements d'identifiants soient comparés est égale à $\frac{2}{j-i+1}$, soit $\mathbf{E}(X_{i,j}) = \frac{2}{j-i+1}$.
Donc :
$\mathbf{E}(h_m) = 1 + \frac{1}{n} \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k}$ *(linéarité de l'espérance)*
$= 1 + \frac{1}{n} \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k}$ *(inversion des sommes)*
$= 1 + \frac{1}{n} \sum_{k=2}^{n} \frac{2(n+1-k)}{k}$
$= 1 - \frac{2(n-1)}{n} + \frac{2(n+1)}{n} \sum_{k=2}^{n} \frac{1}{k}$
$= 1 - \frac{2(n-1)}{n} + \frac{2}{n} \sum_{k=2}^{n} \frac{1}{k} + 2 \sum_{k=2}^{n} \frac{1}{k}$

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

Par conséquent,
$-3 + 2(1+\frac{1}{n})\int_1^{n+1}\frac{1}{x}dx \leq \mathbf{E}(h_m) \leq -3 + 2(1+\frac{1}{n})(1+\int_1^n\frac{1}{x}dx)$
$2(1+\frac{1}{n})\log(n+1) - 3 \leq \mathbf{E}(h_m) \leq 2(1+\frac{1}{n})(1+\log(n)) - 3.$

c.q.f.d. $\Diamond\Diamond\Diamond$

En supposant que les enregistrements sont recherchés avec la même probabilité, $\mathbf{E}(h_m)$ nous fournit le nombre de noeuds visités lors d'une recherche fructueuse. Nous en conclusons que la complexité d'une recherche fructueuse dans un arbre est en $\Theta(\log(n))$.

Les cas de l'ajout et de la recherche infructueuse sont identiques. Supposons que l'on recherche (ou ajoute) un noeud d'identifiant id placé entre id_i et id_{i+1}. La somme des probabilités d'une comparaison avec tous les noeuds nous fournit le nombre moyen de noeuds visités par cette recherche (noté nb). En distinguant selon les identifiants plus petits et plus grands, on obtient :
$nb = \sum_{k=1}^{i}\frac{1}{k} + \sum_{k=1}^{n-i}\frac{1}{k}$
$\leq 1 + \int_1^i\frac{1}{x}dx + 1 + \int_1^{n-i}\frac{1}{x}dx$
$= 2 + \log(i) + \log(n-i) \leq 2 + 2\log(\frac{n}{2})$ *(concavité du* log*)*
$\leq 2\log(n) + 1$

Par conséquent, en moyenne la recherche infructeuse et l'ajout se font aussi en $O(\log(n))$.

Dans ce qui suit, on note h_n la v.a. h_{max} d'un arbre obtenu par ajout aléatoire de n enregistrements et $f_n = 2^{h_n}$ la *hauteur exponentielle* de l'arbre. Nous allons raisonner sur f_n et ceci parce que l'opérateur max apparait dans nos équations de récurrence. La majoration simple $\max(x,y) \leq x+y$ conduit dans le cas défavorable où $x = y$, à remplacer x par $2x$. En passant à l'exponentielle, on obtient : $2^{\max(x,y)} = \max(2^x, 2^y) \leq 2^x + 2^y$. D'où $\max(x,y) \leq \log_2(2^x + 2^y)$. Cette fois-ci dans le cas défavorable où $x = y$, on remplace x par $x+1$!

Proposition 9 $\forall n, \mathbf{E}(f_n) \leq n^3 + 1$

Preuve
Le premier enregistrement inséré est choisi de manière équiprobable parmi les n enregistrements. S'il s'agit du i^{eme} enregistrement (par ordre croissant), alors son sous-arbre gauche aura $i-1$ enregistrements et une hauteur dont la distribution est celle de h_{i-1} et son sous-arbre droit aura $n-i$ enregistrements et une hauteur dont la distribution est celle de h_{n-i}. Notons $f_{n,i}$ la v.a. représentant la hauteur exponentielle sachant que le premier enregistrement inséré est celui d'identifiant id_i. Alors :
$f_{n,i} = 2^{1+\max(h_{i-1},h_{n-i})} = 2\cdot 2^{\max(h_{i-1},h_{n-i})} \leq 2(2^{h_{i-1}} + 2^{h_{n-i}}) = 2(f_{i-1} + f_{n-i})$
Par conséquent :
$\mathbf{E}(f_n) = \frac{1}{n}\sum_{i=1}^{n}\mathbf{E}(f_{n,i}) \leq \frac{1}{n}\sum_{i=1}^{n}2(\mathbf{E}(f_{i-1}) + \mathbf{E}(f_{n-i})) = \frac{4}{n}\sum_{i=0}^{n-1}\mathbf{E}(f_i)$

Vérifions l'inéquation de la proposition par récurrence.
Elle est satisfaite pour $n = 0$ puisque $f_0 = 1$.
Elle est satisfaite pour $n = 1$ puisque $f_1 = 2$.
Soit $n \geq 2$. Supposons-la satisfaite jusqu'à $n-1$. Alors :

$\mathbf{E}(f_n) \leq \frac{4}{n}\sum_{i=0}^{n-1}\mathbf{E}(f_i) \leq \frac{4}{n}\sum_{i=0}^{n-1}(i^3+1)$
$= \frac{4}{n}(\frac{n^2(n-1)^2}{4}+n) = n(n-1)^2+4 \leq n^3+1$

c.q.f.d. ◊◊◊

Nous sommes maintenant en mesure de borner la hauteur moyenne d'un arbre.

Proposition 10 $\forall n \geq 1, \mathbf{E}(h_n) \leq 3\log_2(n)+1$

Preuve
$n^3+1 \geq \mathbf{E}(f_n) = \mathbf{E}(2^{h_n}) \geq 2^{\mathbf{E}(h_n)}$ *(convexité de* 2^x *et lemme 4)*
D'où :
$\mathbf{E}(h_n) \leq \log_2(n^3+1) = \log_2(n^3(1+\frac{1}{n^3})) = \log_2(n^3)+\log_2(1+\frac{1}{n^3}) \leq 3\log_2(n)+1$

c.q.f.d. ◊◊◊

2.4.2 Listes à trous

Nous nous intéressons toujours au problème classique d'insérer, de supprimer et de rechercher des données dotées d'une clef à valeurs dans un ensemble totalement ordonné. Les solutions les plus efficaces reposent sur des arbres binaires de recherche « équilibrés ». Il existe plusieurs possibilités d'équilibrer les arbres mais elles requièrent toutes une certaine ingéniosité. La complexité des opérations est alors au pire des cas en $O(\log(n))$ où n est le nombre d'éléments présents dans la structure au moment de l'opération.

Si on se passe de ce mécanisme d'équilibrage, alors moyennant des contraintes d'équiprobabilité sur l'occurrence des clefs et sur l'absence de suppression, on démontre que le temps moyen des opérations se fait aussi en $O(\log(n))$ (voir la section précédente).

Nous allons illustrer l'intérêt des algorithmes probabilistes en montrant qu'on arrive au même résultat (et même mieux) sans aucune hypothèse sur les données et les opérations.

