Gilles HUNAULT

An 2001

Algorithmiques Raisonnées

Université d'Angers

Mesdames et messieurs, j'ai à traiter d'un sujet qui conduit tantôt à énoncer l'évidence et tantôt à poser une contradiction. Il s'agit en effet d'examiner les relations entre deux grandes entités qui sont respectivement la langue et la société.

Le langage est pour l'homme un moyen, en fait le seul moyen d'atteindre l'autre homme, de lui transmettre et de recevoir de lui un message. Par conséquent, le langage pose et suppose l'autre. Immédiatement, la société est donnée avec le langage. La société à son tour ne tient ensemble que par l'usage commun de signes de communication. Immédiatement, le langage est donné avec la société.[...]

Il y a donc deux propriétés inhérentes à la langue, à son niveau le plus profond. Il y a la propriété qui est constitutive de sa nature d'être formée d'unités signifiantes, et il y a la propriété qui est constitutive de son emploi de pouvoir agencer ces signes de manière signifiante. Ce sont là deux propriétés qu'il faut tenir distinctes, qui commandent deux analyses différentes et qui s'organisent en deux structures particulières. Entre ces deux propriétés le lien est établi par une troisième propriété. Nous avons dit qu'il y a d'une part des unités signifiantes, en second lieu la capacité d'agencer ces signes en matière signifiante et en troisième lieu, dirons-nous, il y a la propriété **syntagmatique**, *celle de les combiner dans certaines règles de consécution et seulement de cette manière.*

Emile Benveniste,
problèmes de linguistique générale, 2 :
structure de la langue et structure de la société

Table des matières

Introduction

Tout programme met en oeuvre une méthode, des idées, ce qui signifie que tout programme repose sur un algorithme, qu'il soit implicitement écrit dans les neurones du programmeur ou explicitement écrit sous forme d'algorithme. Dès lors qu'il existe, pourquoi ne pas l'écrire ? La traduction de l'algorithme en programme, traduction non pas mécanique mais fine, adaptée aux finesses (ou aux roueries) du langage, voire du système d'exploitation, en sera facilitée, voire reléguée à d'autres, notamment à des programmes de traduction qui assureront une première version exécutable que le spécialiste du langage pourra optimiser, peaufiner.

Un langage algorithmique est obligatoire, à la fois pour s'affranchir de la machine, des langages de programmation et pour formaliser le travail à accomplir, pour spécifier l'enchainement des actions, nommer les variables...

Si vous n'avez jamais programmé, ces lignes sont pour vous : ces pages ont pour but de vous apprendre ce que comprend l'ordinateur, comment il répond mécaniquent aux instructions et donc comment mettre bout à bout ces instructions. Cela vous permettra de structurer votre pensée, de communiquer vos méthodes de résolution à d'autres humains avant de les soumettre à l'ordinateur.

Si vous avez déjà programmé, ces lignes sont aussi pour vous : les langages changent, les algorithmes restent. De plus, pour produire du code clair, lisible, "propre", rien de tel que de s'abstraire de la syntaxe spécifique à un langage particulier. Et puis, il est possible qu'ayant commencé à programmer directement dans un langage vous ayez pris des mauvaises habitudes (ou simplement appris des recettes) sans savoir exactement ce que cela représente.

Si vous devez communiquer vos algorithmes, discuter des méthodes ou **si vous devez enseigner la programmation, ces lignes sont aussi pour vous** : vous y trouverez un vocabulaire simple, une syntaxe formalisée mais concise et lisible, une sémantique non ambigue et l'esprit dans lequel adapter notre langage algorithmique pour le parler à votre façon.

Dans la mesure où de nombreux langages algorithmiques existent (malheureusement la plupart du temps peu formalisés), nous explicitons dans ce chapitre le pourquoi et les objectifs des algorithmes et d'un langage algorithmique tourné vers l'analyse (et pas seulement vers la description). Nous en profiterons pour détailler au chapitre 1 un langage qui répond simplement et lisiblement à ces objectifs et nous l'illustrerons au travers de plusieurs exemples détaillés élémentaires au chapitre 3 puis plus conséquents au chapitre 4. Le chapitre 2 comporte des compléments qui peuvent être ignorés en première lecture mais qui assurent une vision complète de l'algorithmique.

Le but de ce chapitre n'est pas de convaincre qu'il faut utiliser notre langage (quoique...) mais d'insister sur le bien fondé de la réflexion, l'obligation d'écrire des algorithmes, d'utiliser un "bon" langage algorithmique, quitte à en inventer un dans l'esprit présent tout au long de cet ouvrage.

Une erreur classique de débutants en programmation mais aussi malheureusement de certains enseignants en informatique est de croire qu'il n'y a pas besoin de détailler les langages algorithmiques, que leur syntaxe peut rester floue. Il en résulte que la syntaxe des algorithmes change au fil des jours, que certains fustigent (voire sanctionnent) des algorithmes auquel il manque un mot, une fin de structure alors que d'autres auraient accepté l'algorithme.

Ecrire un algorithme est un travail structuré, réfléchi, qui doit obéir à des règles strictes de vocabulaire, de syntaxe et de sémantique. Le risque à trop formaliser, à trop imposer est de tomber dans un langage algorithmique trop proche d'un langage de programmation, voire de confondre le langage algorithmique avec un langage de programmation francisé, perdant à la fois souplesse, concision et lisibilité.

Entre trop de rigueur et trop peu de rigueur, il est difficile de trouver la juste mesure. Pourtant avec de la réflexion, de la mise en commun, il est possible de rendre l'écriture des algorithmes faisable, voire agréable. C'est ce que montre, nous l'espérons, cet ouvrage.

Pourquoi faut-il écrire des algorithmes ?

Ecrire un programme est un travail comparable à la rédaction d'un exposé que l'on devra traduire dans une ou plusieurs langues étrangères ; d'ailleurs un langage de programmation est relativement semblable à une "autre" langue. Chacun sait qu'il ne suffit pas de connaitre des mots pour savoir faire des phrases, pas plus que savoir lire, écrire ou maitriser un traitement de textes ne suffisent à savoir rédiger. Pour mettre en forme ses idées, un plan, un brouillon, voire plusieurs essais sont souvent nécessaires. Ces essais, ces redites, se font dans notre langue maternelle, surtout si on veut discuter de ses idées avec d'autres, si on ne sait pas très bien dans quelle(s) langue(s) l'exposé sera traduit. L'algorithmique se veut le moyen de mettre en place et en forme les idées, les actions que le programme viendra réaliser. C'est pour cela que les algorithmes sont obligatoires, ce sont des passages obligés entre les idées et les programmes.

On ne construit pas une maison sans plan, ni sans justifier ses choix : ici *une cloison de 10*, là *un mur de 30 à cause de la charpente....* Les algorithmes sont au programme ce que le plan est à la maison et on ne peut donc pas s'en passer : ils disent ce qu'il faut faire et pourquoi, ils argumentent alors que les programmes ne font qu'implémenter. Lors de l'écriture de grandes applications, les algorithmes sont la suite logique des cahiers des charges fonctionnels, la partie amont de la documentation de l'application.

De plus, un langage de programmation s'utilise dans le cadre de la communication entre l'homme et la machine, alors que les algorithmes sont faits pour la communication entre l'homme et l'homme. Et en particulier pour la communication avec soi-même dans le but de mettre au clair (et formellement) sa pensée, dans le but de produire un tout cohérent et compréhensible pour un humain, facile à traduire pour un ordinateur.

Un lecteur, une lectrice peu expérimenté(e) en programmation découvriront avec peine au fil des heures passées sur le clavier que sans algorithmes, les programmes sont difficiles à relire, à maintenir, qu'ils mélangent instructions et astuces, que les langages imposent des contraintes techniques qu'un algorithme peut ignorer (mais que la traduction en programme devra gérer).

Les vrais programmeurs, eux, savent que l'algorithme se situe à la source du travail de programmation : véritable *fédérateur* de la méthode, *cheville* entre les idées et les langages, *dénominateur commun* et premier au sens où il se situe avant tout programme.

Il arrive souvent aux programmeurs professionnels de ne pas coucher sur le papier leurs algorithmes parce qu'ils l'ont dans la tête et qu'ils écrivent directement le code correspondant, parce qu'ils n'ont pas le temps, parce que... mais ils sont tous unanimes à reconnaitre qu'ils s'exposent à de grands problèmes de relecture, de maintenance, de reprise, surtout si ce sont d'autres personnes qui doivent continuer le programme.

Pourquoi n'écrit-on pas des algorithmes ?

Il y deux raisons fondamentales pour ne pas écrire d'algorithmes : le manque de temps et l'obligation d'écrire des programmes avec des interfaces. Le manque de temps est un mal des sociétés dites "à fort développement" : il faut faire vite, il faut aller loin, ne serait-ce que pour aller chercher le pain avec sa voiture, au risque d'aller à un rythme qui n'est pas ni naturel ni sain... Sans entrer dans un débat sociologique, économique ou écologique, les raisons qui poussent à écrire des algorithmes doivent dépasser ce manque qui est souvent de la fainéantise déguisée, voire de la paresse quand ce n'est pas de la négligence et de la fuite systématiques.

L'obligation d'écrire des programmes avec des interfaces est un argument plus sérieux : un algorithme s'accorde mal avec la description de menus, de boutons, labels et autres composants graphiques de l'interface utilisateur. Heureusement, les environnements de développement fournissent des moyens visuels de mettre en place ces interfaces. Il reste que derrière ces composants il y a des programmes qui s'exécutent et ce sont ces programmes (ou plutot ces sous-programmes) que les algorithmes mettent en place.

Il est possible que dans un futur proche, il n'y ait plus besoin d'écrire de programmes : on se contentera de cliquer dans des menus les actions à effectuer et l'environnement se chargera, comme pour l'interface graphique, d'écrire le code.

Si on peut regretter qu'il faille aujourd'hui quand même écrire des programmes, on pourra apprécier le fait que le premier jet de la traduction d'algorithmes bien écrits en programmes est facile : on pourra par exemple utiliser notre programme galg, disponible à l'adresse

http ://www.info.univ-angers.fr/pub/gh/Galg/

qui analyse les algorithmes, les traduit en Rexx, C, C++, Java... et permet de les exécuter dans la foulée via ces langages.

Choix d'un langage algorithmique

Pour bien parler une langue, il faut non seulement connaitre le **vocabulaire**, c'est à dire les mots mais aussi la **syntaxe** c'est à dire l'ordre dans lequel on met les mots. Enfin, il faut maitriser la **sémantique** c'est à dire le sens des mots. Pour la programmation, l'algorithmique, c'est la même chose, d'où cet ouvrage pour aider à bien écrire les algorithmes avant de les traduire en programmes.

Un langage d'analyse algorithmique n'existe que par son but ; celui-ci peut être d'apprendre à raisonner "formellement", de détailler un cheminement de la pensée ou de commander "mécaniquement" la machine pour effectuer un certain travail. Ces trois buts n'entrainent pas les mêmes contraintes. Ainsi pour débuter, certains mots ("instructions") peuvent être omis, simplifiés ou pré-supposés, certaines fonctions prédéfinies ou interdites : par exemple on peut décider que date() affiche la date du jour, que heure() affiche l'heure courante, sans se soucier de comment cela se dit vraiment dans un langage de programmtion.

Par contre, pour commander la machine si on veut obtenir un programme opérationnel, il faut tout (ou presque tout) détailler, savoir quel langage de programmation (éventuellement quelle version) on utilisera de façon à ce que la traduction soit cohérente. Par exemple, si on écrit sousChaine(phrase,1,5) dans un algorithme pour extraire les 5 premiers caractères de la phrase mais que le programme utilise substr(phrase,0,5) parce que dans le langage on commence à compter à partir de zéro, il y a conflit. On peut s'en sortir avec une instruction algorithmique sousChaineZero(phrase,1,5) au prix d'une complication qu'on réservera aux algorithmes avancés.

En fonction du but recherché, le langage algorithmique sera donc restreint ou étendu, partiellement inhibé ou volontairement flou, très descriptif ou globalisant.

Utiliser un seul langage pour réaliser ces trois buts est un pari difficile à tenir si ce langage n'est pas capable de faire face à toutes ces exigences. Il faudra donc "moduler" le langage à partir d'un ensemble cohérent d'instructions qu'on apppelera la *base du langage algorithmique*. Utiliser cette base pour débuter, pour exprimer le cheminement de la pensée se révèlera très pratique mais insuffisant dès qu'on voudra entrer dans les détails, aller vers la résolution sachant le langage de programmation retenu. Il faudra donc progressivement rajouter des mots, des instructions, aller vers plus de rigidité.

Choisir dès le départ trop de mots, ou un langage trop rigide (avec trop de mots réservés) ne permettra pas de passer facilement d'une structure (de données par exemple ou d'appels de modules) à une autre et viendra donner l'illusion de la difficulté, en confondant *ce qui est difficile* avec *ce qui est complexe* et avec *ce qui se dit avec des mots techniques.*

La caractéristique fondamentale d'un langage d'analyse algorithmique est sans aucun doute la **clarté** ce qui implique la non-ambiguité. Associée à la **modularité** et à la **souplesse**, la clarté rend possible le suivi, la maintenance, actions toutes deux obligatoires dès que les algorithmes et leurs sous-algorithmes dépassent plusieurs centaines de lignes (car un "bon" algorithme est certainement un algorithme court).

Une autre caractéristique qui devrait être facile à réaliser est la précision des termes, la non ambiguité sémantique (quoique celle-ci est fortement liée à la souplesse). Ainsi la notion de point virgule comme séparateur d'instructions ou comme terminateur d'instruction (distinction que de nombreux programmeurs maitrisent encore très mal) ne doit pas figurer car la finesse, la précision d'une telle notion est obtenue au détriment de la clarté.

C'est pourquoi un algorithme correct n'a pas *a priori* besoin de différencier les modes de lecture au clavier comme `LIRE_ENTIER` et `LIRE_REEL`, ni de s'occuper de la précision de l'affichage... ce qui n'interdit pas de mettre des indications quant au formatage en commentaires ou d'utiliser une fonction `FORMAT` pour rappeler qu'il faudra veiller à la qualité de l'affichage.

Par contre, si on désire utiliser des piles, des queues, des appels récursifs, des fichiers en accès direct, des bases de données... il ne faut pas avoir recours à une partie spécifique du langage (ce qui nécessiterait l'évolution constante du langage algorithmique, de ses restrictions, de son vocabulaire et en ferait l'union inutilisable de tous les langages) mais utiliser des modules (sous-programmes, fonctions...) construits sur la syntaxe de base du langage, inventer des TDA (types de données abstraits) dont on précise la syntaxe, ce que les langages de programmation objets et à compilation séparable nomment d'ailleurs "interface" puisque pour eux l'interface n'est pas dans l'interaction clavier, souris ou écran mais bien dans la communication entre modules.

Ainsi, le nom des colonnes des tableaux rectangulaires qu'on nomme base de données est accessible par la fonction *FIELD()* dans le langage Dbase. On pourra raisonnablement inventer ou présupposer une fonction `CHAMP()` en algorithmique, si le besoin s'en fait sentir ou supposer existant un tableau (global) `NOMCOL` tel que `NOMCOL[n]` renvoie le n-ième nom de colonne.

Un langage algorithmique doit rester ouvert, (voire modifiable), par le jeu des appels de modules, même si sa structure minimale de base (commentaires, affectations, tests en si, boucles pour et tant que...) reste figée. Un langage algorithmique trop calqué sur un langage existant (Pascal, notamment) sera trop fortement orienté vers un type de solution, ne permettra pas la souplesse qu'offrent les autres langages.

Les caractéristiques **clarté**, **modularité** et **souplesse** ne sont pas entre elles complètement compatibles et un langage algorithmique ne peut pas être parfait (car sinon on n'aurait pas besoin de langages de programmation), mais l'humain qui le lit est capable de pallier les imperfections **qui ne sont pas des erreurs** quand l'algorithme est clair, non ambigu.

Enfin, un dernier point concernant le soin qui doit être apporté à la rédaction, quelque soit le langage algorithmique considéré : la documentation de l'algorithme, comme celle du programme est un véritable **document**, donc un texte bien présenté, bien commenté, avec, s'il le faut, des spécifications techniques, des restrictions, un manuel d'utlisation, des exemples d'entrées, des exemples de sorties, des références bibliographiques ou des renvois à des ouvrages expliquant la méthode, justifiant ou démontrant les formules...

La *base du langage* que nous avons choisi d'utiliser est d'abord volontairement simple et concise, car son but est de montrer l'enchainement rigoureux des actions dégagées par l'analyse. Nous décrivons dans le chapitre 2 rapidement son **vocabulaire** et sa **syntaxe** tout en détaillant ce qu'il fait ou ce qu'il ne fait pas et pourquoi, toujours dans un but pédagogique de clarté, de lisibilité et de relecture.

Nous savons par expérience que la maintenance d'un (gros) programme est toujours plus lourde et délicate que le premier jet, que l'impatience dans la mise au point et la mise à jour est l'ennemie principale du développeur, comme s'il fallait programmer vite parce que l'ordinateur, lui, va vite ! C'est pourquoi les algorithmes doivent être simples et lisibles, pour que le programmeur puisse d'y référer, laissant le détail de l'interface utilisateur à l'environnement de développement. Un algorithme, un programme bien documentés qui comportent des copies d'écran, les tables de correspondances entre les options des menus et le nom des sous-programmes associés, des jeux d'essais, le détail des parties techniques sont des vrais régals comparés à l'affichage brut et stupide des lignes du code constituant le seul programme.

A cette base de langage on doit certainement ajouter des instructions, des fonctions pour obtenir un langage utilisable. Chacun, chacune y rajoutera ce qui lui semble bon. C'est pourquoi le titre de ce manuel est Algorithmiques Raisonnnées au pluriel.

Notre langage est presque instantanément compilable. Même s'il suppose une compréhension intelligente (heureusement "évidente" et "intuitive") du cadrage de la part du lecteur, il est aisément et facilement traduisible dans n'importe quel langage procédural (dans le genre de Java, Pascal, Perl ou C). On pourra s'en convaincre grace au programme galg qui analyse les algorithmes, les traduit en Rexx, C, C++, Java... et permet de les exécuter dans la foulée via ces langages. Ce programme est accessible sur le *Web* à l'adresse http ://www.info.univ-angers.fr/pub/gh/Galg/.

On trouvera également sur ce site tous les algorithmes de cet ouvrage ainsi que les légères modifications obligatoires pour en faire des algorithmes traduisibles en programmes par galg.

