Chapitre 3 : Développer du logiciel avec méthode

3.1 Développement méthodique du logiciel

- la production du logiciel
- conception structurée descendante et machines abstraites
- notion d'algorithme
- un langage de description d'algorithmes le LDFA
- le dossier de programmation
- trace formelle d'un algorithme
- traducteur LDFA - Pascal
- facteurs de qualité du logiciel

Machines abstraites **: exemple de traitement sur les chaînes**

- cas où la version du pascal contient un type chaîne
- cas où la version du pascal ne contient pas de type chaîne
- programme pascal obtenu
- autres versions d'implantation en pascal

3.2.Modularité

- définition : B.Meyer
- la modularité en pascal avec les **Unit**

3.3. Complexité, tri, recherche

- Notions de complexité temporelle et spatiale
- Mesure de la complexité temporelle d'un algorithme
- Notation de Landau O(n)
-

Trier des tableaux en mémoire centrale

- Le Tri à bulles
- Le Tri par sélection

- Le ri par insertion
- Le Tri rapide QuickSort
- Le Tri par tas HeapSort

Rechercher dans un tableau

- Dans un tableau non trié
- Dans un tableau trié

Exercices: algorithmes et leur traduction

3.1 : développement méthodique du Logiciel

Plan du chapitre:

1.Historique des langages

Introduction

1. Production du logiciel

1.1 Génie logiciel
1.2 Cycle de vie ***du logiciel***
1.3 Maintenance *d'un logiciel*
1.4 Production industrielle ***du logiciel***

2. Conception structurée descendante

2.1 Critère simple d'automatisation
2.2 Analyse méthodique descendante
2.3 Analyse ascendante
2.4 ***Programmation descendante*** avec retour sur un niveau
2.5 Machines abstraites ***et niveaux logiques***

3. Notion d'ALGORITHME

3.1 Langage algorithmique
3.2 Objets de base ***d'un langage algorithmique***
3.3 Opérations sur les objets de base ***d'un langage algorithmique***

4. Un langage de description d'algorithme : LDFA

4.1 Atomes ***du LDFA***
4.2 Information ***en LDFA***
4.3 Vocabulaire terminal ***du LDFA***
4.4 Instructions simples ***du LDFA***

5. Le Dossier de développement

5.1 Enoncé et spécification
5.2 Méthodologie
5.3 Environnement
5.4 Algorithme en LDFA
5.5 Programme en langage Pascal

6. Trace formelle d'un algorithme

Introduction

Le bon sens est la chose du monde la mieux partagée...la diversité de nos opinions ne vient pas de ce que les uns sont plus raisonnables que les autres, mais seulement de ce que nous conduisons nos pensées par diverses voies, et ne considérons pas les mêmes choses. Car ce n'est pas assez d'avoir l'esprit bon, mais le principal est de l'appliquer bien.
R Descartes Discours de la méthode, première partie, 1637.

Le développement méthodique d'un logiciel passe actuellement par une démarche de " descente concrète " de la connaissance que l'humain a sur la problématique du sujet, vers l'action élémentaire exécutée par un ordinateur. Le travail du programmeur étant alors ramené à une traduction permanente des actions humaines en actions machines (décrites avec des outils différents).

Nous pouvons en première approximation différentier cette " descente concrète " en un classement selon quatre niveaux d'abstraction :

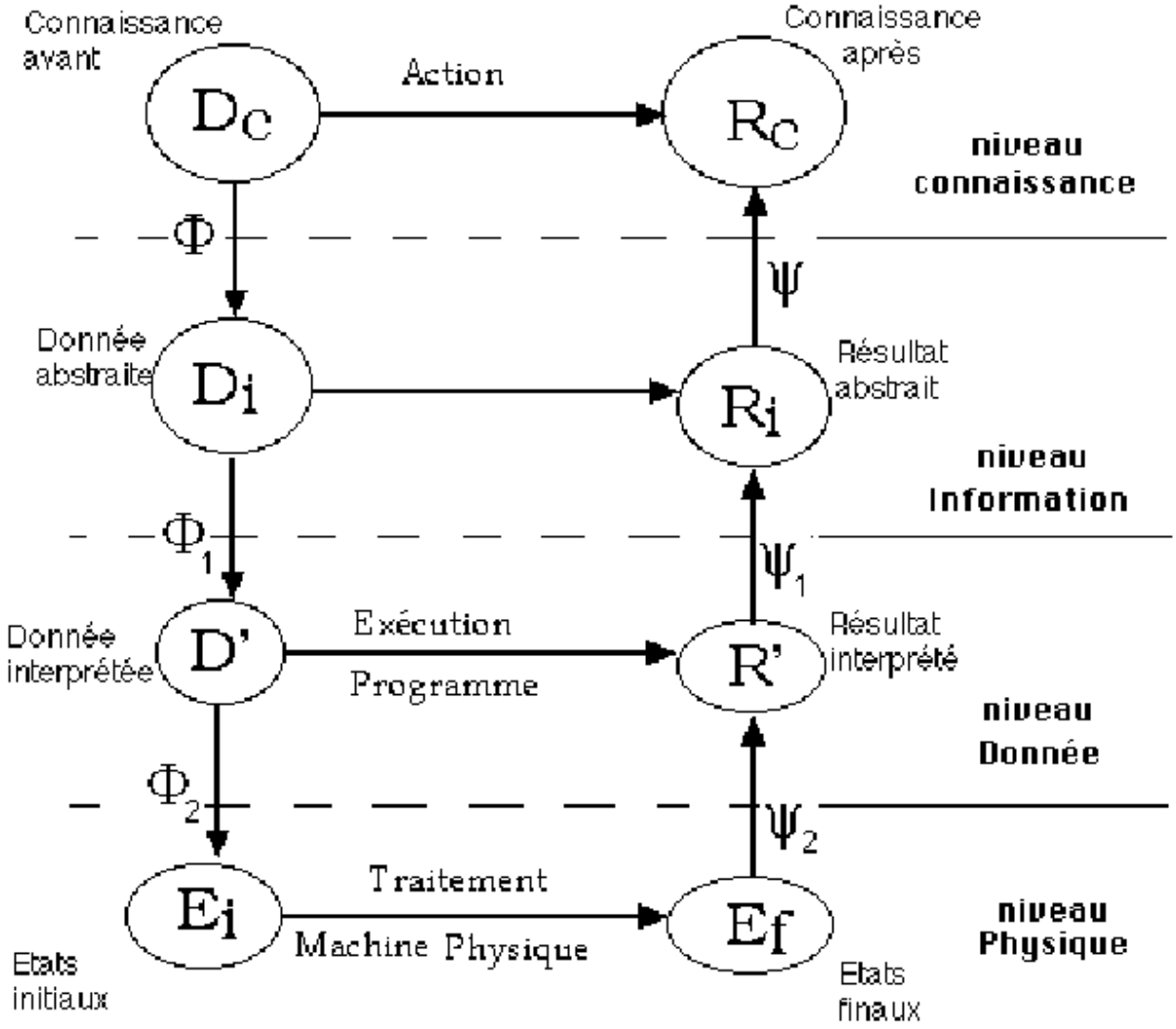


fig - schéma de descente concrète

Nous voyons que toute activité de programmation consiste à transformer un problème selon une descente graduelle de l'humain vers la machine. Ici nous avons résumé cette décomposition en 4 niveaux. La notion de programmation structurée est une réponse à ce type de décomposition graduelle d'un problème. L'algorithmique est la façon de décrire cette méthode de travail.

1. Production du logiciel

1.1 Génie logiciel

A une certaine époque, à ses débuts, l'activité d'écriture du logiciel ne reposait que sur l'efficacité personnelle du programmeur laissé pratiquement seul devant la programmation d'un problème.

De nos jours, le programmeur dispose d'outils et de méthodes lui permettant de concevoir et d'écrire des logiciels. Le terme **logiciel**, ne désigne pas seulement les programmes associés à telle application ou tel produit : il désigne en plus la documentation nécessaire à l'installation, à l'utilisation, au développement et à la maintenance de ce logiciel. Pour de gros systèmes, le temps de réalisation peut être aussi long que le temps du développement des programmes eux-mêmes.

Le **génie logiciel** concerne l'ensemble des méthodes et règles relatives à la production rationnelle des logiciels.

L'activité de développement du logiciel, vu les coûts qu'elle implique, est devenue une *activité économique* et doit donc être planifiée et soumise à des normes sinon à des attitudes équivalentes à celles que l'on a dans l'industrie pour n'importe quel produit.

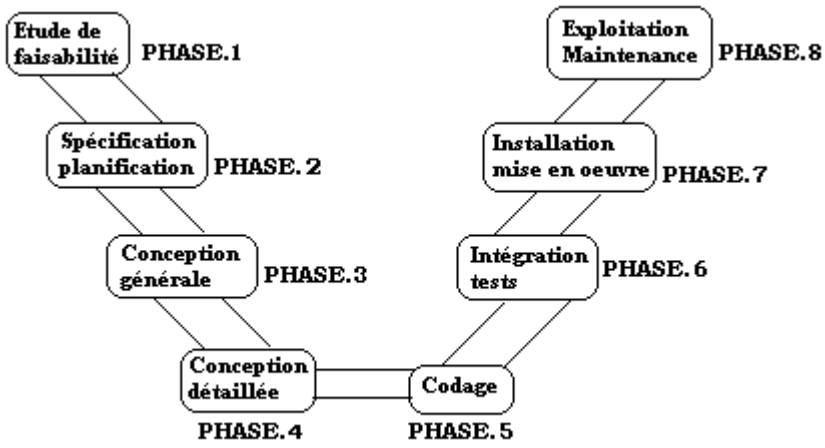
C'est pourquoi dans ce cours, le mot-clef est le mot **"composant logiciel"** qui tient à la fois de l'activité créatrice de l'humain et du composant industriel incluant une activité *disciplinée et ordonnée* basée pour certaines tâches sur des outils formalisés.

D'autre part le génie logiciel intervient lorsque le logiciel est trop grand pour que son développement puisse être confié à un seul individu ; ce qui n'est pas le cas pour des débutants, à qui il n'est pas confié l'élaboration de gros logiciels. Toutefois, il est possible de sensibiliser le lecteur débutant à l'habitude d'élaborer un logiciel d'une manière systématique et rationnelle à l'aide d'outils simples.

1.2 Cycle de vie du logiciel

Comme il faut un temps très important pour développer un grand système logiciel, et que d'autre part ce logiciel est prévu pour être utilisé pendant longtemps, on sépare fictivement des étapes distinctes dans ces périodes de développement et d'utilisation.

Le modèle dit de la cascade de Royce (1970) accepté par tout le monde informatique est un bon outil pour le débutant. S'il est utilisé pour de gros projets industriels, en supprimant les recettes et les validations en fin de chaque phase, nous disposons en initiation d'un cadre méthodologique. Il se présente alors sous forme de 8 diagrammes ou phases :



1.3 Maintenance d'un logiciel

Dans beaucoup de cas le coût du logiciel correspond à la majeure partie du coût total d'une application informatique. Dans ce coût du logiciel, la maintenance a elle-même une part prépondérante puisqu'elle est estimée de nos jours au minimum à **75% du coût total du logiciel**.

La maintenance est de trois sortes :

- adaptative (s'adapter à un nouvel environnement...)
- corrective (corrections d'erreurs...)
- perfective (améliorations demandées par le client...)

1.4 Production industrielle du logiciel

La production du logiciel étant devenue une activité industrielle et donc économique, elle n'échappe pas aux données économiques classiques. On répertorie un ensemble de caractéristiques associées à un projet de développement, chaque caractéristique se voyant attribuer un ratio de productivité.

Le ratio de productivité d'une caractéristique

C'est le rapport entre la productivité (exprimée en nombre d'Instructions Sources Livrées, par homme et par mois) d'un projet exploitant au mieux cette caractéristique, et la productivité d'un projet n'exploitant pas du tout cette caractéristique.

Le tableau suivant est tiré d'une étude de B.Boehm (Revue TSI 1982 : les facteurs de coût du logiciel):

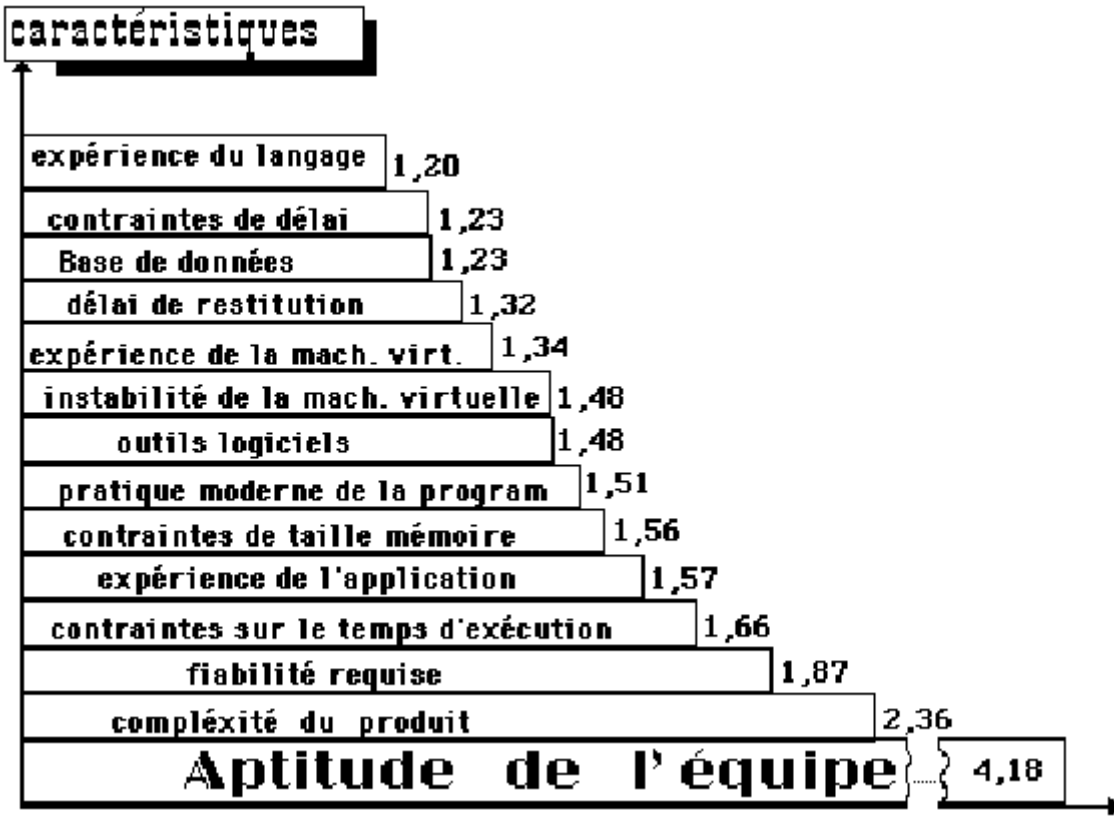


Tableau comparatif des divers ratios de productivité (B.Boehm)

Vous aurez remarqué en observant le graphique précédent que le facteur le plus important n'est pas l'expérience d'un langage (erreur commise par les néophytes). Ce qui explique entre autres arguments que l'enseignement de la programmation ne soit pas l'enseignement d'un langage.

Il apparaît que le facteur le plus coûteux reste un facteur sur lequel la technologie n'a aucune prise : l'aptitude qu'ont des individus à communiquer entre eux !

Pour l'élaboration d'un logiciel, nous allons utiliser deux démarches classiques : la méthode structurée ou algorithmique et plus tard une extension orientée objet de cette démarche.

2. Conception structurée descendante

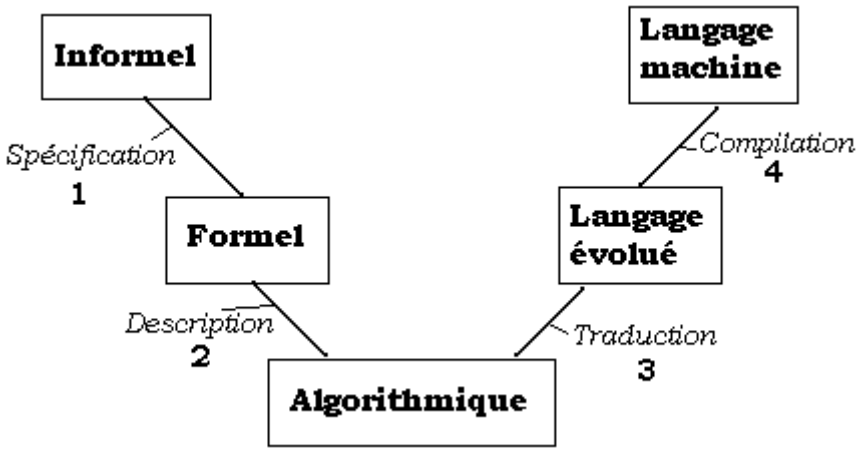
2.1 Critère simple d'automatisation

Un problème est **automatisable** (traitable par informatique) si :

- l'on peut parfaitement définir les données et les résultats,
- l'on peut décomposer le passage de ces données vers ces résultats en une suite d'opérations élémentaires dont chacune peut être exécutée par une machine.

Dans le cadre d'une initiation à la programmation, dans le cycle de vie déjà présenté plus haut, nous ne considérerons que les phases 2 à 6, en supposant que la faisabilité est acquise, et qu'enfin les phases de mise en œuvre et de maintenance sont mises à part.

Dans cette perspective, le schéma de la programmation d'un problème se réduit à 4 phases :



- La phase 1 de spécification utilisera les types abstraits de données (TAD),
- la phase 2 (correspondant aux phases 3 et 4 du cycle de vie) utilisera la méthode de programmation algorithmique,
- la phase 3 (correspondant à la phases 5 du cycle de vie) utilisera un traducteur manuel pascal,
- la phase 4 (correspondant à la phases 6 du cycle de vie) correspondra au passage sur la machine avec vérification et jeux de tests.

Nous utiliserons un " langage algorithmique " pour la description d'un algorithme résolvant un problème. Il s'agit d'un outil textuel permettant de passer de la conception humaine à la conception machine d'une manière souple pour le programmeur.
Nous pouvons résumer dans le tableau ci-dessous les étapes de travail et les outils conceptuels à utiliser lors d'une telle démarche.

ETAPES PRATIQUES	Matériel et moyens techniques à disposition
Analyse	Papier, Crayon, Intelligence, Habitude.
Mise en forme de l'algorithme	C'est l'aboutissement de l'analyse, esprit logique et rationnel.
Description	Utilisation pratique des outils d'une méthode de programmation, ici la programmation structurée.
Traduction	Transfert des écritures algorithmiques en langage de programmation.
Tests et mise au point	Mise au point du programme sur des valeurs tests ou à partir de programmes spécialisés.
Exécution	Phase finale : le programme s'exécute sans erreur.

2.2 Analyse méthodique descendante

Le second [précept], de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre.
R Descartes Discours de la méthode, seconde partie, 1637.

Définir le problème à résoudre:
expliciter les données
  préciser: leur nature
  leur domaine de variation
  leurs propriétés
expliciter les résultats
  préciser: leur structure
  leur relations avec les données
fin définir;
Décomposer le problème en sous-problèmes**;**
Pour chaque sous-problèmes identifié faire
  si solution évidente **alors** écrire le morceau de programme
  sinon appliquer la méthode au sous-problème
  fsi
fpour.

démarche proposée par J.Arsac

Cette démarche méthodique a l'avantage de permettre d'isoler les erreurs lorsqu'on en commet, et elles devraient être plus rares qu'en programmation empirique (anciens organigrammes).

Il apparaît donc plusieurs niveaux de décomposition du problème (niveaux d'abstraction descendants). Ces niveaux permettent d'avoir une description de plus en plus détaillée du problème et donc de se rapprocher par raffinements successifs d'une description prête à la traduction en instructions de l'ordinateur.

Afin de pouvoir décrire la décomposition d'un problème à chaque niveau, nous avons utilisé un langage algorithmique (et non pas un langage de programmation) qui emprunte beaucoup au langage naturel (le français pour nous).

2.3 Analyse ascendante

Le troisième [précept], de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusqu'à la connaissance des plus composés; et supposant même de l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres.
R Descartes Discours de la méthode, seconde partie, 1637.

Nous essaierons de partir de l'existant (les fichiers sources déjà écrits sur le même sujet) et de reconstruire par étapes la solution. Le problème dans cette méthode est d'assurer une bonne cohérence lorsque l'on rassemble les morceaux.

Les méthodes objets que nous aborderons plus loin, sont un bon exemple de cette démarche. Nous n'en dirons pas plus dans ce paragraphe en renvoyant le lecteur intéressé au chapitre de la programmation orientée objet de cours.

2.4 Programmation descendante avec retour sur un niveau

Comme partout ailleurs, une attitude appuyée sur les deux démarches est le gage d'une certaine souplesse dans le travail. Nous adopterons une démarche d'analyse essentiellement descendante, avec la possibilité de remonter en arrière dès que le développement paraît trop complexe.

Nous adopterons dans tout le reste du chapitre une telle méthode descendante (avec quelques retours ascendants). Nous la dénommerons " programmation algorithmique ".

Nous utilisons les concepts de **B.Meyer** pour décomposer un problème en niveaux logiques puis en raffinant successivement les différentes étapes.

2.5 Machines abstraites et niveaux logiques

Principe :
On décompose chacune des étapes du travail en niveaux d'abstractions logiques. On suppose en outre qu'à chaque niveau logique fixé, il existe une machine abstraite virtuelle capable de comprendre et d'exécuter la description du problème sous la forme algorithmique en cours. Ainsi, en descendant de l'abstraction vers le concret, on passe graduellement d'un énoncé de problème au niveau humain à un énoncé du même problème à un niveau où la machine devient capable de l'exécuter.

Niveau logique	Machine abstraite	Enoncé du problème en
0	M_0 = l'humain	A_0 = langage naturel
1	M_1 = mach. Abstraite	A_1 = lang.algorithmique
. . .	. . .	. . .
n	M_n = machine+OS	A_n = langage évolué
n+1	M_{n+1}= machine physique	A_{n+1}= langage binaire

A partir de cette décomposition on construit un " arbre " de programmation représentant graphiquement les hiérarchies des machines abstraites.

Voici un exemple d'utilisation de cette démarche dans le cas de la résolution générale de l'équation du second degré dans **R**.

Le problème se décompose en deux sous-problèmes " :

- le premier concerne la résolution d'une équation du premier degré strict
- le second est relatif à la " résolution d'une équation du second degré strict " .