La structure de données que nous étudions s'appelle liste à trous. Elle est représentée sur la figure 2.8. Il s'agit d'un ensemble de listes doublement chainées superposées par double chaînage. Ces listes vérifient les propriétés suivantes :

- Toutes les listes sont triées.
- La liste la plus basse contient tous les éléments de la structure ainsi que les valeurs bornantes $-\infty, +\infty$.
- Toute autre liste contient un sous-ensemble non nécessairement strict des éléments de la liste en-dessous d'elle ainsi que les valeurs bornantes $-\infty, +\infty$.
- Seule la liste la plus haute est réduite aux valeurs bornantes $-\infty, +\infty$.

Lorsqu'un nouvel élément doit être inséré, on détermine sa position dans la liste la plus basse par le mécanisme de recherche que nous détaillons plus loin. On l'insère dans cette liste, puis par un tirage aléatoire équiprobabiliste, on choisit ou non de l'insérer dans la liste supérieure. Le mécanisme de recherche garantit que cette insertion se fait en temps constant. Si l'élement a été inséré alors on réitère le procédé. Lorsqu'on insère dans la liste la plus haute, on crée une liste au-dessus. Notons que l'on peut être amené à créer plusieurs listes par l'insertion d'un unique élément.

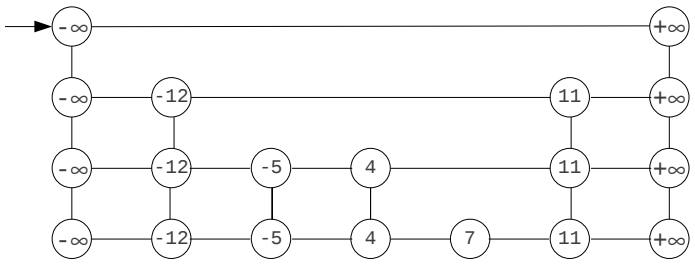

FIGURE 2.8: Listes à trous

Analyse de complexité

Afin d'estimer les temps d'opérations nous établissons d'abord un résultat sur le nombre de listes superposées.

Proposition 11 *Soit H la v.a. associée au nombre de listes. Alors dans une structure à $n > 0$ éléments on a $Pr(H \geq 3\log(n) + 2) \leq \frac{1}{n^2}$*

Preuve
La probabilité qu'il y ait $h+2$ listes correspond à la probabilité qu'au moins un élément de la liste ait été dupliqué h fois. En appliquant la majoration « union-somme », on obtient $Pr(H \geq h + 2) \leq \frac{n}{2^h}$. Par conséquent,
$Pr(H \geq 3\log_2(n) + 2) \leq \frac{n}{2^{3\log_2(n)}} = \frac{1}{n^2}$.

c.q.f.d. ◊◊◊

Nous décrivons maintenant les opérations. La recherche d'un élément se pratique ainsi. On parcourt la liste la plus haute jusqu'à ce qu'on rencontre l'élément recherché ou que l'élément suivant soit strictement plus grand que l'élément recherché. Dans le premier cas on descend à la verticale (en vue d'une éventuelle suppression) et dans le deuxième cas on descend à la liste inférieure et on réitère le procédé. A la fin de la procédure, si l'élément est absent, on dispose sur toutes les listes d'un pointeur sur le plus grand élément qui le précède. Nous avons illustré sur la figure 2.9 la recherche de l'élément de clef 8.

Proposition 12 *Soit R la v.a. associée au nombre de parcours de pointeurs lors d'une recherche. Alors dans une structure à $n \geq 16$ éléments on a :*
$Pr(R > \log_2(n)(12\log_2(n) + 8)) \leq \frac{2}{n}$ *et* $E(R) \leq 6\log_2(n) + 6$

Preuve
Dans la formule ci-dessous on intègre le parcours du pointeur vertical qui accède à la liste i et N_i le nombre de parcours de pointeurs horizontaux de la liste i.
$Pr(R > \log_2(n)(12\log_2(n) + 8))$
$= Pr(\sum_{1\leq i} 1_{H\geq i}(1 + N_i) > \log_2(n)(12\log_2(n) + 8))$
$\leq Pr(\sum_{1\leq i\leq 3\log_2(n)+2}(1+N_i)+\sum_{3\log_2(n)+2<i} 1_{H\geq i}(1+N_i) > \log_2(n)(12\log_2(n)+8))$
$\leq Pr(\sum_{1\leq i\leq 3\log_2(n)+2}(1 + N_i) > \log_2(n)(6\log_2(n) + 4))$

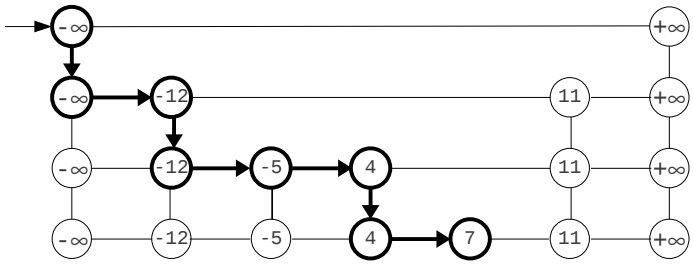

FIGURE 2.9: Recherche de la clef 8

$+Pr(\sum_{3\log_2(n)+2<i} 1_{H\geq i}(1+N_i) > \log_2(n)(6\log_2(n)+4))$
$\leq Pr(\sum_{1\leq i\leq 3\log_2(n)+2}(1+N_i) > \log_2(n)(6\log_2(n)+4)) + Pr(H > 3\log_2(n)+2)$
$\leq \sum_{1\leq i\leq 3\log_2(n)+2} Pr(1+N_i > 2\log_2(n)) + \frac{1}{n^2}$
Pour étudier le parcours de recherche d'un point de vue probabiliste, rien n'empêche d'imaginer qu'il démarre du dernier élément du parcours et remonte jusqu'à l'origine des listes à trous. Soit un élément examiné $e \neq -\infty$ de la liste i, l'élément précédemment examiné peut être soit un certain $e' < e$ sur la liste i, soit e sur la liste $i+1$. Le deuxième cas intervient avec une probabilité $1/2$, d'après le mécanisme d'insertion. Dans le cas où $e = -\infty$, cette probabilité est égale à 1. Par conséquent pour tout i, la v.a. N_i est t.q. $Pr(N_i \geq m) \leq \frac{1}{2^m}$. En reportant dans la dernière inéquation, on obtient :
$Pr(R > \log_2(n)(12\log_2(n)+8)) \leq \frac{3\log_2(n)+2}{n^2} + \frac{1}{n^2}$
$\leq \frac{2}{n}$ dès que $n \geq 16$.
Pour calculer l'espérance de R, on décompose dans le cas défavorable en parcours de pointeurs horizontaux et verticaux. Remarquons que le nombre de parcours de pointeurs horizontaux est inférieur ou égal à n. On se sert aussi de l'inégalité $3\log_2(n+1) \leq 3\log_2(n)+1$ dès que $n \geq 4$.
$E(R) = E(\sum_{1\leq i} 1_{H\geq i}(1+N_i))$
$\leq E(\sum_{1\leq i\leq 3\log_2(n)+2}(1+N_i) + nPr(H > 3\log_2(n)+2) + \sum_{3\log_2(n)+2<i} Pr(H \geq i)$
$\leq E(\sum_{1\leq i\leq 3\log_2(n)+2}(1+N_i)) + 1 + \sum_{3\log_2(n)+2<i} \frac{1}{i^2}$
$\leq \sum_{1\leq i\leq 3\log_2(n)+2} E(1+N_i) + 2$
$\leq 6\log_2(n) + 6$

c.q.f.d. ◊◊◊

La suppression se fait en effectuant une recherche puis en supprimant l'élément dans toutes les listes en partant de la liste la plus basse (ceci peut entraîner des suppressions des listes les plus hautes). L'insertion se fait en effectuant une recherche puis en insérant l'élément comme indiqué précédemment, la recherche ayant permis d'obtenir les pointeurs appropriés. Dans les deux cas, le surcoût est indépendant de n et suit en première approximation une loi binomiale de paramètre $1/2$.