Pour des langages déclaratifs de type Prolog qui se prêtent mal à ce genre d'écriture, pour des langages minimalistes comme Apl ou pour des langages "tout objet" comme SmallTalk, nous conseillons de détailler les méthodes jusqu'à obtenir des enchaînements de modules que l'unification exécutera, que les symboles ou les objets implémenteront, d'écrire en détail tous les modules récursifs et de présenter rigoureusement leur enchainement. On pourra à ce sujet consulter le chapitre 3.

La simplicité de notre langage et le refus de toute construction syntaxique spéciale et sophistiquée en font un support de description rapide mais formel et rigoureux de tout algorithme traditionnel, de toute méthode détaillée. Il reste près de la méthode sans introduire d'arbitraire trop calqué sur un langage précis, même si comme nous l'avons dit, il n'est pas complètement adapté aux langages comme Apl, Prolog, Smalltalk, ou même, dans une certaine mesure, à Lisp pour lesquels une méthode détaillée se traduit morceau par morceau, fonction par fonction, objet par objet et où la récursivité et où dans une certaine mesure les structures de données font partie de la résolution.

Ce ne sera pas vraiment un problème puisque le programmeur saura s'adapter, reprendre ce qui ne va pas, ré-écrire les algorithmes dans une optique plus "lispienne", plus "apliste" etc. avant de recourir à ces langages. Dans le même genre d'idées, une fois l'algorithme validé, il faudra peut-être utiliser les ressources d'un langage particulier pour optimiser son exécution.

La présentation de notre langage algorithmique que l'on trouvera dans le chapitre suivant reprend l'ordre classique de présentation pour un langage : commentaires, affectations, entrées/sorties, contrôles du flot (tests et boucles), structures de données (tableaux) et de modules (sous-programmes). même si cet ordre n'est pas celui de la petite école : on traite calculer (affecter) avant lire/écrire dans les petites classes, ce qui montre bien la différence entre langage pour machine et langue humaine.

La description de notre langage, les quelques dizaines d'exemples d'algorithmes disséminés ça et là ne suffiront certainement pas au lecteur inexpérimenté pour savoir écrire des algorithmes, pas plus que la description des mouvements à effectuer pour faire du vélo ne permettent de monter sur la selle et de commencer à avancer. Notre but est plus d'insister sur l'*esprit algorithmique* que sur la syntaxe algorithmique, de toutes façons obligatoire.

Comme pour d'autres arts, c'est à force de pratique que le lecteur, la lectrice seront imprégnés de ce qu'il faut faire ou de ce qu'il vaudrait mieux faire et surtout pourquoi on le fait. Les exemples du chapitre 3 par exemple seront donc à faire et à refaire, à réécrire, voire à être discutés.

Pour terminer ce chapitre, voici quelques exemples de ce qu'un langage algorithmique cherche à éviter comme seul et unique document de programmation, mais qu'il peut aider à produire, à condition bien sûr d'utiliser << *forces commentaires et moultes progressions* >> qui doivent être dans le document algorithmique. Signalons au passage que ces exemples sont tirés de programmes réels trouvés sur le *Net* :

- une ligne d'APL "sèche"

  $$((\mathrm{LNG}\leftarrow 7,)\mathrm{PREC})\,\overline{\Phi}\,(\mathrm{I}\circ.\mathrm{FC}\ \mathrm{I}\leftarrow \iota\mathrm{NBC}\leftarrow 1\downarrow\rho\mathrm{S})\div 1\uparrow\mathrm{MC}$$

- une expression PERL efficace mais non expliquée

```
push(@n, $+) while $t =~ m{
"([^\"\\]*(?:\\.[^\"\\]*)*)",?|  ([^,]+),?| , }gx;
```

- des constructions C justes mais confuses

```
for (tk = e->edmdr; tk != NULL && newnb >= tk->tifudst;
tk = tk->tifudnxt) ed = tk;
if (NULL != exectifx(tokhdr + tokcnt - 3,R_B,0)) {
e->lurid = tk;  tifudnw(e,newnb,cp,stmtlen,cpn,tokcnt-3);
rrn; }
```

- un message SMALLTALK incompréhensible hors-contexte

```
big_Sf at: valnoK put:
   (big_Sf at: valnoK) + 1 +  (small_S at: oldvalI) asMinCnt
```

- du PASCAL qui "marche" mais illisible

```
a  := init(-lambda) ; n :=0 ; u : = recalc() ;
while u>a do begin u :=u*recalc() ; n := n+1 ; end;
etvoila := log(n) ;
end;
```

- une procédure TCL un peu technique non commentée

```
proc testip {ip} {
  set tmp [split $ip .]
  if {[llength $tmp] != 4} then {
    return 0
  }
  set index 0
  foreach i $tmp {
    if {((![regexp \[^0-9\] $i])
      || ([string length $i] > 3)
      || (($index == 3) && (($i > 254)
      || ($i < 1)))
      || (($index <= 2) && (($i > 255)
      || ($i < 0))))} then {
      return 0
    }
    incr index
  }
  return 1
}
```

Chapitre 1.

Anatomies des langages algorithmiques

1.1 Les Commentaires

Un commentaire est un texte (tout ou partie d'une ligne, ou plusieurs lignes) écrit dans l'algorithme. Le but d'un commentaire est de d'expliciter le texte de l'algorithme sans le paraphraser et on pourra donc admettre un certain flou quand à l'écriture, l'indentation des commentaires. Dans notre langage, les commentaires n'ont pas de symbole défini, mais # ou * en début de ligne (comme en Awk et Dbase respectivement) ou les symboles de Pascal ou C peuvent être utilisés.

Il peut être intéressant de commenter tout une partie de texte, avec un symbole de début de commentaire et un symbole de fin de commentaire. Du coup, on peut utiliser { et } pour les titres, # pour numéroter les actions s'il le faut, (* et *) et /* ou */ pour les commentaires "locaux" (c'est à dire moins importants) ce qui autorise un éditeur ou à un traitement de texte évolué à masquer les lignes qui ne sont pas des commentaires, à retrouver le plan, à n'afficher que les commentaires principaux etc.

Le refus de choisir un symbole particulier comme marque de commentaire doit rassurer le débutant, de lui laisser le choix, de l'attirer par une marque comme ♠ ou même comme . Cela permet aussi d'expliquer que l'important est de repérer le commentaire, de savoir ce qu'il indique, d'induire une certaine souplesse dans le processus d'écriture.

On pourra d'ailleurs remarquer que dans ce chapitre nous avons fait en sorte de varier les symboles de commentaire, sans que cela choque ou soit illisible.

Nous insistons sur le fait de taper au clavier les algorithmes car il est plus agréable de recopier des blocs de ligne, d'automatiser une même syntaxe avec un ordinateur. Le choix des symboles sera donc celui d'une touche ou d'une suite de touches faciles à produire, mais des symboles spéciaux (voire une couleur différente) sont faciles à mettre en oeuvre à la souris, avec une touche de fonction, une barre d'outils, avec un menu déroulant...

Ecrire un algorithme avec l'ordinateur ne dispense pas de griffonner au brouillon le début de l'algorithme, de préparer un exemple de données, de réfléchir sans écrire aux différents cas possibles. Mais passer des idées à un texte imprimable oblige déjà à un effort de rédaction, de présentation.

Le choix du nombre de commentaires (ni trop, ni trop peu) est un problème délicat, de même que la rédaction à l'intérieur des commentaires. Il faut rappeler que les commentaires constituent une documentation interne, qui ne se substitue ni à la description de la méthode, ni aux formules de résolutions, que leur but est d'éclairer l'écriture formelle des actions, d'aider à détailler et à nommer ce qui est fait. Mais ce choix est pourtant facile à mettre en évidence : si on ne garde que les commentaires de l'algorithme (ce qui se fait facilement avec de nombreux éditeurs de texte, ou avec des commandes comme *grep*) on doit obtenir un document lisible, compréhensible qui permet de suivre les différentes parties de l'algorithme.

Cette règle n'est bien sûr pas informatisable puisqu'elle suppose une capacité à juger de la lisibilité, de l'expressivité des phrases de commentaires mais elle montre bien l'esprit des commentaires, comme plus tard le choix du nom des variables : il s'agit, hors instructions, de donner un tout lisible, cohérent, facile à suivre.

Pour écrire un algorithme, la meilleure façon de mettre des commentaires est de commencer par écrire tous les commentaires principaux, puis de mettre les instructions algorithmiques entre les commentaires et enfin de rajouter des commentaires lorsque le code est délicat ou qu'il résulte de choix, de suppositions, d'astuces... C'est la seule façon d'assurer que les commentaires seront écrits, et qu'ils le seront à la bonne place. Rappelons d'ailleurs que pour justifier des mauvaises pratiques, à l'adage *"j'écris le programme aujourd'hui ; quand il sera au point, j'écrirais l'algorithme"*, on peut substituer l'adage *"j'écris les instructions aujourd'hui ; j'écrirai les commentaires demain"*.

De telles mauvaises pratiques couvrent plus de 80 % des cas, sans parler des un-peu-moins-fainéants qui écrivent un algorithme, le traduisent en programme puis modifient le programme sans mettre à jour l'algorithme !

Par exemple, l'algorithme suivant qui affiche de façon bien cadrée la table de multiplication d'un nombre entré au clavier et vérifié comme entier peut être :

```
#########################################################
#                                                       #
# tabmult.alg -- la table de multiplication             #
#                                        auteur : gh    #
#                                                       #
#########################################################

#-# 1. demande initiale

écrire  " Donner un entier "
lire nbChoisi

#-# 2. relance éventuelle

tant_que (non entier(nbChoisi))
    écrire " nombre incorrect. Redonner un nombre ENTIER "
    lire nbChoisi
fin_tant_que # nombre invalide

#-# 3. boucle d'affichage bien cadré

écrire " Table de " , nbChoisi
pour indb de1a 10
    affecter produit <-- nbChoisi*indb
    affecter find    <-- format(indb,2,0)
    affecter fpro    <-- format(produit,5,0)
    écrire find , " fois " , nbChoisi , " = " , fpro
fin_pour # indb de1a 10
```

Toute ligne qui commence par `#` est un commentaire. Astucieusement, on a mis `-#` pour repérer les parties importantes. Si on ne garde que les lignes qui contiennent `#-#` on voit donc :

```
#-# 1. demande initiale
#-# 2. relance éventuelle
#-# 3. boucle d'affichage bien cadré
```

ce qui fournit les grandes étapes de l'algorithme que l'on peut alors documenter...

1.2 Les Variables

Une variable est la désignation d'un "endroit" pour contenir une valeur ou plusieurs valeurs. On parle de variable indicée ou de variable-tableau ou plus simplement de *tableau* quand plusieurs valeurs sont associées à un même nom. Nous ne traitons ici que les valeurs simples (on dit souvent variable *scalaire*) car nous traiterons les tableaux plus loin.

Une variable a un nom et on confond dans l'écriture des algorithmes le nom et la valeur contenue dans la variable (ce qui n'est pas le cas pour tous les langages de programmation, donc attention à la traduction), de même que les majuscules et minuscules : `NBVAL` et `NbVal` correspondent donc à la même variable (ce qui, là encore, n'est pas le cas pour tous les langages de programmation, donc attention là encore à la traduction). Une variable peut contenir du texte (nommé traditionnellement *chaine de caractères*) ou une valeur numérique. Un texte est entouré de guillemets droits (*"..."*), penchés (≪ ... ≫) ou encore d'apostrophes (*'...'*). On distingue donc la valeur numérique `6` du texte `"6"`, la variable de nom `IND` du texte `"IND"`.

En algorithmique élémentaire, il n'y a pas besoin de déclarer les variables, c'est à dire de leur donner un *type de données*. L'instruction `N` ← `1` qui utilise le symbole d'affectation que nous verrons à la section suivante indique clairement que `N` est un nombre et que la variable correspondante vaut 1. Intuitivement, la valeur `1` est un nombre, sans qu'il soit besoin de préciser que ce nombre est entier, réel, complexe ou décimal (avec une partie fractionnaire nulle). Déclarer une variable par un type c'est présupposer que le langage est typé, or tous les langages ne sont pas typés. Spécifier que `N` est entier ou réel introduit une distinction parfois superflue ou inconnue du langage (Awk, Rexx ainsi qu'Apl par exemple sont non typés explictement).

Typer une variable peut avoir des résultats inattendus et dangereux pour l'exécution du programme. Ainsi `123456789` n'est en général pas considéré comme un "entier" par de nombreux compilateurs car les entiers informatiques sont souvent limités à une puissance de 2 moins un, par exemple limités à `32767` soit $2^{15}-1$, à `65535` soit $2^{16}-1$ ou même à `2147483647` soit $2^{31}-1$. Certains versions de langages (dont C et Pascal) vont même jusqu'à attribuer la valeur `-32768` (oui, c'est bien un chiffre négatif!) comme résultat de l'expression `32767+1` à cause de la représentation en machine sur 16 bits dont 1 bit de signe. Par contre, il n'y a aucune raison de supposer en algorithmique que les mathématiques "normales" ne s'appliquent pas, et donc en algorithmique, `32767+1` doit toujours faire `32768`.

La déclaration des variables est sans doute utile pour un langage de programmation. Elle permet la détection de nombreuses erreurs, notamment au niveau du passage des paramètres. De plus elle précise la taille mémoire à utiliser, ce qui est important pour la machine ; par contre, elle n'apporte rien à la résolution et donc à l'algorithme.

Plus que la déclaration, l'important est la **désignation** des variables ; la spécification éventuelle de leur plage de variation, voire de leur type ne doit intervenir qu'après, au cours de la traduction dans un langage (si tant est qu'elles puissent être connues, à défaut d'être estimées ou prévues).

A défaut de nom explicite, les variables doivent être commentées car le sens, l'utilisation des variables définit implicitement leur type. On trouvera donc des indications comme

```
DernP ← 1 # indice du DERNier nombre Positif
```

où les majuscules dans le commentaire indiquent la formation de l'identificateur, à moins qu'une liste de variables soit indiquée en début d'algorithme, comme :

```
#
#     on utilise une liste notée LT contenant NBL éléments ;
#     chaque élément se compose d'un identificateur ID_NBL
#     et d'une valeur quantitative QT_NBL ;
#     enfin, ST_QT désigne la somme des valeurs quantitatives.
#
```

Attention toutefois aux identificateurs de variables qui comportent des lettres accentuées : cela risque de poser un problème de traduction pour la plupart des langages de programmation.

Dans le même esprit d'enseignement des problèmes de traduction, il faut aussi noter que certains identificateurs correspondent à des mots réservés pour certains langages, comme `IN`, `LST` etc. pour le Pascal. Pour programmer "parfaitement" il faudrait connaitre tous les mots réservés de tous les langages, pour ne pas courir le risque de "coincer" au niveau d'une traduction. Ce n'est pas une mince affaire car certains langages ont jusqu'à plusieurs dizaines de mots réservés.

Une pratique courante est d'utiliser au moins 4 ou 5 caractères par variable. Rien n'interdit par contre de saisir des identicateurs courts au clavier, `I` par exemple, puis, une fois l'algorithme terminé, d'utiliser la fonction *Remplacer* des traitements de texte pour changer tous les `I` en `IndB` si `I` désigne un indice de boucle. Ce qui signifie qu'il faut certainement aérer le texte saisi pour ne pas remplacer tous les autres `I` du texte. Une habitude à prendre est notamment d'entourer les variables par des espaces (notés ici ␣) et donc d'écrire ␣`I`␣`+`␣`1`␣ plutôt que ␣`I+1`␣.

1.3 Les Nommages et les Affectations

Le nommage se fait par le symbole ⟵ et suit la syntaxe :

$$[\ un\_nom\_de\_variable\ ] \leftarrow [\ une\_expression\_calculable\ ]$$

Son but est d'effectuer un calcul ou d'évaluer une expression et de mettre le résultat du calcul, de l'expression dans la variable indiquée. L'affectation suit la même syntaxe mais autorise la même variable à apparaitre à droite et à gauche du symbole ←. La différence entre un nommage et une affectation est importante : un nommage ressemble à une opération "comme en mathématiques" et attribue une valeur unique à une variable, valeur immuable pour tout l'algorithme. Une affectation qui n'est pas un nommage vient perdre cette "transparence référencielle" et ne doit donc être utilisée qu'avec parcimonie. D'ailleurs, seuls les langages impératifs implémentent la notion d'affectation. Les autres langages mettent en place des équivalents de nommage : ainsi le `let` du Lisp, l'instanciation d'une variable libre avec `is` de Prolog ressemblent à peu près au nommage.

Un nommage est une instruction qui a un "sens directionnel" et ce sens est donné explicitement par la flèche. Nous préférons ce symbole au mot `PPV` (abréviation de *"Prend Pour Valeur"*) qui nous parait moins évocateur. Le symbole `=` ne peut être utilisé pour l'affectation car il est souvent réservé au test ; de plus il ressemble trop au symbole `=` des mathématiques (qui lui, commutatif, n'a pas sa place ici). De nombreux langages font aussi cette distinction mais ← et `=` sont plus explicites que `:=` et `=` ou que `=` et `==`. La flèche fait bien ressortir le coté dyssymétrique de l'opération ; elle explicite bien le fait que `3` ← `X` est impossible, ce qui est moins évident avec `3 = X`.

Un nommage ne doit pas être multiple, de façon à favoriser l'association d'**une** valeur et d'**une** variable. L'instruction algorithmique `A ← B ← 0` est donc inacceptable et ne sert qu'aux gens pressés. De même pour l'instruction `A , B ← 0 , 1` qui signifie sans doute `A ← 0` suivi de `B ← 1`.

Pour ceux et celles qui savent ce qu'est une variable indicée (un "*tableau*"), on ne peut pas se servir de l'affectation ou du nommage sur le nom du tableau pour l'initialiser car un tableau ne peut être affecté que localement par indice. En d'autres termes, si `T` est un tableau, l'instruction `T ← 0` pour initialiser tous les éléments du tableau `T`est incorrect. On peut par contre inventer un module `Initialiser(T,0)` si on veut une action générale. Cela renforcera l'idée que tout est possible à écrire à l'aide sous-programmes (comme modifier tout un tableau en une seule instruction) mais que chaque symbole a un seul sens. Certains langages, dont Apl, ont jusqu'à 8 sens pour un même symbole — ce qu'on appelle la surcharge des symboles, des opérateurs — mais ce qui est acceptable pour un langage de développement (et pratique, concis) ne l'est pas pour un langage d'analyse algorithmique.

Supposer que le langage algorithmique initialise de lui-même chaque variable à 0 est une erreur. Les langages de programmation Awk et Rexx initialisent dynamiquement, mais bien sûr pas avec ce que l'on pense : Awk initialise avec "" (la chaine vide) sauf quand le contexte est explicitement numérique auquel cas il initialise à `0` et Rexx initialise avec le nom de la variable en majuscules (ce qui permet de voir rapidement une faute de frappe sur le nom des variables).

