collection **Informatique**

Algorithmique et programmation

Michael Griffiths

HERMES

http://dzdelphi.blogspot.com/

© Hermbs, Paris, 1992
Éditions Hermès
34, rue Eugène Flachat
75017 Paris

ISBN 2-86601-323-g

Table des matières

http://dzdelphi.blogspot.com/

http://dzdelphi.blogspot.com/

Tables et figures

http://dzdelphi.blogspot.com/

Les programmes

http://dzdelphi.blogspot.com/

Chapitre 1

Introduction

Depuis de nombreuses années, dans différents pays, les informaticiens ayant quelques prétentions académiques ont lutté pour établir leur discipline de manière indépendante. Sans vouloir dire que la lutte est terminée (certains n'ayant pas encore accepté que la terre n'est pas plate), on peut constater que, dans les universités respectées, il existe des laboratoires d'informatique independants, des diplômes spécialisés, et les enseignants et/ou chercheurs en informatique sont désormais considérés comme des scientifiques à part entière.

Pourquoi cette lutte ? Et pourquoi en parler dans un livre sur l'algorithmique ? Le fait est que les informaticiens ont représenté — et représentent toujours — un enjeu économique. Comme cet enjeu a été concrétisé par des moyens matériels et financiers mis à la disposition des enseignants et des chercheurs en informatique, tout un chacun a éprouvé le besoin de réclamer l'étiquette. Le tri entre les "vrais" et "faux" informaticiens n'est pas terminé. D'ailleurs, à notre avis il ne le sera jamais, et c'est peut-être heureux ainsi.

Malheureusement, en voulant affirmer leur indépendance par rapport aux autres disciplines, les informaticiens ont perdu une partie de l'essentiel. En se concentrant sur les techniques non-numériques (importantes et indispensables), ils ont perdu jusqu'à la notion de l'existence des nombres réels. De même, un peu en singeant les mathématiciens, qui ont montré la voie vers la tour d'ivoire, le besoin scientifique mais aussi psychologique de la construction dune (ou de plusieurs) théorie(s) a fait perdre la vraie justification de notre existence : l'écriture de programmes qui sont utiles. On est donc en présence de plusieurs guerres, numérique contre non-numérique, théorie contre application, utilisateurs contre spécialistes, vendeurs de vent contre professionnels sérieux. Si certaines guerres peuvent être utiles et salutaires, d'autres resteront toujours stériles.

http://dzdelphi.blogspot.com/

Ce livre ne saurait bien sûr en corriger les écarts. Mais nous voulons témoigner de notre foi dans l'existence de l'informatique en tant que discipline indépendante, mais surtout utile. L'informaticien comme le mathématicien — même si ce dernier l'a peut-être oublié — est un esclave des autres. Sa raison d'être est de rendre service, c'est-à-dire résoudre des problbmes d'application. Il n'y a pas d'informatique académique différente de celle de l'application numérique ou de gestion. Il n'y a pas une micro-informatique différente de l'informatique "classique". Il y a une seule discipline, appelée à intervenir dans un très grand nombre de domaines de l'activité humaine.

Dans cette optique, la formation des informaticiens dans les universités et les écoles d'ingénieurs doit se faire de manière équilibrée. Un de nos critères est que si nous ne traitions pas tel sujet, nous pourrions par la suite avoir honte de nos élèves ?". Le monde du travail s'attend à ce que nos élèves sachent programmer (dans le langage qu'utilise la compagnie concernée), qu'ils connaissent un minimum de méthodes de résolution de problèmes numériques, et que "probabilités" et "statistiques" ne soient pas des mots ésotériques à rechercher dans le dictionnaire. Le cours décrit dans ce livre essaie de prendre en compte ces considérations. Nous l'enseignons, dans des variantes appropriées, depuis vingt-cinq ans, à des élèves de premier et de deuxième cycle, en formation permanente, dans des diplômes spécialisés ou non. L'expérience a montré que l'enseignement de ce cours de base de l'informatique est difficile, que personne ne possbde une recette miracle, mais que tout le monde (surtout ceux qui n'ont jamais écrit un logiciel utilise par d'autres personnes) pense l'enseigner mieux que les autres.

Nous présentons donc ici, humblement (ce n'est pas notre tendance profonde), quelques idées d'un cours d'algorithmique et de programmation dont le niveau correspond à la première année d'un deuxibme cycle pour spécialistes en informatique, en supposant que les élèves concernés n'ont pas nécessairement subi une formation préalable dans la matière, mais qu'ils sachent tous un peu programmer Cela pose d'ailleurs un problème particulier. Nous avons l'habitude de mélanger des étudiants d'origines divers. Les uns peuvent avoir obtenu de bons résultats dans un IUT d'informatique, tandis que d'autres n'ont que peu touché un clavier (situation de plus en plus rare, mais l'écriture d'un programme de 10 lignes en BASIC peut être considérée comme une expérience vide, voire négative, en informatique). A la fin de la première année de deuxième cycle, il reste une corrélation entre les études préalables et les résultats académiques. Cette corrélation disparaît au cours de la deuxième année, provoquant quelques remontées spectaculaires au classement. La seule solution que nous avons trouvé à ce problème de non-homogénéité est de l'ignorer en ce qui concerne le cours, mais d'aménager des binômes "mixtes" en TP. L'expérience jusqu'alors est positive.

C'est en respectant l'idée que tous nos élèves savent un minimum sur la programmation qu'a disparu le chapitre de ce livre destinés aux débutants. Ainsi, les types de données de base (entiers, réels, caractères), leurs représentations en machine et les instructions simples (affectations, boucles, conditions) ne sont pas définis dans ce volume.

Dans sa forme actuelle, le cours dure une année scolaire, au rythme de deux heures par semaine. Il est nécessairement accompagné de travaux diriges et de travaux pratiques. Chez nous, il y a trois heures de travaux dirigés et quatre heures de travaux pratiques par semaine. De plus, la salle informatique est ouverte en libre service aux Ctudiants autant de temps que possible en respectant les règles élémentaires de sécurité. Ce n'est qu'au prix de la mise à disposition d'un matériel suffisant que les étudiants peuvent réellement apprendre.

Nous proposons de nombreux exercices et problbmes. Le sujet que nous attaquons nécessite un investissement personnel important. Le cours doit servir à stimuler des efforts individuels et la réalisation de projets de toutes sortes. Il doit obligatoirement se terminer par la création d'un logiciel en grandeur nature, de préférence produit par un petit groupe d'élèves (deux à quatre).

Les exemples dans ce livre sont rédigés dans un langage de programmation fictif qui ressemble à PASCAL. Comme boutade bien connue des quelques universités l'ayant subi, le langage dans lequel nous rédigeons nos algorithmes a pris le nom "GRIFFGOL", qui résulte de réflexions de l'époque de MEFIA [Cunin 1978]. C'est un style de programmation relativement indépendant d'un langage de programmation particulier, c'est-à-dire que nous utilisons les concepts fondamentaux dans la forme qui nous semble la plus appropriée. La construction des algorithmes se passe mieux quand on se permet une certaine liberté d'expression. Leur mise en œuvre dans un ordinateur nécessite une simple mise en forme en fonction du langage et du compilateur disponibles.

Un corollaire est que nous disons à nos étudiants qu'ils doivent toujours répondre "oui" à la question "connaissez-vous le langage X ?", quelle que soit la valeur de X. En effet, sous condition que le langage soit de style algorithmique classique, l'apprentissage d'un langage et/ou d'un compilateur inconnu ne devrait durer que quelques jours. D'ailleurs, un exercice classique que nous pratiquons est de faire recoder un travail pratique dans l'un ou l'autre des langages à grande diffusion que nos étudiants ne connaissent pas. C'est ainsi qu'ils absorbent, par exemple, FORTRAN. En pratique, à l'heure actuelle, ils programment dans une version de PASCAL, avec des prolongements en C. Le choix est remis en cause à chaque rentrée universitaire, car nous ne sommes pas des missionnaires d'un langage quelconque.

http://dzdelphi.blogspot.com/

1.1. Quelques mots sur l'environnement

Une petite phrase ci-dessus mérite qu'on s'y attarde un peu plus longtemps. Nous avons parlé de la disponibilité de matériels, essentielle à l'apprentissage de la programmation, qui est une activité constructive. On ne peut apprendre qu'en s'exerçant. Or pour s'exercer de manière satisfaisante, l'idéal, si nous pouvons nous permettre le parallèle, est que les ordinateurs doivent être comme les WC : il y en a toujours un de libre quand on en a besoin, voire même quand on en a envie. (Cette phrase a été prononcée pour la première fois, à notre connaissance, par P.M. Woodward, du Royal Radar Establishment à Malvern, Angleterre. C'est un plaisir de rendre cet amical hommage à un maître mal connu, en se rappelant qu'en 1964 il fallait une vision très large pour s'exprimer ainsi.).

Certains de nos collègues restreignent volontairement le côté expérimental de la programmation, dans le but d'imposer, dès le début de l'apprentissage, toute la rigueur nécessaire. C'est une réaction saine par rapport à une situation historique datant de l'époque où la programmation se faisait n'importe comment. Mais nous n'allons pas jusqu'à empêcher nos élèves de faire leurs bêtises. C'est en comparant des versions sauvages de programmes avec d'autres qui sont bien faites qu'ils comprennent réellement l'intérêt d'une méthodologie.

Ainsi, nous voulons que les élèves passent beaucoup de temps dans la salle des machines. Au début, ils travaillent mal, mais deviennent — pour la plupart — raisonnables à la fin de la première année. Cette approche profite d'une certaine fascination pour l'ordinateur (la jouissance de le dominer ?), qui s'estompe après cette première année. Le fait de vouloir réfléchir, plutôt que de se lancer immédiatement sur la machine, est un signe majeur de maturité chez l'élève. Cette étape ne peut être franchie que parce que le matériel est toujours disponible. Un étudiant ne doit pas être stressé par la longueur d'une file d'attente, ni frustré par des difficultés matérielles. Nous notons d'ailleurs que, bien que les programmes écrits en deuxième année soient assez importants, l'occupation des postes de travail diminue.

L'entraînement à la programmation est une nécessité pour tous les informaticiens, quelle que soit leur expérience. Une source de stimulation pour les élèves est de travailler en contact avec des enseignants qui programment bien. Tant qu'ils sentent qu'il leur reste du chemin à faire pour arriver au niveau de rendement de ce modèle, ils se piquent au jeu. Cela signifie que l'enseignant doit lui-même continuer à programmer régulièrement, comme le musicien qui continue à faire des gammes. Même si nous n'avons plus le temps de produire de gros logiciels, il faut s'astreindre à résoudre régulièrement des petits problèmes. Ceux-ci peuvent, par la suite, contribuer au renouvellement de notre stock d'exemples et d'exercices.

1.2. Notes bibliographiques sommaires

Il existe déjà, en français, un certain nombre de livres [Arsac 1980], [Arsac 19831, [Berlioux 19833, [Boussard 19831, [Courtin 1987a,b], [Gerbier 19771, [Grégoire 1986, 1988], [Lucas 1983a,b], [Meyer 19781, [Scholl 19841 dans le domaine que nous abordons ici. Un pourcentage élevé porte un nom d'auteur qui nous est familier, car il s'agit de collègues et souvent amis. Cela indique que les reflexions concernant ce sujet sortent, en France au moins, d'un nombre limite d'écoles qui, en plus, ont pris l'habitude de discuter. Le nombre de livres indique l'importance que chacun accorde au sujet, et les différences d'approche démontrent qu'il est loin d'être épuise. Comme dans la tradition littéraire, il y aura toujours des idées différentes sur ce sujet. En continuant le parallble avec l'écriture, nous recommandons aux étudiants de prendre connaissance de plusieurs styles différents, puis de développer leur propre style en profitant des apports de chacun.

Ayant indiqué quelques livres en français, il serait déraisonnable de laisser l'impression que la France possède un monopole des idées. Au niveau international, il existe une bibliographie conséquente en langue anglaise, dont voici les références qui correspondent à une sélection personnelle parmi les ouvrages les plus connus : [Aho 1974, 1983], [Dijkstra 1976], [Gries 1981], [Ledgard 1975], [Wirth 1976, 19771.

1.3. Remerciements

Sur un sujet tel que le nôtre, il serait impensable de citer tous ceux qui ont influencé notre reflexion. Nous nous limitons donc à la mention de quelques groupes de personnes, en nous excusant vis-à-vis de tous les collègues que nous ne citons pas individuellement.

Comme première influence directe, il y a eu le groupe de P.M. Woodwardà Malvem au début des années soixante. L'auteur y a fait ses premieres armes, et ses premiers cours, à partir de 1962, sous la direction de J.M. Foster et D.P. Jenkins, avec I.F. Currie, A.J. Fox et P.R. Wetherall comme compagnons de travail. Le foisonnement d'idées à Grenoble entre 1967 et 1975 a été d'une grande importance. Signalons seulement une collaboration directe avec P.Y. Cunin, P.C. Scholl et J. Voiron [Cunin 1978, 1980], bien que la liste aurait pu être nettement plus longue. L'école de C. Pair à Nancy a montre la rigueur et a donné des opportunités de comparaison de styles. Finalement, la création du diplôme d'ingénierie informatique à Marseille en 1985 a provoqué un effort de synthèse dont le résultat est ce volume. De nouvelles versions du polycopié ont vu le jour à Nantes en 1990 et 1991.

http://dzdelphi.blogspot.com/

Des rencontres ponctuelles ont aussi eu leur importance, en particulier à travers les écoles organisés par F.L.Bauer [Bauer 1974, 1975, 1976, 1979]. Celles-ci ont permis de travailler avec son équipe et de rencontrer et de se confronter avec E.W.Dijkstra, G.Goos, D.Gries, J.J.Horning, P.C.Poole et W.M.Waite, parmi de nombreux autres collègues.

Le feedback de générations d'étudiants nous a permis de constater que l'apprentissage de l'informatique n'est pas toujours aise si l'on veut atteindre un bon niveau. Nous remercions ces jeunes (certains ne le sont plus) pour leur apport, dont ils ne se sont pas toujours rendu compte (nous aussi, nous apprenons !). Emmanuel Gaston du DU-II et un groupe du mastère d'intelligence artificielle de Marseille, promotion 198889, ont corrige un certain nombre d'erreurs de français. Nathalie Wurbel a aidé en signalant des erreurs de français et de programmation. Christian Paul et Annie Tartier ont également apporté des corrections au niveau du français. Au cours d'un projet du DU-II, Vita Maria Giangrasso et Thierry Guzzi [Giangrasso 1989] ont mené à bien une étude comparative de certains algorithmes. Le journal *Jeux et Stratégie,* source de problèmes intéressants, nous a aimablement permis d'en utiliser dans certains exemples.

Ce livre a été préparé sur des micro-ordinateurs mis à notre disposition par différents organismes, dont feu le Centre mondial d'informatique et ressources humaines, le CNRS, l'université d'Aix-Marseille III, l'institut méditerranéen de technologie et l'université de Nantes, que nous remercions.

Un dernier mot sera pour mon collègue, et surtout ami, Jacek Gilewicz, professeur à l'université de Toulon. Au début, nous avions envie de préparer un livre combinant l'algorithmique numérique et non numérique. Pour différentes raisons, ce projet n'a pas vu le jour. L'utilisation du "nous" dans cette introduction représente ce pluriel. Je l'ai laissé dans cette version définitive, en espérant que Jacek rédigera le volume numérique dont nous avons besoin, en même temps que je lui témoigne ma reconnaissance pour son soutien et son amitié sans faille. Ce livre lui est dédié.

1.4. Le choix de programmes

On apprend à écrire des programmes en pratiquant. C'est pour cette raison que nous travaillons à partir d'exemples. Ceux-ci sont de plusieurs types :

- les classiques, qui se trouvent déjà dans d'autres ouvrages, mais qui sont essentiels à la culture d'un informaticien,
- les pédagogiques, que nous avons crées ou repris comme matériel de base. Ici, on aurait pu en choisir d'autres, mais chaque enseignant a sa liste d'exercices, souvent partagée avec des collègues,
- les amusements, qui sont la parce qu'ils nous ont fait plaisir, mais qui présentent néanmoins un intérêt pour l'étudiant.

Libre à chacun d'apprécier notre choix. De toute façon, il est entendu que chaque enseignant mettra la matière "à sa sauce".

Dans la mesure du possible (ou en fonction de nos connaissances), nous avons essayé d'attribuer la paternité des exemples, mais beaucoup d'entre eux ont des origines déjà inconnues. Toute information supplémentaire sera le bienvenue.

http://dzdelphi.blogspot.com/

Chapitre 2

Des programmes pour commencer

2.1. Le mode d'un vecteur

Ce problème nous est venu de D. Gries, qui l'utilise depuis longtemps dans ses cours d'introduction à l'informatique. Il a été repris par différents auteurs dans le cadre de l'analyse d'algorithmes [Arsac 1984], [Griffiths 19761.

On considère un vecteur, disons d'entiers, dont les éléments sont ordonnés. Son mode est la valeur de l'élément qui y figure le plus grand nombre de fois. Ainsi, prenons le vecteur suivant :

(1 3 3 6 7 7 7 11 13 13)

Le mode de ce vecteur est la valeur 7, qui figure trois fois. Nous allons écrire un progmmme qui prend en entrée un vecteur ordonné et qui livre le mode du vecteur en sortie.

2.1.1. *Construction de la première version du programme*

Commençons par une première idée d'algorithme. On note que l'ordonnancement du vecteur fait que les différentes occurrences d'un élément qui se répète sont contiguës. Il s'agit donc de trouver la plus longue chaîne d'éléments identiques. On va considérer les éléments à tour de rôle, en se rappelant de la plus longue chaîne vue jusqu'à présent. Pour chaque nouvel élément, on se posera la question de savoir si sa lecture mène à une chaîne qui est plus longue que la précédente. Dans ce cas, c'est la chaîne courante qui devient la'chaîne la plus longue.

http://dzdelphi.blogspot.com/

Pour garder les informations nécessaires, il faut disposer des valeurs suivantes :
- n est le nombre d'éléments dans le vecteur v,
- i est le nombre d'éléments qui ont déjà été considérés,
- lmax est la longueur de la plus longue chaîne en v[1..i],
- m est l'index en v d'un élément dans la chaîne de longueur lmax :

$$v[m] = mode(v[1..i]),$$

- lc est la longueur de la chaîne courante (à laquelle appartient v[i]).

On appellera cet ensemble de définitions *l'état du* monde ("state of the world").

Le programme 2.1, qui met en œuvre ces idées, est une application du schéma suivant :

```
Initialiser
TANTQUE NON fini
FAIRE avancer
FAIT
```

Le programme est commenté par la suite.

```
DEBUT DONNEES n: entier;
        v: TABLEAU [1 ..n] DE entier;
    VAR i, lmax, m, lc: entier;
    i:=1 ; lmax:=1 ; m:=1 ; lc:=1;
    TANTQUE i<n
    FAIRE i:=i+l ;
        SI v[i] = v[i-1]
        ALORS lc:=lc+1;
            SI lc>lmax
            ALORS lmax:=lc; m:=i
            FINSI
        SINON lc:= 1
        FINSI
    FAIT
FIN
```

Programme 2.1. *Le mode d'un vecteur*

Les données du problème sont n, la longueur du vecteur, et le vecteur ordonné, v. Les deux sont initialisés par ailleurs. Après les déclarations des objets de l'état du

monde (i, lmax, m, lc), la phase d'initialisation sert à leur donner des valeurs permettant de démarrer. Cette ligne correspond au traitement du premier élément : la chaîne maximum est la chaîne courante, de longueur 1. Dans le TANTQUE, comme i est le nombre d'éléments déjà traités, le test de la fin est bien i<n (autrement dit, il reste des éléments à considérer). Pour avancer, on prend le prochain élément (i:=i+1), en se demandant si l'examen de cet élément met en cause l'état du monde. Il y a deux cas : soit on allonge la chaîne courante (v[i] = v[i-1]), soit on commence une nouvelle chaîne. La nouvelle chaîne, décrite dans la partie SINON, est de longueur 1 (lc:=1) et ne met pas en cause lmax ou m (qui ne sont donc pas modifiés). Si la chaîne courante a été allongée (partie ALORS), on augmente lc (lc:=lc+1) avant de tester si la chaîne courante est devenue la chaîne maximale (SI lolmax). Dans ce cas, hnax et m sont mis à jour (hnax:=lc; m:=i).

Nous prétendons que cette construction démontre la justesse du programme donné. La démonstration dépend de l'acceptation de la récurrence (à partir du cas i-l on crée le cas i, le cas 1 étant traité dans l'initialisation) et de la validité de l'état du monde. En particulier, dans la boucle, avancer est fait par i:=i+l et les autres instructions remettent les éléments de l'état du monde à jour en fonction de ce que l'on y trouve.

Pour être complet, revenons sur la confirmation du nouvel état du monde. i a avancé de 1, ce qui est correct. lc a pris une valeur de 1 (nouvelle chaîne) ou de lc+1 (allongement de la chaîne en cours). La chaîne courante devient la chaîne la plus longue si lolmax. Dans ce cas, lmax et m reçoivent des valeurs appropriées. Comme tous les éléments de l'état du monde ont des valeurs correctes, le programme entier est juste.

2.1.2. Remarques méthodologiques

La construction d'un programme correct dépend de la logique employée par son créateur. Cette logique ne peut s'exercer que si les bases déclaratives sont saines. C'est ici qu'une bonne formalisation de l'état du monde est fondamentale. Cet état du monde peut être décrit en français (ou en anglais ou en polonais, nous ne sommes pas racistes) ou en style mathématique. Le choix entre les deux (ou les quatre) styles est une question de goût et de type d'application.

Mais, que l'état du monde soit écrit en français ou en formules mathématiques, il se doit d'être précis. De nombreux programmeurs agrémentent leurs programmes avec des commentaires du type "i est l'index de l'élément de v". Ils font souvent l'erreur classique de confondre le dernier élément traite avec son suivant (celui qui va être mité).

http://dzdelphi.blogspot.com/

L'utilisation de tableaux et de boucles impose d'autres contraintes. En particulier, il faut démontrer la terminaison de chaque boucle et confirmer que l'index de chaque référence à un élément de tableau est entre les bornes (i doit être contenu en [1..n]).

Pour la terminaison de la boucle, il n'y a pas de problème ici. La variable de contrôle i est augmentée de 1 à chaque tour de la boucle, et arrivera donc à n pour arrêter le massacre.

Les seules références à des éléments de tableaux sont dans le test SI v[i]=v[i-1]. i commence à un et la boucle s'arrête quand i=n. Avant l'affectation i:=i+1 à l'intérieur de la boucle, on a donc l'inégalité suivante :

$$1 \leq i < n$$

Apres l'augmentation de i, ceci devient :

$$1 < i \leq n$$

On voit facilement que i et i-l sont tous les deux à l'intérieur du domaine l..n.

Cette solution du problbme, d'une grande simplicité, est correcte, mais elle n'est pas la meilleure possible. Il existe une variante qui est moins longue, qui utilise moins de variables, et qui s'exécute plus vite. Il est même surprenant que l'amélioration échappe à la quasi-totalité des programmeurs. Nous y reviendrons dans un paragraphe ultérieur (cf. 7.1).

Par ailleurs, la solution donnée comporte une faiblesse. En général, il y a au moins un étudiant, ou une étudiante, dans la classe qui reste suffisamment réveillé(e) pour poser la question suivante :

"Qu'est ce qui se passe si deux chaînes sont de même longueur, c'est-à-dire si deux modes se présentent ?".

Bonne question ! La réponse peut être de prendre n'importe lequel des modes possibles, mais la faiblesse réside dans le fait que notre spécification du problème ne parlait pas de cette possibilité. On voit qu'il n'est pas simple de bien spécifier des algorithmes, surtout quand il faut être complet.

2.2. Recherche d'un objet

Dans ce nouveau programme, il s'agit de trouver l'emplacement qu'occupe un objet (disons un entier) dans un tableau. C'est une opération menée fréquemment dans des programmes de tous types. Supposons en premier lieu que nous ne savons

rien du tableau, c'est-à-dire qu'il faut en examiner tous les éléments pour démontrer l'absence eventuelle de l'objet recherche. Bien sûr, le programme s'arrêtera dès que l'objet est trouvé. Par la suite, nous montrerons comment rendre l'algorithme plus efficace en introduisant des connaissances concernant la forme des données.

2.2.1. Recherche linéaire

L'algorithme le plus simple examine à tour de rôle chaque élément jusqu'à la réussite de la recherche ou l'épuisement des candidats. Il nécessite l'état du monde suivant :

```
DONNEES
    n: entier, le nombre d'éléments de t,
    t: TABLEAU [1 ..n] DE entier, le tableau à examiner,
    objet: entier, l'objet à trouver,

VARIABLES
    i: entier, le nombre d'éléments déjà examinés,
    trouvé: bool, trouvé ≡ t[i]=objet.
```

Le programme 2.2 est une mise en œuvre possible.

```
DONNEES n, objet: entier;
    t: TABLEAU [1 ..n] DE entier;
PRECOND n>0
DEBUT VAR i: entier, trouvé: bool;
    i:=O; trouvé:=FAUX;
    TANTQUE i<n ET NON trouvé
    FAIRE i:=i+l ; trouvé := t[i]=objet
    FAIT
POSTCOND (trouvé ET objet=t[i]) OU (NON trouvé ET i=n)
FIN
```

Programme ***2.2. Recherche d'un objet dans un vecteur***

Le programme est encore une application du schéma suivant :

```
Initialisation
TANTQUE NON fini
FAIRE avancer
FAIT
```


http://dzdelphi.blogspot.com/

L'initialisation de l'état du monde dit que, le programme n'ayant rien vu (i:=0), l'objet n'a pas encore été trouvé (trouvé:=FAUX). Pour avancer, on augmente i de 1 et on remet trouvé à jour. Le processus peut terminer soit par la réussite (trouve devient VRAI), soit par l'inspection du dernier élément (i=n). Notons que les deux conditions peuvent être vérifiées en même temps, car l'objet peut se trouver en t[n]. La boucle termine, car i augmente à chaque tour. L'index de la seule référence à t[i] est juste, car l'initialisation et la condition de la boucle font que $0 \leq i < n$ à l'entrée dans la boucle. Avec l'augmentation de la valeur de i, ceci devient $0 < i \leq n$ au moment de la référence, c'est-à-dire $1 \leq i \leq n$.

Dans ce programme, nous avons introduit deux nouvelles notions : la pré-*condition* ("precondition") et la *post-condition* ("postcondition"). La pré-condition est ce qui doit être vrai pour que le programme soit exécutable. Ici, il faut qu'il y ait au moins un élément dans le tableau. Notons en fait que, si cette condition n'était pas vérifiée, par exemple n=O, le programme dirait que l'objet est absent, mais ceci est un hasard. La post-condition est ce que l'on sait à la fin du programme. Elle doit servir de spécification du résultat.

Prenons l'exemple d'une procédure de calcul d'une racine carrée. On aurait la structure suivante :

```
DONNEE x: réel;
PRECOND x≥0;
Procédure de calcul de y=sqrt(x)
POSTCOND y . y = x (en général, à epsilon près)
```

On sait calculer les racines carrées des nombres non négatifs (pré-condition) et le résultat est un nombre dont le carre est la donnée d'origine (post-condition).

Dans notre programme de recherche linéaire, la pré-condition est raisonnable, mais la post-condition n'est pas entièrement satisfaisante. Elle comporte deux lacunes : elle n'est pas directement déductible du texte et le sens de la variable trouve n'est pas bien défini, même si le programme est juste. Regardons maintenant le processus de déduction.

Reprenons le programme avec sa pré-condition, mais sans post-condition pour le moment :

```
PRECOND n>0;
DEBUT VAR i: entier, trouvé: bool;
    i:=0; trouvé:=FAUX;
    TANTQUE i<n ET NON trouvé
    FAIRE i:=i+l ; trouvé := objet=t[i]
    FAIT
FIN
```


Que sait-on sur ce programme ? Une première déduction, évidente, est que, comme le programme a quitte la boucle, la condition de continuation de la boucle est maintenant fausse. Ainsi, après FAIT, on peut déduire :

NON (i<n ET NON trouvé).

Par application de la loi de Morgan, ceci devient :

$i \geq n$ OU trouvé.

Comme i n'avance que d'un pas à la fois, on peut déduire :

$i = n$ OU trouvé.

Les OU de l'informatique étant inclusifs, les deux clauses peuvent être vraies en même temps (cas de t[n] = objet). Les post-conditions du programme donné suivent logiquement, en considérant l'affectation à la variable trouvé dans l'intérieur de la boucle.

En général, on peut souhaiter que la post-condition se retrouve à partir de la pré-condition, de déductions du type donné ci-dessus et des définitions de l'état du monde. Mais la post-condition que nous avons jusqu'ici est incomplète. Quand l'objet a été trouvé, i indique bien son emplacement, mais la non-réussite est mal décrite. Nous n'avons pas encore expliqué que NON trouvé veut dire que l'objet est absent.

L'erreur, classique, a été de construire les pré- et post-conditions aprbs coup. Il aurait fallu spécifier le programme avant de l'écrire, les conditions servant de spécification. Dans notre cas, on aurait :

```
PRECOND
  Soient un objet et un vecteur t de longueur n, n>0;

POSTCOND
  Ou t[i]=objet,
  ou il n'existe pas i, 0<i≤n, tel que t[i]=objet.
```

Notons que la variable trouvé ne figure plus dans la post-condition. Elle sert à distinguer les deux cas de solution. Il faut maintenant démontrer la vérité de cette nouvelle post-condition. Quand l'objet a été trouvé, la logique précédente est suffisante. Pour montrer l'absence de l'objet, il nous faut une clause de plus. Reprenons le texte avec de nouvelles décorations :

http://dzdelphi.blogspot.com/

```
DONNEES objet, n: entier; t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR i: entier, trouvé: bool;
  i:=0; trouvé:=FAUX;
  TANTQUE i<n ET NON trouvé
  FAIRE INVARIANT 0<j≤i => t[j] ≠ objet;
    i:= i+1; trouvé := t[i]=objet
  FAIT
POSTCOND t[i]=objet OU "i (0<i≤n => t[i]≠objet)
FIN
```

L'invariant dit que la boucle n'est exécutée que si l'objet n'a pas encore été trouvé. La post-condition peut maintenant être démontrée. En effet, à la terminaison de la boucle, si trouvé est FAUX, alors i=n ET t[i] n'est pas l'objet. Mais avant d'exécuter l'instruction i:=i+1, aucun t[j], 0<j≤i, n'était l'objet. Après l'augmentation de i, si t[i] n'est pas l'objet, l'invariant reste confirmé. Quand i=n, la deuxième partie de la post-condition est correcte.

L'utilisation d'assertions telles que les invariants de boucles permet d'arriver à des preuves formelles de programmes. En pratique, la preuve complète d'un programme de taille industrielle s'avère longue et coûteuse. Mais ce n'est pas une raison pour l'étudiant de ne pas connaître ces techniques. Une familiarité avec les bases des preuves de programmes est une des clés de l'amélioration de la performance d'un programmeur. Il crée de la sorte des schémas mentaux qui font qu'il analyse ses problèmes de manière plus rigoureuse, produisant des programmes de meilleure qualité en commettant moins d'erreurs.

2.2.2. Un piège

Les programmeurs de la génération précédente n'aimaient pas les variables booléennes. Ils préféraient écrire le programme ci-dessus dans une forme à priori plus simple :

```
i:=0;
TANTQUE i<n ET t[i+1 ]≠objet
FAIRE i:=i+1
FAIT
```

En supprimant la variable trouvé, le test de réussite se fait sur le prochain élément t[i+1]. Afin d'éviter le calcul de l'index, on peut redéfinir i comme l'index de l'élément à traiter et non plus l'élément qui vient d'être traité :


```
i:=1;
TANTQUE i≤n ET t[i]≠objet
FAIRE i:=i+1
FAIT
```

Malheureusement, ces deux programmes comportent une faiblesse qui ne se montrera qu'avec certains compilateurs. Supposons, dans la dernière version ci-dessus, que l'objet n'est pas présent dans le vecteur. La boucle tourne pour la dernière fois avec i=n. L'exécution de la boucle augmente la valeur de i, donnant i=n+l. On teste de nouveau avec TANTQUE. La première partie de la condition (i≤n) est fausse, mais on peut quand même en évaluer la deuxième partie afin de disposer des deux opérandes de l'opérateur ET. Cette deuxième partie comporte une référence à t[i], c'est-à-dire à t[n+1]. Or, cet élément n'existant pas, le programme peut terminer par l'erreur "index en dehors des bornes".