2.5 Complexité amortie

2.5.1 Principe

Supposons qu'une structure de données soit manipulée par des opérations et que nous désirions mesurer la complexité de ces opérations. Une première approche consiste à étudier chaque opération (disons `op`) individuellement et établir une borne supérieure de son temps d'exécution dans le pire des cas (disons $t^+(\texttt{op})$).

Lorsqu'on analyse ensuite la complexité moyenne d'une suite d'opérations $\texttt{op}_1, \ldots, \texttt{op}_k$ on effectue le calcul $(1/k)\sum_{i=1}^{k} t^+(\texttt{op}_i)$. Cependant pratiquer de cette manière est souvent approximatif car, par exemple, le pire cas d'exécution de $\texttt{op}_1$ pourrait être suivi du meilleur cas d'exécution de $\texttt{op}_2$.

Nous illustrons notre propos sur une structure de données très simple, un compteur représenté en binaire, et une unique opération, l'incrémentation détaillée au niveau des bits par l'algorithme 15.

Algorithme 15: Incrémentation d'un compteur

Data : *cpt* un tableau de n bits, i.e. un compteur

`Incrémenter`()
$i \leftarrow 1$; $cpt[i] \leftarrow cpt[i] + 1 \mod 2$
while $i < n \wedge cpt[i] = 0$ **do** $i \leftarrow i + 1$; $cpt[i] \leftarrow cpt[i] + 1 \mod 2$

Une analyse isolée de la complexité de l'incrémentation conduit à un temps d'exécution de n opérations de bits. Imaginons maintenant une suite de k opérations. Le bit de poids le plus faible est modifié par chaque opération, le deuxième bit une fois sur deux, le troisième bit une fois sur quatre, etc. Par conséquent, on obtient comme borne supérieure de complexité :

$$\sum_{i=1}^{n} \left\lceil \frac{k}{2^{i-1}} \right\rceil \leq \sum_{i=1}^{n} (\frac{k}{2^{i-1}} + 1) \leq 2k + n$$

et par conséquent, une complexité moyenne égale à $2 + \frac{n}{k}$.

Dans le cas particulier où le compteur est initialement nul, on obtient :

$$\sum_{i=1}^{n} \left\lfloor \frac{k}{2^{i-1}} \right\rfloor \leq 2k$$

et par conséquent, une complexité moyenne égale à 2.

2.5.2 Méthode du potentiel

La méthode du potentiel mesure la complexité de la structure de données sur laquelle portent les opérations par une fonction appelée potentiel, que nous notons `Pot`. Nous introduisons quelques notations afin de raisonner sur cette fonction : S est l'espace des états de la structure, $c(\texttt{op})$ est la complexité d'une opération et $s \xrightarrow{\texttt{op}} s'$ signifie que l'opération `op` transforme l'état s en état s'.

La complexité amortie d'une opération $a(\texttt{op})$ tient compte du changement d'état opéré sur la structure de données *via* la fonction de potentiel. Si $s \xrightarrow{\texttt{op}} s'$ alors :

$$a(\texttt{op}) = c(\texttt{op}) + \texttt{Pot}(s') - \texttt{Pot}(s)$$

La proposition suivante illustre le lien entre complexité amortie et complexité réelle.

Proposition 13 *Soient une suite d'opérations* $s_0 \xrightarrow{\texttt{op}_1} s_1 \dots s_{k-1} \xrightarrow{\texttt{op}_k} s_k$. *On a l'égalité suivante :*

$$\sum_{i=1}^{k} a(\texttt{op}_i) = \sum_{i=1}^{k} c(\texttt{op}_i) + \texttt{Pot}(s_k) - \texttt{Pot}(s_0)$$

Preuve
La preuve est immédiate par récurrence.

c.q.f.d. ◊◊◊

Appliquons ce résultat au problème du compteur. La complexité de la structure peut être mesurée en nombre de bits à 1 car ces bits provoqueront plus tard une retenue. Calculons la complexité amortie d'une incrémentation :

- S'il n'y a pas de débordement et m modifications de bits alors $m-1$ bits sont mis à zéro et un bit est mis à 1. D'où une complexité amortie égale à 2.
- S'il y a un débordement (soit n modifications de bits) alors n bits sont mis à zéro. D'où une complexité amortie nulle.

Dans tous les cas, la complexité amortie d'une opération est bornée par 2 et la différence de potentiel est minorée par $-n$ (et par 0 dans le cas d'un compteur initialement nul). On retrouve ainsi de manière immédiate les complexités que l'on avait obtenues précédemment.

2.5.3 Une illustration : la gestion de pile avec mémoire statique

Supposons que l'on doive implémenter une pile en évitant (tant que faire se peut) le débordement. Le système d'exploitation propose une allocation de tableau dont on fixe la taille (et une libération de tableau). Aussi une idée simple lors d'un empilement consiste, lorsque le tableau alloué est plein, à allouer un tableau plus grand et recopier la pile dans le nouveau tableau. Ceci est illustré par l'algorithme 16. L'opération de dépilement se fait de manière standard sans désallocation (voir les exercices pour un dépilement avec désallocation).

La fonction f détermine l'accroissement en fonction de la taille courante. Il n'est pas évident de déterminer une bonne fonction. Dans le cas sans débordement, l'empilement se fait en $O(1)$ (et le dépilement se fait toujours en $O(1)$). Aussi nous voudrions obtenir ce même $O(1)$ en complexité amortie. Afin d'y parvenir, nous choisissons $f(n) = 2n$ et initialement un tableau est déjà alloué de taille $n = 1$.

La complexité de la structure de données du point de vue de l'opération est inversement proportionnelle au nombre de cellules vides. Immédiatement après

Algorithme 16: Empilement avec allocation

Data : PT un pointeur sur un tableau

Data : n la taille du tableau

Data : *top* l'indice du sommet de pile

`Empiler`(v)
Data : $PT2$ un pointeur sur un tableau

Data : i un indice

$top \leftarrow top + 1$
if $top > n$ **then**
  $PT2 \leftarrow$ `Allouer`($f(n)$)
  for i **from** 1 **to** n **do** $*PT2[i] \leftarrow *PT[i]$
  `Libérer`($*PT$) ; $PT \leftarrow PT2$; $n \leftarrow f(n)$
end
$*PT[top] \leftarrow v$

une nouvelle allocation, le nombre de cellules vides est exactement égal à la moitié de la taille du tableau. Ceci suggère de choisir pour fonction de potentiel $\texttt{Pot} = 2top - n$. Ainsi la fonction de potentiel est égale à 2 après une allocation et croît par pas de 2 ensuite à chaque nouvel empilement.