Nous avons employé le terme "expression calculable" pour désigner le résultat du nommage ou de l'affectation ; nous aurions pu mettre "terme à résultat numérique ou caractère". Ces termes sont suffisamment vagues dans un premier temps pour qu'on puisse écrire des instructions comme

```
ValeurSuivantMajoration ← AppliqueCodeTva(CoutHorsTaxe,2)
```

ou comme

```
PmEm ← Majuscules( Mot(Phrase,IndM) )
```

Souvent, une expression calculable sera définie par des fonctions ou *modules* non encore explicités : on peut écrire par exemple `Racine(2)` pour sous entendre un module "racine carrée", par exemple. En principe, on indique en commentaire dans le programme principal quels modules on utilise. Dans un premier temps, on peut utiliser tous les mots possibles, comme si tout était déjà écrit quelquepart.

Une expression numériquement calculable se doit d'être simple et bien parenthésée. C'est déjà un problème que de choisir les noms de variables, leur degré de lisibilité maximal sans avoir aussi à réfléchir sur le sens des opérations. L'expression `A/B/C` sera donc proscrite au profit de `A/(B*C)` ou sera prosaiquement remplacée par `(A/B)/C` en fonction de ce qu'on voulait dire (même si toutes ces expressions calculent la même valeur).

Sur le clavier, le symbole ← peut être obtenu par la juxtaposition des caractères `<` et `-`, quitte à doubler ce denier pour obtenir une flèche plus longue comme `<--`.

On encouragera aussi l'écriture systématique de variable de transfert pour favoriser l'optimisation. Par exemple, les deux instructions

```
IndFin       ←   2*Indice+1
ValeurAlors  ←   T[IndFin]
```

seront utilisées plutôt que la seule ligne `ValeurAlors ←T[2*Indice+1]` dès que l'expression `2*Indice+1` apparaîtra plus d'une fois dans le même corps d'instruction.

Insistons sur un point : une affectation, comme une expression calculable, et plus généralement un algorithme doit pouvoir s'écrire avec un ordinateur.

```
                NUM
FRAC  6 +  --------
                DEN
```

est donc invalide (*"algorithmiquement incorrect"*) et doit être remplacée par l'expression `FRAC ← 6 + NUM / DEN` ou par `FRAC ← 6 + NUM÷DEN`.

De même, on évitera d'utiliser le symbole $\sqrt{}$ pour désigner la racine carrée d'un nombre ; on écrira donc `racine(expr)` plutôt que $\sqrt{expr}$. Dans le même genre d'idées, on remplacera x^y par $puissance(x, y)$, e^x par $exp(x)$, α par alpha etc.

Lors de l'affectation de texte, le choix de plusieurs jeux de symboles pour les *chaines de caractères* (mots et textes) évite le problème classique de se demander comment mettre ' dans une chaine si ' sert à délimiter la chaine. Surtout, on ne fera pas de distinction subtile entre 'X' et "X" (comme en C) pour désigner le contenu d'une variable de type caractère ou de type chaîne de caractère à 1 caractère.

Avec seulement deux symboles, celui de commentaire et celui d'affectation, on peut déjà écrire des milliers d'algorithmes. Ils n'afficheront rien mais calculent quand même quelquechose. C'est d'ailleurs souvent le cas de nombreux algorithmes : la partie lecture des données et écriture des résultats est ignorée, car confiée à un autre algorithme. Par exemple, l'algorithme

```
nbAnBi          <-- 500  # nombre d'années bissextiles (2000/4)
nbJenJ          <--  31  # nombre de jours en janvier
nbJours         <-- (2000*365) + nbAnBi + nbJenJ
nbHeur          <-- nbJours * 24
nbSec           <-- (nbHeur*60)* 60
```

calcule, au premier février 2001, le nombre de secondes écoulées depuis la naissance (supposée) du Christ, soit à peu près 63117878400 secondes (6,3 milliards) en admettant qu'il y eu une année bissextile tous les 4 ans et en oubliant qu'il y eut en France une nuit du 9 au 20 décembre 1582 (ce qui ne s'invente pas).

Mais on peut aussi jouer avec les affectations comme par exemple avec l'agorithme suivant, qui, pour reprendre de mauvais souvenirs de lycée (pour certains), calcule les solutions de l'équation du second degré $ax^2+bx+c=0$ avec a et c de signe contraire :

```
# résolution de l'équation classique du second degré
# (on suppose donc a non nul)

delta   <--   b*b - 4*a*c
rd      <--   racine( delta )
s1      <--   ( - b - rd ) / ( 2*a )
s2      <--      (c/a) / s1
# meilleur que   s2  <--  ( -b + rd ) / ( 2*a )
#       et que   s2  <--  ( b/a ) - s1
# mais seulement pour b négatif
```

ou utiliser un algorithme non simple pour permuter le contenu de deux variables numériques comme :

```
# permutation du contenu de a et b
# a et b doivent être numériques
# cet algorithme est moins bon que les instructions
# à droite des signes ##-##
     a  <--  a + b              ##-##    c  <--   a
     b  <--  a - b              ##-##    a  <--   b
     a  <--  a - b              ##-##    b  <--   c
```

L'affectation `Ok ←(X=2)` est tout à fait valide. Elle signifie que la variable `Ok` est vraie si `X` vaut 2 et fausse sinon. Ce genre d'affectation sert à éviter de refaire des tests, à simplifier l'algorithme (tant que `X` ne change pas de valeur). De même, l'affectation `X ← X + 1` est valide (en mode impératif seulement) et signifie qu'on rajoute 1 à la variable `X`. Le détail de l'affection `Var ← Expr` se fait en calculant la valeur de `Expr` puis en mettant dans `Var` le résultat de l'expression. On voit bien sur ces deux dernières affectations toute l'ambiguité qu'il y aurait à utiliser le même symbole `=` pour faire à la fois le test et l'affectation.

1.4 Les Entrées/Sorties

Les entrées et les sorties sont ici celles qui gèrent l'interaction avec l'utiisateur, à savoir la sortie à l'écran et l'entrée au clavier, avec les actions correspondantes de lecture et écriture. Pour ce qui est des entrées et sorties sur imprimante, sur fichier, nous renvoyons à la section **1.9**. Nous ignorons aussi volontairement les entrées souris, code-barre et autres subtilités technologiques qui ne sont toutefois pas à négliger lors de la traduction des algorithmes en programmes.

Le coeur de l'algorithmique est de dégager les "fonctions" c'est à dire les *modes opératoires* qui passent des variables données en entrée $a, b, c...$ aux variables produites en sorties $x, y, z...$, que ces variables soient des "scalaires" c'est à dire des valeurs simples ou des structures (comme les tableaux, les arbres, les objets). L'algorithmique théorique n'a donc pas besoin d'instructions comme `LIRE` ou `ECRIRE` car elle précise en commentaire quelles variables sont utilisées, quelles paramètres sont requis. Toutefois, les instructions `LIRE` et `ECRIRE` sont suffisamment simples et sans pièges pour trouver une (petite) place ici, ce qui permet d'écrire des algorithmes en interaction avec l'utilisateur avec une traduction simple dans un langage de programmation.

1.4.1 Les Sorties (écran)

Pour écrire sur l'écran, on utilise le mot `ECRIRE`. L'instruction `ECRIRE NbVal` affiche le contenu de la variable `NbVAL` à l'écran. `ECRIRE` est notre premier mot algorithmique. **Il est en français**. Nous insistons pour que les algorithmes soient écrits dans la langue maternelle du programmeur (le mot "algorithmeur" ne semble pas exister).

Seuls les anglicistes en seront peut-être gênés car les algorithmes ressembleront alors trop aux programmes (qui, sauf pour Apl, utilisent des mots anglais). S'il y a plusieurs termes à écrire, ces différents termes seront séparés par des virgules. Un exemple d'affichage est par exemple

```
ECRIRE " il vous reste  : " , MinSec , " seconde(s) "
```

On pourra, si besoin est, terminer la spécification des termes à écrire sur plusieurs lignes en laissant la virgule de séparation en fin de ligne, comme suit

```
ECRIRE " au bout de " , NbEss , " essai(s), il reste  : " ,
       MinRst " minute(s) "
```

Ce que renvoie exactement la machine (espaces blancs entre variables, spécification de format de sortie) est volontairement flou. On peut toutefois indiquer en commentaire qu'il faudra prévoir un cadrage particulier, utiliser un exemple de sortie, rappeler que les décimales doivent être alignées, etc. Par exemple :

```
# il faudra donner le temps moyen arrondi
  ECRIRE " Moyenne : "  , TempsMoy
# et le temps total avec 2 decimales
  ECRIRE " Temps total : " FORMAT(TempsTotal,5,2)
```

Il n'y a pas lieu de distinguer écriture et écriture avec passage à la ligne (comme `Write` et `Writeln` du Pascal, `printf` et `print("\n")` du C, etc.) en algorithmique élémentaire. Voir le chapitre 2 si cette distinction fait partie du problème posé.

1.4.2 Les Entrées (clavier)

Le plus simple est de n'avoir qu'une seule instruction pour gérer toutes les entrées, par exemple : `LIRE`. Contrairement à une instruction d'un langage de programmation comme Pascal ou C, cette instruction accepte "tout et n'importe quoi". Comme notre langage algorithmique n'existe pas en tant que langage réel sur machine, cette absence de restriction n'est pas gênante. Au niveau des entrées clavier, `LIRE A,B` est inacceptable car les demandes doivent suivre le schéma

" *une demande = une question + une réponse ==> une variable* "

`Lire A` est donc inacceptable aussi quand l'instruction précédente n'est pas l'écriture de la question demandant la valeur de `A` (et d'ailleurs un identificateur a une seule lettre, ce n'est pas suffisant). Demander la valeur de `A` sans dire à quoi correspond `A` est bien sur stupide, sauf si on décrit les variables. Ainsi

```
Ecrire " Etudes des champignons. Que vaut NbTest ? "
Lire   Nbtest
```

est maladroit. Par contre,

```
Ecrire " Etude des valeurs de... sur l'intervalle [a,b]."
Ecrire " que vaut a ? "
Lire   borneA
Ecrire " et b ? "
Lire   borneB
```

est acceptable. Au pire, on peut prévoir une module fictif qui lit les données et qui les vérifie. Par exemple :

```
Ecrire " Etude des valeurs de ... sur l'intervalle [a,b]."
SaisieAvecControle(borneA,borneB)

# A partir d'ici a et b sont connus
# mais de plus leurs valeurs sont correctes
# pour la suite de l'algorithme
```

1.5 Les Structures Conditionnelles

Une structure conditionnelle sert à choisir quelle(s) instruction(s) à exécuter, en fonction d'une *condition*. Le propre d'une bonne structure algorithmique est d'être explicite et facilement visualisable. Le début et la fin de chaque structure doivent être explicites et nom ambigues, ce qui permet à un bon traitement de texte de repérer le début et la fin de la structure, de mettre par exemple en couleur toute la structure.

Nous utilisons `SI` pour indiquer le debut de la conditonnelle simple et `FINSI` (ou `FIN SI`, ou `FIN␣SI`, ou `FIN_SI`) pour en indiquer la fin. Le mot `ALORS` qui pourrait être facultatif, a pour but d'aider à la lecture. S'il y a quelque chose à faire quand la condition n'est pas vérifiée, on utilise le mot `SINON`.

L'ambiguité du `SINON` (appartient-il au test 1 ou au test 2 ?) comme dans l'extrait suivant d'algorithme

```
SI condition1
ALORS action1
SI condition2
ALORS action2

SINON action3
```

est levée par l'indentation et le mot `FINSI`. Les deux seuls algorithmes corrects possibles (car non ambigus) sont donc ici

```
# sinon dans test 2

SI condition1 ALORS
   action1
   SI condition2
      ALORS action2
      SINON action3
   FINSI
FINSI
```

```
# sinon dans test 1

SI condition1
   ALORS action1
     SI condition2
        ALORS action2
     FINSI
   SINON action3
FINSI
```

Ce qui est nécessaire n'est pas toujours suffisant. Il **faut** aussi commenter les `FINSI` (ce que peut faire pour nous un bon éditeur de textes), aérer les structures.

Les "bons" algorithmes qui correspondent aux précédents sont donc

```
# sinon dans test 2


SI condition1 ALORS
|  action1
|
|   SI condition2
|   |  ALORS action2
|   |  SINON action3
|   FINSI { 2 }
|
FINSI  { 1 }
```

```
# sinon dans test 1


SI condition1
|  ALORS action1
|
|        SI condition2
|        |  ALORS action2
|        FINSI { 2 }
|
|  SINON action3
FINSI { 1 }
```

Dans le cas où il n'y a pas de sinon, on peut gagner de la place en mettant le `ALORS` sur la même ligne que le `SI` comme dans

```
SI   NumClient = 100 ALORS
     ECRIRE " on vient de passer la centaine de clients."
FINSI  { NumClient = 100 }
```

S'il y a beaucoup de tests à effectuer sur une même variable ou sur plusieurs, la structure `SI` est lourde à écrire, et sans doute peu efficace une fois traduite (voir l'exemple des affichages par ordre croissant au chapitre 3.). On peut alors recourir à une **structure de cas** via la syntaxe

```
AUX CAS OU
    CAS [condition 1]
         instructions du cas 1...
    CAS [condition 2]
         instructions du cas 2...
    ...
    AUTRES CAS
         instructions...
FIN DES CAS
```

à condition de bien noter que les cas doivent doivent être disjoints, et qu'une fois un cas choisi, l'algorithme reprend après la structure de cas : l'ordre des cas ne doit pas intervenir dans le résultat de l'algorithme.

Par exemple, l'extrait d'algorithme suivant

```
AUX CAS OU
    CAS (4*x+6-x*x>0) ET (y*y-5*y+1<0)
        ECRIRE "Oui"
    CAS (x*y+x-2*y+y*y-1>0)
        ECRIRE "Non"
    AUTRES CAS
        ECRIRE "p'tet !"
FIN DES CAS
```

est incorrrect car le couple $X = 5$ et $Y = 2$ est solution des deux cas (ce qui n'est pas si évident que cela à trouver) et dans le premier cas, on dit `OUI` alors que dans le second, on dit `NON`.

1.6 Les Boucles

Une boucle sert à répéter un certain nombre de fois la même action, c'est à dire la même suite d'instructions, nommée "corps de boucle", éventuellement réduite à une seule instruction. Une boucle peut servir à compter, à passer en revue les éléments d'une structure, etc.

Une boucle est soit implicite soit explicite. Une boucle implicite s'obtient grâce à un appel récursif d'un "module" (ou "sous-programme"). Une boucle explicite utilise le mode itératif ; en fonction du type de nombre de fois (fixe, connu à l'avance, réévaluable), on utilise les boucles explicites `POUR` ou les boucles explicites `TANT QUE` (nous déconseillons les boucles `JUSQU'A` car deux types de boucles, c'est déjà beaucoup).

Choisir entre une boucle implicite et une boucle explicite est affaire de gout, de langage : si on sait que le langage n'a pas de boucles explicites, on sera forcé d'utiliser une boucle implicite. Si on s'intéresse à la traduction de l'algorithme en programme, il y a de fortes chances qu'on s'intéresse aussi à la vitesse d'exécution du programme. Si le langage est adapté à la récursivité (comme Lisp, Prolog...) on pourra laisser les algorithmes récursifs tels quels. Sinon, il faudra "dérécursiver" les algorithmes avec des piles, des queues et gérer la récursivité terminale, l'encombrement en mémoire des variables, etc.

Conceptuellement, le mode itératif et le mode récursif sont équivalents : on peut toujours convertir un algorithme itératif en un algorithme récursif et réciproquement. Ce n'est pas toujours facile ni lisible, mais c'est faisable.

Il reste néanmoins qu'avec de l'expérience, les solutions récursives sont souvent plus élégantes, mais plus délicates à écrire.

1.6.1 la boucle POUR

Une boucle `POUR` suit la syntaxe

```
POUR [variable] DE1A [constante ou variable]

     suite d'instructions...

FINPOUR [rappel de la variable]
```

On pourra ainsi écrire

```
# algorithme : megalomane et "megalowoman"

POUR ind DE1A 1000
     SI sexe="F"
        ALORS ECRIRE " je suis la plus belle"
     SINON
        ALORS ECRIRE " je suis le plus beau"
     FINSI { sexe }
FINPOUR { ind }
```

Le terme `DE1A` doit être systématiquement utilisé pour des algorithmes élémen-taires. Cela permet de prévoir le nombre d'itérations de la boucle sans aucun calcul contrairement à la boucle

```
POUR <variable> DE <variable> A <variable> PAR_PAS_DE <variable>
```

que nous refusons (sans l'interdire, bien sûr).

On tolèrera `POUR <variable> DE0A <variable>` si on doit commencer à zéro dans quelques cas bien particuliers (car quand on compte sur ses doigts on commence à 1 et pas à 0) mais on n'utilisera jamais

```
POUR <variable> DE <variable> A <variable> PAR_PAS_DE <variable>
```

Cette syntaxe de boucle, souvent implémentée dans les langages de programmation, est trop générale à notre goût : elle ne commence pas forcément à 1, n'incrémente pas forcément de 1 l'indice de boucle et il est donc difficile sans calcul de savoir combien de fois on va exécuter le corps de boucle. Nous conseillons d'utiliser la boucle `TANT QUE` lorsque la boucle `POUR` de base ne suffit pas.

Si vraiment le problème s'y prête (cela fait donc de de très rares cas), on tolèrera la boucle `POUR <variable> DE <variable> A <variable>` avec un pas implicite de 1, mais avec au plus trois ou quatre instructions dans le corps de boucle.

Afin que les algorithmes donnent des résultats "intuitifs", il faut imposer trois règles à respecter dans le corps de la boucle `POUR`

1 - on n'a pas le droit de modifier l'indice de boucle,
2 - on n'a pas le droit de modifier la borne de boucle,
3 - on ne peut pas réutiliser la dernière valeur d'indice de boucle.

On se méfiera du fait que les langages de programmation ne respectent pas forcément ces règles.

Ainsi, pour afficher les n nombres 1, -2, 4, -8 ... on peut écrire

```
signe  <-- 1
x      <-- 1
POUR k DE1A n
   x     <-- signe * x
   ecrire x
   x     <--   x * 2
   signe <--  (- signe)
FINPOUR k
```

mais il est donc interdit d'écrire `k` ← `k+1` ou tout autre expression comme `k` ←... dans le corps deboucle, au même titre que `n` ←... ; une fois la boucle exécutée, il serait faux de supposer que `k` contient `n`. Il faut écrire explicitement `k` ← `n` si on veut repartir de cette valeur.