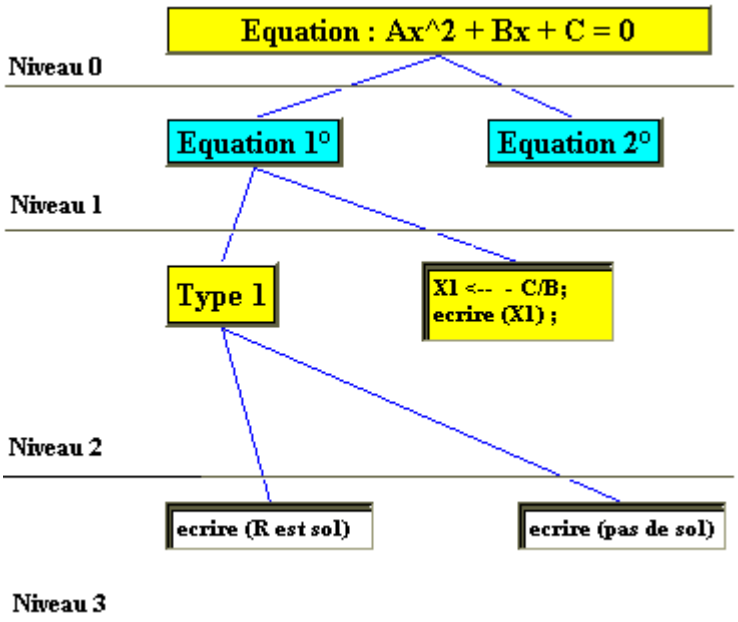


figure de la branche d'arbre 1er degré

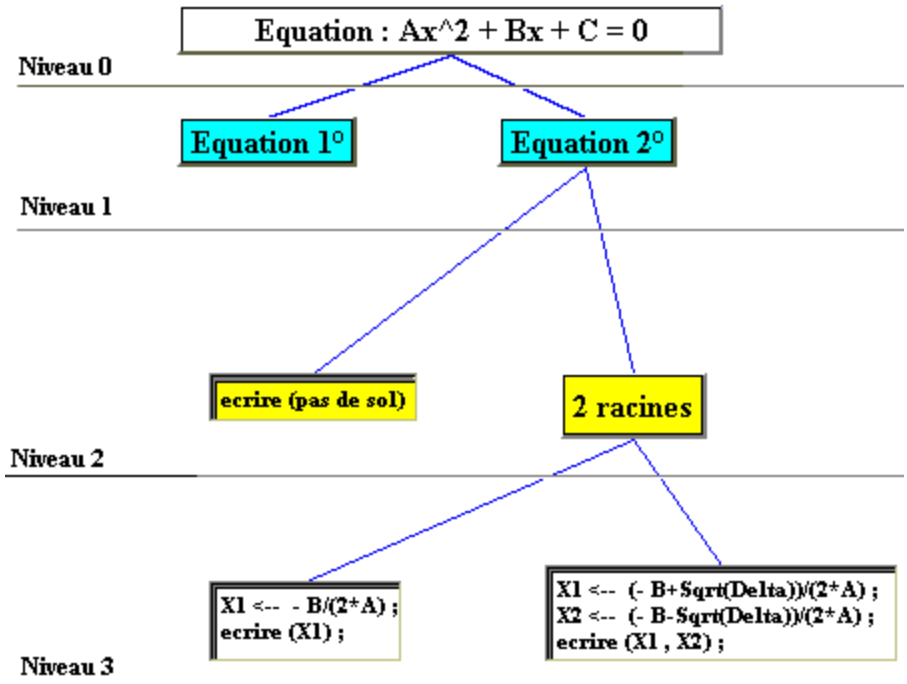


figure de la branche d'arbre 2ème degré

Nous avons utilisé comme langage de description des étapes intermédiaires un langage algorithmique basé sur des mots du français. Nous le détaillerons plus tard.

3. Notion d'ALGORITHME

(D.E. Knuth)

Un algorithme est un ensemble de règles qui décrivent une séquence d'opérations en vue de résoudre un problème donné bien spécifié. Un algorithme doit répondre aux 5 caractéristiques suivantes :

- La finitude
- La précision
- Le domaine des entrées
- Le domaine des sorties
- L'exécutabilité

Notons qu'un algorithme exprime donc un procédé séquentiel (or dans la vie courante tout n'est pas nécessairement séquentiel comme par exemple écouter un enseignement et penser aux prochaines vacances), et ne travaille que sur des problèmes déjà transformés de la phase 1 à la phase 2 (la spécification). I

Il n'est pas demandé aux débutants de travailler sur cette étape du processus. C'est pourquoi la plupart des exercices de débutant sont déjà spécifiés dans l'énoncé, ou bien leur spécification est triviale.

Indiquons les éléments de définition des cinq autres caractéristiques demandées à un algorithme :

- **Finitude** : Le nombre d'étapes d'un algorithme doit être fini. Le temps d'exécution pourra être évalué.
- **Précision** : Chaque étape doit être parfaitement définie. Toutes les actions élémentaires doivent être connues.
- **Domaine des entrées** : Le champ des données d'entrée doit être spécifié.
- **Domaine des sorties** : Un algorithme ayant un résultat, il faut donner les champs correspondants aux résultats de sortie, ou du moins les relations entre les données d'entrée et les données de sortie.
- **Exécutabilité** : Un algorithme doit déboucher sur un programme exécutable en un temps fini et raisonnable.

On appelle environnement d'un algorithme l'ensemble des entités utilisés par le processeur pendant le déroulement de l'algorithme.

Nous allons définir un langage de description des algorithmes qui nous permettra de décrire les arbres de programmation et le fonctionnement des machines abstraites de la programmation structurée.

Voici classiquement ce que tous les auteurs utilisent comme système de description d'un algorithme lorsqu'ils le font avec un langage. *Les deux sous-paragraphes qui suivent, fournissent les définitions des éléments fondamentaux d'un tel langage algorithmique, le paragraphe d'après construit un langage algorithmique fondé sur ces éléments fondamentaux*.

Nous verrons que l'algorithmique est par nature plus proche de l'étudiant que la machine. En effet dans la suite du cours, l'étudiant s'apercevra par exemple, que les nombres rationnels ne sont pas représentables simplement en machine, encore moins les nombres réels. Les langages d'implémentations impératifs comme Pascal, Java, C# etc… étant relativement pauvres à cet égard.

L'étudiant ne doit pas croire que l'informatique s'est résignée à ne travailler que sur les entiers et les décimaux, mais plutôt se rendre compte qu'il existe une palette importante de certains produits informatiques qui traitent plus ou moins efficacement les insuffisances des langages classiques par exemple vis à vis des rationnels (les systèmes de calcul formel comme MAPLE (étudié en Taupe),MATHEMATICA,... sont une réponse à ce genre d'insuffisance). Nous ne nous préoccupons absolument pas, dans un premier temps en algorithmique, ni de la vérification, ni du contrôle, ni des restrictions d'implantation des données. Notre préoccupation première est d'écrire des algorithmes justes qui fonctionnent sur des données justes.

3.1 Objets de base d'un langage algorithmique

Contenant

Nous appelons **contenant** toute cellule mémoire d'une machine abstraite d'un niveau fixé.

Contenu

Nous appelons **contenu** l'information représentée par l'état du contenant.

Atomes

Pour un contenant fixé on note A l'ensemble de tous ses états possibles, on dit aussi ensemble des **atomes** du niveau n (niveau du contenant).

Remarques :

a) un atome de niveau n est donc un état possible d'un contenant,
b) pour un niveau logique fixé, il y a un nombre d'atomes fini,
c) lorsque l'on est au niveau machine :

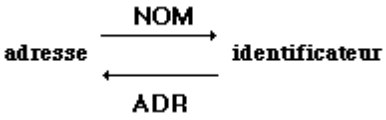
- le contenant est à p positions binaires(p est le nombre de bits du mot, p>1).
- A ={0,1}x....x {0,1} , p fois

Adresse fictive

Toute machine abstraite de niveau fixé dispose d'autant de cellules mémoires que nécessaire. Elles sont repérées par une **adresse fictive** (à laquelle nous n'avons pas accès).

Nom

Par définition, à toute adresse nous faisons correspondre bijectivement par l'opération **nom**, un identificateur unique définissant pour l'utilisateur la cellule mémoire repérée par cette adresse :



Nous définissons aussi un certain nombre de fonctions :

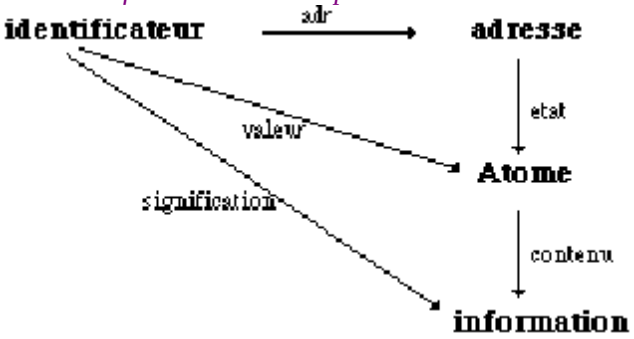
Etat : Adresse →Atome *(donne l'état associé à une adresse)*

valeur: identificateur →Atome *(donne l'état associé à un identificateur, on dit la valeur)*

contenu: Atome →information *(donne le contenu informationnel de l'atome)*

signification: identificateur →information *(sémantique de l'identificateur)*

Ces 4 fonctions sont liées par le schéma suivant :



3.2 Opérations sur les objets de base d'un langage algorithmique

Les parenthèses d'énoncé en LDFA seront algol-like : nous disposerons d'un marqueur du genre **debut** et d'un second du genre **fin** .

Exécutant ou processeur algorithmique

Nous appelons **exécutant** ou **processeur**, la partie de la machine abstraite capable de lire, réaliser, exécuter des opérations sur les atomes de cette machine, ceci à travers un langage approprié.

***Remarque**: l'opérateur formel **exécutant** dépend du temps.*

Instruction simple

C'est une instruction exécutable en un temps fini par un processeur et elle n'est pas décomposable en sous-tâches exécutables ou en autres instructions simples. Ceci est valable à *un niveau fixé*.

Instruction composée

C'est une instruction simple, ou bien elle est décomposable en une suite d'instructions entre parenthèses.

Composition séquentielle

Si i,j,...,t représentent des instructions simples ou composées, nous écrirons la composition séquentielle avec des " ; ". La suite d'instructions " i ; j; ..... ; t " est appelée une **suite d'instructions séquentielles**.

Schéma fonctionnel

C'est :

- soit un identificateur,
- soit un atome,
- soit une application **f** à n variables (où n>0):
  f : (identificateur)^n $\rightarrow$ identificateur

Espace d'exécution

L'espace d'exécution d'une instruction, c'est le n-uplet des n identificateurs ayant au moins une occurrence dans l'instruction (ceci à un niveau fixé).

Soit une instruction i_k, l'ensemble E_k des variables, ayant au moins une occurrence dans l'instruction i_k est noté : $E_k = \{x_1, x_2, ..., x_p\}$ (espace d'exécution de l'instruction i_k)

Environnement

C'est l'ensemble des objets et des structures nécessaires à l'exécution d'un travail donné pour un processeur fixé (niveau information).

Action

C'est l'opération ou le traitement déclenché par un événement qui modifie l'environnement (ou bien toute modification de l'environnement);

Action primitive

Pour un processeur donné(d'une machine abstraite d'un niveau fixé)une action est dite primitive, si l'énoncé de cette action est à lui seul suffisant pour que le processeur puisse l'exécuter sans autre éléments supplémentaires. Une action primitive est décrite par une *instruction simple* du processeur.

Action complexe

Pour un processeur donné(d'une machine abstraite d'un niveau fixé)une action complexe est une action **non-primitive**, qui est **décomposable** en actions primitives (à la fin de la phase de conception elle pourra être exprimée soit par un **module de traitement**, soit par une **instruction composée**).

Remarques :

- Ce qui est action primitive pour une machine abstraite de niveau n, peut devenir une action complexe pour une machine abstraite de niveau n+1, qui est l'expression de la précédente à un plus bas niveau (d'abstraction).
- Les instructions du langage doivent être les mêmes pour tous les niveaux de machine abstraite, sinon la programmation devient trop lourde à gérer.
- Tout langage de description de machine abstraite n'est pas implantable sur ordinateur (*au plus partiellement sinon ce serait tout simplement un langage de programmation*). Il ne peut servir qu'à décrire en partie la spécification et la conception. De plus il doit utiliser les idées de la programmation structurée descendante modulaire.

4. Un langage de description d'algorithme : LDFA

Avertissement

L'apprentissage d'un langage de programmation ne sert qu'aux phases 3 et 4 (traduction et exécution) et ne doit pas être confondu avec l'utilisation d'un langage algorithmique qui prépare le travail et n'est utilisé que comme plan de travail pour la phase de traduction. En utilisant la construction d'une maison comme analogie, il suffit de bien comprendre qu'avant de construire la maison, le chef de chantier a besoin du plan d'architecte de cette maison pour passer à la phase d'assemblage des matériaux ; il en est de même en programmation.

Enonçons un langage simple, extensible, qui est utilisé dans tout le reste du document et qui va servir à décrire les algorithmes. Nous le dénotons dans la suite du document comme **LDFA** pour **L**angage de **D**escription **F**ormel d'**A**lgorithme (terminologie non standard utilisée par l'auteur pour dénommer rapidement un langage algorithmique précis).

4.1 Atomes du LDFA

- Les ensembles de nombres comme **N**,**Z**,**Q**,**R** (les vrais ensembles classiques des mathématiques et leurs structures connues).
- La grammaire mathématique et celle du français.
- {**V**,**F**} comme éléments logiques (({**V**,**F**},¬ , ∧ , ∨) étant une algèbre de Boole)
- Les prédicats.
- Les caractères du français et les chaînes de caractères C des machines.

4.2 Information en LDFA

On rappelle qu'une information en LDFA est obtenue par le contenu d'un atome et se construit à l'aide de :

- la grammaire du français et le sens commun des mots,
- les théorèmes et les résultats obtenus des théories mathématiques(le sens étant le sens habituel donné à tous les symboles),
- toutes les manipulations générales (algorithmes en particulier) sur les structures de données.

4.3 Vocabulaire terminal du LDFA

V_T = **{** <–– , Ω , **lire( ) , ecrire( ) , si , tantque , alors , ftant , faire , fsi , sinon , sortirSi, pour , repeter , fpour , jusque , ; , entrée , sortie , Algorithme , local , global , principal , modules , specifications , types-abstraits , debut , fin , (,) , [,] , * , + , - , / ,** ¬ , ∧ , ∨ **}**

4.4 Instructions simples du LDFA :

Instruction vide

syntaxe : Ω

sémantique : ne rien faire pendant un temps de base du processeur.

Affectation

syntaxe : a <–– α
où : a∈ **identif, et** α est un schéma fonctionnel.

sémantique :
1) si α =identificateur alors val(a)=val(α)
2) si α est un atome alors val(a)=α
3) si α est une application / α : $(id_1,.....,id_p) \rightarrow \alpha(id_1,.....,id_p)$
alors val(a)= α'(val(id_1),.....,val(id_p))
où α' est l'interprétation de α sur l'ensemble des valeurs des val(id_k)

Lecture

syntaxe : **lire**(a) (où a ∈ **identif**)

sémantique : le contexte de la phrase précise où l'on lit pour "remplir" a, sinon on indique **lire**(a) **dans** .....

Elle permet d'attribuer une valeur à un objet en allant lire sur un périphérique d'entrée et elle range cette valeur dans l'objet.

Ecriture

syntaxe : **ecrire**(a) (où a ∈ **identif**)

sémantique : le contexte de la phrase précise où l'on écrit pour "voir" a, sinon on indique **ecrire**(a) **dans** .....

Ordonne au processeur d'écrire sur un périphérique (Ecran, Imprimante, Port, Fichier etc...)

Condition

syntaxe : **si** P **alors** E1 **sinon** E2 **fsi**
où P est un prédicat ou proposition fonctionnelle,
E1 et E2 sont deux instructions composées.

sémantique : classique de l'instruction conditionnelle, si le processeur n'est pas lié au temps on peut écrire :

si P **alors** E1 **sinon** Ω **fsi** = **si** P **alors** E1 **fsi**

Nous notons = la relation d'équivalence entre instructions. Il s'agit d'une équivalence sémantique, ce qui signifie que les deux instructions donnent les mêmes résultats sur le même environnement.

Boucle tantque

syntaxe : **tantque** P **faire** E **ftant**
où P est un prédicat et E une instruction composée)

sémantique :
tantque P **faire** E **ftant** = **si**P **alors** (E ; **tantque** P **faire** E **ftant**) **fsi**

Remarques :

Au sujet de la relation "=" qui est la notation pour l'équivalence sémantique en LDFA, on considère un "programme" LDFA non pas comme une suite d'instructions, mais comme un environnement donné avec un état initial E0 , puis on évalue la modification de cet environnement que chaque action provoque sur lui:
$\{E_0\} \rightarrow \{E_1\} \rightarrow \{E_2\} \rightarrow \ldots\ldots\ldots \rightarrow \{E_k\} \rightarrow \{E_{k+1}\}$

où action n+1 : $\{E_n\} \rightarrow \{E_{n+1}\}$.

On obtient ainsi une suite d'informations sur l'environnement :$(E_0, E_1, \ldots, E_{k+1})$

Nous dirons alors que deux instructions (simples ou composées) sont sémantiquement équivalentes (notation =)si leurs actions associées sur le même environnement de départ provoquent la même modification.

A chaque instruction est associée une action sur l'environnement, c'est le résultat qui est le même (même état de l'environnement avant et après) :
Soient : Instr1 → action1 (*action associée à Instr1*),
Instr2 → action2 (*action associée à Instr2*),

Soient E et E' deux états de l'environnement,
si nous avons : **{E} action1 {E'}** et **{E} action2 {E'} ,**alors Instr1 et Instr2 sont sémantiquement équivalentes, nous le noterons :

Instr1 = Instr2

Boucle répéter

syntaxe : **repeter** E **jusqua** P
(où P est un prédicat et E une instruction composée)

sémantique :
repeter E **jusqua** P = E ; **tantque** **not** P **faire** E **ftant**

Exemple d'équivalence entre itérations:

tantque P **faire** E **ftant** = **si P alors (repeter E jusqua not** P) **fsi**
repeter E **jusqua** P = E ; **tantque not** P **faire E ftant** *(par définition)*

Boucle pour

syntaxe : **pour x** <-- a **jusqua b faire** E **fpour**
(où E est une instruction composée, x une variable, a et b des expressions dans un ensemble fini F totalement ordonné, la relation d'ordre étant notée ≤ , le successeur d'un élément x dans l'ensemble est noté Succ(x) et son prédécesseur pred(x))

sémantiques :
Cette boucle fonctionne à la fois en suivant automatiquement l'ordre croissant dans l'ensemble fini F ou en suivant automatiquement l'ordre décroissant, cela dépendra de la position respective de départ de la borne a et de la borne b. La variable x est appelée un indice de boucle.

sémantique dans le cas ordre croissant à partir du tantque :

```
x <-- a ;
tantque x ≤ succ(b)faire
  E ;
  x <-- succ(x) ;
ftant
```

sémantique dans le cas ordre décroissant à partir du tantque :

```
x <-- a ;
tantque x ≥ pred(b) faire
  E ;
  x <-- pred(x) ;
ftant
```

Exemple simple :

- **E = N** *(entiers naturels)* et la relation d'ordre : ≤ = *inférieur ou égal dans* **N**
- **pour** i ≤ x **jusquà** y **faire** R **FinPour**
  (ici i prendra toutes les valeurs successives dans **N** comprises entre x et y soient, x+y-1 valeurs et s'incrémentera de 1 à chaque fois)

Sortie de boucle

syntaxe : **SortirSi** P (où P est un prédicat ou une instruction vide)ne peut être utilisée qu'à l'intérieur d'une itération (tantque, répéter, pour).

sémantique : termine par anticipation et immédiatement l'exécution de la boucle dans laquelle l'instruction **SortirSi** se trouve.