Cette faiblesse peut s'éliminer avec l'introduction de l'opérateur ETPUIS ("CAND"), qui ordonne les tests. Ainsi, on pourrait écrire :

```
TANTQUE i≤n ETPUIS t[i]≠objet
```

Quand i>n, c'est-à-dire que le premier opérande est FAUX, on n'évalue pas le deuxième. Cela dépend du fait que (FAUX ET b) est toujours FAUX, quelle que soit la valeur de b. Il existe également l'opérateur OUALORS ("COR"). Les définitions de ces opérateurs sont les suivantes :

```
a ETPUIS b ⌡ SI a ALORS b SINON FAUX FINSI
a OUALORS b ⌡ SI a ALORS VRAI SINON b FINSI
```

Notons que ces deux définitions sont celles qui se trouvent dans beaucoup de livres de logique pour ET et OU, mais ces opérateurs ne sont pas mis en œuvre de cette façon dans tous les compilateurs.

2.2.3. La dichotomie

Quand on ne sait rien sur les éléments d'un tableau, pour établir qu'une valeur donnée ne s'y trouve pas, il faut inspecter tous les éléments, car la valeur peut figurer à n'importe quelle place. Maintenant, nous allons considérer un cas plus intéressant, où les éléments du tableau sont ordonnés. C'est comme l'accès à un annuaire téléphonique. Pour trouver le numéro de M.Dupont, on ne regarde pas toutes les entrées de A à DUPONT, On procède par des approximations.

Pour nous faciliter la programmation, nous allons procéder par des approximations simples. Dans un annuaire de 1000 pages, on regarde la page 500.

http://dzdelphi.blogspot.com/

Si l'élément recherché est alphabétiquement plus petit, on a restreint la recherche aux pages 1 à 499, ou, s'il est plus grand, aux pages 501 à 1000. Chaque regard coupe le domaine de recherches en deux. Les recherches s'arrêtent si l'élément examine est le bon, ou si le domaine de recherches est devenu nul (l'objet est absent).

Considérons une première version de ce programme, avec l'état du monde suivant :

bas, haut: entier, SI t[i]=objet ALORS bas ≤ i≤ haut
centre: entier, t[centre] est l'élément à examiner
trouvé: booléen, trouvé ∫ t[centre]=objet

Appliquons comme d'habitude le schéma :

Initialisation
TANTQUE NON fini
FAIRE avancer
FAIT

Le programme 2.3 est une solution possible.

```
DONNEES n: entier, t: TABLEAU [1 ..n] DE entier;
PRECOND n>1 ET (0<i<j≤n => t[i]≤t[j])
DEBUT VAR bas, haut, centre: entier, trouvé: bool;
      bas:=1; haut:=n; trouvé:=FAUX;
      TANTQUE haut-bas>1 ET NON trouvé
      FAIRE centre:=entier((haut+bas)/2);
            CHOIX t[centre]<objet: bas:=centre,
                  t[centre]=objet: trouvé:=VRAI,
                  t[centre]>objet: haut:=centre
            FINCHOIX
      FAIT;
      SI NON trouvé
      ALORS SI t[bas]=objet
            ALORS centre:=bas; trouvé:=VRAI
            SINON SI t[haut]=objet
                  ALORS centre:=haut; trouvé:=VRAI
                  FINSI
            FINSI
      FINSI
FIN
POSTCOND trouve => t[centre]=objet,
   NON trouvé => " i, O&n, t[i]≠objet
```

Programme *2.3. Recherche par dichotomie*

La pré-condition exprime le fait que nous disposons d'un vecteur ordonné d'au moins deux éléments. L'initialisation indique que le domaine de recherches est t[1..n], l'objet n'étant pas encore trouvé. La condition après TANTQUE mérite des explications. La proposition NON trouvé est évidente, mais la partie avant le ET l'est moins. Pour considérer une valeur intermédiaire, nous imposons qu'elle soit différente des deux extrêmes, c'est-à-dire que l'inégalité suivante soit vraie :

Proposition A. bas < centre < haut

Cette inégalité nécessitant l'existence d'au moins une valeur entre bas et haut, on retrouve :

Proposition B. haut - bas > 1

Dans la boucle, le calcul de la valeur de centre nécessite une conversion, par la fonction entier, du résultat de la division, qui est un nombre réel, en un nombre entier. On confirme facilement que centre est bien entre haut et bas en appliquant la proposition B ci-dessus. Quand la somme (haut + bas) est impaire, la division par deux donne un nombre réelle de la forme n,5. Que l'arrondi vers un entier donne n ou n+1 n'a aucun effet sur l'algorithme (les deux possibilités respectent la proposition A). Par la suite, la clause CHOIX force un choix entre les trois possibilités ouvertes après comparaison de l'objet avec t[centre]. Ou l'objet a été trouvé (t[centre] = objet), ou le domaine de recherches est coupe en deux (bas:=centre ou haut:=centre, suivant la condition).

Si l'objet est trouvé, tout va bien. Si la boucle se termine sans trouver l'objet, haut et bas sont maintenant deux indices successifs :

haut = bas + 1

La dernière partie du programme teste si l'objet se trouve en t[haut] ou en t[bas]. Ce test est irritant pour le programmeur. Il n'est utile que si t[1]=objet ou t[n]=objet, car dès que haut ou bas change de valeur, ils prennent celle de centre, t[centre] n'étant pas l'objet. Donc le test ne sert que si l'une des variables haut ou bas a garde sa valeur initiale.

En fait, le test est dû à une faiblesse de spécification. Il faut décider si oui ou non t[haut] ou t[bas] peuvent contenir l'objet, et cela de manière permanente. Essayons deux nouvelles versions du programme. La première respecte les initialisations de l'original, ce qui veut dire que t[haut] ou t[bas] peut toujours contenir l'objet :

http://dzdelphi.blogspot.com/

```
DONNEES n: entier, t: TABLEAU [1 ..n] DE entier;
PRECOND n>1 ET 0<i<j≤n => t[i]≤t[j]
DEBUT VAR bas, haut, centre: entier, trouvé: bool;
    bas:=1 ; haut:=n; trouvé:=FAUX;
    TANTQUE haut≥bas ET NON trouvé
    FAIRE centre:=entier((haut+bas)/2);
        CHOIX t[centre]<objet: bas:= centre + 1,
              t[centre]=objet: trouvé:= VRAI,
              t[centre]>objet: haut:= centre - 1
        FINCHOIX
    FAIT
FIN
POSTCOND trouvé => t[centre]=objet,
        NON trouvé => " i, 0<i≤n, t[i]≠objet
```

Programme ***2.4. Dichotomie, version 2***

Trois changements figurent dans cette version du programme par rapport à l'original. Dans le choix, quand **t[centre]** n'est pas l'objet, la nouvelle valeur de bas (ou haut) ne doit pas se situer sur le centre, mais un pas plus loin, sur le premier candidat possible (centre est le dernier élément rejeté). En plus, dans le **TANTQUE,** au lieu de proposition B, nous avons simplement :

haut ≥ bas

Quand la proposition B est vraie, la situation n'a pas **changé** par rapport **à** la version originale. Quand **haut=bas,** centre prend cette même valeur et l'on teste le dernier candidat. Si ce n'est pas le bon, on ajuste haut ou bas, avec le résultat :

bas > haut

Quand haut suit immédiatement bas, centre va prendre une valeur qui est soit celle de haut, soit celle de bas (il n'y a pas d'espace entre les deux). Mais, grâce au fait que bas, ou haut, prend sa nouvelle valeur un cran plus loin que le **centre,** an prochain tour de la boucle, on aura :

bas = haut ou bas > haut

La boucle termine toujours. La preuve de la correction de ce programme peut se faire **à** partir des assertions suivantes :

A1. 0<i<bas => t[i]<objet
A2. haut<i≤n => t[i]>objet

La condition de terminaison bas > haut montre alors l'absence de l'objet dans **t.** Cela correspond **à** la **déduction** que l'on peut faire à la sortie de la boucle :

NON (haut ≥ bas ET NON trouvé)

C'est-à-dire :

bas > haut OU trouvé

bas > haut est la condition d'absence, trouvé implique **t[centre]** = objet.

La deuxième bonne solution à notre problème est de faire en sorte que ni **t[bas]** ni **t[haut]** ne peut être l'objet. Considérons la version du programme 2.5.

```
DONNEES n: entier, t: TABLEAU [1 ..n] DE entier;
PRECOND n>1 ET 0<i<j≤n => t[i]≤t[j]
DEBUT VAR bas, haut, centre: entier, trouvé: bool;
    bas:=0; haut:=n+l ; trouvé:=FAUX;
    TANTQUE haut-bas>1 ET NON trouvé
    FAIRE centre:=entier((haut+bas)/2);
        CHOIX t[centre]<objet: bas:=centre,
              t[centre]=objet: trouvé:=VRAI,
              t[centre]>objet: haut:=centre
        FINCHOIX
    FAIT
FIN
POSTCOND trouvé => t[centre]=objet,
        NON trouvé => " i (0<i≤n => t[i]≠objet)
```

Programme ***2.5. Dichotomie, version 3***

Dans cette version, seules les initialisations de bas et haut ont été modifiées. Ces variables prennent des valeurs d'indices inexistantes. Mais ce n'est pas grave, car les **éléments** correspondants ne seront jamais référencés. La preuve de cette version du programme est facile. Elle est la même que celle de la version originale, sauf pour **l'arrivée à** la situation :

haut = bas + 1

http://dzdelphi.blogspot.com/

Dans ce cas, comme ni t[haut], ni t[bas] n'est l'objet, celui-ci n'est pas dans t, car il n'y a pas d'élément entre t[haut] et t[bas].

La technique consistant à utiliser comme butée une valeur inexistante (ici en considérant [0..n+1] au lieu de [1..n]) est utilisée fréquemment par de bons programmeurs. Notons qu'il n'a pas été nécessaire de créer réellement les éléments fictifs introduits.

2.3. De la complexité des algorithmes

Le parallèle avec l'annuaire téléphonique montre que les performances de ces deux algorithmes (recherche linéaire et recherche dichotomique) sont très différentes. Leur complexité est facile à établir.

Pour la recherche linéaire :

- Si l'objet est présent dans le tableau, il peut être n'importe où. En moyenne, le programme examine n/2 éléments avant de trouver le bon.
- Si l'objet est absent, on examine tous les n éléments.

La complexité de l'algorithme est donc de o(n). Cela veut dire que si l'on doublait le nombre d'éléments, les recherches dureraient en moyenne deux fois plus longtemps.

La dichotomie est tout autre. Ici, un doublement de la taille du tableau nécessite un seul pas supplémentaire (chaque examen d'un élément divise la taille du domaine de recherche par deux). Considérons le cas d'un tableau de 2^k éléments. Apres un pas, on a 2^{k-1} éléments à considérer, après deux pas, 2^{k-2}, ... après k pas, un seul élément (2^0). La dichotomie a donc une complexité de $o(\log_2(n))$.

La différence entre ces deux courbes est très importante, surtout quand le nombre d'éléments est grand. La table 2.1 compare le nombre maximum de pas (n) dans le cas de recherche linéaire avec le nombre maximum de pas avec la dichotomie.

n	dichotomie	
10	4	($2^4 = 16$)
100	7	($2^7 = 128$)
1000	10	($2^{10} = 1024 = 1k$)
1 000 000	20	($2^{10} = 1024k = 1\,048\,576$)

Table 2.1. *Comparaison entre la recherche linéaire et la dichotomie*

Dans le jargon de l'informatique, on parle de la réduction d'un problème en n vers un problème en n-1 (recherche linéaire), ou vers un problème en n/2 (dichotomie). Par la suite, dans le chapitre 3 sur les tris, nous verrons que dans certains cas, on réduit un problème en n vers deux problbmes en n/2. On appelle cette dernière technique *l'art de diviser pour régner* ("divide and conquer").

Nous tirons deux conclusions de cette brève discussion. La première est que des connaissances minimales sur le calcul de la complexité des algorithmes sont nécessaires si l'on veut pratiquer de l'informatique à un bon niveau. Le sujet, assez difficile, est très étudié par des chercheurs. Ces recherches demandent surtout des compétences élevées en mathématiques. Mais on peut faire des estimations utiles avec un bagage limite. La deuxième conclusion concerne l'efficacité des programmes. On voit souvent des programmeurs s'échiner sur leurs programmes pour gagner une instruction ici ou là. Evidemment, dans certains cas précis, cela peut devenir nécessaire, mais c'est rare. Le gain d'efficacité est limité. Mais le problème n'est pas le même au niveau des algorithmes, comme l'attestent les chiffres de la table 2.1. Des gains d'efficacité à travers l'algorithmique peuvent être importants. Il n'est pas très sensé d'optimiser un mauvais algorithme — mieux vaut commencer avec un bon. L'optimisation locale peut se faire par la suite en cas de besoin.

2.4. Résumé des principes introduits

Au cours de ce chapitre, nous avons introduit plusieurs principes importants, qui forment la base de notre technique de programmation.

Nous travaillons souvent à partir d'un *schéma de programme* ("program scheme"). C'est' une maquette qui indique la structure générale. Le seul schéma utilise jusqu'ici est celui d'un processus linéaire :

```
Initialiser
TANTQUE NON fini
FAIRE  avancer
FAIT
```

On complète le schéma linéaire en utilisant un *état du monde,* qui est une définition précise des variables. Cet état permet de confirmer la justesse du programme en l'exploitant comme une liste à cocher en réponse à des questions du type suivant :

- Est ce que l'état du monde est complètement initialisé avant la boucle ?
- Quel est l'effet de l'opération avancer sur chaque élément de l'état du monde ?

L'existence d'une definition précise des variables facilite la définition de la

http://dzdelphi.blogspot.com/

condition de terminaison. De même, expliciter la notion d'avancer diminue la probabilité de l'écriture de boucles infinies. Néanmoins, pour éviter cette mésaventure, on démontre consciemment (et consciencieusement) la terminaison de chaque boucle. Le réflexe de démonstration de validité doit aussi jouer chaque fois que l'on repère un élément de tableau. On démontre que les indices sont nécessairement entre les bornes.

Nous avons utilisé les notions de ***pré-condition*** et ***post-condition.*** La pré-condition indique les limitations du programme, c'est-a-dire les caractéristiques des données en entrée. Elle sert pour des démonstrations de correction, mais aussi pour la documentation d'un programme. Avec la pré-condition, un utilisateur éventuel peut confirmer qu'il a le droit d'appeler le programme avec les données dont il dispose,

La post-condition indique ce que le monde extérieur sait après l'exécution du programme. On doit pouvoir remplacer tout programme par n'importe quel autre qui respecte les mêmes pré-condition et post-condition, sans que l'utilisateur éventuel s'en rende compte.

Une partie difficile du processus de la mise en œuvre est la *spécification* du programme. Mais le travail fait à ce niveau est payant. En effet, le coût dune erreur augmente avec le temps qu'elle reste présente. Mieux vaut passer un peu plus de temps en début du processus que de payer très cher, plus tard, l'élimination des erreurs.

Pour démontrer la correction d'un programme, on utilise des ***assertions*** et des ***invuriants.*** Les assertions sont des formules logiques qui sont vraies aux endroits où elles figurent dans le programme. Un invariant est une assertion qui est vraie à chaque tour d'une boucle. Une boucle est complètement définie par son état du monde et son invariant. Certains auteurs incluent l'état du monde dans l'invariant. Les deux techniques sont équivalentes.

2.4.1. *Un aparté sur les preuves de programmes*

Les techniques résumées ci-dessus reprennent des notions émanant des travaux sur la preuve de programmes. Le but est d'imprégner les cerveaux des étudiants de mécanismes mentaux allant dans cette direction, sans passer à une approche trop rigoureuse pour être soutenue dans la pratique. On doit savoir pourquoi le programme marche, sans avoir explicité tout le développement mathématique.

Pour le puriste, ou le mathématicien, cette approche n'est pas satisfaisante. Il serait normal — dans leur monde idéal que tout programme soit accompagné d'une preuve formelle. Ce sont les impératifs économiques qui font que le monde n'est pas idéal, surtout en acceptant les capacités et les motivations des programmeurs.

Le style présenté est donc un effort de compromis, maîtrisable par les étudiants dont nous disposons tout en les guidant. Au cours de leurs travaux dirigés, ils mènent à bien au moins une preuve formelle complète afin de comprendre les outils sous-jacents.

2.4.2. *Le style d'écriture*

Cet ouvrage s'adresse aux problèmes algorithmiques. Aucun des programmes présentés ne dépasse la taille d'un module dans un programme complet. La mise ensemble d'unités de programme pour la construction d'un produit industriel est un problème aborde ailleurs (cours de génie logiciel, projets).

Mais il ne faut pas perdre de vue cette question de modularité. L'utilisation de la clause DONNEES, avec les pré-conditions et post-conditions, vise, parmi d'autres buts, à préparer la définition de modules, avec leurs spécifications, interfaces et corps. En pratique, dans le cours enseigné, ces notions forment la matibre de discussions, préparant ainsi le travail en profondeur à venir.

2.5. Adressage dispersé

Les premiers programmes dans ce chapitre sont des exemples simples, introduits pour des raisons pédagogiques. Mais, en même temps, nous avons examiné des méthodes de recherche d'un élément dans un vecteur. Dans une première liste d'algorithmes de ce type, il faut introduire celui de l'adressage ***dispersé*** ("hash code"), qui est fréquemment utilisé dans la gestion, dans les compilateurs ou dans l'intelligence artificielle. La méthode a été inventée pour accélérer des recherches de positions jouées dans le jeu de dames [Samuels 1959].

Supposons que nous voulons créer un annuaire téléphonique au fur et à mesure de l'arrivée de numéros connus, sans faim de tri à chaque arrivée. C'est ce que l'on fait habituellement dans un carnet d'adresses. Dans un tel carnet, pour éviter d'avoir à examiner tous les noms de personnes, on les divise en 26 classes, en fonction de la première lettre du nom. Dans le carnet, on commence une nouvelle page pour chaque lettre. Les recherches vont plus vite parce que les comparaisons ne sont faites qu'avec les noms ayant la même première lettre. La première lettre sert ici de *clé* ("key"). Une clé est une fonction des caractères constituant le mot qui sert à diviser l'ensemble de mots possibles dans des classes. Si les noms Ctaient distribués de manière égale entre les classes, on divise le nombre de comparaisons par le nombre de classes. Pour un carnet, on ne regarde qu'un nom sur 26. Notons que ce rendement n'est pas atteint en pratique, parce que, par exemple, les pages K, X, Y, . . . contiennent moins de noms que certaines autres.

http://dzdelphi.blogspot.com/

L'adressage dispersé est une mise en œuvre du principe du carnet, avec quelques changements en raison des caractéristiques des ordinateurs. En particulier, un carnet comporte beaucoup de lignes vides sur les pages moins remplies. Nous décrivons deux versions de l'algorithme d'adressage dispersée, une première avec un nombre de clés plus petit que la taille de la mémoire disponible, et une deuxième où le nombre de clés est le même que le nombre de cases disponibles.

2.5.1. *Algorithme avec chaînage*

Dans cet algorithme, le nombre de clés est plus petit que le nombre de cases disponibles en mémoire. Considérons le carnet, où il y a 26 clés. On crée un tableau du type donné dans la figure 2.2.

Index	Nom	Suivant
1		-1
2		-1
3		-1
4	DUPONT	27
5		-1
6		-1
7		-1
8		-1
9		-1
10		-1
11		1
12		1
13		
14		-1
15		1
16		-1
17		1
18		
19		1
20	TOTO	28
21		1
22		1
23		-1
24	X	-1
25	Y	-1
26		-1
27	DURAND	29
28	TINTIN	-1
29	DUPOND	-1
...	...	...

Figure 2.2. *Table pour l'adressage dispersé avec chaînage*

Dans ce tableau, les 26 premières cases sont réservées pour le premier nom reçu de chaque classe (en supposant que la clé est la première lettre). La colonne de droite contient l'index de l'entrée contenant le prochain nom de même clé. Un successeur (suivant) d'index -1 indique que le nom dans cette entrée est le dernier rencontré dans sa classe (il n'a pas de successeur). Une case vide a également -1 comme successeur. Les noms comportent 8 caractères, étant complétés par des espaces. Une case vide comporte 8 espaces. La figure montre l'état de la table après la lecture des noms suivants :

X, DUPONT, TOTO, Y, DURAND, TINTIN, DUPOND.

A la réception d'un nom, on calcule sa clé i, ici la première lettre. Différents cas se présentent :

- Le nom est la premier occurrence d'un nom avec cette clé. Alors, la case d'index i est vide. Le nom s'insère à cette place et le processus est terminé.
- Un nom de clé i a déjà été rencontré. On compare le nouveau nom avec celui d'index i dans la table. Si les deux sont identiques, le nom est trouvé et le processus se termine.
- Si les deux noms sont différents, il faut examiner les successeurs éventuels comportant la même clé. Un successeur de valeur - 1 indique que la liste est terminée. On ajoute le nouveau nom à la première place disponible et le processus est terminé.
- Si le successeur existe, son index est donné dans la colonne correspondante. On compare le nouveau nom avec celui du successeur, en se ramenant au cas précédent.

Cela donne l'algorithme du programme 2.6, page ci-après, appelé à l'arrivée de chaque occurrence d'un nom.

A la fin de cet algorithme, la variable adresse contient l'index du nom dans la table. La variable pl indique la première case de débordement libre dans la table (avec la valeur de 27 au départ de l'algorithme). L'algorithme ne traite pas le problème d'un débordement éventuel du tableau.

http://dzdelphi.blogspot.com/

```
DONNEES clé: PROC(chaîne) -> entier;
      nom: chaîne(8);
      cars: TAB [1 ..taille] DE chaîne(8);
      succ: TAB [1 ..taille] DE entier;
DEBUT VAR i, adresse: entier;
            trouvé: bool;
            pl: entier INIT 27;
      i:=clé(nom);
      S I cars[i] = " "
      ALORS cars[i]:=nom;    % complété par des espaces %
            adresse:=i; trouvé:=VRAI
      SINON trouvé:=FAUX;
            TANTQUE NON trouvé
            FAIRE SI nom = cars[i]
                  ALORS adresse:=i; trouvé:=VRAI
                  SINON SI succ[i] = -1
                        ALORS cars[pl]:=nom;    % avec des espaces %
                              succ[i]:=pl; succ[pl]:= - 1; adresse:=pl;
                              trouvé:=VRAI; pl:=pl+1
                        SINON i:=succ[i]
                        FINSI
                  FINSI
            FAIT
      FINSI
FIN
```

Programme 2.6. ***Adressage dispersé***

2.5.2. *Autant de clés que de cases*

L'établissement d'un chaînage entre les différents noms d'une même clé gaspille de la mémoire. Pour éviter cette dépense, on peut choisir une fonction qui donne autant de clés que de cases dans le tableau. En restant avec notre clé (peu réaliste) de la première lettre, on peut refaire la table de la figure 2.2 pour obtenir celle de la figure 2.3.

Index	Nom
1	
2	
3	
4	DUPONT
5	DURAND
6	DUPOND
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	TOTO
21	TINTIN
22	
23	
24	X
25	Y
26	

Figure ***2.3. Table sans zone de débordement***

Cette figure reprend la situation de la figure 2.2. On note que les noms supplémentaires commençant par D ont pris les places des noms commençant par E et F. Que se passe-t-il alors si l'on rencontre le nom ESSAI ? On comparera ESSAI avec DURAND (case 5), puis avec DUPOND (case 6), avant de découvrir que la case 7 est vide. ESSAI rentrera dans la case 7. La clé sert à établir un point de départ des recherches, donnant ainsi l'algorithme du programme 2.7.

http://dzdelphi.blogspot.com/

```
DONNEES clé: PROC(chaîne) -> entier;
      nom: chaîne(8);
      tab: TAB [ 1 ..taille] DE chaîne(8);
DEBUT VAR i, adresse: entier;
          trouvé: bool;
      i:=clé(nom); trouvé:=FAUX;
      TANTQUE NON trouvé
      FAIRE SI tab[i] = "      "   % Case vide %
            ALORS trouvé:=VRAI; tab[i]:=nom
            SINON SI tab[i]=nom
                  ALORS trouvé:=VRAI
                  SINON SI i=taille  % Dernière case du tableau %
                        ALORS i:=1   % Recommencer en haut %
                        SINON i:=i+1
                        FINSI;
                        SI i=clé(nom) ALORS table pleine FINSI
                  FINSI
            FINSI
      FAIT
FIN
```

Programme *2.7. Adressage dispersé, version 2*

La table doit être circulaire (on recommence à regarder en haut si l'on arrive à la fin). Si la table est pleine, la recherche d'un nouveau nom fait tout le tour du tableau. Sauf si la table est pleine et le nom est absent, à la fin du programme, i contient l'index du nom dans le tableau.

2.5.3. *Choix de clé et efficacité*

Jusqu'alors, nous avons utilisé comme clé la première lettre du nom. Cette clé n'est pas la meilleure dans la plupart des cas. Le choix de la bonne clé dépend des caractéristiques des noms que l'on va rencontrer. Dans les compilateurs, on utilise souvent comme clé la somme des représentations internes des caractères (code ASCII ou EBCDIC) modulo la taille de la table. La taille choisie est habituellement une puissance de deux afin de calculer le modulus par décalage sur les bits. Ce choix est assez bon pour les identificateurs dans des programmes. Il permet d'utiliser le deuxième algorithme, sans chaînage.

Le deuxième algorithme est assez efficace tant que la table ne se remplit pas. Son rendement devient mauvais si la table est presque pleine. Le premier algorithme

garde ses qualités jusqu'au remplissage du tableau, au prix d'une occupation de mémoire supérieure.

Rappelons que les deux algorithmes nécessitent une fonction qui distribuent bien les noms parmi les clés. Si cette distribution est mauvaise, les deux algorithmes se rapprochent de l'algorithme de recherche linéaire.

Notons finalement que les algorithmes d'adressage dispersé ne diminuent pas la complexité théorique des recherches par rapport à celle des recherches linéaires. L'amélioration de cette méthode réside dans la diminution de la constante dans le formule. Toute la famille est d'ordre o(n) pour la recherche d'un nom.

2.6. Exercices

1. Avant de passer au chapitre qui donne la solution, chercher l'amélioration du programme de recherche d'un mode de vecteur. Ce nouveau programme est plus court et plus efficace que l'ancien. En particulier, il comporte une variable de moins et un test (SI ALORS . . .) de moins.

2. Considérons une nouvelle version du programme de dichotomie, écrit dans le style "sans booléens" :

```
DEBUT VAR bas, haut, centre: entier;
      bas:=1; haut:=n; centre:=entier((n+1)/2);
      TANTQUE haut>bas ET t[centre]≠objet
      FAIRE SI t[centre]<objet
            ALORS bas:=centre
            SINON haut:=centre
            FINSI;
            centre:=entier((haut+bas)/2)
      FAIT;
      SI t[centre]=objet
      ALORS . . .          % c'est trouvé %
      SINON . . .          % absent %
      FINSI
FIN
```

Cette version, qui est souvent proposée par des élèves, contient un piège. Lequel ? Comme mise sur la voie, on considérera le problème de la terminaison de la boucle.

http://dzdelphi.blogspot.com/

3. Mise en œuvre dans l'ordinateur des algorithmes d'adressage disperse. On prendra des textes de programmes afin de disposer de suites de noms (identificateurs). En prenant des programmes de grande taille, on peut mesurer l'efficacité des différents algorithmes de recherche d'objets (linéaire, dichotomie, adressage dispersé) en dressant des graphiques du temps d'exécution contre le nombre de noms lus. On examinera aussi l'influence du choix de la fonction de calcul des clés sur les algorithmes d'adressage dispersé.

Chapitre 3

Les tris

Ce n'est pas la première fois qu'un livre sur l'algorithmique et la programmation aborde le sujet des ***tris*** ("sorting"), loin de là. Mais le sujet est essentiel — on ne peut pas s'appeler informaticien sans avoir quelques connaissances des algorithmes de base. En plus, la matière est un excellent terrain d'entraînement. C'est donc sans honte que nous abordons ce chapitre, même si d'illustres prédécesseurs ont tracé la route. Parmi ceux-ci, accordons une mention particulière à [Knuth 1973], pour un volume tout entier consacré aux recherches et aux tris.

Trier un vecteur, c'est l'ordonner. On peut trier des entiers, des réels, des chaînes de caractères, . . . Il suffit qu'il existe une relation d'ordre entre chaque paire d'éléments, c'est-à-dire que, si a et b sont des éléments, une et une seule des relations suivantes est vraie :

a<b a = b a>b

Les axiomes habituels sont vérifiés :

a<b <=> b>a
(a<b ET b<c) => a<c
a=b <=> b=a

On peut également décréter que les éléments du vecteur sont tous différents, évitant ainsi de considérer l'égalité. Cette restriction n'apporte pas grand-chose pour la programmation.

http://dzdelphi.blogspot.com/

3.1. Recherche du plus petit élément

Un premier algorithme, que nous ne recommandons pas pour des applications pratiques, consiste en la recherche du plus petit élément restant. Ainsi, au premier tour, on recherche **t[min],** le plus petit élément en **t[1..n].** 11 devient **t[1]** dans le vecteur trié. Le plus simple est d'échanger **t[1]** avec **t[min],** ce dernier arrivant ainsi **à** sa place définitive. Le **problème** en n est réduit **à** un problème en n-l, car il reste **à** trier le vecteur **t[2..n].** Au deuxibme tour on recherche le plus petit élément en **t[2..n]** pour l'échanger avec **t[2],** et ainsi de suite. On aurait pu prendre l'élément le plus grand, en l'échangeant avec **t[n]** . . .

Le programme 3.1 montre un schéma d'algorithme appliqué **à** un vecteur d'entiers :

```
DONNEES n: entier;
    t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT MONDE i: entier, t[1 ..i] est trié;
    i:=0;
    TANTQUE i < n-l
    FAIRE i:=i+l ;
        trouver j tel que t[j] est le plus petit élément dans t[i..n]
        échanger(t[i], t[j])
    FAIT
FIN
POSTCOND t[1 ..n] est trié, càd "i,j (0<i<j≤n => t[i]≤t[j])
```

Programme 3.1. *Schéma du tri par recherche du plus petit élément*

La seule instruction nécessitant une explication est la condition **après TANTQUE.** 11 est nécessaire de trier n-l éléments, le dernier, **t[n],** étant nécessairement **à** sa place **à** la fin (tous les éléments qui le précèdent sont plus petits que lui par construction, sauf cas d'éléments égaux).

Reste **à** coder le contenu de la boucle dans ce texte. Ce sera une nouvelle boucle (programme 3.2).


```
MONDE k: entier, l'index du dernier élément considéré dans la
        recherche du plus petit;
    min: entier, t[min] est le plus petit élément en t[i..k];
k:=i; min:=i;
TANTQUE k<n
FAIRE k:=k+l ;
    SI t[k]<t[min]
    ALORS min:=k
    FINSI
FAIT
échange(t[i],t[min])
```

Programme 3.2. *Boucle interne*

Cette boucle ne nécessite pas d'explication. Notons néanmoins que le programme accepte l'égalité. Si le plus petit élément existe en plusieurs exemplaires, on prend le premier arrivé, le restant (ou les restants) étant considéré(s) au tour suivant. Le programme 3.3, complet, utilise les mondes décrits **précédemment.**

```
DONNEES n: entier;
    t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR i, k, min, temp: entier;
    i:=0;
    TANTQUE i<n-1
    FAIRE i:=i+l ; k:=i; min:=i;
        TANTQUE k<n
        FAIRE k:=k+l ;
            SI t[k]<t[min]
            ALORS min:=k
            FINSI
        FAIT;
        temp:=t[i]; t[i]:=t[min]; t[min]:=temp
    FAIT
FIN
POSTCOND t[l ..n] est trié
```

Programme *3.3. Programme complet*

http://dzdelphi.blogspot.com/

L'échange a été programmé de manière évidente, avec une variable temporaire. La complexité de cet algorithme est simple à calculer. Au premier tour, n-1 comparaisons sont nécessaires pour trouver le plus petit élément en t(1..n). Au deuxième tour il en faut n-2, puis n-3, . . . Le nombre de comparaisons est donc :

$$1 + 2 + 3 + \ldots + (n-1) = n*(n-1)/2$$

Comme les autres opérations (il y a n-1 échanges) se font moins souvent, l'algorithme est d'ordre $o(n^2)$. Le nombre de comparaisons est indépendant du contenu du vecteur.

3.2. Tri par insertion

Dans le programme du paragraphe précédent, après i tours, le vecteur t[1..i] est trié, mais en plus, ses i premiers éléments sont déjà à leur place, c'est-à-dire :

$$0<j<k\leq i \Rightarrow t[j]\leq t[k]$$
$$i<k\leq n \Rightarrow t[i]\leq t[k]$$

La première assertion dit que les i premiers éléments sont triés, la deuxième dit que les éléments après t[i] sont plus grands ou égaux à ceux déjà triés, t[i] étant le plus grand des éléments triés.