Nous avons déjà insisté sur la fait qu'avec la boucle `POUR`, les indices commencent à 1, que l'incrémentation est effectuée de 1 en 1 et que la borne de la boucle est limitée à une variable simple. Si ces restrictions rendent la boucle `POUR` peu générale, elle lui confèrent simplicité et lisibilité. Pour les autres cas de boucles, il vaut mieux utiliser la boucle `TANT QUE`.

1.6.2 la boucle TANT QUE

Elle doit avoir pour syntaxe

```
[ initialisation des variables du test ]
TANT QUE [test ou variable logique]
     [ suite d'instructions ]
     [ modification des variables du test ]
FIN TANT QUE [ rappel du test ]
```

Et en particulier, la boucle `POUR k DE1A n` équivaut à

```
k <-- 1
TANT QUE k <= n
     [ suite d'instructions ]
     k <-- k + 1
FINTANTQUE
```

Dans cette écriture, on peut mettre `k ← k + 2`, pour aller de 2 en 2, changer `n` en `n-1` si on veut dans le corps de boucle, etc. et on dispose alors de toute la souplesse nécessaire pour une boucle à géométrie variable.

D'autres boucles sont envisageables, mais nous n'en voulons qu'une autre car ces boucles suffisent à couvrir tous les cas : la boucle `POUR CHAQUE`. Voir la section suivante car cette boucle utilise des tableaux.

Une boucle `TANT QUE` typique est celle qui gère une entrée, disons celle de la variable `x`, soumise à des contraintes (comme la positivité, ou d'être inférieure à 10, etc.). Elle s'écrit

```
Demande( x )
TANT QUE Condition( x )
   Relance( x )
FINTANTQUE
```

où `Demande(x)` et où `Relance(x)` effectuent toutes les opérations nécessaires de lecture et écriture et où `Condition(x)` vient tester si la condition sur `x` est vérifiée ou non. L'algorithme de la section 1.1 en est un exemple détaillé.

1.7 Les Tableaux (ou "*variables indicées*")

Une variable indicée permet d'associer plusieurs valeurs à un même nom. Ces valeurs sont accessibles par un indice, souvent numérique entier supérieur ou égal à 1, mis entre crochets droits. On peut alors interpréter l'expression `Vente[12]` comme la vente numéro 12. Il n'y a pas en principe de contre-indication quand aux valeurs des indices, à la première valeur utilisable. Commencer par `1` assure une certaine cohérence (la première valeur est alors la valeur numéro 1) mais commencer à `0` peut s'imposer lorsqu'on sait que le langage sans doute utilisé pour la traduction commence à 0.

Utiliser un indice ne signifie pas que les autres indices existent. Si un algorithme doit gérer les valeurs obtenues dans un sondage pour les années 1990, 1995, 2000 on peut très bien utiliser seulement les valeurs `NbReponses[1990]`, `NbReponses[1995]`... sans risque de nuire à la clarté de l'algorithme . Ce qui se passerait dans la mémoire de l'ordinateur est un problème distinct que l'on ignore ici mais qui devra être évoqué puis traité en temps et en heure, à savoir au moment de la traduction de l'algorithme en programme.

Mettre des indices avec des nombres décimaux a également un sens : certains problèmes requiérent des valeurs indicées décimales. Par exemple, si `5000` personnes font `1.78 m` il est tout naturel d'écrire `nbPers[1.78]` ← `5000`.

Il est tout aussi facile d'étendre les indices à des valeurs caractères, ce qu'on nomme *tableaux associatifs* ou *hash* car l'indexation par des mots est courante dans la vie de tous les jours. Ainsi `age["pierrette"]` ← `32` est très compréhensible. On n'hésitera donc pas à utiliser des indices caractères quitte, là encore, au niveau de la traduction en programme, à se préoccuper de ce qu'accepte le langage de programmation choisi.

Si on utilise des indices autres que numériques de `1` à `N` ou avec un pas autre que `1`, la boucle `POUR` traditionnelle n'est pas adaptée. On pourra utiliser la boucle

```
POUR CHAQUE  [indice] DANS [tableau]
    [corps de boucle]
FIN POUR CHAQUE [indice]
```

Cette boucle se comprend bien, mais elle n'est pas directement traduisible dans tous les langages.

A un niveau élémentaire, nous ne conseillons pas d'utiliser les deux tableaux classiques induits par un tableau associatif, à savoir celui des valeurs et celui des indices (comme avec Perl). Mais rien n'empêche d'utiliser une fonction qui renvoie les indices dans un tableau et une autre fonction qui renvoie les valeurs (uniques ? non uniques ?) dans un tableau comme dans

```
# transfert des indices du tableau

  tabIndAge <-- indicesTableau( age )

# transfert des valeurs du tableau

  tabValAge <-- valeursTableau( age )
```

A un niveau élémentaire, il n'y a pas besoin d'autre structure de données. Rien n'interdit, bien sur d'inventer des structures et des sous-programmes adaptées. Par exemple, on peut assez simplement définir les ensembles, avec les opérations *union*, *intersection* à l'aide de tableaux... Le chapitre 4 donne des exemples d'algorithmes qui implémentent divers cas pour répondre à ces besoins.

1.8 Les Sous-programmes (ou *modules*)

S'il fallait écrire en détail tous les algorithmes d'un coup, on remplirait des pages et des pages de texte et ce ne serait pas très clair. L'usage est en général de remplacer des parties d'algorithmes par une seule instruction avec un nouveau mot bien choisi et de donner ensuite le détail des instructions correspondant à ce mot dans un autre algorithme. L'algorithme simplifié utilise les autres algorithmes et ne contient souvent que le fil conducteur des actions à effectuer. On le nomme *algorithme principal* et les autres algorithmes secondaires, ou sous-algorithmes, ou encore plus fréquemment *sous-programmes*.

Il faut regrouper avec soin les parties importantes et leur donner avec des noms symboliques explicites, en précisant quelles variables sont utilisées. Traditionnellement, pour un algorithme un peu long, on utilise trois modules :

```
SAISIE(...)     # des valeurs
CALCULS(...)    # si besoin est
AFFICHAGES(...) # correspondants
```

Par exemple, en marketing, on peut envisager de facturer la vente à partir du code client, et du code article et de la quantité d'articles achetés. La saisie des variables correspondantes, à savoir `CodeArticle`, `CodeClient`, `Quantite` ne pose en principe aucun problème majeur. Mais plutot que d'écrire

```
# Demande du code client
  ECRIRE " Quel est le code client ? "
  LIRE     CodeClient
# Demande du code article
  ECRIRE " Quel est le code article ? "
  LIRE     CodeArticle
# Demande du nombre d'articles
  ECRIRE " Quel est le nombre d'article(s) ? "
  LIRE     Quantite
```

on pourra se contenter d'utiliser un module `SaisieVente`. Comme ce module doit récupérer nos trois variables, nous écrirons dans l'algorithme principal

```
APPELER SaisieVente( CodeClient, CodeArticle, Quantite )
```

On peut, si l'on veut, omettre le mot `APPELER`.

Certaines modules se réduisent parfois à calculer une valeur. Ainsi le prix toutes taxes comprises se déduit du prix hors taxe et du taux de taxe. Au lieu de la syntaxe

```
CalculePrixTtc( HorsTaxe, TauxTaxe , AvecTaxe )
```

on pourra introduire la notion de *module-fonction utilisateur* ou, plus simplement, de *fonction* ce qui permet de simplifer l'écriture précédente en

```
AvecTaxe <-- CalculePrix( HorsTaxe, TauxTaxe )
```

Formellement, on prendra les précautions suivantes : une fonction ne peut renvoyer qu'une seule variable (simple ou tableau) et la valeur renvoyée doit être la dernière instruction du module.

Lorsqu'un module s'appelle lui-même (par exemple $f(x, y)$ se calcule à partir de $f(x-1, y-2)$ sauf pour les premières valeurs qui sont données explicitement), le module est récursif. On peut choisir d'écrire des modules récursifs ou interdire d'utiliser la récursivité selon le langage visé ou selon le problème posé.

L'écriture formelle d'un module se fait avec la syntaxe suivante :

```
MODULE [nom]
       VARIABLES PARAMETRES ...
       VARIABLES LOCALES    ...
       VARIABLES GLOBALES   ...
DEBUT_DU_MODULE [nom]
       ...
FIN_DU_MODULE [nom]
```

et pour chaque variable paramètre ou globale, on devra indiquer si elle est en lecture (seule), c'est à dire non modifiable, ou si elle est en lecture et écriture. Par exemple, le module `CalculePrixTtc` doit s'écrire

```
MODULE  CalculePrixTtc
        VARIABLES PARAMETRES
                  HorsTaxe, TauxTaxe (lecture)
                  AvecTaxe  (écriture)
        VARIABLES LOCALES
                  ValeurTaxe
        VARIABLES GLOBALES (aucune)
```

```
DEBUT_DU_MODULE CalculePrixTtc
        ValeurTaxe <-- HorsTaxe * TauxTaxe
        AvecTaxe   <-- HorsTaxe + ValeurTaxe
FIN_DU_MODULE CalculePrixTtc
```

alors que la fonction `CalculePrix` utilisera le mot-clé `RENVOYER` et s'écrira

```
MODULE_FONCTION   CalculePrix

        VARIABLES  PARAMETRES
                   HorsTaxe, TauxTaxe (lecture)
        VARIABLES  LOCALES
                   ValeurTaxe
                   PrixAvec
        VARIABLES  GLOBALES (aucune)

DEBUT_DU_MODULE CalculePrixTtc
        ValeurTaxe <-- HorsTaxe * TauxTaxe
        PrixAvec   <-- HorsTaxe + ValeurTaxe
        RENVOYER PrixAvec
FIN_DU_MODULE CalculePrixTtc
```

Cette syntaxe, un peu lourde (mais explicite), pourra être produite automatiquement sur ordinateur ou allégée. On peut ainsi noter `FONCTION` au lieu de `MODULE_FONCTION`, `LOCALES` au lieu de `VARIABLES LOCALES`, omettre `VARIABLES GLOBALES` s'il n'y en a aucune. L'important, là encore est la lisibilité de l'algorithme et la traçabilité des variables.

Une variable *globale* est "extérieure" au sous-programme, c'est à dire considérée comme connue de la part de ce sous-programme. Ce pourrait être une constante (comme le taux du change de l'euro en francs) mais c'est le plus souvent une variable de l'algorithme principal qu'on ne veut pas passer en paramètres. La distinction entre variable paramètre et locale résulte du choix du programmeur. Ainsi, au lieu d'utiliser un taux de taxe, nous pourrions, pour une comptabilité française, utiliser un code de T.V.A., à savoir la valeur 1, 2 ou 3 et disposer des taux dans une variable globale, en l'occurence, le tableau `TauxTaxe`.

L'appel de la fonction serait alors `CalculePrix( HorsTaxe, CodeTaux )`.

Le texte du module pourrait être

```
MODULE_FONCTION   CalculePrix

        VARIABLES  PARAMETRES
                   HorsTaxe, CodeTaux(lecture)
        VARIABLES  GLOBALES (
                   # tableau des valeurs de taux :
                   TauxTaxe (lecture)
        VARIABLES  LOCALES
                   ValeurTaxe
                   PrixAvec
        VARIABLES  GLOBALES (aucune)

DEBUT_DU_MODULE CalculePrixTtc

        ValeurTaxe <-- HorsTaxe * TauxTaxe[ CodeTaux ]
        PrixAvec   <-- HorsTaxe + ValeurTaxe
        RENVOYER PrixAvec

FIN_DU_MODULE CalculePrixTtc
```

On voit sur cet exemple que l'ordinateur est d'un précieux secours pour écrire les algorithmes : s'il fallait écrire à la main le texte précédent, on n'aurait sans doute écrit que `vt` au lieu de `ValeurTaxe`, `DEBUT` au lieu de `DEBUT_DU_MODULE CalculePrixTtc`. La fonction *Copier/Coller* et la fonction *Dupliquer* (quand elle existe) d'un éditeur de texte permettent d'obtenir rapidement et sans effort toutes ces redites qui facilitent la relecture.

L'écriture des modules doit en principe distinguer les variables réelles, utilisées dans l'exécution de l'algorithme principal et les valeurs formelles mises en jeu par le sous-programme. Par exemple, si l'algorithme principal utilise `f(x,y)` le module ne doit pas nommer ses paramètres `x` et `y` car le lecteur sera induit en erreur si l'algorithme comporte quelques instructions plus loin `f(y,x)` ou encore `f(x+y,x-y)`. Pour de telles utilisations, des noms de paramètres très distincts comme `parmA` et `parmB` éviteront toute confusion.

Pour notre exemple de calcul de prix, en toute rigueur (mais là encore ce n'est pas long de le faire sur ordinateur), il aurait fallu distinguer la variable `HorsTaxe` de l'algorithme principal et la variable `HorsTaxe` du sous-programme, par exemple en la renommant en `pHorsTaxe` (le `p` vient indiquer qu'il s'agit d'un paramètre).

On aurait alors l'algorithme

```
MODULE_FONCTION   MeilleurCalculePrix
# meilleur car on nomme "correctement" les variables

       VARIABLES PARAMETRES
                  pHorsTaxe, pCodeTauxTaxe (lecture)
       VARIABLES GLOBALES (
                  TauxTaxe (lecture) # tableau des valeurs de taux
       VARIABLES LOCALES
                  ValeurTaxe
                  PrixAvec
       VARIABLES GLOBALES (aucune)

DEBUT_DU_MODULE CalculePrixTtc

        ValeurTaxe <-- pHorsTaxe * TauxTaxe[ pCodeTauxTaxe ]
        PrixAvec   <-- pHorsTaxe + ValeurTaxe
        RENVOYER PrixAvec

FIN_DU_MODULE CalculePrixTtc
```

Pour en finir partiellement avec les modules, on pourra toujours supposés existant les modules dont on a besoin, comme le tri, l'extraction du minimum, le comptage de valeurs correspondant à des conditions... L'important est ici de bien indiquer quelle opération "évidente ou cohérente" on envisage, d'en préciser la syntaxe exacte, quitte à écrire soi-même le sous-programme (ou à le trouver dans un recueil d'algorithmes).

Prenons comme exemple le tri d'un seul tableau. On peut se contenter de nommer `Tri` la fonction correspondante, l'appel étant alors `Tri(parmT)` où `parmT` est un tableau, mais on on peut éventuellement envisager de ne trier que les indices `x` à `y` du tableau, de choisir le sens croissant ou décroissant, etc. L'appel pourra alors être

```
Tri(parmT,indDeb,indFin,sensTri)
```

où `sensTri` sera au choix un code (`1` pour croissant, `2` pour décroissant) ou un caractère (`"A"` ou `"<"` pour croissant, `"D"` ou `">"` pour décroissant).

Les modules sont une partie fondamentale de l'algorithmique : décomposer un problème en sous-problèmes (le fameux *"divide and conquer"*) puis nommer les modules et choisir les paramètres est un art qui peut s'apprendre en lisant les modules des autres, de la littérature classique.

1.9 Les Opérations sur fichiers

Il n'y a que 4 opérations élémentaires pour un fichier : *ouvri*r le fichier, y *lire*, y *écrire* et le *fermer*. Puisqu'il doit forcément y avoir une correspondance entre une variable de fichier et le fichier véritable, le mieux est d'utiliser la syntaxe

```
OUVRIR [nomVeritable] COMME [variableFichier]
```

S'il le faut, on précise à l'ouverture si l'on veut lire ou écrire avec la syntaxe

```
OUVRIR [nomVeritable] EN_LECTURE  COMME [variableFichier]
OUVRIR [nomVeritable] EN_ECRITURE COMME [variableFichier]
```

La lecture sur le fichier peut lire autant de variable que nécessaire : on les sépare par des virgules. De même pour l'écriture. Et on précise quel fichier est utilisé avec le mot `SUR`. La syntaxe est donc

```
LIRE   var1 [ ,var2, var3... ] SUR variableFichier
ECRIRE var1 [ ,var2, var3... ] SUR variableFichier
```

Enfin, la fermeture n'a besoin d'utiliser que la variable de fichier et a pour syntaxe

```
FERMER  variableFichier
```

Si l'on veut simplifier la lecture et l'écriture des fichiers, on peut se restreindre à l'écriture et à la lecture de lignes de texte, quitte à les redécomposer ensuite. Les fichiers mis en jeu sont alors des fichiers-textes lisibles à l'écran, qu'on peut modifier sous éditeur de texte. Cela restreint la portée, l'efficacité de l'algorithme par rapport à un programme véritable avec des fichiers séquentiels, indexés mais c'est plus simple à gérer. Ainsi la lecture du nom, du prénom et de l'age d'une personne sur le fichier employes.txt associé à la variable `fEmpl` peut se faire par

```
LIRE ligneEmploye SUR fEmpl
nom    <-- mot(ligneEmploye,1)
prenom <-- mot(ligneEmploye,2)
age    <-- valeur( mot(ligneEmploye,3) )
```

qui utilise des fonctions "évidentes" comme `mot`, `valeur`.

Si on préfère, on peut écrire un module spécial de découpage de la phrase comme :

```
LIRE ligneEmploye SUR fEmpl
decoupeLigne(ligneEmploye,nom,prenom,age)
```

Une seule fonction est en principe nécessaire pour gérer l'utilisation des fichiers : celle qui teste si on est rendu au bout du fichier. On la nomme `fin_de_fichier` (ou `fdf` s'il faut l'écrire à la main).

Voici par exemple un algorithme qui met dans le fichier nommé *Resultat.Dat* (variable `Trois`) le contenu du fichier *Premier.tst* (variable `Un`) puis le contenu du fichier *Final.tst* (variable `Deux`) :

```
# recopie du fichier Un
#    puis du fichier Deux
#    dans le fichier Trois

OUVRIR "Resultat.Dat" COMME Trois
## sous-entendu : en ecriture

{ 1.  recopie du fichier Un }

OUVRIR "Premier.tst" COMME Un
## sous-entendu : en lecture
TANT QUE non fin_de_fichier( Un )
     # ttl : Toute La Ligne
     LIRE   ttl sur Un
     ECRIRE ttl sur Trois
FIN  TANT QUE { non fin_de_fichier( Un ) }
FERMER Un
```

```
{ 2.  recopie du fichier Deux }

OUVRIR "Premier.tst" COMME Deux
TANT QUE non fin_de_fichier( Deux )
     LIRE   ttl sur Deux
     ECRIRE ttl sur Trois
FIN  TANT QUE { non fin_de_fichier( Deux ) }
FERMER Deux

# il ne reste plus qu'a
FERMER Trois
```

Des algorithmes classiques et faciles à écrire sont par exemple ceux qui mettent dans un tableau les lignes d'un fichier texte, la ligne numéro `i` devant l'élément numéro `i` du tableau et son réciproque, c'est à dire l'algorithme qui transfère le contenu d'un tableau dans un fichier.