Calculons la complexité (amortie) de l'opération d'empilement en nombre d'affectations d'une cellule d'un tableau (que ce soit PT ou $PT2$). Il y a deux cas à considérer :

- Il n'y a pas de débordement. Seule une cellule est affectée et la fonction de potentiel croît de 2. D'où une complexité amortie égale à 3.
- Il y a un débordement. Par conséquent, $n+1$ cellules seront affectées. La fonction de potentiel est égale à n et sa nouvelle valeur est 2. D'où une complexité amortie égale à 3.

Sachant que la fonction de potentiel est initialement égale à -1 et qu'ensuite la fonction de potentiel est toujours strictement positive, on en déduit qu'une suite de n empilements conduit à une complexité strictement inférieure à $3n$.

2.6 Bornes inférieures de complexité

Les bornes inférieures de complexité caractérisent la difficulté intrinsèque d'un problème et non pas l'efficacité d'un algorithme particulier. Aussi leur obtention requiert généralement une analyse plus approfondie du problème et des techniques de preuve ad hoc. Nous illustrons cette diversité des méthodes sur deux exemples et on en donne un exemple supplémentaire dans les exercices.

2.6.1 Borne inférieure pour le tri par comparaison

De très nombreux algorithmes de tri par comparaison effectuent un nombre de comparaisons en $O(n \log(n))$ au pire des cas (avec n le nombre d'entrées à trier). Il s'avère que cet ordre de grandeur est optimal même en cas d'une analyse en moyenne.

Nous associons à un tri, un arbre binaire dont chaque feuille est étiquetée par une permutation différente de $\{1, \ldots, n\}$ et telles que toutes les permutations soient présentes. Les autres noeuds sont étiquetés par des sous-ensembles non vides de permutations.

Initialement l'arbre est réduit à la racine étiquetée par l'ensemble des permutations et à chaque permutation σ, on associe le tableau T défini par $T[i] = \sigma(i)$. Pour chaque noeud qui n'est pas une feuille, on poursuit l'exécution de l'algorithme jusqu'à l'exécution d'une comparaison (i.e. un `if` ou un `while`). Si pour toutes les permutations du sous-ensemble courant et leur tableau associé, le résultat du test est le même, on poursuit l'exécution.

Sinon on crée deux fils au noeud courant, étiquetés par la partition du sous-ensemble courant en fonction du résultat du test et à chaque nouveau noeud correspond un point d'exécution différent.

Nécessairement l'algorithme ne peut s'arrêter lorsque le sous-ensemble n'est pas un singleton car alors pour au moins une entrée le tri serait erroné. On a donc bien établi que ce procédé construit l'arbre recherché.

Définissons la *hauteur* d'un noeud comme la longueur du chemin de la racine à ce noeud. Etant donnée une permutation, la hauteur de la feuille fournit le nombre de comparaisons « utiles » au tri et donc constitue un minorant du nombre de comparaisons effectuées par l'algorithme dans le cas du tableau associé à la permutation.

La *hauteur moyenne* d'un arbre est la moyenne des hauteurs de ses feuilles.

Proposition 14 *La hauteur moyenne d'un arbre binaire comportant n feuilles est supérieure ou égale à* $\log_2(n)$.

Preuve
Nous établissons la preuve par récurrence. Dans le cas $n = 1$, on a l'égalité. Supposons l'inégalité établie jusqu'à $n-1$. Soit un arbre binaire $\mathcal{A}$ comportant n feuilles et soient $\mathcal{A}_g$ et $\mathcal{A}_d$ ses sous-arbres comportant p et $n-p$ feuilles.

D'après l'hypothèse de récurrence, la somme des hauteurs des feuilles de $\mathcal{A}_g$ (resp. $\mathcal{A}_d$) est supérieure ou égale à $p \log_2(p)$ (resp. $(n-p)\log_2(n-p)$). Par conséquent la somme des hauteurs des feuilles de $\mathcal{A}$ est supérieure ou égale à $n + p\log_2(p) + (n-p)\log_2(n-p)$. La fonction $x \log_2(x)$ est convexe. Donc $1/2(p\log_2(p) + (n-p)\log_2(n-p)) \geq (n/2)\log_2(n/2)$. Ce qui est équivalent à $n + p\log_2(p) + (n-p)\log_2(n-p) \geq n + n(log_2(n) - 1) = n\log_2(n)$. D'où une hauteur moyenne supérieure ou égale à $\log_2(n)$.

c.q.f.d. ◊◊◊

On en déduit immédiatement la borne inférieure recherchée puisque l'arbre d'un algorithme a $n!$ feuilles.

Proposition 15 *Sous hypothèse d'équiprobabilité des permutations des valeurs, le nombre moyen de comparaisons d'un algorithme de tri (à base de comparaisons) est au moins égal à* $\log_2(n!) \sim n\log_2(n)$.

2.6.2 Bornes inférieures pour le calcul arithmétique

Un *calcul arithmétique* sur un corps K à partir d'un ensemble de paramètres $\{a_1, \ldots, a_n\}$ est une suite d'instructions de type :

$$f_1 \leftarrow o_1 \ \mathsf{op_1} \ o'_1; f_2 \leftarrow o_2 \ \mathsf{op_2} \ o'_2; \ldots; f_k \leftarrow o_k \ \mathsf{op_k} \ o'_k;$$

avec pour tout $1 \leq i \leq k$, f_i est une variable du calcul, $\mathsf{op_i} \in \{+, -, \times\}$ et les opérandes o_i, o'_i sont soit des élements de K soit des paramètres soit l'une des variables $\{f_1, \ldots, f_{i-1}\}$.

Le programme ci-dessous calcule le produit de deux nombres complexes $(a + ib)(c + id)$ (le résultat est $f_3 + if_6$). Ici les paramètres sont a, b, c, d.

$$f_1 \leftarrow a \times c; f_2 \leftarrow b \times d; f_3 \leftarrow f_1 - f_2; f_4 \leftarrow a \times d; f_5 \leftarrow b \times c; f_6 \leftarrow f_4 + f_5;$$

On peut trouver un autre calcul de ce produit qui n'utilise que 3 multiplications. qui repose sur les égalités $ad + bc = (a + b)c + a(d - c)$ et $ac + bd = (a + b)c - b(c + d)$:

$$f_1 \leftarrow a + b; f_2 \leftarrow f_1 \times c; f_3 \leftarrow d - c; f_4 \leftarrow a \times f_3;$$
$$f_5 \leftarrow f_4 + f_2; f_6 \leftarrow d + c; f_7 \leftarrow b \times f_6; f_8 \leftarrow f_2 - f_7;$$

Soit un calcul qui renvoie r résultats et qui comprend s multiplications. On remarque que la valeur de f_i appartient à $\mathsf{K}[a_1, \ldots, a_n]$ (l'anneau des polynômes dont les variables sont $a_1, \ldots, a_n$), on définit le vecteur $\mathbf{e}$ comme les f_i résultats d'une multiplication. Chaque f_i résultat est alors obtenu par additions et soustractions à partir des $a_1, \ldots, a_n$ et des coefficients de $\mathbf{e}$. Par conséquent, le vecteur des résultats, noté $\mathbf{v}$, vérifie $\mathbf{v} = \mathbf{Me} + \mathbf{h}$ où $\mathbf{M}$ est une matrice $r \times s$ à valeurs dans K, $\mathbf{e}$ est un vecteur de dimension s à valeurs dans $\mathsf{K}[a_1, \ldots, a_n]$ et $\mathbf{h}$ est un vecteur de dimension r dont les coefficients sont de la forme $c_0 + \sum_{i=1}^{n} c_i a_i$ avec $c_i \in \mathsf{K}$ pour tout i.

