

Algorithmique

F. Didier

IREM Campus de Luminy

Table des matières

1	Algorithmique	5
1.1	Généralités	6
1.2	Présentation d'un langage d'écriture des algorithmes	9
1.2.1	Les données	9
1.2.2	Les instructions	13
1.3	Quelques algorithmes classiques	16
1.3.1	Construction de la liste des nombres premiers par la méthode du crible d'Erathostène	16
1.3.2	Décomposition d'un entier n en produit de facteurs premiers	18
1.3.3	Décomposition en base b d'un entier a	19
1.3.4	Evaluation d'un nombre écrit en base b	20
1.3.5	Développements décimaux illimités	21
1.3.6	L'algorithme de Hörner	23
1.3.7	Algorithme d'exponentiation rapide	25
1.3.8	Un principe fondamental : la dichotomie	26
1.4	Illustration des notions de preuve et de terminaison d'un algorithme	30
1.4.1	Algorithme de multiplication : Un premier algorithme	30
1.4.2	Algorithme de multiplication : Un deuxième algorithme	31
1.4.3	Algorithme de multiplication : Un troisième algorithme	33
1.4.4	Algorithme d'Euclide pour le calcul du PGCD de nombres entiers	34
1.4.5	Algorithme d'Euclide étendu	35
1.4.6	Division euclidienne	37
1.5	Construction de programmes itératifs par la mise en évidence de l'invariant	38
1.5.1	La démarche	39
1.5.2	Tri d'un tableau par insertion	40
1.5.3	Tri d'un tableau par sélection	42
1.5.4	La recherche séquentielle	44
1.5.5	La recherche dichotomique	46

1.5.6	Recherche d'un plus petit élément dans un tableau	54
1.5.7	Recherche de l'emplacement du plus petit élément dans un tableau	55
1.5.8	Recherche dans un tableau $C1$ d'une sous suite extraite égale à un tableau $C2$ donné	56
1.5.9	Partition dans un tableau	57
1.6	Analyse des algorithmes	62
1.6.1	Quelques exemples de calculs de complexité	63

Chapitre 1

Algorithmique

Ce chapitre a pour but d'apporter un complément d'information dans le domaine de l'algorithmique aux professeurs de mathématiques qui enseignent au Lycée. Il peut être aussi une aide pour les enseignants de la spécialité ISN en terminale S. Il comporte six parties.

- **Généralités.** On introduit naïvement la notion d'algorithme et on évoque les problèmes importants qui s'y rattachent.
- **Présentation du langage.** On aborde dans cette partie la notion de variable informatique et des données manipulées par un algorithme. On présente ensuite la syntaxe des instructions qui seront utilisées pour décrire les algorithmes de ce livre.
- **Quelques algorithmes élémentaires.** La plupart des algorithmes présentés sont issus du domaine de l'arithmétique et sont abordables au lycée. On y présente également trois algorithmes connus pour leur efficacité, le principe de dichotomie, l'algorithme de Hörner et l'algorithme d'exponentiation rapide.
- **Illustration des notions de preuves et de terminaison.** La notion de preuve de la validité d'un algorithme est abordée par la notion d'invariant de boucle. Elle donne un autre éclairage du raisonnement par récurrence, raisonnement que les élèves ont souvent du mal à maîtriser.
- **Une méthode pour élaborer des algorithmes itératifs.** On présente une méthode pour aider à imaginer et ensuite écrire des algorithmes mettant en jeu une itération. Les exemples développés pour illustrer cette façon de procéder sont tous non numériques. Cette partie peut s'avérer utile pour les enseignants de la spécialité ISN.

- **Analyse de la complexité des algorithmes.** Cette partie aborde le point important de la comparaison des algorithmes sous l'angle de leur efficacité.

1.1 Généralités

Les algorithmes sont très souvent considérés comme du domaine des mathématiques et de l'informatique, leur champ d'application est en réalité beaucoup plus vaste. Les historiens s'accordent pour dire que le mot "algorithme" vient du nom du grand mathématicien persan *AlKhowârizmî* (an 825 environ) qui introduisit en Occident la numération décimale et les règles de calcul s'y rapportant. La notion d'algorithme est donc historiquement liée aux manipulations numériques, mais ces dernières décennies elle s'est progressivement développée sur des objets plus complexes, graphes, textes, images, sons, formules logiques, robotique, etc... Le but de ce chapitre est de préciser ce qu'on entend par algorithme, de présenter un langage dans lequel on pourra les exprimer, d'évoquer et d'illustrer les problèmes qui se posent dans leur élaboration.

Approche naïve : C'est une méthode i.e une façon systématique de procéder pour faire quelque chose : trier, rechercher, calculer,... Il répond à des questions du type : comment faire ceci ?, obtenir cela ?, trouver telle information ?, calculer tel nombre ?, ... C'est un concept pratique, qui traduit la notion intuitive de procédé systématique, applicable mécaniquement, sans réfléchir, en suivant simplement un mode d'emploi précis.

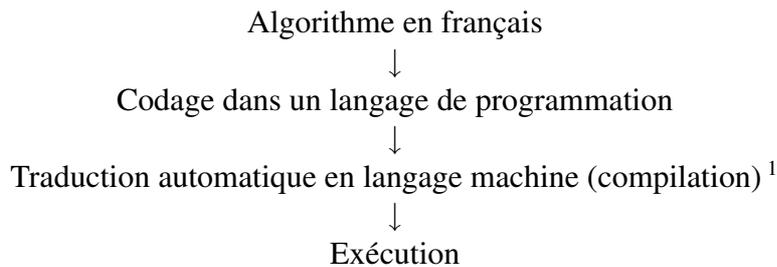
Tentative de définition : Suite d'ordres précis et compréhensibles, exécutables de manière non ambiguë par un automate suivant un ordre parfaitement déterminé en vue de résoudre une classe de problèmes.

Par exemple : classer des suites de nombres, chercher des mots dans un dictionnaire sont des exemples de classe de problèmes que des algorithmes sont capables de résoudre.

On évitera de prendre des exemples de la vie courante pour illustrer le concept d'algorithme : une recette de cuisine est un mauvais exemple d'algorithme car elle est toujours trop imprécise : une pincée de sel (combien de grammes ?), faire bouillir dix minutes (la température d'ébullition dépend entre autre de l'altitude à laquelle on se trouve),...

En général l'automate est une machine, et très souvent un ordinateur. L'activité principale en algorithmique est de concevoir et d'exprimer un algorithme. La

plupart du temps cet algorithme sera par la suite codé dans un langage de programmation. En général, on l'exprime en langue naturelle, en utilisant des expressions et des tournures parfaitement définies qui pourront être facilement transcrites dans un langage de programmation.



Points importants de la conception d'un algorithme : Une des difficultés dans l'élaboration d'un algorithme est de contrôler son aspect dynamique (exécution) à partir de son écriture statique (suite finie d'instructions). Un algorithme de tri peut s'écrire en quelques lignes et peut trier aussi bien une suite de 10 éléments qu'une suite de 100 000 éléments. Les problèmes importants qui se posent dans la conception d'un algorithme sont :

- **La preuve :** prouver que l'algorithme résout la classe de problèmes pour laquelle il a été écrit.
- **L'arrêt :** prouver que quelles que soient les données fournies en entrée l'algorithme s'arrête toujours.
- **Le choix des structures des données :** la façon d'organiser les données manipulées influence l'écriture et la performance de l'algorithme.
- **Son efficacité.** Il est très important d'avoir une idée du comportement d'un algorithme lorsque la taille des données qui lui sont transmises en entrée augmente.

L'essentiel, dans l'élaboration d'un algorithme, est de percevoir les éléments clés d'un processus quelconque, et d'imaginer la suite d'opérations logiques les plus pertinentes et les plus efficaces pour le mettre en œuvre de manière automatique et performante.

Un algorithme peut être vu comme le squelette abstrait du programme informatique, sa substantifique moelle, indépendant du codage particulier qui permettra sa mise en œuvre avec un ordinateur ou une machine mécanique.

Ainsi l'activité algorithmique est indépendante des langages de programmation.

1. Les deux dernières étapes ne font plus qu'une seule dans le cas où le langage est interprété

Domaines où interviennent les algorithmes : L'algorithmique intervient dans des domaines divers et variés :

- Numériques.
- Tri, recherche de mots, plus généralement recherche d'information (génomes, internet, ...).
- Algorithmes géométriques, images de synthèse.
- Reconnaissance de formes.
- Compression des données.
- Cryptographie.
- ...

Malgré les énormes progrès de la technologie, les ordinateurs seront toujours soumis à des limitations physiques : nombre d'opérations par seconde, taille de la mémoire, ... Une part importante de la recherche en algorithmique consiste à élaborer des algorithmes de plus en plus efficaces. Par exemple, en 1950 on pouvait calculer quelques milliers de décimales de π et en 2002 on en était à plus d'un trillion (10^{18}). C'est à peu près à part égale en raison des avancées technologiques et algorithmiques. La recherche systématique d'efficacité passe aussi par la recherche de nouveaux types de représentation et d'organisation des données : classement des mots, organisation arborescente, ...

D.Knuth² considère les arbres comme la structure la plus fondamentale de toute l'informatique.

2. informaticien célèbre pour ses travaux en algorithmique, également créateur de T_EX

1.2 Présentation d'un langage d'écriture des algorithmes

1.2.1 Les données

D'une manière générale les algorithmes manipulent des données qui représentent une abstraction de la réalité dans le sens que certaines propriétés et caractéristiques des objets réels sont ignorées car non pertinentes pour le problème particulier que l'on a à traiter. Le choix de la représentation des données est souvent délicat et ne découle pas du langage dans lequel sera codé l'algorithme. C'est très souvent à la lumière des principales opérations que l'on va faire sur ces données que l'on choisit leur représentation.

Le problème du choix des structures des données n'est pas le sujet de ce livre. Les langages de programmation permettent de raisonner plus aisément en termes de notions familières comme nombres, listes, tableaux, répétitions,..., qu'en termes de bits, stockages, adresse mémoire, registres, sauts,... L'idée, que certains ont eu il y a quelques décennies, de définir un langage universel a été rapidement abandonnée et l'on a vu ces dernières années les langages de programmation se multiplier. Chacun est spécifique au domaine dans lequel il est utilisé (Cobol pour la gestion, Matlab pour le calcul matriciel, C++ pour développer des logiciels systèmes, Php pour construire et manipuler des pages web, etc...).

Quelques définitions et remarques

Comme on vient de le dire les algorithmes sont amenés à manipuler des données. Ces données seront repérées par des noms. En informatique le nom d'une donnée est appelé **identificateur**. La règle communément adoptée pour écrire un identificateur est une suite de lettres ou de chiffres commençant par une lettre, le caractère espace étant interdit, pour faciliter la lecture des noms composés le caractère "_" (appelé "blanc souligné", ou "tiret bas", ou encore "underscore") est considéré comme une lettre.

Une donnée possède toujours un **type**.

Types élémentaires

- **caractère**

Lettres minuscules, lettres majuscules, chiffres, symboles comme +, *, /, -, @, <,....

- **texte**

Un texte, appelé en informatique chaîne de caractères (string en anglais) est une suite finie de caractères.

– **Booléen**

Une donnée de type booléen ne peut avoir que deux valeurs possibles **vrai** ou **faux** (**true** ou **false**).

Concernant les expressions booléennes, on peut dire simplement qu'elles sont obtenues en comparant entre elles deux expressions, dont la comparaison est possible, à l'aide d'opérateurs de comparaison : $<$, \leq , $>$, \geq , $=$, \neq , ... Des expressions plus complexes peuvent être construites en utilisant les opérateurs booléens :

ou, et, non

qui dénotent respectivement la disjonction, la conjonction et la négation.

– **Nombre entier**

– **Nombre réel**

Le terme réel est employé par opposition au type entier. Ce terme est en fait impropre car on ne peut représenter en informatique que des approximations des nombres réels.

Deux types composés

Les types **tableau** (array) et **liste** (list) représentent une séquence **linéaire d'éléments**. Le terme **élément** signifiant une information de n'importe quel type (liste linéaire d'entiers, de réels, de chaînes, ...) et le terme **linéaire** caractérise la linéarité des positions des éléments entre eux. A savoir qu'il y a un premier élément, un dernier élément et que l'élément de rang i est précédé (s'il n'est pas le premier) de l'élément de rang $i - 1$ et suivi (s'il n'est pas le dernier) de l'élément de rang $i + 1$.

– **Tableau**

Dans ce type de données il est possible d'accéder à un élément par son numéro. On peut ainsi le consulter, le modifier, insérer avant ou après un autre élément. On utilise la notation $t[i]$ pour désigner la composante de rang i du tableau t .

En informatique, lorsqu'on programme, ce qui caractérise un tableau est que ses éléments, tous du même type, sont stockés séquentiellement, c'est à dire que les éléments se trouvent à des adresses consécutives en mémoire. Plus précisément si la variable t désigne un tableau, l'adresse a associée à t est l'adresse du premier élément. L'élément de rang i se trouve lui à l'adresse $a + i * (\text{taille d'un élément})$, lorsque les éléments sont indexés à partir de 0. La taille étant exprimée en nombre d'octets. Les tableaux sont toujours indexés par un ensemble discret, par des entiers consécutifs la plupart du temps, mais il est parfois possible, suivant les langages, de les indexer par autre chose, par des caractères par exemple. Lorsque les tableaux

sont indexés par des entiers, l'indexation commence tantôt à 0 (Caml, C, Python), tantôt à 1 (Pascal). Signalons encore que certains langages offrent la possibilité d'indexer les tableaux par un intervalle quelconque d'entiers consécutifs (Maple, Pascal). Une caractéristique importante des tableaux est que quelle que soit la position de l'élément, le temps pour y accéder est le même : on effectue dans tous les cas une addition et une multiplication. En revanche, les opérations d'insertion et de suppression sont coûteuses car elles impliquent de décaler des éléments. Dans la plupart des langages $t[i]$ (appelée **variable indicée**) désigne suivant son emplacement, tantôt la valeur, tantôt l'adresse de l'élément de rang i . Signalons que quelques rares langages (Caml, Matlab) utilisent comme notation les parenthèses à la place des crochets.

– **Liste**

Comme pour une donnée de type tableau il est possible d'accéder à un élément par son numéro pour le consulter, le modifier. On utilise la notation $l[i]$ pour désigner la composante de rang i de la liste l . On désigne la liste vide par []. Les écritures e, l et l, e désignent l'ajout d'un élément e respectivement en tête et en queue de la liste l .