Exemple récapitulatif complet

Reprenons l'exemple précédent de l'équation du second degré en décrivant dans l'arbre de programmation l'action de la machine abstraite de chaque niveau à l'aide d'instructions du langage algorithmique LDFA :

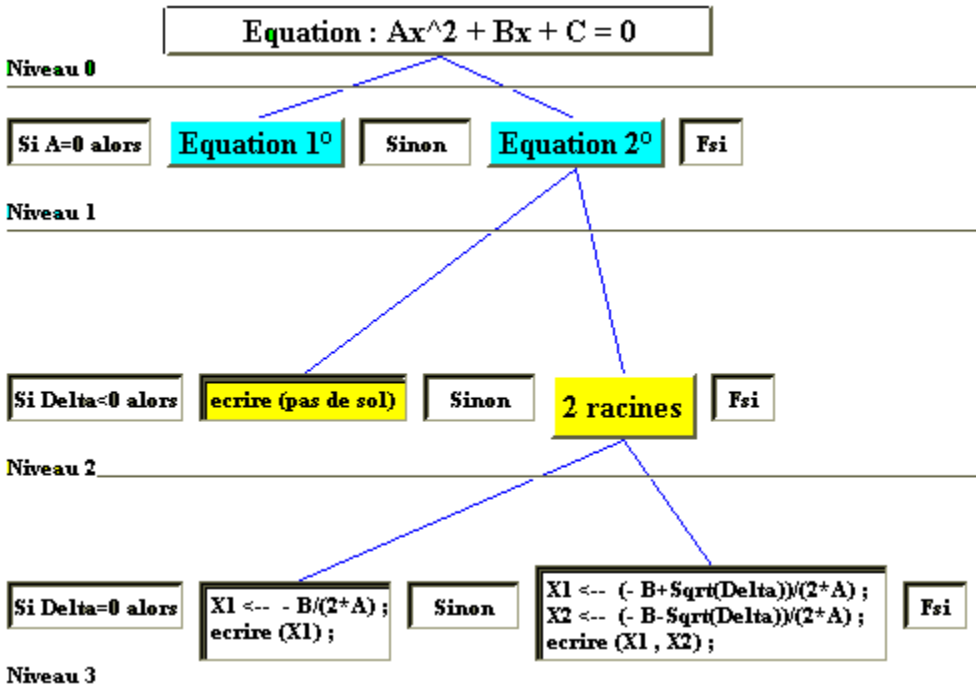


figure de la branche d'arbre 2ème degré

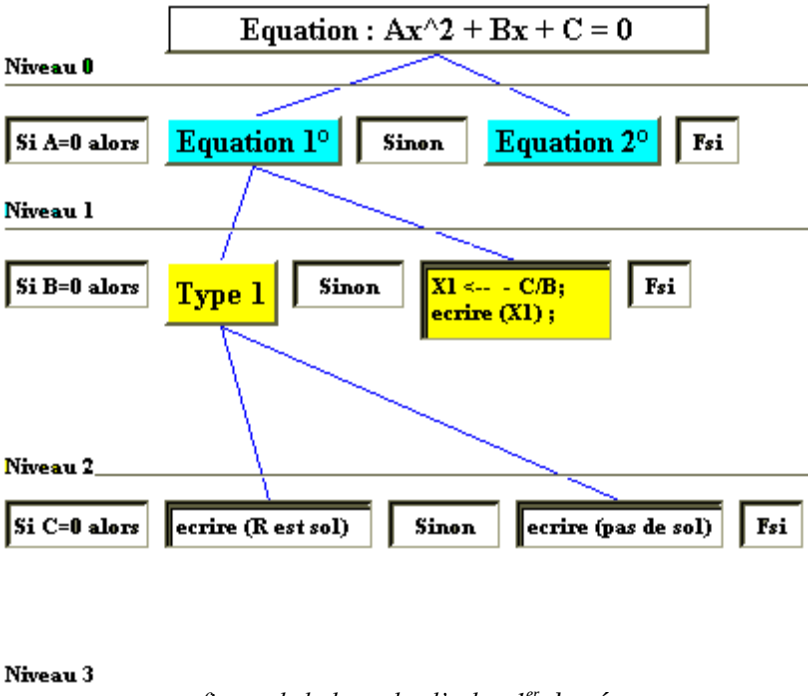


figure de la branche d'arbre 1er degré

Ecriture de l'algorithme

En relisant cet arbre selon un parcours en préordre *(il s'agit de parcourir l'arbre en partant de la racine et descendant toujours par le fils le plus à gauche, puis ensuite de passer au fils droit suivant etc…)*, l'on obtient après avoir complété l'algorithme une écriture linéaire comme suit :

Algorithme Equation
  Entrée: A,B,C ∈ **R**3
  Sortie: X1 ,X2 ∈ **R**2
  Local: Δ ∈ **R**
début
  lire(A,B,C);
  Si A=0 **alors** *{ A=0 }*
    Si B = 0 **alors**
      Si C = 0 **alors**
        écrire(R est solution)
      Sinon *{ C ≠ 0 }*
        écrire(pas de solution)
      Fsi
    Sinon *{ B ≠ 0 }*
      X1 ← -C/B;
      écrire (X1)
    Fsi
  Sinon *{ A ≠ 0 }*

```
    Sinon  { A ≠ 0 }
    |   Δ ← B² - 4*A*C ;
    |   Si Δ < 0 alors
    |         écrire(pas de solution)
    |   Sinon    { Δ ≥ 0 }
    |     Si Δ = 0 alors
    |        X1 ← -B / (2*A);
    |        écrire(X1)
    |     Sinon  { Δ > 0 }
    |        X1 ← (-B+sqrt(Δ)) / (2*A);
    |        X2 ← (-B-sqrt(Δ)) / (2*A);
    |        écrire( X1 , X2 )
    |     Fsi
    |   Fsi
    Fsi
FinEquation
```

Nous regroupons toutes les informations de conception dans un document que nous appelons le dossier de développement.

5. Le Dossier de développement

C'est un document dans lequel se trouvent consignés tous les éléments relatifs à la construction et à l'écriture de l'algorithme et du programme résolvant le problème cherché. Nous le divisons en 5 parties.

5.1 Enoncé et spécification

Enoncé du problème résolu par ce logiciel.

- **Spécifications opérationnelles** des abstractions de plus haut niveau du logiciel, en exprimant celles-ci à l'aide de types abstraits et de spécifications de plus bas niveau.
- **Spécifications des types abstraits de données** utilisés.
- **Spécifications d'interface** pour les abstractions de plus bas niveau.

On utilisera ces trois techniques de spécification de manière descendante, quitte à remonter corriger des spécifications de niveau plus haut lorsque des erreurs seront apparues dans une spécification de plus bas niveau. Ces spécifications sont destinées au niveau " concepteur de logiciel ", plutôt qu'à l'utilisateur. Cette partie rassemble les définitions abstraites des composants. Un utilisateur de base n'ayant à priori pas à consulter ce paragraphe, les termes employés seront les plus rigoureux possibles relativement à un formalisme éventuel.

Analyse des besoins :
son utilité principale est de fournir à l'utilisateur la description des services que lui rendra ce logiciel. Les termes utilisés doivent être compris par l'utilisateur.

5.2 Méthodologie

Dans ce paragraphe se situent tous les documents et les explications qui ont pu mener à la décision de résoudre le problème posé par la méthode que l'on a choisie. Le programmeur dispose ici de toute latitude pour s'exprimer à l'aide de texte en langue naturelle, de représentation graphique, d'outils ou de supports permettant au lecteur de se faire une idée précise du pourquoi des choix effectués.

5.3 Environnement

L'étudiant pourra présenter sous forme d'un tableau les principales informations concernant les données de son algorithme.

Exemple :

Nom	**genre**	**localisation**	**utilisation**
PHT	reel	Entrée	prix hors taxe
TVA	reel	local	TVA en %
PTTC	reel	sortie	Prix TTC

5.4 Algorithme en LDFA

Ici se situe la description de l'algorithme proposé pour résoudre le problème proposé. Il est obtenu entre autre à partir de l'arbre de programmation construit pendant l'analyse et la conception. Ci-dessous le modèle général d'un algorithme :

```
Algorithme XYZT;
    global :
    local :
    entrée :
    sortie :
    modules utilisés :
Spécifications : (TAD)
  Types Abstraits de Données utilisés
début
  ( corps d'algorithme en LDFA)
fin XYZT.
```

Nous verrons ailleurs ce que représentent les notions de TAD et de module.

5.5 Programme en langage de programmation (Pascal par exemple)

Dans ce paragraphe nous ferons figurer la " traduction " en langage de programmation de l'algorithme du paragraphe précédent.

6. Trace formelle d'un algorithme

Et le dernier [précept], de faire partout des dénombrements si entiers, et des revues si générales, que je fusse assuré de ne rien omettre.
R Descartes Discours de la méthode, seconde partie, 1637.

Nous proposons au débutant de vérifier l'exactitude de certaines parties de son algorithme en utilisant un petit outil permettant l'exécution formelle (c'est à dire sur des valeurs algébriques ou symboliques plutôt que numériques) de son algorithme. La trace numérique et les vérifications associées seront effectuées lors de l'exécution par la machine.

6.1 Espace d'exécution d'une instruction composée

On appelle espace d'exécution d'une séquence ou d'un bloc d'instructions $i_1 ... i_n$ l'ensemble $E = \bigcup_{k=1}^{n} E_k$ où E_k est l'espace d'exécution de l'instruction i_k.

Rappelons que l'on peut considérer un " programme " LDFA sous un autre point de vue : non pas comme une suite d'instructions, mais comme un environnement donné avec un état initial E_0 , puis on évalue la modification de cet environnement que chaque instruction provoque sur lui.

On considère les instructions i_k comme des transformateurs d'environnement E_n:
$\{E_0\} \rightarrow \{E_1\} \rightarrow \{E_2\} \rightarrow ..... \rightarrow \{E_k\} \rightarrow \{E_{k+1}\}$

L'instruction i_{n+1} fait alors passer l'environnement de l'état E_n à l'état E_{n+1}.
Nous écrirons ainsi : $i_{n+1} : \{E_n\} \rightarrow \{E_{n+1}\}$

Ces actions déterminent alors une suite d'états de l'environnement $(E_0,E_1,....E_{k+1})$ que l'on peut observer.

C'est ce point de vue qui permet d'exécuter un suivi d'exécution symbolique d'un algorithme. Nous le nommerons " trace formelle ".

On adoptera pour une trace formelle une disposition en tableau de l'espace d'exécution comme suit :

Etats	V_1	V_2	.....	V_n
E_1	--	--		**y**
E_2	**x**	--		**y+1**

La colonne Etats représente donc les états successifs de l'environnement (ou espace d'exécution) figuré ici par les variables $V_1,V_2,...,V_n$. Les contenus des cellules du tableau sont les valeurs symboliques des variables au cours du déroulement de l'exécution. On peut considérer l'image mentale suivante de la trace formelle comme étant une succession de "photographies instantanées" de l'environnement prises après chaque instruction.

6.2 Exemple complet avec trace formelle

Nous traitons un exemple complet avec son dossier de développement et une trace formelle.

Enoncé

Calculer $S=\sum_{i=1}^{n} i$ sans utiliser de formule (*car l'on sait que S=(n+1)n/2*)

Spécification : flux d'information

En Entrée	**En Sortie**
Un nombre $n \in N^*$	Ecrire la somme voulue S.

Méthodologie

S est la somme des termes d'une suite récurrente :

$$s_i \begin{cases} s_0 = 0 \\ s_i = s_{i-1} + i \end{cases}$$

Environnement

Nom	**genre**	**localisation**	**utilisation**
N	Entier	Entrée	Nombre d'éléments à saisir
S	Entier	Sortie	Variable de cumul pour la somme
I	Entier	local	Gestion des boucles : compteur

Algorithme

Algorithme Somentier
| N ∈ $\mathbf{N}^*$
| S , I ∈ $\mathbf{N}^2$
|
Début *{Somentier}*
| *(E_0)*
| **Lire** (N) ;
| *(E_1)* ⟸
| S ← 0;
| *(E_2)* ⟸
| I ← 1;
| *(E_3)* ⟸
| **TantQue** I ≤ *? ? ?* **faire**
| | *(E_4)* ⟸
| | S ← S+I;
| | *(E_5)* ⟸
| | I ← I+1;
| | *(E_6)* ⟸
| **FinTant**;
| *(E_7)* ⟸
| Ecrire(S);
Fin Somentier

Ceci est un algorithme incomplet dans lequel on a déjà intercalé les états (E_n) entre les instructions. On ne sait pas exactement quel sera le test d'arrêt de la boucle (remplacé par ? ? ?), on sait seulement que c'est la valeur de la variable de compteur **I** qui le fournira.

Utilisation de la trace formelle

Nous allons montrer à l'aide de la trace formelle que cet algorithme fournit bien la somme des n premiers entiers dans la variable S, relativement aux préconditions { S = 0 et i = 1}.

Nous allons donc faire de la démonstration de programme :

Précondition	Action	Postcondition
{S= 0 et i= 1}	**Algorithme**	{ S = $\sum_{i=1}^{n} i$ }

Nous pouvons avoir une hésitation quant à la borne du test "**TantQue** I ≤ ??? ", faut-il s'arrêter à N, N-1 ou N+1 ?

Posons comme hypothèse que le test s'arrête à la valeur N, soit : "**TantQue** I ≤ N …

Exécutons manuellement et pas à pas l'algorithme précédent en supposant que le test d'arrêt n'est pas franchi, c'est à dire que l'on a I > N.

Voici le début des résultats de sa trace formelle dans le tableau ci-dessous :

Etats	I	N	S
E_0	-	-	--
E_1	-	**n**	--
E_2	-	**n**	**0**
E_3	**1**	**n**	**0**
$E_4=E_3$	**1**	**n**	**0**
E_5	**1**	**n**	**1**
E_6	**2**	**n**	**1**
$E_4=E_6$	**2**	**n**	**1**
E_5	**2**	**n**	**3**
E_6	**3**	**n**	**3**
$E_4 = E_6$ etc..	**3**	**n**	**3**

isolons les deux premiers " tours " de boucle :

$E_4 = E_6$	**2**	**n**	**1**
$E_4 = E_6$…	**3**	**n**	**3**

Nous voyons que juste avant la sortie de boucle (état E_6) au premier tour I=2 et S=1, au deuxième tour I=3 et S=3 .

Nous posons l'hypothèse de récurrence qu'au kème tour i=k+1 et S= $\sum_{i=1}^{k} i$ (somme des k premiers entiers). Nous allons utiliser l'exécution formelle pas à pas d'un tour de boucle afin de voir si après un tour de plus cette hypothèse se vérifie au rang k+1 :

Etats	I	N	S
.....	**...**	**...**	**...**
$E_4 = E_6$	**k+1**	**n**	**S=** $\sum_{i=1}^{k} i$
E_5	**k+1**	**n**	**S=** $\sum_{i=1}^{k} i$ **+ k+1**
E_6	**k+2**	**n**	**S=** $\sum_{i=1}^{k} i$ **+ k+1**

Or S= $\sum_{i=1}^{k} i$ +k+1 = $\sum_{i=1}^{k+1} i$ (la somme des k+1 premiers entiers).

Nous venons donc de montrer qu'à l'état E6 cet algorithme donne :

Etats	I	N	S
E_6	k+1	n	$S=\sum_{i=1}^{k} i$

En particulier, lorsque k = n nous avons dans S la somme des n premiers entiers $\sum_{i=1}^{n} i$:

Etats	I	N	S
E_6	n+1	n	$S=\sum_{i=1}^{n} i$

Nous pouvons déjà écrire que : $\forall$ n, n > 0, $S = \sum_{i=1}^{n} i$

En plus ce dernier tableau nous permet immédiatement de trouver la valeur exacte de la variable de contrôle de la boucle (ici la variable I qui vaut n+1) et donc d'écrire un test d'arrêt de boucle juste.

On peut alors choisir comme test I<>n+1 ou bien I< n+1 etc... ou tout autre prédicat équivalent.

Il était possible de programmer directement cet algorithme avec les deux autres boucles (pour... et répeter...). Ceci est proposé en exercice au lecteur.

7. Traducteur élémentaire LDFA - Java / Pascal

Nous venons de voir qu'un algorithme devait se traduire en langage de programmation (dit évolué). Nous fournirons ici un tableau qui sera utile à l'étudiant pour la traduction des instructions algorithmiques en langage de programmation.

7.1 Traducteur

Afin de bien montrer que l'écriture algorithmique est plus abstraite qu'un langage de programmation nous donnons un tableau de traduction LDFA dans deux langages : en Pascal de base et en Java2 restreint aux instructions seulement: (dans le tableau, P est un prédicat et E une instruction composée)

LDFA	Java	Pascal
Ω (instruction vide)	pas de traduction	pas de traduction
debut i1 ; i2; i3; ..... ; ik **fin**	**{** i1 ; i2; i3; ..... ; ik **}**	**begin** i1 ; i2; i3; ..... ; ik **end**
x ← a	x = a ;	x := a
;	pas de traduction	(ordre d'exécution) ;
Si P **alors** E1 **sinon** E2 **Fsi**	**if** (P) E1 ; **else** E2 ; *(attention, pas de fermeture !)*	**if** P **then** E1 **else** E2 *(attention, pas de fermeture !)*
Tantque P **faire** E **Ftant**	**while** (P) E ; *(attention, pas de fermeture)*	**while** P **do** E *(attention, pas de fermeture)*
répeter E **jusquà** P	**do** E ; **while** (! P) ;	**repeat** E **until** P
lire (x1,x2,x3.....,xn)	System.in.read( ) ; System.in.readln( ) ;	read(fichier,x1,x2,x3.....,xn) readln(x1,x2,x3.....,xn) Get(fichier)
ecrire (x1,x2,x3.....,xn)	System.in.print( ) ; System.in. println( ) ;	write(fichier,x1,x2,x3.....,xn) writeln(x1,x2,x3.....,xn) Put(fichier)
pour x ← a **jusquà** b **faire** E **Fpour**	**for** (**int** x= a; x <= b; x++) E ;	**for** x:=a **to** b **do** E *(croissant)* **for** x:=a **downto** b **do** E *(décroissant)* *(attention, pas de fermeture)*
SortirSi P	**if** (P) break **;**	**if** P **then** Break

Ce tableau de traduction permet déjà d'écrire très rapidement des programmes Pascal et Java simples à partir d'algorithmes étudiés et écrits.

7.2 Exemple

En appliquant le traducteur à l'algorithme de l'équation du second degré nous obtenons le programme Pascal suivant :

```
program equation;
var
   A,B,C:real;
   X1,X2:real;
   Delta:real;
begin
   readln(A,B,C);
   if A = 0 then {A=0}
     if B = 0 then
       if C = 0 then
          writeln('R est solution')
       else
          writeln('pas de solution')
```

```
    else
    begin
      X1 := - C/B;
      writeln('x=',X1)
    end
else
begin
  Delta := B*B-4*A*C;
  if Delta < 0 then
       writeln('pas de solution')
  else
   if Delta=0 then
   begin
     X1 := -B/(2*A);
     writeln('x=',X1)
   end
   else
   begin
     X1 := (-B + Sqrt(Delta)) / (2*A);
     X2 := (-B - Sqrt(Delta)) / (2*A);
     writeln('x1=',X1,'x2=',X2)
   end
end
end.
```

En appliquant le traducteur Java2 à ce même algorithme de l'équation du second degré, nous obtenons le squelette de programme Java2 suivant :

```
if (a ==0)
  if (b ==0)
    if (c ==0)
      System.out.println("tout reel est solution") ;
    else
     System.out.println("il n'y a pas de solution") ;
  else   {
     x = -c/b ;
     System.out.println("la solution est " + x) ;
    }
 else   {
    delta = b*b -4*a*c ;
    if (delta <0)
      System.out.println("il n'y a pas de solution dans les reels") ;
    else
      if (delta == 0)   {
         x1 = -b / (2*a) ;
         System.out.println("il y a une solution double : "+x1) ;
      }
       else    {
      x1 = (-b + Math.sqrt(delta)) / (2*a) ;
      x2 = (-b - Math.sqrt(delta)) / (2*a) ;
      System.out.println("il y deux solutions égales a "+x1+" et " + x2) ;
      }
 } etc…
```

7.3 Sécurité et ergonomie

L'utilisation du traducteur manuel LDFA –—> Pascal fournit une version préliminaire de programme pascal fonctionnant sur des données correctes sans aucune présentation.

Il appartient au programmeur de compléter dans un deuxième temps la partie sécurité associée aux contraintes du domaine de définition des variables et aux contraintes matérielles d'implantation. Enfin, dans un troisième temps, l'ergonomie (forme de l'échange d'information entre le programme et le futur utilisateur) sera envisagée et programmée.

Voyons sur l'exemple de la somme des n premiers entiers déjà cité plus haut, comment ces trois étapes s'articulent .

Etape de traduction-*somme des n premiers entiers*

Texte final de l'algorithme de départ :	Texte de sa traduction en pascal :

Algorithme Somentier
N ∈ **N***
S , I ∈ **N**^{2}
Début *{Somentier}*
Lire (N) ;
S ← 0;
I ← 1;
TantQue I < N+1 **faire**
S ← S+I;
I ← I+1;
FinTant;
Ecrire(S);
Fin Somentier
	program Somentier ;
	var N : integer ;
	S,I : integer ;
	begin
	readln(N) ;
	S :=0 ;
	I :=1 ;
	while I < N+1 **do**
	begin
	S := S +I;
	I := I+1;
	end;
	writeln(S)
	end.

Etape de sécurisation-*somme des n premiers entiers*

Sécurité due aux domaines de définition des données

La traduction ne permet pas d'écrire les domaines de définition des variables : en l'occurrence ici la variable N∈**N*** est traduite par " **var** N : **integer** ", or le type prédéfini **integer** est un sous-ensemble des entiers relatifs **Z**, il est donc nécessaire d'éliminer les entiers négatifs ou nuls comme choix possible.

Dès que l'utilisateur aura entré son nombre, le programme devra tester l'appartenance au bon intervalle afin de protéger la partie de code, par exemple avec une instruction de condition :

```
if  N > 0  then begin

// code protégé
end
```

Protection programmée dans les pointillés en dessous dans le cadre de droite :

```
program Somentier ;
var N : integer ;
    S,I : integer ;
begin
readln(N) ;

S :=0 ;
I :=1 ;
while I < N+1 dobegin
  S := S + I;
  I := I+1;
end;
writeln(S)

end.
```

```
program Somentier ;
var N : integer ;
    S,I : integer ;
begin
readln(N) ;
if  N > 0  then begin
    S :=0 ;
    I :=1 ;
    while I < N+1 do begin
      S := S + I;
      I := I+1;
    end;
    writeln(S)
 end
end.
```

Sécurité due aux contraintes d'implantation

Si nous exécutons ce programme pour la valeur N=500, la valeur fournie en sortie est " -5822 " sur un pascal 16 bits comme TP-pascal, le résultat n'est pas correct. Nous sommes confrontés au problème de la représentation des entiers machines déjà cité. Ici le type **integer** est restreint à l'intervalle *[-32768,+32767]* ; il y a manifestement dépassement de capacité (overflow) et le système a allègrement continué les calculs malgré ce dépassement. En effet, la somme vaut 500*501/2 soit **125250**, cette valeur n'appartient pas à l'intervalle des **integer**.

Le programmeur doit donc remédier à ce problème par un effort personnel de sécurisation de son programme en n'autorisant les calculs que pour des valeurs valides offrant un maximum de sécurité.

Ici la variable S contient la somme $\sum_{i=1}^{k} i$, nous savons que $\sum_{i=1}^{k} i$ = k(k+1)/2, donc il suffira de résoudre dans **N** l'inéquation $k(k+1)/2 \leq 32767$ où n est l'inconnue. L'unique solution positive a pour partie entière 255, qui est la valeur maximale avant dépassement de capacité. Donc il suffit de protéger le code par un test supplémentaire sur la variable N :

```
if (N > 0) and (N < 256) then begin
  S :=0 ;
  I :=1 ;
  while I< N+1 do begin
      S := S +I;
      I := I+1;
   end;
   writeln(S)
end
```

En vérifiant sur l'exécution, nous trouvons que S = 32640 pour N = 255. Ce qui nous donne la version suivante du programme :

```
program  Somentier ;
var N : integer ;
      S , I : integer ;
begin
   readln(N) ;
   if (N > 0) and (N < 256) then begin
         S :=0 ;
         I :=1 ;
         while I< N+1 do begin
            S := S +I;
            I := I+1;
         end;
         writeln(S)
   end
end.
```

Etape d'ergonomie-*somme des n premiers entiers*

Dans cet exemple, l'information à échanger avec l'utilisateur est très simple et ne nécessite pas une interface spéciale. Il s'agira de lui préciser les contraintes d'entrée et de lui présenter d'une manière claire le résultat.

```
program  Somentier ;
var N : integer ;
      S , I : integer ;
begin
   Write('Entrez un entier entre 0 et 255') ;
   readln(N) ;
   if (N > 0) and (N < 256) then begin
         S :=0 ;
         I :=1 ;
         while I< N+1 do begin
            S := S +I;
            I := I+1;
         end;
         writeln('la somme des ',N,' premiers entiers vaut ',S)
   end
   else
      writeln('Calcul impossible ! !')
end.
```

Vous remarquerez que les adjonctions supplémentaires de code *(en italique)* dans le programme final se montent à environ 50% du total du code écrit, car un logiciel n'est pas uniquement un algorithme traduit.