Soient $\mathbf{v}_1, \ldots, \mathbf{v}_m$ des vecteurs de dimension r à coefficients dans $\mathsf{K}[a_1, \ldots, a_n]$, on dit que $\mathbf{v}_1, \ldots, \mathbf{v}_m$ sont *linéairement indépendants modulo* K si :

$$\forall c_i \in \mathsf{K} \ \sum_{i=1}^{n} c_i \mathbf{v}_i \in \mathsf{K}^r \Rightarrow \forall c_i \ c_i = 0$$

Le *rang ligne (resp. colonne) modulo* K d'une matrice à coefficients dans $\mathsf{K}[a_1, \ldots, a_n]$ est le nombre maximal de vecteurs lignes (colonnes) linéairement indépendants modulo K.

Nous établissons une première borne inférieure de complexité pour un produit matrice vecteur.

Proposition 16 *Soit un calcul dont les paramètres sont* $a_1, \ldots, a_n, x_1, \ldots, x_p$. *Ce calcul effectue le produit matrice-vecteur* $\mathbf{Ax}$ *où* $\mathbf{A}$ *est une matrice* $r \times p$ *à coefficients dans* $\mathsf{K}[a_1, \ldots, a_n]$ *et* $\mathbf{x} = (x_1, \ldots, x_p)$. *Alors le nombre de multiplications de ce calcul est au moins égal au rang ligne modulo* K *de* $\mathbf{A}$.

Preuve

On peut supposer que le rang ligne de la matrice $\mathbf{A}$ modulo K est son nombre de lignes. En effet ce calcul est aussi un calcul de la matrice restreinte à une famille maximale de vecteurs lignes indépendants. On note l'équation correspondant à ce calcul $\mathbf{Ax} = \mathbf{Me} + \mathbf{h}$.

Supposons que s le nombre de multiplications vérifie $s < r$. Par conséquent les vecteurs lignes de $\mathbf{M}$ sont linéairement dépendants ; autrement dit, il existe un vecteur $\mathbf{y} \neq 0$ de $\mathbf{K}^r$ tel que $\mathbf{y}^T\mathbf{M} = 0$. En appliquant $\mathbf{y}^T$ à l'équation, on obtient : $\mathbf{y}^T\mathbf{A}\mathbf{x} = \mathbf{y}^T\mathbf{h}$.

Par définition, $\mathbf{y}^T\mathbf{h}$ est une expression de type $c_0 + \sum_{i=1}^{n} c_i a_i + \sum_{i=1}^{s} c'_i x_i$ avec $c_i \in \mathsf{K}$. Par conséquent le vecteur $\mathbf{y}^T\mathbf{A}$ a ses coefficients dans K sinon on obtiendrait un monôme de degré au moins 2 dans $\mathbf{y}^T\mathbf{A}\mathbf{x}$. Par conséquent les vecteurs lignes de $\mathbf{A}$ sont liés modulo K contrairement à l'hypothèse.

c.q.f.d. ◊◊◊

A titre d'application, on en déduit qu'un calcul de $ac, bd, ad + bc$ requiert au moins trois multiplications. En effet, le résultat s'exprime comme le produit matrice-vecteur suivant :

$$\begin{pmatrix} a & 0 \\ 0 & b \\ b & a \end{pmatrix} \quad \times \quad \begin{pmatrix} c \\ d \end{pmatrix}$$

Montrons que le rang modulo K de la matrice est 3. Soient $c_1, c_2, c_3 \in \mathsf{K}$ tels que $ac_1 + bc_3, bc_2 + ac_3 \in \mathsf{K}$ ce qui implique d'abord $c_1 = c_3 = 0$ puis $c_2 = 0$.

Afin d'établir une deuxième borne inférieure de complexité, nous utilisons le lemme suivant.

Lemme 5 *Soit $\{\mathbf{v}_1, \ldots, \mathbf{v}_m\}$ un ensemble de vecteurs de dimension r à coefficients dans $\mathsf{K}[a_1, \ldots, a_n]$ contenant q vecteurs linéairement indépendants modulo K. Alors pour tout $b_2, \ldots, b_m \in \mathsf{K}$, l'ensemble $\{\mathbf{v}_2 + b_2\mathbf{v}_1, \ldots, \mathbf{v}_m + b_m\mathbf{v}_1\}$ contient $q - 1$ vecteurs linéairement indépendants modulo K.*

Preuve

En renumérotant les vecteurs si nécessaire, il n'y a que deux cas à considérer : soit les vecteurs $\{\mathbf{v}_1, \ldots, \mathbf{v}_q\}$ sont linéairement indépendants, soit les vecteurs $\{\mathbf{v}_2, \ldots, \mathbf{v}_{q+1}\}$ le sont (modulo K ce que nous ne répéterons pas dans la suite de la preuve).

On pose $\mathbf{v}'_i = \mathbf{v}_i + b_i\mathbf{v}_1$.

Supposons que les $\mathbf{v}_1, \ldots, \mathbf{v}_q$ sont linéairement indépendants. Si $\sum_{2 \leq i \leq q} c_i\mathbf{v}'_i \in \mathsf{K}^r$ alors $\sum_{1 \leq i \leq q} c_i\mathbf{v}_i \in \mathsf{K}^r$ avec $c_1 = \sum_{2 \leq i \leq q} c_i b_i$. Par conséquent tous les c_i sont nuls. Autrement dit, les $\mathbf{v}'_2, \ldots, \mathbf{v}'_q$ sont linéairement indépendants.

Supposons que les $\mathbf{v}_2, \ldots, \mathbf{v}_{q+1}$ sont linéairement indépendants.

Cas n°1. Les $\mathbf{v}'_2, \ldots, \mathbf{v}'_q$ sont linéairement indépendants, il n'y a rien à démontrer.

Cas n°2. Les $\mathbf{v}'_2, \ldots, \mathbf{v}'_q$ sont linéairement dépendants. Par conséquent, il existe $c_2, \ldots, c_q \in \mathsf{K}$ non tous nuls tels que $\sum_{2 \leq i \leq q} c_i\mathbf{v}'_i \in \mathsf{K}^r$. En renumérotant si nécessaire on suppose que $c_2 \neq 0$. Par conséquent avec les mêmes notations, $\sum_{1 \leq i \leq q} c_i\mathbf{v}_i = \mathbf{w} \in \mathsf{K}^r$ avec la même définition pour c_1. Nécessairement $c_1 \neq 0$. D'où : $\mathbf{v}_1 = c_1^{-1}(\mathbf{w} - \sum_{2 \leq i \leq q} c_i\mathbf{v}_i)$.