En informatique, ce qui caractérise une "vraie" liste (Lisp, Caml) est que les éléments ne sont plus situés à des emplacements consécutifs de la mémoire. Les éléments d'une liste, contrairement aux éléments d'un tableau, peuvent être de différents types. Pour réaliser cela, un mécanisme d'adressage par liens (transparent pour l'utilisateur) est utilisé. Chaque élément contient, en plus de l'information qui lui est associée, l'adresse où est située l'élément qui le suit. Ainsi pour accéder à l'élément de rang i , on est obligé de parcourir tous les éléments qui le précèdent. Contrairement aux tableaux le temps d'accès n'est plus constant. Les principales opérations caractéristiques que l'on peut effectuer sur une liste sont la possibilité d'accéder à l'élément en **tête** (head), de considérer la **queue** de la liste (tail, la liste formée par tous les éléments, sauf la tête), d'ajouter un élément en tête (**construire**), de **concaténer** deux listes. Ces opérations (exceptée la première) ne sont pas réalisables de manière efficace avec des tableaux.

Remarques

Plus récemment les langages ont tendance à confondre ces deux notions fondamentalement différentes de tableaux et de listes. Par exemple en Maple, on peut accéder à l'élément de rang i d'une liste en utilisant l'indexation comme pour un tableau. En Matlab on peut réaliser sur les tableaux certaines opérations réservées aux listes (ajout d'un élément en tête, concaténation de tableaux, . . .). Dans le manuel de référence du langage Python, on ne trouve que le terme liste. Mais sous Python, l'accès à l'élément d'une liste est tout aussi efficace que pour un tableau, et les opérations d'insertion et de suppression sont coûteuses comme dans un ta-

bleau ! En fait, dans ces langages, les tableaux se comportent comme des listes et évoluent dynamiquement au grès des concaténation et/ou ajouts et les listes sont en fait des tableaux de pointeurs. La façon dont se fait l'accès est transparente pour le programmeur. En Python, il est possible d'utiliser une bibliothèque pour manipuler de "vrais" tableaux, mais on perd alors en clarté !

Variables

Dans la description d'un algorithme on est amené à donner des noms aux données manipulées par celui-ci. Ainsi à chaque donnée est associé un **nom** et un **type**. Le nom permet de désigner la donnée et le type permet de préciser la nature de cette donnée, ce qui détermine les opérations que l'on peut lui appliquer.

En informatique, le terme **variable** est utilisé pratiquement par tous les manuels de référence des divers langages de programmation. Dans ce cadre là, une variable peut être vue comme un quadruplet (nom, type, adresse, valeur) :

- le nom d'une variable est un identificateur ;
- le type d'une variable précise la nature des données qu'elle contient ;
- l'adresse désigne l'emplacement dans la mémoire où est rangée la valeur associée à la variable ;
- la valeur est décrite par la configuration des bits des octets qui sont associés à cette variable. Le premier octet est repéré par l'adresse, le nombre d'octets est lui fonction du type. Cette valeur varie au cours de l'exécution.

Une variable peut avoir des significations différentes suivant son emplacement dans le programme. Cela dépend des langages. Par exemple si elle est située dans une instruction de lecture (*lire a*) ou à gauche du symbole d'affectation ($a \leftarrow a+1$) la variable désigne l'**adresse** qui lui est associée, ailleurs elle désigne sa **valeur**. D'autres langages ont fait le choix de préciser ce que la variable désigne par une marque syntaxique. Par exemple en Caml une variable *a* définie par *let a = ref 0* indique que *a* est l'adresse d'un entier initialisé à 0, si l'on veut utiliser la valeur de *a* on le fait précéder par un point d'exclamation. Ainsi pour incrémenter *a* de 1 on écrira $a := !a + 1$ ($:=$ étant le symbole de l'affectation en Caml). Notons que la notion de variable en informatique n'a pas grand chose à voir avec la notion de variable en mathématique.

Certains langages (Pascal, C, ...) obligent le programmeur à préciser le **type** de chaque variable avant son utilisation, d'autres ont fait le choix inverse (Caml, Python). Dans ce dernier cas c'est la façon de les manipuler (initialisations, opérateurs utilisés, ...) qui permet de leur attribuer un type. D'autres (Maple) encore laissent le choix au programmeur. Dans tous les cas, à une variable est toujours associé un type.

Le choix que nous avons arrêté pour l'écriture des algorithmes est de ne pas écrire de partie déclarations de variables.

C'est lors de l'énumération des variables que l'algorithme attend en entrée que l'on précisera, en "français", le type de chacune d'elles. Ainsi, lors du codage, chacun s'adaptera aisément aux contraintes imposées par le langage qu'il aura choisi.

1.2.2 Les instructions

L'instruction d'affectation

Cette instruction permet de modifier la valeur d'une variable. La syntaxe que nous avons retenue est :

"expression partie gauche" \leftarrow "expression partie droite"

L'expression en partie gauche (très souvent une variable) doit impérativement désigner une **adresse**. Cette instruction a pour but de ranger la **valeur** de l'expression partie droite à l'adresse indiquée par la partie gauche.

Exemple :

$k \leftarrow k + 1$, la variable k reçoit la valeur de l'expression $k + 1$. On dit aussi que l'on **affecte** à k la valeur de l'expression $k + 1$

D'une manière générale, dans les langages de programmation, la syntaxe de cette instruction est :

"expression partie gauche" "symbole" "expression partie droite"

Seul le symbole varie suivant les langages.

- := en Algol, Pascal, Maple,...
- = en Basic, C, Python...
- \leftarrow en LSE³

Dans tous les cas elle signifie que *"la valeur de l'expression partie droite est rangée à l'adresse associée à l'expression partie gauche"*. Signalons que sur certaines calculatrices le symbole \rightarrow est utilisé pour l'affectation. On écrit donc "expression" \rightarrow "variable". Cela est très ennuyeux car les élèves qui ont l'habitude de manipuler ces matériels ont tendance à inverser le sens de l'affectation lorsqu'ils se mettent à programmer sur ordinateurs... Nous avons choisi comme symbole \leftarrow car il est cohérent avec les notions de "partie gauche" et "partie droite" qui se retrouvent dans tous les langages utilisés et il permet d'éviter le symbole = qui perturbe les élèves (confusion avec l'égalité).

3. Langage Symbolique d'Enseignement créé spécifiquement, au début des 70, pour l'introduction de l'informatique au Lycée

Les instructions conditionnelles

Ces instructions sont le *si alors* et le *si alors sinon*.

L'instruction *si alors*

Elle s'écrit :

si "expression booléenne" **alors** "instruction"

L'instruction qui suit le **alors** n'est exécutée que si l'expression booléenne est vraie. Dans le cas contraire, c'est l'instruction suivante (celle qui suit le *si alors*) qui est exécutée.

Dans le cas où l'on souhaiterait exécuter après le **alors** non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical qui englobe toutes ces instructions.

```

si "expression booléenne" alors
    | instruction1
    | instruction2
    |  $\vdots$ 
    | instruction n

```

L'instruction *si alors sinon*

Elle s'écrit :

si "expression booléenne" **alors** "instruction 1" **sinon** "instruction 2"

Si l'évaluation de l'expression donne la valeur **vrai**, c'est l'instruction 1 qui suit immédiatement le **alors** qui est exécutée, sinon c'est l'instruction 2 qui suit le **sinon** qui est exécutée.

Dans le cas où l'on souhaiterait exécuter non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical.

```

si "expression booléenne" alors
    | instruction1
    | instruction2
    |  $\vdots$ 
    | instruction n
sinon

```

```

| instruction1
| instruction2
|  $\vdots$ 
| instruction p

```

Les instructions itératives

La boucle *tant que*

Elle s'écrit :

tant que "expression booléenne " **faire** " instruction".

Si l'"expression" booléenne est vraie on exécute l'instruction qui suit **faire**, puis l'"expression" booléenne est réévaluée à nouveau et ainsi de suite. La boucle s'arrête lorsque l'évaluation de l'"expression booléenne" donne la valeur faux. On passe alors à l'instruction suivante. Dans le cas où l'on souhaiterait exécuter non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical.

```

tant que "expression booléenne " faire
| instruction1
| instruction2
|  $\vdots$ 
| instruction n

```

La boucle *pour*

Elle s'écrit :

pour "variable" **variant de** "expr. init" **jusqu'à** "expr. fin" **faire** "instruction"

ou encore

pour "variable " **variant de** "expr. init" **jusqu'à** "expr. fin" **avec un pas de**
"expr. pas" **faire** "instruction"

L'instruction qui suit **faire** est exécutée pour les différentes valeurs de "variable". "variable" est appelée variable de contrôle de la boucle. Si l'*expression pas* n'est pas précisée, la variable de contrôle est incrémentée de 1 après chaque tour de boucle. Dans le cas où l'on souhaiterait exécuter non pas une, mais plusieurs instructions, on écrira ces instructions sur plusieurs lignes (une instruction par ligne) précédées par un trait vertical.

```

pour "variable" variant de "expr. init" jusqu'à "expr. fin" faire
    | instruction1
    | instruction2
    |  $\vdots$ 
    | instruction n

```

En général la variable de contrôle, l'expression initiale, l'expression finale et l'expression pas (si elle est présente) ont des valeurs entières. Ce sera toujours le cas dans les algorithmes étudiés.

On s'interdira de modifier la variable de contrôle à l'intérieur de la boucle et de l'utiliser en dehors de celle-ci.

L'instruction *résultat*

Sa syntaxe est :

résultat "expression"

Elle indique, lorsqu'elle est présente, le résultat calculé par l'algorithme. Lors de l'écriture du programme correspondant elle pourra être traduite, soit par une instruction de sortie (print, write, display, ...), soit par une instruction indiquant le résultat d'une fonction (return, ...) si l'on a choisi d'adapter l'algorithme en écrivant une fonction.

Contrairement à ce que permettent la plupart des langages actuels, **on s'imposera de n'utiliser qu'une seule instruction résultat et de placer celle-ci tout à la fin de l'algorithme.**

1.3 Quelques algorithmes classiques

Nous allons présenter ici quelques algorithmes élémentaires en arithmétique comme création de la liste des nombres premiers, décomposition d'un nombre entier en facteurs premiers, liste des diviseurs d'un nombre entier, changement de base, ... et trois algorithmes fondamentaux : dichotomie, Hörner et exponentiation rapide.

1.3.1 Construction de la liste des nombres premiers par la méthode du crible d'Ératostène

Cette méthode permet d'établir la liste de nombres premiers inférieurs à un nombre entier n donné. Son principe est très simple. On écrit la séquence de tous les nombres entiers de 1 jusqu'à n , on barre 1 qui n'est pas premier, on garde 2 qui

est premier et on barre tous les nombres multiples de 2, on garde 3 et on barre tous ses multiples, puis on recherche à partir de 3 le premier nombre non barré, on le garde et on élimine, en les barrant, tous les multiples de ce nombre, et on continue ainsi jusqu'à épuiser toute la liste. Les nombres non barrés constituent la liste des nombres premiers inférieurs ou égaux à n . Avant d'écrire plus précisément cet algorithme, il nous faut choisir une représentation informatique qui traduit le fait qu'un nombre soit barré ou non. Pour cela on peut utiliser un tableau t de n éléments indexés de 1 jusqu'à n . Les composantes de ce tableau pouvant avoir soit une valeur booléenne (**faux**, **vrai**), soit une des deux valeurs 0 ou 1. Ainsi la valeur de la composante de rang i nous indiquera si le nombre i est premier ou non : la valeur **vrai** ou 1 signifie oui, la valeur **faux** ou 0 signifiant non.

Algorithme 1 (Crible d'Erathostène).

Entrée : Un nombre entier naturel n .

Sortie : La liste L des nombres premiers inférieurs ou égaux à n .

```

t ← un tableau de taille n
pour i variant de 1 jusqu'à n faire
    | t[i] ← vrai
t[1] ← faux
pour i variant de 2 jusqu'à n faire
    | si t[i] alors # On barre les multiples de i
        | k ← 2
        | tant que k * i ≤ n faire
            | t[k * i] ← faux
            | k ← k + 1
L ← []
pour i variant de 2 jusqu'à n faire
    | si t[i] alors
        | L ← L, i
résultat L

```

On peut écrire à partir de cet algorithme une version un peu plus performante en tenant compte des remarques suivantes :

- On peut passer d'un multiple m de i au multiple suivant en y ajoutant i , une addition est plus rapide qu'une multiplication.
- Il est inutile d'examiner les multiples m de i inférieurs à i^2 , car ils ont été déjà barrés.
- Il est inutile de chercher à barrer des nombres plus grands que \sqrt{n} , car tous ceux qui ne sont pas premiers l'ont déjà été. En effet un nombre plus grand que \sqrt{n} qui n'est pas premier a forcément un facteur premier plus petit que \sqrt{n} , donc il aura été barré lorsque les multiples de ce facteur l'ont été.

Ce qui conduit à écrire l'algorithme suivant :

Algorithme 2 (Crible d'Erathostène, version optimisée).

```

 $t \leftarrow$  un tableau de taille  $n$ 
pour  $i$  variant de 1 jusqu'à  $n$  faire
   $t[i] \leftarrow$  vrai
 $t[1] \leftarrow$  faux
pour  $i$  variant de 2 jusqu'à  $\sqrt{n}$  faire
  si  $t[i]$  alors # On barre les multiples de  $i$ 
     $m \leftarrow i^2$ 
    tant que  $m \leq n$  faire
       $t[m] \leftarrow$  faux
       $m \leftarrow m + i$ 
 $L \leftarrow []$ 
pour  $i$  variant de 2 jusqu'à  $n$  faire
  si  $t[i]$  alors
     $L \leftarrow L, i$ 
résultat  $L$ 

```

1.3.2 Décomposition d'un entier n en produit de facteurs premiers

On sait qu'un nombre entier se décompose de manière unique en un produit de facteurs premiers. Le principe de la décomposition est très simple. Il consiste à essayer de diviser n par les nombres premiers successifs. Lorsqu'un nombre premier p divise n , il faut continuer à diviser n par ce nombre tant que cela est possible afin de connaître l'exposant de p . Voici l'algorithme :

Algorithme 3 (Décomposition d'un entier n en produits de facteurs premiers).

Entrée : Un nombre entier naturel n .