En continuant d'appliquer le principe de la programmation structurée, il est bon de bien séparer lors du développement la partie algorithmique des parties sécurité et ergonomie. Le programmeur débutant y gagnera en clarté dans sa méthode de travail.

8. Facteurs de qualité du logiciel

B.Meyer et G.Booch

Constat

Un utilisateur, lorsqu'il achète un produit comme un appareil électro- ménager ou une voiture, attend de son acquisition qu'elle possède un certain nombre de **qualités** (fiabilité, durabilité, efficacité, ...). Il en est de même avec un logiciel.

Voici une liste minimale de critères de qualité du logiciel (proposée B.Meyer, G.Booch):

Correction	**Robustesse**	**Extensibilité**
Réutilisabilité	**Compatibilité**	**Efficacité**
Portabilité	**Vérificabilité**	**Intégrité**
Facilité utilisation	**Modularité**	**Lisibilité**
Abstraction		

Reprenons les définitions communément admises par ces deux auteurs sur ces facteurs de qualité.

Correction — La **correction** est la qualité qu'un logiciel a de respecter les spécifications qui ont été posées.

Robustesse — La **robustesse** est la qualité qu'un logiciel a de fonctionner en se protégeant des conditions de dysfonctionnement.

Extensibilité — **L'extensibilité** est la qualité qu'un logiciel a d'accepter des modifications dans les spécifications et des adjonctions nouvelles.

Réutilisabilité — La **réutilisabilité** est la qualité qu'un logiciel a de pouvoir être intégré totalement ou partiellement sans réécriture dans un nouveau code.

Compatibilité — La **compatibilité** est la qualité qu'un logiciel a de pouvoir être utilisé avec d'autres logiciels sans autre effort de conversion des données par exemple.

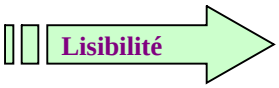
L'efficacité est la qualité qu'un logiciel a de bien utiliser les ressources.

La **portabilité** est la qualité qu'un logiciel a d'être facilement transféré sur de nombreux matériels, et insérable dans des environnements logiciels différents.

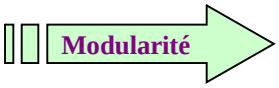
La **vérificabilité** est la qualité qu'un logiciel a de se plier à la détection des fautes, au traçage pendant les phases de validation et de test.

L'intégrité est la qualité qu'un logiciel a de protéger son code et ses données contre des accès non prévus.

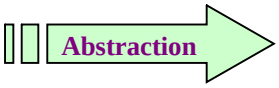
La **facilité** d'utilisation est la qualité qu'un logiciel a de pouvoir être appris, utilisé, interfacé, de voir ses résultats rapidement compris, de pouvoir récupérer des erreurs courantes.



La **lisibilité** est la qualité qu'un logiciel a d'être lu par un être humain.



La **modularité** est la qualité qu'un logiciel a d'être décomposable en éléments indépendants les uns des autres et répondants à un certain nombre de critères et de principes.



L'abstraction est la qualité qu'un logiciel a de s'attacher à décrire les opérations sur les données et à ne manipuler ces données qu'à travers ces opérations.

La production de logiciels de qualité n'est pas une spécificité des professionnels de la programmation ; c'est un état d'esprit induit par les méthodes du génie logiciel. Le débutant peut, et nous le verrons par la suite, construire des logiciels ayant des " qualités " sans avoir à fournir d'efforts supplémentaires.

Bien au contraire la réalité a montré que les étudiants " bricoleurs " passaient finalement plus de temps à " bidouiller " un programme que lorsqu'ils décidaient d'user de méthode de travail. Une amélioration de la qualité générale du logiciel en est toujours le résultat.

Machines abstraites : exemple

Traitement descendant modulaire d'un exemple complet

Objectif : développer un exemple simple de construction d'une machine abstraite par décomposition descendante sur 4 niveaux.

ENONCE

On donne une liste de n noms (composés de lettres uniquement). Extrayez ceux qui sont le premier et le dernier par ordre alphabétique. Ecrire un programme Pascal effectuant cette opération.

SPECIFICATIONS :(il s'agit d'éclaircir certaines décisions)

Plan:

Objets utilisés,
machine abstraite,
spécification de données.

Objets utilisés au niveau 1

Identification	Signification
(**Liste** , <<)	**Liste** est un ensemble fini de noms où << est une relation d'ordre total
Noms	Ensemble de tous les noms possibles (chacun est constitué de lettres)
élément	Fonction fournissant le kième élément de la liste : **élément : N*** x Liste → Liste
Grand	Un élément de l'ensemble ***Noms*** : Grand ∈ ***Noms***
Petit	Un élément de l'ensemble ***Noms*** : Petit ∈ ***Noms***
Long	Fonction fournissant le nombre d'éléments de la liste : Long : Liste → **N***

Machine abstraite de niveau 1

(description de haut niveau d'abstraction de l'algorithme choisi)

```
Grand ← élément ( 1 , Liste) ;
Petit  ← élément ( 1 , Liste) ;
Pour indice  ← 2 jusquà Long (Liste) faire
  Si Grand << élément (indice, Liste) alors
      Grand  ← élément (indice, Liste)
  fsi ;
  Si élément (indice ,Liste) << Petit alors
       Petit  ← élément (indice, Liste)
  fsi
Fpour ;
```

{Grand = le dernier et Petit = le premier }

Les données au niveau 1

Identification	Signification
Noms	Un ensemble de caractères
(**Liste** , <<)	**Liste** est un ensemble fini muni d'une relation d'ordre total <<
élément	Fonction **élément : N*** x Liste → Liste (k, Liste) → **élément** (k, Liste) ∈ Liste
Long	Fonction fournissant le nombre d'éléments de la liste : Long : Liste → **N*** Liste → Long (Liste) = n

Nous avons ici une spécification abstraite de haut niveau. Il est impératif de prendre des décisions sur les structures de données qui vont être utilisées. Nous allons envisager le cas le plus simple : celui où la structure choisie pour représenter la liste est un tableau.

Les données au niveau 2

Reprise des objets abstraits en les exprimant de la façon suivante :

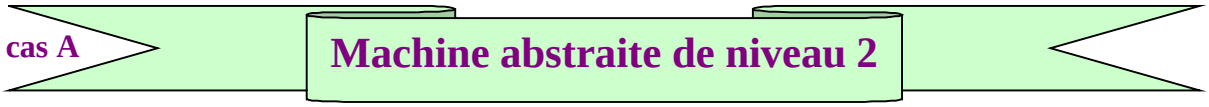
Cas A où la version pascal contient déjà les outils de chaînes

- Liste = Tableau
- élément (i, Liste) = Liste[i]
- Long(Liste) = n , taille du tableau
- ***Noms*** : Type string
- << : ≤ (relation de comparaison lexicographique sur les chaînes)

Nous continuons à descendre dans les niveaux d'abstraction. Nous devons prendre des décisions sur le langage-cible. Il est dit dans l'énoncé que ce doit être Pascal, mais lequel ?

Nous avons choisi dans ce premier cas, une version simple en prenant par exemple comme dialecte deux descendants de l'UCSD-Pascal, à savoir Think Pascal (Mac) ou Borland Pascal-Delphi (Windows-Linux) qui contiennent en prédéfini le type de chaîne de caractères.

Ces spécifications de données étant établies, la machine précédente devient :



Algorithme EXTRAIT0
Global : **n** ∈ **N*** , Long_mot ∈ **N***
Local : indice ∈ **N***, (Grand , Petit) ∈ **Noms** ^{2}
Spécification:
Noms = Type **string** prédéfini par une version d'implémentation du pascal.
Liste = Tableau de Nom, de Taille **n** ∈ **N*** fixée.
Début

```
Grand ← Liste[1] ;
Petit ← Liste[1] ;
Pour indice ←2 jusquà n faire
    Si Grand < Liste[indice] alors Grand ← Liste[indice] fsi ;
    Si Liste[indice] < Petit alors Petit ← Liste[indice] fsi
Fpour ;
```

Fin EXTRAIT0

Cet algorithme se traduit immédiatement en pascal. Nous voyons donc qu'il nous a été possible d'écrire ce petit programme en descendant uniquement sur 2 niveaux de spécifications.

Qu'en est-il lorsque l'on travaille avec un autre langage cible (nous allons juste utiliser une version différente du même langage cible) ? Nous allons voir que nous devrons descendre alors plus loin dans les spécifications et développer plus de code c'est l'objectif de la suite de ce document.

Cas B où la version pascal ne contient pas les outils de chaînes

Reprenons partiellement la même spécification de données par un tableau de taille n pour la Liste, mais supposons que le langage-cible soit du **Pascal ISO**, dans lequel le type string n'existe pas.

cas B

Les données au niveau 2

Cas B où la version pascal ne contient pas d'outils de chaînes

- ***Noms*** : Nous choisissons, afin de ne pas nous perdre en complexités inutiles, de spécifier la *l'ensemble des caractères* par *un tableau de caractères :* notons le **Tchar**
- Liste = Tableau de **Tchar**
- élément (i, Liste) = Liste[i]
- Long(Liste) = n , taille du tableau .
- << : CMP (opérateur de relation de comparaison sur les **Tchar**)

cas B

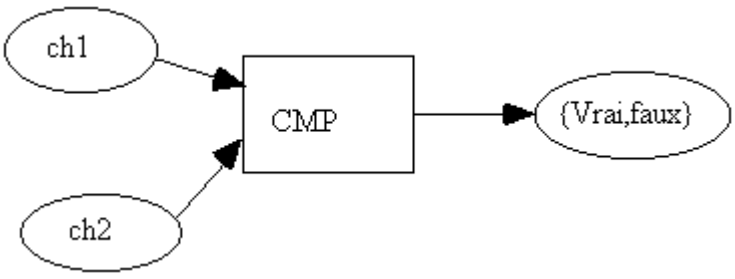
Machine abstraite CMP

Ces choix de spécification induisent un choix de développement d'une machine abstraite spécifique à la relation d'ordre << , qui n'est pas un opérateur simple du langage : nous avons dénoté la machine par le nom CMP.

Noms = Tchar
Taille de Tchar = n
ch1 ∈ Tchar , ch2 ∈ Tchar

Spécification de l'opérateur 'CMP 'de comparaison de chaînes :

CMP : **Tchar** x **Tchar** → { **Vrai** , **Faux** }
CMP (ch1,ch2) = **Vrai** **ssi** (ch1 < ch2) ou (ch1 = ch2)
CMP (ch1,ch2) = **Faux** **ssi** ch2 < ch1



ch1 ∈ ***Noms***,
ch2 ∈ ***Noms***

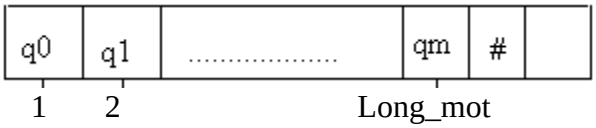
Descendons dans les spécifications plus concrètes de la machine abstraite de comparaison, spécifions d'une manière plus détaillée, les données **Noms** et **Liste** de la machine abstraite CMP, en répondant aux deux questions ci-dessous :

Noms = Tableau de caractères noté **Tchar** , comment est-il représenté ?
Liste = Tableau de **Tchar** , comment est-il représenté ?

cas B

Les données au niveau 3

Un nom **Noms** :



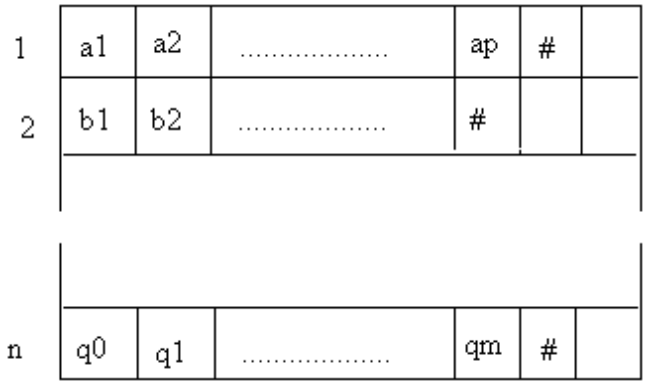
Noms = Tableau de caractères

Noms [i] = le caractère de rang i-1

Attributs :

- **Taille = Long_mot**
- **caractère spécial = #**

Une liste de noms **Liste** :



Liste = Tableau de noms

Liste[i] = le **Noms** de rang i

Attribut :

Taille = n

On dispose d'une relation d'ordre sur les caractères (ordre ASCII) notée ≤ .

cas B

Machine CMP de niveau 3

Décrivons une première version de CMP en tenant compte des spécifications de données précédentes.

Tantque (les caractères lus de ch1 et de ch2 sont les mêmes)
et (ch1 non entièrement exploré) **et** (ch2 non entièrement exploré) **faire**
passer au caractère lu suivant dans ch1 ;
passer au caractère lu suivant dans ch2 ;
Ftant ;
Si (ch1 et ch2 finis en même temps) **alors** ch1=ch2 **fsi** ;
Si (ch1 fini avant ch2) **ou** (car_Lu de ch1 < car_Lu de ch2) **alors** ch1 < ch2 **fsi** ;
Si (ch2 fini avant ch1) **ou** (car_Lu de ch2 < car_Lu de ch1) **alors** ch2 < ch1 **fsi** ;

Descendons plus bas dans les niveaux d'abstraction.

cas B

Machine CMP de niveau 4

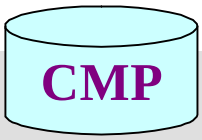
Notre travail va consister à expliciter, à l'aide des structures de données choisies les phrases de haut niveau d'abstraction de la spécification de niveau 3 de CMP (en italique le niveau 3, en gras le niveau 4):

spécification de niveau 3 de CMP	spécification de niveau 4 de CMP
car_Lu de ch1	**ch1[i]** (ième caractère de ch1)
car_Lu de ch2	**ch2[k]** (kème caractère de ch2)
ch1 fini ou entièrement exploré	**ch1[i] = #**
Ch2 fini ou entièrement exploré	**ch2[k] = #**
les caractères lus de ch1 et de ch2 sont les mêmes	**ch1[i] = ch2[i]**
caractère suivant de ch1 , ch2	Si caractère actuel = **ch1[k]** alors caractère suivant = **ch1[k+1] ,** **idem pour ch2**

La spécification opérationnelle de niveau 4 de CMP devient alors :

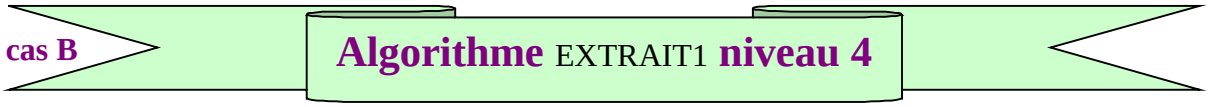
Tantque (ch1[k] = ch2[k]) et (ch1[k] ≠ #) et (ch2[k] ≠ #) **faire**
  k ← k+1
Ftant ;
Si (ch1[k] = #) **et** (ch2[k] = #) **alors** CMP ← Vrai **fsi** ;
Si (ch1[k] = #) **ou** (ch1[k] < ch2[k]) **alors** CMP ← Vrai **fsi** ;
Si (ch2[k] = #) **ou** (ch2[k] < ch1[k]) **alors** CMP ← Faux **fsi** ;

Il faut prévoir d'initialiser le processus au premier caractère k=1 d'où maintenant une spécification de l'algorithme :



Algorithme CMP
Local : k ∈ **N***
entrée : (ch1 , ch2) ∈ **Noms**^{2}
sortie : CMP ∈ { Vrai, Faux }
Spécification:
Noms = Tableau de Taille Long_mot fixée disposant d'un caractère de fin (#)
Début
 k ← 1 ;
 Tantque(ch1[k] = ch2[k]) et (ch1[k] ≠ #) et (ch2[k] ≠ #) **faire**
    k ← k+1
 Ftant ;
 Si(ch1[k] = '#') **et** (ch2[k] = '#') **alors** CMP ← Vrai **fsi** ;
 Si (ch1[k] = '#') **ou** (ch1[k] < ch2[k]) **alors** CMP ← Vrai **fsi** ;
 Si (ch2[k] = '#') **ou** (ch2[k] < ch1[k]) **alors** CMP ← Faux **fsi** ;
Fin_CMP.

Puis en intégrant la machine abstraite CMP de niveau 4 avec les spécifications de TAD décrites précédemment, le tout dans la spécification de niveau 3 de l'algorithme choisi, nous obtenons l'algorithme final suivant :



Algorithme EXTRAIT1
Global : n ∈ **N*** , Long_mot ∈ **N***
Local : indice ∈ **N***, (Grand , Petit) ∈ **Noms**^{2}

module utilisé : 

Spécification:
Noms = Tableau de caractères, de Taille Long_mot fixée disposant d'un attribut marqueur de fin, qui est le caractère spécial # .
Liste = Tableau de ***Noms***, de Taille **n** ∈ **N*** fixée.

TAD utilisés:
- Tableau de caractère de dimension 1.
- Tableau de ***Noms*** de dimension 1.

Début

```
Grand ← Liste[1] ;
Petit ← Liste[1] ;
Pour indice ← 2 jusquà n faire
  Si CMP(Grand , Liste[indice] ) alors Grand ← Liste[indice] fsi ;
  Si CMP(Liste[indice] , Petit ) alors Petit ← Liste[indice] fsi
Fpour ;
```

Fin_EXTRAIT1.

Voici une traduction possible en Pascal de cet algorithme.

```
Program EXTRAIT1;

Const
 Taille = 5;
 Long_mot = 20;
Type
   Nom = array[1..Taille] of Char;
   List_noms = array[1..Long_mot ] of Nom;
Var
   Liste : List_noms;
   indice : integer;
   Grand,Petit : Nom;
```

```
function CMP (ch1,ch2:Nom):Boolean;
var
  k : integer;
begin
   k:=1
   While (ch1[k]=ch2[k])and(ch1[k]<'#')and(ch2[k]<'#') do k:=k+1;
   if (ch1[k]='#')and(ch2[k]='#') then result :=True;
   if (ch1[k]='#')or(ch1[k]<ch2[k]) then result :=True;
   if (ch2[k]='#')or(ch2[k]<ch1[k]) then result :=False;
end;{CMP}

procedure INIT_Liste;
begin
      {initialise la liste des noms terminés par des #}
end;

procedure ECRIRE_Nom (name:Nom);
begin
    {écrit sur une même ligne les caractères qui composent la variable name, sans le #}
end;{ECRIRE_Nom}

Begin{EXTRAIT}
   INIT_Liste;
   Grand:=Liste[1];
   Petit:=Liste[1];
   for indice:=2 to taille do
   begin
     if CMP(Liste[indice],Petit) then Petit := Liste[indice];
     if CMP(Grand,Liste[indice]) then Grand := Liste[indice];
   end;
   write('Le premier est : ');
   ECRIRE_Nom(Petit);
   write('Le dernier est : ');
   ECRIRE_Nom(Grand);
End.{ EXTRAIT }
```

Le lecteur comprendra à partir de cet exemple que les langages de programmation sont très nombreux et que le choix d'un langage pour développer la solution d'un problème est un élément important.

Autres versions possibles à partir de CMP

La version d'implantation de CMP du niveau 4' a été conçue sur une structure de données tableau terminée par une sentinelle (le caractère #). Elle a été implantée par une fonction en pascal.

Il est possible de réécrire d'autres version d'implantation de cette même machine CMP avec des structures de données différentes comme un tableau avec un attribut de longueur ou bien une structure liste dynamique :

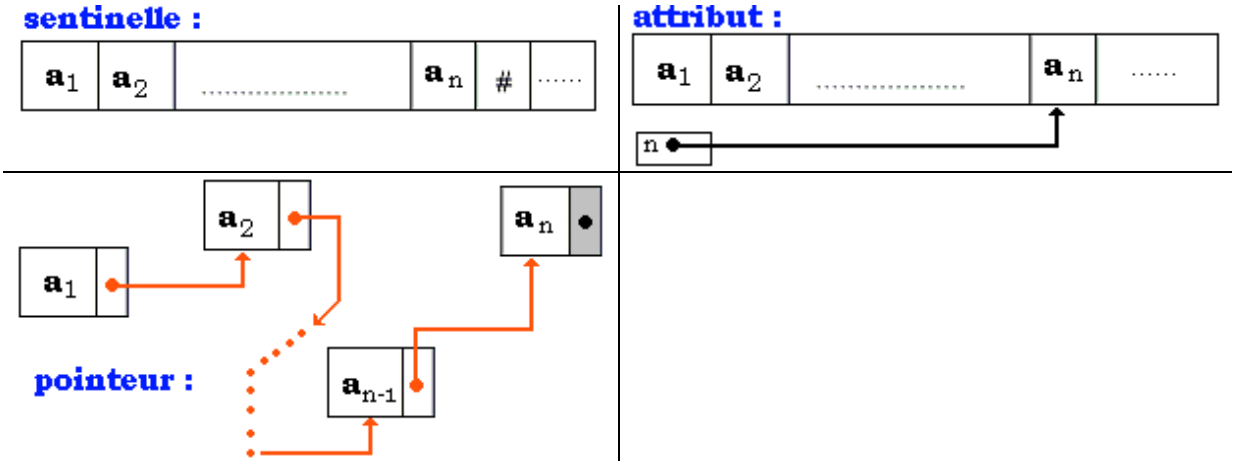
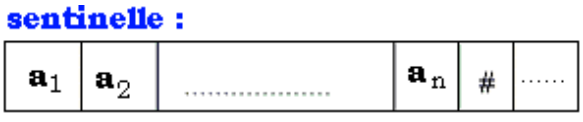


Fig - schéma des trois représentations des données (a_1, … , a_n)

Nous engageons le lecteur à écrire à chaque fois l'algorithme associé et à le traduire en un programme pascal.