Une nouvelle version du tri ne nécessite pas cette deuxième condition. Après i tours, les i premiers éléments sont triés. On considère t[i+1]. Cet élément va être inséré à sa place en t[1..i+1], ce qui implique, en général, la recopie un cran plus loin de chaque élément de t[1..i] qui est plus grand que t[i+1] (voir le programme 3.4).


```
DONNEES n: entier;
    t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR i, j, temp: entier;
          arrêt: bool;
      MONDE i: t[1 ..i] est trié;
      i:=1;
      TANTQUE i<n
      FAIRE i:=i+1 ;
            MONDE j est la position du trou,
                  temp = t[i],
                  arrêt = t[j-1]≤temp;
            temp:=t[i]; j:=i; arrêt:= t[j-1]≤temp;
            TANTQUE j>1 ET NON arrêt
            FAIRE SI t[j-1]>temp
                  ALORS t[j]:=t[j-1]; j:=j-1
                  SINON arrêt:=VRAI
                  FINSI
            FAIT;
            t[j]:=temp
      FAIT
FIN
POSTCOND t[1 ..n] est trié
```

Programme *3.4. Tri par insertion*

Au tour i, ce programme fait "remonter" t[i] à sa place, ou plutôt il fait descendre les éléments de t[1..i-1] qui sont plus grands que t[i], afin de lui laisser sa place. On a donc l'idée du "trou". On enlève l'élément à considérer du vecteur, en le mettant dans la variable temporaire temp. Cela laisse un trou. On regarde le prédécesseur du trou. S'il est plus grand que temp, il descend, c'est-à-dire le trou monte. Si le predécesseur n'est pas plus grand, ou s'il n'y a pas de prédécesseur (l'élément considéré est le plus petit vu jusqu'ici), la recherche s'arrête, l'élément en temp trouvant sa place dans le trou.

Pour confirmer la validité des indices, notons que la boucle externe impose $i<n$. Apres $i:=i+1$, on a $i\leq n$. Dans la boucle interne, $j>1$ (condition) et $j\leq i$ (initialisation, avec j:=j-1 dans la boucle). On déduit :

$1<j\leq n$, donc t[j] et t[j-1] existent.

Les boucles se terminent, car i et j avancent, l'un vers n (i:=i+1), l'autre vers 1 (j:=j-1).

http://dzdelphi.blogspot.com/

La preuve de ce programme est faite en démontrant que le nouvel tlément retrouve bien sa place, ce qui revient à démontrer qu'à la fm de la boucle interne on a :

$$0<k<j \Rightarrow t[k] \leq temp$$
$$j<k \leq i \Rightarrow temp<t[k]$$

Comme les éléments Ctaient déjà ordonnés, et comme le nouveau est compatible avec l'ordonnancement par les deux assertions ci-dessus, l'invariant de la boucle externe (t[l..i] est trié) reste vrai.

On pourrait supprimer la variable booléenne arrêt en utilisant l'opérateur ETPUIS (programme 3.5).

```
DONNEES n: entier;
    t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR i, j, temp: entier;
    MONDE i: t[l ..i] est trié;
    i:=1;
    TANTQUE i<n
    FAIRE i:=i+l ;
        MONDE j est la position du trou,
        temp:=t[i]; j:=i;
        TANTQUE j>1 ETPUIS t[j-1]>temp
        FAIRE t[j]:=t[j-1]; j:=j-1
        FAIT;
        t[j]:=temp
    FAIT
FIN
POSTCOND t[ 1 ..n] est trié
```

Programme 3.5. ***Version avec ETPUIS***

On peut aussi faire la même chose avec une butée (voir exercice à la fin du chapitre). La complexité de cet algorithme dépend du nombre d'éléments qui "descendent" à chaque tour. Avec une distribution aléatoire, la place de l'élément considéré va être en moyenne au milieu des autres. Ainsi, le nombre d'éléments à déplacer est :

$$(1 + 2 + 3 + \ldots + n\text{-}l)/2 = n*(n\text{-}1)/4$$

On est toujours en ordre $o(n^2)$, mais cet algorithme est moins mauvais que le précédent (sans pour autant être recommande dans le cas général). En effet, si le tableau est déjà trié ou presque, le premier algorithme fait le même nombre de comparaisons que dans le cas d'une distribution aléatoire, tandis que dans le second, on constate n fois de suite qu'aucun mouvement n'est nécessaire, descendant ainsi à l'ordre o(n). Cet algorithme est donc bon pour des vecteurs que l'on sait déjà tries, ou presque.

3.3. Tri par bulles

Bien connu aussi, ce tri a une mauvaise réputation du point de vue de l'efficacid, réputation qui n'est pas tout à fait justifiée. A chaque parcours du vecteur, on compare successivement chaque paire de voisins. Si leur ordre n'est pas le bon, on les échange. Une des raisons de la mauvaise réputation de l'algorithme est que certains enseignants montrent la version du programme 3.6, qui est effectivement particulièrement inefficace.

```
DONNEES n: entier;
    t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR fin, i: entier;
    MONDE fin: t[l ..fin] reste à trier,
    fin:=n;
    TANTQUE fin>1
    FAIRE MONDE i: on va comparer t[i] et t[i+l];
        i:=l ;
        TANTQUE i<fin
        FAIRE ASSERTION 0<j<i => t[j]≤t[i];
            SI t[i+l]ct[i]
            ALORS échange(t[i],t[i+l])
            FINSI;
            i:=i+l
        FAIT;
        ASSERTION 0<j<fin => t[j]≤t[fin];
        fin:=fin-1
    FAIT
FIN
POSTCOND t[l ..n] est trié.
```

Programme ***3.6. Tri par bulles primitif***

http://dzdelphi.blogspot.com/

Cette version de l'algorithme est démontrable à partir des assertions données dans le texte. A la fin de chaque tour, l'élément "le plus lourd" est passe à la fin de la zone considérée. C'est une version moins efficace de l'algorithme du §3.1, en trouvant le plus grand élément au lieu du plus petit.

Il y a deux façons d'améliorer cette version :

- Considérons le vecteur suivant :

(23456789101)

A chaque passage dans le vecteur, le seul échange se fera entre la valeur 1 et son prédécesseur immédiat. Nous proposons de faire des passages alternativement de gauche à droite, puis de droite à gauche. Dans ce cas, le deuxième passage ramènerait le 1 au début du vecteur, qui serait donc trie après deux passages.

- Mais cela ne suffit pas, car l'algorithme ne se rend pas compte que le vecteur est trie. Considérons le vecteur suivant :

(21345678910)

On voit qu'il sera trié à la fin du premier passage. Ce qu'il faut ajouter au programme est la réalisation du fait que, étant donné qu'aucun échange n'a eu lieu, pendant ce premier passage, à partir de la valeur 3, t[2..10] est trié et les éléments sont déjà à leurs places définitives. La preuve de cette affirmation vient du fait que le test démontre que ces éléments sont ordonnés deux à deux. L'ordonnancement étant transitif (a≤b ET b≤c => tic), ils sont donc tous ordonnés. En plus, l'assertion montre que le dernier objet à prendre sa place (ici t[2]) est le plus grand de tous ceux vu jusqu'alors. Il en résulte que l'on peut réduire l'espace à trier plus rapidement.

Ces améliorations, qui prennent en compte le travail accompli "en route", c'est-à-dire les échanges intermédiaires, donnent lieu à l'algorithme du programme *3.7,*


```
DONNEES n: entier,
      t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR bas, haut, i: entier;
      MONDE bas, haut: t[bas..haut] reste à trier;
      INVARIANT t[l ..bas], t[haut..n] sont ordonnés,
            t[l ..bas-1], t[haut+l ..n] sont placés;
      bas:=1 ; haut:=n;
      TANTQUE bas<haut
      FAIRE MONDE i: t[i..haut] à comparer deux à deux,
                  der: dernier objet bougé;
            INVARIANT t[der..i] est ordonné;
            i:=bas;  der:=i;
            TANTQUE i<haut
            FAIRE SI t[i+1]<t[i]
                  ALORS échange(t[i],t[i+1]); der:=i
                  FINSI;
                  i:=i+l
            FAIT;
            MONDE i: t[bas..i] à comparer,
                  der: dernier objet bougé;
            INVARIANT t[i..der] est ordonné;
            haut:=der; i:=haut;
            TANTQUE bas<i
            FAIRE SI t[i]<t[i-1]
                  ALORS échange(t[i],t[i-1]); der:=i
                  FINSI;
                  i:=i-1
            FAIT;
            bas:=der
      FAIT
FIN
POSTCOND t[1..n] est trié
```

Programme 3.7. ***Tri par bulles normal***

Dans ce texte, ordonne veut dire que le vecteur indiqué est trie, c'est-à-dire que ses éléments sont dans l'ordre, mais qu'ils pourront changer de place dans le vecteur final en fonction d'autres éléments à insérer. Un vecteur place est non seulement ordonné, mais ces éléments occupent les places qu'ils auront dans le vecteur final. Ainsi :

http://dzdelphi.blogspot.com/

t[i..j] est ordonné <=> (i≤k<l≤j => t[k]≤t[l])
t[i..j] est placé <=> (t[i..j] est ordonné ET
(0<k<i, i≤l≤j => t[k]≤t[l]) ET
(i≤l≤j, j<m≤n => t[l]≤t[m]))

Les invariants permettent la démonstration directe de la justesse du programme.

On peut comparer la complexité de cet algorithme avec celle de l'algorithme par insertion. Les deux mènent à un même nombre d'échanges, que l'on calcule de la manière suivante : en prenant les éléments deux à deux, le nombre d'échanges est le nombre de fois où une paire d'éléments n'est pas ordonnée. Mais le nombre de comparaisons n'est pas le même pour les deux algorithmes. Malheureusement, ni l'un, ni l'autre n'est le meilleur dans tous les cas. Les deux algorithmes sont du même ordre de complexité, et ils partagent les propriétés d'être bons dans le cas d'un vecteur bien conditionné et d'être mauvais pour un vecteur mal conditionné.

3.4. Diviser pour régner

Les trois algorithmes de tri étudiés jusqu'ici sont tous d'une complexité d'ordre $o(n^2)$. Dans chaque cas, un passage du vecteur réduit un problème en n à un problème en (n-l). Comme pour la dichotomie par rapport à la recherche linéaire, il est possible de faire mieux en réduisant un problème en n à deux problèmes en n/2. Le terme généralement employé dans ces cas est celui de *diviser pour régner*. On divise le vecteur en deux moitiés, on trie chaque moiti4 et on remet les deux moitiés ensemble.

Il existe deux groupes d'algorithmes dans cette catégorie : les tris par partition et les tris par fusion. Le tri par partition permet de travailler "sur place", c'est-à-dire en gardant les éléments dans le vecteur en évitant de copier le vecteur entier dans un nouvel emplacement dans la mémoire. Le hi par fusion, utilisé du temps héroïque des programmes de gestion sur support de bandes magnétiques, nécessite la création d'un deuxième vecteur dans la mémoire (éventuellement secondaire), vecteur qui reçoit les éléments dans le nouvel ordre.

Pour des raisons évidentes, nous accordons plus d'importance, dans cet ouvrage, au tri par partition. Le tri par fusion est souvent pris comme exemple dans les cours de programmation avec le langage PROLOG.

3.4.1. *Diviser pour régner avec partition*

Dans le cas d'une distribution aléatoire de valeurs, surtout si n est grand, l'algorithme de diviser pour régner avec partition est le meilleur de ceux présent&

dans cet ouvrage. Le principe est de partitionner les éléments en deux classes en les comparant avec un élément dit *pivot*. Tous les éléments plus petits que le pivot vont se trouver à sa gauche, les autres se trouvant à sa droite. Considérons le programme 3.8, qui met en œuvre cette classification par rapport au pivot t[1].

```
DONNEES n: entier,
      t: TABLEAU [1 ..n] DE entier;
PRECOND n>0;
DEBUT VAR pivot, comp, bas, haut: entier;
      MONDE pivot: base de la comparaison,
            comp: valeur à comparer au pivot,
            bas: index du "trou" gauche,
            haut: index du "trou" droit;
      pivot:=t[l]; comp:=t[n]; bas:=l; haut:=n;
      TANTQUE bas<haut
      FAIRE SI comp<pivot
            ALORS t[bas]:=comp; bas:=bas+l ; comp:=t[bas]
            SINON t[haut]:=comp; haut:=haut-1 ; comp:=t[haut]
            FINSI
      FAIT;
      t[bas]:=pivot
FIN
POSTCOND 0<i<bas<j≤n => t[i]<t[bas]≤t[j], bas=haut, t[bas]=pivot.
```

Programme 3.8. *Partition d'un vecteur*

On crée deux trous dans le vecteur, en extrayant t[1], le pivot, et t[n], un élément de comparaison. Si l'élément de comparaison est plus petit que le pivot, il est placé dans le trou gauche, sinon dans le trou droit. Un nouveau trou est créé à côté de celui que l'on vient de remplir, en extrayant comme élément de comparaison le voisin du trou rempli. Le processus continue jusqu'à la rencontre des deux trous. On insère le pivot dans le trou (unique) qui résulte de cette rencontre, sachant que tous les éléments plus petits que le pivot sont à sa gauche, les autres étant à sa droite. Notons que cette formulation permet l'existence de valeurs Cgales, une valeur égale au pivot étant mise à sa droite.

La division du vecteur en deux parties a réduit le problème de hi en deux sous-problèmes. Il nous reste à trier les éléments à gauche du pivot, puis ceux à sa droite, le pivot étant à sa place (il ne bougera plus). L'algorithme est applique récursivement, c'est-à-dire que chaque moitié du tableau est de nouveau divisée en deux par comparaison avec un pivot à lui, et ainsi de suite. Après avoir divisé par deux un certain nombre de fois, il reste au plus un seul élément dans chaque classe (certaines sont vides), qui est automatiquement à sa place s'il existe.

http://dzdelphi.blogspot.com/

Pour mener à bien cette opération récursive, nous avons besoin de paramétrer le programme ci-dessus, qui va devenir une procédure dans le programme 3.9.

```
PROC tri(i, j);
GLOBAL n: entier, t: TABLEAU [1 ..n] DE entier;
SPEC i, j: entier;
     trier t[i..j] par la méthode de diviser pour régner;
PRECOND 0<i,j≤n;
DEBUT VAR pivot, comp, bas, haut: entier;
     SI j>i
     ALORS pivot:=t[i]; comp:=t[j]; bas:=i; haut:=j;
          TANTQUE bas<haut
          FAIRE SI comp<pivot
               ALORS t[bas]:=comp; bas:=bas+l; comp:=t[bas]
               SINON t[haut]:=comp; haut:=haut-1 ; comp:=t[haut]
               FINSI
          FAIT;
          t[bas]:=pivot; tri(i, bas-l); tri(bas+1, j)
     FINSI
FINPROC
```

Programme 3.9. *Insertion dans une procédure récursive*

Le terme GLOBAL indique que la variable n est déclarée à l'extérieur de la procédure (dans le programme englobant). Ainsi, la valeur de n est la même pour chaque appel de la procédure, tandis qu'il existe un nouvel exemplaire de i et de j pour chaque appel (chacun possède le sien). Le tableau t existe également en un seul exemplaire, manipule par chacun des appels de la procédure.

Cette procédure mène à bien le tri complet. Elle comporte le programme écrit précédemment, avec les changements nécessaires pour trier t[i..j] au lieu de t[1..n]. Une fois que la division en deux zones a eu lieu, il reste à les trier, l'une après l'autre, par deux nouveaux appels de tri avec des paramètres appropriés :

tri(i, bas-l) ET tti(bas+l , j)

La procédure teste par i<j qu'il y a au moins deux éléments à trier. Ce test confiie que les appels récursifs ne peuvent pas constituer un ensemble infini, car le nombre d'éléments à trier diminue avec chaque appel. Evidemment, on trie le vecteur complet par un appel de la procédure de la forme suivante :

tri(1, n)

La spécification dune procédure (SPEC) prend la place de la clause DONNEES d'un programme. On y trouve la définition du jeu de parambtres, avec une indication de ce que la procédure doit faire. Cette indication est la post-condition de la procédure.

L'algorithme a une complexité théorique d'ordre $o(n*\log_2(n))$, comme le montre le raisonnement suivant :

Le cas parfait de cet algorithme arrive quand $n=2^i-1$ et le pivot divise toujours la zone en deux parties de longueurs égales. A la fin du premier tour, on a un pivot place et deux zones de taille $(2^i-1-1)/2 = 2^{i-1}-1$. On voit que le nombre de comparaisons dans chaque zone est sa taille moins 1 (le pivot). En remultipliant par le nombre de zones, on déduit que le nombre total de comparaisons par tour est successivement :

n-l, n-3, n-7, n-15, . . .

Le nombre de tours est i, c'est-à-dire $\log_2(n+1)$. On obtient, pour des données bien conditionnées, la complexité théorique annoncée.

Ce calcul suppose que le pivot divise chaque fois son monde en deux parties de tailles comparables. Considérons maintenant un vecteur déjà ordonne. Comme nous prenons le premier élément pour pivot, un tour va réduire le problème en n dans un problème en 0 et un autre en n-l. Le processus est de nouveau d'ordre $o(n^2)$. Ce résultat, surprenant, montre que diviser pour régner, tout en étant le meilleur algorithme dans le cas d'une distribution aléatoire avec une grande valeur de n, est le plus mauvais pour un vecteur déjà trié (ou presque trié). On peut toujours prendre le pivot au centre du vecteur pour améliorer quelque peu l'algorithme dans ce cas.

3.42. Solution sans appel récursif

Le dernier programme ci-dessus comporte deux appels récursifs de la procédure. Si, pour une raison quelconque, on voulait enlever la récursivité, on utiliserait une *pile* ("stack" ou "LIFO • Last In, First Out"). Cette dernière sert à mémoriser, pendant le tri d'une zone, les limites des zones restant à trier. Ainsi, on trie t[i..j], avec, au début :

i=1 ET j=n

Apres un tour, avec p l'index de la position finale du pivot, on a la situation suivante :

http://dzdelphi.blogspot.com/

t[l ..p-1] est à trier
t[p] est placé
t[p+l ..n] est à trier.

Les bornes 1 et p-l sont mises sur la pile et le prochain tour redémarre avec :

i=p+1 ET j = n

Après chaque partition, une paire de bornes est mise sur la pile, l'autre étant traitée de suite. Après un certain nombre de partitions, la taille de la zone à traiter est 1 (ou 0), c'est-à-dire qu'elle est terminée. Dans ce cas, on recommence avec la première paire de bornes sur la pile, et ainsi de suite. Cette technique donne lieu au programme 3.10.

```
DONNEES n: entier,
    t: TABLEAU [ 1 ..n] DE entier,
PRECOND 0<i,j≤n;
DEBUT VAR i, j, pivot, comp, bas, haut, pl: entier,
    fini: bool,
    pile: TABLEAU [1 ..taille] DE entier;
    i:=1 ; j:=n; fini:=FAUX; pl:=1 ;
    MONDE fini: t[l ..n] est tri6
        pl: index de la première case libre dans la pile
        t[i..j] est à trier
        les zones indiquées dans la pile sont à trier
    TANTQUE NON fini
    FAIRE MONDE comme d'habitude
        TANTQUE j>i
        FAIRE pivot:=t[i]; comp:=t[j]; bas:=i; haut:=j;
            TANTQUE bas<haut
            FAIRE SI comp<pivot
                ALORS t[bas]:=comp; bas:=bas+1 ; comp:=t[bas]
                SINON t[haut]:=comp; haut:=haut-1; comp:=t[haut]
                FINSI
            FAIT;
            t[bas]:=pivot; pile[pl]:=i; pile[pl+1]:=bas-1 ; pl:=pl+2; i:=bas+1
        FAIT;
        fini:= pl=1;
        SI NON fini
        ALORS pl:=pl-2; i:=pile[pl]; j:=pile[pl+1]
        FINSI
    FAIT
FIN
```

Programme 3.10. ***Version avec pile***

On reviendra sur les piles au cours du chapitre 4, en les utilisant souvent par la suite.

3.4.3. Quelques commentaires sur la récursivité

Dans le programme de recherche linéaire, ou dans les tris primitifs, la base des algorithmes a été la réduction d'un problbme en n vers un problème en (n- 1). Cette réduction est triviale à mettre en œuvre; à chaque tour d'une boucle, on exécute l'affectation :

n:=n-1

et le tour est joué.

En passant à la recherche dichotomique, il s'agit de réduire un problème en n vers un problème en n/2. On cherche un objet dans le domaine d'indices [g..d]. Par rapport à un élément médiane d'index c on réduit le domaine soit à [g..c], soit à [c..d]. La mise en œuvre est également triviale, s'agissant de l'une des deux affectations suivantes :

d:=c OU g:=c

Notons néanmoins que ces utilisations de l'affectation n'existent que pour des raisons de mise en œuvre dans un ordinateur. Le raisonnement sous-jacent est de type récurrent. Par exemple, pour la recherche linéaire, on aurait pu écrire la fonction suivante :

```
chercher(objet, t[l ..n]):
    SI t[n] = objet
    ALORS trouvé
    SINON chercher(objet, t[l ..n-1])
    FINSI
```

Cette écriture fonctionnelle peut s'exprimer par l'utilisation d'une procédure récursive ou par une boucle avec affectation. Toute boucle peut se réécrire en forme de procédure récursive de manière directe.

Considérons maintenant le tri par diviser pour régner avec partition. Ici, on réduit un problbme en n vers deux problèmes en n/2. Une forme réduite de la fonction de tri est la suivante :

http://dzdelphi.blogspot.com/

```
tri(t[g..d]):
    SI d>g
    ALORS partition(t[g..d], c); tri(t[g..c-1]); tri(t[c+l ..d])
    FINSI
```

Une simple affectation à chaque tour d'une boucle ne suffit plus, car, si l'affectation relance le premier des appels de tri résultant, les paramètres du deuxibme appel doivent être stockés quelque part afin de revenir dessus à la fin du premier appel. Notons que chaque appel provoque, à son tour, deux nouveaux appels, jusqu'à l'arrivée de vecteurs de longueur 1 (ou 0).

On peut envisager la cascade d'appels dans la forme d'un arbre (figure 3.1).

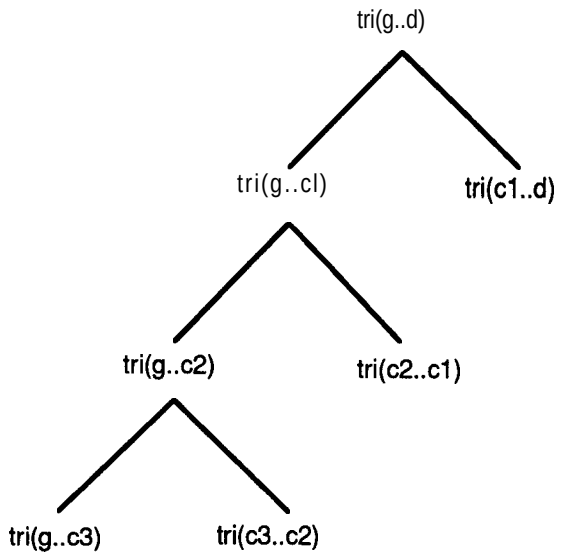

Figure 3.1. ***Appels aprèspartition***

Cette figure montre les appels en cours après trois partitions, les indices ayant été simplifiés pour alléger le dessin (suppression des +1 et -1). Après chaque partition, on reprend la branche gauche, en laissant la branche droite en suspens. Le successeur gauche est de nouveau partitionné, et ainsi de suite.

La modélisation par procédure récursive est donc une mise en œuvre directe et simple de l'algorithme. Par la suite, nous avons montré une mise en œuvre avec une pile. La pile a servi à se rappeler ce qui reste à faire à un moment donné. Elle

contient, à chaque instant, les paires d'indices correspondant aux appels laissés en suspens. Notons que le programme du paragraphe précédent trie la partie droite du vecteur, en mettant les indices de la partie gauche en suspens sur la pile. Dans la figure 3.1, l'ordre est inversé. En fait, l'ordre est indifférent; à la rigueur, on pourrait lancer les deux appels récursifs en parallèle sur deux processeurs indépendants dans le cadre d'un matériel multiprocesseur.

L'utilisation d'une pile est un moyen général pour mettre en œuvre la récursivité. Si la pile est programmée explicitement, comme dans le paragraphe précédent, on y stocke les informations permettant de savoir où l'on en est dans la cascade d'appels. On verra d'autres exemples dans le chapitre sur la marche arrière.

En fait, lors de l'utilisation d'une procédure récursive, un compilateur engendre des instructions qui gèrent une pile. Y sont gardées les valeurs des paramètres de l'appel en cours et des variables locales à chaque niveau d'appel. Ce sujet est couvert avec plus de détails dans des cours de compilation tels que [Cunin 1980].

On peut se demander quels sont les schémas récurrents qui permettent une traduction facile vers une boucle avec affectation directe de variables. Un problème qui se réduit de manière récurrente en deux sous-problèmes, ou plus, ne permet pas une telle traduction, car il faut toujours garder la trace des appels en suspens. Dans un schéma à un seul appel, la traduction vers une boucle est directe dans le cas d'une ***récursivité terminale.*** Dans ce cas, aucun calcul ne reste à exécuter après la fin de l'exécution de l'appel récursive, comme dans le cas de la recherche linéaire. Le problbme sera reconsidéré dans le chapitre 7, qui traite de la transformation de programmes.

3.4.4. Deux pivots

Comme la division d'un problbme en deux sous-problèmes — chacun de la moitié du coût du premier — représente un grand gain d'efficacité, on peut se demander si l'idée de diviser un problème en trois représente encore une amélioration. C'est purement au titre d'une spéculation intellectuelle que la question se pose pour le tri par diviser pour régner avec partition. L'auteur ne connaît pas de publication ni d'application de l'idée.

La mise en œuvre d'un tel algorithme (programme 3.11) nécessite donc deux pivots, avec un élément de comparaison. Avec les deux pivots, on partage les éléments en trois ensembles : ceux qui sont plus petits que le plus petit des pivots, ceux qui sont plus grands que le plus grand des pivots et ceux qui sont entre les deux. On trie par la suite les trois ensembles par trois appels récursifs.

http://dzdelphi.blogspot.com/

```
DEBUT DONNEES n: entier;
        t: TAB [1 ..n] DE entier;
      PROC tri(min, max: entier):
        VAR tg, td, pl, p2, t1, t2, t3: entier;
        SI max>min
        ALORS pl :=t[min]; t2:=min+1; p2:=t[t2]; tl :=min;
              test:=t[max]; t3:=max;
              TANTQUE t2<t3
              FAIRE SI test>t2
                   ALORS t[t3]:=test; t3:=t3-1; test:=t[t3]
                   SINON SI test<t1
                        ALORS t[tl]:=test; tl :=tl+l; t[t2]:=t[t1]
                        SINON t[t2]:=test
                        FINSI;
                        t2:=t2+1; test:=t[t2]
                   FINSI
              FAIT;
              t[t 1]:=p1; t[t2]:=p2;
              tri(min, tg-1); tri(tg+1, td-1); tri(td+l, max)
        FINSI
      FINPROC;
      tri(1, n)
FIN
```

Programme 3.11. *Diviser pour régner avec deux pivots*

La seule difficulté dans ce programme réside dans le maniement des trous et les ensembles pendant la partition. Il y a trois trous : deux pivots et un élément de comparaison. Les trois trous définissent quatre zones dans le vecteur : trois ensembles à trier et une quatribme zone qui contient les éléments en attente de partition. La figure 3.2 montre la distribution de trous et de zones pendant l'opération de partition.

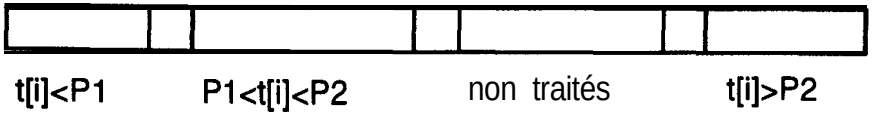

Figure *3.2. Vecteur en cours de partition*

Dans cette figure, P1 est la valeur du plus petit des deux pivots, P2 celle du plus grand. Les éléments les plus petits occupent la première zone dans le vecteur, les plus grands occupant la dernière (quatrième) zone. Les éléments ayant des valeurs intermédiaires occupent la deuxième zone, laissant la troisième pour les éléments n'ayant pas encore été considérés.

Introduire un élément dans la quatribme zone ne pose aucun problème; il pousse le trou droit un cran à gauche, le dernier élément non traité étant pris comme élément de comparaison. De même, on peut introduire un élément à la fin de la deuxième zone, repoussant le trou central un cran à droite et en considérant le premier des éléments non traités. Pour introduire un élément dans la première zone, il faut "mordre" sur la deuxibme. Le premier élément de la deuxième zone passe à la fin de cette zone, en poussant le trou central. Le premier trou peut ainsi avancer d'un cran, laissant de la place pour l'élément à insérer. On considère le premier élément non traité, éjecté par l'avancée du trou central.

Cette méthode apporte-t-elle des améliorations ? En théorie, oui; la complexité est d'ordre $o(n*\log_3(n))$ au lieu de $o(n*\log_2(n))$. En pratique, il faut des vecteurs de très grande taille pour s'en apercevoir.

3.4.5. Tri par fusion

Le tri par fusion date d'une époque où les mémoires centrales étaient petites par rapport aux fichiers à trier, ces derniers étant stockés sur des bandes magnétiques. Le principe est simple : on coupe le fichier en deux moitiés, on trie chaque moitié, puis on fusionne les deux moitiés triées en intercalant les valeurs de l'une et de l'autre dans le bon ordre. L'opération est répétée récursivement autant de fois que nécessaire. L'opération de fusion nécessite une deuxième copie du fichier, ce qui double l'espace mémoire occupé. En pratique, la fusion se faisait avec deux bandes magnétiques en entrée, chacune avec une moitié triée du vecteur, et une troisième bande en sortie pour recevoir le tout.

La récursivité ne va pas jusqu'au bout; on divise le vecteur par deux jusqu'à ce que le résultat de la division puisse tenir dans la mémoire principale. On peut dès lors trier cette zone du vecteur en mémoire avec un algorithme du type déjà vu. Tout l'art de la programmation de ce type de situation comportant des bandes magnétiques consistait à bien organiser les informations sur les bandes afin d'éviter de coûteux allers et retours pour rechercher le prochain bloc.

En ignorant l'existence des bandes magnétiques, l'algorithme peut travailler de la manière suivante :

- On divise le vecteur t1 autant de fois que nécessaire pour que les zones soient

http://dzdelphi.blogspot.com/

de longueur au plus 1.

- On fusionne des zones de longueur 1, par paires, pour créer des zones de longueur 2 dans une nouvelle copie t2 du vecteur.
- Les zones de longueur 2 sont fusiondes pour créer des zones de longueur 4, de nouveau en t1.
- On continue de la sorte jusqu'à la fin du processus.

Prenons comme exemple un vecteur de 8 éléments, trié à l'envers. Les états successifs de fusion sont les suivants :

t1:	(87654321)	Vecteur de départ
t2:	(78563412)	Fusion d'éléments créant des paires
t1:	(56781234)	Fusion par paires
t2:	(12343678)	Fusion par groupes de quatre

Le programme 3.12 montre une mise en œuvre correspondante.


```
DONNEES n: entier, t: TAB [1..n] DE entier;
DEBUTVAR l, pl, p2, p3, il, i2, i: entier;
    MONDE on fusionne des paires de zones de longueur l en zones de
          longueur 2*l;
       pl est l'index du premier élément de la première zone, p2 l'index
          du début de la seconde zone, p3 de la zone suivante (n+l s'il
          n'y en a pas);
       i éléments ont été recopiés vers le nouvel exemplaire du
          vecteur, il est l'index du premier élément non recopié de la
          première zone, i2 de la deuxième;
    l:=l;
    TANTQUE l<n
    FAIRE i:=0; pl :=1;
        ANTQUE i<n
        FAIRE p2:=p1+l; p3:=min(p2+l, n+1); il :=p1; i2:=p2;
            ANTQUE i<p3-1
            FAIRE i:=i+l;
                SI i1=p2
                ALORS t2[i]:=t1[i2]; i2:=i2+1
                SINON SI i2=p3 OUALORS t1[i1]<t1[i2]
                    ALORS t2[i]:=t1[i1]; il :=i1 +1
                    SINON t2[i]:=t1[i2]; i2:=i2+1
                    FINSI
                FINSI
            FAIT;
            pl :=p3
        FAIT;
        l:=2*l;
        SI l<n
        ALORS pl :=1; i:=0;
            TANTQUE i<n
            FAIRE p2:=l+1; p3:=min(p2+l, n+1); il :=pl ; i2:=p2;
                TANTQUE i<p3-1
                FAIRE i:=i+1;
                    SI i1=p2
                    ALORS t1[i]:=t[i2]; i2:=i2+1
                    SINON SI i2=p3 OUALORS t2[i1]<t2[i2]
                        ALORS t1[i]:=t2[i1]; il:=il+l
                        SINON t1[i]:=t2[i2]; i2:=i2+1
                        FINSI
                    FINSI
                FAIT;
                p1:=p3
            FINSI;
        l:=2*l
    FAIT
FIN
```

Programme 3.12. ***Triparfusion***

http://dzdelphi.blogspot.com/

3.5. Résumé de la complexité des algorithmes

En fonction des discussions précédentes, on peut dégager des raisons menant à un choix sensé d'algorithme de tri pour une situation donnée. On aboutit aux conclusions suivantes :

- La méthode de recherche du plus petit élément n'est jamais bonne. On ne l'utilisera pas.