Sortie : La liste des nombres premiers p_1, p_2, \dots, p_r et de leur exposant respectif e_1, e_2, \dots, e_r tel que n soit égal à $p_1^{e_1} p_2^{e_2} \dots p_r^{e_r}$.

```

P ← la liste des nombres premiers inférieurs ou égaux à n
F ← [] # On initialise la liste des facteurs F
E ← [] # On initialise la liste des exposants E
i ← 1
tant que n ≠ 1 faire
    si p[i] divise n alors
        k ← 0
        tant que p[i] divise n faire
            n ← n/p[i]
            k ← k + 1
        F ← F, p[i] # pi est un facteur premier on l'ajoute à la liste F
        E ← E, k # On ajoute son exposant k à la liste E des exposants
    i ← i + 1
résultat F, E

```

1.3.3 Décomposition en base b d'un entier a

De façon générale, l'écriture d'un nombre a en base b est définie par la formule

$$(a_k a_{k-1} \dots a_1 a_0)_b = a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b + a_0 \quad (1.1)$$

où les coefficients a_i sont des nombres entiers vérifiant $0 \leq a_i < b$. Le système décimal étant le cas particulier où b est égal à 10, les coefficients a_i prenant pour valeur un des dix chiffres 0, 1, 2, ..., 9.

Décomposer un nombre a en base b revient à trouver les $k + 1$ coefficients a_i , $0 \leq a_i < b$, tels que $a = a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b + a_0$. Pour cela il suffit de savoir faire la division euclidienne de a par b . En effet, on obtient facilement le coefficient de poids le plus faible a_0 en calculant le reste de la division de a par b , puis on considère le nombre entier a' , quotient de a par b ,

$$a' = a \text{ div } b = a_k b^{k-1} + a_{k-1} b^{k-2} + \dots + a_2 b + a_1 \quad (1.2)$$

Le reste de la division euclidienne de a' par b nous donne a_1 , et ainsi de suite pour obtenir les autres coefficients de l'écriture binaire de a .

Algorithme 4 (Décomposition d'un entier n en base b).

Entrée : Un nombre entier naturel n .

Sortie : La liste L des coefficients a_i de l'écriture de n en base b .

```

a ← n
L ← []
tant que a ≠ 0 faire
    | r ← a mod b
    | L ← r, L # On ajoute le reste r en tête de la liste L
    | a ← a div b
résultat L

```

1.3.4 Evaluation d'un nombre écrit en base b

Soit $(a_k a_{k-1} \dots a_1 a_0)_b$ l'écriture en base b du nombre que l'on veut évaluer en nombre entier. Pour obtenir cette conversion, il suffit d'évaluer $a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b + a_0$.

On suppose dans les algorithmes présentés ci-dessous que les coefficients de l'écriture en base b sont rangés dans un tableau.

Algorithme 5 (Evaluation d'un nombre écrit en base b).

Entrée : Un nombre $(a_k a_{k-1} \dots a_1 a_0)_b$ écrit en base b .

Sortie : L'entier n qui correspond à l'évaluation de $(a_k a_{k-1} \dots a_1 a_0)_b$.

```

n ← 0
pour i variant de 0 jusqu'à k faire
    | n ← n + a[i] × bi
résultat n

```

L'algorithme précédent n'est pas performant car il effectue un trop grand nombre de multiplications. En effet pour calculer b^i , il est nécessaire de faire $i - 1$ multiplications, ainsi à l'étape i on effectue i multiplications, soit au total $1 + 2 + \dots + k = \frac{k \times (k+1)}{2}$ multiplications. C'est beaucoup trop !

Voici un algorithme qui n'en effectue plus que $2(k + 1)$ pour calculer la même expression.

Algorithme 6 (Evaluation d'un nombre écrit en base b).

Entrée : Un nombre $(a_k a_{k-1} \dots a_1 a_0)_b$ écrit en base b .

Sortie : L'entier n qui correspond à l'évaluation de $(a_k a_{k-1} \dots a_1 a_0)_b$.

```

f ← 1
n ← 0
pour i variant de 0 jusqu'à k faire
    | n ← n + a[i] × f
    | f ← f × b
résultat n

```

On peut encore diminuer le nombre de multiplications en utilisant l'algorithme de Hörner présenté ci-après.

1.3.5 Développements décimaux illimités

Présentation

Un nombre rationnel peut être représenté soit par son écriture fractionnaire $Q = \frac{N}{D}$ où N est un entier relatif et D un entier naturel non nul ou par un développement décimal illimité périodique de la forme :

$Q = 0, \underbrace{r_1 r_2 \cdots r_i}_{\text{Partie régulière}} \overbrace{p_1 p_2 \cdots p_j}^{\text{Partie périodique}} \times 10^n$ où les r_k et les p_k sont des chiffres du système décimal et n un entier naturel.

Obtention de la fraction à partir du développement

Si on pose $a = 0, r_1 r_2 \cdots r_i$ et $b = 0, \overline{p_1 p_2 \cdots p_j}$ on a alors :

$$Q = (a + b \times 10^{-i}) \times 10^n$$

$$\text{avec } a = \frac{r_1 r_2 \cdots r_i}{10^i} \text{ et } 10^j \times b = p_1 p_2 \cdots p_j, \overline{p_1 p_2 \cdots p_j} = p_1 p_2 \cdots p_j + b$$

$$\text{d'où } b = \frac{p_1 p_2 \cdots p_j}{10^j - 1}$$

$$\text{donc } Q = \left(\frac{r_1 r_2 \cdots r_i}{10^i} + p_1 p_2 \cdots p_j \times \frac{10^{-i}}{10^j - 1} \right) \times 10^n$$

$$Q = \frac{(r_1 r_2 \cdots r_i) \times (10^j - 1) + (p_1 p_2 \cdots p_j)}{10^i \times (10^j - 1)} \times 10^n$$

Il suffit maintenant de rendre cette fraction irréductible.

Il n'y a donc pas d'algorithme de calcul de Q sous forme de fraction mais simplement une méthode mathématique.

Obtention du développement à partir de la fraction

$Q = \frac{N}{D} = E + \frac{N'}{D}$ où E est la partie entière de Q et $\frac{N'}{D}$ sa partie décimale (illimitée)

On aura donc :

$$\frac{N'}{D} = 0, \underbrace{r_1 r_2 \cdots r_i}_{\text{Partie régulière}} \overbrace{p_1 p_2 \cdots p_j}^{\text{Partie périodique}}$$

Pour déterminer les suites de décimales r_k et p_k ainsi que i et j on construit les suites (N_k) , (Q_k) et (R_k) de la façon suivante :

$$- R_k = N_k \text{ mod } D \text{ (reste euclidien de } N_k \text{ par } D)$$

- $Q_k = \frac{N_k - R_k}{D}$ (quotient euclidien de N_k par D)
- $N_{k+1} = 10 \times R_k$ avec $R_0 = N'$

Remarques

- a) R_k peut prendre D valeurs entières comprises entre 0 et $D - 1$
- b) Si $R_k = 0$ alors tous les restes suivants sont nuls
- c) Il en découle que $i + j < D$

Calcul des décimales

Il faut vérifier que les quotients calculés sont bien les décimales du développement cherché

Procédons par récurrence sur k

On a bien $N_1 = 10 \times N'$ et $Q_1 = \text{partie entière} \left(\frac{N_1}{D} \right) = r_1$ puisque $\frac{10 \times N'}{D} = r_1, r_2 \dots$

Si on suppose qu'au rang k , $\frac{N_k}{D} = r_k, r_{k+1}r_{k+2} \dots$ alors

$$Q_k = \text{partie entière} \left(\frac{N_k}{D} \right) = r_k \text{ et } \frac{N_{k+1}}{D} = \frac{10 \times R_k}{D} = \frac{10 \times (N_k - D \times Q_k)}{D} =$$

$$10 \left(\frac{N_k}{D} - Q_k \right) = 10(r_k, r_{k+1}r_{k+2} \dots - r_k) = 10 \times (0, r_{k+1}r_{k+2} \dots) = r_{k+1}, r_{k+2} \dots$$

donc $Q_{k+1} = \text{partie entière} \left(\frac{N_{k+1}}{D} \right) = r_{k+1}$

Conclusion : les Q_k sont les chiffres du développement décimal illimité associé à $\frac{N'}{D}$.

Calcul de i et j

D'après la remarque a) à partir d'un certain rang on va retrouver un des restes précédents.

Soit i et j les plus petits entiers tels que $R_{i+j} = R_i$. On a donc $Q_{i+j+1} = Q_{i+1}$ et $p_{j+1} = r_{i+j+1} = Q_{i+j+1} = Q_{i+1} = r_{i+1} = p_1$ ce qui prouve que la période du développement décimal est j .

Algorithme 7 (Développement décimal d'un nombre rationnel $\frac{N}{D}$).

Entrée : Un nombre rationnel sous la forme $\frac{N}{D}$.

Sortie : Le quotient entier e de $\frac{N}{D}$, la liste *NonPeriodique* des chiffres de la partie non périodique, la liste *Periodique* des chiffres de la partie périodique.

```

num ← N
den ← D
# TabReste[i] indique la position où le reste i a été obtenu
pour i variant de 0 jusqu'à den - 1 faire TabReste[i] ← 0
e ← num div den
r ← num mod den
# Initialisation de la liste des quotients
ListeQuotient ← []
i ← 0
tant que TabReste[r] = 0 faire
    | i ← i + 1
    | TabReste[r] ← i
    | num ← 10 × r
    | q ← num div den
    | ListeQuotient ← ListeQuotient, q
    | r ← num mod den
k ← TabReste[r]
NonPeriodique ← ListeQuotient[0..k - 1]
Periodique ← ListeQuotient[k..i]
résultat e, Nonperiodique, Periodique

```

1.3.6 L'algorithme de Hörner

L'algorithme de Hörner est très connu et très facile à mettre en oeuvre. Soit x un nombre réel et $p(x) = a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$ un polynôme de degré k dont les coefficients sont aussi des réels, a_k distinct de 0. En remarquant qu'on peut écrire le polynôme sous la forme suivante :

$$\begin{aligned}
 p(x) &= (a_k x^{k-1} + a_{k-1} x^{k-2} + \dots + a_2 x + a_1)x + a_0 \\
 p(x) &= ((a_k x^{k-2} + a_{k-1} x^{k-3} + \dots + a_2)x + a_1)x + a_0 \\
 p(x) &= (((a_k x^{k-3} + a_{k-1} x^{k-4} + \dots + a_3)x + a_2)x + a_1)x + a_0
 \end{aligned}$$

.....

$$p(x) = ((\dots((a_k)x + a_{k-1})x + \dots + a_2)x + a_1)x + a_0$$

Ainsi pour calculer $p(x)$, on organise les calculs de manière à calculer successivement les valeurs b_k, b_{k-1}, \dots, b_0 définies par :

$$\begin{aligned}
 b_k &= a_k \\
 b_{k-1} &= b_k * x + a_{k-1} \\
 &\dots \\
 b_i &= b_{i+1}x + a_i \\
 &\dots \\
 b_0 &= b_1x + a_0
 \end{aligned}$$

On a donc $b_0 = p(x)$, c'est ce procédé qui est parfois appelé "schéma de Hörner".

Algorithme 8 (Schéma de Hörner).

Entrées : Un tableau a de $k + 1$ coefficients indexés de 0 à k et un nombre réel x .

Sortie : Le nombre $s = \sum_{i=0}^k a_i x^i$.

```

s ← 0
pour i variant de k jusqu'à 0 avec un pas de -1 faire
    | s ← s × x + a[i]
résultat s

```

Schéma de Hörner appliqué à l'évaluation d'un nombre écrit en base b

Cet algorithme n'effectue plus que $k + 1$ multiplications pour évaluer l'expression polynomiale $a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b + a_0$.

Algorithme 9 (Evaluation d'un nombre écrit en base b).

Entrée : Un nombre $(a_k a_{k-1} \dots a_1 a_0)_b$ écrit en base b .

Sortie : Le nombre n qui correspond à sa conversion en base 10.

```

n ← 0
pour i variant de k jusqu'à 0 avec un pas de -1 faire
    | n ← n × b + a[i]
résultat n

```

Construction d'un nombre décimal n inférieur à 1 à partir de la liste a de ses chiffres

Cet algorithme est encore une application directe de l'algorithme de Hörner.

En effet si la liste a est $[a_1, a_2, \dots, a_{k-1}, a_k]$, le nombre n correspondant est :

$$n = a_k 10^{-k} + a_{k-1} 10^{-(k-1)} + \dots + a_1 10^{-1}$$

Ce qui conduit à écrire l'algorithme suivant :

Algorithme 10 (Schéma de Hörner appliqué à la construction d'un nombre décimal plus petit que 1).

Entrée : Un tableau de chiffres a indexé de 1 à k .

Sortie : Un nombre décimal $n = \sum_{i=1}^k a_i 10^{-i}$ inférieur à 1.

```

n ← 0
pour i variant de k jusqu'à 1 avec un pas de -1 faire
    | n ←  $\frac{n}{10} + a_i$ 
résultat n

```

Génération d'un nombre entier aléatoire n dont la décomposition binaire comporte exactement k bits

Cet algorithme est encore une application directe de l'algorithme de Hörner. En effet il suffit que le bit de poids fort soit égal à 1 et de choisir aléatoirement entre 0 et 1 pour les $k - 1$ autres bits. On évaluera au fur et à mesure l'entier décimal correspondant, exactement comme dans l'algorithme "Hörner" dans le cas où la base est 2. Voici l'algorithme correspondant :

Algorithme 11 (Génération d'un nombre entier aléatoire n dont la décomposition binaire comporte exactement k bits).

Entrée : k un nombre entier.

Sortie : Un nombre entier aléatoire n écrit en décimal dont la décomposition binaire comporte exactement k bits.

```

 $n \leftarrow 1$ 
pour  $i$  variant de 1 jusqu'à  $k$  faire
  |  $n \leftarrow 2 \times n + \text{hasard}(2)$ 
résultat  $n$ 

```

où *hasard* est une fonction qui renvoie pour résultat un nombre entier aléatoire supérieur ou égal à 0 et strictement plus petit que son argument.

1.3.7 Algorithme d'exponentiation rapide

Pour x nombre réel et n entier naturel, on veut calculer x^n de manière efficace, c'est à dire en minimisant le nombre d'opérations arithmétiques. En considérant l'écriture en base de 2 de l'entier $n = \sum_{i=0}^k a_i 2^i$ où les coefficients a_i valent 0 ou 1, on a :

$$x^n = x^{\sum_{i=0}^k a_i 2^i} = \prod_{i=0}^k x^{a_i 2^i}$$

On constate que les facteurs qui interviennent dans le produit sont soit 1 si a_i vaut 0, soit de la forme x^{2^i} si a_i vaut 1. Ainsi pour calculer x^n il suffit de décomposer n en base 2 et de ne retenir dans le produit que les facteurs qui correspondent à des a_i valant 1. Tout cela peut se faire au fur et à mesure de la décomposition de n en calculant le nouveau facteur. Il suffit à chaque étape d'élever le facteur précédent au carré. On obtient l'algorithme :

Algorithme 12 (Exponentiation rapide).