Cas n°2.1 Les $\mathbf{v}'_3, \ldots, \mathbf{v}'_{q+1}$ sont linéairement indépendants, il n'y a rien à démontrer.

Cas n°2.2 Les $\mathbf{v}'_3, \ldots, \mathbf{v}'_{q+1}$ sont linéairement dépendants. En raisonnant de manière analogue, on obtient : $\mathbf{v}_1 = d_1^{-1}(\mathbf{z} - \sum_{3 \leq i \leq q+1} d_i \mathbf{v}_i)$ où z joue le rôle de w et les d_i jouent le rôle des c_i.

En égalisant les deux expressions pour $\mathbf{v}_1$ et après des manipulations algébriques élémentaires, on trouve :
$d_1 c_2 \mathbf{v}_2 + \sum_{3 \leq i \leq q} (d_1 c_i - d_i c_1)\mathbf{v}_i - c_1 d_{q+1} \mathbf{v}_{q+1} = d_1 \mathbf{w} - c_1 \mathbf{z}$
Mais cette égalité (et le fait que $d_1 c_2 \neq 0$) contredit l'indépendance de $\mathbf{v}_2, \ldots, \mathbf{v}_{q+1}$.

c.q.f.d. ◊◊◊

Soit un calcul dont les paramètres sont $a_1, \ldots, a_n, x_1, \ldots, x_p$. Ce calcul effectue le produit matrice-vecteur $\mathbf{A}\mathbf{x} + \mathbf{y}$ où $\mathbf{A}$ est une matrice $r \times p$ à coefficients dans $\mathsf{K}[a_1, \ldots, a_n]$, $\mathbf{x} = (x_1, \ldots, x_p)$ et $\mathbf{y} = (y_1, \ldots, y_r)$ avec les $y_1, \ldots, y_r \in \mathsf{K}[a_1, \ldots, a_n]$. Une multiplication de ce calcul est dite *active* si l'une des opérandes contient un x_i et l'autre opérande n'est pas un élément de K.

Nous établissons maintenant une deuxième borne de complexité pour un produit matrice vecteur « étendu ».

Proposition 17 *Soit un calcul dont les paramètres sont $a_1, \ldots, a_n, x_1, \ldots, x_p$. Ce calcul effectue l'opération. $\mathbf{A}\mathbf{x} + \mathbf{y}$ où $\mathbf{A}$ est une matrice $r \times p$ à coefficients dans $\mathsf{K}[a_1, \ldots, a_n]$, $\mathbf{x} = (x_1, \ldots, x_p)$ et $\mathbf{y} = (y_1, \ldots, y_r)$ avec les $y_1, \ldots, y_r \in \mathsf{K}[a_1, \ldots, a_n]$. Alors le nombre de multiplications actives d'un tel calcul est au moins égal au rang colonne modulo K de $\mathbf{A}$.*

Preuve
On va établir le résultat par récurrence sur le rang colonne q.

Nous démarrons la récurrence à $q = 1$ afin de nous servir de ce résultat dans l'étape inductive ($q = 0$ est trivial). S'il n'y a pas de multiplications actives, alors les résultats s'écrivent $\sum_i c_i x_i + P(a_1, \ldots, a_n)$ où P est un polynôme. Les c_i sont exactement les coefficients de la matrice A. Par conséquent la matrice A est à coefficients dans K et son rang colonne est nul contrairement à l'hypothèse.

Soit $q > 1$. D'après l'hypothèse de récurrence ($q - 1 > 0$), il y a au moins une multiplication active. Considérons la première multiplication active $f_i \leftarrow g \times h$ avec $g = \sum_i c_i x_i + P(a_1, \ldots, a_n)$ (même raisonnement que pour le cas de base). On suppose sans perte de généralité que $c_1 \neq 0$.

On « fabrique » un nouveau calcul à partir du calcul considéré de la manière suivante. Si la première composante x_1 du vecteur $\mathbf{x}$ n'est plus un paramètre mais s'écrit $-c_1^{-1}(\sum_{i \geq 2} c_i x_i + P(a_1, \ldots, a_n))$ alors en remplaçant $f_i \leftarrow g \times h$ par $f_i \leftarrow 0$, on effectue bien le calcul $\mathbf{A}\mathbf{x} + \mathbf{y}$ dans ce cas particulier. Mais ce produit s'exprime aussi $\mathbf{A}'\mathbf{x}' + \mathbf{y}'$ avec $\mathbf{x}' = (x_2, \ldots, x_p)$, $\mathbf{y}' = \mathbf{y}' - c_1^{-1} P(a_1, \ldots, a_n)\mathbf{A}[-, 1]$ et pour $i \geq 2$, $\mathbf{A}'[-, i] = \mathbf{A}[-, i] - c_1^{-1} c_i \mathbf{A}[-, 1]$.

D'après le lemme, le rang colonne de $\mathbf{A}'$ est au moins égal à $q-1$ donc le nouveau calcul comporte au moins $q - 1$ multiplications actives et le calcul considéré en comporte au moins q.

c.q.f.d. ◊◊◊

En spécialisant ce résultat, on obtient la borne inférieure la plus significative.

Proposition 18 *Le calcul du produit d'une matrice $n \times p$ par un vecteur de dimension p où les paramètres sont les coefficients de la matrice et du vecteur requiert au moins np multiplications.*

Preuve
Soit la matrice $\mathbf{A}$ dont les élements sont les $a_{i,j}$ et le vecteur $\mathbf{x} = (x_1, \ldots, x_p)$. L'astuce consiste à réécrire le produit ainsi :

$$\begin{pmatrix} x_1 & \ldots & x_p & 0 & \ldots & 0 & 0 & \ldots & 0 \\ . & & & & & & & & . \\ . & & & & & & & & . \\ . & & & & & & & & . \\ 0 & \ldots & 0 & 0 & \ldots & 0 & x_1 & \ldots & x_p \end{pmatrix} \qquad \begin{pmatrix} a_{1,1} \\ . \\ . \\ . \\ a_{1,p} \\ . \\ . \\ . \\ a_{n,1} \\ . \\ . \\ . \\ a_{n,p} \end{pmatrix}$$

Il est alors immédiat que les np vecteurs colonnes de cette nouvelle matrice sont linéairement indépendants modulo K.

c.q.f.d. ◊◊◊

2.7 Exercices

Exercice 1. On dispose d'un tableau d'entiers $T[1..n]$. On cherche à calculer simultanément le minimum et le maximum de ce tableau.

1. Proposer un algorithme naïf. Donner le nombre exact de comparaisons effectuées lors du calcul dans le pire des cas.
2. Une idée est de regrouper les éléments par paires afin de diminuer le nombre de comparaisons à effectuer. Proposer un algorithme suivant ce principe et donner le nombre exact de comparaisons effectuées par celui-ci dans le pire des cas.

Exercice 2. Soit t une fonction des puissances de 2 vérifiant l'équation de récurrence pour $n > 1$:

$$t(n) = 2t(n/2) + n\log_2(n)$$

Calculer $t(n)$.