On trouvera sur notre site *Web* à l'adresse

http ://www.info.univ-angers.fr/pub/gh/Galg/

d'autres exemples d'algorithmes mettant en jeu des fichiers. On y trouvera aussi des algorithmes qui utilisent la syntaxe de galg pour que ces algorithmes qui utilisent des fichiers soient automatiquement traduits en Rexx, C, C++, Java...

Les opérations sur fichier constituent la dernière partie de la base de nos langages algorithmiques. Avec tous les mots, symboles et structures introduits, et de par la souplesse induite par la notion de module, tout programme peut être décrit par un algorithme dans ces langages.

Cette description, cette codification peut se révéler fastidieuse, en fonction du sujet choisi. Pour gagner en concision, mais peut être perdre en clarté, il faut ajouter d'autres conventions, d'autres notations. C'est le but du chapitre suivant. Nous concluons juste en indiquant les deux types principaux de langages que l'on peut batir sur ces éléments algorithmiques.

1.10 Quels langages algorithmiques, alors ?

Nous pouvons enfin détailler pourquoi ce chapitre s'intitule langages algorithmiques au pluriel. Avec les éléments de de base proposés, il est possible de construire deux algorithmiques fondamentales assez différentes mais avec le même langage.

La première est dite "**fonctionnelle**". Elle n'utilise que la récursivité, les nommages (les affectations et les boucles `pour` et `tant que` y sont donc interdites, de même que le passage de variables en écriture), elle ignore les entrées-sorties et les fichiers.

La seconde algorithmique, dite "**impérative**" utilise les nommages, autorise les affectations, les boucles `pour` et `tant que`, le passage de variables en écriture. Elle ne met pas pas forcément en jeu la récursivité.

Ces deux algorithmiques ne sont pas meilleures l'une que l'autre. Elles sont complémentaires, autant pour l'apprentissage de la programmation que pour l'implémentation.

Traditionnellement, l'algorithmique fonctionnelle mène à la "preuve de programme" et aux langages de spécification (comme le langage B, le langage Z), prépare à la programmation en Prolog et en Lisp, facilite l'utilisation de sous-programmes fonctionnels comme les "packages" en Maple et Mathematica.

D'un autre coté, l'algorithmique impérative est directement traduisible en Rexx, Awk, C, Pascal, Perl. Elle "colle" plus à la machine, aux variables. Pour de petits problèmes, elle offre des solutions directes (sans module ni fonction). De plus, la *séquentialité* explicite des boucles, des affectations permet de tracer facilement l'exécution.

En toute rigueur, il faudrait aussi présenter deux autres algorithmiques : une algorithmique "logique" et une algorithmique "objets". La première serait basée sur l'unification de prédicats, la seconde serait issue de l'analyse et la conception par objets. Toutefois, ces deux autres algorithmiques ne sont pas élémentaires et c'est pourquoi nous ne les présentons pas ici.

Une dernière remarque

Si vous préférez plutôt voir tout de suite des algorithmes classiques écrits avec la base des langages fournis, passez aux chapitres **3.** et **4.** puis revenez ensuite lire le chapitre **2.** afin de comprendre comment compléter la base de ces langages algorithmiques dans ce qui sera votre langage, mais compréhensible par tous.

Chapitre 2.

Compléments algorithmiques

Nous avons présenté au chapitre 1 les fondements de nombreux langages algorithmiques. C'est un tout clair, cohérent mais volontairement limité. Ce chapitre va essayer de justifier nos choix puis va présenter quelles extensions on peut leur adjoindre et surtout dans quel esprit construire de telles extensions.

Si les instructions de programmation sont en général des **actions** à exécuter, doù les verbes comme `LIRE`, `ECRIRE`, il ne nous a pas paru lisible de faire débuter chaque instruction par un verbe. La syntaxe

```
valX  <-- 2 + NbPommes
```

nous parait suffisante, sans avoir à rajouter le mot `CALCULER` ou `AFFECTER` comme dans

```
AFFECTER  2 + NbPommes A valX
```

ou dans

```
CALCULER valX  <-- 2 + NbPommes
```

De même, pour les boucles, nous n'avons pas trouvé judicieux de rajouter le mot `FAIRE` comme dans

```
FAIRE POUR [variable] DE1A [constante ou variable]
      [suite d'instructions]
FIN FAIRE POUR [rappel de la variable]
```

Tout est affaire de compromis et de contrat : compromis entre la formalisation et la lisibilité, entre la précision et la clarté ; contrat entre la méthode et les instructions pour la réaliser, entre la volonté du programmeur et l'attention du lecteur.

C'est aussi pourquoi nous ne nous embarassons pas de taille mémoire, de taille de tableau ou de limite sur les fichiers. Une algorithmique simple est une algorithmique efficace. Si la lecture sur fichier pose un problème particulier, si la taille des tableaux justifie une approche différente, on viendra simplement le signaler en commentaire dans l'algorithme et transformer les simples affectations et lectures comme

```
ValMax <-- TabVal[ indMax ]
ECRIRE ValMax SUR FicRes
```

par des appels de modules équivalents comme

```
AffecteTableau( ValMax , TabVal , indMax )
EcritureFichier( ValMax , FicRes )
```

qui se réécrivent en trois coups de manipulation avec un bon éditeur de textes.

2.1 Modules et Fonctions susceptibles d'exister

La base de notre langage utilise peu de mots, peu de fonctions. Pourtant, dès lors qu'on veut effectuer des calculs avec des nombres ou des opérations sur chaines de caractères, il faut utiliser de nombreuses fonctions. Comme les critères importants sont la **lisibilité** et la **clarté**, il faut toujours préciser la **syntaxe** des fonctions. Ainsi pour des calculs, on peut avoir besoin de calculer des puissances. L'appel de la fonction `puissance` par l'instruction `puissance(a,b)` a autant de chances de calculer a^b que b^a. Il faut donc bien expliciter la syntaxe, voire les conditions d'utilisations s'il y en a.

Par exemple — et vous pouvez omettre ce paragraphe si vous n'êtes pas ma-thématicien(ne) ou si un mot comme *logarithme* ressemble à un gros mot voire à un cauchemar — les fonctions mathématiques sont à la fois un vocabulaire rassurant et la source de nombreux pièges.

Ainsi le mot *log* désigne-t-il le "brave" logarithme népérien ou correspond-t-il au logarithme décimal ? Autorise-t-on à calculer la factorielle d'un nombre non entier (car il s'agit de la fonction Γ) ? Quelle précision veut-on lorsqu'on demande la racine de tel nombre ? Pour tous ces calculs, un **consensus** doit être clairement établi, qui permet de verrouiller l'appel des fonctions, qui fait la part entre la mécanique calculatoire des instructions et le flou qu'on s'autorise quant à l'implémentation machine.

Pour les chaines de caractères, là encore il faut un **consensus**. Utiliser toutes les fonctions disponibles de C, Pascal, Rexx, Perlou Tcl/Tk risque d'aboutir à un *salmigondis de pseudo-fonctions francisées*, si copiées sur un langage que l'algorithme ne pourra pas être traduit sans tout réécrire. Par contre, n'utiliser qu'un minimum de fonctions rallongera désespérément les algorithmes.

Certes, un langage complet doit comporter toutes les fonctions élémentaires sur les chaines de caractères comme

```
chaineDateHeure      chaineMajuscule      chaineMinuscule
copiesDe             dernierePosition     formateChaine
initialeMajuscule    longueur             motNumero
nombreDeFois         nombreDeMots         partieDroite
partieGauche         premierePosition     sousChaine
```

et toutes les fonctions avancées comme

```
compareChaineAvecFonction          convertitEnCaractere
convertitEnDecimal                 convertitEnHexadecimal
decoupeSuivantModele               insereDans
insereMotDans                      marqueEnHtmlComme
remplaceAvecExpressionReguliere    remplaceDans
remplaceMotDans                    supprimeBlancsInutilesDe
supprimeDans                       supprimeDoublonsDe
supprimeMotDans                    retournerParRapportA
```

mais cela ferait beaucoup trop de fonctions dès le départ à connaitre, à manipuler, sachant que certaines actions sont soumises à des choix. Par exemple la notion de mot, de caractéres accentués sont des notions délicates et croire qu'un mot a la même définition pour tous les langages serait masquer de nombreux problèmes, d'autant plus qu'en mathématiques, en informatique et en traitement de texte, les "mots" existent aussi, avec des définitions différentes.

Pour un problème donné, on peut imposer une syntaxe, faire comme si on avait une bliblothèque de fonctions. Par exemple au chapitre 4 nous utilisons `dernPos( aiguille, botte )` pour indiquer la dernière position, en tant que caractères, de la chaine `aiguille` dans la chaine `botte`. Ce qui signifie que `dernPos( "co", "chocolat", )` vaut 4 car le `"c"` de `"co"` est la 4-ième lettre de `"chocolat"`. Cette fonction a une définition "intuitive" à savoir *trouver la dernière position de ... dans la phrase* mais il reste à préciser ce qui se passe si la chaine n'est pas vue, ce qui se fait en général en faisant renvoyer par la fonction une valeur numérique spéciale.

Si la fonction renvoie un nombre entier strictement positif lorsque la chaine recherchée est présente au moins une fois — et comme c'est souvent le cas dans les langages de programmation, cette valeur est la position du premier caractère de la chaine cherchée en tant que sous-chaine dans l'autre chaine – il faut quand même fixer la valeur de retour si elle n'y est pas. Utiliser la valeur 0 présente des avantages mais utiliser une valeur négative aussi. D'où la nécessité de choisir et de convenir de ce choix.

Par exemple :

1. Si on choisit de renvoyer `0` lorsque la chaine n'est pas vue, il est simple de tester si un mot au moins n'est pas vu puisque la multiplication par `0` donne `0`, comme pour

   ```
   # au moins un mot non vu
   SI dernPos("chat",nomAnimal)*dernPos("chien",listeZoo) = 0
   ```

   par contre, il n'est pas très facile de distinguer si un seul mot n'est pas vu ou si les deux ne sont pas vus.
2. Si on choisit de renvoyer un nombre négatif, par exemple `-1`, le cas où un seul des deux mots n'est pas vu s'écrit directement par

   ```
   # exactement un seul mot non vu
   SI dernPos("chat",nomAnimal)*dernPos("chien",listeZoo ) < 0
   ```

   puisque `-1` multiplié par `-1` vaut `+1`.

On voit bien ici que le choix de la valeur de retour peut simplifier ou au contraire compliquer les algorithmes.

Passer d'un choix à l'autre (par exemple en vue de la traduction dans un langage précis qui utilise ce code-retour) est très simple et ne demande pas beaucoup d'efforts.

Ainsi, on pourra écrire

```
FONCTION AutreDernPos
  # la fonction dernPos renvoie 0 si la chaine n'est pas vue
  # la fonction AutreDernPos renvoie -1 dans ce cas

  PARAMETRES aiguille, botte (lecture) # comme pour dernPos
  LOCALE     retour

DEBUT DE FONCTION AutreDernPos
  retour <-- dernPos( aiguille, botte )
  SI retour = 0 ALORS
     retour <-- (-1)
  FINSI
  RENVOYER retour
FIN DE FONCTION AutreDernPos
```

et là encore avec un coup d'éditeur de textes, tout est réglé en remplaçant `dernPos` par `AutreDernPos`.

Le même problème se pose avec les tableaux et les fonctions sur tableau. Il serait bon de disposer des fonctions suivantes sur tableau

```
concateneTableaux                convertitChaineEnTableau
convertitTableauEnChaine         maximumTableau
minimumTableau                   moyenneTableau
rechercheDansTableau             sousTableau
tableauAleatoire                 tailleTableau
trieTableau                      indexeTableau
```

et aussi des fonctions suivantes sur tableau associatif

```
compteElement                    dernierElement
elementPrecedent                 elementSuivant
lineariseTableau                 premierElement
supprimeElement                  appliqueFonction
```

Toutefois, pour ne pas surcharger le langage de base algorithmique, ces fonctions, sophistiquées et dont le détail de comportement doit être précisé et détaillé sont laissées au gout du programmeur.

D'autres fonctions, encore, liées au temps sont importantes comme la durée de l'exécution d'une instruction ou d'un sous-programme, l'attente au clavier pendant un certain temps, la conversion d'une date, d'une heure en secondes écoulées depuis une date de référence, la conversion d'une date en numéro de jour, en nom de jour etc. Là encore ces fonctions, nombreuses, peuvent faire partie d'une extension du langage.

Enfin, les fonctions qui touchent aus système d'exploitation seraient à envisager, comme :

```
changeCd                    cheminFichier
copieFichier                dateFichier
deplaceFichier              detruitFichier
droitsFichier               heureFichier
listeRepertoire             nombreDeLignes
nomDeBaseFichier            nomOs
parcoursRepertoire          renommeFichier
tailleFichier               testeExistence
```

mais vous avez surement compris ce qu'il faut en faire !

2.2 Faut-il une algorithmique objet ?

La *programmation objets* va beaucoup plus loin que la simple réflexion sur les actions à mener pour résoudre un problème. Elle vient aussi structurer les données. Programmer avec des objets, c'est construire des classes d'objets et définir les champs (variables) et les méthodes (modules) *ad hoc*, avec un mécanisme d'héritage hiérarchique. Bref, c'est inévitable ET inabordable pour un(e) novice.

Cela ne signifie pas qu'on ne peut pas écrire des algorithmes avec des objets. Au contraire : il suffit de convenir que la syntaxe

```
objet.module( parametre_1[, parametre_2...] )
```

ou

```
objet --> module( parametre_1[, parametre_2...] )
```

correspond à l'appel du module pour l'objet considéré avec le(s) paramètre(s) indiqué(s) et que la syntaxe `objet[champ]` — ou `objet.champ` si on préfère — désigne la variable correspondante de l'objet.

L'algorithmique objets n'a pas sa place dans l'optique de description des actions à mener pour décrire la méthode de résolution. Par contre, elle prend tout son sens dans un cadre large de programmation, avec une forte dose d'analyse, de spécifications, de contraintes. C'est donc une algorithmique qui dépasse notre tâche modeste, qui ne peut se satisfaire de notre langage de base algorithmique.

Toutefois, pour une initiation à la programmation objets, on peut concevoir d'utiliser les termes suivants :

```
CLASSE          DEBUT_DE_CLASSE          FIN_DE_CLASSE
METHODE         CHAMP                    HERITE_DE
```

Par exemple, pour inventer la classe des noms de fichiers à partir de la classe des chaines de caractères, avec les méthodes `chemin` et `estValide`, on pourrait écrire

```
CLASSE nomsFichier  HERITE_DE chaine
DEBUT_DE_CLASSE
  METHODE chemin
  METHODE estValide
FIN_DE_CLASSE
```

puis rédiger quelque part le texte des algorithmes `nomsFichier.chemin` et `nomsFichier.estValide`. Autre exemple : pour définir la classe des ensembles à partir de la classe des tableaux on pourrait écrire

```
CLASSE ensembles
  HERITE_DE tableaux
DEBUT_DE_CLASSE
  METHODE union
  METHODE intersection
  METHODE complementaire
  METHODE differenceSymetrique
FIN_DE_CLASSE
```

On voit bien sur ces deux exemples en quoi la programmation objets est loin de l'algorithmique traditionnelle : une bonne programmation objets, c'est d'abord une bonne structuration du monde en objets. Les instructions de communication des messages entre objets sont alors presque "évidents". Par contre, et ce peut être un bon compromis, les méthodes des objets peuvent être écrits avec des algorithmes traditionnels, en utilisant la syntaxe décrite ci-dessus. Mais il ne faut pas s'y tromper : la programmation objets, c'est 90 % d'objets et 10 % d'algorithmes !

2.3 Traduire les algorithmes en programmes

La traduction d'une langue à une autre est un art, la traduction d'un algorithme en programme aussi. Chaque langage de programmation a ses avantages, ses inconvénients. Tous ont des contraintes, des astuces, des ambiguités, des limitations machines. Comme pour l'algorithmique, apprendre un langage consiste à connaitre le vocabulaire, à suivre la syntaxe, à s'imprégner de la sémantique.

Le passage de l'algorithme au programme comporte de nombreux pièges. Le typage, par exemple est un passage délicat car il suppose des choix, des connaissances sur l'utilisation du programme. Par exemple la variable `PRIX` de l'algorithmique correspond à un nombre. Pour des fruits et légumes, un typage en *int* (pour le C) ou *integer* (pour le Pascal) est correct. Par contre, s'il s'agit d'une voiture, les *int* et les *integer* sont limités à `32767` donc cela provoquera des erreurs de calcul, éventuellement non détectables à la compilation.

La concision permise par un langage est souvent utilisée au détriment de la lisibilité. Ainsi en C, l'instruction `tabC[ i++] += j++` réalise les trois instructions

```
tabC[ i ] =  tabC[ i ] + j
i         =  i + 1
j         =  j + 1
```

Traduire directement n'est pas toujours possible. Il faut parfois adapter les algorithmes. Par exemple le calcul du maximum du tableau `T` se fait en Apl par les trois caractères ⌈/T et la conversion en l'élimination des espaces en début de chaine de caractère de la variable `C` se fait en Perl par `C=~s/^ //g;`. Et encore, il s'agit d'expressions simples. Pour vous convaincre du chemin entre l'algorithme de départ et les instructions une fois la traduction terminée et optimisée, vous pouvez retourner à la page 9 pour essayer de trouver les algorithmes de départ pour chaque série d'instructions...

Une fois les algorithmes écrits, vérifiés, rédigés, avec des jeux d'essais, des batteries de test, voire des fichiers d'exemples, il y a donc encore un gros travail à fournir, travail plus ou moins important selon le niveau d'expertise dans le langage considéré. Toutefois, il est relativement simple de traduire nos algorithmes en C, Pascal, Perl etc. de façon automatique par programme, étant entendu que la traduction sera alors grossière mais correcte.

C'est d'ailleurs un exercice que nous vous proposons que d'écrire l'algorithme qui traduit un texte d'algorithme en programme, disons en Perl.

Rappelons que notre programme galg disponible sur la page *Web*

http ://www.info.univ-angers.fr/pub/gh/Galg/

effectue ces traductions, au prix de quelques petits aménagements des algorithmes. Par exemple, toute instruction commence par un mot clé et il faut donc écrire

```
AFFECTER x <-- 1
```

au lieu de

```
x <-- 1
```

Heureusement, galg a une option qui rajoute le mot `AFFECTER` quand elle détecte les affectations. De plus galg utilise des commentaires spéciaux comme `#!:` et `#>` pour rajouter des morceaux d'instructions en ligne ou rajouter des sous-programmes. N"hésitez pas à consuter ce site, vous y trouverez de nombreux algorithmes et leurs aménagements directement traduisibles en C, C++, Dbase, Java, Perl, Rexx et Tcl/Tk.