Entrée : Un nombre entier naturel n et un nombre réel x

Sortie : x^n

```

 $f \leftarrow x$ 
 $p \leftarrow 1$ 
tant que  $n \neq 0$  faire
    | si  $n \bmod 2 = 1$  alors
    | |  $p \leftarrow p \times f$ 
    | |  $f \leftarrow f \times f$ 
    | |  $n \leftarrow n \text{ div } 2$ 
résultat  $p$ 

```

Cet algorithme devient très intéressant lorsque n est très grand, en effet le nombre d'itérations est égal à $k = \lfloor \log_2(n) \rfloor + 1$.

1.3.8 Un principe fondamental : la dichotomie

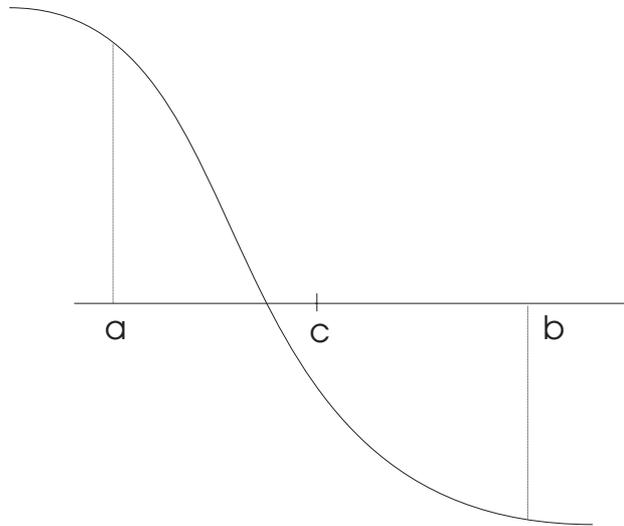
Le mot dichotomie provient du grec di (deux) et tomein (couper), ce qui signifie littéralement " couper en deux ". Ce principe très efficace et relativement facile à mettre en œuvre intervient dans de nombreux algorithmes :

- Calcul des zéros d'une fonction
- Algorithmes de recherche
- Algorithmes de classement
- ...

Sa mise en œuvre permet d'aborder et d'illustrer les problèmes fondamentaux que nous avons évoqués précédemment.

Méthode de dichotomie pour le calcul d'un zéro d'une fonction

Soit f une fonction continue sur un intervalle $[a, b]$, qui change de signe entre a et b . Le principe de dichotomie permet en divisant à chaque étape l'intervalle de moitié, en considérant le milieu c , de trouver une valeur approchée d'une racine de f à ϵ près. Pour atteindre la précision ϵ , arbitrairement petite, il suffit d'itérer ce processus n fois, où n est le plus petit entier tel $\frac{|a-b|}{2^n} \leq \epsilon$. On peut aussi dire que n est le plus petit entier supérieur ou égal à $\log_2\left(\frac{|a-b|}{\epsilon}\right)$.



À chaque étape, il faut veiller à ce que la fonction f ait une racine sur le nouvel intervalle. Cette condition, indispensable pour la validité de l'algorithme n'est pas toujours respectée, et bon nombre de versions que l'on peut rencontrer dans la littérature peuvent dans certains cas donner des résultats erronés. On rencontre aussi, très souvent, une erreur dans la condition d'arrêt : le test n'est pas fait sur la comparaison de la longueur de l'intervalle avec ϵ , mais sur la comparaison $|f(c)| \leq \epsilon$, où c est un point de l'intervalle qui contient une racine exacte. Clairement, $|f(c)| \leq \epsilon$ n'implique pas $|x_0 - c| \leq \epsilon$.

Un premier algorithme

Algorithme 13 (Recherche d'une racine par dichotomie).

Entrées : a réel, b réel, $a < b$, f fonction continue sur $[a, b]$ telle que $f(a) * f(b) < 0$, ϵ un nombre réel arbitrairement petit.

Sorties : Un nombre réel qui approche une racine x_0 de f à ϵ près.

Invariant : $(f(a) * f(b) < 0)$ ou $(a = b \text{ et } f(a) = 0)$.

```

tant que  $b - a > \epsilon$  faire
  |
  |  $c \leftarrow \frac{a + b}{2}$ 
  | si  $f(c) = 0$  alors
  |   |
  |   |  $a \leftarrow c$ 
  |   |  $b \leftarrow c$ 
  |   |
  |   | sinon
  |   |   |
  |   |   | si  $f(a) * f(c) < 0$  alors
  |   |   |   |
  |   |   |   |  $b \leftarrow c$ 
  |   |   |   |
  |   |   |   | sinon
  |   |   |   |   |
  |   |   |   |   |  $a \leftarrow c$ 
résultat  $a$ 

```

Dans la pratique, il est peu probable que la variable c prenne pour valeur une des racines de la fonction f , ainsi le test $f(c) = 0$ devient inutile et nuit à l'efficacité de ce premier algorithme. En effet les valeurs données aux bornes de l'intervalle de départ sont très souvent des nombres entiers alors que les racines de f sont généralement des nombres irrationnels.

Un deuxième algorithme

Algorithme 14 (Recherche d'une racine par dichotomie, 2^{ème} version).

Entrées : a réel, b réel, $a < b$, f fonction continue sur $[a, b]$ telle que $f(a) * f(b) \leq 0$, ϵ un nombre réel arbitrairement petit.

Sortie : Un nombre réel qui approche une racine de f à ϵ près.

Invariant : $f(a) * f(b) \leq 0$

```

tant que  $b - a > \epsilon$  faire
  |
  |  $c \leftarrow \frac{a + b}{2}$ 
  | si  $f(a) * f(c) \leq 0$  alors
  |   |
  |   |  $b \leftarrow c$ 
  |   |
  |   | sinon
  |   |   |
  |   |   |  $a \leftarrow c$ 
résultat  $a$ 

```

On aurait pu aussi écrire autrement le corps de la boucle en gardant le même invariant :

Algorithme 15. Autre version de l'algorithme 14.

```

tant que  $b - a > \epsilon$  faire
  |  $c \leftarrow \frac{a + b}{2}$ 
  | si  $f(a) * f(c) > 0$  alors
  |   |  $a \leftarrow c$ 
  | sinon
  |   |  $b \leftarrow c$ 
résultat  $a$ 

```

Mais la version qui suit, bien que très proche, peut s'avérer erronée :

Algorithme 16. Version erronée de l'algorithme 14.

```

tant que  $b - a > \epsilon$  faire
  |  $c \leftarrow \frac{a + b}{2}$ 
  | si  $f(a) * f(c) < 0$  alors
  |   |  $b \leftarrow c$ 
  | sinon
  |   |  $a \leftarrow c$ 
résultat  $a$ 

```

En effet, l'invariant n'est plus satisfait dans le cas où le milieu de l'intervalle c est par "malchance" une racine de la fonction f . Ce cas, peu fréquent, comme on l'a déjà dit, peut néanmoins se produire, et l'algorithme ci-dessus fournit alors un résultat faux ! Dans cette configuration, le produit $f(a) * f(c)$ est nul, a prend donc la valeur c , et par la suite le produit $f(a) * f(c)$ n'étant jamais négatif, on trouve comme valeur approchée de la racine un point proche de b à ϵ près ! À chaque itération les algorithmes précédents évaluent deux fois la fonction f et effectuent un produit, on peut être plus efficace en écrivant les choses différemment.

Un troisième algorithme

Algorithme 17 (Recherche d'une racine par dichotomie, 3^{ème} version).

Entrées : a réel, b réel, f fonction continue sur $[a, b]$ telle que $f(a) \leq 0$ et $f(b) > 0$, ϵ un nombre réel arbitrairement petit.

Sortie : Un nombre réel qui approche une racine de f à ϵ près.

Invariant : $(f(a) \leq 0$ et $f(b) > 0$

```

tant que  $b - a > \epsilon$  faire
  |  $c \leftarrow \frac{a + b}{2}$ 
  | si  $f(c) \leq 0$  alors
  |   |  $a \leftarrow c$ 
  |   | sinon
  |     |  $b \leftarrow c$ 
résultat  $a$ 

```

1.4 Illustration des notions de preuve et de terminaison d'un algorithme

Ces notions sont illustrées à travers cinq exemples. On présente tout d'abord trois algorithmes qui permettent de calculer le produit $a \times b$ de nombres entiers naturels a et b , puis le très connu algorithme d'Euclide pour calculer le PGCD de deux nombres entiers et enfin un algorithme un peu moins connu permettant de calculer le quotient et le reste de la division euclidienne de deux nombres entiers. On constatera ici, surtout sur le dernier exemple présenté, qu'il n'est pas du tout évident de deviner ce que calcule un algorithme en le faisant "tourner à la main". Même si dans le cas présent on arrive (difficilement) à dire qu'il calcule le reste et le quotient de la division euclidienne à chaque essai, cela ne constitue en aucun cas une preuve. En revanche, la preuve de sa validité par l'utilisation de l'invariant nous convainc parfaitement.

1.4.1 Algorithme de multiplication : Un premier algorithme

Algorithme 18. Ce premier algorithme calcule le produit de deux nombres entiers ($a \times b$) en n'effectuant que des additions.

Entrées : Deux nombres entiers naturels a et b .

Sortie : Le produit $a \times b$.

```

 $x \leftarrow a$ 
 $y \leftarrow b$ 
 $z \leftarrow 0$ 
tant que  $y \neq 0$  faire
  |  $z \leftarrow z + x$ 
  |  $y \leftarrow y - 1$ 
résultat  $z$ 

```

Terminaison : y étant décrémenté de 1 à chaque itération, il deviendra nécessairement nul au bout de b itérations.

Validité : Il suffit d'établir que $z + x \times y$ reste égal à $a \times b$ à chaque itération. On dit que cette quantité est invariante. C'est l'invariant qui caractérise la boucle. Cet invariant joint à la condition d'arrêt ($y = 0$) prouve que la variable z vaut $a \times b$ à la sortie de la boucle.

- C'est trivialement vrai avant la boucle.
- Supposons que $z + x \times y$ est égal à $a \times b$ au début de l'itération, montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x' , y' et z' les valeurs respectives de x , y et z à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x , y et z pour l'itération suivante. On a $x' = x$, $y' = y - 1$ et $z' = z + x$.

$$\begin{aligned} z' + x' \times y' &= z + x + x \times (y - 1) \\ &= z + x \times y \\ &= a \times b \text{ par hypothèse.} \end{aligned}$$

1.4.2 Algorithme de multiplication : Un deuxième algorithme

Algorithme 19. Ce second algorithme calcule le produit de deux nombres entiers en n'effectuant que des multiplications et divisions par 2.

Entrées : Deux nombres entiers naturels a et b .

Sortie : Le produit $a \times b$.

```

x ← a
y ← b
z ← 0
tant que y ≠ 0 faire
    si y impair alors
        | z ← z + x
        | x ← 2 × x
        | y ← y div 2
résultat z
    
```

$a \text{ div } b$ et $a \text{ mod } b$ désignent respectivement le quotient et le reste de la division euclidienne de a par b .

Terminaison : L'entier y étant divisé par 2 à chaque itération, il deviendra nécessairement nul après un nombre fini d'itérations.

Validité : Il suffit d'établir que $z + x \times y$ reste en permanence égal à $a \times b$. Cet invariant joint à la condition d'arrêt ($y = 0$) prouve, ici encore, que la variable z vaut $a \times b$ à la sortie de la boucle.

- C'est trivialement vrai avant la boucle.

- Supposons que $z + x \times y$ est égal à $a \times b$ au début de l'itération, montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x' , y' et z' les valeurs respectives de x , y et z à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x , y et z pour l'itération suivante. Deux cas sont à envisager :
 - y pair
 - On a $x' = 2 \times x$, $y' = y/2$ et $z' = z$.
 - $z' + x' \times y' = z + (2 \times x) \times (y/2)$
 - $= z + x \times y$
 - $= a \times b$ par hypothèse de récurrence .
 - y impair
 - On a $x' = 2 \times x$, $y' = (y - 1)/2$ et $z' = z + x$.
 - $z' + x' \times y' = z + x + (2 \times x) \times ((y - 1)/2)$
 - $= z + x + x \times (y - 1)$
 - $= z + x \times y$
 - $= a \times b$ par hypothèse.

Cet algorithme est connu sous divers noms : multiplication russe, multiplication égyptienne, ... Notons que son codage en informatique donne un algorithme très efficace car multiplications et divisions par 2 consistent en des décalages, et ces opérations élémentaires dans tout langage machine sont effectuées très rapidement.

On peut donner une justification de sa validité autre que celle utilisant la notion d'invariant.

Il suffit de considérer l'écriture binaire de y , $y = \sum_{i=0}^k y_i 2^i$ où les coefficients y_i valent 0 ou 1. Le produit $x \times y$ est donc égal à $\sum_{i=0}^k y_i \times (x \times 2^i)$. Tous les termes intervenant dans la somme sont de la forme $x \times 2^i$ et correspondent à des coefficients y_i non nuls, i.e aux valeurs impaires que prend la variable y dans l'algorithme 2.

Dans la pratique on organise les calculs en faisant deux colonnes, une contenant respectivement les valeurs des quotients successifs de y par 2 et l'autre les valeurs de la forme $x \times 2^i$. Il suffit alors de sommer les éléments de la deuxième colonne (en gras dans l'exemple ci-dessous) qui sont en face d'éléments impairs de la première colonne.

Exemple :

Soit à calculer le produit de $a= 12$ par $b= 22$.

Calcul en décimal		Calcul en binaire	
y	x	y	x
22	12	10110	1100
11	24	1011	11000
5	48	101	110000
2	96	10	1100000
1	192	1	11000000

On trouve que le résultat de 12×22 est égal à $24 + 48 + 192 = 264$
 En binaire, $11000 + 110000 + 11000000 = 100001000$.

1.4.3 Algorithme de multiplication : Un troisième algorithme

Algorithme 20. Ce troisième algorithme est celui que l'on apprend à l'école primaire, il utilise des multiplications et divisions par 10 (système décimal) et l'algorithme de multiplication d'un nombre entier quelconque par un nombre ayant un seul chiffre (utilisation des tables de multiplication).