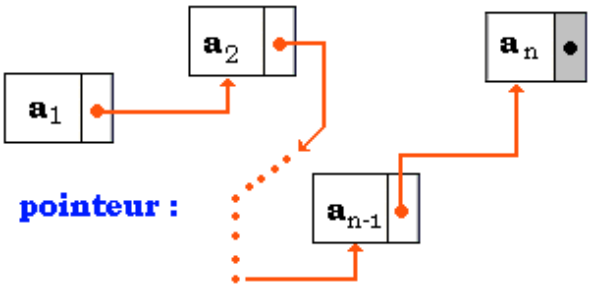
Nous donnons ci-après, au lecteur les trois versions d'implantation en pascal de la fonction CMP associée (sentinelle, pointeur, attribut).

CMP
programme avec sentinelle



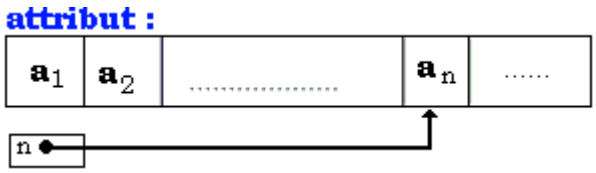

```
function CMP(ch1,ch2:Nom):Boolean;
var
   k : integer;
begin
  k:=1
  While (ch1[k]=ch2[k])
            and(ch1[k]<'#')
            and(ch2[k]<'#') do
     k:=k+1;
  if (ch1[k]='#')and(ch2[k]='#') then
        result :=True;
  if (ch1[k]='#')or(ch1[k]<ch2[k]) then
        result :=True;
  if (ch2[k]='#')or(ch2[k]<ch1[k]) then
        result :=False;
end;{CMP}
```

CMP
programme avec pointeur




```
type pchaine=^chaine;
 chaine=record
   car:char;
   suiv:pchaine;
 end;


function CMP(ch1,ch2:Nom):Boolean;
begin
 while ((ch1^.car=ch2^.car)
    and (ch1^.suiv<nil)
    and (ch2^.suiv<nil))      do
 begin
  ch1:=ch1^.suiv;
  ch2:=ch2^.suiv;
 end;
 if ((ch1^.suiv=nil) and (ch2^.suiv=nil)) then
CMP:=true;
 if (((ch1^.suiv=nil) and (ch2^.suiv<nil))
    or (ch1^.car<ch2^.car)) then result:=true;
 if (((ch2^.suiv=nil) and (ch1^.suiv<nil))
  or (ch1^.carch2^.car)) then result:=false;
end;{CMP}
```

CMP
programme avec attribut




```
const MaxCar=1000;
type inter=0..MaxCar;
chaine=record
  long:integer;
  car:array[1..MaxCar] of char;
end;


function CMP(ch1,ch2:Nom):Boolean;
var n:integer;
begin
 n:=1;
 while (ch1.car[n]=ch2.car[n])
          and ((n<n1)
          and (n<n2)) do
      n:=n+1;
 if ((n=ch1.long) and (n=ch2.long)) then
      result :=true;
 if (((n=ch1.long) and (n<ch2.long))
          or (ch1.car[n]<ch2.car[n])) then
     result:=true;
 if((n=ch2.long) and (n<ch1.long))
         or (ch1.car[n]ch2.car[n]) then
     result:=false;
end;{CMP}
```

3.2 : Modularité

Plan du chapitre:

1. La modularité

1.1 Notion de module

1.2 Critères principaux de modularité
- **La décomposabilité modulaire**
- **La composition modulaire**
- **La continuité modulaire**
- **La compréhension modulaire**
- **La protection modulaire**

1.3 Préceptes minimaux de construction modulaire
- **Interface de données minimale**
- **Couplage minimal**
- **Interfaces explicites**
- **Information publique et privée**

2. La modularité par les unit en pascal UCSD

2.1 Partie " public " d'une UNIT : " Interface "
2.2 Partie " privée " d'une UNIT : " Implementation "
2.3 Partie initialisation d'une UNIT

1. Modularité (selon B.Meyer)

1.1 Notion de module

Le mot **MODULE** est un des mots les plus employés en programmation moderne. Nous allons expliquer ici ce que l'on demande, à une méthode de construction modulaire de logiciels, de posséder comme propriétés, puis nous verrons comment dans certaines extensions de Pascal sur micro-ordinateur (Pascal, Delphi), cette notion de module se trouve implantée.

B.Meyer est l'un des principaux auteurs avec G.Booch qui ont le plus travaillé sur cette notion. Le premier a implanté ses idées dans le langage orienté objet " Eiffel ", le second a utilisé la modularité du langage Ada pour introduire le concept d'objet qui a été la base de méthodes de conception orientées objet : OOD, HOOD,UML...

Nous nous appuyons ici sur les concepts énoncés par B.Meyer fondés sur 5 critères et 6 principes relativement à une méthodologie d'analyse de type modulaire. Une démarche (et donc le logiciel construit qui en découle) est dite modulaire si elle respecte au moins les concepts ci-après.

1.2 Critères principaux de modularité

Les 5 principes retenus :

- **La décomposabilité modulaire**
- **La composition modulaire**
- **La continuité modulaire**
- **La compréhension modulaire**
- **La protection modulaire**

Définitions et réalisations en Pascal de ces cinq principes

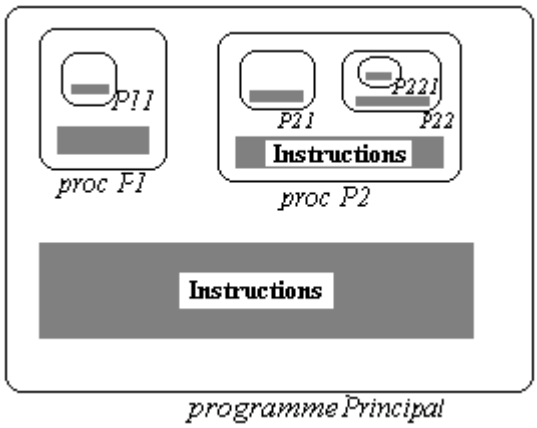
La décomposabilité modulaire :

capacité de décomposer un problème en sous-problèmes, semblable à la méthode structurée descendante.

***Réalisation de ce critère en Pascal* :**

La hiérarchie descendante des procédures et des fonctions.

Illustration de la décomposabilité en Pascal :



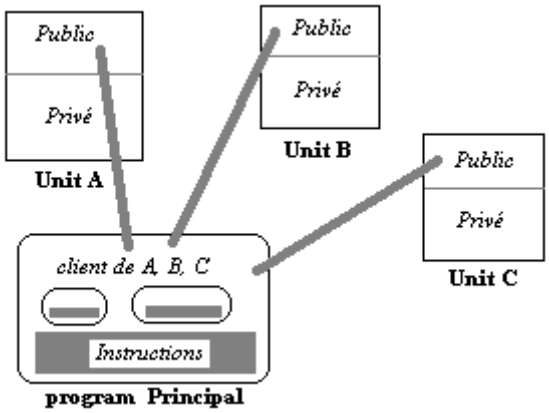
La composition modulaire :

capacité de recombinaison et de réagencement de modules écrits, semblable à la partie ascendante de la programmation structurée.

***Réalisation de ce critère en Pascal* :**

N'existe pas en Pascal standard, toutefois la notion d'**UNIT** en Pascal UCSD (dont le Delphi est un descendant) et de **Library** en sont deux implantations partielles.

Illustration de la composition en Pascal :



La continuité modulaire :

capacité à réduire l'impact de changements dans les spécifications à un minimum de modules liés entre eux, et mieux à un seul module.

***Réalisation de ce critère en Pascal* :**

Partiellement ; le cas particulier des constantes symboliques en Pascal standard, au paragraphe **const,** montre l'intérêt de ce critère**.**

Exemple : **const** n=10 ;
....
for i :=1 **to** n **do** ....
...
if T1 < n **then** ....

Il suffit de changer la ligne const n=10 pour modifier automatiquement les instructions où intervient la constante n, sans avoir à les réécrire toutes. Cette pratique en Pascal est très utile en particulier lorsqu'il s'agit de compenser le défaut dans la continuité modulaire, apporté par la notion de tableau statique dont les bornes doivent être connues à l'avance.

La compréhension modulaire :

capacité à l'interprétation par un programmeur du fonctionnement d'un module ou d'un ensemble de modules liés, sans avoir à connaître tout le logiciel.

Réalisation de ce critère en Pascal :

Partiellement à la charge du programmeur en écrivant des procédures et des fonctions qui s'appellent le moins possible. Chaque procédure ou fonction doit être dédiée à une tâche autonome.

Plus efficace dans Delphi grâce à la notion d'UNIT.

La protection modulaire :

capacité à limiter les effets produits par des incidents lors de l'exécution à un nombre minimal de modules liés entre eux, mieux à un seul module.

Réalisation de ce critère en Pascal :

Correcte en Pascal grâce au contrôle des types et des bornes des paramètres d'entrées ou de sorties d'une procédure ou d'une fonction. Les variables locales permettent de restreindre la portée d'un incident.

Attention *Les pointeurs en Pascal*

Le type pointeur met fortement en défaut ce critère, car sa gestion mémoire est de bas niveau et donc confiée au programmeur ; les pointeurs ne respectent même pas la notion de variable locale!
En général le passage par adresse met en défaut le principe de protection modulaire.

1.3 Préceptes minimaux de construction modulaire

Etant débutants, nous utiliserons quatre des six préceptes énoncés par B.Meyer. Ils sont essentiels et sont adoptés par tous ceux qui pratiquent des méthodes de programmation modulaire :

- **Interface de données minimale**
- **Couplage minimal**
- **Interfaces explicites**
- **Information publique et privée**

Précepte 1 : Interface de données minimale

Un module fixé doit ne communiquer qu'avec un nombre " minimum " d'autres modules du logiciel. L'objectif est de minimiser le ***nombre*** d'interconnexions entre les modules. Le graphe établissant les liaisons entre les modules est noté " graphe de dépendance ". Il doit être le moins maillé possible. La situation est semblable à celle que nous avons rencontré lors de la description des différentes topologies des réseaux d'ordinateurs : les liaisons les plus simples sont les liaisons en étoile, les plus complexes (donc ici déconseillées) sont les liaisons totalement maillées.

L'intérêt de ce précepte est de garantir un meilleur respect des critères de continuité et de protection modulaire. Les effets d'une modification du code source ou d'une erreur durant l'exécution dans un module peuvent se propager à un nombre plus ou moins important de modules en suivant le graphe de liaison. Un débutant optera pour une architecture de liaison simple, ce qui induira une construction contraignante du logiciel. L'optimum est défini par le programmeur avec l'habitude de la programmation.

***Réalisation de ce précept en Pascal* :**

Le graphe de dépendance des procédures et des fonctions sera arborescent ou en étoile.

Précepte 2 : Couplage minimal

Lorsque deux modules communiquent entre eux, l'échange d'information doit être minimal. Ce précepte ne fait pas double emploi avec le précédent. Il s'agit de minimiser la *taille* des interconnexions entre modules et non leur *nombre* comme dans le précepte précédent.

***Réalisation de ce précept en Pascal* :**

En général, nous avons aussi un couplage fort lorsqu'on introduit **toutes** les variables comme globales (donc à éviter, ce qui se produit au stade du débutant). D'autre part la notion de visibilité dans les blocs imbriqués et la portée des variables Pascal donne accès à des données qui ne sont pas toutes utiles au niveau le plus bas.

Précepte 3 : Interfaces explicites

Lorsque deux modules M1 et M2 communiquent, l'échange d'information doit être lisible explicitement dans l'un des deux ou dans les deux modules.

***Réalisation de ce précept en Pascal* :**

L'utilisation des données globales ou de la notion de visibilité nuit aussi à ce principe. Le risque de battre en brèche le précepte des interfaces explicites est alors de conduire à des accès de données injustifiés (problème classique des effets de bord, où l'on utilise implicitement dans un bloc une donnée visible mais non déclarée dans ce bloc).

Précepte 4 : Information publique et privée

Toute information dans un module doit être répartie en deux catégories : l'information privée et l'information publique.

Ce précepte permet de modifier la partie privée sans que les clients (*modules utilisant ce module*) aient à supporter un quelconque problème à cause de modifications ou de

changements. Plus la partie publique est petite, plus on a de chances que des changements n'aient que peu d'effet sur les clients du module.

- La partie publique doit être la description des opérations ou du fonctionnement du module.
- La partie privée contient l'implantation des opérateurs et tout ce qui s'y rattache.

Réalisation de ce précept en Pascal :

Le Pascal standard ne permet absolument pas de respecter ce principe dans le cadre général. Delphi, avec la notion d'UNIT cotient une approche partielle mais utile de ce principe. Les prémisses de cette approche existent malgré tout dans les notions de variables et de procédures locales à une procédure. La notion de classe en Delphi implante complètement ce principe.

Enfin et pour mémoire nous citerons l'existence du précepte d'ouverture-fermeture et du précepte d'unités linguistiques.

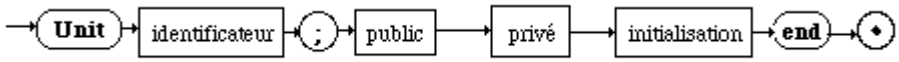
2. La modularité par les Unit avec Delphi

La notion de UNIT a été introduite en Pascal UCSD ancêtre de Delphi

Rappelons que Delphi et les versions de compilateurs libres gratuit comme FreePascal compiler, Obéron etc... présentes sur Internet fonctionnant sur les micro-ordinateurs type PC, ainsi que le Think Pascal de Symantec fonctionnant sur les MacIntosh d'Apple, sont tous une extension du Pascal UCSD. Il est donc possible sur du matériel courant d'utiliser la notion d'UNIT simulant le premier niveau du concept de module.

Cet élément représente une unité compilable séparément de tout programme et stockable en bibliothèque. Une **Unit** comporte une partie " public " et une partie " privé ". Elle implante donc l'idée de module et étend la notion de bloc (procédure ou fonction) en Pascal.

Syntaxe :



Exemple :

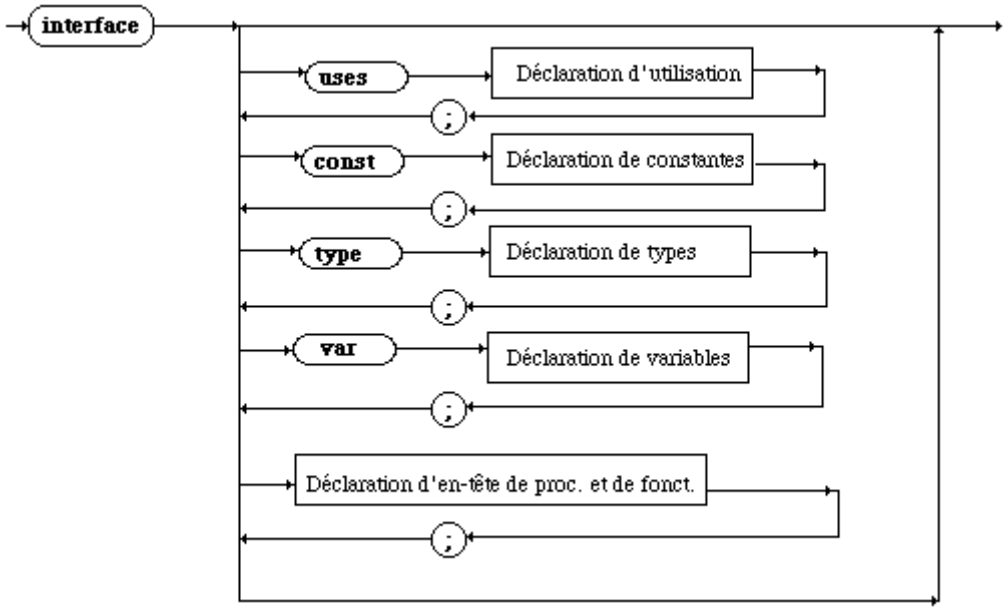
```
Unit Truc;
<partie public>
<partie privée>
<initialisation>
end.
```

2.1 Partie " public " d'une UNIT : " Interface "

Correspond exactement à la partie publique du module représenté par la UNIT. Cette partie décrit les en-têtes des procédures et des fonctions publiques utilisables par les clients. Les clients peuvent être soit d'autres procédures Pascal, des programmes Delphi ou d'autres Unit.

La clause Uses XXX dans un programme Delphi, permet d'indiquer la référence à la Unit XXX et autorise l'accès aux procédures et fonctions publiques de l'interface dans tout le programme.

Syntaxe :



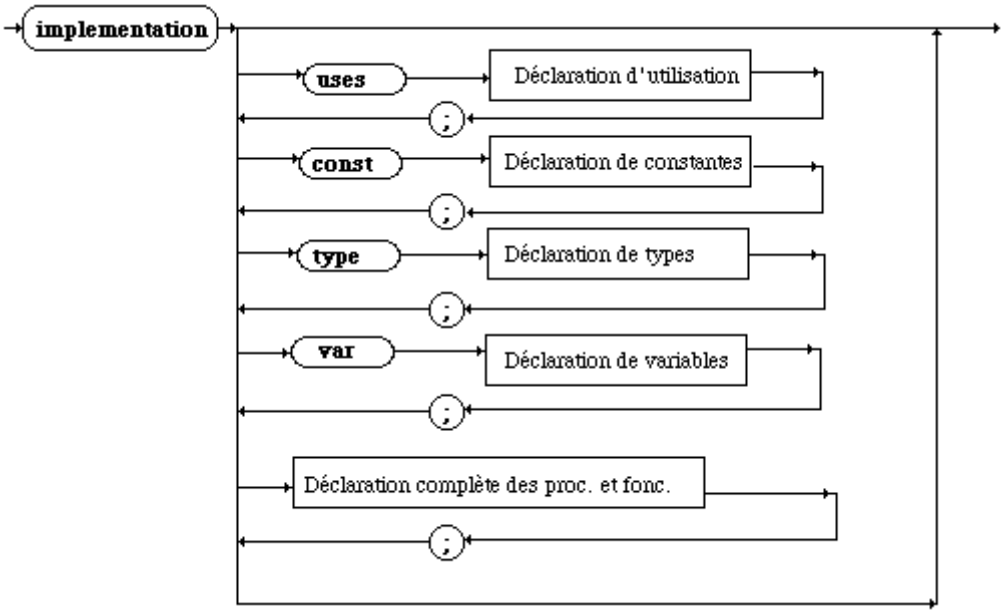
Exemple :

```
Unit Truc ;
interface
Uses Machin, Chose;
const
   a=10;
   x='a';
Type
   amoi=12..36;
var
   x, y : integer;
    z : amoi;
implementation

end.
```

2.2 Partie " privée " d'une UNIT : " Implementation "

Correspond à la partie privée du module représenté par la UNIT. Cette partie intimement liée à l'interface, contient le code interne du module. Elle contient deux sortes d'éléments : les déclarations complètes des procédures et des fonctions privées ainsi que les structures de données privées. Elle contient aussi les déclarations complètes des fonctions et procédures publiques dont les en-têtes sont présentes dans l'interface.

Syntaxe :



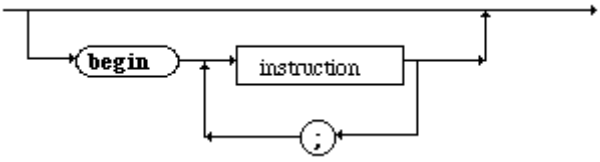
Exemple :

```
Unit Truc ;
interface
Uses Machin, Chose;
const
   a=10;
   x='a';

Type
   amoi = 12..36;

var
  x, y : integer;
  z : amoi;


procedure P1(x:real;var u:integer);
procedure P2(u,v:char;var x,y,t:amoi);
function F(x:real):boolean;
```

```
implementation

procedure P1(x:real;var u:integer);
begin
< corps de procédure>
end;

procedure P2(u,v:char;var x,y,t:amoi);
begin
< corps de procédure>
end;

function F(x:real):boolean;
begin
< corps de fonction>
end;

end.
```

2.3 Partie initialisation d'une UNIT

Il est possible d'initialiser des variables et d'exécuter des instructions au lancement de l'UNIT. Elles correspondent à des instructions classiques Pascal sur des données publiques ou privées de la **Unit** (initialisation de tableaux, mise à zéro de divers indicateurs, chargement de fichiers etc...):



3.3 : Complexité, tri, recherche

Plan du chapitre: 

1. Complexité d'un algorithme

1.1 Notions de complexité temporelle et spatiale
1.2 Mesure de la complexité temporelle d'un algorithme
1.3 Notation de Landau O(n)

2. Trier des tableaux en mémoire centrale

2.1 Tri interne, tri externe
2.2 Des algorithmes classiques de tri interne

- *Le Tri à bulles*
- *Le Tri par sélection*
- *Le ri par insertion*
- *Le Tri rapide QuickSort*
- *Le Tri par tas HeapSort*

3. Rechercher dans un tableau

3.1 Recherche dans un tableau non trié
3.2 Recherche dans un tableau trié

1. Complexité d'un algorithme et performance

Nous faisons la distinction entre les méthodes (algorithmes) de tri d'un grand nombre d'éléments (plusieurs milliers ou plus), et le tri de quelques éléments (quelques dizaines, voir quelques centaines). Pour de très petits nombres d'éléments, la méthode importe peu. Il est intéressant de pouvoir comparer différents algorithmes de tris afin de savoir quand les utiliser. Ce que nous énonçons dans ce paragraphe s'applique en général à tous les algorithmes et en particulier aux algorithmes de tris qui en sont une excellente illustration.

1.1 Notions de complexité temporelle et spatiale

L'efficacité d'un algorithme est directement liée au programme à implémenter sur un ordinateur. Le programme va s'exécuter en un **temps fini** et va mobiliser des **ressources mémoires** pendant son exécution; ces deux paramètres se dénomment **complexité temporelle** et **complexité spatiale**.

Dès le début de l'informatique les deux paramètres "temps d'exécution" et "place mémoire" ont eu une importance à peu près égale pour comparer l'efficacité relative des algorithmes. Il est clair que depuis que l'on peut, à coût très réduit avoir des mémoires centrales d'environ 1 Giga octets dans une machine personnelle, les soucis de place en mémoire centrale qui s'étaient fait jour lorsque l'on travaillait avec des mémoires centrales de 128 Kilo octets (pour des gros matériels de recherche des années 70) sont repoussés psychologiquement plus loin pour un utilisateur normal. Comme c'est le système d'exploitation qui gère la mémoire disponible (RAM, cache, virtuelle etc...), les analyses de performances de gestion de la mémoire peuvent varier pour le même programme.

Le facteur temps d'exécution reste l'élément qualitatif le plus perceptible par l'utilisateur d'un programme ne serait ce que parce qu'il attend derrière son écran le résultat d'un travail qui représente l'exécution d'un algorithme.