- Pour de grands vecteurs dont les éléments sont distribués aléatoirement, la méthode de diviser pour régner avec partition est la meilleure. Elle est mauvaise dans le cas d'un vecteur déjà trié, ou presque trié. Pour limiter les dégâts dans ce cas, il vaut mieux prendre le pivot au milieu de la zone, ce qui complique légèrement le programme (voir exercice).

- Pour des vecteurs presque triés, les deux méthodes d'insertion ou par bulles sont de bonnes candidates, à condition d'utiliser la version optimisée pour la deuxième. L'une ou l'autre peut être la meilleure, en fonction des propriétés particulières de l'ordonnancement approximatif déjà existant.

3.6. Exercices

1. Dans le tri par insertion, l'utilisation de ETPUIS permet de supprimer la variable arrêt. Dans un langage sans ETPUIS, comment arriver au même résultat par l'introduction d'une butée ?

2. Donner une règle permettant de calculer le nombre de comparaisons nécessaires danslecas:
   - d'un tri par insertion,
   - d'un tri par bulles.

3. Dans la dernière version du tri par diviser pour régner, on fera la modification consistant à prendre comme pivot l'élément au milieu de la zone à trier.

4. Un bon exercice au niveau dune classe est de comparer les différents tris, mis en machine par différents élèves. On mesurera, par l'horloge et en comptant les opérations, le coût de l'exécution de chaque tri, en essayant une série de vecteurs différents. On variera la distribution d'éléments (triés, presque triés, aléatoire, triés à l'envers, . ..) et la longueur du vecteur. L'établissement de courbes pour chaque méthode permet d'en confirmer la complexité théorique.

Chapitre 4

Des structures de données

Les langages de programmation classiques permettent l'utilisation de variables simples (entiers, réels, . . .) et de tableaux, voire, éventuellement d'autres types de données composées (records, . ..). Dans le dernier exemple du chapitre 3 (tri par diviser pour régner), nous avons eu besoin d'une structure de données particulière, la pile, qui n'existe pas directement dans ces langages. En effet, dans la plupart des langages de programmation, on ne dispose pas d'un type "pile".

Un certain nombre de structures de données de ce type reviennent constamment dans les programmes. Dans ce chapitre, nous allons décrire les principales structures, avec les moyens de les représenter dans des langages existants. Avec le développement de nouveaux langages, on peut s'attendre à voir ces objets devenir des types standards. En premier lieu, nous allons considérer les ***piles,*** les ***files*** ("queue" ou "FIFO - First In, First Out"), les ***arbres*** ("tree"), les ***treillis*** ("lattice") et les graphes ("graph").

4.1. Les piles

Une pile est un ensemble ordonné d'objets de même type (entiers, réels, . ..). C'est comme si l'on gardait une pile de livres sur son bureau. On peut poser un nouveau livre sur la pile, ou reprendre le livre qui est en haut de la pile. Extraire un livre du milieu est tellement difficile que nous renonçons à cet exercice.

On dispose d'un vecteur dont les éléments sont du type approprié. Il existe deux opérations fondamentales : poser un objet ***(empiler,*** ou "push") et en retirer un ***(dépiler,*** ou "pull"). Le vecteur sert à stocker les objets posés. Pour savoir combien d'objets sont dans la pile à un moment donné, on utilise un pointeur de niveau. Le pointeur sert aussi à retrouver, dans le vecteur, le dernier objet déposé.

http://dzdelphi.blogspot.com/

Le programme 4.1 utilise l'index de la première case libre (pl) dans une pile d'entiers.

```
DONNEES taille: entier,
    pile: TABLEAU [1 ..taille] DE entier;
VAR pl: entier INIT 1;
PROCEDURE empiler(x: entier);
PRECOND pl≤taille;
    pile[pl]:=x; pl:=pl+1
FINPROC;
PROCEDURE dépiler(x:entier);
PRECONQ pl>1;
    pl:=pl-1; x:=pile[pl]
FINPROC
```

Programme 4.1. *Mise en œuvre d'une pile*

On voit que le nombre d'objets dans la pile à un moment donné est pl-1. Les procédures supposent que la pile ne déborde pas (PRECOND plltaille) et que l'appel de dépiler n'a pas lieu avec une pile vide (PRECOND pl>1). En pratique, on teste ces conditions par des SI ALORS spécifiques en début de procédure.

Dans le chapitre précédent, nous avons programmé directement la pile sans utiliser les procédures empiler et dépiler. La taille de la pile était supposée suffisante. Une pile vide était le signe de terminaison de l'algorithme.

La pile donnée ici est une pile d'entiers. Le même schéma de programme peut servir pour la construction d'une pile de réels, de booléens, . . . (il suffit de changer le type des éléments du tableau pile).

4.2. Les files

Une file est une structure qui ressemble à une pile, à la différence près que l'on retire l'élément le plus ancien au lieu du plus récent. Une telle structure est particulièrement utile, par exemple, dans un système d'exploitation, pour la gestion des files d'attente. On pourrait imaginer la paire de procédures du programme 4.2.

PROGRAMME DISCUTABLE

```
DONNEES taille: entier; file: TABLEAU [1 ..taille] DE entier;
VAR ancien: entier INIT 1; libre: entier INIT 1;
PROCEDURE mettre(x: entier);
PRECOND libre≤taille;
    file[libre]:=x; libre:=libre+1
FINPROC;
PROCEDURE enlever(x: entier);
PRECOND ancien<libre;
    x:=file[ancien]; ancien:=ancien+l
FINPROC
```

Programme 4.2. *Une file discutable*

Pour mettre un objet dans la file, il faut qu'il y ait une place libre (PRECOND libreltaille). Pour en enlever, il faut qu'il y ait au moins un objet dans la file (PRECOND ancienclibre). Les deux indices ancien et libre "cernent" les objets restants dans la file, qui se trouvent en file[ancien..libre-1].

Pourquoi avoir dit que le programme ci-dessus est discutable ? Tout simplement parce qu'il ne réutilise pas l'espace dans la file. Une fois arrivé au bout (libre=taille+1), on ne peut plus y mettre de nouveaux éléments, même si le retrait d'autres éléments a libéré de la place. Il faut rendre la file circulaire, en testant autrement la présence d'au moins un élément (programme 4.3).

```
DONNEES taille: entier, file: TABLEAU [Maille] DE entier;
VAR n: entier INIT 0; ancien: entier INIT 1; libre: entier INIT 1;
PROCEDURE mettre(x: entier);
PRECOND n<taille;
    file[libre]:=x;
    SI libre=taille
    ALORS libre:=1
    SINON libre:=libre+l
    FINS1
FINPROC;
PROCEDURE enlever(x: entier);
PRECOND n>0;
    x:=file[ancien];
    SI ancien=taille
    ALORS ancien:=1
    SINON ancien:=ancien+l
    FINS1
FINPROC
```

Programme 4.3. *Une file circulaire*

http://dzdelphi.blogspot.com/

La nouvelle variable n indique le nombre d'éléments actuellement présents dans la file. Les pré-conditions de non-plénitude et de non-vide dépendent donc de la valeur de n. Les variables libre et ancien avancent chaque fois de 1, en recommençant au début de la file une fois arrivées à la fin. Dans un système d'exploitation, on parle souvent d'un producteur ("producer") et d'un consommateur ("consumer") à la place de mettre et enlever (voir [Griffiths 19881).

4.3. Les arbres

L'arbre est une structure fondamentale dans l'algorithmique. Nous en avons déjà vu un dans l'algorithme de tri par diviser pour régner. Le pivot sert à diviser le vecteur en trois zones :

- une partie gauche,
- le pivot,
- une partie droite.

Les parties gauche et droite sont de nouveau divisées, chacune en trois nouvelles zones, et ainsi de suite. La figure 4.1 présente ces divisions en forme d'arbre.

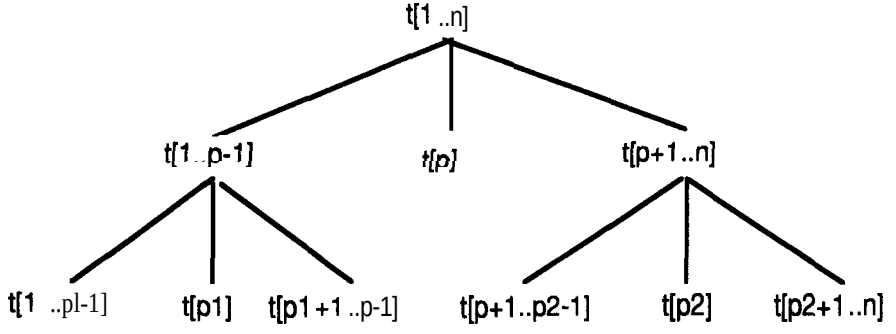

Figure *4.1. Arbre du tri diviserpour régner*

Notons que les informaticiens ont pris l'habitude d'inverser les arbres : la situation initiale, la *racine* ("root"), est en haut, et les éléments terminaux, les *feuilles* ("leaf, leaves"), en bas. Des arbres australiens, en quelque sorte . . . On parle donc d'algorithmes qui descendent de la racine jusqu'aux feuilles.

Formellement, un arbre est une structure composée de *nœuds* ("node") et de *branches* (arc, "branch"). Une branche menant du nœud nl au nœud n2 fait que n2 est un *successeur* ("successor") de nl, nl étant le *prédécesseur* ("predecessor") de n2. Chaque nœud, à l'exception de la racine, possède un et un seul prédécesseur. Un nœud peut avoir un nombre quelconque de successeurs. Un nœud sans successeur est une feuille.

Suivre un chemin dans l'arbre se fait en suivant des branches successives. Ainsi, si n2 est un successeur de nl et n3 est un successeur de n2, alors il existe un chemin de nl à n3 (par l'intermédiaire de n2). Nous limitons nos arbres à des arbres connexes, c'est-Mire que tout nœud n est *accessible* à partir d'une unique racine. Accessible veut dire qu'il existe un chemin de la racine de l'arbre jusqu'au nœud n. Notons que l'unicité des prédécesseurs fait que ce chemin est unique.

4.3.1. *Arbres binaires et arbres n-aires*

La définition donnée ci-dessus permet à un nœud d'un arbre d'avoir un nombre quelconque de successeurs. En pratique, les programmeurs se limitent souvent à l'utilisation d'arbres binaires, dans lesquels un nœud a au plus deux successeurs. A priori, cela pourrait constituer une limitation du pouvoir d'expression, mais en fait ce n'est pas le cas. On démontre que-tout arbre n-aire peut être représenté sous la forme d'un arbre binaire, sans perte d'information.

La forme binaire d'un arbre n-aire s'appelle un *diagramme en forme de vigne* ("vine diagram"). Au lieu de garder, pour chaque nœud, des pointeurs vers ses successeurs immédiats, on garde un pointeur vers son premier successeur (fils aîné) et un deuxième vers le prochain successeur de son prédécesseur (frère cadet). La figure 4.2 en donne un exemple.

On voit sur la figure 4.2 les couples de pointeurs représentés dans des boîtes à deux cases. Une case barrée indique l'absence de successeur. Cette façon de décrire un arbre correspond d'assez près à sa représentation physique dans la mémoire de l'ordinateur.

http://dzdelphi.blogspot.com/

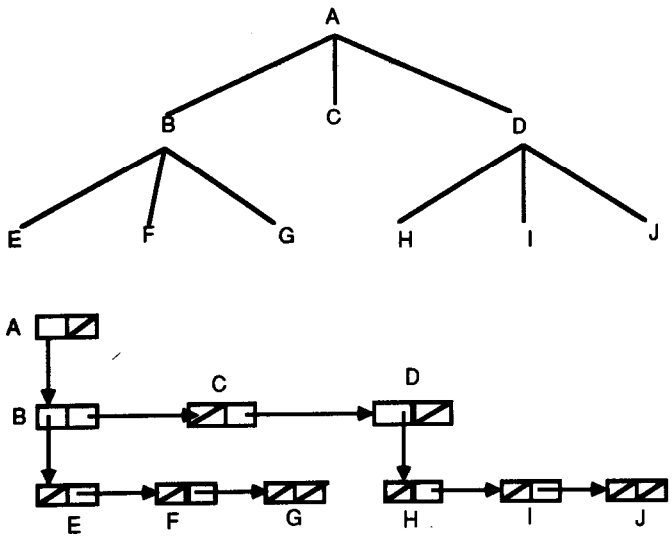

Figure *4.2. Transformation d'arbre n-aire en arbre binaire*

4.3.2. Représentation d'arbres

Pour illustrer la représentation physique des arbres binaires, reprenons les boîtes de la figure 4.2, en les mettant dans une table (en pratique on utilise deux vecteurs). Les noms des **nœuds** deviennent des entiers qui servent d'index dans ses vecteurs. En reprenant l'arbre de la figure 4.2, avec a=l, **b=2,** et ainsi de suite, on obtient les vecteurs de la figure 4.3.

nom	index	succg	succd
A	1	2	0
B	2	5	3
c	3	0	4
D	4	8	0
E	5	0	6
F	6	0	7
G	7	0	0
H	8	0	9
1	9	0	10
J	10	0	0

Figure *4.3. Représentation de l'arbre de la figure 4.2*

Les deux premières colonnes de cette figure ne sont données que pour faciliter la lecture. Dans la réalité, seuls sont conservés les vecteurs succg (successeur gauche) et succd **(successeur droit).**

Etant donné la possibilité de transformer un arbre n-aire en arbre binaire, cette **représentation** est toujours suffisante. La structure avec deux boîtes par nœud date de LISP **[McCarthy** 19601. Si l'on voulait conserver la forme originale, avec n successeurs possibles, la table de la figure 4.3 comporterait autant de colonnes que le nombre maximum de successeurs.

4.3.3. Parcours d'arbres

L'algorithme classique de parcours d'un arbre utilise un double appel récursif (programme 4.4).

```
DONNEES succg, succd: TABLEAU [1..taille] DE entier;
PROC parcours(n: entier);
    SI succg(n)≠0
    ALORS parcours(succg[n])
    FINSI;
    SI succd(n)≠0
    ALORS parcours(succd[n])
    FINSI
FINPROC
```

Programme *4.4. Parcours d'un arbre binaire*

Le parcours complet d'un arbre commence par un appel de

parcours(racine)

Le parcours d'un **nœud** implique le parcours de tous ses successeurs. Les tests sur les successeurs établissent leur existence.

Pour éviter de tester successivement l'existence **d'un successeur gauche,** puis celle d'un **successeur** droit, on peut avoir recours à la notion de butée. Un successeur inexistant a l'index 0. Il suffit de créer un **nœud** d'index 0 comme butée (programme 4.5).

http://dzdelphi.blogspot.com/

```
DONNEES succg, succd: TABLEAU [0..taille] DE entier;
PROC parcours(n: entier);
    SI n≠0
    ALORS parcours(succg[n]);
          parcours(succd[n])
  FINSI
FINPROC
```

Programme ***4.5. Utilisation d'une butée***

Chaque fois qu'un nœud n'a pas de successeur (gauche ou droit), le programme appelle parcours(O). Le test au début de la procédure fait que l'exécution ne va pas plus loin.

Pour éliminer la récursivité, on peut utiliser une pile, dans laquelle on garde le souvenir des nœuds qui restent à parcouru (programme 4.6).

```
DONNEES succg, succd: TABLEAU [0..taille] DE entier;
    racine: entier;
DEBUT VAR n: entier INIT racine, pl: entier INIT 1,
          fini: bool INIT FAUX,
          pile: TABLEAU [1 ..tpile] DE entier;
    MONDE pl: première case libre dans pile;
    TANTQUE NON fini
    FAIRE SI n=O
        ALORS fini:= pl=1 ;
              SI NON fini
              ALORS dépiler(n)
              FINSI
        SINON empiler(succd[n]); n:=succg[n]
        FINSI
    FAIT
FIN
```

Programme ***4.6. Parcours avec pile***

43.4. Parcours préfixé et post-fixé

Considérons l'arbre binaire complet à trois niveaux de profondeur qui se trouve en figure 4.4. Dans un arbre binaire complet, tous les nœuds autres que les feuilles ont exactement deux successeurs. De plus, tout chemin de la racine à une feuille est de la même longueur (trois dans notre cas).

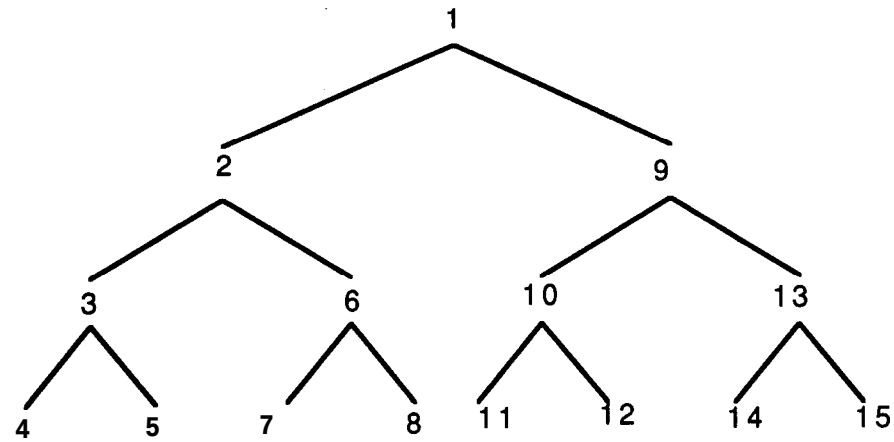

Figure ***4.4. Parcours en profondeur d'abord (ordre préfixé)***

La numérotation des nœuds de cet arbre correspond à l'ordre de visite des nœuds *dans* un parcours dit "*de profondeur d'abord*" ("depth first search"). Nous verrons par la suite que cet ordre est celui de l'évaluation dite *préfixée* ("prefixed"). Il est possible d'imaginer un parcours de l'arbre en largeur *d'abord* ("breadth first search", figure 4.5), mais le programme est plus difficile à écrire (voir exercice en fin de chapitre).

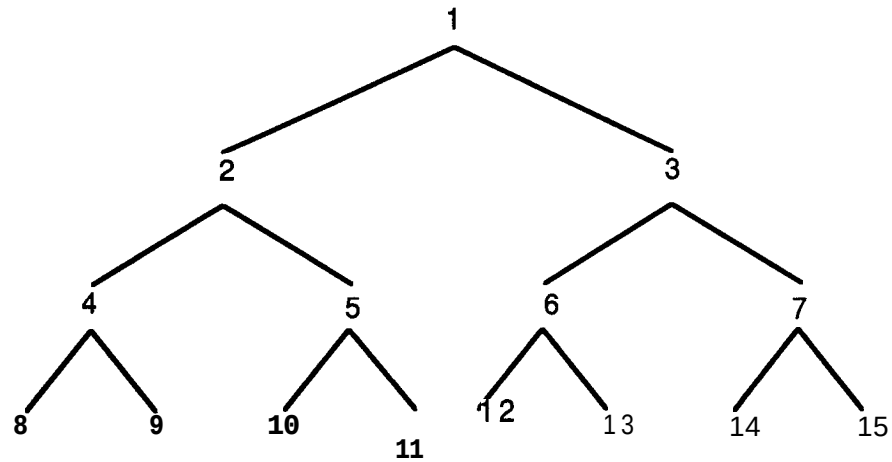

Figure ***4.5. Parcours en largeur d'abord***

http://dzdelphi.blogspot.com/

Revenons sur le parcours en profondeur d'abord, en décorant le programme avec une mention pour chaque visite d'un nœud (programme 4.7).

```
PROCEDURE  parcours(n);
    SI n>0
    ALORS visite-l ;
        parcours(sg[n]);
        visite-2;
        parcours(sd[n]);
        visite3
    FINSI
FINPROC
```

Programme ***4.7. Triple visite d'un nœud***

Ces numéros de visite correspondent au fait que, dans un parcours d'arbre, on visite chaque nœud trois fois (figure 4.6), une fois en descendant à gauche, une fois ayant terminé le parcours des successeurs gauche et avant de parcourir les successeurs droits, et une dernière fois en remontant à droite à la fin.

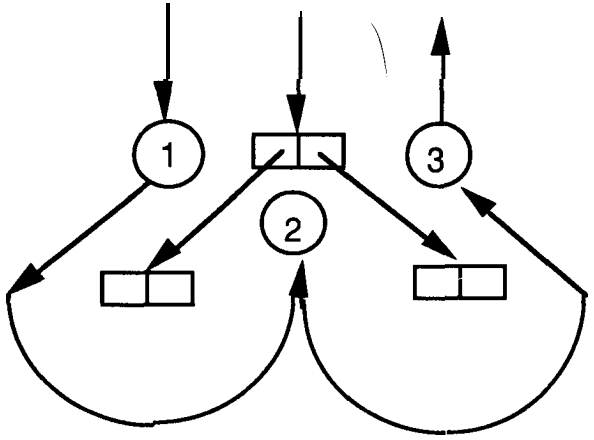

Figure **4.6.** ***Triple visite des nœuds d'un arbre***

Supposons qu'au cours du parcours en profondeur d'abord de l'arbre dans le programme 4.7, on imprimait le numéro de nœud à la place de visite-l. Alors l'ordre d'impression des nœuds serait celui de la figure 4.4, ordre dit préfixe. Imprimer les

numéros des nœuds à la place de visite-2 donnerait l'ordre ***infixé*** **("infixed")** ***(figure 4.7),*** celui donné par visite-3 étant ***post-fixé*** **("postfixed")** (figure 4.8). Ces différents ordres ont leur importance lors du traitement des expressions dans un compilateur.

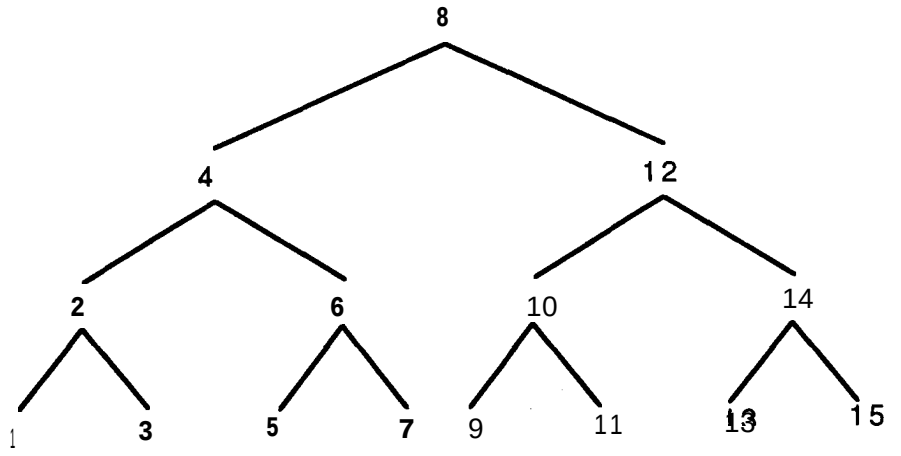

Figure ***4.7. Parcours en ordre infixé***

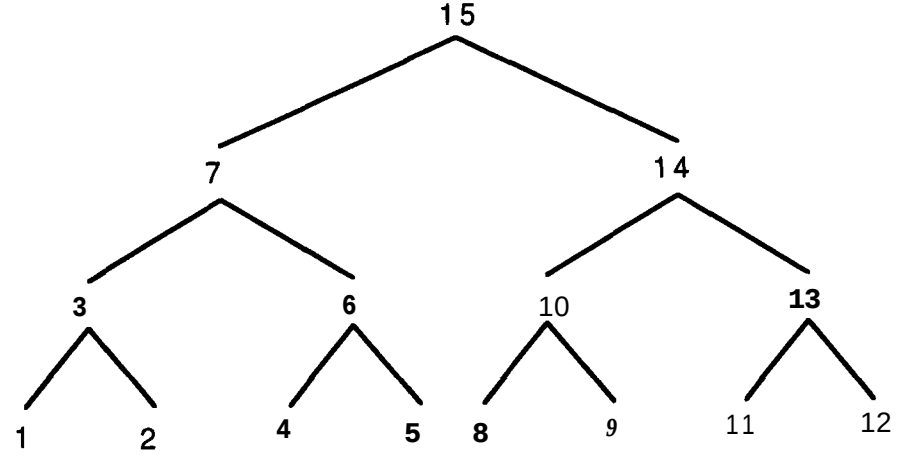

Figure ***4.8. Parcours en ordre post-fixé***

http://dzdelphi.blogspot.com/

4.4. Les treillis

Un treillis ressemble à un arbre, mais en permettant à un nœud d'avoir plus d'un prédécesseur. Les branches peuvent donc se rejoindre. Néanmoins, il n'est pas possible de boucler, c'est-à-dire que tous les chemins vont vers l'avant, en terminant sur une feuille. Nous retrouverons le problème des boucles dans le paragraphe sur les graphes. La figure 4.9 donne un exemple de treillis.

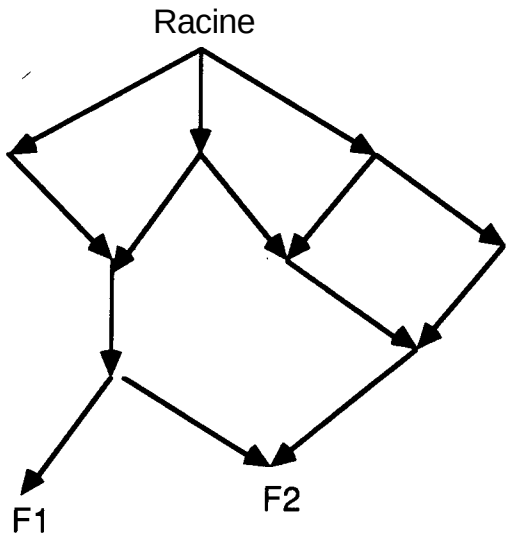

Figure ***4.9. Un treillis***

Notons, dans cette figure, que tous les chemins qui commencent à'la racine terminent sur l'une des feuilles F1 ou F2 dans un nombre fini de pas.

On peut parcourir un treillis comme un arbre, par exemple avec un des programmes du paragraphe précédent. Ce parcours visitera certains nœuds plusieurs fois, car ils ont plus d'un prédécesseur. En fait, chaque nœud sera visité autant de fois qu'il y a de chemins qui mènent de la racine jusqu'à lui. Il se peut que l'existence de visites multiples soit gênante, pour des raisons d'efficacité ou de répétition intempestive d'opérations. Pour visiter chaque nœud une et une seule fois, on applique l'algorithme de parcours d'un graphe (voir ci-dessous). En fonction des besoins immédiats, on traite donc un treillis comme un arbre ou, le plus souvent, comme un graphe.

4.5. Les graphes

Un graphe est toujours une collection de nœuds et de branches, mais sans limitations. Ceci permet des boucles, c'est-à-dire qu'il peut exister un chemin qui mène du nœud n jusqu'à lui-même. Le réseau du métro parisien est un exemple de graphe, où l'on peut revenir au point de départ d'un voyage. On voit qu'un treillis est un cas particulier du graphe, un arbre étant un cas particulier du treillis.

Pour des raisons pragmatiques n'ayant rien à faire avec la théorie, les informaticiens limitent souvent les graphes qu'ils manipulent. Ainsi, on prendra souvent la forme de vigne, afin de ne parler que de graphes binaires. De même, certains programmes ne travaillent que sur des graphes possédant une racine, c'est-à-dire que seront considérés comme étant nœuds du graphe, les nœuds qui sont accessibles à partir de la racine. La notion de racine dans un graphe n'a pas de justification théorique; elle correspond à une volonté de simplifier les programmes de traitement.

Dans le cas général, parcourir un graphe binaire à partir de sa racine avec l'algorithme donné pour les arbres mènerait à une boucle, car l'algorithme suit tous les chemins possibles. Pour visiter chaque nœud une fois, sans boucler, on a recours à la technique de marquage. Le programme garde un drapeau booléen pour chaque nœud, indiquant si le nœud a déjà été visité. On ne visite pas deux fois un nœud (programme 4.8).

```
DONNEES taille: entier;
    succg, succd: TABLEAU [0..taille] DE entier;
    racine:  entier;
DEBUT VAR marque: TABLEAU [0..taille] DE bool INIT FAUX;
    PROCEDURE visiter(n: entier);
        SI NON marque[n]
        ALORS marque[n]:=VRAI;
              visiter(succg[n]);      visiter(succd[n])
        FINSI
    FINPROC;
    visiter(racine)
FIN
```

Programme ***4.8. Visite d'un graphe***

Ce programme suppose l'existence d'un nœud de butée 0, qui est marqué. Ainsi, le test de marquage et le test d'existence ne font qu'un. Notons que la

http://dzdelphi.blogspot.com/

représentation d'un graphe sur un couple de vecteurs succg et succd est la même que celle d'un arbre. On peut enlever la récursivité de cet algorithme en utilisant une pile, exactement comme dans le cas d'un arbre. Par la suite, nous montrerons une méthode de parcours d'un graphe sans récursivité mais aussi sans pile.

En général, on parcourt un treillis comme un graphe, c'est-à-dire en marquant les nœuds visites. Cela évite les visites répétées aux nœuds qui sont sur plusieurs chemins.

4.51. *Algorithme de Schorr-Waite*

L'algorithme classique de parcours d'un graphe, comme celui d'un arbre, met en jeu, soit un système d'appels récursifs, soit une pile. Une application fréquente de l'algorithme se trouve dans des interpréteurs pour des langages tels que LISP, où, quand la mémoire affectée est pleine, il faut examiner les données en cours afin de récupérer de l'espace dans la mémoire principale. Pendant l'exécution d'un programme, on crée des nœuds et des arcs, on en supprime et on les modifie. Après un certain temps, la partie de la mémoire affectée au graphe est pleine. Mais, suite aux différentes manipulations, certains nœuds ne sont plus utiles. L'utilité d'un nœud est définie par son accessibilité. Tout nœud accessible à partir de la racine est utile (il peut toujours servir). Un nœud qui n'est pas accessible ne peut plus servir. Pour récupérer l'espace occupé par les nœuds inaccessibles, on commence par le marquage de tous les nœuds accessibles. Par la suite, dans une deuxième phase, on examine tout l'espace utilisable, en récupérant les cases non marquées. Ce processus s'appelle le *ramasse* miettes ("garbage collection").

Le problème vient du fait que le ramassage des miettes intervient au moment où l'on constate que la mémoire est pleine. On ne souhaite pas alors lancer une procédure récursive, étant donné que son exécution nécessite l'ouverture d'une pile de taille non prévisible. Une solution de ce problème existe [Schorr 19671. Il s'agit de réutiliser les pointeurs pour mettre en œuvre la notion de prédécesseur.

Considérons le parcours classique d'un graphe. La figure 4.6 a déjà montré ce parcours sur trois nœuds d'un arbre. Chaque nœud est visité trois fois. Pour se rappeler où il en est, l'algorithme de Schorr-Waite garde dans la marque le nombre de visites au nœud déjà effectuées:

marque-0	nœud non visité
marque=1	en train de visiter la descendance gauche
marque=2	descendance gauche terminée, visite de la descendance droite en cours
marque=3	le marquage de tous les successeurs du nœud a été effectué

Avec cette représentation de la marque, l'algorithme classique prend la forme du programme 4.9.

```
PROC parcours(n);
    SI m[sg[n]]=O
    ALORS m[n]:=I ;
        parcours(sg[n])
    FINSI;
    SI m[sd[n]]=0
    ALORS m[n]:=2;
        parcours(sd[n])
    FINSI;
    m[n]:=3
FINPROC
```

Programme *4.9. Marquage avec troisvaleurs*

En enlevant la récursivité de cette procédure, on peut déduire le programme 4.10.

```
DEBUT n:=racine;
    TANTQUE NON fini
    FAIRE CAS m[n]
        DANS
        0:    m[n]:=1;
              SI m[sg[n]]=O
              ALORS n:=sg[n]
              FINSI,
        1:    m[n]:=2;
              SI m[sd[n]]=0
              ALORS n:=sd[n]
              FINSI,
        2:    m[n]:=3; n:=pred[n]
        FINCAS
    FAIT
FIN
```

Programme 4.10. ***Version sans récursivité***

http://dzdelphi.blogspot.com/

Dans ce programme, la condition de terminaison est que tous les successeurs de la racine ont été marqués. Dans ce cas, l'algorithme remonte à la racine pour la dernière fois, c'est-à-dire que m[racine]=3. Le programme utilise toujours la notion de butée (un successeur absent pointe vers le nœud fictif 0, qui est marqué), mais il teste si le successeur éventuel (droite ou gauche) est marqué avant d'y aller.

La difficulté de cette version du parcours vient du fait que l'on ne sait pas "remonter" vers le prédécesseur (n:=pred[n]) après avoir passé la valeur de la marque à 3. La structure des données ne comporte que des pointeurs vers les successeurs de chaque nœud. Pour résoudre le problème, Schorr et Waite ont proposé de renverser le pointeur vers le successeur suivi afin d'indiquer, de manière temporaire, le prédécesseur du nœud. Pour ce faire, on garde, à tout moment, dans une variable p, la valeur précédente de n, le nœud courant. Considérons la situation quand l'algorithme arrive pour la première fois sur le nœud 1 de la figure 5.6 :

n=I, marque[1]=0, p=prédécesseur(I), sg[1]=2, sd[1]=3.