Entrées : Deux nombres entiers naturels a et b .

Sortie : Le produit $a \times b$.

```

 $x \leftarrow a$ 
 $y \leftarrow b$ 
 $z \leftarrow 0$ 
tant que  $y \neq 0$  faire
    |  $z \leftarrow z + x \times y \bmod 10$ 
    |  $x \leftarrow 10 \times x$ 
    |  $y \leftarrow y \text{ div } 10$ 
résultat  $z$ 
```

Terminaison : L'entier y étant divisé par 10 à chaque itération, il deviendra nécessairement nul après un nombre fini d'itérations.

Validité : Il suffit d'établir que $z + x \times y$ reste en permanence égal à $a \times b$. Cet invariant joint à la condition d'arrêt ($y = 0$) prouve, ici encore, que la variable z vaut $a \times b$ à la sortie de la boucle.

- C'est trivialement vrai avant la boucle.
- Supposons que $z + x \times y$ est égal à $a \times b$ au début de l'itération, montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x' , y' et z' les valeurs respectives de x , y et z à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x, y et z

pour l'itération suivante. On a $x' = 10 \times x$, $y' = y \text{ div } 10$ et $z' = z + x \times (y \text{ mod } 10)$.

$$\begin{aligned} z' + x' \times y' &= z + x \times (y \text{ mod } 10) + (10 \times x) \times (y \text{ div } 10) \\ &= z + x \times (y \text{ mod } 10 + 10 \times (y \text{ div } 10)) \\ &= z + x \times y \\ &= a \times b \text{ par hypothèse.} \end{aligned}$$

1.4.4 Algorithme d'Euclide pour le calcul du PGCD de nombres entiers

Algorithme 21 (Euclide).

Entrées : Deux nombres entiers naturels a et b .

Sortie : Le *pgcd* de a et de b .

```

x ← a
y ← b
tant que y ≠ 0 faire
    r ← x mod y
    x ← y
    y ← r
résultat x

```

Le principe de l'algorithme d'Euclide s'appuie sur les deux propriétés suivantes :

- Le *pgcd*($a, 0$) est a .
- Le *pgcd*(a, b) est égal au *pgcd*(b, r) où r est le reste de la division euclidienne de a par b .

Cette dernière propriété est très facile à établir. On démontre que l'ensemble des diviseurs de a et b est le même que l'ensemble des diviseurs de b et r . Il est évident que tout nombre divisant a et b divise r , car $r = a - b \times q$ et réciproquement, tout nombre divisant b et r divise aussi a , car $a = r + b \times q$.

Pour démontrer que cet algorithme calcule bien le *pgcd* de a et b , il faut démontrer que cet algorithme fournit dans tous les cas une réponse (**terminaison** de l'algorithme) et que cette réponse est bien le *pgcd* de a et b (**validité** de l'algorithme).

Terminaison : À chaque itération y est affecté avec le reste de la division euclidienne de x par y . On sait, par définition de la division euclidienne, que ce reste r est strictement inférieur au diviseur y . Ainsi à chaque itération y , entier naturel, diminue strictement. Il existe donc une étape où y recevra la valeur 0, permettant ainsi à la boucle de se terminer.

1.4 Illustration des notions de preuve et de terminaison d'un algorithme 35

Validité : Il suffit d'établir que le $pgcd(x, y)$ reste en permanence égal au $pgcd(a, b)$. On dit que cette quantité est invariante. C'est l'invariant qui caractérise la boucle. Plus précisément l'invariant est la proposition logique $pgcd(x, y) = pgcd(a, b)$.

- C'est trivialement vrai avant la boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. Pour cela nous noterons x' et y' les valeurs de x et y à la fin d'une itération. Bien évidemment, ces valeurs deviendront les nouvelles valeurs de x et y pour l'itération suivante.
On a x' qui est égal à y et y' qui est égal à r (r reste de la division euclidienne de x par y), ainsi le $pgcd(x', y')$ est égal au $pgcd(y, r)$ qui est égal au $pgcd(x, y)$ (propriété 2 ci-dessus), or par hypothèse de récurrence on a $pgcd(x, y) = pgcd(a, b)$. Ainsi $pgcd(x', y') = pgcd(a, b)$.

L'invariant joint à la condition d'arrêt ($y = 0$) prouve qu'à la sortie de la boucle x est égal au $pgcd(a, b)$.

Cette façon d'établir la validité de l'algorithme d'Euclide paraît pour le moins aussi simple que la démonstration qui introduit des suites indexées.

1.4.5 Algorithme d'Euclide étendu

Algorithme 22 (Euclide étendu).

Entrées : Deux nombres entiers naturels a et b .

Sorties : Cet algorithme calcule leur $pgcd$, ainsi que deux entiers u et v tels que $a \times u + b \times v = pgcd(a, b)$. A la fin de cet algorithme, x est le $pgcd$ de a et b , car si l'on examine les lignes qui ne concernent que x et y (en gras) on retrouve exactement l'algorithme d'Euclide.

```

x ← a
y ← b
u ← 1
v ← 0
u1 ← 0
v1 ← 1
tant que y ≠ 0 faire
    q ← x div y
    r ← x mod y
    x ← y
    y ← r
    aux ← u
    u ← u1
    u1 ← aux - q × u1
    aux ← v
    v ← v1
    v1 ← aux - q × v1
résultat x, u, v

```

L'invariant de la boucle est la conjonction des deux égalités :

$$a \times u + b \times v = x \text{ et } a \times u_1 + b \times v_1 = y$$

Ceci est trivialement vrai avant la boucle.

Notons $x', y', u', v', u'_1, v'_1$ les valeurs respectives de x, y, u, v, u_1, v_1 à la fin d'une itération. Les valeurs ayant un prime devenant les nouvelles valeurs au début de l'itération suivante. Ainsi par hypothèse de récurrence on a $a \times u + b \times v = x$ et $a \times u_1 + b \times v_1 = y$. Montrons qu'à la fin d'une itération les égalités

$$a \times u' + b \times v' = x' \tag{1.3}$$

et

$$a \times u'_1 + b \times v'_1 = y' \tag{1.4}$$

sont encore satisfaites :

1. $a \times u' + b \times v' = x'$.

Dans la boucle, on a les affectations :

$$u' \leftarrow u_1$$

$$v' \leftarrow v_1$$

$$x' \leftarrow y$$

Ainsi $a \times u' + b \times v' = a \times u_1 + b \times v_1$ qui vaut y par hypothèse, donc x' .

2. $a \times u'_1 + b \times v'_1 = y'$.

Dans la boucle, on a les affectations :

$$u'_1 \leftarrow u - q \times u_1$$

$$v'_1 \leftarrow v - q \times v_1$$

$y' \leftarrow r$, où q et r sont respectivement le quotient et le reste de la division euclidienne de x par y .

$$\begin{aligned} a \times u'_1 + b \times v'_1 &= a \times (u - q \times u_1) + b \times (v - q \times v_1) \\ &= a \times u + b \times v - q \times (a \times u_1 + b \times v_1) \\ &= x - q \times y \text{ par hypothèse de récurrence} \\ &= r \text{ par définition du reste} \\ &= y' \end{aligned}$$

Ainsi, lorsque l'algorithme se termine on a bien : $a \times u + b \times v = x = \text{pgcd}(a, b)$.

1.4.6 Division euclidienne

Algorithme 23. (Division euclidienne) Cet algorithme effectue la division euclidienne de deux nombres entiers a et b en calculant le quotient q et le reste r . Les opérations utilisées sont la soustraction, la multiplication par 2 et la division par 2.

Entrées : Deux nombres entiers naturels a et b , avec $b \neq 0$.

Sorties : Le couple (q, r) où q et r désignent respectivement le quotient et le reste de la division euclidienne de a par b .

```

q ← 0
w ← b
r ← a
tant que w ≤ r faire
    | w ← 2 × w
tant que w ≠ b faire
    | q ← 2 × q
    | w ← w div 2
    | si w ≤ r alors
        | | r ← r - w
        | | q ← q + 1
résultat (q, r)
    
```

Cet algorithme est très performant lorsque la base est une puissance de 2, car multiplications et divisions consistent alors en des décalages. L'invariant de la seconde boucle est la conjonction des deux propriétés $q \times w + r = a$ et $0 \leq r < w$.

Terminaison : w est de la forme $2^p b$ après la première boucle et est divisé par 2 à chaque itération. Il deviendra nécessairement égal à b après p itérations.

Validité : Il suffit d'établir que $q \times w + r = a$ et $0 \leq r < w$ reste vrai à chaque étape.

Cet invariant joint à la condition d'arrêt ($w = b$) prouve, ici encore, que l'on a bien calculé le quotient et le reste de la division euclidienne de a par b .

- C'est vrai avant la seconde boucle. En effet $q \times w + r = a$ car $q = 0$ et $r = a$ et de plus $r < w$ est la condition d'arrêt de la première boucle.
- Montrons que cette propriété reste vraie à la fin de chaque itération. On commence par modifier q et w :

$$q' = 2 \times q \text{ et } w' = \frac{w}{2} \text{ (} w \text{ étant divisible par 2).}$$

Deux cas sont à envisager :

- $r < w'$

$$\text{On a } q' = 2 \times q, w' = \frac{w}{2} \text{ et } r' = r.$$

$$\begin{aligned} q' \times w' + r' &= 2 \times q \times \frac{w}{2} + r \\ &= q \times w + r \\ &= a \text{ par hypothèse.} \end{aligned}$$

Par ailleurs on a $0 \leq r' < w'$

- $w' \leq r$

$$\text{On a } q' = 2 \times q + 1, w' = \frac{w}{2} \text{ et } r' = r - w'.$$

$$\begin{aligned} q' \times w' + r' &= (2 \times q + 1) \times \frac{w}{2} + r - \frac{w}{2} \\ &= q \times w + \frac{w}{2} + r - \frac{w}{2} \\ &= q \times w + r \\ &= a \text{ par hypothèse.} \end{aligned}$$

On est dans le cas où $w' \leq r$ et $r' = r - w'$, ainsi on a $0 \leq r'$. D'autre part par hypothèse de récurrence $r < w$, donc $r - \frac{w}{2} < w - \frac{w}{2} = \frac{w}{2}$, ainsi $r' < w'$.

Ainsi à la fin d'une étape on a encore les inégalités : $0 \leq r' < w'$.

1.5 Construction de programmes itératifs par la mise en évidence de l'invariant

La démarche présentée tente, sans introduire un formalisme excessif, d'apporter des éléments de réponses aux questions fondamentales qui se posent en algorithmique :

- Comment élaborer un algorithme ?
- L'algorithme s'arrête-t-il toujours ?
- Comment prouver qu'un algorithme résout effectivement le problème posé ?

Une des principales difficultés dans l'élaboration d'un algorithme est de contrôler son aspect dynamique (c'est à dire la façon dont il se comporte en fonction des données qu'on lui fournit en entrée), en ne considérant que son aspect statique (on n'a sous les yeux qu'une suite finie d'instructions). La méthode ci-dessous permet d'apporter une aide à la mise au point d'algorithmes itératifs et de contrôler cet aspect dynamique.

L'idée principale de la méthode consiste à raisonner en terme de **situation** alors que l'étudiant débutant, lui, raisonne en terme **d'action**. Par exemple, la description d'un algorithme de tri commence le plus souvent par des phrases du type : "je compare le premier élément avec le second, s'il est plus grand je les échange, puis je compare le second avec le troisième.". On finit par s'y perdre et ce n'est que très rarement que l'algorithme ainsi explicité arrive à être codé correctement. Le fait de raisonner en terme de situation permet, comme on va le voir, de mieux expliciter et de mieux contrôler la construction d'un algorithme. Pour terminer cette introduction, on peut ajouter que cette façon de procéder peut aussi s'avérer utile, dans le cas où un algorithme est connu, pour se convaincre ou convaincre un auditoire que l'algorithme résout bien le problème posé.

1.5.1 La démarche

On sait que ce qui caractérise une itération est son invariant. Trouver l'invariant d'une boucle n'est pas toujours chose aisée. L'idée maîtresse de la méthode est de construire cet invariant parallèlement à l'élaboration de l'algorithme et non pas de concevoir l'algorithme itératif pour ensuite en rechercher l'invariant. Etant donné un problème dont on soupçonne qu'il a une solution itérative, on va s'efforcer de mettre en évidence les étapes suivantes :

- 1 Proposer une situation générale décrivant le problème posé (hypothèse de récurrence). C'est cette étape qui est peut être la plus délicate car elle exige de faire preuve d'imagination. On peut toujours supposer que l'algorithme (on le cherche !) a commencé à "travailler" pour résoudre le problème posé et qu'on l'arrête avant qu'il ait fini : On essaye alors de décrire, de manière très précise, une situation dans laquelle les données qu'il manipule puissent se trouver. Il est possible, comme on le verra sur des exemples, d'imaginer plusieurs situations générales.
- 2 Chercher la condition d'arrêt. A partir de la situation imaginée en [1], on doit formuler la condition qui permet d'affirmer que l'algorithme a terminé son travail. La situation dans laquelle il se trouve alors, est appelée situation finale.
- 3 Se "rapprocher" de la situation finale, tout en faisant le nécessaire pour conserver une situation générale analogue à celle choisie en [1].

- 4 Initialiser les variables introduites dans la description de la situation générale pour que celle-ci soit vraie au départ (c'est à dire avant que l'algorithme ait commencé à travailler).

Une fois cette étude conduite l'algorithme aura la structure suivante :

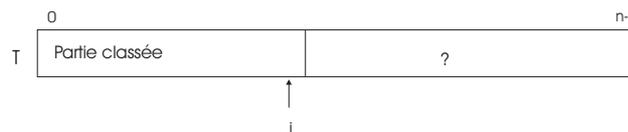
[4]
tant que non [2] faire [3]

Cette façon de procéder montre comment on prouve la validité de l'algorithme au fur et à mesure de son élaboration. En effet la situation générale choisie en [1] est en fait l'invariant qui caractérise la boucle *tant que*. Cette situation est satisfaite au départ à cause de l'étape [4], elle reste vraie à chaque itération (étape [3]). Ainsi lorsque la condition d'arrêt est atteinte cette situation nous permet d'affirmer que le problème est résolu. C'est en analysant l'étape [3] qu'on prouve la terminaison de l'algorithme.

1.5.2 Tri d'un tableau par insertion

Le premier exemple consiste à établir un algorithme permettant de classer suivant l'ordre croissant un tableau de n nombres. Essayons de trouver une situation générale décrivant le problème posé. Pour cela supposons que l'on arrête un algorithme de tri avant que tout le tableau soit classé, on peut imaginer qu'il a commencé à mettre de l'ordre et que par exemple $T[0..i]$ est classé. La notation $T[0..i]$ désigne les $i + 1$ premières composantes du tableau T .