L'informatique reste une science de l'ingénieur ce qui signifie ici, que malgré toutes les études ou les critères théoriques permettant de comparer l'efficacité de deux algorithmes dans l'absolu, dans la pratique nous ne pourrons pas dire qu'il y a un **meilleur** algorithme pour résoudre tel type de problème. Une méthode pouvant être lente pour certaines configurations de données et dans une autre application qui travaille systématiquement sur une configuration de données favorables la méthode peut s'avérer être la "meilleure".
La recherche de la performance à tout prix est aussi inefficace que l'attitude contraire.

Prenons à notre compte les recommandations de R.Sedgewick :

> Quel que soit le problème mettez d'abord en œuvre l'algorithme le plus simple, solution du problème, car le temps nécessaire à l'implantation et à la mise au point d'un algorithme "optimisé" peut être bien plus important que le temps requis pour simplement faire fonctionner un programme légèrement moins rapide.

Il nous faut donc un outil permettant de comparer l'efficacité ou complexité d'un algorithme à celle d'un autre algorithme résolvant le même problème.

1.2 Mesure de la complexité temporelle d'un algorithme

Nous prenons le parti de nous intéresser uniquement au temps théorique d'exécution d'un algorithme. Pourquoi théorique et non pratique ? Parce que le temps pratique d'exécution d'un programme, comme nous l'avons signalé plus haut dépend :

- de la machine (par exemple processeur travaillant avec des jeux d'instructions optimisées ou non),
- du système d'exploitation (par exemple dans la gestion multi-tâche des processus),
- du compilateur du langage dans lequel l'algorithme sera traduit (compilateur natif pour un processeur donné ou non),
- des données utilisées par le programme (nature et/ou taille),
- d'un facteur intrinsèque à l'algorithme.

Nous souhaitons donc pouvoir utiliser un instrument mathématique de mesure qui rende compte de l'efficacité spécifique d'un algorithme indépendamment de son implantation en langage évolué sur une machine. Tout en sachant bien que certains algorithmes ne pourront pas être analysés ainsi soit parce que mathématiquement cela est impossible, soit parce que les configurations de données ne sont pas spécifiées d'un manière précise, soit parce que le temps mis à analyser correctement l'algorithme dépasserait le temps de loisir et de travail disponible du développeur !

Notre instrument, la complexité temporelle, est fondé sur des outils abstraits (qui ont leur correspondance concrète dans un langage de programmation). L'outil le plus connu est l'opération élémentaire (quantité abstraite définie intuitivement ou d'une manière évidente par le développeur).

Notion d'opération élémentaire

Une opération élémentaire est une opération fondamentale d'un algorithme si le temps d'exécution est directement lié (par une formule mathématique ou empirique) au nombre de ces opérations élémentaires. Il peut y avoir plusieurs opérations élémentaires dans un même algorithme.

Nous pourrons ainsi comparer deux algorithmes résolvant le même problème en comparant ce nombre d'opérations élémentaires effectuées par chacun des deux algorithmes.

1.2.1 La complexité temporelle : notation

C'est le décompte du nombre d'opérations élémentaires effectuées par un algorithme donné.

Il n'existe pas de méthodologie systématique (art de l'ingénieur) permettant pour un algorithme quelconque de compter les opérations élémentaires. Toutefois des règles usuelles sont communément admises par la communauté des informaticiens qui les utilisent pour évaluer la complexité temporelle.

Soient $\mathbf{i_1}$, $\mathbf{i_2}$, ... , $\mathbf{i_k}$ des instructions algorithmiques (affectation, itération, condition,...)
Soit une opération élémentaire dénotée **OpElem**, supposons qu'elle apparaisse n_1 fois dans l'instruction $\mathbf{i_1}$, n_2 fois dans l'instruction $\mathbf{i_2}$, ... n_k fois dans l'instruction $\mathbf{i_k}$. Nous noterons $Nb(i_1)$ le nombre n_1, $Nb(i_2)$ le nombre n_2 etc.

Nous définissons ainsi la fonction Nb (i_k) indiquant le nombre d'opérations élémentaires dénoté **OpElem** contenu dans l'instruction algorithmique i_k :

Nb( ) : Instruction → Entier .

1.2.2 Complexité temporelle d'une séquence d'instructions

Soit **S** la séquence d'exécution des instructions $\mathbf{i_1}$; $\mathbf{i_2}$; ... ; $\mathbf{i_k}$, soit n_k =Nb (i_k) le nombre d'opérations élémentaires de l'instruction $\mathbf{i_k}$.

Le nombre d'opérations élémentaires **OpElem** de S, Nb(**S**) est égal par définition à la somme des nombres: $n_1 + n_2 + ... + n_k$:

S =
début
$\mathbf{i_1}$;
$\mathbf{i_2}$;
... ;
$\mathbf{i_k}$
fin

$$Nb(\mathbf{S}) = \sum Nb(i_p) = n_1 + n_2 + ... + n_k$$

1.2.3 Complexité temporelle d'une instruction conditionnelle

Dans les instructions conditionnelles étant donné qu'il n'est pas possible d'une manière générale de déterminer systématiquement quelle partie de l'instruction est exécutée (le **alors** ou le **sinon**), on prend donc un majorant :

Cond =
si Expr **alors** E1
sinon E2
fsi

$$Nb(\mathbf{Cond}) < Nb(Expr) + \max(Nb(E1), Nb(E2))$$

1.2.4 Complexité temporelle d'une itération finie bornée

Dans le cas d'une boucle finie bornée (comme pour...fpour) contrôlée par une variable d'indice "i", l'on connaît le nombre exact d'itérations noté ***Nbr_d'itérations*** de l'ensemble des instructions composant le corps de la boucle dénotées S (où S est une séquence d'instructions), l'arrêt étant assuré par la condition de sortie Expr(i) dépendant de la variable d'indice de boucle i.

La complexité est égale au produit du nombre d'itérations par la somme de la complexité de la séquence d'instructions du corps et de celle de l'évaluation de la condition d'arrêt Expr(i).

Iter =
Itération Expr(i)
S
finItér

Nb(**Iter**) = [Nb(S) + Nb(Expr(i))] x ***Nbr_d'itérations***

*Exemple dans le cas d'une boucle **pour**...**fpour** :*

Iter =
pour i<-- a **jusquà** b **faire**
i_1 ;
i_2 ;
... ;
i_k
fpour

La complexité de la condition d'arrêt est par définition de **1** (<= le temps d'exécution de l'opération effectuée en l'occurence un test, ne dépend ni de la taille des données ni de leurs valeurs), en **notant |b-a| le nombre exact d'itérations exécutées** (lorsque les bornes sont des entiers **|b-a|** vaut exactement la valeur absolue de la différence des bornes) nous avons :
Nb(**Iter**) = ($\sum$ Nb(**i_p**) + 1) . |b-a|

Lorsque le nombre d'itérations n'est pas connu mais seulement majoré (nombre noté ***Majorant_Nbr_d'itérations***), alors on obtient un majorant de la complexité de la boucle (le majorant correspond à la complexité dans le pire des cas).

Complexité temporelle au pire :

Majorant_Nb(**Iter**) = [Nb(S) + Nb(Expr(i))] x ***Majorant_Nbr_d'itérations***

1.3 Notation de Landau O(n)

Nous avons admis l'hypothèse qu'en règle générale **la complexité en temps** dépend de la taille **n** des données (plus le nombre de données est grand plus le temps d'exécution est long).

Cette remarque est d'autant plus proche de la réalité que nous étudierons essentiellement des algorithmes de tri dans lesquels les **n** données sont représentées par une liste à **n** éléments.

Afin que notre instrument de mesure et de comparaison d'algorithmes ne dépende pas de la machine physique, nous n'exprimons pas le temps d'exécution en unités de temps (millisecondes etc..) mais en unité de taille des données.

Nous ne souhaitons pas ici rentrer dans le détail mathématique des notations O(f(**n**)) de Landau sur les infiniment grands équivalents, nous donnons seulement une utilisation pratique de cette notation.

Pour une fonction f (n) dépendant de la variable n, on écrit :
f est O(g(n)) g(n) où g est elle-même une fonction de la variable entière n, et l'on lit **f est de l'ordre de grandeur de g(n)** ou plus succinctement **f est d'ordre g(n)**, lorsque :

f est d'ordre g(n) :

Pour tout valeur entière de **n**, il existe deux constantes a et b positives telles que : $a.g(\mathbf{n}) < f(\mathbf{n}) < b.g(\mathbf{n})$

Ce qui signifie que lorsque **n** tend vers l'infini (**n** devient très grand en informatique) le rapport f(**n**)/g(**n**) reste borné.

f est d'ordre g(n) :

$a < f(\mathbf{n})/g(\mathbf{n}) < b$ quand $\mathbf{n} \rightarrow \infty$
Lorsque **n** tend vers l'infini, le rapport f(**n**)/g(**n**) reste borné.

Exemple :

Supposons que f et g soient les polynômes suivants :
$f(n) = 3\mathbf{n}^2 - 7\mathbf{n} + 4$
$g(n) = \mathbf{n}^2$;
Lorsque **n** tend vers l'infini le rapport f(**n**)/g(**n**) tend vers 3:
$f(\mathbf{n})/g(\mathbf{n}) \rightarrow 3$ quand $\mathbf{n} \rightarrow \infty$
ce rapport est donc borné.

donc **f est d'ordre $\mathbf{n}^2$** ou encore **f est O($\mathbf{n}^2$)**

C'est cette notation que nous utiliserons pour mesurer la complexité temporelle C en nombre d'opérations élémentaires d'un algorithme fixé. Il suffit pour pouvoir comparer des complexités temporelles différentes, d'utiliser les mêmes fonctions g(n) de base.

Les informaticiens ont répertorié des situations courantes et ont calculé l'ordre de complexité associé à ce genre de situation.

Les fonctions g(n) classiquement et pratiquement les plus utilisées sont les suivantes :

$g(n) = 1$
$g(n) = \log_k (n)$
$g(n) = n$
$g(n) = n.\log_k(n)$
$g(n) = n^2$

Ordre de complexité C	Cas d'utilisation courant
$g(n) = 1 \Rightarrow$ **C est O(1)**	Algorithme ne dépendant pas des données
$g(n) = \log_k (n) \Rightarrow$ **C est O($\log_k$(n))**	Algorithme divisant le problème par une quantité constante (base k du logarithme)
$g(n) = n \Rightarrow$ **C est O(n)**	Algorithme travaillant directement sur chacune des n données
$g(n) = n.\log_k(n) \Rightarrow$ **C est O(n.$\log_k$(n))**	Algorithme divisant le problème en nombre de sous-problèmes constants (base k du logarithme), dont les résultats sont réutilisés par recombinaison
$g(n) = n^2 \Rightarrow$ **C est O(n^2)**	Algorithme traitant généralement des couples de données (dans deux boucles imbriquées).

2. Trier des tableaux en mémoire centrale

Un tri est une opération de classement d'éléments d'**une liste selon un ordre total** défini. Sur le plan pratique, on considère généralement deux domaines d'application des tris: les tris internes et les tris externes.

Que se passe-t-il dans un tri? On suppose qu'on se donne une suite de nombres entiers (ex: 5, 8, -3 ,6 ,42, 2,101, -8, 42, 6) et l'on souhaite les classer par ordre croissant (relation d'ordre au sens large). La suite précédente devient alors après le tri (classement) : (-8, -3, 2, 5, 6, 6, 8, 42, 42, 101). Il s'agit en fait d'une nouvelle suite obtenue par une permutation des éléments de la première liste de telle façon que les éléments résultants soient classés par ordre croissant au sens large selon la relation d'ordre totale " $\leq$ " : ($-8 \leq -3 \leq 2 \leq 5 \leq 6 \leq 6 \leq 8 \leq 42 \leq 42 \leq 101$).

Cette opération se retrouve très souvent en informatique dans beaucoup de structures de données. Par exemple, il faut établir le classement de certains élèves, mettre en ordre un dictionnaire, trier l'index d'un livre, etc...

2.1 Tri interne, tri externe

Un tri interne s'effectue sur des données stockées dans une table en mémoire centrale, **un tri externe** est relatif à une structure de données non contenue entièrement dans la mémoire centrale (comme un fichier sur disque par exemple).

Dans certains cas les données peuvent être stockées sur disque (mémoire secondaire) mais structurées de telle façon que chacune d'entre elles soit représentée en mémoire centrale par **une clef associée à un pointeur**. Le pointeur lié à la clef permet alors d'atteindre l'élément sur le disque (n° d'enregistrement...). Dans ce cas seules les clefs sont triées (en table ou en arbre) en mémoire centrale et il s'agit là d'un tri interne. Nous réservons le vocable tri externe uniquement aux manipulations de tris directement sur les données stockées en mémoire secondaire.

2.2 Des algorithmes classiques de tri interne

Dans les algorithmes référencés ci-dessous, nous notons ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) la liste à trier. Etant donné le mode d'accès en mémoire centrale (accès direct aux données) une telle liste est généralement implantée selon un tableau à une dimension de **n** éléments (cas le plus courant). Nous attachons dans les algorithmes présentés, à expliciter des tris majoritairement sur des tables, certains algorithmes utiliserons des structures d'arbres en mémoire centrale pour représenter notre liste à trier ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$).

Les opérations élémentaires principales les plus courantes permettant les calculs de complexité sur les tris, sont les suivantes :

Deux opérations élémentaires

La comparaison de deux éléments de la liste $\mathbf{a}_i$ et $\mathbf{a}_k$, (**si** $\mathbf{a}_i > \mathbf{a}_k$, **si** ai < ak,....) .
L'échange des places de deux éléments de la liste $\mathbf{a}_i$ et $\mathbf{a}_k$, (place ($\mathbf{a}_i$) $\leftrightarrow$ place ($\mathbf{a}_k$)).

Ces deux opérations seront utilisées afin de fournir une mesure de comparaison des tris entre eux. Nous proposons dans les pages suivantes cinq tris classiques, quatre concerne le tri de données dans un tableau, le cinquième est un tri de données situées dans un arbre binaire, ce dernier pourra en première lecture être ignoré, si le lecteur n'est pas familiarisé avec la notion d'arbre binaire

Tris sur des tables :

- Tri itératif à bulles
- Tri itératif par sélection
- Tri itératif par insertion
- Tri récursif rapide QuickSort

Tris sur un arbre binaire :

- Le Tri par tas / HeapSort

Le tri à bulle

C'est le moins performant de la catégorie des tris par échange ou sélection, mais comme c'est un algorithme simple, il est intéressant à utiliser pédagogiquement.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri à bulle est de parcourir la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en intervertissant toute paire d'éléments consécutifs ($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) non ordonnés.

Ainsi après le premier parcours, l'élément maximum se retrouve en $\mathbf{a}_n$. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).
On recommence l'opération avec la nouvelle sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_{n-1}$), et ainsi de suite jusqu'à épuisement de toutes les sous-suites (la dernière est un couple).

Le nom de tri à bulle vient donc de ce qu'à la fin de chaque itération interne, les plus grands nombres de chaque sous-suite se déplacent vers la droite successivement comme des bulles de la gauche vers la droite.

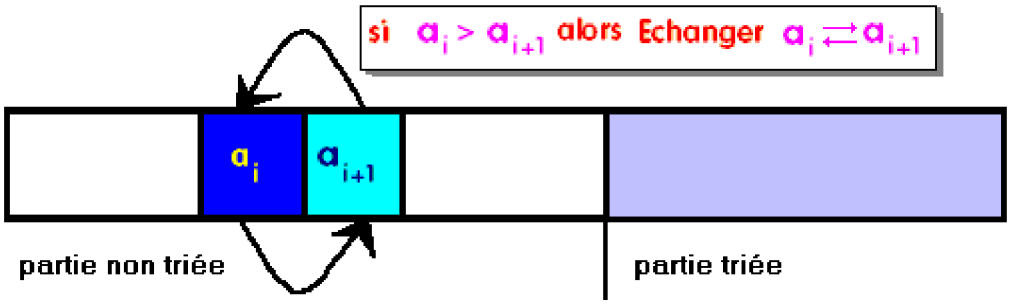
B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] en mémoire centrale.

Le tableau contient une partie triée (en foncé à droite) et une partie non triée (en blanc à gauche).

On effectue plusieurs fois le parcours du tableau à trier.

Le principe de base est de ré-ordonner les couples ($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) non classés (en inversion de rang soit $\mathbf{a}_{i-1} > \mathbf{a}_i$) dans la partie non triée du tableau, puis à déplacer la frontière (le maximum de la sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_{n-1}$)) d'une position :



Tant que la partie non triée n'est pas vide, on permute les couples **non ordonnés** (($\mathbf{a}_{i-1}$, $\mathbf{a}_i$) tels que $\mathbf{a}_{i-1} > \mathbf{a}_i$)) pour obtenir le maximum de celle-ci à l'élément frontière. C'est à dire qu'au premier passage c'est l'extremum global qui est bien classé, au second passage le second extremum etc...

C) Algorithme :

```
Algorithme Tri_a_Bulles
 local:   i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
début
 pour i de n jusquà 1 faire // recommence une sous-suite (a1, a2, ... , ai)
  pour j de 2 jusquà i faire // échange des couples non classés de la sous-suite
   si Tab[ j-1 ] > Tab[ j ] alors // aj-1et aj non ordonnés
     temp ← Tab[ j-1 ] ;
     Tab[ j-1 ] ← Tab[ j ] ;
     Tab[ j ]  ← temp //on échange les positions de aj-1 et aj
   Fsi
  fpour
 fpour
Fin Tri_a_Bulles
```

Exemple : soit la liste (**5 , 4 , 2 , 3 , 7 , 1**), appliquons le tri à bulles sur cette liste d'entiers. Visualisons les différents états de la liste pour chaque itération externe contôlée par l'indice i :

i = 6 / pour j **de** 2 **jusquà** 6 **faire**

5	4	2	3	7	1	*5 > 4 donc permutation des deux cellules*
4	5	2	3	7	1	*5 > 2 donc permutation des deux cellules*
4	2	5	3	7	1	*5 > 3 donc permutation des deux cellules*
4	2	3	5	7	1	*5 < 7 donc aucune action sur ces deux cellules*
4	2	3	5	7	1	*7 > 1 donc permutation des deux cellules*
4	2	3	5	1	7	*A la fin de la boucle externe le max 7 est rangé*

i = 5 / pour j **de** 2 **jusquà** 5 **faire**

4	2	3	5	1	7	*4 > 2 donc permutation des deux cellules*
2	4	3	5	1	7	*4 > 3 donc permutation des deux cellules*
2	3	4	5	1	7	*4 < 5 donc aucune action sur ces deux cellules*
2	3	4	5	1	7	*5 > 1 donc permutation des deux cellules*
2	3	4	1	5	7	*A la fin de la boucle externe le max 5 est rangé*

i = 4 / pour j **de** 2 **jusquà** 4 **faire**

2	3	4	1	5	7	*2 < 3 donc aucune action sur ces deux cellules*
2	3	4	1	5	7	*3 < 4 donc aucune action sur ces deux cellules*
2	3	4	1	5	7	*4 > 1 donc permutation des deux cellules*
2	3	1	4	5	7	*A la fin de la boucle externe le max 4 est rangé*

i = 3 / pour j **de** 2 **jusquà** 3 **faire**

2	3	1	4	5	7	*2 < 3 donc aucune action sur ces deux cellules*
2	3	1	4	5	7	*3 > 1 donc permutation des deux cellules*
2	1	3	4	5	7	*A la fin de la boucle externe le max 3 est rangé*

i = 2 / pour j **de** 2 **jusquà** 2 **faire**

2	1	3	4	5	7	*2 > 1 donc permutation des deux cellules*
1	2	3	4	5	7	*A la fin de la boucle externe le max 2 est rangé*

i = 1 / pour j **de** 2 **jusquà** 1 **faire** ***(boucle vide)***

1	2	3	4	5	7	*Il ne reste plus d'éléments à comparer !*

D) Complexité :

Choix opération

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Le nombre de comparaisons "**si** Tab[j-1] > Tab[j] **alors**" est une valeur qui ne dépend que de la longueur **n** de la liste (**n** est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "**pour** i **de** **n** **jusquà** 1 **faire**" s'exécute n fois (donc une somme de n termes) et qu'à chaque fois la boucle "**pour** j **de** 2 **jusquà** i **faire**" exécute (i-2)+1 fois la comparaison "**si** Tab[j-1] > Tab[j] **alors**".

La complexité en nombre de comparaison est égale à la somme des n termes suivants (i = n, i = n-1,.... i = 1)

C = (n-2)+1 + ([n-1]-2)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité du tri à bulle en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons comme opération élémentaire **l'échange de deux cellules** du tableau.

Calculons par dénombrement le nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse et donc chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.

La complexité du tri à bulle au pire en nombre d'échanges est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

E) Programme Delphi (tableau d'entiers):

```
program TriParBulle;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [1..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

procedure TriBulle (var Tab:TTab) ;
{ Implantation Pascal de l'algorithme }
var i, j, t : integer;
begin
  for i := N downto 1 do
    for j := 2 to i do
      if Tab[j-1] > Tab[j] then
      begin
        t := Tab[j-1];
        Tab[j-1] := Tab[j];
        Tab[j] := t;
     end;
end;


procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
```

```
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
       writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI PAR BULLE');
  writeln;
  Impression(Tab);
  TriBulle(Tab);
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR BULLE

------------------------------------------------------------
 32 |  60 |  60 |  54 |  70 |  53 |  64 |  91 |  69 |  81 |
------------------------------------------------------------
 32 |  53 |  54 |  60 |  60 |  64 |  69 |  70 |  81 |  91 |
------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriBulle  {

 static int[] table = new int[10] ; // le tableau à trier en attribut
```

```
 static void TriBulle ( )  {
  int n = table.length-1;
  for ( int i = n; i>=1; i--)
   for ( int j = 2; j<=i; j++)
    if (table[j-1] > table[j])
    {
     int temp = table[j-1];
     table[j-1] = table[j];
     table[j] = temp;
    }
 }
```

```
 static void Impression ( ) {
  // Affichage du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void Initialisation  ( )  {
  // remplissage aléatoire du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
 }

 public static void main(String[ ] args)  {
  Initialisation ( );
  System.out.println("Tableau initial :");
  Impression ( );
  TriBulle ( );
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
}
```

Le tri par sélection

C'est une version de base de la catégorie des tris par sélection.