Appelons les cases qui indiquent les successeurs d'un nœud sg[n] et sd[n]. L'algorithme va procéder au marquage de la descendance gauche. n va donc prendre la valeur de sg[n], p prenant la valeur de n (il suit toujours n avec un temps de retard). Comme n a pris la valeur du successeur gauche, la case qui contient l'index de ce successeur fait maintenant double emploi. On le réutilise temporairement pour se rappeler du prédécesseur (l'ancienne valeur de p). On arrive à la situation suivante :

n=2, marque[I]=I , p=1, sg[I]=prédécesseur(I), sd[1]=3.

A la fin du marquage de la descendance gauche, c'est-Mire quand marque[2] passe à 3, l'algorithme remonte sur le nœud 1 dans l'état suivant, avec p toujours juste derrière n :

n=I, marque[1]=1, p=2, sg[I]=prédécesseur(I), sd[1]=3.

On va maintenant repartir à droite, ayant restauré le successeur gauche et en conservant le prédécesseur du nœud 1 dans sd[1] pendant la visite :

n=3, marque[1]=2, p=1, sg[1]=2, sd[I]=prédécesseur(I).

En remontant de la droite, on aura :

n=1, marque[1]=2, p=3, sg[1]=2, sd[1]=prédécesseur(1).

n va remonter vers le prédécesseur, en laissant le nœud 1 dans son état de départ :

n=prédécesseur(1), marque[1]=3, p=I, sg[1]=2, sg[1]=3.

On retrouvera un résumé de ces états dans la figure 5.6.

Avec cette introduction, nous arrivons au programme 4.11.

```
DONNEES Un graphe codé en forme de vecteurs comme avant
DEBUT VAR n, p: entier;
    n:=racine; p:=0;
    TANTQUE m[racine]<3
    FAIRE CAS m[n]
        DANS
        0 : m[n]:=1 ;
            SI m[sg[n]]=0
            ALORS % La voie est libre à gauche %
                cycle(n, sg[n], p)
            SINON % La voie n'est pas libre, mais on fait comme si l'on y
était allé et comme si l'on en revenait %
                cycle(sg[n], p)
            FINSI,
        1 : m[n]:=2;
            SI m[sd[n]]=O
            ALORS % La voie est libre à droite %
                cycle(n, sd[n], sg[n], P)
            SINON % Comme si l'on revenait de droite %
                cycle(sg[n], p, sd[n])
            FINSI,
        2: m[n]:=3; cycle(n, sd[n], p)
        FINCAS
    FAIT
FIN
```

Programme 4.11 . *Mise en œuvre du prédécesseur*

Le programme comporte deux points particuliers : l'instruction cycle et ce qui se fait quand le chemin n'est pas libre, à droite ou à gauche.

L'instruction cycle correspond à une affectation multiple. Ainsi, considérons le cycle suivant :

cycle(n, sg[n], P)

http://dzdelphi.blogspot.com/

Il correspond à l'affectation multiple :

```
(n, sg[n], p) := (sg[n], p, n)
```

Pour exécuter une affectation multiple, on calcule les valeurs à droite, puis on les affecte, en parallble, dans les variables indiquées à gauche. Cette façon d'écrire évite de considérer des effets de bord qui pourraient résulter de l'ordonnancement des affectations individuelles. Par exemple, la séquence suivante d'affectations donnerait un résultat incorrect :

```
n:=sg[n];
sg[n]:=p;
p:=n
```

La première affectation à n fait que les deux autres affectations repèrent la nouvelle valeur de n quand c'est l'ancienne qui est recherchée. Il faudrait sauvegarder l'ancienne valeur de n :

```
xn:=n;
n:=sg[xn];
sg[xn]:=p;
p:=xn
```

D'autres séquences de code donnent le même résultat.

Quand le successeur prévu est marqué, soit parce qu'il n'existe pas, soit parce qu'il est en cours de traitement sur un autre chemin, le programme ne doit pas le prendre en compte. Mais les pointeurs doivent être mis dans un état qui correspond à l'augmentation de la valeur de m[n]. L'algorithme met à jour les pointeurs comme si n revenait du chemin qui mène au successeur marqué, bien qu'il n'y soit jamais allé.

4.5.2. Amélioration de l'algorithme de Schorr-Waite

Cette présentation vient de [Griffiths 1979], en réponse à celle de [Gries 1979]. C'est ce dernier qui a transmis l'idée de l'astuce qui permet de raccourcir le programme ci-dessus. Nous ne connaissons pas l'identité de l'inventeur de l'astuce.

Considérons les états par lesquels passent les noeuds au cours du traitement (figure 4.10).

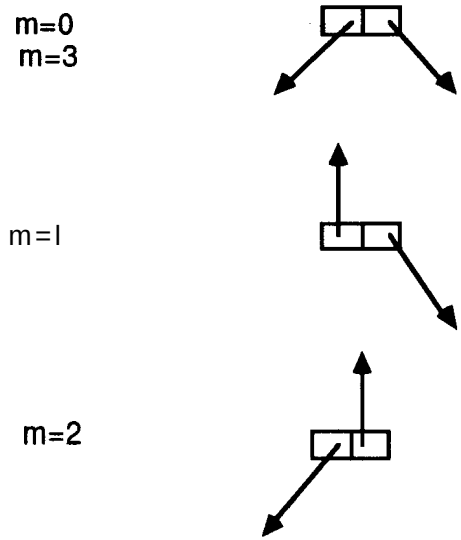

Figure 4.10. *Etats dans Schorr-Waite*

La nouvelle id& est de garder le pointeur vers le prédécesseur dans la case du successeur que l'on ne prend pas. La figure 4.11 montre les états résultants, dans lesquels le pointeur vers le descendant restant doit obligatoirement changer de place.

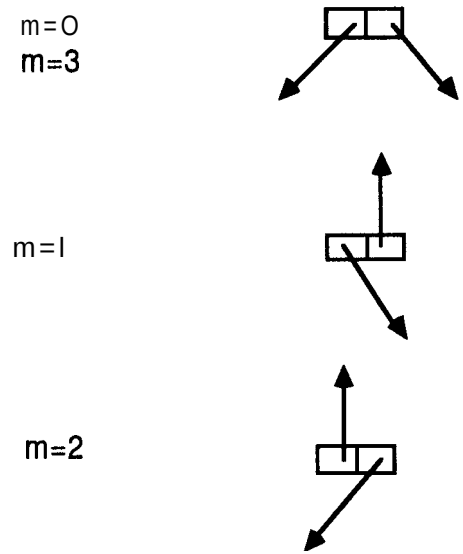

Figure 4.11 . *Etats dans Schorr-Waite amélioré*

http://dzdelphi.blogspot.com/

Cette idée, à priori un peu bizarre, donne lieu au programme 4.12.

```
DONNEES Un graphe codé en forme de vecteurs comme avant
DEBUT VAR n, p: entier;
      n:=racine; p:=0;
      TANTQUE m[racine]<3
      FAIRE CAS m[n]
            DANS
            0: m[n]:=I ;
                 SI m[sg[n]]=0
                 ALORS cycle(n, sg[n], sd[n], p)
                 SINON cycle(sg[n], sd[n], p)
                 FINSI,
            1 : m[n]:=2;
                 SI m[sg[n]]=0
                 ALORS cycle(n, sg[n], sd[n], p)
                 SINON cycle(sg[n], sd[n], p)
                 FINSI,
            2: m[n]:=3; cycle(n, sg[n], sd[n], p)
            FINCAS
      FAIT
FIN
```

Programme 4.12. *Schorr-Waite amélioré*

On voit que, dans cette version du programme, les mêmes cycles se répètent. En regroupant, d'une part, les cycles identiques et, d'autre part, les augmentations des valeurs de m[n], on obtient le programme 4.13.

```
DEBUT VAR n, p: entier;
      n:=racine; p:=0;
      TANTQUE m[racine]<3
      FAIRE m[n]:=m[n]+I ;
            SI m[n]=3 OUALORS m[sg[n]]=0
            ALORS cycle(n, sg[n], sd[n], p)
            SINON cycle(sg[n], sd[n], p)
            FINSI
      FAIT
FIN
```

Programme *4.13. Version condensée*

Cette version finale du programme, très condensée, est assez agréable du point de vue de l'esthétique, mais elle a demandé un travail considérable. C'est un bon exercice de style. En plus, le programme résultant peut être repris tel quel et installe dans un ramasse-miettes performant.

4.5.3. *Représentation de graphes sur une matrice binaire*

On peut aussi représenter un graphe sur une matrice binaire, ce qui facilite le traitement de graphes n-aires. Un 1 à l'intersection de la ligne Y et de la colonne X veut dire que le nœud X est un successeur du nœud Y. Considérons le graphe de la figure 4.12.

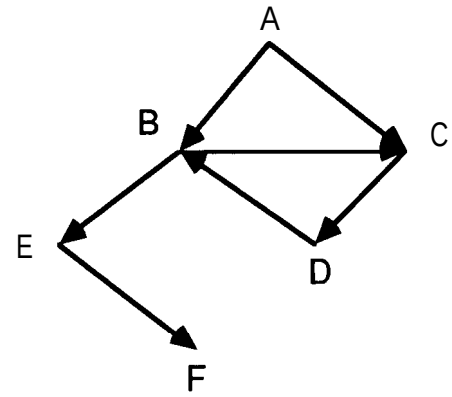

Figure 4.12. *Un graphe*

Sa représentation en forme de matrice binaire se trouve sur la figure 4.13.

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	0	1	0
c	0	0	0	1	0	0
D	0	1	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Figure 4.13. *Représentation sur une matrice binaire*

http://dzdelphi.blogspot.com/

Un 1 représente un arc qui mène du nœud indiqué par la ligne à celui de la colonne, un 0 indiquant l'absence d'un tel arc. Cette représentation est plus compacte que la table utilisée auparavant dans le cas d'un graphe permettant de multiples successeurs (graphes n-aires). Elle se prête aussi à certaines classes de manipulations, comme dans le paragraphe suivant.

4.5.4. *Fermeture transitive*

La fermeture d'une relation est une opération bien connue des mathématiciens. Elle résulte de la transitivité de l'opération considérée. Par exemple, cette propriété nous permet de déduite:

```
SI a<b ET b<c
ALORS a<c
FINSI
```

Intuitivement, on dit que "les amis des amis sont des amis". Considérons la notion de successeur dans un graphe. Dans la matrice du paragraphe précédent, un 1 dans la position (x,y) indique qu'il existe un arc entre le nœud y et le nœud x (x est successeur de y). Maintenant, si x est un successeur de y, et si y est un successeur de z, on peut déduire que x est un successeur de z (en passant par y), c'est-à-dire qu'il existe un chemin entre z et x. La fermeture transitive d'un graphe donne sa connectivité. Pour chaque nœud n, on obtient la liste complète des nœuds qui sont accessibles à partir de n.

La figure 4.14 montre la fermeture transitive du graphe de la figure 4.13. On voit, dans la première ligne, qu'il est possible, en partant de A, d'atteindre les nœuds B, C, D, E, F. En revanche (première colonne), A n'a pas de prédécesseur. On voit bien que A est la racine d'un graphe connexe. De la même façon, dans la dernière ligne, on voit que F n'a pas de successeur, c'est-à-dire que F est une feuille.

	A	B	C	D	E	F
A	0	1	1	1	1	1
B	0	1	1	1	1	1
c	0	1	1	1	1	1
D	0	1	1	1	1	1
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Figure *4.14. Fermeture transitive du graphe de la figure 4.13*

Les nœuds B, C, D présentent la particularité d'être leurs propres successeurs (un 1 sur la diagonale principale). C'est une indication de la présence dune boucle (on peut aller de B jusqu'à B). Le graphe n'est donc pas un treillis et, à fortiori, n'est pas un arbre.

L'algorithme direct de calcul dune fermeture transitive suit la définition : "SI B est un successeur de A, ALORS tous les successeurs de B sont également successeurs de A". Considérons la ligne A de la figure 4.13. Elle comporte des 1 dans les positions B et C. Ainsi, les successeurs de B sont des successeurs de A, comme le sont les successeurs de C. Pour les successeurs de B, on prend la ligne B en l'ajoutant à la ligne A par l'opération OU inclusive. On fait de même pour chaque 1 de la ligne A, que le 1 soit d'origine ou qu'il résulte dune addition. On ne reprend pas deux fois un 1 dans la même position. Pour la ligne A, cela donne les opérations de la figure 4.15.

	A	B	C	D	E	F	
ligne A au départ :	0	1	1	0	0	0	
addition de la ligne B :	0	1	1	0	1	0	
addition de la ligne C :	0	1	1	1	1	0	
addition de la ligne D :	0	1	1	1	1	0	(sans changement)
addition de la ligne E :	0	1	1	1	1	1	
addition de la ligne F :	0	1	1.1	1	1		(sans changement)

Figure *4.15. Fermeture transitive, une ligne*

Aucun nouvel 1 n'ayant été introduit, on a terminé avec la ligne A. La même opération a lieu pour chaque ligne du graphe. Cet algorithme n'est pas optimal. Il a été amélioré à plusieurs reprises [Warshall 1962], [Griffiths 1969].

4.6. Ordres partiels et totaux

Les quatre structures, les files, les arbres, les treillis et les graphes, ont des propriétés d'ordonnancement importantes. Dans une file, il existe un ordre total, c'est-à-dire que pour chaque couple (x, y) d'éléments, une et une seule des relations suivantes est vraie (pos est la position de l'élément donné) :

```
pos(x) < pos(y)
ou pos(x) > pos(y)
```


http://dzdelphi.blogspot.com/

Cette propriété fait qu'il n'existe qu'un seul ordre d'écriture dans lequel les éléments se trouvent chacun à leur place. Elle est évidente dans le cas d'une file (voir le cas des entiers positifs).

Dans un arbre ou dans un treillis, il n'existe que des ordres partiels. Un ordre doit respecter la règle suivante :

SI **x est successeur de y**
ALORS pos(x) > pos(y)
FINSI

Considérons l'arbre binaire trivial de la figure 4.16.

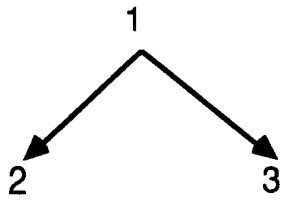

Figure *4.16. Arbre binaire simple*

La règle d'ordonnancement est respectée par les deux séquences suivantes :

1 2 3
1 3 2

La figure 4.4 (préfixé ou en profondeur), comme la figure 4.5 (en largeur), présentent des ordres partiels dans un arbre. Les ordres des figures 4.7 (infixé) et 4.8 (post-fixé) ne respectent pas la règle. Les arbres et les treillis permettent toujours les ordonnancements partiels des éléments les constituant (il existe un seul ordre, c'est-à-dire un ordre total, si et seulement si aucun nœud n'a plus d'un successeur direct).

L'ordonnancement n'est plus possible pour un graphe qui comporte une boucle. En effet, comme la boucle indique qu'il existe au moins un élément qui est son propre successeur, il n'est plus possible de respecter la règle d'ordonnancement par la relation de succession.

4.7. Exercices

REMARQUE. — La notion de pointeur n'a pas été introduite. Il faut donc gérer des tables d'index.

1. On considère des familles de type un peu particulier. Il y a un seul parent, avec au plus deux enfants. Un enfant peut lui-même être parent d'autres enfants, et ainsi de

suite. Le tout constitue donc un arbre binaire dont les nœuds sont décorés (dans le champ val) avec le nom de la personne représentée. Un nom est une suite de lettres de 8 caractères au plus. La racine de l'arbre s'appelle "adam" (ou "ève", si l'on préfère). On écrira les procédures récursives suivantes :

- **PROC trouver(nom, n) -> (bool, entier).**
  La variable nom contient le nom d'une personne. Le booléen retourné en résultat sera VRAI si et seulement si la personne nommée est présente dans l'arbre ou le sous-arbre de racine n. Si le nom est présent l'entier retourné en résultat contiendra l'index du nœud comportant le nom.

- **PROC parent(p, enf, n) -> (bool, entier).**
  Comme pour trouver, mais on cherche un couple parent - enfant de noms p et enf. L'entier résultant contiendra, le cas échéant, l'index du parent.

- **PROC cousin_germain(a, b, n) -> (bool, entier, entier).**
  Recherche de cousins germains de noms a et b.

- **PROC aïeul(a, enf, n) -> (bool, entier, entier).**
  Recherche des liens entre l'aïeul de nom a et son descendant enf. On imprimera (dans le bon ordre) les noms des personnes dans la descendance (a est parent de b, b est parent de c, . . . , i est parent de enf), ou un message approprié si enf n'est pas un descendant de a.

2. Par la suite, on peut se poser les mêmes questions avec des familles plus naturelles, où chaque enfant a un père et une mère, avec des mariages stables et monogames. On garde le contrôle des naissances (pas plus de deux enfants par famille). On engendre un treillis.

On bâtira un système conversationnel qui accepte des informations des types suivants :

- a se marie avec b,
- naissance de l'enfant e du couple m, f.

L'arbre familial est un fichier sur disquette. Les procédures de recherche peuvent prendre la forme de questions (interface ?) comme :

- **Qui est le père de x ?**
- Qui est la grand-mère maternelle de y ?
- Des questions de l'exercice 1.

3. Dans un arbre binaire complet à n niveaux, combien y a-t-il de nœuds ? Combien de feuilles ?

4. Ecriture du programme de parcours d'un arbre binaire en largeur d'abord.

http://dzdelphi.blogspot.com/

Chapitre 5

Récurrence et récursivité

Que ce soit pour la résolution de problèmes, pour la spécification d'algorithmes, voire même pour leur mise en œuvre, la *récurrence* ("induction") est un outil mathématique essentiel. Tout informaticien se doit de la maîtriser. Heureusement, la récurrence est une technique simple, en dépit de sa puissance.

Dans les chapitres précédents, nous avons déjà vu des programmes basés sur la récurrence. Il est d'ailleurs difficile d'imaginer des programmes qui n'y font pas appel, car la simple boucle en est une forme. L'informaticien parle souvent de réduire un problème en n vers un problème en n-l. Dans d'autres cas (diviser pour régner ou parcours de graphes, par exemple), on parle de la réduction d'un problème en n vers deux problèmes en n/2. Il s'agit maintenant de rendre le lecteur plus conscient du mécanisme et de sa puissance.

Au niveau de la programmation, nous avons déjà vu que la récurrence peut se *traduire* par la *récursivité* ("recursion") ou par des itérations, avec ou sans pile. Le choix de méthode dépend de la complexité du problème considéré, de la puissance du langage de programmation utilisé et de l'efficacité requise.

Comme d'habitude, nous abordons les techniques à travers des exemples. Ceux-ci sont souvent bien connus, car c'est un sujet sur lequel beaucoup de collègues ont déjà travaillé.

5.1. L'exemple type - les tours d'Hanoi

Si ce problème est utilisé comme exemple dans tous les cours de ce type, ce n'est pas un hasard. Il est même arrivé au niveau des journaux de vulgarisation

http://dzdelphi.blogspot.com/

[Bezert 1985]. La source la plus ancienne que nous connaissons est de E. Lucas, sous le pseudonyme de M. Claus, en 1883. Les tours d'Hanoi donnent lieu à une solution élégante par la récurrence, une analyse itérative demandant plus de travail.

Le titre du problème vient de l'histoire racontée habituellement, qui est celle de moines bouddhistes en Asie du sud-est qui égrènent le temps en transférant des disques, tous de tailles différentes, sur un jeu de trois piquets (voir figure 5.1). Dans cette figure il n'y a que cinq disques, mais la tradition veut que les moines jouent avec 64. L'histoire est une invention du dix-neuvibme siècle, Lucas la plaçant à Bénarès (en Inde). On ne sait pas comment elle s'est resituée à Hanoi . . .

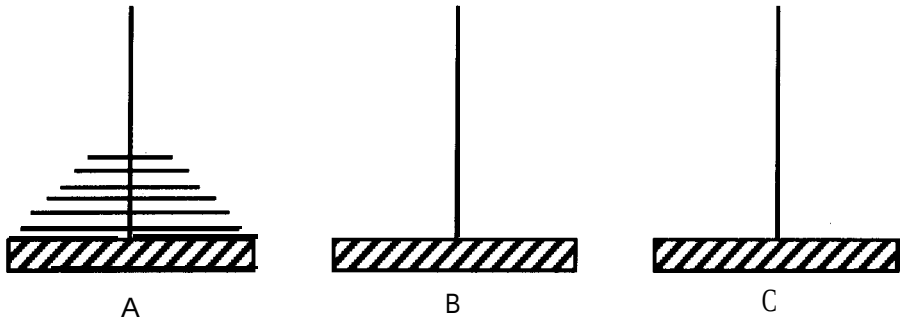

Figure 5.1. ***Position de départ des tours d'Hanoi***

Le jeu consiste à transférer la pile de disques du piquet A vers le piquet B, en utilisant C comme piquet de manœuvre, tout en respectant les deux règles suivantes :

- un disque ne peut pas être posé sur plus petit que lui,
- on ne déplace qu'un disque à la fois.

La solution la plus simple vient de la réponse à la question suivante : "si je savais transférer n-l disques, saurais-je transférer n ?". La réponse est oui, car autrement nous n'aurions pas posé la question. Ainsi, pour transférer n disques de A à B, on commence par le transfert de n-l disques de A à C, suivi du déplacement du dernier disque (le plus gros) de A à B, suivi du transfert des n-l disques de C à B. Cela donne la procédure du programme 5.1.


```
PROCEDURE hanoi(n, a, b);
    SI n=l
    ALORS déplacement(a, b);
    SINON hanoi(n-1, a, 6-a-b);
          déplacement(a, b);
          hanoi(n-1,6-a-b, b)
    FINSI
FINPROC
```

Programme 5.1. ***Tours d'Hanoi, version de base***

Dans cette procédure, nous avons supposé que les piquets sont numérotés 1.2, 3. Déplacer un disque se fait par la procédure déplacement. Pour n=l le deplacement est immédiat, autrement on applique l'algorithme décrit ci-dessus, 6-a-b étant le numéro du troisième piquet (6=1+2+3, donc en soustrayant deux des trois possibilités de 6 on obtient la troisième).

La figure 5.2 montre les appels en cascade de la procédure hanoi (notée h) pour le transfert de trois disques. Le déplacement d'un disque est noté d.

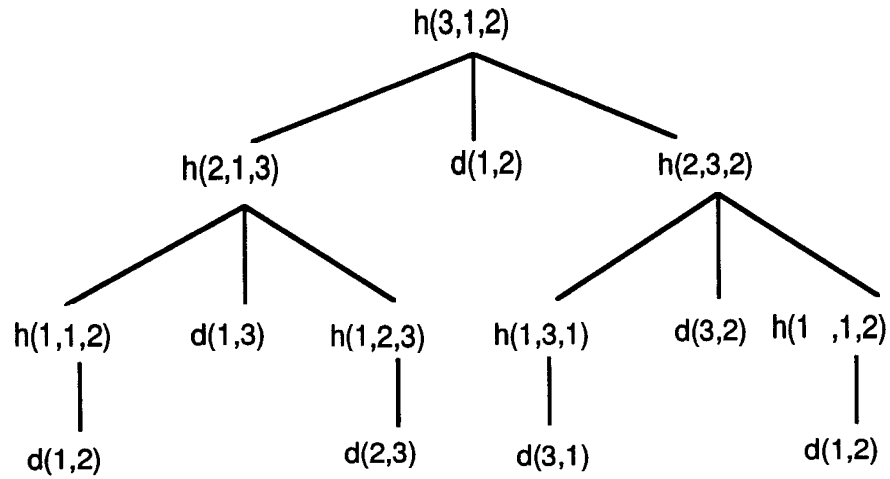

Figure ***5.2. Arbre d'appels pour trois disques***

L'arbre se lit de la manière suivante :

- Pour transférer 3 disques du piquet 1 au piquet 2 (racine de l'arbre), on

http://dzdelphi.blogspot.com/

transfère deux disques de 1 à 3, un disque de 1 à 2, puis deux disques de 3 à 2 (successeurs de la racine).

- Les successeurs de type h sont décomposés de la même manière, en appliquant récursivement la procédure.
- Les feuilles de l'arbre sont des déplacements d'un seul disque.

En reprenant les feuilles de l'arbre dans l'ordre, de gauche à droite, on déduit les déplacements simples, dans l'ordre, pour le cas de trois disques :

d(1,2) d(1,3) d(2,3) d(1,2) d(3,1) d(3,2) d(1,2)

Les sceptiques peuvent toujours essayer avec trois pièces de monnaie !

5.1.1. Coût de l'algorithme

Se pose maintenant la question de savoir combien de déplacements individuels sont nécessaires pour transférer n disques. En considérant le programme du paragraphe précédent, on voit que le transfert de n disques coûte deux fois le prix du transfert de n-l disques plus 1 déplacement, par le fait que :

h(n)= h(n-1)+ 1 + h(n-1)

On obtient les équations de récurrence suivantes, avec c représentant le coût de l'opération, mesuré en nombre de déplacements individuels :

$c_1 = 1$

$c_n = 2 * c_{n-1} + 1$ pour n>1

Pour voir la solution de ces équations de récurrence, considérons quelques cas :

n	c_n
1	1
2	3
3	7
4	15
...	

On voit intuitivement que

$c_n = 2^n - 1$

En substituant cette valeur dans l'équation, la solution est confirmée.

5.1.2. Une analyse plus poussée

La solution récursive au problbme des tours d'Hanoi est satisfaisante pour l'esprit et marche dans l'ordinateur. Mais elle comporte l'inconvénient de ne pas être très pratique pour le joueur humain. Mettons nous un instant à la place des moines, qui déplacent leurs disques à la main. Quel disque faut-il déplacer à un moment donné ? L'algorithme récursif nécessite la mise à jour constante d'une pile pour prendre en compte les appels en cours, mais il ne serait pas très pratique pour les moines de gérer une telle pile.

En fait, un peu de réflexion nous permet de faire mieux, au moins pour jouer à la main. Pour obtenir des règles simples pour le joueur humain, nous avons besoin de quelques théorèmes.

Théorème 1.

Dans une position quelconque, il y a au plus trois déplacements possibles.

Preuve.

Supposons qu'aucun piquet n'est vide. Comparons les tailles des trois disques en haut des piquets. Le plus gros des trois ne peut pas bouger, car il faudrait le poser sur plus petit que lui. Le moyen peut se poser sur le plus gros (une possibilité), mais pas sur le plus petit (qui est plus petit que lui). Le petit peut passer sur l'un ou l'autre des deux autres piquets, ce qui donne trois possibilités en tout. Maintenant, si un des piquets est vide, il joue le rôle du plus gros disque, qui ne peut pas bouger de toute façon, et l'argument ne change pas. Si tous les disques sont sur le même piquet (deux piquets vides), c'est-a-dire dans la position de départ, il n'y a que deux possibilités : le plus petit peut aller sur l'un ou l'autre des deux piquets restants.

Théorème 2.

Le disque moyen ne doit pas bouger deux fois de suite.

Preuve.

Sur les piquets de départ et d'arrivée, les autres disques sont tous plus gros que lui. Sur le troisième piquet, il y a le plus petit. Après un déplacement, le moyen reste donc le moyen. Son seul déplacement possible serait de retourner d'où il vient, ce qui serait parfaitement inutile.

http://dzdelphi.blogspot.com/

Théorème 3.

Le plus petit ne doit pas bouger deux fois de suite.

Preuve.

Ce n'est pas la peine, car il aurait pu y aller en un coup.

Tbéoréme 4.

Tous les déplacements d'index impair sont des déplacements du plus petit disque, les déplacements pairs étant du moyen.

Preuve.

C'est le petit qui commence dans la position de départ, déplacement 1. Par la suite, le moyen et le plus petit alternent, d'après les théorèmes 1-3.

Nous avons démontre que pour savoir qui bouge, il suffit de savoir si le dernier transfert était du petit disque ou du moyen. Le prochain déplacement concerne donc l'autre disque mobile. Comme les déplacements du disque moyen sont imposés (une seule possibilité), ils ne posent pas de problbme. Reste à trouver une règle pour les déplacements du plus petit.

Théorème 5.

Le plus petit disque parcourt les piquets dans un ordre fixe, c'est-à-dire qu'il fait des cercles.

Preuve.

Par récurrence sur n. Supposons que la propriété soit vraie pour n- 1, c'est-à-dire que le petit disque tourne, par exemple, dans l'ordre a b c a b c . . . On sait que (formule F) :

```
n>1 => h(n, a, b) = h(n-1, a, c); d(a,b); h(n-1, c, b)
```

Comme le d(a,b) ne concerne pas le plus petit disque, si les deux appels de h(n-1, . ..) font tourner le petit disque dans le même sens, alors la propriété est vraie

pour n (le tour continue). Comme la propriété est vraie au niveau de trois disques (voir les déplacements résultants de la figure 5.2), elle est vraie pour n disques, n>3.

Cette preuve est juste, mais elle pose souvent des probltmes aux élèves en raison du changement apparent de la direction du cercle entre le niveau n et le niveau n-l. C'est un faux problème, mais il indique une autre propriété intéressante. Supposons que le transfert complet souhaite est de A à B. Alors, si le nombre total de disques est impair, le premier déplacement (nécessairement du plus petit) est également de A à B. Mais, si le nombre total de disques est pair, le premier déplacement est de A à C (l'autre possibilité). Avec un nombre impair de disques, le cercle est

```
abcabc...
```

Avec un nombre pair, il est

```
acbacb...
```

Si n est pair, n-l est impair, et inversement. Mais, en regardant les paramètres des appels de n et de n-l dans le formule F ci-dessus, on se rend compte de l'inversion de la direction.

Nous pouvons donc déduire la règle pour les moines, qui doivent transférer, rappelons le, 64 disques. Pour effectuer un transfert de A à B, le premier deplacement est de A à C (64 est un nombre pair, pour un nombre impair de disques le deplacement aurait été de A à B). Par la suite, il suffit de se rappeler si le coup qui vient d'avoir lieu était un déplacement du plus petit disque, ou non :

- Si le déplacement précédent était du plus petit disque, on déplace le moyen (une seule possibilité).
- Si le déplacement précédent était du disque moyen, on déplace le plus petit vers le prochain piquet dans le cycle A C B A C B . . .

5.2. Des permutations

La génération des permutations de n objets est un exercice d'algorithmique et de programmation qui présente un intérêt pédagogique certain tout en menant à des programmes utiles. La littérature scientifique comporte de nombreux papiers sur le sujet.

Nous abordons ces algorithmes d'un point de vue personnel, en fonction de découvertes aléatoires. Le premier programme date d'une des écoles de FL. Bauer.

MEDIATHEQUE LA VILLETTE

http://dzdelphi.blogspot.com/

5.2.1. Permutations par échanges de voisins

La spécification de ce probleme nous a été donnée dans les années 70, par un collègue. Malheureusement, nous ne nous rappelons plus de qui il s'agissait. Si jamais il lit ces lignes, ce serait avec plaisir que nous lui attribuerons son bien . . . Le problème est d'engendrer toutes les permutations des n éléments d'un vecteur, en appliquant les seules règles suivantes.

- Entre deux permutations successives, la seule différence résulte de l'échange de deux voisins. Notons qu'en général **v[i]** a deux voisins, **v[i-1]** et **v[i+1],** mais **v[1]** et v[n] n'ont chacun qu'un seul voisin, ce qui veut dire que le vecteur n'est pas circulaire et, en particulier, **v[1]** et **v[n]** ne sont pas des voisins.

- En numérotant chacun des objets par l'index de sa position de départ (l..n), l'échange qui a lieu entre deux permutations successives concerne l'élément le plus grand possible, sans reproduire une permutation déjà produite.

Ces deux règles impose un ordre unique de production des permutations, même si cet ordre n'est pas évident à première vue. Afin de bien cerner le problème, considérons l'ordre imposé par cet algorithme pour les permutations de quatre entiers (figure 5.3).