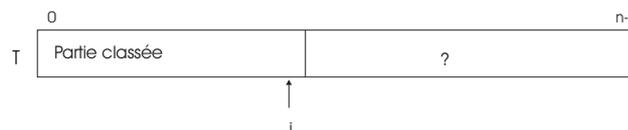
Pour illustrer cette situation on peut soit faire un dessin :



Soit procéder, plus formellement, en décrivant la situation par une formule logique :

$$\{\forall j, j \in [0, i[\Rightarrow T[j] \leq T[j + 1]]\}$$

[1]



[2] L'algorithme a terminé lorsque $i = n - 1$.

[3] Pour se rapprocher de la fin, on incrémente d'abord i de 1, mais si on veut conserver une situation analogue à celle choisie, cela ne suffit pas, car il faut que

le tableau soit classé entre 0 et i . Pour cela "on doit amener $T[i]$ à sa place dans $T[0..i]$ ". Ceci est un autre problème qui sera résolu en utilisant la même démarche.

[4] $T[0..0]$ étant trié, l'initialisation $i \leftarrow 0$ convient.

Cette première analyse nous conduit à écrire la séquence suivante :

```

i ← 0
tant que i ≠ n - 1 faire
    | i ← i + 1
    | "amener T[i] à sa place dans T[0..i]"

```

L'invariant : $\{\forall j, j \in [0, i[\Rightarrow T[j] \leq T[j + 1]]$ joint à la condition d'arrêt $i = n - 1$ s'écrit à la sortie de la boucle :

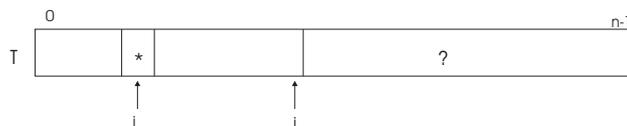
$\{\forall j, j \in [0, n - 1[\Rightarrow T[j] \leq T[j + 1]]$

Ce qui veut précisément dire que tout le tableau est classé. La démonstration de la terminaison de cet algorithme est triviale car i est initialisé avec 0 et chaque fois que l'étape [3] est exécutée i est incrémenté de 1, la condition d'arrêt $i = n - 1$ sera forcément vérifiée au bout de n étapes.

Toute réalisation de "amener $T[i]$ à sa place dans $T[0..i]$ " nous donne un algorithme de tri.

Par hypothèse on sait que le tableau est classé entre 0 et $i - 1$, on suppose qu'un algorithme a commencé à travailler et l'élément qui était initialement en i s'est "rapproché" de sa place par des échanges successifs et se trouve en j lorsqu'on a interrompu l'algorithme. Sur le schéma qui suit, l'élément qui était initialement en i est matérialisé par une étoile.

[1] Avec $T[0..j - 1]$ et $T[j..i]$ classés.



[2] C'est terminé lorsque $j = 0$ ou $T[j - 1] \leq T[j]$.

[3]

Echanger $T[j - 1]$ et $T[j]$

$j \leftarrow j - 1$

[4] L'initialisation $j \leftarrow i$ satisfait la situation choisie en [1].

Ce qui nous conduit finalement à écrire l'algorithme de tri par insertion :

Algorithme 24 (Tri par insertion).

Entrée : Un tableau T de n éléments munis d'une relation d'ordre.

Sortie : Le tableau T trié par ordre croissant.

```

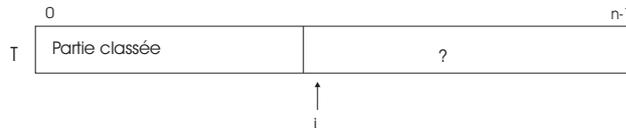
i ← 0
tant que i ≠ n − 1 faire
    | i ← i + 1
    | j ← i
    | tant que j ≠ 0 et  $T[j - 1] > T[j]$  faire
    | | Echanger  $T[j - 1]$  et  $T[j]$ 
    | | j ← j − 1

```

On remarquera que dans cette ultime version les expressions booléennes qui suivent les *tantque* sont les négations des conditions d'arrêt.

On aurait pu raisonner à partir d'une situation générale sensiblement différente de celle choisie :

[1]



La nuance réside dans le fait que la partie triée est $T[0..i - 1]$, et non $T[0..i]$ comme dans la première version. Cette situation est tout aussi acceptable que la précédente. En raisonnant avec celle-ci on obtient un algorithme, certes voisin du précédent, mais comportant plusieurs modifications. La condition d'arrêt devient $i = n$, le corps de l'itération consiste à "amener l'élément qui se trouve en i à sa place dans $T[0..i - 1]$, puis vient l'incrément de i , l'initialisation devient $i ← 1$. Comme on peut le constater la situation choisie guide totalement l'écriture de l'algorithme.

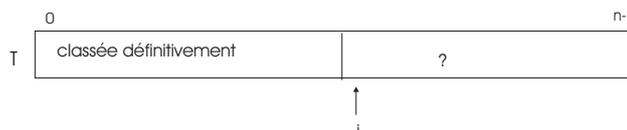
1.5.3 Tri d'un tableau par sélection

Comme il a déjà été dit et constaté la situation générale n'est pas unique. Par exemple, toujours pour le même problème, on peut faire une hypothèse plus forte sur la partie classée en ajoutant qu'elle est aussi définitivement en place. Ce qui se traduit formellement par une conjonction de formules logiques :

$$\begin{aligned}
 & \{ \forall j, j \in [0, i - 1[\Rightarrow T[j] \leq T[j + 1] \} \\
 & \quad \text{et} \\
 & \{ (\forall j, j \in [0, i - 1] \text{ et } \forall k, k \in [i, n - 1]) \Rightarrow T[j] \leq T[k] \}
 \end{aligned}$$

La première formule indique que le tableau est classé jusqu'à $i - 1$, la seconde formule que cette partie est définitivement en place.

[1]



[2] C'est terminé lorsque $i = n - 1$.

[3]

"Rechercher l'indice k du plus petit élément de $T[i..n - 1]$ "

Echanger $T[i]$ et $T[k]$.

Incrémenter i de 1.

[4] L'initialisation $i \leftarrow 0$ convient.

Ce qui nous conduit à écrire l'algorithme suivant :

$i \leftarrow 0$

tant que $i \neq n - 1$ **faire**

"Rechercher l'indice k du plus petit élément de $T[i..n - 1]$ "

Echanger $T[i]$ et $T[k]$

$i \leftarrow i + 1$

Toute réalisation de "rechercher l'indice k du plus petit élément de $T[i..n - 1]$ " conduit à un algorithme de tri. L'analyse de ce sous problème est laissée au soin du lecteur. Cet algorithme se termine car la variable i est initialisée avec 0 et est incrémentée chaque fois que l'étape 3 est exécutée, après n itérations la condition d'arrêt $i = n - 1$ est atteinte. A la sortie de l'itération i vaut donc $n - 1$. Ainsi, en remplaçant i par $n - 1$ dans les formules logiques qui décrivent la situation générale choisie, on obtient précisément la définition du fait que le tableau T est classé, ce qui prouve la validité de notre algorithme.

Revenons un instant sur l'étape [4]. Pourquoi l'initialisation $i \leftarrow 0$ satisfait-elle la situation choisie au départ ? On peut justifier ce choix soit intuitivement, soit formellement.

Intuitivement :

Lorsque i vaut 0, $T[0..i - 1]$ désigne le tableau vide. On peut dire du tableau vide qu'il a toutes les propriétés que l'on veut, puisqu'il ne possède aucun élément. En particulier, on peut dire que ses éléments sont classés et en place.

Formellement :

A ce stade il est nécessaire de faire quelques rappels :

- a) Si P est fausse, l'implication $P \Rightarrow Q$ est vraie.
 b) La propriété $\forall x, x \in \emptyset$ est toujours fausse (\emptyset désigne l'ensemble vide).
 Examinons maintenant la formule logique décrivant la situation générale.

$$\{\forall j, j \in [0, i - 1[\Rightarrow T[j] \leq T[j + 1]\} \\
\text{et} \\
\{(\forall j, j \in [0, i - 1] \text{ et } \forall k, k \in [i, n - 1]) \Rightarrow T[j] \leq T[k]\}$$

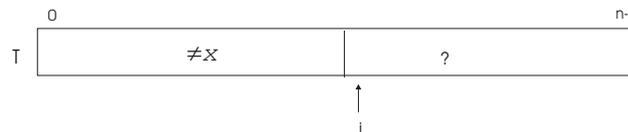
Il faut qu'au départ (i vaut 0) cette formule soit vraie. Pour qu'une conjonction de deux formules soit vraie il suffit que chacune des deux formules soit vraie. Il est facile de constater que la première formule est vraie lorsque i vaut 0. En effet cette formule est de la forme $P \Rightarrow Q$, où P est de la forme $\forall x, x \in \emptyset$, car lorsque i vaut 0 l'intervalle semi-ouvert $[0..i - 1[$ possède zéro élément.

La seconde formule et les rappels a) et b) nous permettent encore de conclure.

1.5.4 La recherche séquentielle

Cette méthode permet de rechercher la première occurrence, si elle existe, d'un élément x dans une table T ayant n éléments. Pour cela on peut imaginer la situation où l'algorithme a commencé à rechercher x parmi les i premiers éléments et ne l'a pas encore trouvé. Plus précisément cela signifie que les i éléments de rang compris entre 0 et $i - 1$ sont tous différents de x .

[1]



[2] C'est terminé lorsque $i = n$ ou $x = T[i]$

[3] $i \leftarrow i + 1$

[4] L'initialisation $i \leftarrow 0$ convient.

L'étape [3] conserve de façon évidente la situation choisie en [1]. D'autre part, à chaque étape, i augmente, la condition d'arrêt sera toujours atteinte. On obtient :

Algorithme 25 (Recherche séquentielle).

Entrée : Un tableau T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement de la première occurrence de x dans T , -1 si x est absent.

```
 $i \leftarrow 0$   
tant que  $i \neq n$  et  $x \neq T[i]$  faire  
  |  $i \leftarrow i + 1$   
si  $i < n$  alors  
  |  $r \leftarrow i$   
sinon  
  |  $r \leftarrow -1$   
résultat  $r$ 
```

Attention, si l'on programme tel quel cet algorithme, le programme ne sera correct que si le langage optimise l'évaluation des expressions booléennes. Plus précisément si le premier opérande d'un **et** (resp. **ou**) est faux (resp. vrai) le second opérande ne sera pas évalué l'expression sera évaluée comme fausse (resp. vraie). En effet, lors du dernier tour de boucle (i vaut n) dans le cas où x n'est pas présent, l'expression booléenne qui suit le **tant que** est évaluée une dernière fois. L'évaluation du premier opérande $i \neq n$ donne **faux**, ce qui implique que l'expression totale a pour valeur **faux**, évitant ainsi l'évaluation du second opérande $x \neq T[i]$. C'est l'évaluation de ce second opérande qui pose problème dans le cas où i vaut n , car les seuls indices permis pour désigner une composante du tableau T sont $0, 1, \dots, n - 1$. L'indice n fait "sortir" du tableau !

Pour certains langages, c'est par une option à la compilation que le programmeur choisit lui même le type d'évaluation, pour d'autres on pas pas le choix, l'évaluation est totale. Dans la plupart des langages de programmation actuels leur évaluation est optimisée. Voici une autre version qui ne pose aucun problème quel que soit le langage choisi :

Algorithme 26 (Recherche séquentielle).

Entrée : Un tableau T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement de la première occurrence de x dans T , -1 si x est absent.

```

trouve ← faux
i ← 0
tant que i ≠ n et non trouve faire
    | si  $x = T[i]$  alors
    | | trouve ← vrai
    | | i ← i + 1
si trouve alors
    | r ← i - 1
sinon
    | r ← -1
résultat r

```

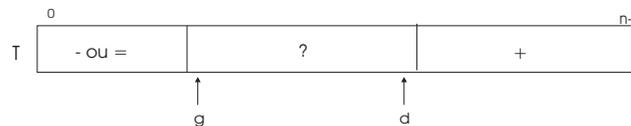
1.5.5 La recherche dichotomique

Soit $T[0..n-1]$ un tableau de n éléments classés par ordre croissant. Le problème consiste à établir un algorithme qui permette de dire si un élément x appartient ou pas au tableau T . L'algorithme recherché ne doit pas effectuer une recherche séquentielle, mais doit utiliser le principe de la dichotomie : à chaque étape de l'itération, l'intervalle de recherche doit être divisé par deux. Pour rechercher un élément dans un tableau ayant un million d'éléments, au plus vingt comparaisons suffiront !

Cet exemple de la recherche dichotomique est très riche en soi, car les variations de situation sont multiples et la méthode permet de parfaitement contrôler ce que l'on écrit. Il faut ajouter que les versions, la plupart du temps erronées, que les étudiants écrivent, peuvent être plus facilement corrigées si l'on cherche à mettre en évidence la situation générale à partir de laquelle leur algorithme semble construit.

Version a

[1]



où + indique la zone des éléments de T qui sont strictement plus grands que x et - ou = indique la zone des éléments inférieurs ou égaux à x .

[2] C'est terminé lorsque $d < g$

[3]

$k \leftarrow (g + d) \text{ div } 2$

si $T[k] \leq x$ alors $g \leftarrow k + 1$ sinon $d \leftarrow k - 1$

[4] Les initialisations $g \leftarrow 0$ et $d \leftarrow n - 1$ conviennent.

L'étape [3] conserve de façon évidente la situation choisie en [1]. D'autre part, à chaque étape, soit g augmente, soit d diminue, la condition d'arrêt sera toujours atteinte. On obtient :

Algorithme 27 (Recherche dichotomique, version a).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

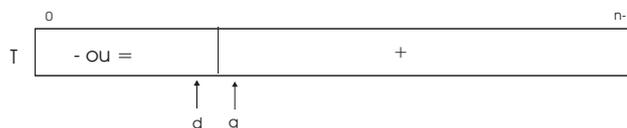
Sortie : L'indice qui indique l'emplacement de la dernière occurrence de x dans T , -1 si x est absent.

```

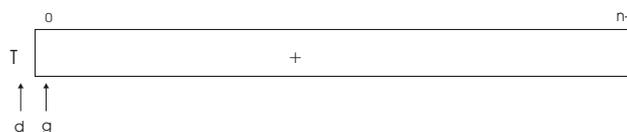
g ← 0
d ← n - 1
tant que g ≤ d faire
    | k ← (g + d) div 2
    | si T[k] ≤ x alors
    | | g ← k + 1
    | sinon
    | | d ← k - 1
si d ≥ 0 et T[d] = x alors
    | r ← d
sinon
    | r ← -1
résultat r
    
```

Lorsque la situation finale est atteinte, il suffit d'examiner les diverses possibilités pour conclure.