Exercice 3. On représente un polynôme P par un tableau de ses coefficients. Nous souhaitons écrire un algorithme effectuant la multiplication de deux polynômes P, Q de degré au plus $n-1$.

1. Proposer un algorithme naïf et donner sa complexité.
2. Soit $k = \lceil n/2 \rceil$. P et Q se décomposent de manière unique en :

   $$P = P^{(0)} + P^{(1)}X^k \quad Q = Q^{(0)} + Q^{(1)}X^k$$

   avec $P^{(0)}, P^{(1)}, Q^{(0)}, Q^{(1)}$ des polynômes de degré $< k$. Exprimer le produit PQ en fonction de ces polynômes.
3. Transformer cette expression en une expression qui ne fasse intervenir que 3 multiplications de polynômes.
4. En déduire un algorithme récursif efficace de produit de polynômes et analyser sa complexité.

Exercice 4. Dans cet exercice, on suppose que $n = 2^k$ et que dans l'anneau $\mathbb{A}$ considéré, 2 est inversible et on dispose d'une racine $n$ième primitive de l'unité ω, i.e. $\omega^n = 1 \wedge \forall 0 < n' < n\ \omega^{n'} - 1$ n'est pas un diviseur de 0. Soit $P[X] = \sum_{i=0}^{n-1} p_i X^i$, un polynôme de degré inférieur à n., on décompose $P = P^{(0)} + XP^{(1)}$ avec :

— $P^{(0)} = \sum_{i=0}^{n/2-1} p_{2i}(X^2)^i$
— $P^{(1)} = \sum_{i=0}^{n/2-1} p_{2i+1}(X^2)^i$

Montrer que

— $\forall 0 \leq k < n/2\ P^{(0)}(\omega^k) = P^{(0)}(\omega^{n/2+k}) = \sum_{i=0}^{n/2-1} p_{2i}(\omega^{2k})^i$
— $\forall 0 \leq k < n/2\ P^{(1)}(\omega^k) = P^{(1)}(\omega^{n/2+k}) = \sum_{i=0}^{n/2-1} p_{2i+1}(\omega^{2k})^i$
— $\forall 0 \leq k < n/2$
$P(\omega^k) = P^{(0)}(\omega^k) + \omega^k P^{(1)}(\omega^k)$ et $P(\omega^{n/2+k}) = P^{(0)}(\omega^k) - \omega^k P^{(1)}(\omega^k)$

En déduire un algorithme récursif qui évalue P pour les racines $n$ièmes de l'unité $1, \omega, \omega^2, \ldots, \omega^{n-1}$. Analyser sa complexité.

L'application qui associe au vecteur des coefficients de P le vecteur $(P(1), P(\omega), \ldots, P(\omega^{n-1}))$ est une application linéaire dont la matrice $M_{\omega,n}$ est décrite ci-dessous.

$$(P(1), P(\omega), \ldots, P(\omega^{n-1})) = \begin{pmatrix} 1 & 1 & \ldots & 1 \\ 1 & \omega & \ldots & \omega^{n-1} \\ \vdots & & & \vdots \\ 1 & \omega^{n-1} & \ldots & \omega^{(n-1)^2} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix}$$

Montrer que cette application est inversible et calculer la matrice inverse.

En déduire comment multiplier deux polynômes P et Q de $\mathbb{A}[X]/(X^n - 1)$ en $O(n \log(n))$ opérations arithmétiques, puis comment multiplier P et Q dans $\mathbb{A}[X]$.

Exercice 5. Soit T un tableau de n cellules contenant des valeurs numériques et $k \leq n$ un entier.

1. Ecrire un algorithme naïf qui renvoie la $k$ième plus petite valeur du tableau T et l'indice correspondant dans le tableau.
2. Ecrire un algorithme alternatif avec tri.
3. Comparer la complexité des deux algorithmes. Indiquez les cas favorables à l'un ou l'autre des algorithmes.
4. Soit l'algorithme 17. Démontrer que cet algorithme renvoie la $k$ième plus petite valeur du tableau T. Quel type de partition rend l'algorithme le plus efficace ?
5. En tenant compte de la question précédente, on propose l'algorithme 18 qui est une adaptation du précédent. Démontrer que n_1 et n_3 sont tous les deux inférieurs à $3n/4$.
6. Soit $Time(n)$, le pire temps d'exécution de cet algorithme pour un tableau de taille inférieure ou égale à n. Démontrer que pour une constante c bien choisie :

$$\forall n < 50 \; Time(n) \leq c \cdot n$$

$$\forall n \geq 50 \; Time(n) \leq c \cdot n + Time(\frac{n}{5}) + Time(\frac{3n}{4})$$

   En déduire la complexité de cet algorithme.

Exercice 6. Une matrice de Toeplitz est une matrice A de dimension $n \times n$ telle que $A[i, j] = A[i - 1, j - 1]$ pour $2 \leq i, j \leq n$.

1. Que peut-on dire de la somme et du produit de deux matrices de Toeplitz ?
2. Donner un algorithme qui additionne les matrices de Toeplitz en $O(n)$ et un algorithme qui effectue leur produit en $O(n^2)$.
3. Comment calculer le produit d'une matrice de Toeplitz $n \times n$ par un vecteur de longueur n ? Donner la complexité de l'algorithme.

Exercice 7. Soit $T[1..n]$ un tableau. Une valeur e présente dans T est dite majoritaire si T contient strictement plus de $\lfloor \frac{n}{2} \rfloor$ occurrences de e.

— Proposer un algorithme de recherche de valeur majoritaire en $O(n)$ en vous servant de l'exercice 5.

Algorithme 17: Un algorithme par partition

`Selectionne`(T, n, k) : entier

Input : T un tableau, n sa dimension, $k \leq n$ un entier
Output : un entier correspondant à la $k$ième plus petite valeur de T

Data : T_1, T_3 tableaux
Data : $valeur, n_1, n_2, n_3, i$ entiers

$n_1 \leftarrow 0$; $n_2 \leftarrow 0$; $n_3 \leftarrow 0$
`// On sélectionne une valeur du tableau`
$valeur \leftarrow T[1]$
for $i \leftarrow 1$ **to** n **do**
  if $T[i] < valeur$ **then**
    $n_1 \leftarrow n_1 + 1$
    $T_1[n_1] \leftarrow T[i]$
  else if $T[i] = valeur$ **then**
    $n_2 \leftarrow n_2 + 1$
  else
    $n_3 \leftarrow n_3 + 1$
    $T_3[n_3] \leftarrow T[i]$
  end
end
`// On recherche l'élément dans un des tableaux`
`// par un appel récursif ou`
`// on renvoie` $valeur$ `suivant les valeurs de` n_1`,` n_2 `et` n_3
if $n_1 \geq k$ **then return** `Selectionne`(T_1, n_1, k)
else if $n_1 + n_2 \geq k$ **then return** $valeur$
else return `Selectionne`$(T_3, n_3, k - (n_1 + n_2))$