Numéro	Permutation	Pivot	Monde des trois
	1 2 3 4	4	1 2 3
2	1 2 4 3	4	
3	1 4 2 3	4	
4	4 1 2 3	3	
5	4 1 3 2	4	1 3 2
6	1 4 3 2	4	
7	1 3 4 2	4	
8	1 3 2 4	3	
9	3 1 2 4	4	3 1 2
10	3 1 4 2	4	
11	3 4 1 2	4	
12	4 3 1 2	2	
13	4 3 2 1	4	3 2 1
14	3 4 2 1	4	
15	3 2 4 1	4	
16	3 2 1 4	3	
17	2 3 1 4	4	2 3 1
18	2 3 4 1	4	
19	2 4 3 1	4	
20	4 2 3 1	3	
21	4 2 1 3	4	2 1 3
22	2 4 1 3	4	
23	2 1 4 3	4	
24	2 1 3 4		

Figure *5.3. Permutations de quatre entiers*

Sur cette figure, on voit que le 4 "traverse" les autres objets de droite à gauche, puis retourne au point de départ par une traversée de gauche à droite. Au moment de son changement de direction, le 4 ne peut pas revenir immédiatement, car l'échange reproduirait la permutation précédente. Chaque fois que le 4 termine une traversée, et donc va changer de direction, l'algorithme effectue un échange dans le monde des trois". Cela veut dire que les trois objets restants mènent leur vie au niveau inférieur. Le 4 traverse chaque permutation du monde des trois avant que l'algorithme en produise la prochaine. Dans le monde des trois, le trois traverse chaque permutation du monde des deux. Ainsi, on justifie l'algorithme par la formule de récurrence suivante : l'objet n traverse complètement chaque permutation des n-l objets restants. On engendre ainsi toutes les permutations de n objets. La demibre colonne de la figure 5.3 indique les permutations des trois objets qui sont engendrées à chaque changement de direction du quatrième. Le sens du mot pivot deviendra clair par la suite.

52.2. Le programme

Dans ce programme, nous travaillons sur l'entier i, qui est l'index de l'échange considéré. L'échange i transforme la permutation i en la permutation i+l. La valeur de i varie donc de 1 à factoriel(n)-1, où n est le nombre d'objets, l'échange d'index factoriel(n) reproduisant la position de départ. Les objets sont représentés par les entiers (l..n). Le vecteur v contient, à chaque instant, la dernière permutation produite.

Avec les conventions ci-dessus, nous allons considérer les questions suivantes :

- A l'échange d'index i, quel est l'objet qui doit s'échanger avec un voisin plus petit que lui ? On appelle cet objet le **pivot.**
- Dans quelle direction le pivot doit-il bouger ?
- Où se trouve le pivot dans v ?

La figure **5.3** indique le pivot dans le cas des permutations de quatre éléments. Reprenons cette séquence (figure 5.4).

```
123456789101112131415161718192021222324
444344434 4 4 2 4 4 4 3 4 4 4 3 4 4 4 fin
```

Figure *5.4. Valeurs de i et du pivot dans permutations(4)*

Dans cette figure, la Premiere ligne indique les index des échanges successifs, la deuxième le pivot pour l'échange. On voit que le quatre prend trois échanges pour

http://dzdelphi.blogspot.com/

traverser la permutation courante du monde des trois, suivi d'un échange dans ce monde des trois. On voit que chaque fois que 4 n'est pas un diviseur de i, c'est le 4 qui est le pivot. Considérons maintenant le monde des trois. Les échanges le concernant sont les 4, 8, 12, 16 et 20, tous les i multiples de 4. Il suffit de diviser ces valeurs par 4 pour retrouver la séquence 1, 2, 3, 4, 5. De manière récurrente, dans le monde des trois, le pivot est le 3, sauf si 3 divise le nouvel i (l'ancien i/4). Pour le cas général, on obtient le programme 5.2 pour le calcul du pivot.

```
DEBUT nn:=n; ni:=i;
      TANTQUE nn DIVISE ni
      FAIRE ni:=ni/nn; nn:=nn-1
      FAIT
      % nn est le pivot pour l'échange i %
FIN
```

Programme 5.2. Calcul *du pivot*

Le fait de recopier n et i dans les variables temporaires nn et ni évite de les détruire. L'opérateur DIVISE donne un résultat VRAI si son deuxième opérande est un multiple exact du premier, FAUX autrement. Dans la plupart de langages de programmation, il faudrait écrire un test du type suivant :

```
SI nn*entier(ni/nn)=ni ALORS . . .
```

Examinons maintenant le problème de savoir où en est le pivot dans ses traversées du monde inférieur. A la fin de la boucle ci-dessus, nn est le pivot et ni l'index de l'échange dans le monde des nn. Considérons les valeurs de p et q, calculées comme suit :

```
p:=entier(ni/nn); q:=reste(ni/nn);
```

Notons que q>0, car la division laisse nécessairement un reste (condition de terminaison de la boucle). On démontre que p est le nombre de traversées déjà effectuées par nn, q étant l'index de l'échange dans la traversée actuelle. Cela résulte du fait qu'une traversée nécessite nn-1 échanges avec nn comme pivot.

Avec cette information, on peut déduire les faits suivants.

- Si p est pair, le pivot procède de droite à gauche, et si p est impair, de gauche à droite. Effectivement, la première traversée, avec p=0, a lieu de droite à gauche, la deuxième, avec p=1, de gauche à droite et ainsi de suite.
- Le pivot a effectué q-l échanges dans cette traversée. Il a donc avancé de q-l

positions à partir de son point de départ.

- Le point de départ du pivot est complètement à gauche ou complètement à droite de l'ensemble des éléments du monde des nn, qui se trouvent dans des positions successives (tout élément d'ordre supérieur est en fin de traversée).

Le point de départ naturel dune traversée est la position 1 (traversée de gauche à droite) ou nn (de droite à gauche). Mais considérons, dans la figure 5.3, l'échange numéro 4, qui est le premier du monde des 3. Le pivot, dans ce cas le 3, n'est pas en position 3, mais en position 4. La raison est que le 4 a terminé une traversée de droite à gauche sans revenir. On voit qu'il faut déplacer le point de départ naturel par le nombre d'éléments d'ordre supérieur qui ont accompli un nombre impair de traversées. Tout cela nous donne le programme 5.3.

```
DEBUT DONNEES n: entier;
      VAR i, ni, nn, p, q, d, pos: entier;
      VAR perm: TABLEAU [ 1 ..n] DE entier;
      POUR i:= 1 JUSQUA n FAIRE perm[i]:=i FAIT;
      POUR i:= 1 JUSQUA fact(n)-1
      FAIRE ni:=i; nn:=n; d:=0;
           TANTQUE nn DIVISE ni
           FAIRE ni:=ni/nn; nn:=nn-1;
                SI impair(ni) ALORS d:=d+l FINSI
           FAIT;
           p:=entier(ni/nn); q:=reste(ni/nn);
           SI pair(p)
           ALORS pos:=nn+d-q+l ; échanger(perm[pos], perm[pos-1])
           SINON pos:=q+d; échanger(perm[pos], perm[pos+l ])
           FINSI
      FAIT
FIN
```

Programme 5.3. *Permutationspar échanges*

Dans ce programme, d est le déplacement créé par les traversées d'éléments d'ordre supérieur. Après la division ni/nn, la nouvelle valeur de ni représente le nombre de traversées déjà effectuées par l'ancien M. Si ce nombre est impair, nn est reste à gauche, provoquant une augmentation de la valeur de d. Le calcul de pos, la position du pivot, prend en compte les faits déduits ci-dessus.

Le calcul de pos, avec le cumul de d, peut être évité en gardant en mémoire le vecteur inverse du vecteur perm. L'élément perm[i] indique l'objet qui se trouve en

http://dzdelphi.blogspot.com/

position i. L'élément **inv[i]** va indiquer la position de l'objet i en perm. C'est donc une représentation duale de la relation biunivoque

position <-> objet

On obtient le programme 5.4.

```
DEBUT DONNEES n: entier;
    VAR i, ni, nn, p, q, pos, voisin: entier;
    VAR perm: TABLEAU [1 ..n] DE entier;
    POUR i:= 1 JUSQUA n
    FAIRE perm[i]:=i; inv[i]:=i
    FAIT;
    POUR i:= 1 JUSQUA fact(n)-1
    FAIRE ni:=i; nn:=n; d:=0;
        TANTQUE nn DIVISE ni
        FAIRE ni:=ni/nn; nn:=nn-1
        FAIT;
        p:=entier(ni/nn); q:=reste(ni/nn); pos:=inv(nn);
        SI pair(p)
        ALORS voisin:=perm[pos-1]; perm[pos-1]:=nn;
              inv[nn]:=pos-1; inv[voisin]:=inv[voisin]+l
        SINON voisin:=perm[pos+l]; perm[pos+1]:=nn;
              inv[nn]:=pos+1; inv[voisin]:=inv[voisin]-1
        FINSI;
        perm[pos]:=voisin
    FAIT
FIN
```

Programme 5.4. ***Permutations par échanges, version 2***

Dans cette version, pos est la position du pivot, voisin l'objet avec lequel il va s'échanger. Les deux vecteurs sont mis à jour de manière évidente.

L'idée de garder la représentation inverse d'une relation revient souvent dans les programmes, comme dans les bases de données. Considérons le fichier de propriétaires de voitures tenu à jour par une préfecture. On peut, par exemple, le trier dans l'ordre alphabétique des noms des propriétaires. Dans ce cas, connaître le numéro d'immatriculation de DUPONT est une opération rapide (avec un algorithme de recherche dichotomique, par exemple). Mais découvrir le nom du propriétaire du véhicule 9999XX44 devient difficile, car il faut inspecter chaque entrée dans le

fichier. On voit que la représentation choisie doit être une fonction de l'utilisation que l'on va en faire. La double représentation est souvent nécessaire, c'est-à-dire que la préfecture garderait deux fichiers, l'un trié en fonction des noms des propriétaires, l'autre par des numéros d'immatriculation.

5.2.3. Relation avec les tours d'Hanoi

La technique d'utiliser l'index du coup à jouer marche aussi dans les tours d'Hanoi. Supposons que les disques sont numérotés de 1 (le plus grand) à n (le plus petit). Pour savoir quel est l'index du disque qui doit bouger, on peut utiliser un programme semblable à celui que nous avons déduit ci-dessus pour les permutations (programme 5.5).

```
DEBUT nn:=n; ni:=i;
    TANTQUE 2 DIVISE ni
    FAIRE ni:=ni/2; nn:=nn-1
    FAIT;
% C'est le disque nn qui bouge %
FIN
```

Programme ***5.5. Encore les tours d'Hanoi***

Ce calcul dépend du fait que le plus petit disque bouge une fois sur deux, que le second bouge une fois sur deux des coups restants, et ainsi de suite.

L'idée de travailler sur les propriétés numériques de l'index du coup s'avère intéressante dans un ensemble d'applications.

5.2.4. Une variante

En relaxant la règle concernant l'échange de voisins, on peut produire un deuxième programme, similaire au premier, qui engendre toutes les permutations de n objets dans un autre ordre. Nous allons travailler sur la récurrence suivante : le pivot n traverse de droite à gauche chaque permutation du monde des n-l. Pour quatre objets, les permutations sont engendrées dans l'ordre de la figure 5.5.

http://dzdelphi.blogspot.com/

Index	Mondes		
	Les4	Les 3	Les2
0	1234	123	12
1	1243		
2	1423		
3	4123		
4	1324	132	
5	1342		
6	1432		
7	4132		
8	3124	312	
9	3142		
10	3412		
11	4312		
12	2134	213	21
13	2143		
14	2413		
15	4213		
16	2314	231	
17	2341		
18	2431		
19	423 1		
20	3214	321	
21	3241		
22	3421		
23	4321		

Figure *5.5. Nouvelles permutations de quatre objets*

Considérons le programme 5.6.


```
DEBUT DONNEES n: entier;
    VAR i, j, nn, p, q: entier;
    perm: TABLEAU [1 ..n] DE entier;
    i:=0;
    TANTQUE i<fact(n)
    FAIRE nn:=n; p:=i;
        POUR j:=1 JUSQUA n FAIRE perm[j]:=0 FAIT;
        TANTQUE nn>0
        FAIRE % placer l'objet nn %
            q:=reste(p/nn); p:=entier(p/nn); j:=n;
            TANTQUE q>O
            FAIRE % sauter q positions vides %
                SI perm[j]=0 ALORS q:=q-1 FINSI;
                j:=j-1
            FAIT,
            TANTQUE perm[j]>0
            FAIRE % trouver la prochaine position vide %
                j:=j-1
            FAIT;
            perm[j]:=nn; nn:=nn-1
        FAIT;
        % perm contient la permutation d'index i %
        i:=i+1
    FAIT
FIN
```

Programme 5.6. ***Traversée unidirectionnelle***

Pour chaque valeur de i, le programme calcule la position de chaque objet nn **(1≤nn≤n)** dans le vecteur perm. Par la même logique que celle de l'algorithme précédent, on prend la partie entière (en p) et le reste (en **q) de** la division de p (une copie de i au départ) par nn. La partie entière indique combien de traversées complètes ont été effectuées par nn. Le reste indique le nombre d'avancées déjà effectuées par nn dans la traversée actuelle. Les index i vont de 0 **à fact(n)-1** pour faciliter les calculs.

La valeur de q permet de calculer la position de l'objet nn dans perm. L'objet nn a déjà avancé q fois. En commençant à droite, l'algorithme passe sur q positions vides dans perm. L'objet nn va occuper la prochaine position vide. La boucle est répétée pour chaque objet n, n-l, n-2, ... 1. La nouvelle valeur de p est l'index de la permutation dans le monde des nn-1 .

http://dzdelphi.blogspot.com/

Ce nouvel algorithme a un avantage important par rapport au précédent : il calcule chaque permutation à partir des valeurs de i et. de n, sans consulter les permutations précédentes. Il est donc capable de calculer la permutation i sans engendrer ses prédécesseurs. Cette possibilité sert souvent, car de nombreuses applications nécessitent de tirer au sort une permutation. Après le tirage au sort d'un entier aléatoire entre 1 et factoriel(n), l'algorithme peut engendrer la permutation correspondante.

5.2.5 *Une version récursive*

Ce n'est pas par goût de la difficulté que les programmes pour les permutations présentés en premier lieu nécessitent une certaine réflection, mais justement pour apprendre à raisonner. Evidemment, on peut travailler directement, en écrivant une procédure récursive de manière "naturelle" (programme 5.7).

```
VAR pos: TAB [0..max-1];
PROC perm(i, n):
    VAR p, q, j: entier;
    SI n=l
    ALORS pos[1]:=1
    SINON p:=entier(i/n); q:=reste(i/n); perm(p, n-l);
        POUR j:=n-1 PAR -1 JUSQUA q+1 FAIRE pos[j]:=pos[j-1] FAIT;
        pos[q]:=n
    FINSI
FINPROC
```

Programme *5.7. Version récursive*

Dans cette procédure, i est l'index de la permutation à engendrer, 0≤i<factoriel(n-1). n est le nombre d'éléments à placer dans pos[0..n-1]. La numérotation à partir de 0 facilite l'utilisation du système de partie entière et reste, déjà étudié.

Ainsi, s'il existe plusieurs éléments à placer (n>1), on place n-l Cléments par l'appel récursif, puis on insère le n-ibme à sa place en décalant les q derniers éléments chacun d'une case vers la droite. Dans le monde des n-l, l'index de la permutation concernée est p (entier(i/n)).

Cet algorithme, appelé à répétition pour les valeurs de i de 0 à factoriel(n)-1, engendre les permutations dans l'ordre alphabétique. Cet ordre, classique, correspond

à la définition suivante : Les éléments dans chaque permutation sont considérés comme des caractères (numériques) d'un mot. Ces mots sont organisés en ordre alphabétique (ordre numérique si l'on considère les Cléments comme les chiffres d'un nombre).

5.3. Exercices

Sur les tours d'Hanoi

1. En supposant que les moines de Hanoi veuillent transférer le jeu complet de 64 disques, en prenant une seconde pour le déplacement individuel d'un disque, combien leur faudrait-il de temps ? La légende veut que la terminaison du transfert signale la fin du monde.

2. Etablir une courbe de complexité de l'algorithme des tours d'Hanoi en mesurant le temps nécessaire au calcul pour différents nombres de disques. L'exercice sert à faire prendre conscience des problèmes d'explosion combinatoire.

3. Avec 64 disques, et en numérotant les disques d_1 (le plus petit), d_2, ... d_{64} (le plus gros), dans quel ordre d_2 parcourt-il les piquets et, en général, quelle est la règle pour d_i ?

4. Considérons les index des déplacements simples i_1 (le premier coup), i_2, . . . Quel est le numéro du disque qui bouge au coup i_j ?

Sur les permutations

5. La possibilité de tirer au sort une permutation est utile dans de nombreuses applications. On reprendra la méthode de génération des permutations par échange de voisins (§5.2.2), en la modifiant afin de pouvoir calculer la permutation numéro i sans en engendrer les i-l précédentes.

6. Ecrire un programme qui engendre les permutations dans l'ordre alphabétique (la valeur numérique de l'entier produit en concatenant, dans l'ordre, les objets de la permutation i, est plus petite que celle de la permutation i+l) sans utiliser une procédure récursive. Le programme possédera la propriété de la question précédente (engendrer une permutation sans engendrer ses prédécesseurs).

7. On considère les nombres pouvant être formés à partir des chiffres 1, 2, ... 9, chaque nombre comportant une et une seule occurrence de chaque chiffre. En

http://dzdelphi.blogspot.com/

organisant ces nombres dans l'ordre numérique, quelle est la valeur du 100.000ème nombre ? (Concours de jeux mathématiques de France, 1986).

8. Considérons l'ensemble de valeurs duales des permutations du §5.2.2. (vecteur inv). Nous l'avons utilisé pour faciliter la recherche de la position courante de chaque objet. Est ce que cet ensemble forme un nouvel ordonnancement des permutations de n objets ?

9. Un théorème bien connu des mathématiciens est que toute permutation de n objets est un produit unique de cycles. On démontrera le théorème en l'accompagnant d'un programme qui identifie les cycles concernés pour une permutation donnée.

10. Le code de Gray est bien connu dans le monde de la transmission d'informations. Il est défini de la manière suivante :

On considère l'ensemble de caractères formés d'un nombre fixe de bits. Comme d'habitude, n bits permettent de disposer de 2" caracti?res. Dans le code de Gray, deux caractères qui se suivent ne diffèrent que par la valeur d'un seul bit.

On écrira un programme qui engendre les caractères successifs du code de Gray sur n bits, sachant que le bit qui change de valeur à chaque fois est celui du poids le plus faible possible. Par exemple, pour n=3, on aura l'ordre suivant :

```
000
001
011
010
110
111
101
100
```


Chapitre 6

La marche arrière

Dans ce chapitre nous abordons une nouvelle catégorie d'algorithmes : ceux qui mettent en jeu la *marche* arrière ("backtracking"). Il s'agit de résoudre des problèmes non déterministes. Un problbme qui n'est pas déterministe ne peut pas être résolu de manière directe. Considérons la situation de quelqu'un qui arrive à un carrefour dans un labyrinthe. Il ne sait pas si la bonne décision est de tourner à gauche ou à droite. La solution est d'essayer une des possibilités ouvertes, en la poursuivant aussi longtemps que possible. Si la décision mène à un blocage, il faut revenir sur ses pas afin d'essayer une autre possibilité. C'est cette idée de revenir sur ses pas qui donne lieu à l'expression "marche arrière".

6.1. La souris et le fromage

Nous considérons comme application de la marche arrière un problème de labyrinthe artificiel. Sur un échiquier classique (8 cases par 8), il y a un morceau de fromage que veut manger une souris. Les coordonnées des cases qu'ils occupent sont [xf, yf] et [xs, ys]. Le problème est d'établir un chemin que peut prendre la souris pour aller jusqu'au fromage, sachant qu'elle ne peut avancer que d'une case à la fois, soit horizontalement, soit verticalement, et qu'il existe des barrières infranchissables entre certaines cases voisines. Le problème est ancien, mais la première utilisation ayant un rapport avec l'informatique que nous connaissons est [Zemanek 1971]. Il a été utilisé par un des auteurs comme exercice de programmation depuis une école d'été en 1972 et a servi comme support dans un cours d'intelligence artificielle [Griffiths 1986, 19871.

http://dzdelphi.blogspot.com/

Nous allons donc écrire un programme mettant en œuvre la marche arrière pour la recherche d'un chemin, sans essayer de trouver le chemin le plus court. Le programme présente les caractéristiques suivantes :

- A chaque tour de la boucle principale, il considère, s'il en existe, un pas en avant, sinon il en fait un en arrière.
- Le pas en avant est dans la première direction disponible (qui n'a pas encore été essayée) à partir de la case courante.
- L'ordre de disponibilité est : nord, est, sud, ouest, qui sont représentes par les entiers 1, 2, 3, 4.
- En arrivant sur une case, le programme la marque. Les marques ne sont jamais enlevées. La souris ne repasse jamais en marche avant dans une case déjà marquée, car l'algorithme essaie toutes les possibilités à partir de chaque case visitée, et il ne faut pas qu'il boucle.
- Faire marche arrière consiste en un retour vers le prédécesseur de la case courante, suivi d'un essai dans la prochaine direction disponible à partir de cette case (s'il en existe).
- La marche arrière résulte du constat que l'on a essayé les quatre directions à partir d'une case sans trouver un chemin.
- Pour faire un pas en arrière, il faut savoir d'où on est venu. Ainsi, à chaque décision, la direction prise est sauvegardée sur une pile. A chaque retour en arrière, on revient sur la décision la plus récente, qui est en tête de la pile. Si la situation est toujours bloquée, on prends encore le prédécesseur, et ainsi de suite.

Le programme 6.1 est donné en entier pour faciliter la présentation, les commentaires étant regroupés à la fin.


```
1  DEBUT DONNEES barrière: TAB [1..8, 1..8, 1..4] DE bool;
2        xs, ys, xf, yf: entier;
3     VAR marque: TAB [1..8, 1..8] DE bool INIT FAUX;
4        pile: TAB [1..64] DE entier;
5        x, y, xn, yn, dir, pl: entier;
6     x:=xs; y:=ys; marque[x, y]:=VRAI; dir:=0; pl:=1;
7     TANTQUE NON (x=xf ET y=yf) ET pl>0
8     FAIRE SI dir<4
9        ALORS dir:= dir+l ;
10            SI NON barrière[x, y, dir]
11            ALORS xn:= CAS dir DANS (x, x+1, x, x-l) FINCAS;
12                 yn:= CAS dir DANS (y+1 , y, y-l, y) FINCAS;
13                 SI NON marque[xn, yn]
14                 ALORS x:=xn; y:=yn; marque[x, y]:=VRAI;
15                      pile[pl]:=dir; pl:=pl+1 ; dir:=0
                   FINSI
               FINSI
16       SINON pl:=pl-1;
17            SI pl>0
18            ALORS dir:=pile[pl];
19                 x:= CAS dir DANS (x, x-l, x, x+1) FINCAS;
20                 y:= CAS dir DANS (y-l, y, y+1 , y) FINCAS
               FINSI
         FINSI
      FAIT
   FIN
```

Programme 6.1. *La souris et le fromage*

Les numéros de ligne servent aux commentaires.

1. barrière[i, j, dir] est VRAI si et seulement s'il y a une barrière entre la case [i, j] et sa voisine en direction dir. Le tableau est initialisé ailleurs. Le bord de l'échiquier est entoure de barrieres pour éviter de faire référence à des cases inexistantes.

2. [xs, ys] est la position de départ de la souris, [xf, yf] celle du fromage.

3. marque[i, j] est VRAI si et seulement si la souris a déjà visité la case [i, j]. Ce tableau est initialisé à FAUX.

4. Une pile de 64 Cléments suffit.

5. [x, y] est la position courante de la souris. [xn, yn] est une position visée par la souris. dir représente le nombre de directions que la souris a déjà essayées à partir de la case [x, y].

http://dzdelphi.blogspot.com/

6. L'initialisation met [x, y] à la position de départ [xs, ys], qui est marquée. Aucune direction n'a été essayée. La pile est vide.

7. On boucle tant que le fromage n'est pas atteint et qu'il existe des chemins à essayer.

8. Existe-t-il encore une direction à essayer ?

9. Alors on prend la prochaine.

10. Si la route n'est pas barrée, on calcule les coordonnées [xn, yn] de la case voisine en direction dir.

11,12. dir=1 veut dire vers le nord, c'est-a-dire augmenter y de 1, et ainsi de suite.

13. Dans l'absence de marque sur [xn, yn], on peut y aller. Si la case est marquée, on essaiera la prochaine valeur de dir au prochain tour de la boucle.

14. Le mouvement a lieu, avec marquage.

15. La décision est enregistrée sur la pile. Pour la nouvelle case le nombre de directions déjà essayées est 0.

16. Comme il n'existe plus de directions à essayer, il faut faire marche arrière.

17. Si la pile n'est pas vide, on prend la direction prise pour arriver sur la case courante, qui se trouve en haut de la pile (18). pl=0 arrive quand tous les chemins possibles ont été essayés à partir de [xs, ys] et le fromage n'a pas été atteint. Il n'y a donc pas de chemin possible.

19, 20. On calcule, en inversant les lignes 11 et 12, les coordonnées du prédécesseur, et le programme fait un nouveau tour de la boucle.

A la fin du programme, si un chemin existe, la pile contient toutes les décisions prises sur lesquelles l'algorithme n'est pas revenu. Ainsi la pile contient une solution du probleme. Il n'est pas intéressant de programmer ici une boucle d'impression, mais le principe de la disponibilité des résultats acquis est valable pour les algorithmes de marche arrière en général. Pendant l'exécution du programme, la pile contient à chaque moment les décisions en cours d'essai.

6.1.1. *Version récursive*

Dans le cas d'une recherche complète sur toutes les possibilités, la marche arrière est l'équivalent d'un parcours d'arbre ou de graphe (voir §6.1.2). Dans le probleme considéré ici, nous sommes en présence d'un graphe, car il serait possible de revenir sur la case que l'algorithme vient de quitter, et l'on pourrait boucler. Cela est une autre explication du besoin de marquage, pour empêcher le programme de suivre les boucles. Comme dans tous les algorithmes de ce type, on peut écrire le programme dans la forme d'une procédure récursive (programme 6.2).


```
DEBUT DONNEES xs, ys, xf, yf: entier;
    barré: TAB [1..8, 1..8, 1..4] DE bool;
  VAR marqué: TAB [1..8, 1..8] DE bool INIT FAUX;
    existe_chemin: bool;
  PROC joindre(x, y) -> bool;
    VAR trouve: bool;
                dir: entier;
    trouvé:= x=xf ET y=yf;
    SI NON marqué[x, y]
    ALORS marqué[x, y]:=VRAI; dir:=0;
      TANTQUE NON trouvé ET dir<4
      FAIRE dir:=dir+l ;
        SI NON barré[x, y, dir]
        ALORS trouvé:=joindre(CAS dir DANS x, x+1, x, x-l FINCAS,
                              CAS dir DANS y+1 , y, y-l, y FINCAS)
        FINSI
      FAIT;
      St trouvé ALORS imprimer(x, y) FINSI
    FINSI;
    RETOURNER trouvé
  FINPROC;
  existe_chemin:=joindre(xs, ys)
FIN
```

Programme 6.2. *Forme récursive*

Le lecteur ayant tout suivi jusqu'ici n'aura pas de probleme pour comprendre ce programme.

6.1.2. *Marche arrière, arbres et graphes*

La marche arrière est un moyen d'explorer systématiquement un arbre de possibilités. Evidemment, l'exploration systématique ne termine que si le nombre de possibilités est fini. Il faut aussi noter l'existence de l'explosion combinatoire, où le nombre de possibilités, bien que fini, est tellement grand que l'exploration est en dehors des capacités des ordinateurs.

La figure 6.1 montre quelques branches de l'arbre de possibilités pour la souris. Commençant sur la case [xs, ys], elle a quatre directions disponibles (en ignorant des barrières et les limites de l'échiquier). Considérons la direction "est", menant à la case [xs+1, ys]. A partir de cette nouvelle case, quatre directions sont encore imaginables, dressées sur la figure.

http://dzdelphi.blogspot.com/

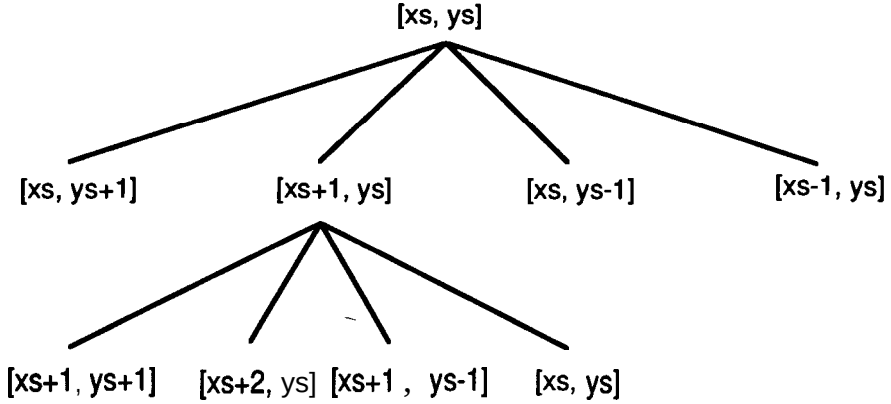

Figure 6.1. *Arbre de possibilités pour la souris*

Or, une de ces directions mbne la souris de retour à sa case de départ. Si l'on continuait à dresser un arbre, il serait de taille infinie, car on recommence indéfiniment avec la même case. Pour représenter le labyrinthe par une structure formelle, il faut passer à un graphe, avec un nœud par case. Les arcs sont toujours bidirectionnels, reliant des cases voisines qui ne sont pas séparées par des barrières. Notons que ce n'est pas seulement le fait de pouvoir retourner directement vers la case précédente qui impose un graphe; on peut aussi arriver, par exemple, à la case [xs+1, ys+1] en deux pas de deux façons différentes (par [xs, ys+1] et par [xs+1, ys]).

Le marquage des cases du labyrinthe correspond donc exactement au marquage des nœuds dans le parcours d'un graphe. Dans le cas du labyrinthe, le graphe est le graphe de connectivité des cases, représentées par des nœuds. Le programme 6.2 a la forme d'un parcours de graphe n-aire.

6.2. Les huit reines

Il s'agit encore d'un exercice archi-classique, mais sa présence ici est toujours justifiée par des raisons pédagogiques. Il faut placer huit reines sur un échiquier de telle façon qu'aucune reine ne soit en prise par rapport à une autre. Deux reines sont mutuellement en prise si elles se situent sur la même ligne, la même colonne ou la même diagonale. Le problème est ancien [Bezzel 18481 [Gauss 1850], mais il a été remis au goût du jour avec l'arrivée des ordinateurs. De nombreux articles modernes lui sont consacrés, jusqu'à dans des revues populaires [Paclet 19841. Il y a 92 solutions, dont évidemment un grand nombre dues à la symétrie.

Dans le programme 6.3, np est un entier qui indique le nombre de reines actuellement placées. En marche avant, le programme considère la prochaine case [x, y]. Le test x>8 au début de la boucle sert à passer à la ligne suivante (y:=y+1) en cas de besoin. Si la case [x, y] n'est pas couverte par une dame déjà placée, on peut y placer une nouvelle dame. Si la case n'est pas libre, la prochaine case sera considérée au prochain tour de la boucle (x:=x+1 à la dernière ligne).

```
DEBUT VAR np: entier INIT 0; % Le nombre de reines placées %
  x, y: entier;      % coordonnées d'une case à essayer %
  x:=1; y:=1;
  TANTQUE np c 8
  FAIRE SI x > 8 ALORS x:=1 ; y:=y+1 FINSI;
    SI y < 9
    ALORS SI libre(x, y)
      ALORS np:=np+I ; empiler(x, y);
      FINSI
    SINON dépiler(x, y); np:=np-1
    FINSI;
    x:=x+1
  FAIT
FIN
```

Programme 6.3. *Les huit reines*

L'indication de la nécessité de faire marche arrière est qu'il ne reste plus de case à essayer (y>8). Dans ce cas, on dépile x et y pour revenir dans l'état précédent, en diminuant le compteur np. Le prochain tour de la boucle considérera la prochaine case. Le processus s'arrête quand np arrive à 8, c'est-Mire quand solution est trouvée. Comme nous savons qu'une telle solution existe, il n'est pas nécessaire de considérer le cas où la marche arrière ne peut pas avoir lieu.