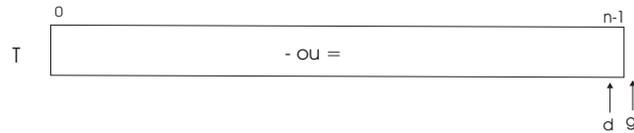
Les deux indices d et g ont changé de valeur :



Seul d a changé de valeur :



Seul g a changé de valeur :

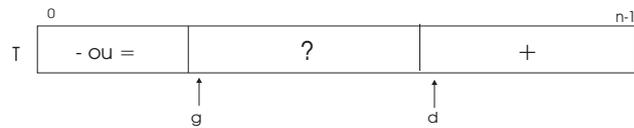


Attention, il ne faut pas oublier de tester si $d < 0$, car dans ce cas, accéder à $T[d]$ déclencherait une erreur.

Version b

Voici une autre situation, très voisine de la précédente, qui conduit à un algorithme sensiblement différent.

[1]



[2] C'est terminé lorsque $d = g$

[3]

$k \leftarrow (g + d) \text{ div } 2$

si $T[k] \leq x$ **alors** $g \leftarrow k + 1$ **sinon** $d \leftarrow k$

Cette étape conserve de façon évidente la situation choisie en [1]

[4] Les initialisations $g \leftarrow 0$ et $d \leftarrow n$ conviennent.

On obtient :

Algorithme 28 (Recherche dichotomique, version *b*).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement de la dernière occurrence de x dans T , -1 si x est absent.

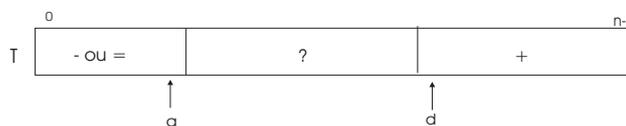
```

g ← 0
d ← n
tant que g ≠ d faire
    | k ← (g + d) div 2
    | si T[k] ≤ x alors
    | | g ← k + 1
    | sinon
    | | d ← k
si g > 0 et T[g - 1] = x alors
    | r ← g - 1
sinon
    | r ← -1
résultat r
    
```

Version c

Voici une autre situation, très voisine des précédentes, qui conduit à un algorithme différent.

[1]



[2] C'est terminé lorsque $d = g + 1$

[3]

$k \leftarrow (g + d) \text{ div } 2$

si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k$

Cette étape conserve de façon évidente la situation choisie en [1] [4] Les initialisations $g \leftarrow -1$ et $d \leftarrow n$ conviennent.

On obtient :

Algorithme 29 (Recherche dichotomique, version c).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement de la dernière occurrence de x dans T , -1 si x est absent.

```

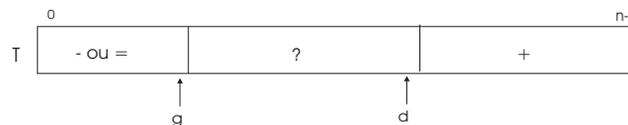
 $g \leftarrow -1$ 
 $d \leftarrow n$ 
tant que  $g + 1 \neq d$  faire
    |  $k \leftarrow (g + d) \text{ div } 2$ 
    | si  $T[k] \leq x$  alors
    |   |  $g \leftarrow k$ 
    |   sinon
    |     |  $d \leftarrow k$ 
si  $g \geq 0$  et  $T[g] = x$  alors
    |  $r \leftarrow g$ 
sinon
    |  $r \leftarrow -1$ 
résultat  $r$ 

```

Version d

Voici une autre situation, très voisine des précédentes, qui montre combien il est important de vérifier tous les points avant de conclure à la validité d'un algorithme.

[1]



[2] C'est terminé lorsque $g = d$

[3]

$k \leftarrow (g + d) \text{ div } 2$

si $T[k] \leq x$ **alors** $g \leftarrow k$ **sinon** $d \leftarrow k - 1$

Cette étape conserve de façon évidente la situation choisie en [1]

[4] Les initialisations $g \leftarrow -1$ et $d \leftarrow n - 1$ conviennent.

Tout semble parfaitement correct, et pourtant il y a une erreur. On n'est pas assuré que la condition d'arrêt soit atteinte dans tout les cas. En effet, dans le cas où d devient égal à $g + 1$, $k \leftarrow (g + d) \text{ div } 2$ est égal à g et si le test $T[k] \leq x$ est satisfait l'instruction $g \leftarrow k$ ne permet pas à g d'augmenter et le programme boucle !

Pour être sûr de la terminaison, il faut écrire $g \geq d - 1$ pour condition d'arrêt. Lorsque cette condition d'arrêt est atteinte il faut examiner attentivement toutes

les situations finales possibles avant de conclure. La conclusion est beaucoup plus délicate à écrire.

Voici l'algorithme modifié :

Algorithme 30 (Recherche dichotomique, version d).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement de la dernière occurrence de x dans T , -1 si x est absent.

```

 $g \leftarrow -1$ 
 $d \leftarrow n - 1$ 
tant que  $g < d - 1$  faire
     $k \leftarrow (g + d) \text{ div } 2$ 
    si  $T[k] \leq x$  alors
         $g \leftarrow k$ 
    sinon
         $d \leftarrow k - 1$ 
si  $g \geq 0$  et  $T[g] = x$  alors
     $r \leftarrow g$ 
sinon
    si  $d \geq 0$  et  $T[d] = x$  alors
         $r \leftarrow d$ 
    sinon
         $r \leftarrow -1$ 
résultat  $r$ 

```

On peut aussi modifier le calcul de k pour s'assurer de l'arrêt de l'algorithme. En posant $k \leftarrow (g + d + 1) \text{ div } 2$ et en supposant $g \neq d$, on a les inégalités :

$$g < k \leq d$$

Lorsque l'étape 3 est exécutée soit $g \leftarrow k$, soit $d \leftarrow k - 1$, dans tous les cas soit g augmente strictement, soit k diminue strictement. La condition d'arrêt $g = d$ sera donc toujours atteinte. On obtient l'algorithme :

Algorithme 31 (Recherche dichotomique, version d).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement de la dernière occurrence de x dans T , -1 si x est absent.

```

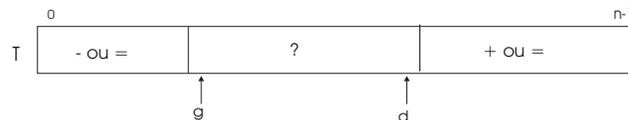
 $g \leftarrow -1$ 
 $d \leftarrow n - 1$ 
tant que  $g \neq d$  faire
    |  $k \leftarrow (g + d + 1) \text{ div } 2$ 
    | si  $T[k] \leq x$  alors
    | |  $g \leftarrow k$ 
    | sinon
    | |  $d \leftarrow k - 1$ 
si  $g \geq 0$  et  $T[g] = x$  alors
    |  $r \leftarrow g$ 
sinon
    |  $r \leftarrow -1$ 
résultat  $r$ 

```

Version e

Voici encore une autre situation, qui conduit à l'écriture d'un algorithme que l'on trouve parfois dans les livres d'informatique.

[1]



[2] C'est terminé lorsque $g > d$

[3]

$k \leftarrow (g + d) \text{ div } 2$

si $T[k] \leq x$ **alors** $g \leftarrow k + 1$

si $T[k] \geq x$ **alors** $d \leftarrow k - 1$

Cette étape conserve de façon évidente la situation choisie en [1]

[4] Les initialisations $g \leftarrow 0$ et $d \leftarrow n - 1$ conviennent.

On obtient :

Algorithme 32 (Recherche dichotomique, version e).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement d'une occurrence de x dans T , -1 si x

est absent.

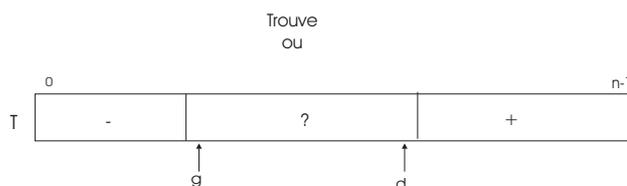
```

g ← 0
d ← n - 1
tant que g ≤ d faire
    | k ← (g + d) div 2
    | si T[k] ≤ x alors
    |   | g ← k + 1
    | si T[k] ≥ x alors
    |   | d ← k - 1
si g = d + 2 alors
    | r ← k
sinon
    | r ← -1
résultat r
    
```

Version f

Voici une dernière situation qui, comme dans la version précédente, conduit à interrompre l'itération dans le cas où la dichotomie "tombe pile" sur l'élément recherché.

[1]



[2] C'est terminé lorsque $g > d$ ou *trouve*

[3]

```

k ← (g + d) div 2
si T[k] = x alors trouve ← vrai
sinon si T[k] < x alors g ← k + 1 sinon d ← k - 1
    Cette étape conserve de façon évidente la situation choisie en [1]
    
```

[4] Les initialisations $g \leftarrow 0$, $d \leftarrow n - 1$ et *trouve* ← *faux* conviennent. On obtient :

Algorithme 33 (Recherche dichotomique, version *f*).

Entrée : Un tableau trié T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice qui indique l'emplacement d'une occurrence de x dans T , -1 si x est absent.

```

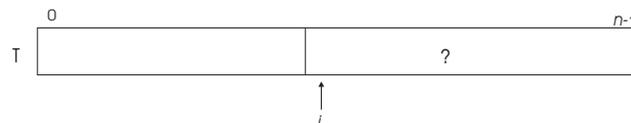
g ← 0
d ← n - 1
trouve ← faux
tant que g ≤ d et non trouve faire
    | k ← (g + d) div 2
    | si T[k] = x alors
    |   | trouve ← vrai
    | sinon
    |   | si T[k] < x alors
    |   |   | g ← k + 1
    |   | sinon
    |   |   | d ← k - 1
si trouve alors
    | r ← k
sinon
    | r ← -1
résultat r

```

1.5.6 Recherche d'un plus petit élément dans un tableau

Soit $T[0..n-1]$ un tableau de n éléments. On cherche un algorithme qui après avoir parcouru le tableau T une seule fois permet de connaître la valeur du plus petit élément de ce tableau. On peut imaginer comme situation générale celle où l'algorithme a examiné les i premières "cases" et repéré parmi celles-ci la valeur min du plus petit élément. Ce qui conduit à l'algorithme suivant.

[1]



et $\{\forall j, j \in [0, i[\Rightarrow min \leq T[j]\}$

[2] C'est terminé lorsque $i = n$

[3]
si $T[i] \leq min$ **alors** $min \leftarrow T[i]$
 $i \leftarrow i + 1$

[4] Les initialisations $min \leftarrow T[0]$ et $i \leftarrow 1$ conviennent.

On obtient :

Algorithme 34 (Recherche du minimum).

Entrée : Un tableau T de n composantes, x un élément du même type que les composantes de T .

Sortie : La valeur du plus petit élément.

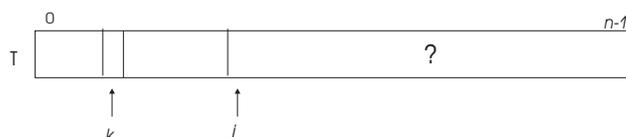
```

min ← T[0]
i ← 1
tant que  $i \neq n$  faire
    si  $T[i] \leq min$  alors
        |  $min \leftarrow T[i]$ 
        |  $i \leftarrow i + 1$ 
résultat  $min$ 
    
```

1.5.7 Recherche de l'emplacement du plus petit élément dans un tableau

Soit $T[0..n-1]$ un tableau de n éléments. On cherche un algorithme qui après avoir parcouru le tableau T une seule fois permet de connaître l'indice de la "case" où se trouve un (il peut y en avoir plusieurs) plus petit élément de ce tableau. On peut imaginer comme situation générale celle où l'algorithme a examiné les i premières "cases" et repéré parmi celles-ci l'indice k de l'endroit où se trouve le plus petit élément. Ce qui conduit à l'algorithme suivant.

[1]



et $\{\forall j, j \in [0, i[\Rightarrow T[k] \leq T[j]\}$

[2] C'est terminé lorsque $i = n$

```

[3]
si  $T[i] \leq T[k]$  alors  $k \leftarrow i$ 
 $i \leftarrow i + 1$ 

```

[4] Les initialisations $k \leftarrow 0$ et $i \leftarrow 1$ conviennent.

On obtient :

Algorithme 35 (Recherche de l'emplacement du minimum).

Entrée : Un tableau T de n composantes, x un élément du même type que les composantes de T .

Sortie : L'indice de l'emplacement de la dernière occurrence du plus petit élément.

```

 $k \leftarrow 0$ 
 $i \leftarrow 1$ 
tant que  $i \neq n$  faire
    | si  $T[i] \leq T[k]$  alors
    | |  $k \leftarrow i$ 
    |  $i \leftarrow i + 1$ 
résultat  $k$ 

```

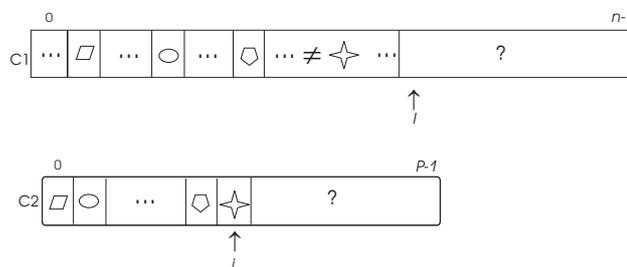
1.5.8 Recherche dans un tableau $C1$ d'une sous suite extraite égale à un tableau $C2$ donné

Soit $C1[0..n-1]$ un tableau de n éléments et $C2[0..p-1]$ un tableau de p éléments. On cherche un algorithme qui répond *vrai* si tous les éléments du tableau $C2$ apparaissent dans le tableau $C1$ et ce dans le même ordre d'apparition, *faux* sinon. Ainsi, deux éléments contigus dans le tableau $C2$ ne sont pas forcément contigus dans le tableau $C1$. Les tableaux $C1$ et $C2$ sont quelconques et peuvent contenir des nombres, des caractères, ou tout autre chose. L'important est que l'on puisse tester l'égalité de deux objets entre eux.