Chapitre 3.

Algorithmes élémentaires

3.1 Permutation de deux variables

Voici un premier problème simple : *"Comment permuter deux variables ?"* Par exemple, si la variable `valA` vaut 2 et la variable `valB` vaut 6, quelles instructions écrire pour que `valA` contienne la valeur de `valB` c'est à dire 6 et que `valB` contienne la valeur de `valA` c'est à dire 2 ? L'algorithme immédiat

```
valA  <--  valB
valB  <--  valA
```

est bien sur faux car dès la première instruction il ne reste plus aucune trace de la valeur de `valA`. Avec un peu de réflexion, on peut trouver divers algorithmes qui viennent résoudre le problème (ce qui va nous amener à établir des critères de choix entre les algorithmes).

Comme première solution, on peut utiliser la méthode des "petits pas" et écrire soigneusement un algorithme en quatre instructions avec une sauvegarde de chacune des variables `valA` et `valB` :

```
# permutation des variables valA et valB
# solution 1 : les "petits pas " (ou : la double sauvegarde)

  sovA  <-- valA // sauvegardes
  sovB  <-- valB

  valA  <-- sovB // affectations
  valB  <-- sovA
```

D'aucuns préféreront une seule sauvegarde et des transpositions menant à ce qu'on appelle en mathématiques une permutation cyclique (ou circulaire) :

```
# permutation des variables valA et valB
# solution 2 : les permutations circulaires

  vin   <-- valA
  valA  <-- valB
  valB  <-- vin
```

Cette méthode s'appelle aussi la méthode "de la bouteille" (d'où l'appellation "`vin`" comme nom de la variable de sauvegarde). Elle correspond aux manipulations à effectuer (rinçage des bouteilles non compris) quand on veut transvaser le contenu de deux bouteilles nommées `valA` et `valB` : on commence par mettre le contenu de `valA` dans une autre bouteille (`vin`) puis on remplit `valA` avec le contenu de `valB` et il ne reste plus qu'à remplir `valB` avec le contenu de `vin`. D'autres encore préféreront ne pas faire de sauvegarde mais de savants calculs, comme :

```
# permutation des variables valA et valB
# solution 3 : les calculs bijectifs

  valA   <-- valA + valB
  valB   <-- valA - valB
  valA   <-- valA - valB
```

Il y a bien sûr bien d'autres algorithmes répondant au problème, notamment ceux qui sont duaux de ceux présentés, (qu'on obtient en interchangeant le nom des variables) ou similaires ; ainsi le dernier algorithme peut se réécrire

```
# permutation des variables valA et valB
# solution 3 bis : d'autres calculs bijectifs

  valB   <-- valB - valA
  valA   <-- valA + valB
  valB   <-- valA - valB
```

Par contre, ce serait maladroit de réécrire un algorithme avec les opérations `*` et `/` à la place de `+` et `-` (même si l'algorithme est acceptables pour des valeurs non nulles).

Un algorithme tel que

```
# permutation des variables valA et valB
# solution 3 ter : calculs bijectifs maladroits

  valA   <-- valA * valB
  valB   <-- valA / valB
  valA   <-- valA / valB
```

est donc déconseillé car il introduit des cas particuliers (celui où `valA` vaut 0, celui où `valB` vaut 0) et peut se révéler incorrect lors de calcul d'arrondi (une division par 3 ou par 7 par exemple, peut ne pas donner le "bon" résultat en pratique).

Si le choix de l'algorithme se fait sur la justesse, il faut rejeter la solution 3 ter. Si on cherche la lisibilité, il faut écarter la solution 3. Les solutions 1 et 2 sont peu différentes et notre préférénce ira à la solution 2 car plus courte. Signalons au passage que ces solutions 1 et 2 sont générales : elles s'appliquent aussi lorsque `valA` et `valB` sont des variables caractères, ce qui n'est pas le cas des solutions calculatoires.

Il peut paraitre étonnant de trouver autant de solutions pour un problème aussi simple. Après réflexion, pour chaque situation il y a souvent de nombreuses façons d'arriver au but. L'analyse et la discussion algorithmique servent aussi à explorer cette diversité des méthodes, à fixer les choix (ou tout au moins à les expliciter), sachant que le libre arbitre et les justifications sont ici affaire de gout et d'esprit plutot qu'application de la science.

3.2 Calcul du maximum d'un tableau

On se propose ici de fournir la valeur `vMax` d'un tableau `tVal` de `NbVal` valeurs numériques ou caractères si on admet qu'on peut utiliser le même opérateur de comparaison `>` aussi bien pour les nombres que pour les chaines.

Avec de très légères modifications, l'algorithme peut donner le maximum d'une structure à condition de disposer d'une "fonction d'évaluation". Par exemple au lieu de chercher le plus grand élément d'un tableau, on pourrait chercher la personne la plus agée si chaque ligne contient le nom, le prénom et l'age des personnes.

L'algorithme suivant détermine le maximum d'un tableau :

```
# Calcul du maximum VMAX du tableau TVAL
# TVAL contient NBVAL valeurs
# dont les indices sont 1, 2... NBVAL

vMax <-- tVal[ 1 ]
POUR indB DE1A NbVal
     eltCour <-- tVal[ indB ]
     SI eltCour > vMax ALORS
        vMax <-- eltCour
     FIN SI { eltCour > vMax }
FIN POUR { indB DE1A NbVal }
```

Remarques : si on avait voulu connaitre la première position du maximum, il aurait suffi de mettre l'instruction

```
indMax <-- 1
```

juste avant la boucle `POUR` et l'instruction

```
indMax <-- indB
```

dans la partie `ALORS` du `SI` sans changer le test soit l'algorithme

```
# Calcul de la première occurence indMax
# du maximum VMAX dans le tableau TVAL
# TVAL contient NBVAL valeurs
# dont les indices sont 1, 2... NBVAL

vMax   <-- tVal[ 1 ]
indMax <-- 1
POUR indB DE1A NbVal
     eltCour <-- tVal[ indB ]
     SI eltCour > vMax ALORS
        indMax <-- indB
        vMax   <-- eltCour
     FIN SI { eltCour > vMax }
FIN POUR { indB DE1A NbVal }
```

Par contre, pour obtenir la dernière position, il faut remplacer le test en `>` par un test en `>=` soit l'algorithme

```
# Calcul de la dernière occurence indMax
# du maximum VMAX dans le tableau TVAL
# TVAL contient NBVAL valeurs
# dont les indices sont 1, 2... NBVAL

vMax   <-- tVal[ 1 ]
indMax <-- 1
POUR indB DE1A NbVal
     eltCour <-- tVal[ indB ]
     SI eltCour >= vMax ALORS
        indMax <-- indB
        vMax   <-- eltCour
     FIN SI { eltCour > vMax }
FIN POUR { indB DE1A NbVal }
```

On remarquera l'utilisation de `eltCour` pour éviter deux fois l'appel par référence. Cela signifie qu'au lieu d'écrire

```
SI tVal[ indB ] > vMax ALORS
   vMax <-- tVal[ indB ]
```

on a préféré mettre la valeur courante du tableau `tVal[ indB ]` dans une variable simple nommée `eltCour` pour écrire

```
eltCour <-- tVal[ indB ]
SI eltCour > vMax ALORS
   vMax <-- eltCour
```

On y gagne en lisibilité.

Un puriste de l'optimisation pourrait faire remarquer que l'on vient la première fois (lorsque `indB` vaut 1) tester `tVall[ 1 ]` avec lui-même et qu'on effectue donc `NbVal` tests alors qu'on pourrait n'en faire que `NbVal-1`. C'est exact. Il suffirait d'écrire

```
POUR indB DE 2 A NbVal
```

pour éviter ce problème mais nos règles algorithmiques ne tolérent pas de commencer à 1.

Il faudrait, pour n'effectuer que `NbVal-1` tests écrire comme début d'algorithme

```
vMax <-- tVal[ NbVal ]
POUR indB DE1A NbVal-1
```

Notre faveur reste cependant à l'algorithme initial et ce, pour les raisons suivantes : il correspond à la méthode naturelle qui consiste à commencer avec la valeur numéro 1, puis avec la valeur numéro 2, etc. ; il n'effectue qu'un seul test de plus, ce qui est négligeable par rapport à `NbVal` si `NbVal` est grand.

Par contre, l'algorithme

```
# Calcul du maximum VMAX du tableau TVAL
# TVAL contient NBVAL valeurs
# dont les indices sont 1, 2... NBVAL

POUR indB DE1A NbVal
  eltCour <-- tVal[ indB ]
  SI indB = 1
  |  ALORS  vMax <-- eltCour
  |  SINON  Si  eltCour > vMax Alors
  |         ||      vMax <-- eltCour
  |         Fin si { eltCour > vMax }
  FIN SI { indB = 1 }
FIN POUR { indB de1a NbVal }
```

est vraiment plus lent que le premier car il effectue systématiquement `NbVal` tests sur `indB`. Par contre il est plus joli (!) grâce aux barres `|` qui indiquent le corps des instructions `SI`.

3.3 Affichages en ordre croissant

La position du premier problème lié à ces affichages en ordre croissant se réduit à l'énoncé succint "afficher par ordre croissant deux nombres entrés par l'utilisateur au clavier". Là encore nous allons essayer de dégager une "bonne solution générale" qui permettra de gérer des variables caractères aussi bien que des nombres. Cet exemple montrera aussi comment un langage algorithmique permet de dégager et d'éclairer les différentes solutions avant de commencer à programmer dans un langage particulier.

A l'énoncé du problème, il ne parait pas judicieux d'écrire d'emblée un algorithme de tri car il n'y a que deux valeurs à gérer. De même, il n' y a pas clairement trois étapes à réaliser : saisir les valeurs, les ranger par ordre croissant puis les afficher car on peut ici réaliser l'affichage et le choix de l'ordre en même temps comme le montre l'algorithme solution 1 :

```
# affichage en ordre croissant : solution 1

ECRIRE "Saisie de deux valeurs et affichage croissant"

{ saisie des valeurs }

  ECRIRE " donner votre premier nombre "
  LIRE     ValeurA
  MCRIRE " et le second  "
  LIRE     ValeurB

{ affichage croissant }

  SI  ValeurA > ValeurB
      ALORS   ECRIRE ValeurA, ValeurB
      SINON   ECRIRE ValeurB, ValeurA
  FIN SI { finsi ValeurA > ValeurB }
```

Le choix du couple écrire / lire pour la demande d'UNE valeur nous parait fondamental. Certains pourraient préférer une seule instruction contenant à la fois le texte de la question et le nom de la variable de réponse mais cela ne se justifie pas : d'une part, `ECRIRE` et `LIRE` sont des opérations différentes ; d'autre part, les instructions `ECRIRE X` et `LIRE Y` peuvent être mises dans un sous-programme `demande` et remplacées par l'appel `demande(X,Y)`

Par contre, le choix de `MCRIRE` ou de `ECRIRE` ou de tout autre mot très proche de `ECRIRE` (pour afficher à l'écran sans passer à la ligne) dans la deuxième question est plus discutable : le fait d'avoir à l'écran sur une même ligne le texte de la question et la valeur de la réponse est plus agréable, plus rapide à l'oeil ; de plus, cela permet de mettre plus d'informations sur un même écran ; au pire (au mieux ?) 24 lignes permettent alors la saisie de 24 valeurs alors qu'avec écrire/lire on ne pourra en avoir que 12. En revanche, cette distinction à l'affichage n'est pas fondamentale pour le problème posé et rajoute un mot de plus au langage algorithmique.

Nous disions que trois étapes ne nous paraissaient pas plus adaptées que deux seulement ; ce choix peut être discuté car le problème n'est pas d'envergure. En admettant la validité d'un tel découpage, le choix de la méthode de rangement en ordre croissant est là aussi sujette à controverse.

On peut penser à permuter les deux variables. On obtient alors, une fois les demandes effectuées, l'algorithme

```
# affichage en ordre croissant : solution 2

{ tri éventuel des valeurs }

SI  ValeurA < ValeurB
    ALORS permuter(ValeurA,ValeurB)
FIN SI  { ValeurA < ValeurB }

{ affichage }

ECRIRE ValeurA, ValeurB
```

avec une fonction "évidente" ou "naturelle" comme `permuter` (nous avons d'ailleurs présenté en début de ce chapitre comment permuter deux variables). On peut aussi calculer et nommer `Vmin` la plus petite valeur, `Vmax` la plus grande, soit l'algorithme

```
# affichage en ordre croissant : solution 3

{ report du minimum et du maximum }

  Vmin  <-- min(ValeurA,ValeurB)
  Vmax  <-- max(ValeurA,ValeurB)

{ affichage }

  ECRIRE Vmin, Vmax
```

là encore avec des fonctions "évidentes" ou "naturelles" comme `min` et `max`.

L'avantage d'un tel algorithme est de disposer des valeurs dans l'ordre initial et dans l'ordre trié, contrairement aux algorithmes précédents qui n'ont plus de trace de l'ordre historique des valeurs.

Par contre, et nous l'avons parfois vu en cours écrit par nos élèves, c'est une erreur de logique que de penser répoudre le problème avec l'algorithme

```
# affichage en ordre croissant : solution fausse

{ report du minimum et du maximum }

ValeurA  <-- min(ValeurA,ValeurB)
ValeurB  <-- max(ValeurA,ValeurB)

{ affichage }

ECRIRE   ValeurA, ValeurB
```

Cette solution serait correcte si le calcul du `min` et du `max` étaient effectués en parallèle, mais ce n'est pas le cas avec la plupart des langages de programmation ni avec notre langage algorithmique.

Quel algorithme choisir ? Notre choix se porterait plutot vers l'algorithme qui effectue le la permutation éventuelle des valeurs par l'appel du module `permuter` car il sépare le clasement de l'affichage. Pour le module `permuter`, avec la syntaxe décrite au chapitre 1, on aurait comme structure de module :

```
MODULE permuter
   PARAMETRES p_valA ,p_valB (écriture)
   LOCALES ...
DEBUT DU MODULE permuter
  ...
FIN DU MODULE permuter
```

Et le corps du module peut être écrit en utilisant les remarques faites sur la permutation de deux variables au déut de ce chapitre.

Si maintenant on essaie de résoudre le même problème mais pour trois variables et plus, les choses se compliquent. Le but fondamental de l'analyse algorithmique est d'éclairer la méthode choisie ; qu'il soit difficile de trouver une méthode ou de choisir entre plusieurs méthodes équivalentes est un autre problème. Ainsi, des divers algorithmes de permutations, celui qui correspond à la méthode de la bouteille est certainement le meilleur, pour peu que les critères de choix soient la généralité, la simplicité. Celui qui utilise des additions et des soustractions est difficile à lire et on si on ne l'a pas déjà expérimenté, on peut douter de son exactitude.

Le même problème de choix, qui *a priori* ne dépend pas du langage, se retrouve si on veut généraliser l'algorithme précédent d'affichage par ordre croissant à trois valeurs, ou même à un nombre quelconque de valeurs.

La méthode la plus générale sera exprimée en termes de saisie, de tri et d'affichage comme dans l'algorithme

```
Saisie(NbVal,tableauL)
TriCroissant(tableauL,NbVal)
AffichageT(tableauL,NbVal)
```

Cela peut paraitre un peu long et compliqué, surtout s'il n'y a que deux ou trois valeurs. Mais l'expérience montre qu'à décomposer et structurer son travail dès le départ est une bonne chose et que cela permet facilement de maintenir les programmes, de les modifier, de les généraliser. Un algorithme qui viendrait d'abord demander si on a deux ou trois valeurs et qui viendrait utiliser deux méthodes différentes suivant le nombre de valeurs poserait une question de trop et se compliquerait trop la vie : un bon algorithme est un algorithme général (générique ?).

Nous allons désormais supposer que la partie saisie a été effectuée et que nous avons trois valeurs distinctes nommées `valA`, `valB` et `valC`. Ne pas vouloir utiliser un module de tri et écrire un algorithme direct de classement des trois valeurs est un choix classique mais il reflère l'esprit tortueux du programmeur. Des `SI` emboîtés introduisent une lourdeur quasi-illisible :

```
SI valA < valB
   ALORS Si valB < valC
            Alors Ecrire valA, valB, valC
            Sinon si valC < valA
                     alors ecrire valC, valA, valB
                     sinon ecrire valA, valC, valB
                  finsi si valC < valA
         Finsi #  valB < valC
   SINON Si valA < valC
            Alors Ecrire valB, valA, valC
            Sinon si valC < valB
                     alors ecrire valC, valB, valA
                     sinon ecrire valB, valC, valA
                  finsi
         Finsi #...
FINSI # lA < valB
```

Avec des SI à conditions multiples, la lisibilité est là encore quasi-nulle, même avec notre effort d'écriture en MAJUSCULES au premier niveau, Majuscule Initiale au second niveau, minuscules au troisième niveau...

```
SI (valA < valB) et (valB < valC)
   ALORS ECRIRE valA, valB, valC
   SINON Si (valA < valC) et (valC < valB)
            Alors Ecrire valA, valC, valB
            Sinon si (valB < valA) et (valA < valC)
                     alors ecrire valB, valA, valC
                     sinon sI (valB < valC) et (valC < valA)
                              aLORS ecrire valB, valC, valA
                              sINON si (valC<valA) et (valA<valB)
                                       alors ecrire valC, valA, valB
                                       sinon ecrire valC, valB, valA
                                    finsi...
                           fINSI (# valB < valC) et (valC < valA)
                  finsi # (valB < valA) et (valA < valC)
         Finsi # (valA < valC) et (valC < valB)
FINSI # (valA < valB) et (valB < valC)
```

Une solution un peu plus "propre" de classement des trois valeurs serait une structure de cas traditionnelle :

```
aux cas ou
 cas (valA < valB) et (valB < valC)
      ecrire valA, valB, valC
 cas (valA < valC) et (valC < valB)
      ecrire valA, valC, valB
 cas (valB < valA) et (valA < valC)
      ecrire valB, valA, valC
 cas (valB < valC) et (valC < valA)
      ecrire valB, valC, valA
 cas (valC < valA) et (valA < valB)
      ecrire valC, valA, valB
 cas (valC < valB) et (valB < valA)
      ecrire valC, valB, valA
fin des cas
```

mais on "voit" tout de suite que cette solution ne sera pas adaptée si on a 4, 5... 100 valeurs car pour n valeurs on a $n!$ cas à envisager.