Dans ce programme, la pile contient les coordonnées des reines placées. Comme il y a deux valeurs à empiler (x et y), il peut devenir intéressant de définir deux piles, px et py. Dans ce cas, notons que np peut servir pour indiquer la hauteur de chaque pile. Par ailleurs, il nous faut remplacer l'appel de la procédure libre par une boucle qui décide si la case (x,y) est couverte par une reine déjà placée (même ligne, colonne ou diagonale). On obtient le programme 6.4.

http://dzdelphi.blogspot.com/

```
DEBUT VAR px, py: TAB [1..8] DE entier;
      np: entier INIT 0;
      x, y: entier INIT 1;
      libre:  bool;
      i:  entier;
   TANTQUE np < 8
   FAIRE SI x > 8 ALORS  x:=1 ; y:=y+1 FINSI;
      SI y < 9
      ALORS libre:=VRAI; i:=0;
         TANTQUE libre ET i<np
         FAIRE  i:=i+l  ;
            libre:=NON(x=px[i] OU y=py[i] OU abs(x-px[i])=abs(y-py[i]))
         FAIT;
         SI libre
         ALORS  np:=np+l  ;  px[np]:=x;  py[np]:=y
         FINSI
      SINON  x:=px[np];  y:=py[np];  np:=np-1
      FINSI;
      x:=x+1
   FAIT
FIN
```

Programme *6.4. Les huit reines, version 2*

6.2.1. *Une version améliorée*

La solution ci-dessus marche, mais on peut l'améliorer. Profitons du fait que nous savons qu'il y a une et une seule reine par colonne. Il n'est plus nécessaire de garder une pile des x (programme 6.5).


```
DEBUT VAR py: TAB [1..8] DE entier;
      x, y: entier INIT 1;
      libre:  bool;
      i:  entier;
   TANTQUE x < 9
   FAIRE SI y < 9
      ALORS libre:=VRAI; i:=l  ;
         TANTQUE libre ET i<x
         FAIRE libre:= NON( y=py[i] OU abs(x-i)=abs(y-py[i]))
            i:=i+1
         FAIT;
         SI libre ALORS py[x]:=y; x:=x+1; y:=0 FINSI
      SINON x:=x-l  ; y:=py[x];
      FINSI;
      y:=y+l
   FAIT
FIN
```

Programme *6.5. Suppression d'une pile*

Dans cette version du programme, la variable np disparaît (sa valeur est la même que celle de x-l). La variable x est la coordonnée en x de la nouvelle reine à placer, y étant l'autre coordonnée de la prochaine case à essayer. La marche arrière devient nécessaire si y arrive à 9, c'est-à-dire s'il n'y a plus de cases à essayer pour la valeur courante de x. Le test libre ne comporte plus de test sur x, car les x sont tous différents par construction, x servant également de pointeur dans la pile des y (il indique la première place libre dans cette pile).

6.2.2. *Une deuxième approche*

La dernière version ci-dessus ressemble un peu à un compteur octal, qui égrène tous les nombres en octal de 0 à 77777777. Pour faciliter la programmation, nous avons numéroté les cases de l'échiquier de 0 à 7 au lieu de 1 à 8. Considérons le programme 6.6.

http://dzdelphi.blogspot.com/

```
DEBUT VAR comp: TAB [0..7] DE entier INIT 0;
      VAR couvert: bool INIT VRAI;
          i, j: entier;
      TANTQUE couvert
      FAIRE % avancer le compteur %
          i:=7;
          TANTQUE comp[i]=7 FAIRE comp[i]:=0; i:=i-1 FAIT;
          comp[i]:=comp[i]+1;
      % tester la position créée %
          i:=1 ; couvert:=FAUX;
          TANTQUE NON couvert ET i<8
          FAIRE j:=0;
                TANTQUE NON couvert ET j<i
                FAIRE couvert:=comp[i]=comp[j]
                                    OU abs(i-comp[i])=abs(j-comp[j]);
                     j:=j+l
                FAIT;
                i:=i+l
          FAIT
      FAIT
FIN
```

Programme ***6.6. Avec un compteur octal***

Dans ce programme, le compteur (comp) est initialisé à la valeur zéro. Le booléen couvert indique que deux reines "se voient". Son initialisation à VRAI est une évidence : dans la position de départ, toutes les reines "se voient". La boucle externe tourne tant que la dernière position examinée n'est pas la bonne (aucune paire de reines en prise mutuelle).

Le contenu de la boucle est en deux parties : l'avancement du compteur et l'examen de la nouvelle position créée. Les 7 à droite du compteur deviennent des 0, et le prochain chiffre à gauche augmente de 1. Puis, pour chaque reine i (1...7), on teste si la reine j (0...i-1) est compatible. Les tests s'arrêtent à la première incompatibilité.

Le nombre de positions à essayer avant de découvrir toutes les solutions est de 8^8, c'est-à-dire 2^{24}, ou 16 Méga dans le sens de l'informatique (1 Méga = 1024*1024). Considérer un peu plus de 16 millions de cas prend un certain temps. Notons que la marche arrière diminue considérablement ce chiffre. En effet, les premières positions essayées dans l'algorithme du compteur sont les suivantes :


```
00000000
00000001
00000002
```

et ainsi de suite. Avec la marche arrière, on découvre immédiatement que tout nombre commençant par 0 0 . . . est à éliminer. La marche arrière donne un algorithme qui considère, dans l'ordre arithmétique :

```
00000000
01000000
02000000
02100000
```

et ainsi de suite. Compter le nombre de cas essayés est laissé comme exercice.

Cette ligne de réflexion pourrait nous amener à considérer, à la place du compteur octal, les permutations sur 8 objets. On sait en effet que tous les x, comme tous les y, doivent être différents. Le nombre de cas à considérer au total est de factoriel(8), c'est-à-dire 40320 (à comparer avec les plus de 16 millions pour le compteur octal). En faisant de la marche arrière sur les permutations, il deviendrait possible de considérer encore moins de cas.

Le programme de départ dénombre les permutations successives, en utilisant un des programmes du chapitre 5 (programme 6.7).

```
DEBUT VAR i: entier INIT 1, trouvé: bool INIT FAUX;
  TANTQUE NON trouvé
  FAIRE trouvé:=libre(perm(i))
    i:=i+1
  FAIT
FIN
```

Programme ***6.7. Avec despermutations***

En y ajoutant de la marche arrière, on pourrait arriver au programme 6.8.

http://dzdelphi.blogspot.com/

```
DEBUT VAR perm: TAB [1..8] DE entier INIT 0;
          disp: TAB [1..8] DE bool INIT VRAI;
          n, i: entier INIT 0;
     TANTQUE n<8
     FAIRE i:=i+l ;
          TANTQUE i<9 ETPUIS NON disp[i] FAIRE i:=i+l FAIT;
          SI i<9
          ALORS couvert:=FAUX; j:=O;
               TANTQUE j<n ETPUIS NON couvert
               FAIRE j:=j+1 ; couvert:= n+1-j = abs(i-perm[j])
               FAIT;
               SI NON couvert
               ALORS n:=n+l; perm[n]:=i; disp[i]:=FAUX; i:=O
               FINSI
          SINON i:=perm[n]; disp[i]:=VRAI; perm[n]:=0; n:=n-1
          FINSI
     FAIT
FIN
```

Programme 6.8. ***Avec permutations et marche arrière***

Dans ce programme, les permutations à essayer sont engendrees en perm. L'élément disp[i] du tableau booléen disp indique si l'objet i est disponible, c'est-à-dire s'il ne figure pas déjà dans la permutation : disp[i] est VRAI si i ne figure pas dans perm[1..n]. n indique le nombre de reines déjà placées. A chaque moment, la valeur de i est l'index du dernier objet essayé comme candidat à la place perm[n+l], qui est la prochaine reine à placer.

6.3. Exercices

Sur la souris et le fromage

1. Ecrire le programme qui ajoute l'heuristique, c'est-à-dire qui prend comme direction préférée celle du fromage.

2. Ecrire le programme de numérotation successive qui trouve un chemin de longueur minimale.

3. Il existe combien de chemins différents allant de la case [1, 11 à la case [8, 8] sur un échiquier vide ? Une même case ne peut pas figurer deux fois sur un même chemin. On écrira un programme qui compte le nombre de chemins.

Sur les huit reines

4. On modifiera le programme donné au §6.2.1 afin de trouver toutes les solutions.

5. Ecrire la version récursive du programme du §6.2.1.

6. Ecrire un programme qui compte le nombre de cas essayés en marche arrière sur le compteur octal (voir §6.2.2).

7. Faire la même chose pour le programme basé sur les permutations,

D'autres applications de la marche arrière

8. Extrait partiel du concours de jeux mathématiques, 1987.

On considère un triangle de cinq lignes :

```
x
xx
xxx
xxxx
xxxxx
```

Dans chacune des 15 positions, il faut placer un des 15 premiers entiers en respectant la condition suivante : l'entier dans chaque position, sauf dans la dernière ligne, a comme valeur la différence entre les valeurs de ses successeurs immédiats (on considère le triangle comme un treillis binaire). Un début de solution pourrait être :

```
 7
103
11 14
...
```

Chaque entier ne figure qu'une seule fois dans le triangle. On écrira un programme utilisant la marche arrière pour trouver une solution au problème. Combien y a-t-il de solutions en tout ?

http://dzdelphi.blogspot.com/

Chapitre 7

Transformation de programmes

La transformation de programmes, par des techniques automatiques ou manuelles, devrait intéresser l'informaticien professionnel pour plusieurs raisons :

- La transformation automatique de spécifications en programmes exécutables peut diminuer le coût de la production de programmes, tout en améliorant leur fiabilité.
- On peut découvrir des versions plus efficaces de certains programmes.
- En restant proche de la spécification d'un programme, on en facilite la portabilité.
- Le programmeur améliore sa maîtrise des outils, ce qui nous intéresse pour la pédagogie.

Le besoin d'industrialiser la production des logiciels a provoqué l'introduction de nombreux mots-clés nouveaux. Ces néologismes, tels que programmation structurée, génie logiciel, ... ne représentent pas seulement un phénomène de mode, mais recouvrent aussi une réalité technique. Afin de s'assurer de la qualité des produits industriels, on a souvent recours à la spécification formelle. Celle-ci sert à la démonstration éventuelle d'un programme, à sa validation, voire même à sa production. C'est pour ce dernier aspect que les techniques de transformation deviennent utiles. Une spécification est une formulation mathématique, surtout algébrique, d'un problème. Dans les méthodes de démonstration de programmes, on dispose d'une spécification et d'un programme, puis on démontre que le programme est une mise en œuvre de la spécification. Plutôt que démontrer l'équivalence de ces deux textes, on est arrivé naturellement à se poser la question de savoir s'il est possible de dériver le programme à partir de la spécification.

http://dzdelphi.blogspot.com/

Dans certains cas, la reformulation d'un programme peut s'avérer plus efficace que la version d'origine. Nous avons vu l'effet de certaines transformations dans le cadre de programmes récursifs dans un chapitre précédent.

La transformation d'un programme n'est possible que si le programme est écrit dans une forme propre. L'utilisation d'astuces dépendant d'un compilateur particulier est à proscrire. Notons que cela revient à dire les mêmes choses que l'on retrouve dans les ouvrages sur la portabilité des logiciels [Brown 19771.

Enfin, même si les techniques de transformation n'avaient aucune application pratique directe, nous les enseignerions quand même à nos étudiants. Effectivement, en examinant différentes versions d'un même programme, ils séparent mieux ce qui est intrinsèque au problème de ce qui dépend du langage, du compilateur, voire de l'ordinateur particulier qu'ils utilisent à un instant-donné. Les paragraphes suivants donnent quelques exemples de transformations. Pour aller plus loin, on peut consulter utilement des ouvrages tels que [Burstall 1975], [Darlington 1973].

7.1. Revenons sur le mode d'un vecteur

Ce problème a formé le sujet du premier vrai programme de cet ouvrage (§2.1). Il a été le sujet de plusieurs études [Griffiths 19761, [Arsac 1984], car on peut l'améliorer très simplement, cependant l'amélioration reste difficile à trouver. Nous avons découvert la modification en explorant les techniques de transformation. La version de départ de cet algorithme était celle du programme 7.1.

```
DEBUT DONNEES n: entier;
    v: TABLEAU [1 ..n] DE entier;
  VAR i, lcour, lmax, indm: entier INIT 1;
  TANTQUE i<n
  FAIRE i:=i+1 ;
    SI v[i]=v[i-1]
    ALORS lcour:=lcour+1;
      SI lcour>lmax
      ALORS lmax:=lcour; indm<-i
      FINSI
    SINON lcour:=1
    FINSI
  FAIT
FIN
```

Programme 7.1. *Encore le mode d'un vecteur*

C'est en essayant de rédiger la spécification formelle de l'algorithme que nous avons trouvé une amélioration. Essayons de dénombrer les cas qui se présentent. On considère que le mode d'un vecteur est un doublet : la longueur de la chaîne la plus longue et la valeur qui paraît dans cette chaîne. On obtient les cas suivants :

1. n-l: mode(v[l..n]) = (1, v[1])
2. n>1, v[n]≠v[n-1]: mode(v[l ..n]) = mode(v[l ..n-1])
3. n>1, v[n]=v[n-1], long(mode(v[1 ..n]))>long(mode(v[1 ..n-1])):
   mode(v[1 ..n]) = (long(mode(v[l ..n-1]))+1, v[n])
4. n>1, v[n]=v[n-1], long(mode(v[1 ..n]))≤long(mode(v[1 ..n-1])):
   mode(v[l..n]) = mode(v[l..n-1])

L'amélioration est venu en essayant de formaliser le test qui décide si la chaîne courante est plus longue que le mode trouvé jusqu'alors, c'est-à-dire le test suivant :

long(mode(v[l ..n]))>long(mode(v[1 ..n-1]))

Evidemment, on peut laisser le test sous cette forme, mais on cherche naturellement un test qui décide si la nouvelle chaîne est plus longue que l'ancienne. C'est en posant la question de cette manière que nous avons trouvé l'idée (pourtant simple) de considérer le test suivant : soit lm la longueur du mode de v[1..n-1]. Alors, la chaîne dont v[n] fait partie est plus longue que lm si et seulement si

v[n] = v[n-lm]

Dans la transformation de la spécification ci-dessus, on remplacera le test sur les longueurs par ce nouveau test. Par ailleurs, on note que les cas 2 et 4 donne la même solution (le mode est inchangé). Mais, si v[n]#v(n-1), il est certain que v[n]#v[n-hn]. On en déduit que la comparaison de v[n] avec son prédécesseur n'est pas nécessaire. La spécification devient :

1. n=l: mode(v[l..n])=(l, v[1])
3. n>1, v[n]=v[n-long(mode(v[1 ..n-1]))]:
   mode(v[l ..n]) = (long(mode(v[1 ..n-1]))+1, v[n])
4. n>1, v[n]≠v[n-long(mode(v[1..n-1]))]:
   mode(v(1 ..n)) = mode(v(1 ..n-1))

Le test v[n]=v[n-1] a disparu. En transformant cette spécification vers un programme itératif, on obtient le programme 7.2.

http://dzdelphi.blogspot.com/

```
DEBUT DONNEES n: entier,
        v: TAB [1 ..n] DE entier;
      VAR i, indm, lm: entier INIT 1;
      TANTQUE i<n
      FAIRE i:=i+1 ;
            SI v[i]=v[i-lm]          % regard en arrière
            ALORS indm:=i; lm:=lm+1   % nouveau max
            FINSI
      FAIT
FIN
```

Programme 7.2. *Le mode amélioré*

On a gagné une variable et un test par élément du tableau. Cette optimisation, bien que simple, n'est pas évidente au départ

7.2. La multiplication des gitans

L'idée de ce programme nous est venu de P.L. Bauer, qui l'a incorporé dans son cours de programmation [Bauer 19761. Elle vient de l'observation de la méthode de multiplication utilisée par des gens sur des marchés de l'Europe centrale. Montrons cette méthode sur un exemple, 25 * 40 :

```
25   40
12   80
 6  160
 3  320
 1  640
```

On a divisé les nombres de la colonne gauche par deux, en multipliant par deux ceux de la colonne de droite en même temps. L'opération est répétée autant de fois que possible, en laissant tomber les moitiés résultant de la division par deux d'un nombre impair. Par la suite, on supprime, dans la colonne de droite, les valeurs correspondant à un nombre pair dans la colonne de gauche. Les nombres restant à droite sont additionnés :

```
40 + 320 + 640 = 1000
```

Ce qui est le résultat escompté.

Les opérations ci-dessus ne sont pas le "tour de passe-passe" que l'on pourrait imaginer. En fait, 6 * 160 (deuxième ligne) est strictement égal à 3 * 320. Mais,

parce que 3 est un nombre impair, 3 * 320 = 1 * 640 + 1 * 320. Donc il faut garder les valeurs en face des nombres impairs. En écrivant ceci dans la forme d'une procédure, on obtient le programme 7.3.

```
mult(a,b):SI a=0
          ALORS 0
          SINON SI impair(a)
                ALORS b + mult(a-1, b)
                SINON mult(a/2, b*2)
                FINSI
          FINSI
```

Programme *7.3. Multiplication avec récursivité*

Pour enlever la récursivité, on introduit une variable res pour le cumul des valeurs retenues (programme 7.4).

```
multp(a,b,res):TANTQUE    a>0
        FAIRE SI impair(a)
          ALORS res:=res+b;  a:=a-1
          SINON a:=a/2; b:=b*2
          FINSI
        FAIT
```

Programme *7.4. Suppression de la récursivité*

Comme il s'agit dentiers, représentes sous forme binaire dans l'ordinateur, on peut simplifier en faisant appel à des décalages (decg pour décaler dune place vers la gauche, decd à droite). Par ailleurs, si a est impair, a-l est pair, donc le SINON peut disparaître (programme 7.5).

```
multp(a,b,res):TANTQUE    a>0
        FAIRE SI impair(a)
          ALORS res:=res+b;  a:=a-1
          FINSI;
          decd(a);  decg(b)
        FAIT
```

Programme *7.5. Version avec décalages*

http://dzdelphi.blogspot.com/

Etant donné que a est décalé à droite, a:=a-1 n'est pas nécessaire, car le 1 que l'on enlève est le dernier bit à droite, supprime de toute façon par le décalage qui va suivre. Cela nous permet de déduire l'algorithme de multiplication réellement mis en œuvre par les circuits dans les ordinateurs (figure 7.1).

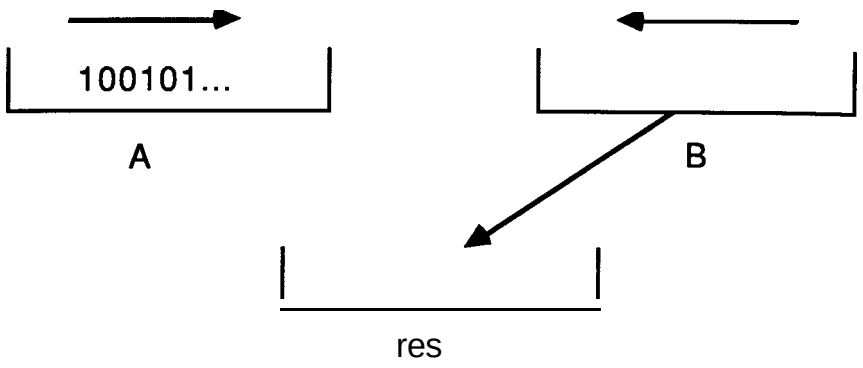

Figure 7.1. *Multiplication de deux entiers*

On décale a vers la droite jusqu'à l'arrivée d'un 1 dans le bit de droite. On décale d'autant b vers la gauche. On ajoute la nouvelle valeur de b dans le registre résultat. La triple opération est répétée pour chaque bit de a ayant la valeur 1.

L'introduction d'une variable permettant de supprimer une récursivité sans recourir à une pile est une technique habituelle. On le fait sans réfléchir dans beaucoup de cas. Considérons le programme bateau de calcul d'un factoriel :

```
fact(n): SI n=1
         ALORS 1
         SINON n . fact(n-1)
         FINSI
```

Aucun programmeur n'hésite à écrire la boucle suivante

```
fact:=1 ;
TANTQUE n>1 FAIRE fact:=fact * n; n:=n-1 FAIT
```

Mais il hésiterait beaucoup plus sur les fonctions suivantes :

```
divfact(n): SI n=1 ALORS 1 SINON n / divfact(n-1) FINSI

div2(n): SI n=1 ALORS 1 SINON div2(n-1) / n FINSI
```


On utilise des connaissances intuitives des propriétés telles que la commutativité, la distributivité, . . . des opérateurs. Une étude des conditions nécessaires et suffisantes pour permettre des transformations ne peut qu'améliorer notre maîtrise de l'outil de programmation.

Bauer signale que le programme de multiplication peut se généraliser. Le même schéma de programme permet la minimalisation du nombre de multiplications nécessaires pour le calcul d'une puissance entière (voir exercice en fin de chapitre).

7.3. Exercices

1. Utiliser le programme du 57.2 comme schéma pour le calcul de a^b.

2. Enlever la récursivité des procédures divfact et div2 du $7.2.

http://dzdelphi.blogspot.com/

Chapitre 8

Quelques structures de données particulières

Dans ce chapitre, nous introduisons plusieurs formes nouvelles d'arbres : les arbres ordonnés ("ordered trees"), les arbres équilibrés ("balanced trees") et les B-arbres ("B-trees")

8.1. Les arbres ordonnés

Pour qu'un arbre soit ordonné, il faut attacher une valeur à chaque nœud. Dans la pratique, comme par exemple dans des systèmes de gestion de fichiers ou de bases de données, cette valeur est souvent une clé, du type décrit pour l'adressage dispersé (voir chapitre 2) ou d'un style approchant. Ordonner un arbre revient à faire en sorte que, si la valeur attachée au nœud n est val[n], chaque nœud dans le sous-arbre dont la racine est le successeur gauche de n a une valeur associée vg tel que vg<val[n], les nœuds à droite possédant des valeurs supérieures. Cette définition implique qu'une valeur ne peut figurer qu'une seule fois dans l'arbre. La figure 8.1 montre un exemple d'arbre binaire ordonné (voir page suivante).

L'arbre de la figure aurait pu être produit par l'algorithme naturel de construction si les éléments considérés s'étaient présentés dans l'un des ordres suivants :

```
71042831
74211083
```

http://dzdelphi.blogspot.com/

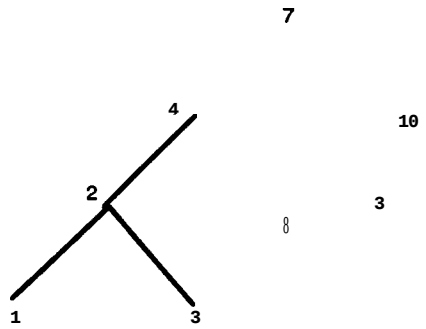

Figure 8.1. ***Un arbre binaire ordonné***

Ces ordres sont des ordres partiels de parcours de l'arbre, où l'ordre partiel doit respecter la règle suivante : un nœud devance, dans l'ordonnancement, ses successeurs.

L'algorithme naturel de construction de l'arbre est aussi celui qui sert à rechercher la présence d'un objet dans l'arbre (programme 8.1).

```
DEBUT DONNEES succg, succd, val: TABLEAU [1 ..taille] DE entier;
    v, racine, pi: entier;
  VAR n: entier;
  n:=racine;
  TANTQUE v≠val[n]
  FAIRE SI v<val[n]
    ALORS SI succg[n]>0
      ALORS  n:=succg[n]
      SINON succg[n]:=pi; n:=pi; pi:=pi+1;
        succg[n]:=0; succd[n]:=0; val[n]:=v
      FINSI
    SINON SI succd[n]>0
      ALORS  n:=succd[n]
      SINON succd[n]:=pi; n:=pi; pi:=pi+1;
        succg[n]:=0; succd[n]:=0; val[n]:=v
      FINSI
    FINSI
  FAIT
FIN
```

Programme 8.1. ***Arbre ordonné***

L'arbre défini par les trois vecteurs succg, succd et val est fourni en entrée, avec sa racine, la valeur v à trouver ou à placer et l'index pi de la première case libre dans chaque tableau. Si v est déjà dans l'arbre, n se positionne sur le nœud qui porte la valeur v, sinon on crée un nouveau nœud à l'endroit adéquat de l'arbre.

Avec un arbre ordonné, la recherche d'un objet se fait plus vite qu'avec les algorithmes précédents (§4.3.3), qui examinaient tous les nœuds de l'arbre. L'ordonnancement permet de suivre toujours le bon chemin, c'est-à-dire celui sur lequel est, ou sera, le nœud possédant la valeur recherchée. Le nombre de tests dépend ainsi de la profondeur de l'arbre.

8.2. Les arbres équilibrés

L'algorithme du paragraphe précédent améliore la vitesse de recherche d'un objet, mais le nombre de nœuds à examiner n'est pas encore minimisé. En effet, considérons l'arbre (figure 8.2) qui résulterait d'une arrivée d'objets dans l'ordre suivant :

12347810

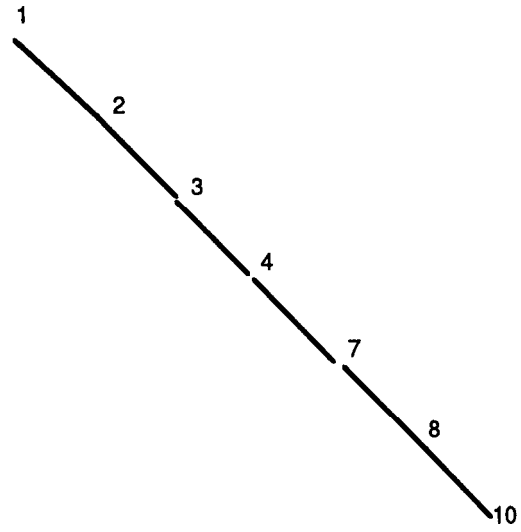

Figure 8.2. ***Arbre ordonné inefficace***

http://dzdelphi.blogspot.com/

Les recherches sont minimisées seulement si l'arbre est équilibré, ou du moins si la profondeur maximale de l'arbre est minimisée. La figure 8.3 montre l'arbre de l'exemple dans sa forme équilibrée.

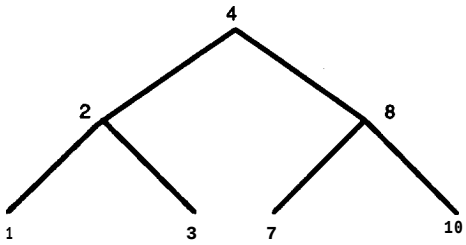

Figure *8.3. Equilibrage de l'arbre de la figure 8.1*

Soient ng le nombre de nœuds dans le sous-arbre gauche d'un nœud donné, nd le nombre de nœuds dans le sous-arbre droit. Alors un arbre est équilibré si et seulement si abs(ng-nd)<2. Si, pour tout nœud, ng=nd, l'arbre est strictement équilibré, comme celui de la figure 8.3. Cela ne peut arriver que si le nombre de nœuds le permet (nombre total de nœuds = 2^i - 1). On voit que l'algorithme de recherche dans un arbre ordonne et équilibré a une efficacité d'ordre $o(\log_2(n))$, tandis que l'algorithme classique est d'ordre o(n), avec n le nombre de noeuds.

Ajoutons maintenant deux nœuds, de valeurs 6 et 11, à l'arbre précédent. On obtient l'arbre de la figure 8.4.

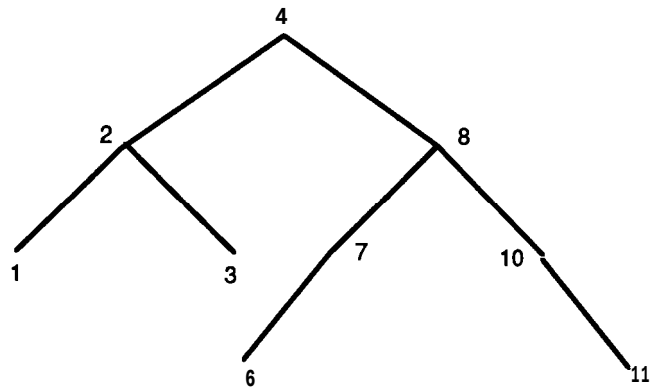

Figure 8.4. ***Avec deux nœuds en plus***

Cet arbre n'est plus équilibré, car le nœud de valeur 4 a 3 nœuds dans son sous-arbre gauche et 5 à droite. Pour le rééquilibrer, il y a plusieurs possibilités, dont celle de la figure 8.5.

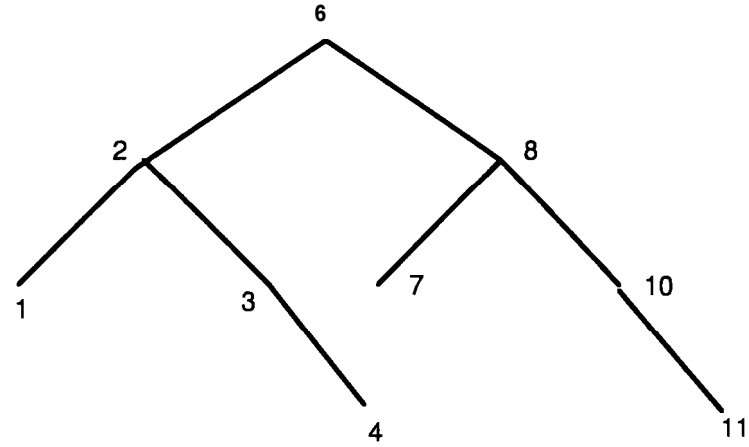

Figure *8.5. Rééquilibrage de l'arbre de la figure 8.4*

Notons que l'arbre de la figure 8.4 est en fait aussi bon que celui de la figure 8.5 en ce qui concerne l'efficacité des recherches, car leurs profondeurs sont les mêmes. On peut définir la profondeur d'un arbre comme étant la longueur du plus long chemin menant de la racine à une feuille. L'efficacité des recherches se mesure en calculant le nombre de tests nécessaires pour atteindre chaque objet de l'arbre, ou pour en placer un nouveau. C'est dans ce sens que les arbres des figures 8.4 et 8.5 sont d'efficacité égale (voir paragraphe ci-dessous sur les B-arbres).

8.2.1. Algorithmes de manipulation d'arbres équilibrés

La création d'un arbre équilibré à partir d'un ensemble d'objets ordonnés n'est pas difficile. Commençons avec un vecteur v qui contient les valeurs, dans l'ordre numérique. On peut considérer le programme 8.2.

http://dzdelphi.blogspot.com/

```
DEBUT DONNEES n: entier;
    v: TABLEAU [1 ..n] DE entier;
  VAR val, sg, sd: TABLEAU [1 ..n] DE entier;
    pl: entier INIT 1;
    racine:  entier;
  PROCEDURE arb(i, j: entier) RETOURNER entier;
    VAR x, k: entier;
    SI i=j
    ALORS val[pl]:=v[i]; sg[pl]:=0; sd[pl]:=0; pl:=pl+1;
      RETOURNER (pl-1)
    SINON SI j>i
      ALORS k:=(i+j)/2; x:=pl; pl:=pl+1; val[x]:=v[k];
        sg[x]:=arb(i,k-1);  sd[x]:=arb(k+1,j);
        RETOURNER x
      SINON RETOURNER 0
      FINSI
    FINSI
  FINPROC;
  racine:=arb(1,n)
FIN
```

Programme 8.2. ***Arbre équilibré***

La procédure arb procède par dichotomie, en retournant l'index de la racine de l'arbre qu'elle crée. Elle crée un arbre équilibré avec les valeurs **v[i]..v[j].** Si i=j, l'unique valeur est insérée dans un nouveau nœud, qui est en fait une feuille. Si **i<j,** l'index médian k est calculé. Un nœud est créé pour **v[k].** Les successeurs de ce nœud sont calcul& par des appels de arb avec les valeurs à gauche de k, puis les valeurs à droite. On retourne l'index du nœud créé. Enfin, si **i>j,** aucun élément n'est à insérer, donc la procédure retourne la valeur 0 (successeur vide).

Le problème se corse quand on dispose d'un arbre équilibré et que l'on veut y insérer un nouvel objet. L'opération est intrinsèquement coûteuse, les algorithmes directs étant particulièrement lourds.

8.3. Les B-arbres

La minimisation du nombre de nœuds à examiner pendant le parcours d'un arbre est particulièrement intéressante pour la recherche d'un fichier à travers le catalogue tenu par le système d'exploitation, ou pour les accès à des objets dans des bases de données. En effet, dans ces cas, chaque examen d'un nœud peut nécessiter

une lecture sur support externe (en général un disque). La multiplication des entrées-sorties est un facteur déterminant pour l'efficacité du système global, surtout dans un contexte conversationnel, où les temps de réponse ne doivent pas dépasser certaines limites. Ce paragraphe reprend des notions exposées dans [Miranda 19871, qui traite de tous les aspects des bases de données.

Les B-arbres [Bayer 19711, [Bayer 19721 apportent une solution efficace au problème, sans pour autant prétendre à une efficacité parfaite. Ils représentent un point d'équilibre, évitant les algorithmes coûteux de restructuration que nécessitent les arbres binaires équilibrés.

Un B-arbre n'est jamais binaire. Un nœud possède au plus **2*d+1** successeurs et **2*d** étiquettes, où d est la densité du B-arbre. Ce sont des arbres ordonnés et équilibrés. Dans le vocabulaire des B-arbres, le terme **"équilibré"** se définit de la manière suivante : la longueur d'un chemin de la racine jusqu'à une feuille est la même, quelque soit la feuille. Nous présentons ici les B-arbres les plus simples, avec d=l, c'est-à-dire que chaque nœud comporte au plus deux étiquettes (les valeurs du paragraphe précédent, appelées clés dans le langage des bases de données) et trois successeurs. La figure 8.6 montre un tel B-arbre, complet à deux niveaux (8 étiquettes pour 4 nœuds).