A) Spécification abstraite

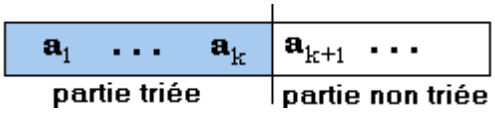
Nous supposons que les données $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ sont mises sous forme d'une liste $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$, la liste $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$ est décomposée en deux parties : une partie liste $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k)$ et une partie non-triée $(\mathbf{a}_{k+1}, \mathbf{a}_{k+2}, \ldots, \mathbf{a}_n)$; l'élément $\mathbf{a}_{k+1}$ est appelé élément frontière (c'est le premier élément non trié).

Le principe est de parcourir la partie non-triée de la liste (**a**k+1, **a**k+2, ... , **a**n) en cherchant l'élément minimum, puis en l'échangeant avec l'élément frontière **a**k+1, puis à déplacer la frontière d'une position. Il s'agit d'une récurrence sur les minima successifs. On suppose que l'ordre s'écrit de gauche à droite (à gauche le plus petit élément, à droite le plus grand élément).
On recommence l'opération avec la nouvelle sous-suite (**a**k+2, ... , **a**n) , et ainsi de suite jusqu'à ce que la dernière soit vide.

B) Spécification concrète

La suite $(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n)$ est rangée dans un tableau à une dimensionT[...] en mémoire centrale.

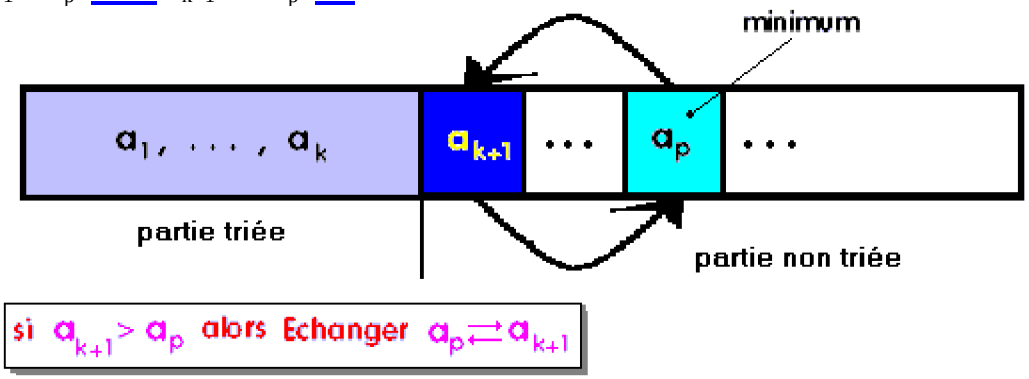
Le tableau contient une partie triée (en foncé à gauche) et une partie non triée (en blanc à droite).



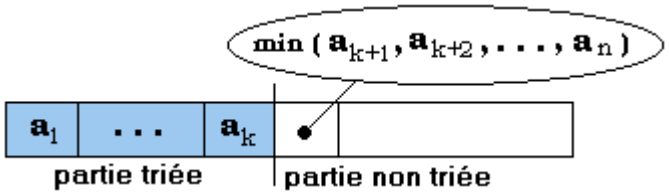
On cherche le minimum de la partie non-triée du tableau et on le recopie dans la cellule frontière (le premier élément de la partie non triée).

Donc pour tout $\mathbf{a}_p$ de la partie non triée on effectue l'action suivante :

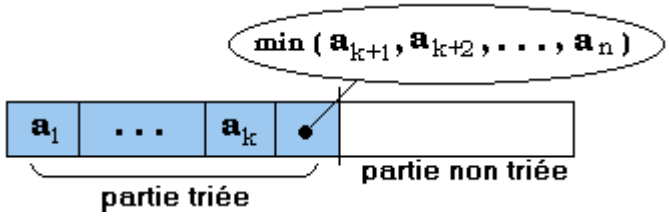
si $\mathbf{a}_{k+1} > \mathbf{a}_p$ **alors** $\mathbf{a}_{k+1} \leftarrow \mathbf{a}_p$ **Fsi**



et l'on obtient ainsi à la fin de l'examen de la sous-liste ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$) la valeur min ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$) stockée dans la cellule $\mathbf{a}_{k+1}$.



La sous-suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$, $\mathbf{a}_{k+1}$) est maintenant triée :



Et l'on recommence la boucle de recherche du minimum sur la nouvelle sous-liste ($\mathbf{a}_{k+2}$, $\mathbf{a}_{k+3}$, ... , $\mathbf{a}_n$) etc...

Tant que la partie non triée n'est pas vide, on range le minimum de la partie non-triée dans l'élément frontière.

C) Algorithme :

Une version maladroite de l'algorithme mais exacte a été fournie par un groupe d'étudiants elle est dénommée / **Version maladroite 1**/.

Elle échange physiquement et systématiquement l'élément frontière Tab[i] avec un élément Tab[j] dont la valeur est plus petite (la suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_i$) est triée) :

Maladroit

```
Algorithme Tri_Selection /Version maladroite 1/
 local:   m, i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
          début
           pour i de 1 jusquà n-1faire // recommence une sous-suite
            m ← i ; // i est l'indice de l'élément frontière Tab[ i ]
            pour j de i+1 jusquà n faire// liste non-triée : (a_{i+1} , a_{i+2} , ... , a_n )
              si Tab[ j ] <  Tab[ m ] alors // a_j est le nouveau minimum partiel
                m ← j ;
                temp ←Tab[ m ] ;
                Tab[ m ] ← Tab[ i ] ;
                Tab[ i ]  ← temp //on échange les positions de a_i et de a_j
                m ← i ;
              Fsi
            fpour
           fpour
          Fin Tri_Selection
```

Voici une version correcte et améliorée du précédent (nous allons voir avec la notion de complexité comment appuyer cette intuition d'amélioration), dans laquelle l'on sort l'échange a_i *et* a_j de la boucle interne "**pour** j **de** i+1 **jusquà n faire**" pour le déporter à la fin de cette boucle.

Amélioration

Au lieu de travailler sur les contenus des cellules de la table, nous travaillons sur les indices, ainsi lorsque *aj* est plus petit que *ai* nous mémorisons l'indice "j" du minimum dans une variable " m ← j ; " plutôt que le minimum lui-même.

Version maladroite	Version améliorée

pour j **de** i+1 **jusquà n faire**
  si Tab[j] < Tab[m] **alors**
  m ← j ;
  temp ←Tab[m] ;
  Tab[m] ← Tab[i] ;
  Tab[i] ← temp
  m ← i ;
  Fsi
fpour
	pour j **de** i+1 **jusquà n faire**
	  si Tab[j] < Tab[m] **alors**
	  m ← j ;
	  Fsi
	fpour;
	temp ← Tab[m] ;
	Tab[m] ←Tab[i] ;
	Tab[i] ← temp

A la fin de la boucle interne "**pour** j **de** i+1 **jusquà n faire**" la variable m contient l'indice de min(**a**i+1, **a**k+2, ... , **a**n) et l'on permute l'élément concerné (d'indice m) avec l'élément frontière ***ai*** :

```
Algorithme Tri_Selection  /Version 2 améliorée/
 local:   m, i , j , n, temp ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 1 à n éléments
          début
           pour i de 1 jusquà n-1 faire // recommence une sous-suite
            m ← i ; // i est l'indice de l'élément frontière ai = Tab[ i ]
            pour j de i+1 jusquà n faire// (a_{i+1} , a_{i+2} , ... , a_n )
             si Tab[ j ] <  Tab[ m ] alors // aj est le nouveau minimum partiel
               m ← j ; // indice mémorisé
             Fsi
            fpour;
            temp ← Tab[ m ] ;
            Tab[ m ] ←Tab[ i ] ;
            Tab[ i ]  ← temp //on échange les positions de a_i et de a_j
           fpour
          Fin Tri_Selection
```

D) Complexité :

Choix opération

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Pour les deux versions 1 et 2 :
Le nombre de comparaisons "**si** Tab[j] < Tab[m] **alors**" est une valeur qui ne dépend que de la longueur **n** de la liste (**n** est le nombre d'éléments du tableau), ce nombre est égal au nombre de fois que les itérations s'exécutent, le comptage montre que la boucle "**pour** i **de** 1 **jusquà n-1 faire**" s'exécute n-1 fois (donc une somme de n-1 termes) et qu'à chaque fois la boucle "**pour** j **de** i+1 **jusquà n faire**" exécute (n-(i+1)+1 fois la comparaison "**si** Tab[j] < Tab[m] **alors**".

La complexité en nombre de comparaison est égale à la somme des n-1 termes suivants (i = 1, ...i = n-1)
C = (n-2)+1 + (n-3)+1 +.....+1+0 = (n-1)+(n-2)+...+1 = n.(n-1)/2 (c'est la somme des n-1 premiers entiers).

La complexité du tri par sélection en nombre de comparaison est de de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons comme opération élémentaire **l'échange de deux cellules** du tableau.

Calculons par dénombrement le nombre d'échanges dans le pire des cas (complexité au pire = majorant du nombre d'échanges). Le cas le plus mauvais est celui où le tableau est déjà classé mais dans l'ordre inverse.

Pour la version 1

Au pire chaque cellule doit être échangée, dans cette éventualité il y a donc autant d'échanges que de tests.

La complexité au pire en nombre d'échanges de la version 1 est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Pour la version 2

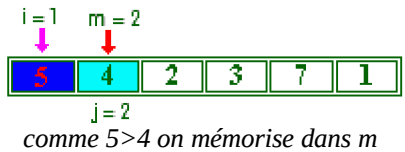
L'échange a lieu systématiquement dans la boucle principale "**pour** i **de** 1 **jusquà** n-1 **faire**" qui s'exécute n-1 fois :

La complexité en nombre d'échanges de cellules de la version 2 est de l'ordre de n, que l'on écrit O(n).

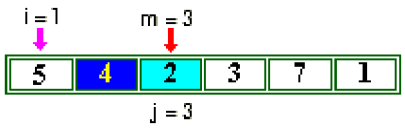
Un échange valant 3 transferts (affectation) la complexité en transfert est O(3n) = O(n)

Toutefois cette complexité en nombre d'échanges de cellules n'apparaît pas comme significative du tri, outre le nombre de comparaison, c'est le nombre d'affectations d'indice qui représente une opération fondamentale et là les deux versions ont exactement la même complexité **$O(n^2)$**.

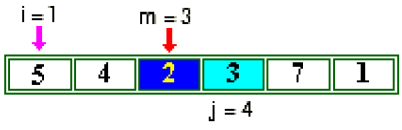
Exemple : soit la liste à 6 éléments (**5 , 4 , 2 , 3 , 7 ,1**), appliquons la version 2 du tri par sélection sur cette liste d'entiers. Visualisons les différents états de la liste pour la première itération externe contrôlée par i (i = 1) et pour les itérations internes contrôlées par l'indice j (de j = 2 ... à ... j = 6) :



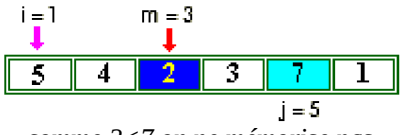
comme 5>4 on mémorise dans m



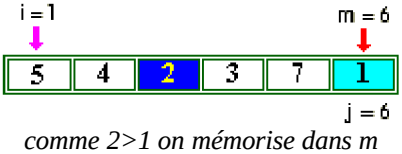
comme 4>2 on mémorise dans m



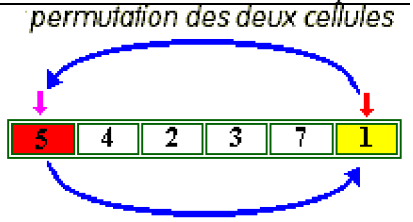
comme 2<3 on ne mémorise pas



comme 2<7 on ne mémorise pas

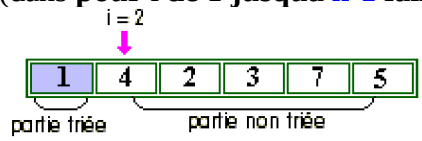


comme 2>1 on mémorise dans m



on échange T[1] et T[6]

L'algorithme ayant terminé l'échange de T[1] et de T[6], il passe à l'itération externe suivante (dans **pour** i **de** 1 **jusqu'à** **n-1** **faire** , il passe à i = 2) :



etc....

E) Programme Delphi (tableau d'entiers) :

```
program TriParSelection;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [1..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

 procedure TriSelection (var Tab:TTab) ;
{ Implantation Pascal de l'algorithme }
var i, j, t, m : integer;
begin
  for i := 1 to N-1 do
  begin
    m := i;
    for j := i+1 to N do
      if Tab[ j ] < Tab[ m ] then m := j;
    t := Tab[m];
    Tab[m] := Tab[i];
    Tab[i] := t;
  end;
end;

procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
       writeln;
end;

begin
  Initialisation(Tab);
```

```
  writeln('TRI PAR SELECTION');
  writeln;
  Impression(Tab);
  TriSelection(Tab);
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR SELECTION

-------------------------------------------------------------
 28 |  51 |  86 |  43 |  32 |   6 |  52 |  51 |  79 |  42 |
-------------------------------------------------------------
  6 |  28 |  32 |  42 |  43 |  51 |  51 |  52 |  79 |  86 |
-------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriSelect
{
 static int[] table = new int[20] ; // le tableau à trier en attribut

 static void Impression ( )  {
  // Affichage du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void Initialisation  ( )  {
  // remplissage aléatoire du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
 }
```

```
 static void TriSelect ( ) {
  int n = table.length-1;
  for ( int i = 1; i <= n-1; i++)
  { // recommence une sous-suite
   int m = i; // élément frontière ai = table[ i ]
   for ( int j = i+1; j <= n; j++)   // (ai+1, a2, ... , an)
    if (table[ j ] < table[ m ]) // aj = nouveau minimum partiel
     m = j ; // indice mémorisé
   int temp = table[ m ];
   table[ m ] = table[ i ];
   table[ i ]= temp;
  }
 }
```

```
 public static void main(String[ ] args)
 {
  Initialisation ( );
  System.out.println("Tableau initial :");
  Impression ( );
  TriSelect ( );
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
}
```

Le tri par insertion

C'est un tri en général un peu plus coûteux en particulier en nombre de transfert à effectuer qu'un tri par sélection (cf. complexité**).**

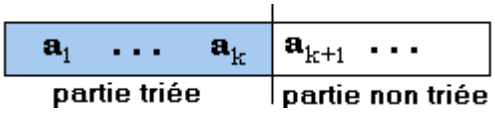
A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par insertion est de parcourir la liste non triée liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en la décomposant en deux parties : une partie déjà triée et une partie non triée.

La méthode est identique à celle que l'on utilise pour ranger des cartes que l'on tient dans sa main : on insère dans le paquet de cartes déjà rangées une nouvelle carte au bon endroit.

L'opération de base consiste à prendre l'élément frontière dans la partie non triée, puis à l'insérer à sa place dans la partie triée (place que l'on recherchera séquentiellement), puis à déplacer la frontière d'une position vers la droite. Ces insertions s'effectuent tant qu'il reste un élément à ranger dans la partie non triée.. L'insertion de l'élément frontière est effectuée par décalages successifs d'une cellule.

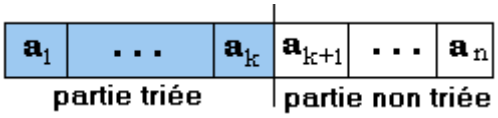
La liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est décomposée en deux parties : une partie triée ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_k$) et une partie non-triée ($\mathbf{a}_{k+1}$, $\mathbf{a}_{k+2}$, ... , $\mathbf{a}_n$); l'élément $\mathbf{a}_{k+1}$ est appelé élément frontière (c'est le premier élément non trié).



B) Spécification concrète itérative

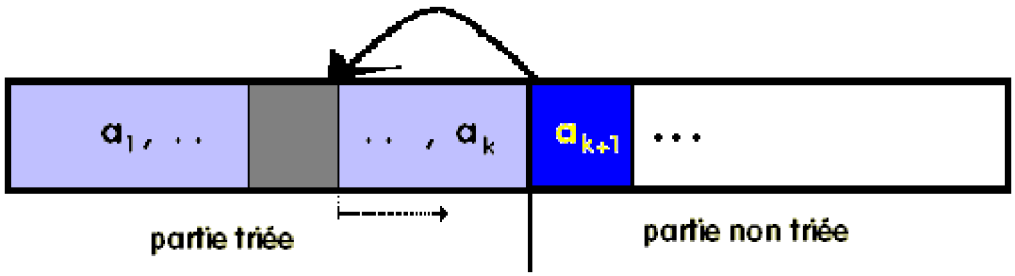
La suite (a_1, a_2, ... , a_n) est rangée dans un tableau à une dimension T[...] en mémoire centrale.

Le tableau contient une partie triée ((a_1, a_2, ... , a_k) en foncé à gauche) et une partie non triée ((a_{k+1}, a_{k+2}, ... , a_n); en blanc à droite) :



On insère l'élément frontière a_{k+1} en faisant varier j de k jusqu'à 2 , afin de balayer toute la partie (a1, a2, ... , ak) déjà rangée, on décale alors d'une place les éléments plus grands que l'élément frontière :

tantque $a_{j-1} > a_{k+1}$ **faire**
décaler a_{j-1} en a_j ;
passer au j précédent
ftant



La boucle s'arrête lorsque $a_{j-1} < a_{k+1}$,ce qui veut dire que l'on vient de trouver au rang j-1 un élément a_{j-1} plus petit que l'élément frontière a_{k+1}, donc a_{k+1} doit être placé au rang j.

C) Algorithme :

```
Algorithme Tri_Insertion
 local:   i , j , n, v ∈  Entiers naturels
 Entrée : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments
 Sortie : Tab ∈  Tableau d'Entiers naturels de 0 à n éléments
{ dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la
boucle tantque .. faire si l'indice j n'est pas inférieur à 1, elle aura une valeur
inférieure à toute valeur possible de la liste
}
début
 pour i de2 jusqà n faire// la partie non encore triée (ai , ai+1 , ... , an )
    v ←Tab[ i ] ;  // l'élément frontière : ai
    j ← i ;        // le rang de l'élément frontière
     Tantque Tab[ j-1 ] > v faire // on travaille sur la partie déjà triée (a1 , a2 , ... , ai)
           Tab[ j ]  ← Tab[ j-1 ]; // on décale l'élément
            j ← j-1;              // on passe au rang précédent
```

```
      FinTant ;
      Tab[ j ]  ← v //on recopie ai dans la place libérée
   fpour
Fin Tri_Insertion
```

Sans la sentinelle en T[0] nous aurions une comparaison sur **j** à l'intérieur de la boucle :

```
Tantque Tab[ j-1 ] > v faire//on travaille sur la partie déjà triée(a1 , a2 , ... , ai)
          Tab[ j ]  ← Tab[ j-1 ]; // on décale l'élément
           j ← j-1;              // on passe au rang précédent
        si j = 0 alors Sortir de la boucle fsi
FinTant ;
```

Exercice

Un étudiant a proposé d'intégrer la comparaison dans le test de la boucle en écrivant ceci :

```
Tantque ( Tab[j-1] > v ) et ( j >0 ) faire
          Tab[ j ]  ← Tab[ j-1 ];
           j ← j-1;
FinTant ;
```

Il a eu des problèmes de dépassement d'indice de tableau lors de l'implémentation de son programme.

Essayez d'analyser l'origine du problème en notant que la présence d'une sentinelle élimine le problème.

D) Complexité :

Choix opération

Choisissons comme opération élémentaire **la comparaison de deux cellules** du tableau.

Dans le pire des cas le nombre de comparaisons "**Tantque** Tab[j-1] > v **faire**" est une valeur qui ne dépend que de la longueur **i** de la partie (**a**1, **a**2, ... , **a**i) déjà rangée. Il y a donc au pire i comparaisons pour chaque i variant de 2 à n :
La complexité au pire en nombre de comparaison est donc égale à la somme des n termes suivants (i = 2, i = 3,.... i = n)
C = 2 + 3 + 4 +...+ n = n(n+1)/2 -1 comparaisons au maximum. (c'est la somme des n premiers entiers moins 1).

La complexité au pire en nombre de comparaison est de l'ordre de n^2, que l'on écrit $O(n^2)$.

Choix opération

Choisissons maintenant comme opération élémentaire **le transfert d'une cellule** du tableau.

Calculons par dénombrement du nombre de transferts dans le pire des cas .

Il y a autant de transferts dans la boucle "**Tantque** Tab[j-1] > v **faire**" qu'il y a de comparaisons il faut ajouter 2 transferts par boucle "**pour** i **de** 2 **jusquà n faire**", soit au total dans le pire des cas :
$C = n(n+1)/2 + 2(n-1) = (n^2 + 5n - 4)/2$

La complexité du tri par insertion au pire en nombre de transferts est de l'ordre de n^2, que l'on écrit $O(n^2)$.

E) Programme Delphi (tableau d'entiers) :

```
program TriParInsertion;
const N = 10;  { Limite supérieure de tableau }
type TTab = array [0..N]of integer;   { TTab : Type Tableau }
var Tab : TTab ;

 procedure TriInsertion (var Tab:TTab) ;
{ Implantation Pascal de l'algorithme }
var i, j, v : integer;
begin
  for i := 2 to N do
  begin
     v := Tab[ i ];
     j := i ;
    while Tab[ j-1 ] > v do
    begin
        Tab[ j ] := Tab[ j-1 ] ;
         j := j - 1 ;
    end;
    Tab[ j ] := v ;
  end
end;

procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
     Tab[i] := random(100);
  Tab[0]:= -Maxint ; // la sentinelle est l'entier le plus petit du type
integer sur la machine
end;
```

```
procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
      writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI PAR INSERTION');
  writeln;
  Impression(Tab);
  TriInsertion(Tab);
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI PAR INSERTION

------------------------------------------------------------
 62 |  15 |  34 |   3 |  25 |  22 |  63 |   3 |  66 |  17 |
------------------------------------------------------------
  3 |   3 |  15 |  17 |  22 |  25 |  34 |  62 |  63 |  66 |
------------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriInsert
{
 // le tableau à trier:
 static int[] table = new int[10] ;
 /*-------------------------------------------------------------------------
  Dans la cellule de rang 0 se trouve une sentinelle chargée d'éviter de tester dans la boucle tantque .. faire si
  l'indice j n'est pas  inférieur à 1, elle aura une valeur inférieure à toute  valeur possible de la liste
   -------------------------------------------------------------------------*/
 static void Impression ( )  {
  // Affichage du tableau
  int n = table.length-1;
  for ( int i = 0; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void Initialisation  ( )  {
  // remplissage aléatoire du tableau
  int n = table.length-1;
  for ( int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
  //sentinelle à l'indice 0 :
  table[0] = -Integer.MAX_VALUE;
 }
```

```
 public static void main(String[ ] args)  {
  Initialisation  ( );
  System.out.println("Tableau initial :");
  Impression ( );
  TriInsert ( );
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
```

```
 static void TriInsert ( ) {
  // sous-programme de Tri par insertion :
  int n = table.length-1;
  for ( int i = 2; i <= n; i++)
  {
   int v = table[i];
   int j = i;
   while (table[ j-1 ] > v)
   {// travail sur la partie déjà triée (a1, a2, ... , ai)
    table[ j ] = table[ j-1 ]; // on décale l'élément
    j--;  // on passe au rang précédent
   }
   table[ j ]  = v ; //on recopie ai dans la place libérée
  }
 }
```

```
}
```

Le tri rapide

méthode Sedgewick

C'est le plus performant des tris en table qui est certainement celui qui est le plus employé dans les programmes. Ce tri a été trouvé par C.A.Hoare, nous nous référons à Robert Sedgewick qui a travaillé dans les années 70 sur ce tri et l'a amélioré et nous renvoyons à son ouvrage pour une étude complète de ce tri. Nous donnons les principes de ce tri et sa complexité en moyenne et au pire.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par insertion est de parcourir la liste L = liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en la divisant systématiquement en deux sous-listes L1 et L2. L'une de ces deux sous-listes est telle que tous ses éléments sont inférieurs à tous ceux de l'autre liste , la division en sous-liste a lieu en travaillant séparément sur chacune des deux sous-listes en appliquant à nouveau la même division à chaque sous-liste jusqu'à obtenir uniquement des sous-listes à un seul élément.