Algorithme 18: Le plus efficace

```
Selectionne(T, n, k) : entier

Input : T un tableau, n sa dimension, k ≤ n un entier
Output : un entier correspondant à la kième plus petite valeur de T

Data : R, S, T1, T3 tableaux
Data : mediane, m, n1, n2, n3, i, j entiers

if n < 24 then
    // On applique l'algorithme de la question 3
    S ← Trie(T, 1, n)
    return S[k]
else
    // On divise T en paquets de 5 éléments et
    // on range dans R, l'élément médian de chaque paquet
    m ← ⌊n/5⌋
    for i ← 0 to m − 1 do
        for j ← 1 to 5 do
            S[j] ← T[5 ∗ i + j]
        end
        S ← Trie(S, 1, 5)
        R[i + 1] ← S[3]
    end
    // On calcule le médian des médians par un appel récursif
    mediane ← Selectionne(R, m, ⌈m/2⌉)
    for i ← 1 to n do
        if T[i] < mediane then
            n1 ← n1 + 1
            T1[n1] ← T[i]
        else if T[i] = mediane then
            n2 ← n2 + 1
        else
            n3 ← n3 + 1
            T3[n3] ← T[i]
        end
    end
    // On recherche l'élément dans un des tableaux
    // par un appel récursif ou
    // on renvoie mediane suivant les valeurs de n1, n2 et n3
    if n1 ≥ k then return Selectionne(T1, n1, k)
    else if n1 + n2 ≥ k then return mediane
    else return Selectionne(T3, n3, k − (n1 + n2))
end
```

— On présente (de façon ludique) un nouvel algorithme lorsqu'on ne peut tester que l'identité des valeurs. On possède n balles de couleurs et on veut savoir s'il y a une couleur majoritaire. On dispose d'une étagère, d'une boîte et d'une poubelle. L'algorithme procède en deux phases :

1. Après avoir posé une balle sur l'étagère, on examine les balles une par une afin de les ranger sur l'étagère sans qu'il y ait deux boules contigües de même couleur. Si la balle courante a une couleur différente de la dernière balle posée sur l'étagère, on la pose et on extrait une balle de la boîte (s'il y en a une) pour la poser sur l'étagère. Dans le cas de couleurs identiques, on insère la balle dans la boîte.
2. Soit C la couleur de la dernière balle posée sur l'etagère. On examine successivement (en ordre inverse de la pose) les balles de l'étagère. Si la balle courante n'a pas la couleur C et que la boîte est vide, on s'arrête et on déclare qu'il n'y a pas de couleur majoritaire. Si par contre la boîte n'est pas vide, on prend la balle courante et une balle de la boîte et on les jette à la poubelle. Si la balle courante a la couleur C et qu'il y a au moins deux balles sur l'étagère, on jette (à la poubelle) les deux dernières balles. Si c'est la dernière balle, on l'insère dans la boîte. Lorsqu'il n'y a plus de balles sur l'étagère on déclare C couleur majoritaire *ssi* la boîte n'est pas vide.

— Prouver qu'à tout moment de la première phase les couleurs de la boîte ont la même couleur que la dernière balle posée sur l'étagère et qu'il n'y a pas deux boules contigües de la même couleur sur l'étagère. En déduire que s'il y a une couleur majoritaire, alors il s'agit de C. En examinant la deuxième phase prouver que l'algorithme est correct.
— Donner son nombre de comparaisons dans le pire des cas.

Exercice 8. Nous établissons une borne inférieure sur un algorithme de recherche d'élément majoritaire qui ne pratique que des tests d'égalité. Cette borne porte sur ce nombre de comparaisons. On note $m = \lfloor n/2 \rfloor + 1$ le seuil de majorité.

Afin d'établir cette borne, nous imaginons que l'algorithme « joue » contre un adversaire qui maintient un ensemble d'entrées possibles en fonction des comparaisons de l'algorithme. Une entrée possible (qu'on appellera dans la suite *affectation*) est une partition de l'ensemble des indices telle que deux cellules du tableau sont identiques ssi les indices correspondants appartiennent à un même sous-ensemble de la partition. Afin d'y parvenir cet adversaire conserve une structure pour les indices du tableau dite *amphithéatre*. L'amphithéatre se décompose entre les gradins et l'arène.

Les indices sont répartis entre les gradins et l'arène. De plus dans l'arène, les indices sont :

— soit groupés en troupeaux ; au sein d'un troupeau les cellules des indices doivent avoir la même valeur. On note Tr, le nombre de troupeaux et t le nombre total d'indices dans un troupeau.
— soit groupés en binômes ; au sein d'un binôme les deux cellules des indices doivent avoir une valeur différente. On note B, le nombre de binômes

Enfin les indices mis dans les gradins doivent chacun avoir une couleur unique. Une affectation qui satisfait ces contraintes est dite *admissible*. On note g le nombre d'indices dans les gradins.

Initialement, tous les indices sont dans l'arène et constituent des troupeaux singletons. Lorsque l'algorithme effectue un test $T[i] = T[j]$, l'adversaire agit ainsi :

1. Si i ou j sont dans les gradins, alors la réponse est non et l'amphithéatre est inchangé.
2. Si i et j forment un binôme alors la réponse est non et l'amphithéatre est inchangé.
3. Si i (resp. j) est dans un binôme et j (resp. i) n'est pas son partenaire, alors la réponse est non et i (resp. j) est envoyé dans les gradins alors que son partenaire forme un troupeau singleton.
4. Si i et j sont dans un même troupeau, alors la réponse est oui et l'amphithéatre est inchangé.
5. Si i et j sont dans des troupeaux différents, alors l'action dépend de la valeur $d = B + t$.
   (a) Si $d > m$ les deux troupeaux sont des singletons (cf la proposition 19) alors la réponse est non et i, j forment un binôme.
   (b) Si $d = m$ la réponse est oui et les troupeaux de i et j fusionnent.

Etant donnée une exécution partielle de l'algorithme, une affectation est dite *cohérente* avec ce déroulement si les résultats des tests de l'algorithme sont ceux qui auraient été obtenus avec cette affectation.

Démontrez les propositions suivantes.

Proposition 19 *A tout instant de l'exécution de l'algorithme,*
— *d ne croît jamais,*
— *$d \geq m$,*
— *$d > m$ implique que les troupeaux sont des singletons.*

Proposition 20 *A tout instant de l'exécution de l'algorithme, une affectation admissible est cohérente.*

Proposition 21 *Lorsque l'algorithme se termine alors l'arène contient un unique troupeau de taille m.*

Proposition 22 *A tout moment de l'exécution le nombre de tests qui renvoient faux est au moins $2g + B$ et le nombre de tests qui renvoient vrai est au moins $t - Tr$.*

Proposition 23 *La complexité au pire des cas de tout algorithme de recherche d'élément majoritaire par test d'égalité est d'au moins $n + \lceil n/2 \rceil - 2$ tests.*

On comparera cette borne avec le nombre de comparaisons effectué par l'algorithme de l'exercice 7.

Exercice 9. Modifier la fonction dépiler de la gestion de la pile avec la mémoire statique de telle façon que les propriétés suivantes soient satisfaites :

1. Il existe $0 < r < 1$ telle que le ratio d'occupation du tableau soit toujours supérieur ou égal à r si la pile n'est pas vide.
2. La complexité amortie d'une opération est en $O(1)$.