On peut imaginer la situation suivante : l'algorithme a commencé sa recherche, a déjà trouvé les j premiers objets du tableau $C2$ dans le tableau $C1$, a commencé à rechercher $C2[j]$ (matérialisé par une étoile) dans le tableau $C1$ et ne l'a pas trouvé dans les cases qui se trouvent entre l'emplacement du dernier objet du tableau $C1$ égal à $C2[j-1]$ et $i-1$.

Ce qui nous conduit à établir la situation générale suivante :

[1]



[2] C'est terminé lorsque $i = n$ ou $j = p$

[3]

si $C2[j] = C1[i]$ alors $j \leftarrow j + 1$
 $i \leftarrow i + 1$

[4] Les initialisations $i \leftarrow 0$ et $j \leftarrow 0$ conviennent.

On obtient :

Algorithme 36 (Recherche d'une sous suite extraite).

Entrée : Deux tableaux $C1$ et $C2$ de taille respectives n et p .

Sortie : vrai si tous les éléments de $C2$ apparaissent dans le même ordre dans $C1$, faux sinon.

```

i ← 0
j ← 0
tant que  $i \neq n$  et  $j \neq p$  faire
    | si  $C2[j] = C1[i]$  alors
    | |  $j \leftarrow j + 1$ 
    | |  $i \leftarrow i + 1$ 
si  $j = p$  alors
    | résultat vrai
sinon
    | résultat faux
    
```

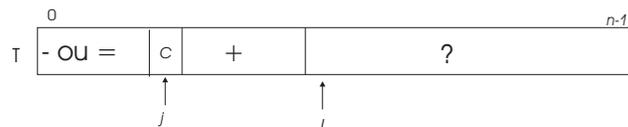
1.5.9 Partition dans un tableau

Soit $T[0..n - 1]$ un tableau de n éléments et soit c l'élément $T[0]$. On cherche un algorithme qui par des échanges successifs, permette de placer cet élément de manière telle que tous les éléments qui lui sont inférieurs ou égaux soient placés à sa gauche et les autres à sa droite. Cet élément de valeur c serait donc à sa place

si l'on classait le tableau par ordre croissant. On impose de n'examiner qu'**une seule fois** les éléments et de ne pas utiliser de tableau auxiliaire. Le tableau T doit rester globalement invariant (i.e. aucun élément ne disparaît et aucun nouveau ne s'introduit), mais on n'exige pas que les deux parties de part et d'autre soient classées. On peut imaginer la situation ci-dessous, où l'étoile matérialise l'élément c qui sert à réaliser la partition. La zone des éléments qui lui sont inférieurs ou égaux est indiquée par **- ou =** et celle des éléments strictement supérieurs par **+**.

Version a

[1]



[2] C'est terminé lorsque $i = n$

[3]

si $T[i] > c$ **alors** $i \leftarrow i + 1$

sinon

échanger $T[j + 1]$ et $T[i]$

échanger $T[j + 1]$ et $T[j]$

$j \leftarrow j + 1$

$i \leftarrow i + 1$

[4] Les initialisations $c \leftarrow T[0]$, $j \leftarrow 0$ et $i \leftarrow 1$ conviennent.

On obtient :

Algorithme 37 (Partition dans un tableau, version a).

Entrée : Un tableau T de taille n .

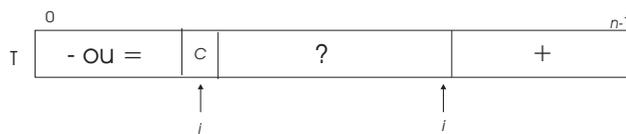
Sortie : L'indice de l'emplacement final où se trouve l'élément qui a servi faire la partition.

```

 $c \leftarrow T[0]$ 
 $j \leftarrow 0$ 
 $i \leftarrow 1$ 
tant que  $i \neq n$  faire
    si  $T[i] \leq c$  alors
        échanger  $T[j + 1]$  et  $T[i]$ 
        échanger  $T[j + 1]$  et  $T[j]$ 
         $j \leftarrow j + 1$ 
     $i \leftarrow i + 1$ 
résultat  $j$ 
    
```

Version b

[1]



[2] C'est terminé lorsque $i = j$

[3]

```

si  $T[j + 1] > c$  alors
    échanger  $T[j + 1]$  et  $T[i]$ 
     $i \leftarrow i - 1$ 
sinon
    échanger  $T[j + 1]$  et  $T[j]$ 
     $j \leftarrow j + 1$ 
    
```

[4] Les initialisations $c \leftarrow T[0]$, $j \leftarrow 0$ et $i \leftarrow n - 1$ conviennent.

On obtient :

Algorithme 38 (Partition dans un tableau, version b).

Entrée : Un tableau T de taille n .

Sortie : L'indice de l'emplacement final où se trouve l'élément qui a servi faire la partition.


```
 $c \leftarrow T[0]$   
 $j \leftarrow 1$   
 $i \leftarrow n - 1$   
tant que  $i \geq j$  faire  
    si  $T[j] > c$  alors  
        échanger  $T[j]$  et  $T[i]$   
         $i \leftarrow i - 1$   
    sinon  
         $j \leftarrow j + 1$   
 $T[0] \leftarrow T[i]$   
 $T[i] \leftarrow c$   
résultat  $i$ 
```

1.6 Analyse des algorithmes

Le temps d'exécution d'un programme donné dépend de deux facteurs :

- l'ordinateur (rapidité des instructions) ;
- le compilateur (efficacité du code généré) ;

On ne peut donc pas affirmer que tel programme s'exécutera en x secondes ! Pour gommer ces critères on analyse les algorithmes plutôt que les programmes afin d'obtenir des mesures indépendantes des compilateurs et des machines. Pour un algorithme donné, on se donne une mesure de complexité T , généralement un décompte d'instructions et une taille, notée n , qui dépend des données fournies en entrée. Pour un algorithme de tri, n est la dimension du tableau à trier, pour le calcul du pgcd, n est le nombre de chiffres des entiers manipulés, pour l'algorithme de Hörner n est le degré du polynôme considéré, ... On peut envisager plusieurs façons de mesurer la complexité.

- La mesure dans le meilleur des cas : cette mesure n'est pas pertinente pour avoir une vraie idée de la complexité d'un algorithme. Si l'on compare plusieurs algorithmes de tri sur des tableaux déjà triés, on obtiendra des mesures de complexité voisines.
- La mesure dans le pire cas : Cette mesure peut avoir parfois son importance selon la probabilité d'apparition de ce cas.
- La mesure en moyenne : c'est elle qui traduira le mieux la performance d'un algorithme.

On fait intervenir un autre critère pour comparer les performances des algorithmes, à savoir la façon dont leur complexité évolue lorsque n croît. Cela est justifié par le fait que cette notion de complexité est cruciale lorsque les données ont une taille importante. Il est donc naturel de se préoccuper de la croissance de cette complexité en fonction de la taille n des données. Pour cela on utilise la notation "grand O " qui permet d'indiquer le comportement asymptotique de $T(n)$ lorsque n tend vers l'infini. Si $f(n)$ est une fonction définie pour $n > 0$, on dira que la complexité $T(n)$ d'un algorithme donné est en $O(f(n))$ si et seulement si il existe un seuil n_0 et un constante $c > 0$ tels que pour tout n supérieur ou égal à n_0 , on ait $T(n) \leq c \times f(n)$

Le calcul de la complexité d'un algorithme met très souvent en jeu des notions issues de plusieurs domaines mathématiques comme l'analyse combinatoire, les probabilités, l'analyse asymptotique, ...

Citons, à titre d'exemples, classées par ordre de complexité croissante en fonction de n , quelques unes des fonctions f qui apparaissent dans l'analyse des algorithmes :

$\log(n)$ (complexité logarithmique), $(\log(n))^2$, \sqrt{n} , n (complexité linéaire), $n \log(n)$, $n(\log(n))^2$, n^2 (complexité quadratique), n^3 (complexité cubique), 2^n et n^n (com-

plexité exponentielle). Les algorithmes exponentiels deviennent rapidement impraticables dès que n atteint quelques dizaines d'unités, à l'opposé les algorithmes de complexité logarithmique sont très efficaces même pour des valeurs de n très grandes. La recherche dichotomique dans un tableau classé est de complexité logarithmique, pour n égal à 10^6 une vingtaine de comparaisons suffiront pour la recherche d'un élément ! Les tris par selection et par insertion sont de complexité quadratique, le produit de matrices est d'une complexité cubique. La constante c peut avoir son importance. Par exemple le tri rapide (*Quicksort*) est en $O(n \log(n))$ en moyenne et en $O(n^2)$ dans le pire des cas alors que le tri par fusion est dans tous les cas en $O(n \log(n))$ et semble meilleur d'après ce critère. Et pourtant le *Quicksort* est dans la plupart des cas deux fois plus rapide. D.Knuth et R.Sedgwick ont démontré dans les années 70 que la constante c dans le $O(n \log(n))$ du coût moyen est environ deux fois meilleure que celle du tri par fusion. C'est pour cette raison que le *Quicksort* est donné comme l'algorithme de tri le plus rapide. La recherche de nouveaux algorithmes pour obtenir une plus faible complexité est une activité importante en informatique.

1.6.1 Quelques exemples de calculs de complexité

Le tri par insertion

```

i ← 0
tant que i ≠ n - 1 faire
    | i ← i + 1
    | j ← i
    | tant que j ≠ 0 et T[j - 1] > T[j] faire
    | | Echanger T[j - 1] et T[j]
    | | j ← j - 1

```

Chaque opération (comparaisons, échanges, ...) se fait en temps constant, sauf les deux boucles **tant que**. La boucle sur i tourne $n - 1$ fois. La boucle sur j tourne $i - 1$ fois. i est borné par n , le nombre de tours est inférieur à n^2 : l'algorithme est en $O(n^2)$. On peut calculer le nombre exact de tours : $1 + 2 + \dots + n - 2 + n - 1 = \frac{n \times (n - 1)}{2}$, complexité dans tous les cas. L'algorithme est donc au pire, au mieux et en moyenne en $O(n^2)$.

La recherche dichotomique dans une table triée

```

g ← 0
d ← n - 1
tant que g ≤ d faire
    | k ← (g + d) div 2
    | si T[k] ≤ x alors g ← k + 1 sinon d ← k - 1
si d ≥ 0 et T[d] = x alors " trouvé en d " sinon " pas trouvé "
```

Au départ l'intervalle de recherche de contient n éléments. En effet g (la borne gauche) est initialisé à 0 et d (la borne droite) est initialisé à $n-1$. A chaque passage dans la boucle **tant que** l'intervalle de recherche est divisé par 2. Ainsi le nombre d'itérations est égal au plus petit entier k tel que $2^k \geq n$, donc $k = \log_2(n)$. Cette version de la recherche dichotomique est donc dans tous les cas en $O(\log_2(n))$.

La recherche séquentielle

On recherche la présence de x dans une table T ayant n éléments. Cet algorithme donne le résultat **vrai** si x est présent dans la table, **faux** sinon.

```

i ← 0
tant que i < n et x ≠ T[i] faire
    | i ← i + 1
résultat i < n
```

La complexité au mieux est en $O(1)$, x est trouvé en $T[0]$ et au pire en $O(n)$. En effet, si x n'est pas présent, on sort de la boucle lorsque x a atteint la valeur n . Pour la complexité en moyenne, en supposant que toutes les positions où l'on peut trouver x sont équiprobables on a :

$$\frac{1 + 2 + \dots + n}{n} = \frac{n \times (n + 1)}{2 \times n} = \frac{n + 1}{2}$$

On effectue donc en moyenne $\frac{n}{2}$ tours de boucle, ainsi la complexité moyenne est encore en $O(n)$.

Recherche du $k^{\text{ème}}$ plus petit élément dans un tableau

On peut réordonner partiellement le tableau T en modifiant le tri par sélection pour mettre à leur place les k plus petits éléments. Le résultat sera l'élément $T[k - 1]$ (le tableau étant indexé à partir de 0). On obtient un algorithme en $O(kn)$. Remarquons que cet algorithme nous fournit l'élément médian d'une suite de nombre en $O(n^2)$.

On peut répondre de manière plus performante au problème posé en utilisant l'algorithme de partition dans un tableau.

Pour cela, on modifiera l'algorithme qui effectue la partition du tableau par rapport au premier élément en le généralisant. Plus précisément au lieu de partitionner le tableau T entre les indices 0 et $n - 1$ par rapport à $T[0]$, on écrira un algorithme qui partitionne une partie du tableau T entre les indices g (gauche) et d (droite) par rapport à $T[g]$. Pour rendre plus lisible l'algorithme on suppose que l'appel de la fonction $partition(T, g, d)$ renverra pour résultat l'emplacement p de l'élément $T[g]$ qui a servi à effectuer la partition. Cet élément est donc "à sa place" si l'on ne considère que la portion du tableau T entre les indices g et d . L'idée est la suivante :

On effectue une première partition du tableau T entre les indices 0 et $n - 1$, si la position p renvoyée est égale à $k - 1$, c'est terminé, sinon on demande à nouveau une partition, soit sur la partie gauche entre les indices 0 et $p - 1$, soit sur la partie droite entre les indices $p + 1$ et $n - 1$ suivant que $p > k - 1$ ou que $p < k - 1$. On réitère ce processus tant que le résultat de la partition est différent de $k - 1$. Ce qui nous conduit à écrire l'algorithme suivant :

Algorithme 40 (Recherche du $k^{\text{ème}}$ plus petit élément dans un tableau).

Entrée : Un tableau T de taille n .

Sortie : La valeur du $k^{\text{ème}}$ plus petit élément.

```

 $g \leftarrow 0$ 
 $d \leftarrow n - 1$ 
 $p \leftarrow partition(T, g, d)$ 
tant que  $p \neq k - 1$  faire
    si  $p > k - 1$  alors
         $p \leftarrow partition(T, g, p - 1)$ 
    sinon
         $p \leftarrow partition(T, p + 1, d)$ 
résultat  $T[k - 1]$ 

```

En quoi cet algorithme est-il plus performant que le précédent qui utilisait le tri par sélection ?

Pour avoir une idée de son comportement lorsque n est très grand, supposons n égal à 2^q et que chaque appel à la fonction partition divise par 2 l'espace de recherche.

On effectuera ainsi n comparaisons lors du premier appel puis $2^{q-1} + 2^{q-2} + \dots + 2 + 1$ comparaisons (dans le pire des cas) par la suite, soit $2^q - 1$. Ainsi l'efficacité de ce second algorithme est en $O(2n)$.