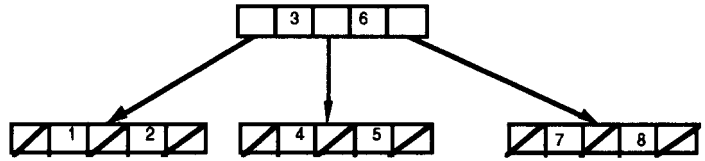

Figure ***8.6. Un B-arbre complet avec d=l***

Par ailleurs, on applique la règle qu'un B-arbre doit être rempli à au moins 50%, ce qui revient à dire, pour d=l, que chaque nœud comporte au moins une étiquette. Un nœud qui n'est pas une feuille a au moins deux successeurs. Pour comprendre la façon de construire un B-arbre, ou de rechercher un objet, prenons l'exemple de l'arrivée d'objets dans l'ordre le plus difficile, soit des objets comportant des étiquettes 1, 2, 3, . . . Les étiquettes 1 et 2 se logent dans la racine (figure 8.7).

Figure ***8.7. Après lecture des 1 et 2***

A la lecture du trois, il faut introduire un nouveau niveau, car la racine est pleine (figure 8.8).

http://dzdelphi.blogspot.com/

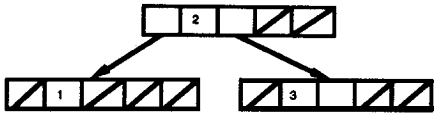

Figure 8.8. *Après lecture du 3*

Le 4 peut s'ajouter dans la feuille qui contient déjà le 3. Mais l'arrivée du 5 provoque à nouveau un bouleversement, car on voudrait l'ajouter dans ce même noeud. Comme il n'y a pas la place pour une troisième étiquette, le 5 provoque "l'éclatement" du nœud comportant le 3 et le 4. L'éclatement consiste en la "remontée" de l'étiquette intermédiaire (dans ce cas le 4), en créant deux nouveaux nœuds pour les étiquettes décentralisées (ici 3 et 5). Le résultat se trouve en figure 8.9.

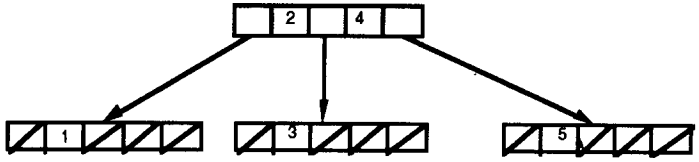

Figure *8.9. Après lecture du 5*

On voit que le 4 peut se loger au niveau au-dessus, car le nœud comportant l'étiquette 2 n'est pas plein. A son tour, le six se logera dans la case qui contient déjà le 5, car il reste un espace. L'arrivée du 7 va encore provoquer des éclatements (figure 8.10).

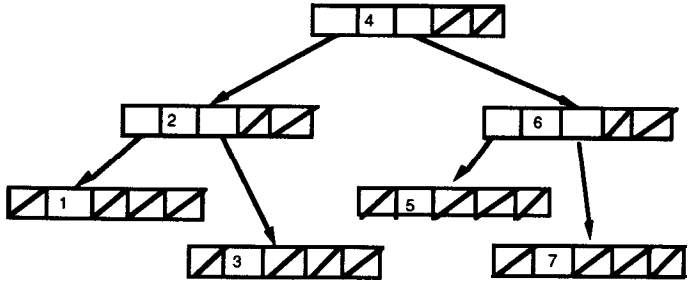

Figure 8.10. *Après lecture du 7*

Le 7 a fait éclater le nœud qui contenait 5 et 6. Il n'y avait pas de place pour le 6 dans le nœud au-dessus, qui contenait déjà 2 et 4. A son tour, ce nœud éclate, provoquant l'arrivée dune nouvelle racine. Cela augmente la profondeur de l'arbre.

Notons que la profondeur augmente toujours par l'adjonction d'une nouvelle racine. Cette propriété assure la continuité de la définition de B-arbre, où chaque feuille est à la même distance de la racine. Les noeuds sont également remplis à 50% dans le plus mauvais des cas, car chaque nœud créé reçoit toujours une étiquette. La figure 8.10 montre les limites de l'algorithme simple, car l'arbre comporte trois niveaux pour un ensemble d'étiquettes qui n'en nécessite que deux. Cela résulte du choix de données de départ, où nous avons considéré l'ordre le plus mauvais. Le résultat est néanmoins tout à fait acceptable, surtout en sachant qu'un ordre aléatoire d'arrivée des étiquettes mène à un bien meilleur taux de remplissage des nœuds.

Dans les bases de données, ou dans le stockage de fichiers en général, les Ctiquettes sont des clés dans le sens de l'adressage dispersée (voir chapitre 2). Sauf accident, ceci assure une distribution correcte de valeurs. Si l'efficacité était vraiment critique, on aurait pu éviter l'introduction d'un nouveau nœud dans la figure 8.10. Rappelons que l'étiquette 7 fait éclater la case qui contenait auparavant le 5 et le 6. La montée du 6 fait éclater la racine, qui comportait le 2 et le 4. La remontée du 4 crée une nouvelle racine. En fait, il y a une place pour le 4 à côte du 3, comme dans la figure 8.11.

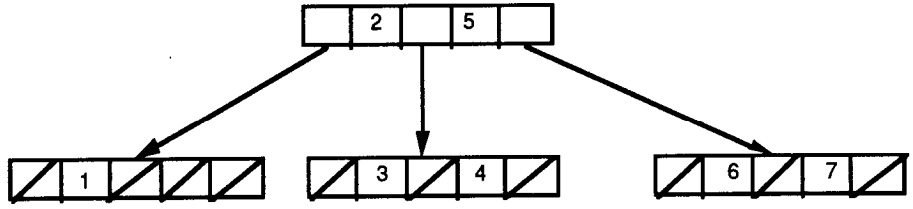

Figure 8.11. *Réorganisation pour éviter d'introduire un niveau*

Il a également fallu réorganiser le trio 5, 6, 7. On note que retrouver une organisation optimale est loin d'être trivial. En général, on accepte l'algorithme de base, qui est "raisonnable". Des évaluations des coûts des différentes opérations se trouvent en [Bayer 1972], [Knuth 19731, [Aho 19741.

Le programme 8.3 recherche une valeur dans un B-arbre, en l'introduisant dans l'arbre si elle n'y est pas encore.

http://dzdelphi.blogspot.com/

```
DONNEES sg, sc, sd, valg, vald, pred: TAB [1 ..taille] DE entier;
        v, racine: entier;
DEBUT VAR n. pos. orph, nn: entier;
          fini, trouve: bool;
      n:=racine; trouvé:= v=valg[n] OU v=vald[n];
      TANTQUE sg[n]>0 ET NON trouve
      FAIRE SI v<valg[n]
            ALORS n:=sg[n]
            SINON SI v>vald[n] ET vald[n]>0
                  ALORS n:=sd[n]
                  SINON n:=sc[n]
                  FINSI
            FINSI;
            trouvé:= v=valg[n] OU v=vald[n]
      FAIT;
      SI NON trouvé
      ALORS orph:=0; fini:=FAUX;
            TANTQUE NON fini
            FAIRE SI vald[n]=0
                  ALORS fini:=VRAI;
                        SI n=0
                        ALORS nn:=nouveau_nœud; sg[nn]:=racine; sc[nn]:=orph;
                              sd[nn]:=0; pred[nn]:=0; valg[nn]:=v; vald[nn]:=0;
                              pred[racine]:=nn; pred[orph]:=nn; racine:=nn
                        SINON SI v>valg[n]
                              ALORS vald[n]:=v; sd[n]:=orph
                              SINON vald[n]:=valg[n]; valg[n]:=v; sd[n]:=sc[n];
                                    sc[n]:=orph
                              FINSI
                        FINSI
                  SINON nn:=nouveau nœud;
                        SI v<valg[n]
                        ALORS échange(v,valg[n]); pos:=1; valg[nn]:=vald[n]
                        SINON SI v>vald[n] ET vald[n]>0
                              ALORS valg[nn]:=v; v:=vald[n]; pos:=2
                              SINON valg[nn]:=vald[n]; pos:=3
                              FINSI
                        FINSI;
                        vald[n]:=0; pred[nn]:=pred[n];
                        CAS pos
                        DANS 1: sc[nn]:=sd[n]; sg[nn]:=sc[n]; sc[n]:=orph,
                              2: sg[nn]:=sd[n]; sc[nn]:=orph,
                              3: sg[nn]:=orph; sc[nn]:=sd[n]
                        FINCAS;
                        sd[n]:=0; sd[nn]:=0; n:=pred[n]; orph:=nn
                  FINSI
            FAIT
      FINSI
FIN
```

Programme ***8.3.*** ***B-arbres***

La complexité de cet algorithme montre le travail à faire pour mettre en route ce genre de technique. Le programme qui restructure un B-arbre pour lui donner une représentation optimale est encore plus difficile. Un traitement complet de ces problèmes dépasse la portée de cet ouvrage.

8.4. Exercices

1. Ecrire un programme qui décide si un arbre binaire donné est ordonné.

2. Ecrire un programme qui décide si un arbre binaire donné est équilibré.

http://dzdelphi.blogspot.com/

Bibliographie et références

CACM : Communications of the Association for Computing Machinery
JACM : Journal of the Association for Computing Machinery
LNCS : Lecture Notes in Computer Science

AHO A.V., HOPCROFT J.E., ULLMANN J.D., ***The design and analysis of computer algorithms,*** Addison Wesley, 1974.

AHO A.V., HOPCROFT J.E., ULLMANN J.D., ***Data structures and algorithms***, Addison Wesley, 1983.

ARSAC J., ***Premières leçons de programmation,*** CEDIC/Fernand Nathan, 1980.

ARSAC J., ***Les bases de la programmation,*** Dunod, 1983.

ARSAC J., "La conception des programmes", ***Pour la science, numéro spécial 85,*** novembre 1984, (édition ***française*** de ***Scientific American).***

BAUER EL., EICKEL J., (eds.), ***Advanced course in compiler construction,*** Springer Verlag, LNCS 21, 1974.

BAUER F.L., (ed.), ***Software engineering,*** Springer Verlag, LNCS 30, 1975.

BAUER EL., Samelson K., ***(eds.), Language hierarchies and interfaces,*** Springer Verlag, LNCS 46, 1976.

BAUER EL., BROY M., (eds.), ***Program construction,*** Springer Verlag, LNCS 69, 1979.

BAYER R., "Binary B-trees for virtual memory", *Proc. ACM-SIGFIDET Workshop, New* York, 197 1.

BAYER R., Mc GREIGHT E., "Organisation and maintenance of large ordered indexes", *Acta Informatica 1, 3*, p. 173-189, 1972.

BERLIOUX P., B IZARD Ph., ***Algorithmique : construction, preuve et évaluation des programmes,*** Dunod, 1983.

BEZERT D., MONET P., "Les tours d'Hanoi, ou comment changer de disques", OI, janvier 1985.

http://dzdelphi.blogspot.com/

BEZZEL *Max, Schach zeitung*, 1848.

BOUSSARD J.-C., MAHL R., ***Programmation avancée : algorithmique et structures de données***, Eyrolles, 1983.

BROWN P.J., (ed.), ***Software portability***, Cambridge university press, 1977.

BURSTALL R.M., DARLINGTON J., "Some transformations for developping recursive programs", ***SIGPLAN notices*** 10, 6, juin 1975.

CARROLL L., ***Symbolic logic,*** part 1 : elementary, Macmillan 1896, traduit en français par GATTEGNO J. et COUMET E., ***La logique sans peine,*** Hermann, 1966.

COURTIN J., KOWARSKI I., ***Initiation à l'algorithmique et aux structures de données,*** 1 - programmation structurée et structures de données élémentaires, 2 - récursivité et structures de données avancées, Dunod, 1987.

CUNIN P.Y., GRIFFITHS M., SCHOLL P.C., "Construction méthodique et vérification systématique de programmes : éléments d'un langage", ***Congrès AFCET,*** novembre 1978.

CUNIN P.Y., GRIFFITHS M., VOIRON J., ***Comprendre la compilation,*** Springer Verlag, 1980.

DARLINGTON J., BURSTALL R.D., "A system which automatically improves programs", ***3rd. conf. on artificial intelligence,*** Stanford, août 1973.

DIJKSTRA E.W., ***A discipline of programming,*** Prentice Hall, 1976.

FEIGENBAUM E.A., FELDMAN J., (eds.), ***Computers and thought,*** McGraw Hill, 1963.

GAUSS Carl, 1850, (lettre du 2/9).

GERBIER A., (nom collectif), ***Mes premières constructions de programmes,*** Springer Verlag, LNCS 55, 1977.

GIANGRASSO V.M., GUZZI T., ***Projet DU-II, Université Aix-Marseille III,*** 1989.

GREGOIRE, (nom collectif), ***Informatique-programmation,*** tome 1 : la programmation structurée, tome 2 : la spécification récursive et l'analyse des algorithmes, Masson, 1986, 1988.

GRIES D., ***The Schorr-Waite marking algorithm,*** dans [Bauer 1979].

GRIES D., ***The science of programming,*** Springer Verlag, 1981.

GRIFFITHS M., Analyse déterministe et compilateurs, thèse, Grenoble, octobre 1969.

GRIFFITHS M., ***Program production by successive transformation, dans*** [Bauer 1976].

GRIFFITHS M., ***Development of the Schorr-Waite algorithm,*** dans [Bauer 1979].

GRIFFITHS M., ***Méthodes algorithmiques pour l'intelligence artificielle,*** Hermès, 1986.

GRIFFITHS M., PALISSER C., ***Algorithmic methods for artificial intelligence,*** Hermès, 1987.

GRIFFITHS M., VAYSSADE M., ***Architecture des systèmes d'exploitation,*** Hermès, 1988.

HEROUX D., "Antiquité binaire : la multiplication égyptienne", ***OI n° 54 bis,*** décembre 1983.

J&S, "Qui est mécanicien ?", ***Jeux et stratégie*** 26, p. 84, avril-mai 1984.

KNUTH D.E., "The art of computer programming", ***vol. 3 : Sorting and searching,*** Addison Wesley 1973.

LEDGARD H.F., ***Programmingproverbs,*** Hayden, 1975, traduit en français par ARSAC J., ***Proverbes de programmation,*** Dunod, 1978.

LUCAS M., PEYRIN J.-P., SCHOLL P.C., ***Algorithmique et représentation des*** *données,* ***1 - Files, automates d'états finis,*** Masson, 1983a.

LUCAS M., ***Algorithmique et représentation des données, 2 - Evaluations, arbres, graphes, analyse de textes,*** Masson, 1983b.

Mc CARTHY J., "Recursive computations of symbolic expressions and their computation by machine", part 1, ***CACM 3, 4,*** avril 1960.

MEYER B., BAUDOIN C., ***Méthodes de programmation,*** Eyrolles, 1978.

MIRANDA S., BUSTA J.-M., ***L'art des bases de données, Tome 1 : Introduction aux bases de données,*** Eyrolles 1987.

OXUSOFF L., RAUZY A., L'évaluation sémantique en calcul propositionnel, thèses, Luminy, janvier 1989.

PACLET Ph., "Aventures en noir et blanc", J&S n° 26, p. 28-30, avril-mai 1984.

QUINE W.V.O., ***Methods of logic,*** Holt, Rinehart & Winston, 1950, traduit en français par CLAVELIN M., Armand Colin, 1972.

SAMUELS A.L., "Some studies in machine leaming using the game of checkers", ***IBM jnl. of res. and dev.,*** 3, 1959, réimprimé dans [Feigenbaum 19631.

SCHOLL P.C., ***Algorithmique et représentation des données, 3 - récursivité et arbres,*** Masson, 1984.

SCHORR H., WAITE W.M., ***An efficient machine independentprocedure for garbage collection in various list structures,*** CACM, août 1967.

SIEGEL P., Représentation et utilisation de la connaissance en calcul propositionnel, thèse, Luminy, 1987.

SMULLYAN R., ***The lady or the tiger,*** Knopf 1982, (traduit en français par MARTHON J., ***Le livre qui rend fou,*** Dunod, 1984).

WARSHALL S., "A theorem on Boolean matrices", ***JACM*** 9, ***1,*** janvier 1962.

WIRTH N., ***Algorithms + Data Structures = Programs,*** Prentice Hall, 1976.

WIRTH N., ***Systematic programming : an introduction,*** Prentice-Hall, 1973, (traduit en français par LECARME O., ***Introduction à la programmation systématique,*** Masson 1977).

http://dzdelphi.blogspot.com/

Glossaire

1. Français - anglais

Adressage dispersée	Hash code	2.5
Arbre	**Tree**	4
Arbre équilibré	**Balanced tree**	8
Arbre ordonné	**Ordered tree**	8
Assertion	Assertion	2.4
B-arbre	**B-tree**	8
Branche	**Branch**	4.3
CHOIX	CHOICE	2.2.3
Classe	Class	2.5
Clé	**Key**	2.5
Consommateur	Consumer	4.2
Dépiler	Pull	4.1
Diagramme de vigne	Vine diagram	4.3.1
Diviser pour **règner**	Divide and **conquer**	2.3
Empiler	Push	4.1
Etat du monde	State of the world	2.1.1
ETPUIS	CAND	2.2.2
Fermeture transitive	Transitive closure	4.5.4
Feuille	**Leaf**	4.3
File	Queue ou FIFO	4
FINCHOIX	ENDCHOICE	2.2.3
Graphe	**Graph**	4
Infixé	**Infixed**	4.3.4
Invariant	Invariant	2.2.1
Largeur d'abord	**Breadth** first search	4.3.4
Marche arrière	Backtracking	6
Nœud	**Node**	4.3
OUALORS	COR	2.2.2
Pile	**Stack** ou LIFO	3.4.1
Post-condition	**Postcondition**	2.2.1
Postfixé	**Postfixed**	4.3.4
Pré-condition	Precondition	2.2.1
Prédécesseur	Predecessor	4.3
Préfixé	Prefixed	4.3.4
Producteur	**Producer**	4.2
Profondeur d'abord	Depth first search	4.3.4
Racine	Root	4.3
Ramasse-miettes	Garbage collection	4.5.1
Récurrence	Induction	5
Récursivité	**Recursion**	5
Schéma de programme	Program scheme	2.4
Schorr-Waite		4.5.1
Spécification	**Specification**	2.4
Successeur	Successor	4.3
Treillis	Lattice	4
Tri	Sort	3

2. Anglais - français

Assertion	Assertion	2.4
B-Tree	B-arbre	8
Backtracking	Marche **arrière**	6
Balancedtree	Arbre **équilibré**	8
CAND	**ETPUIS**	2.2.2
CHOICE	CHOIX	2.2.3
Class	Classe	2.5
Consumer	Consommateur	4.2
COR	**OUALORS**	2.2.2

http://dzdelphi.blogspot.com/

Divide and conquer	Diviser pour régner	3.3
ENDCHOICE	FINCHOIX	2.2.3
FIFO	File	4
Garbage collection	Ramasse miettes	4.5.1
Graph	Graphe	4
Hash code	Adressage dispersée	2.5
Induction	Récurrence	5
Invariant	Invariant	2.2.1
Key	Clé	2.5
Lattice	Treillis	4
LIFO	Pile	4.4.1
Ordered tree	Arbre ordonné	8
Postcondition	Post-condition	2.2.1
Precondi tion	Pré-condition	2.2.1
Producer	Producteur	4.2
Program scheme	Schéma de programme	2.4
Pull	Dépiler	4.1
Push	Empiler	4.1
Queue	File	4
Recursion	Récursivité	5
Schorr-Waite		4.5.1
sort	Tri	3
Specification	Spécification	3.4
Stack	Pile	4.4.1
Transitive closure	Fermeture transitive	4.5.4
Tree	Arbre	4
Vme diagram	Diagramme de vigne	4.3.1

Solutions de quelques exercices

CHAPITRE 2

Question 1

Voir chapitre 7.

Question 2

Supposons que nous arrivons dans la situation suivant :

objet=15, bas=centre=7, haut=8, t[7]=10, t[8]=20

Alors, l'objet n'est pas `trouvé` et **t[centre]<objet.** La **prochaine valeur de centre** sera **entier((7+8)/2),** c'est-Mire 7. L'algorithme boucle.

CHAPITRE 3

Question 1

```
DONNEES n: entier;
      t: TABLEAU [O..n] DE entier;
PRECOND n>0;
DEBUT VAR i, j, temp: entier;
      MONDE i: t[l ..i] est trié;
      i:=1 ; t[0]:= - maxint
      TANTQUE i<n
      FAIRE i:=i+l ;
            MONDE j est la position du trou,
            temp:=t[i]; j:=i;
            TANTQUE j>1 ET t[j-1]>temp
            FAIRE t[j]:=t[j-1]; j:=j-1
            FAIT;
            t[j]:=temp
      FAIT
FIN
POSTCOND t[ 1 ..n] est trié
```

http://dzdelphi.blogspot.com/

Le problème vient du fait que quand **j=1, t[j-1]** n'existe pas. On crée un t[O], fictif, en l'initialisant au plus petit entier négatif. En fait, cette initialisation n'est **pas** nécessaire, car si **t[0]** est **référencé,** il n'est plus vrai que **j>** 1.

CHAPITRE 4

Question 1

```
PROC trouver(nom: chaine(8), n: entier) -> (bool, entier):
    VAR b: bool, x: entier;
    SI n=O
    ALORS RETOURNER (FAUX, 0)
    SINON SI val[n]=nom
        ALORS RETOURNER (VRAI, n)
        SINON (b, x):= trouver(nom, sg[n]);
            SI b
            ALORS RETOURNER (b, x)
            SINON RETOURNER trouver(nom, sd[n])
            FINSI
        FINSI
    FINSI
FINPROC

PROC parent(p, enf: chaine(8), n: entier) -> (bool, entier):
    VAR b: bool, x: entier;
    SI n=O
    ALORS RETOURNER (FAUX, 0)
    FINSI;
    SI val[n]=p
    ALORS SI val[sg[n]]=enf OU val[sd[n]]=enf
            ALORS RETOURNER (VRAI, n)
            FINSI
    FINSI;
    (b, x):= trouver(p, enf, sg[n]);
    SI b
    ALORS RETOURNER (b,x)
    SINON RETOURNER parent(p, enf, sd[n])
    FINSI
FINPROC
```



```
PROC cousin_germain(a, b: chaine(8), n: entier) -> (bool, entier, entier):
    SI n=0
    ALORS RETOURNER (FAUX, 0)
    SINON SI ((val[sg[sg[n]]=a OU val[sd[sg[n]]=a)
            ET (val[sg[sd[n]]=b OU val[sd[sd[n]]=b))
            OU ((val[sg[sg[n]]=b OU val[sd[sg[n]]=b)
            ET (val[sg[sd[n]]=a OU val[sd[sd[n]]=a))
        ALORS RETOURNER VRAI
        FINSI;
        SI cousin_germain(a, b, sg[n])
        ALORS RETOURNER VRAI
        SINON RETOURNER cousin_germain(a, b, sd[n])
        FINSI
    FINSI
FINPROC

PROC aïeul(a, enf, n) -> bool:
    VAR b: bool, x, y: entier;
    (b, x):=trouver(a, n);
    SI b
    ALORS (b, y):=trouver(enf, x); RETOURNER b
    FINSI
FINPROC
```

REMARQUE. — Cette version suppose qu'un nom ne peut figurer qu'une fois dans l'arbre. Elle n'assure pas l'impression, qui pose le problème particulier de l'ordre d'impression demandé. Supposons une **procédure** trouver-bis, qui comporte **un** ordre d'impression. Alors les résultats seraient imprimés avec les noms des enfants avant ceux de leurs parents. Le plus simple est d'empiler le nom à l'intérieur de la procédure (dans **une** pile globale) et d'imprimer la pile à la fin du programme.

Question 3

Un tel arbre comporte (2^n-1) noeuds et 2^{n-1} feuilles.

CHAPITRE 5

Question 1

Le nombre de déplacements est $2^{64} - 1$ (voir §5.1.1)

http://dzdelphi.blogspot.com/

$2^{64} - 1$ secondes = 18446744073709551615 secondes =
= 307445734561825860 minutes 15 secondes
= 5124095576030431 heures 15 secondes
= 213087315667934 jours 15 heures + ...
= 5 838 008 664 843 années 239 jours + ...

(avec l'année à 365 jours, c'est-Mire sans prendre en compte les années bissextiles).

En arrondissant à 5 800 milliards d'années, on voit que la fin du monde n'est pas pour demain. Le résultat est un bel exemple de l'explosion combinatoire.

Question 3

L'ordre est inversé à chaque changement de disque, c'est-à-dire que d_2 fait des cercles dans le sens oppose à celui de d_1 et, plus généralement, d_i tourne dans le opposé à d_{i-1}.

Question 4

```
DEBUT DONNEES i: entier;
      VAR n: entier;
      n:=1;
      TANTQUE 2 DIVISE i FAIRE i:=i/2; n:=n+1 FAIT
FIN
```

i est l'index du coup, n l'index du disque qui bouge à coup i. Tous les coups impairs sont du disque d_1. Pour les impairs, on divise i par deux, en enlevant le disque 1 (passage au monde des 2) . . . Voir le calcul du pivot dans le problème des permutations.

Question 5

Le pivot marche toujours de droite à gauche et on génère le i-ième permutation de n objets


```
perm: TABLEAU (1 :n) DE integer INIT 1;
nn:=n;
TANTQU E nn> 1
FAIRE (p,q):=i DIVPAR nn;
      SI q=0 ALORS p:=p-1 ; q:=nn FINSI;
      j:=n+1;
      TANTQUE q> 1
      FAIRE j:=j-1;
            TANTQUE perm(j)>1 FAIRE j:=j-1 FAIT;
            q:=q-1
      FAIT,
      perm(j):=nn; nn:=nn-1; i:=p+1
FAIT
```

Question 6

Ordre alphabétique, toujours pour tirer le i-ième permutation. C'est la même chose en plaçant les objets à l'envers

```
TANTQUE n>0
FAIRE n:=n-1 ; (p,q):=i DIVPAR fact(n);
      SI q=O ALORS p:=p-1; q:=fact(n) FINSI;
      j:=1;
      TANTQUE perm(j)>0 FAIRE j:=j+I FAIT;
      TANTQUE j<p
      FAIRE j:=j+1 ;
            TANTQUE perm(j)>0 FAIRE j:=j+I FAIT
      FAIT;
      perm(j):=p+1 ; i:=q
FAIT
```

Question 8

Oui. Chacun des fact(n) permutations est différente des autres (la relation avec la permutation d'origine est bi-univoque). Comme elles sont toutes différentes et il y a le compte . . .

http://dzdelphi.blogspot.com/

CHAPITRE 6

Question 1

Version avec heuristique :

```
PROCEDURE prochain(dir) RETOURNER entier;
    RETOURNER(9 dir=0
                ALORS dirpref(x,y,xf,yf)
                SINON SI dir=4
                        ALORS 1
                        SINON dir+1
                        FINSI
                FINSI)
FINPROC;
PROCEDURE dirpref(x,y,xf,yf) RETOURNER entier;
    RETOURNER(SI xcxf ET y<=yf
                ALORS 1
                SINON SI y<yf ET x>=xf
                    ALORS 2
                    SINON SI x>xf ALORS 3 SINON 4 FINSI
                    FINSI
                FINSI)
FINPROC;
x:=xs; y:=ys; marque(x,y):=true; dir:=0; empiler(O);
TANTQUE xoxf OU y<>yf
FAIRE SI prochain(dir)<>dirpref(x,y,xf,yf)
    ALORS dir:=prochain(dir);
        xn:= CAS dir DANS (x,x+1 ,x,x-l) FCAS;
        yn:= CAS dir DANS (y+1 ,y,y-1 ,y) FCAS;
        SI NON barrière(x,y,dir) ETPUIS NON marque(xn,yn)
        ALORS x:=xn; y:=yn; marque(x,y):=vrai;
            empiler(dir); dir:=0
        FINSI
    SINON desempiler(dir);
        SI dir=0
        ALORS -- il n'y a pas de chemin
        FINSI;
        x:= CAS dir DANS (x,x-l ,x,x+1) FINCAS;
        y:= CAS dir DANS (y-l ,y,y+1 ,y) FINCAS
    FINSI
FAIT
```


Question 8

```
DEBUT tab: TAB [1 ..5, 1..5] DE entier INIT 0;
    occupé: TAB [0..16] DE bool INIT FAUX;
    i: entier INIT 1; j: entier INIT 0;
    p, q, val: entier; marche: bool;
    TANTQUE i<6
    FAIRE j:=j+l ;
        TANTQUE occupé[j] ET j<16 FAIRE j:=j+l FAIT;
        SI j<16
        ALORS tab[5,i]:=j; occupé[j]:=VRAI; p:=5; q:=i; marche:=VRAI;
            TANTQUE p>1 ET q>1 ET marche
            FAIRE val:=abs(tab[p,q]-tab[p,q-1]);
                SI occupé[val]
                ALORS marche:=FAUX;
                    TANTQUE pc6
                    FAIRE occupé[tab[p,q]]:=FAUX; tab[p,q]:=0;
                        p:=p+1 ; q:=q+l
                    FAIT
                SINON p:=p-1 ; q:=q-1 ; tab[p,q]:=val; occupé[val]:=VRAI
                FINSI
            FAIT;
            SI marche ALORS i:=i+l ; j:=O FINSI
        SINON i:=i-1 ; x:=5; y:=i; j:=tab[5,i];
            TANTQUE y>0
            FAIRE occupé[tab[x,y]]:=FAUX; tab[x,y]:=0; x:=x-l; y:=y-1
            FAIT
        FINSI
    FAIT
FIN
```

CHAPITRE 7

Question 1

```
puiss(a,b): SI b=O
                ALORS 1
                SINON SI impair(b)
                    ALORS a * puiss(a, b-l)
                    SINON puiss(a*a, b/2)
                    FINSI
                FINSI
```


http://dzdelphi.blogspot.com/

```
puiss(a,b,res): res:=1;
                TANTQUE b>0
                FAIRE SI impair(b)
                      ALORS res:=res*a; b:=b-1
                      FINSI;
                      a:=a*a; b:=b/2
                FAIT
```

Question 2

div2(n) = 1/fact(n)

divfact(n) = $\frac{n*(n-2)*(n-4)*\ldots}{(n-1) * (n-3) * \ldots}$

= SI pair(n)
ALORS $2^n * (fact(n/2))^2 / fact(n)$
SINON $fact(n) / 2^{n-1} / (fact((n-1)/2))^2$
FINSI

```
DEBUT DONNEES n: entier;
    VAR div2: entier INIT1;
    TANTQUE n>1 FAIRE div2:=div2/n; n:=n-1 FAIT
FIN

DEBUT DONNEES n: entier;
    VAR haut, bas, divfact: entier;
    haut:=1; bas:=1;
    TANTQUE n>1
    FAIRE haut:=haut*n; bas:=bas*(n-1); n:=n-2
    FAIT;
    divfact:=haut/bas
FIN
```


CHAPITRE 8

Question 1

```
DEBUT DONNEES taille, racine: entier;
        sg, sd, val: TAB (1 ..taille) DE entier;
    PROCEDURE ord(n, min, max: entier) RETOURNE bool;
        SI n=0
        ALORS RETOURNER VRAI
        SINON SI val(n)>max OU val(n)<min
              ALORS RETOURNER FAUX
              SINON RETOURNER(ord(sg(n), min, val(n)-1)
                            ET ord(sd(n), val(n)+1, max))
              FINSI
        FINSI
    FINPROC;
    imprimer(ord(racine, - maxint, maxint))
FIN
```

Question 2

```
DEBUT DONNEES sg, sd, val: TABLEAU (1 ..taille) DE entier;
        racine: entier;
    VAR x: entier;
    PROC équ(n, poids: entier) RETOURNE bool;
        VAR éqg, éqd: bool;
            pg, pd: entier;
        SI n=0
        ALORS poids:=0; RETOURNER VRAI
        SINON éqg:=équ(sg(n), p g); éqd:=équ(sd(n), p d);
poids:=pg+pd+l ;
              RETOURNER (éqg ET éqd ET abs(pg-pd)<2)
        FINSI
    FINPROC;
    imprimer(équ(racine, x))
FIN
```

C
I 11
GRIFM

la Villette
360243 8

http://dzdelphi.blogspot.com/

Algorithmique et programmation est issu d'un enseignement délivré depuis vingt-cinq ans à des étudiants de 1er et 2e cycles universitaires et dans le cadre de la formation permanente. Il s'adresse donc aussi bien aux étudiants qui désirent se spécialiser en informatique qu'aux professionnels de l'entreprise qui souhaiteraient acquérir des éléments de rigueur en programmation.

A partir des exemples classiques (par exemple les tours d'Hanoi), la description et l'examen des différentes méthodes de construction d'algorithmes sont proposés. Des exercices avec leur solution concluent chaque chapitre et résument l'essentiel des points abordés.

L'auteur

Michael Griffiths, docteur ès sciences et spécialiste de logiciels de base et systèmes compilateurs, a été directeur technique et directeur administratif du Centre de calcul régional de Nancy. Il est actuellement professeur à l'Université de Nantes.
Il est l'auteur, avec M. Vayssade, de *Architecture des systèmes d'exploitation* aux éditions Hermès.

Editions HERMES
34, rue Eugène Flachat
75017 Paris

la Villette
360243 8
http://dzdelphi.blogspot.com/