L'écriture "curieuse" des mots réservés (initiale majuscule, deuxième lettre en majuscule...) de l'algorithme avec des `SI` à condtitions multiples permet de vérifier l'emboitement des structures. On peut aussi prendre l'habitude de mettre la première structure en majuscules, la deuxième avec initiale majuscule et la troisième en minuscule, mais cela freine parfois la frappe. Par contre, on peut utiliser un programme annexe ou un bon éditeur de textes pour gérer cette mise en forme. Cela impose aussi de ne pas dépasser trois niveaux d'imbrication, ce qui nous parait un maximum : au delà, autant appeler des sous-programmes pour garder la lisibilité. Les deux algorithmes présentés avec de nombreux `SI` sont donc inacceptables selon nos principes.

Tous ces algorithmes ne doivent pas faire oublier une hypothèse importante (non exploitée jusqu'ici) que les nombres entrés doivent être différents. Nous laissons au lecteur le soin de démêler les cas possibles s'il faut rajouter des `SI (valA <= valB)` et des `si (valA = valB)`... ce qui justifie là encore le recours à un module de tri qui gère tous les cas.

Signalons enfin que mathématiquement, pour trois variables, des formules donnent le même résultat que ces algorithmes ; toutefois, comme pour la permutation sans troisième variable, ces algorithmes demandent beaucoup d'efforts de compréhension (voire une démonstration) , tel

```
VALu  <-- min( valA, min(valB,valC) )
VALd  <-- min( max(valA,valB),
               min( max(valA,valC),max(valB,valC) ) )
VALt  <-- max( valA, max(valB,valC) )
ECRIRE VALu, VALd, VALt { comme Un, Deux, Trois }
```

On viendra certainement refuser aussi cet algorihtme car difficile non pas à lir mais à comprendre. De plus, si les valeurs sont des chaines des caractères et non pas des nombres, les fonctions `min` et `max` sont à réinventer.

Donc si le problème doit se généraliser à n valeurs, on se retrouve avec un problème classique de tri. Et même si classer 4 valeurs parait anodin ou peu important à réaliser, il faut quand même écrire un programme de tri pour le faire. Comme il existe de nombreuses méthodes de tri, cela ne doit pas être trop dur.

Un tri comme celui par recherche du maximum ou par la méthode des bulles est alors suffisant en première approche, et beaucoup plus rapide à écrire qu'un tri évolué sauf à en avoir déjà écrit.

Par exemple ici, pour moins de 50 valeurs, on peut se contenter de l'algorithme :

```
# tri des NbVal valeurs du tableau Tval
# méthode des bulles

POUR indA DE1A NbVal-1

     indB <-- IndA + 1
     TANT QUE indB <= NbVal
          SI Tval[ IndA ] > Tval[ IndB ] ALORS
             # variante de permutation pour un tableau
             permuterDans( Tval, IndA, indB )
          FINSI { Tval[ IndA ] > Tval[ IndB ] }
          indB <-- IndB + 1
     FIN TANT QUE indB <= NbVal

FIN  POUR indA DE1A NbVal-1
```

D'autres problèmes similaires dont l'énoncé est relativement simple méritent une attention soutenue car ils cachent des difficultés de résolution (ou de lenteur d'exécution). Citons notamment celui qui consiste à trouver les k plus grandes valeurs d'une liste de n éléments, celui qui demande de trouver la plus grande sous-suite de nombres croissants dans un tableau, ou la sous-suite de plus grande somme, etc.

3.4 Normalisation d'un tableau

Voici un petit algorithme classique qui justifie encore à nos yeux le recours aux algorithmes : il s'agit de la normalisation d'un tableau de nombres strictements positifs. Cette opération consiste à diviser tous les éléments du tableau par le plus grand élément de façon à n'avoir que des nombres compris entre 0 et 1. Ce problème simple, comme d'autres, peut être mal résolu, par exemple quand on veut bien faire en utilisant un indice de tableau plutôt que la valeur dans le tableau, comme nous allons le montrer avec l'algorithme qui suit.

Traduire l'algorithme suivant en programme

```
# Normalisation du tableau TBLN avec TAILLE valeurs
# on divise par la plus grande valeur du tableau

# 1. Recherche de l'indice du max

Indi <-- 1
POUR Jones DE1A TAILLE
     SI TBLN[ Jones ] > TBLN[ Indi ] ALORS
        Indi <--  Jones
     FINSI # TBLN[ Jones ] > TBLN[ Indi ]
FIN POUR Jones # DE1A TAILLE

# 2. Division (enfin, presque) par le max

POUR Jones DE1A TAILLE
     TBLN[ Jones ] / TBLN[ Indi ]
FIN POUR Jones # DE1A TAILLE
```

aboutira, quelque soit le langage à une erreur. Non que le langage soit en cause : c'est la méthode qui est mauvaise. Car la division de `TBLN[ Jones ]` par `TBLN[ Indi ]` écrase, lorsque `Jones` vaut `Indi` la valeur du maximum. L'algorithme, testé "à la main" permet de valider la méthode. La traduction ajoutera des problèmes annexes. Autant séparer les problèmes.

Ainsi, il suffit de sauvegarder la valeur du maximum pour disposer d'un algorithme correct et il n'y a plus qu'à traduire :

```
# Normalisation du tableau TBLN avec TAILLE valeurs
# on divise par TMAX, plus grande valeur du tableau

# 1. Recherche de l'indice du max

Indi <-- 1
POUR Jones DE1A TAILLE
     SI TBLN[ Jones ] > TBLN[ Indi ] ALORS
        Indi <--  Jones
     FINSI # TBLN[ Jones ] > TBLN[ Indi ]
FIN POUR Jones # DE1A TAILLE
```

```
# 2. Division par le max

TMAX <-- TBLN[ Indi ]
POUR Jones DE1A TAILLE
     TBLN[ Jones ] / TMAX
FIN POUR Jones # DE1A TAILLE
```

3.5 Découpages : nom de fichier, URL, e-adresse

Pour tous ces problèmes, relativement simples, sur variables caractères, le principe est le même : on cherche un caractère spécial, par exemple `@` ou `\` et on sépare le texte avant et après le caractère.

Prenoms par exemple le découpage d'un nom de fichier sous *Unix* (nous discuterons le découpage sous *Dos/Windows* un peu plus loin). Il s'agit, à partir d'un nom comme

```
/home/info/users/JeanJean/Textes/roman.txt
```

de trouver le chemin d'accès, soit ici `/home/info/users/JeanJean/Textes` et le nom du fichier, soit ici `roman.txt`.

Supposons un instant que

- la fonction `dernPos` donne la dernière position d'une chaine dans une autre avec la syntaxe `dernPos( aiguille, botte )` pour rechercher la chaine `aiguille` dans la chaine `botte`,
- que la fonction `sousChaine` permette d'extraire une chaine à l'intérieur d'une autre chaine avec la syntaxe

  ```
  sousChaine( chaineOrg , debut , nbc )
  ```

  pour extraire les `nbc` caracères à partir du caractère numéro `debut` dans la chaine `chaineOrg`
- que la fonction `Longueur` donne le nombre de caractères d'une chaine.

Afin de couvrir tous les cas, on supposera également que la fonction `dernPos` renvoie 0 si la chaine n'est pas vue, que la fonction `sousChaine` tronque sans erreur si on demande trop de caractères, qu'elle renvoie la chaine vide si on commence à extraire après le dernier caractère etc.

L'algorithme est alors, si `nomLong` contient toute la spécification du fichier :

```
# découpage de nomLong (unix) en nomCourt et chemin
lngN <-- longueur(nomLong)
dps  <-- dernPos( "/" , nomLong )
si dps=0
   alors # pas de slash
         nomCourt <-- nomLong
         chemin   <-- ""
   sinon # avec slash
         nomCourt <-- sousChaine( nomLong , dps+1 , lngN-1 )
         chemin   <-- sousChaine( nomLong , 1     , dps-1  )
finsi dps=0
```

Sous *Dos/Windows*, il gérer le nom d'unité, comme `D:` dans la spécification du fichier `D:\Utilisateur\JeanValjean\Textes\Roman.Txt` ; malheureusement, si aucune unité n'est spécifiée, on ne peut guère en inventer une. Nous choisissons de mettre `C:` qui n'est sans doute pas le seul bon choix mais qui, faute de plus amples renseignements, sera considéré comme "correct". L'algorithme devient

```
# découpage de nomLong (dos) en nomCourt, chemin et unite
lngN <-- longueur(nomLong)
dps  <-- dernPos( "\" , nomLong )
si dps=0
   alors # pas d'anti-slash
         nomCourt <-- nomLong
         chemin   <-- ""
         unite    <-- "C:"
   sinon # avec anti-slash
         nomCourt <-- sousChaine( nomLong , dps+1 , lngN-1 )
         iddp     <-- dernPos( ":" , nomLong )
         si iddp=0 # pas de :
            alors chemin   <-- sousChaine( nomLong , 1      , dps-1  )
                  unite    <-- ""
            sinon chemin   <-- sousChaine( nomLong,  iddp+1 , dps-1  )
                  unite    <-- sousChaine( nomLong,  1      ; iddp   )
         finsi iddp= 0
finsi dps=0
```

Passons maintenant au découpage d'une URL. C'est pratiquement le même algorithme que celui du découpage du nom de fichier sous *Dos* car une *URL* ou *Uniform Resource Locator* se compose au maximum d'un nom de protocole, suivi des symboles `://` puis d'un chemin d'accès avec des `/` et d'un nom de document. D'où l'algorithme :

```
# découpage de URL en protocole, chemin et nomDocument
lngN <-- longueur(URL)
dps  <-- dernPos( "/" , URL )
si dps=0
   alors # pas de slash
         nomDocument <-- URL
         chemin      <-- ""
         protocole   <-- ""
   sinon # avec slash
         nomDocument <-- sousChaine( URL , dps+1 , lngN-1 )
         iddp        <-- dernPos( "://" , URL )
         si iddp=0 # pas de "://"
            alors chemin    <-- sousChaine( URL , 1      , dps-1  )
                  protocole <-- ""
   # on va a la fin de :// soit 3 car de plus que la position du :
            sinon chemin    <-- sousChaine( URL,  iddp+3 , dps-1  )
                  protocole <-- sousChaine( URL,  1      , iddp-1 )
         finsi iddp= 0
finsi dps=0
```

Pour une $e-adresse$, c'est à dire une adresse-mail, comme

```
victor.Hugo@auteurs.net.fr
```

c'est le même principe : le nom de l'utilisateur est avant `@`, le nom de domaine est après le dernier point et la localisation est toute la chaine apres `@`. L'utilisateur est donc ici `victor.Hugo`, le nom de domaine est `fr` et la localisation `auteurs.net.fr`.

Comme pour tous les autres algorithmes élémentaires de ce chapitre, nous supposerons que les chaines entrées sont correctes et qu'il ne s'agit que de faire le découpage. Si par exemple l'*eadr* entrée était

```
victor.Hugo@auteurs.net.fr@bonjour.com
```

notre algorithme ne s'appliquerait pas.

Ce choix soulève encore un point à indiquer dans les commentaires de l'algorithme : fait-on un découpage pour tous les cas de figure, en détectant au passage les erreurs, les incohérences ou assure-t-on le minimum juste pour les bons cas ?

Voici l'algorithme de découpage de l'adresse-mail

```
# découpage de EADR en NOMUT, LOCALISE et NOMDOMAINE
# on suppose que eadr est valide
# grace au retour de la fonction testeAdr( eadr )

lngN <-- longueur(eadr)
dpa  <-- dernPos( "@" , eadr )
si dpa=0
   alors # pas d'arrobas
         nomUt      <-- eadr
         localise   <-- ""
         nomDomaine <-- ""
   sinon # avec arrobas
         nomUt      <-- sousChaine( eadr , 1     , dpa-1 )
         idp        <-- dernPos( "." , eadr )
         si idp=0 # pas de "."
            alors nomDomaine <-- sousChaine( eadr , dpa+1  , lngN  )
                  localise   <-- nomDomaine
            sinon nomDomaine <-- sousChaine( eadr ,  idp+1 , lngN  )
                  localise   <-- sousChaine( eadr ,  dpa+1 , lngN  )
         finsi idp= 0
finsi dpa=0
```

Deux techniques sont envisageables :

1. on utilise une fonction de test (par exemple avec des expressions régulières) et en cas de réussite on effectue le découpage
2. on effectue le test dans le module de découpage et on renvoie une valeur pour indiquer la réussite ou l'échec du découpage

Ces deux techniques sont équivalentes et "bonnes" car elles testent toutes les deux l'adresse. Un algorithme qui ne testerait pas complétement l'adresse serait "mauvais" selon nos critères. Il renverrait bien le nom d'utilisateur `victor.Hugo` pour l'adresse

```
victor.Hugo@auteurs.net.fr@bonjour.com
```

mais n'indiquerait pas que l'adresse est pour le moins suspecte...

Chapitre 4.

Algorithmes standards

4.1 Nombre d'occurences du maximum

On se propose de calculer ici la valeur `VMAX` du plus grand élément du tableau `TVAL` et le nombre de fois `NBOCC` où il apparait.Le faire en deux boucles est élémentaire : la première calcule le maximum, la seconde compte le nombre de fois où il apparait, soit l'algorithme :

```
# Nombre d'occurrences du maximum

# 1. Calcul du maximum VMAX du tableau TVAL
#    (TVAL contient NBVAL valeurs)

 vMax <-- tVal[ 1 ]
 POUR indB DE1A NbVal
      SI tVal[ indB ] > vMax ALORS
         vMax <-- tVal[ indB ]
      FIN SI { tVal[ indB ] > vMax }
 FIN POUR { indB DE1A NbVal }

# 2. Calcul du nombre d'occurrences NbOcc de VMAX

 NbOcc <-- 0
 POUR indB DE1A NbVal
      SI tVal[ indB ] = vMax ALORS
         NbOcc <-- NbOcc + 1
      FIN SI { tVal[ indB ] > vMax }
 FIN POUR { indB DE1A NbVal }
```

Pour calculer `VMAX` et `NBOCC` en une seule boucle, il suffit d'envisager trois cas pour la valeur courante `tVal[ indB ]` du tableau : si cette valeur est inférieure à `VMAX`, on peut l'ignorer. Si elle est strictement supérieure à `VMAX`, il faut donner cette nouvelle valeur à `VMAX` et mettre `NBOCC` à 1 ; enfin, si cette valeur est égale à `VMAX`, il faut rajouter 1 à `NBOCC`, soit l'algorithme :

```
##  Nombre d'occurrences NBOCC du maximum TVAL
##  dans le tableau TVAL avec NBVAL valeurs
##  (en une seule boucle)

# initialisation du maximum et de son nombre
# d'occurrences à partir du dernier élément
# du tableau

vMax  <-- tVal[ NbVal ]
NbOcc <-- 1

# boucle de parcours du tableau
# on a déjà pris le dernier élément, donc
# on n'utilise que les NbVal-1 premiers éléments

POUR indB DE1A NbVal-1

     SI tVal[ indB ] > vMax
     |  ALORS # nouveau maximum
     |        vMax  <-- tVal[ indB ]
     |        NbOcc <-- 1
     |  SINON si  tVal[ indB ] = vMax alors
     |             # une occurence de plus pour
     |             # l'ancien maximum
     |             NbOcc <--  NbOcc + 1
     |        finsi { tVal[ indB ] = vMax }
     FIN SI { tVal[ indB ] > vMax }

FIN POUR { indB DE1A NbVal }
```

4.2 Moyenne et écart-type d'un tableau

Commençons par un peu de réflexion sur la notion de moyenne. Deux petites entreprises déclarent chacune que le salaire moyen des employés est de `15 000 Frs`. Seulement, la première entreprise compte deux employés, de salaire respectif `15 000 Frs` et `15 000 Frs` et la deuxième entreprise compte trois, de salaire respectif `8 000 Frs` et `10 000 Frs` et `27 000 Frs`. Il faut donc se méfier de la moyenne car elle ne révèle pas la disparité entre les valeurs, mais seulement leur somme.

C'est pourquoi en statistique descriptive classique on calcule aussi l'écart-type σ des valeurs, qui correspond à la distance moyenne entre les valeurs et leur moyenne m. On trouve respectivement `0 Frs` et `8 524 Frs` comme écart-type ce qui permet d'avoir une idée de la répartition des salaires autour de la moyenne.

Cette remarque a pour but d'indiquer que le calcul seul de la moyenne n'est pas un bon indicateur et et qu'il faut systématiquement lui adjoindre celui de l'écart-type (et bien sûr le nombre de valeurs mis en jeu).

Sans plus d'effort il faut aussi calculer le coefficient de variation qui n'est jamais que le pourcentage correspondant à σ/m et qui permet de comparer, en relatif et non plus en absolu, deux séries de valeur qui n'ont pas la même moyenne.

Il y a deux formules pour calculer l'écart-type σ de n valeurs X_1, X_2...X_n. La plus adaptée à nos besoins est ici

$$\sigma \;=\; \sqrt{\frac{1}{n}\sum_{i=1}^{n} {X_i}^2 \;-\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right)^2}$$

car elle permet de calculer en une seule boucle la moyenne et l'écart-type. Il ne faut pas se laisser abuser par l'apparente difficulté de la formule pour un(e) non-mathématicien(ne) : elle correspond seulement à la racine de la différence entre la moyenne des carrés et le carré de la moyenne.

L'algorithme de la page suivante calcule tout ce qu'il faut. Pour bien le lire, il faut savoir que la variable `Moy` correspond à la moyenne m, que `Sig` correspond à l'écart-type σ et que `NbVal` est le nombre à n de valeurs ; quant au tableau des valeurs, l'expression `TabVal[ ind ]` correspond à l'élément X_i.

```
#########################################################
#                                                       #
#   Résumé statistique d'une variable quantitative      #
#   (valeurs avec unité comme le kg, le cm...)          #
#                                                       #
#########################################################

# Calcul de la moyenne MOY, de l'écart-type SIG
# et de leur rapport CDV pour NBVAL valeurs
# dans le tableau TABVAL

# 1. Calcul de la somme SOM des valeurs
#    et de la somme SOC des carrés

Som <-- 0
Soc <-- 0

POUR ind DE1A NbVal
     valC <-- TabVal[ ind ] # valeur courante
     Som  <-- Som + valC
     Soc  <-- Soc + valC*valC
FIN_POUR ind DE1A NbVal


# 2. On en déduit la moyenne MOY des valeurs
#    et la moyennne MOC des carrés

Moy <-- Som / NbVal
Soc <-- Soc / NbVal
Sig <-- racine( Moc - Moy )

# et on calcule le coefficient de variation
# si la moyenne n'est pas nulle

SI  Moy <>0
    ALORS Cdv <-- 100*Sig/Moy
    SINON Cdv <-- (-1)
FIN_SI  Moy <>0
```

4.3 Construction de dictionnaires

Soit `nomFichTT` une variable caractère qui contient le nom d'un fichier texte, `"Rapport.txt"` par exemple. On voudrait en connaitre le nombre de mots et en construire le lexique (ou dictionnaire) c'est à dire donner la liste des mots et le nombre de fois où ils apparaissent. Pour que la consultation soit aisée, nous allons produire le fichier `Dico.Alp` qui contient les mots par ordre alphabétique croissant et le fichier `Dico.Ocd` qui contient les mots par ordre décroissant d'occurences : on verra donc d'abord les mots les plus fréquents et les *happax* (mots qui n'apparaissent qu'une seule fois) en fin de fichier.

Le premier algorithme que nous allons utiliser profite des tableaux associatifs et de la boucle `POUR_CHAQUE`. Il est donc très court (hors commentaires). Nous utilisons

- la fonction `motNumero` pour extraire le i-ème mot d'une phrase,
- la fonction `indefini( tabAssoc[ indice ] )` pour savoir si le tableau contient l'indice,
- les fonctions `FormatCar` et `Format` pour justifier en sortie les chaines à gauche et les nombres à droite,
- le module `TrieTableauAssoc` pour trier par ordre croissant le tableau associatif passé en paramètre
- le module `TrieFichierSuivant` qui trie (numériquement) par ordre décroissant le fichier passé en paramètre suivant le n-ième mot, n étant passé en paramètre.

Détaillons ce que fait l'algorithme :

1. On parcourt le fichier texte donné en une boucle "tant que" et effectue une lecture ligne par ligne. Pour chaque ligne lue, on compte le nombre de mots qu'on additionne au total général des mots. Chaque mot de la ligne est ensuite passé en revue. S'il existe déjà comme indice dans le tableau associatif, on incrémente de 1 son nombre d'occurrences ; sinon, on en fait une nouvelle entrée avec un nombre d'occurences égal à 1.
2. Une fois le fichier lu, on trie le tableau associatif et on l'affiche de façon bien cadrée dans le fichier alphabétique et dans le fichier des occurences.
3. Il ne reste plus qu'à trier le fichier des occurences pour conclure.

Voici donc l'algorithme :

```
#########################################################
#                                                       #
#   Création des dictionnaires alphabétique             #
#   et d'occurence du fichier nommé nomFichT            #
#                                                       #
#########################################################

##        Version 1 : TABLEAU ASSOCIATIF et boucle POUR_CHAQUE

## 1. Parcours du fichier texte et comptage des
##    mots à la volée, ligne par ligne pour chaque mot
##    de la ligne

#    FICT est la variable de fichier associée à nomFichT
#    NBMT est le nombre de mots en tout
#    NBML est le nombre de la ligne courante
#    TDIC est un tableau associatif dont les indices sont les mots
#         et dont les valeurs sont les nombre de fois où le mot est vu

OUVRIR nomFichT EN_LECTURE COMME ficT

NbMt <-- 0

TANT_QUE NON fin_de_fichier( ficT )

     LIRE ligneTexte SUR ficT # ligne courante
     NbMl <-- NbMots( ligneTexte )
     # rajout au total des mots
     NbMt <-- NbMt + NbMl

     # gestion de chacun des mots de la ligne

     POUR indM DE1A NbMl
          # gestion du mot courant
          motC <-- MotNumero( indM , ligneTexte
          SI indefini( TDIC[ motC ])
             ALORS TDIC[ motC ] <-- 1
             SINON TDIC[ motC ] <-- TDIC[ motC ] + 1
          FIN_SI indefini( TDIC[ motC ])
     FIN_POUR indM DE1A NbMl

FIN_TANT_QUE NON { fin_de_fichier( ficT ) }
```

```
# 2. Tri du dictionnaire et recopie
#    dans le fichier Dico.Alp
#    (ordre alphabétique croissant)
# on compte au passage le nombre NbDifM
# de mots différents

dicAlpha <-- TrieTableauAssoc( tDic )

OUVRIR FdicAlpha EN_ECRITURE COMME "Dico.Alp"

NbDifM <-- 0
POUR_CHAQUE motC DANS dicAlpha
   NbDifM <--  NbDifM  + 1
   ECRIRE FormatCar( motC , 20 ) ,
          Format( dicAlpha[ motC ] , 6) SUR  FdicAlpha
FIN POUR_CHAQUE motC DANS dicAlpha

FERMER FdicAlpha

# 3. Tri du fichier des mots et recopie
#    dans le fichier Dico.Ocd
#    (ordre d'occurences décroissant)

OUVRIR FdicOccd EN_ECRITURE COMME "Dico.Ocd"

#  on trie sur le mot numéro  2 (nb d'occurences)

TrieFichierSuivant(  FdicAlpha , FdicOccd , 2 )
```

Détaillons maintenant le détail du module `TrieFichierSuivant`. On utilise trois paramètres : `ficEnt`, `ficSor`, `posCrit`. `ficEnt` correspond au fichier d'entrée, `ficSor` correspond au fichier de sortie et `posCrit` indique sur quel mot de la ligne doit se faire le tri. Il s'agit ici d'un tri numérique décroissant. Tout utilisateur d'*Unix* pourra, au lieu des instructions ce module, faire un appel-sytème pour utiliser la commande `sort` dans l'instruction

```
sort -r -n -k 2 Dico.alp  -o Dico.Ocd
```

En effet, l'option `-r` (pour *reverse* en anglais) trie par ordre décroissant, l'option `-n` (pour *numerical* en anglais) trie numériquement et non pas par ordre ascii, l'option `-k` (pour *keyword* en anglais) avec le paramètre 2 indique qu'il faut trier selon le 2ème mot.

Comme ce module est délicat et que de nombreuses explications ne valent pas un bon exemple, il est abondamment commenté par des exemples, aussi bien en début d'algorithme qu'en fin d'algorithme :

```
#########################################################
#                                                       #
#   Tri d'un fichier selon le n-ième mot                #
#                                                       #
#########################################################

#  Méthode : la ligne NUMLIG du fichier FICTXT
#  devient l'élément NUMLIG du tableau TLIG ;
#  au passage, le mot numéro N passé en paramètre
#  devient  l'élément NUMLIG du tableau TMOT.
#
#  Lorsqu'on trie TMOT, on permute au passage les
#  éléments correspondants de TLIG. Enfin, on
#  recopie dans le fichier de sortie

MODULE TrieFichierSuivant

       PARAMETRES    nFicEnt (lecture) # nom du fichier d'entrée
                     nFicSor (lecture) # nom du fichier de sortie
                     N       (lecture) # numéro du mot pour le tri

## en principe, après le module, le fichier d'entrée, qui était en
## ordre alphabétique sur le premier mot, est trié par ordre
## numérique sur le deuxième mot (qui est un nombre).

## cela doit permettre de passer par exemple de

##      Avion    10
##      Avions    3
##      Bateau   12

##  à

##      Bateau   12
##      Avion    10
##      Avions    3
```

```
GLOBALES :
        (aucune)


LOCALES (toutes en écriture)
        vFicEnt     # variable fichier d'entrée
        vFicSor     # variable fichier de sortie
        NumLig      # nombre de lignes dans le fichier
        ligneTexte  # ligne courante sur fichier
        MotN        # mot courant numéro N
        indA        # indice de boucle principal pour le tri
        indB        # indice de boucle secondaire pour le tri
        MotTMP      # mot temporaire pour permutation des mots
        LigTMP      # ligne temporaire pour permutation des lignes

DEBUT DU MODULE TrieFichierSuivant

#  1. Lecture de FICTXT

   OUVRIR nficEnt COMME vficEnt
   NumLig <-- 0
   TANT_QUE NON { fin_de_fichier( vficEnt ) }
       LIRE ligneTexte SUR vficEnt # ligne courante
       MotN <-- MotNumero( N , ligneTexte )
       NumLig <--  NumLig  + 1
       TLIG[ NumLig ] <--  ligneTexte
       TMOT[ NumLig ] <--  MotN
   FIN_TANT_QUE NON { fin_de_fichier( vficEnt ) }
   FERMER vficEnt

## -- La partie suivante effectue le tri.
## -- Attention : en cas d'ex-aequo, aucun tri supplémentaire
## -- n'est effectué. Il faudrait vérifier si on passe de
##
##      Abstention    3
##      Non           5
##      Oui           3
##    à                        ou   à
##      Non           5              Non           5
##      Abstention    3              Oui           3
##      Oui           3              Abstention    3
```

```
#
#  2. Tri décroissant numérique de TMOT (et de TLIG au passage)
#     -- méthode des bulles
#

     POUR indA DE1A NumLig-1

          indB <-- IndA + 1

          TANT QUE indB <= NumLig
               SI TMOT[ IndA ] < TMOT[ IndB ] > ALORS
                    # permutation dans TMOT
                      MotTMP       <-- TMOT[ IndA ]
                      TMOT[ IndA ] <-- TMOT[ IndB ]
                      TMOT[ IndB ] <-- MotTMP
                    # puis permutation dans TLIG
                      LigTMP       <-- TLIG[ IndA ]
                      TLIG[ IndA ] <-- TLIG[ IndB ]
                      TLIG[ IndB ] <-- LigTMP
               FINSI TMOT[ IndA ] < TMOT[ IndB ] >
               indB <-- IndB + 1
          FIN TANT QUE indB <= NumLig

     FIN  POUR indA DE1A NumLig-1

#
#  3. Ecriture de TLIG dans vficSor
#


   OUVRIR nficSor COMME vficSor

   POUR indA DE1A NumLig
        ECRIRE FormatCar( TLIG[ indA ] , 26 ) SUR  vficSor
   FIN POUR indA DE1A NumLig

   FERMER vficSor

FIN DU MODULE TrieFichierSuivant
```

Le second algorithme que nous présentons résoudre pour le même problème utilise des tableaux classiques : au lieu d'une affectation précédente comme

```
TDIC[ "valeur" ] <-- 1
```

où le mot courant, `"valeur"` est un nouveau mot, on écrit

```
TMOT[ 12 ] <-- "valeur"
```

où 12 est ici le premier indice non utilisé dans le tableau. Le nombre d'occurences du mot est au même indice dans le tableau `TOCC`, ce qui rajoute ici l'affectation

```
TOCC[ 12 ] <-- 1
```

Pour savoir si un mot est dans un tableau, on utilise la fonction `position` qui renvoie 0 si le mot n'est pas dans le tableau ou l'indice auquel on trouve le mot dans le tableau. Nous laissons le soin au lecteur d'écrire les modules cités. Pour le module `Trietableaux` on pourra s'inspirer de la partie tri du module `TrieFichierSuivant`.

```
#########################################################
#                                                       #
#   Création des dictionnaires alphabétique             #
#   et d'occurence de la variable fichier FICT          #
#                                                       #
#########################################################

##        Version 2 : tableaux et boucles classiques

# 1. Parcours du fichier texte et comptage des
#    mots à la volée

#    FICT est la variable de fichier associée à nomFichT
#    NBMT   est le nombre de mots en tout
#    NBML   est le nombre de la ligne courante
#    NBDIFM contient le nombres de mots différents
#    TMOT   est le tableau dont les indices
#           sont numériques et les valeurs sont les mots
#    TOCC   est le tableau  qui contient en même
#           position le nombre d'occurence du mot
```

```
NbMt   <-- 0
NbDifM <-- 0
OUVRIR nomFichT EN_LECTURE COMME ficT

TANT_QUE NON fin_de_fichier( ficT )

     LIRE ligneTexte SUR ficT # ligne courante
     NbMl <-- NbMots( ligneTexte )
     NbMt <-- NbMt + NbMl

     # gestion de chacun des mots de la ligne

     POUR indM DE1A NbMl
          # gestion du mot courant
          motC <-- MotNumero( indM , ligneTexte )
          pdM  <-- position( motC , TMOT )
          SI pdM = 0
             ALORS NbDifM <-- NbDifM + 1
                   TMOT[ NbDifM ] <-- motC
                   TOCC[ NbDifM ] <-- 1
             SINON TOCC[ pdM ]    <-- TOCC[ pdM ] + 1
          FIN_SI pdM = 0
     FIN_POUR indM DE1A NbMl

FIN_TANT_QUE NON { fin_de_fichier( ficT ) }


# 2. Tri du dictionnaire et recopie
#    dans le fichier Dico.Alp
#    (ordre alphabétique croissant)

TrieTableaux( tMot, tOcc )

OUVRIR FdicAlpha COMME "Dico.Alp"

POUR indM DE1A NbDifM
   motC <-- tMot[ indM ]
   nbOc <-- tOcc[ indM ]
   ECRIRE FormatCar( motC , 20 ) ,
          Format( nbOc , 6) SUR  FdicAlpha
FIN_POUR indM DE1A NbDifM

FERMER FdicAlpha
```

```
# 3. Tri du fichier des mots et recopie
#    dans le fichier Dico.Ocd
#    (ordre d'occurences décroissant)

OUVRIR FdicOccd COMME "Dico.Ocd"

#  on trie sur le mot numéro  2 (nb d'occurences)

TrieFichierSuivant(  FdicAlpha , FdicOccd , 2 )
```

Il y a une grande différence entre les deux algorithmes, que ne reflète peut-être pas l'écriture ou qui ne se voit pas à la première lecture : le premier est court, prêt à l'emploi pour un langage qui implémente les tableaux associatifs comme Perl ou SmallTalket toutes les fonctions citées existent déjà dans ce langage. Le second est long, oblige à écrire tous les modules, toutes les fonctions. Par contre, il peut être utilisé pour tout langage avec tableau, comme en C clasique ou en Pascal.

Pour conclure :

Ecrire un programme c'est comme faire un voyage

Pour un petit voyage, il suffit de pas grand chose. Par contre, pour un grand voyage, il faut s'y prendre à l'avance, se préparer, choisir l'itinéraire, éventuellement noter les horaires sur un papier, ou faire une liste des villes importantes...

Pour la programmation, on est exactement dans la même situation. Pour écrire "3 bouts de nombre", pas besoin d'une algorithmique sophistiquée, voire même pas d'algorithique du tout. Par contre, dès que le travail à effectuer devient d'ampleur, il faut s'armer de patience, acheter du café ou du jus d'orange et se mettre à réfléchir.

Le programme n'est que le bout de la chaine. Il vient traduire en un langage compréhensible par la machine nos choix, notre savoir faire. Son objectif est souvent de faire vite, d'être à la fois correct, complet et "incassable". Sa rédaction est parfois la n-ième traduction et adaptation de l'algorithme. Dans le pire des cas, ce n'est plus qu'un fichier exécutable sans source, sans origine.

Le document algorithmique est la seule preuve tangible que l'on sait ce que l'on veut. Il atteste seul des choix, de la méthode, des options retenues. Lorsque quelques semaines, quelques mois, ou pire, quelques années plus tard on veut reprendre le programme, le modifier, que les commentaires sont inexistants, que les forêt des astuces du langage cache l'arbre du raisonnement, on se dit << *si j'avais su !* >>.

Ecrire un algorithme, c'est réfléchir au problème, c'est formaliser sa pensée sans être contraint par un langage écrit pour une machine, c'est codifier la progression, l'enchainement des étapes en un tout cohérent, lisible.

Si la machine était plus intelligente, il suffirait de lui donner les algorithmes pour qu'elle écrive les programmes à notre place. Elle commence d'ailleurs à le faire (sans interface, sans optimisation) puisque les algorithmes sont des textes déjà sur fichier, déjà prêts à être traduits.

Pour en finir avec la métaphore du voyage, essayez de vous rappeler un grand voyage que vous avez fait. Vous rappelez-vous du trajet, du véhicule ? Le langage de programmation est le véhicule, il est obligatoire, ce qui ne signifie pas qu'il est très important : une bonne voiture en vaut une autre. Par contre rien ne vaut votre gout, votre choix de la destination. Un dernier mot qui milite pour le langage algorithmique : si vous utilisez *Word* sous *Windows*, ou si vous utilisez *Latex* sous *Linux*, savez-vous dans quel langage ces logiciels ont été programmés ? Est-ce important ? N'est-ce pas plus important de comprendre les algorithmes que ces logiciels utilisent, pour créer la table des matières, par exemple ?

Pour aller plus loin

Si ces quelques pages vous ont appris à conduire, il reste encore à passer du stade "nouveau chauffeur" à "conducteur expérimenté". Pour cela la méthode est simple : il faut programmer, encore et encore. C'est à dire écrire des algorithmes ET des programmes. C'est souvent à force de traduire les algorithmes et de mettre au point les programmes qu'on apprécie le confort des algorithmes. Choisissez un sujet qui vous tient à coeur et essayer d'écrire une bibliothèque de fonctions et de modules pour ce sujet. Après quelques milliers de lignes (hors commentaires), beaucoup de choses seront plus claires.

En ce qui concerne la gestion physique des documents (fichiers, répertoires, etc.), vous découvrirez certainement qu'il faut là encore de l'organisation, une certaine systémisation des noms de fichiers, des noms de modules, et qu'il est parfois dur physiquement de retrouver où on avait mis l'algorithme qui... Vous serez alors mûr(e) pour passer à la programmation approfondie, c'est à dire aux objets et aux "browsers". Nous vous conseillons par exemple de jeter un coup d'oeil aux solutions mises en place par SmallTalk et par Perl pour gérer ces problèmes, ou par un environnement de développement visuel comme Deplhi ou Windev... et vous en concluerez que l'algorithmique se situe plus au niveau des idées que de l'implémentation, donc qu'il faut commencer par elle, et finir par les langages et les environnements de d"éveloppement.

N'hésitez pas à lire les algorithmes de autres, notamment ceux des livres cités en bibliographie. Et si vous trouvez qu'ils ne sont pas lisibles, malgré les explications du texte, c'est peut-être du à leur langage...

BIBLIOGRAPHIE

A. Aho, J. Hopcroft, J. Ullman
Structures de données et Algorithmes
InterEditions, 1987.

Cormen, Leiserson et Rivest
Introduction to algorithms
M.I.T. Press, 1990.

C. Froidevaux, M. C. Gaudel, M. Soria
Types de donnnées et Algorithmes
Ediscience, 1993.

B. Kernighan, R. Pike
The practice of programming
Addison-Wesley, 1999.

D. E. Knuth
The art of computer programming
Addison-Wesley, 1973.

P. Naudin, C. Quitté
Algorithmique algébrique
Masson, 1992.

J. Orwant, J. Hietaniemi, J. Macdonald
Mastering Algorithms with Perl
O'Reilly, 1999.

R. Sedgewick
Algorithms, second edition
Addison-Wesley, 1989.

S. Skiena
The algorithm design manual
Springer-Verlag, 1998.

H. Wilf
Algorithmes et complexité
Masson, 1989.

N. Wirth
Algorithmes et structures de données
Eyrolles, 1987.