C'est un algorithme dichotomique qui divise donc le problème en deux sous-problèmes dont les résultats sont réutilisés par recombinaison, il est donc de complexité **O(n.log(n))**.

Pour partitionner une liste L en deux sous-listes L1 et L2 :

- on choisit une valeur quelconque dans la liste L (la dernière par exemple) que l'on dénomme **pivot**,
- puis on construit la sous-liste L1 comme comprenant tous les éléments de L dont la valeur est inférieure ou égale au **pivot**,
- et l'on construit la sous-liste L2 comme constituée de tous les éléments dont la valeur est supérieure au **pivot**.

Soit sur un exemple de liste L :

L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]
prenons comme pivot la dernière valeur pivot = 16

Nous obtenons deux sous-listes L1 et L2 :

L1 = [4, 14, 3, 2]
L2 = [23, 45, 18, 38, 42]

A cette étape voici l'arrangement de L :

L = L1 + pivot + L2 = [**4, 14, 3, 2**, **16**, **23, 45, 18, 38, 42**]

En effet, en travaillant sur la table elle-même par réarrangement des valeurs, le pivot **16** est placé au bon endroit directement :

[**4<16, 14<16, 3<16, 2<16**, **16**, **23>16, 45>16, 18>16, 38>16, 42>16**]

En appliquant la même démarche au deux sous-listes : L1 (pivot=2) et L2 (pivot=42)
[**4, 14, 3, 2**, **16**, **23, 45, 18, 38, 42**] nous obtenons :

L11=[ ] liste vide
L12=[**3, 4, 14**]
L1=L11 + pivot + L12 = (**2**,**3, 4, 14**)

L21=[**23, 38, 18**]
L22=[**45**]
L2=L21 + pivot + L22 = (**23, 38, 18, 42, 45**)

A cette étape voici le nouvel arrangement de L :

L = [(**2**,**3, 4, 14**), **16**, (**23, 38, 18, 42, 45**)]

etc...
Ainsi
de proche en proche en subdivisant le problème en deux sous-problèmes, à chaque étape nous obtenons un pivot bien placé.

B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau de dimension unT[...] en mémoire centrale.

Le processus de partionnement décrit ci-haut (appelé aussi segmentation) est le point central du tri rapide, nous construisons une fonction **Partition** réalisant cette action .

Comme l'on appliquant la même action sur les deux sous-listes obtenues après partition, la méthode est donc récursive, le tri rapide est alors une procédure récursive.

B-1) Voici une spécification générale de la procédure de tri rapide :

Tri Rapide sur [a..b]
Partition [a..b] renvoie **pivot** & [a..b] = **[x .. pivot']**+**[pivot]**+**[pivot'' .. y]**
Tri Rapide sur **[pivot'' .. y]**
Tri Rapide sur **[x .. pivot']**

B-2) Voici une spécification générale de la fonction de partionnement :

La fonction de partionnement d'une liste [a..b] doit répondre aux deux conditions suivantes :

- renvoyer la valeur de l'indice noté **i** d'un élément appelé pivot qui est bien placé définitivement : pivot = T[i],
- établir un réarrangement de la liste [a..b] autour du pivot tel que :

[a..b] = **[x .. pivot']**+**[pivot]**+**[pivot'' .. y]**

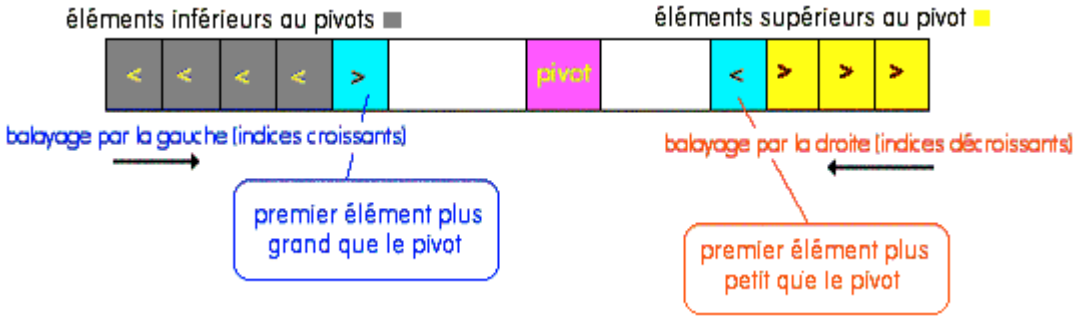
[x .. pivot'] = T[G] , .. , T[i-1]
(où : x = T[G] et pivot' = T[i-1]) tels que les T[G] , .. , T[i-1] sont tous inférieurs à T[i] ,

[pivot'' .. y] = T[i+1] , .. , T[D]
(où : y = T[D] et pivot'' = T[i+1]) tels que les T[i+1] , .. , T[D] sont tous supérieurs à T[i] ,

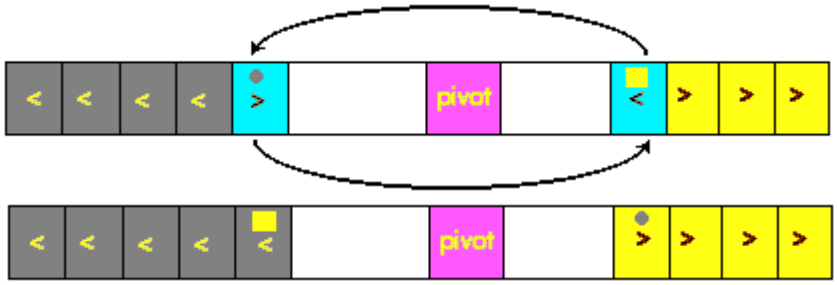
Il est proposé de **choisir arbitrairement le pivot** que l'on cherche à placer, puis ensuite de balayer la liste à réarranger dans les deux sens (par la gauche et par la droite) en construisant une sous-liste à gauche dont les éléments ont une valeur inférieure à celle du pivot et une sous-liste à droite dont les éléments ont une valeur supérieure à celle du pivot .

1) Dans le balayage par la gauche, on ne touche pas à un élément si sa valeur est inférieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous-liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus grande que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous-liste mais plutôt dans l'autre sous-liste.

2) Dans le balayage par la droite, on ne touche pas à un élément si sa valeur est supérieure au pivot (les éléments sont considérés comme étant alors dans la bonne sous-liste) on arrête ce balayage dès que l'on trouve un élément dont la valeur est plus petite que celle du pivot. Dans ce dernier cas cet élément n'est pas à sa place dans cette sous-liste mais plutôt dans l'autre sous-liste.

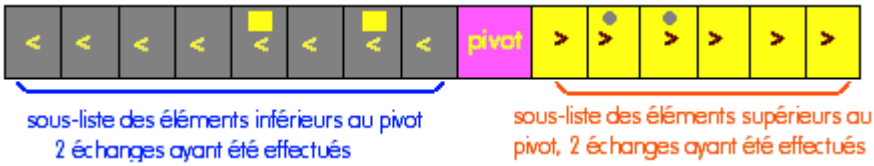


3) on procède à l'échange des deux éléments mal placés dans chacune des sous-listes :



4) On continue le balayage par la gauche et le balayage par la droite tant que les éléments sont bien placés (valeur inférieure par la gauche et valeur supérieure par la droite), en échangeant à chaque fois les éléments mal placés.

5) La construction des deux sous-listes est terminée dès que l'on atteint (ou dépasse) le pivot.



Appliquons cette démarche à l'exemple précédent : L = [4, 23, 3, 42, 2, 14, 45, 18, 38, 16]

- Choix arbitraire du pivot : l'élément le plus à droite ici **16**

- Balayage à gauche :
  4 < 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 16**]
  23 > 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage gauche,
  liste en cours de construction :[**4, 23**, **16**]
- Balayage à droite :
  38 > 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 23**, **16**, **38**]
  18 > 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 23**, **16**, **18**, **38**]
  45 > 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 23**, **16**, **45**, **18**, **38**]
  14 < 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
  liste en cours de construction : [**4, 23**, **16**, **14**, **45**, **18**, **38**]

- Echange des deux éléments mal placés :

  [**4,** **23** , 16, **14** , 45, 18, 38] ----> [**4, 14** , 16, **23, 45, 18, 38**]

- On reprend le balayage gauche à l'endroit où l'on s'était arrêté :
  ----------↓----------
  [**4**, **14**, 3, 42, 2, **23**, **45**, **18**, **38**, 16]
  3 < 16 => il est dans la bonne sous-liste, on continue
  liste en cours de construction : [**4, 14, 3** , 16 , **23, 45, 18, 38**]
  42 > 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête de nouveau le balayage gauche,
  liste en cours de construction : [**4, 14, 3, 42** , 16, **23, 45, 18, 38**]

- On reprend le balayage droit à l'endroit où l'on s'était arrêté :
  ----------↓----------
  [**4**, **14**, 3, 42, 2, **23**, **45**, **18**, **38**, 16]

  2 < 16 => il est mal placé il n'est pas dans la bonne sous-liste, on arrête le balayage droit,
  liste en cours de construction : [**4, 14, 3, 42** , 16, **2, 23, 45, 18, 38**]

- On procède à l'échange des deux éléments mal placés :
  [**4, 14**, **3**, **42**, 16, **2** , **23**, **45**, **18**, **38**] ----> [**4, 14, 3, 2** , 16, **42, 23, 45, 18, 38**]
  et l'on arrête la construction puisque nous sommes arrivés au pivot la fonction partition a terminé son travail elle a évalué :

  - le pivot : **16**
  - la sous-liste de gauche : **L1 = [4, 14, 3, 2]**
  - la sous-liste de droite : **L2 = [23, 45, 18, 38, 42]**
  - la liste réarrangée : [**4**, **14**, **3**, **2**, **16**, **42**, **23**, **45**, **18**, **38**]

  Il reste à recommencer les mêmes opérations sur les parties L1 et L2 jusqu'à ce que les partitions ne contiennent plus qu'un seul élément.

C) Algorithme :

```
Global :Tab[min..max] tableau d'entier

fonction Partition( G , D : entier )  résultat : entier
Local : i , j ,  piv , temp : entier
début
    piv  ← Tab[D];
    i  ← G-1;
    j  ← D;
    repeter
      repeter  i ← i+1 jusquà Tab[i] >= piv;
```

```
      repeter  j ← j-1 jusquà Tab[j] <= piv;
      temp ←  Tab[i];
      Tab[i] ← Tab[j];
      Tab[j] ←  temp
    jusquà j <= i;
    Tab[j] ←  Tab[i];
    Tab[i] ←  Tab[d];
    Tab[d] ←  temp;
    résultat ←  i
FinPartition

Algorithme TriRapide( G  , D : entier );
Local :  i : entier
début
 si D > G alors
   i ←Partition( G , D );
  TriRapide( G , i-1 );
  TriRapide( i+1 , D );
 Fsi
FinTRiRapide
```

Nous supposons avoir mis une sentinelle dans le tableau, dans la première cellule la plus à gauche, avec une valeur plus petite que n'importe qu'elle autre valeur du tableau.

Cette sentinelle est utile lorsque le pivot choisi aléatoirement se trouve être le plus petit élément de la table /pivot = min (**a**1, **a**2, ... , **a**n)/ :

> Comme nous avons:
> ∀j , Tab[j] > piv , alors la boucle :
>
> "**repeter** j ← j-1 **jusquà** Tab[j] <= piv ;"
> pourrait ne pas s'arrêter ou bien s'arrêter sur un message d'erreur.

La sentinelle étant plus petite que tous les éléments y compris le pivot arrêtera la boucle et encore une fois évite de programmer le cas particulier du pivot = min (**a**1, **a**2, ... , **a**n).

D) Complexité :

Nous donnons les résultats classiques et connus mathématiquement (pour les démonstrations nous renvoyons aux ouvrages de R.Sedgewick & Aho-Ullman cités dans la bibliographie).

Choix opération

L'opération élémentaire choisie est **la comparaison de deux cellules** du tableau.

Comme tous les algorithmes qui divisent et traitent le problème en deux sous-problèmes le nombre moyen de comparaisons est en **O(n.log(n))** que l'on nomme **complexité moyenne.** La notation **log** (x) est utilisée pour le logarithme à base 2, $\mathbf{log_2}$ (x).

L'expérience pratique montre que cette complexité moyenne en **O(n.log(n))** n'est atteinte que lorsque les pivots successifs divisent la liste en deux sous-listes de taille à peu près équivalente.
Dans le pire des cas (par exemple le pivot choisi est systématiquement à chaque fois la plus grande valeur) on montre que la complexité est en $\mathbf{O(n^2)}$.

Comme la littérature a montré que ce tri était le meilleur connu en complexité, il a été proposé beaucoup d'améliorations permettant de choisir un pivot le meilleur possible, des combinaisons avec d'autres tris par insertion généralement, si le tableau à trier est trop petit….

Ce tri est pour nous un excellent exemple en **n.log(n)**.illustrant la récursivité.

E) Programme Delphi (tableau d'entiers) :

```
program TriQuickSort;

const N = 10;  { Limite supérieure de tableau }
type TTab = array [0..N] of integer;   { TTab : Type Tableau }
var Tab : TTab ;

function Partition ( G , D : integer) : integer;
  var i , j : Integer;
      piv, temp  : integer;
begin
    i := G-1;
    j := D;
    piv := Tab[D];
    repeat
       repeat i := i+1 until Tab[i] >= piv;
       repeat j := j-1 until Tab[j] <= piv;
       temp :=Tab[i];
       Tab[i] :=Tab[j];
       Tab[j] :=temp;
    until j <= i;
    Tab[j] := Tab[i];
    Tab[i] := Tab[d];
    Tab[d] := temp;
    result := i;
end;{Partition}
```

```
procedure TriRapide( G, D : integer);
var i: Integer;
begin
  if D>G then
  begin
    i := Partition( G , D );
    TriRapide( G , i-1 );
    TriRapide( i+1 , D );
  end
end;{TriRapide}
```

```
procedure Initialisation(var Tab:TTab) ;
{ Tirage aléatoire de N nombres de 1 à 100 }
var i : integer;  { i : Indice de tableau de N colonnes }
begin
  randomize;
  for i := 1 to N do
    Tab[i] := random(100);
  Tab[0] := -Maxint ; // la sentinelle
end;

procedure Impression(Tab:TTab) ;
{ Affichage des N nombres dans les colonnes }
var i : integer;
begin
  writeln('------------------------');
  for i:= 1 to N do write(Tab[i] : 3, ' | ');
      writeln;
end;

begin
  Initialisation(Tab);
  writeln('TRI RAPIDE');
  writeln;
  Impression(Tab);
  TriRapide( 1 , N );
  Impression(Tab);
  writeln('----------------------');
end.
```

Résultat de l'exécution du programme précédent :

```
TRI RAPIDE

-----------------------------------------------------------
 17 |  32 |  14 |  45 |  54 |  50 |  60 |  10 |  68 |  12 |
-----------------------------------------------------------
 10 |  12 |  14 |  17 |  32 |  45 |  50 |  54 |  60 |  68 |
-----------------------------------------------------------
```

E) Programme Java (tableau d'entiers) :

```
class ApplicationTriQSort
{
 static int[] table = new int[21] ; // le tableau à trier en attribut
 /*  Les cellules [0] et [20] contiennent
     des sentinelles,
     Les cellules utiles vont de 1 à 19.
     (de 1 à table.length-2)
 */

 static void impression ( )
 {
  // Affichage sans les sentinelles
  int n = table.length-2;
  for ( int i = 1; i<=n; i++)
   System.out.print(table[i]+" , ");
  System.out.println();
 }

 static void initialisation  ( )
 {
  // remplissage aléatoire du tableau
  int n = table.length-2;
  for(int i = 1; i<=n; i++)
   table[i] = (int)(Math.random()*100);
  // Les sentinelles:
  table[0] = -Integer.MAX_VALUE;
  table[n+1] = Integer.MAX_VALUE;
 }

 // ----> Tri rapide :
```

```
static int partition (int G, int D )
{ // partition / Sedgewick /
 int i, j, piv, temp;
 piv = table[D];
 i = G-1;
 j = D;
 do
  {
   do
    i++;
   while (table[i]<piv);
   do
    j--;
   while (table[j]>piv);
   temp = table[i];
   table[i] = table[j];
   table[j] = temp;
  }
 while(j>i);
 table[j] = table[i];
 table[i] = table[D];
 table[D] = temp;
 return i;
}
```

```
static void QSort (int G, int D )
{ // tri rapide, sous-programme récursif
 int i;
 if(D>G)
 {
  i = partition(G,D);
  QSort(G,i-1);
  QSort(i+1,D);
 }
}
```

```
 public static void main(string[ ] args)
 {
  Initialisation ( );
  int n = table.length-2;
  System.out.println("Tableau initial :");
  Impression ( );
  QSort(1,n);
  System.out.println("Tableau une fois trié :");
  Impression ( );
 }
}
```

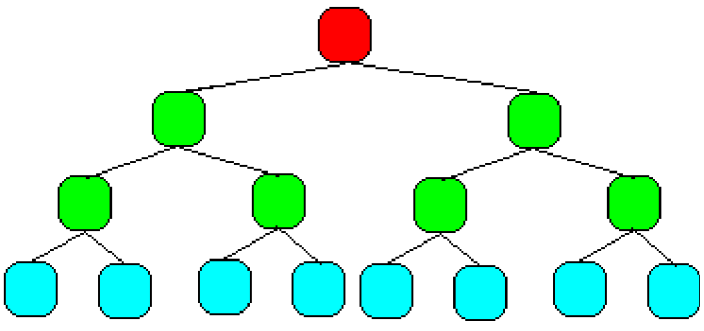
Le tri par arbre

C'est un tri également appelé tri par tas (*heapsort*, en anglais). Il utilise une structure de données temporaire dénommée "tas" comme mémoire de travail.

Définitions préliminaires

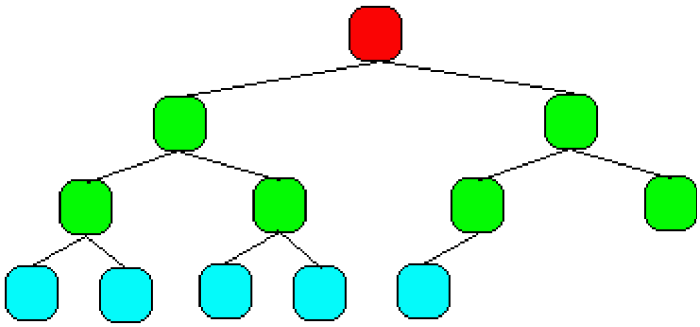
Définition - 1 / Arbre parfait :

c'est un arbre binaire dont tous les noeuds de chaque niveau sont présents sauf éventuellement au dernier niveau où il peut manquer des noeuds (noeuds terminaux = feuilles), dans ce cas l'arbre parfait est un arbre binaire incomplet et les feuilles du dernier niveau doivent être regroupées à partir de la gauche de l'arbre
.

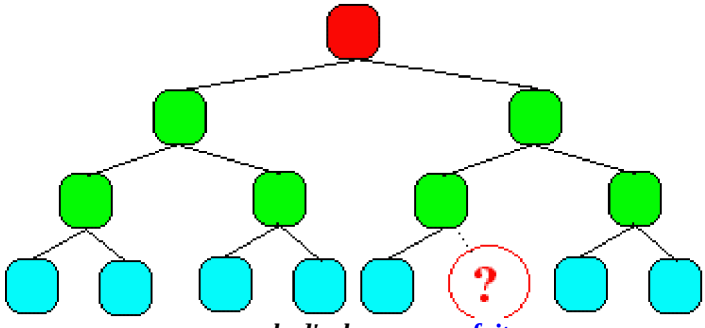


un arbre parfait complet

Amputons l'arbre parfait précédent de ses trois feuilles situées sur le bord droit, les cinq premières feuilles de gauche ne changeant pas, on obtient toujours un arbre parfait mais il est incomplet :



un autre arbre parfait incomplet

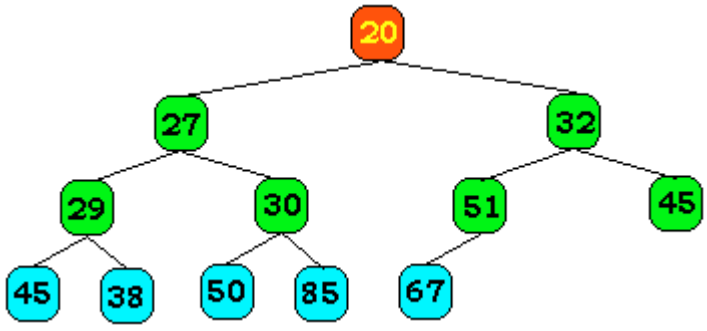


exemple d'arbre non parfait

Définition - 2 / Arbre partiellement ordonné :

C'est un arbre étiqueté dont les noeuds appartiennent à un ensemble muni d'une relation d'ordre total (les nombres entiers, réels etc... en sont des exemples) tel que pour un noeud donné tous ses fils ont une valeur supérieure ou égale à celle de leur père.

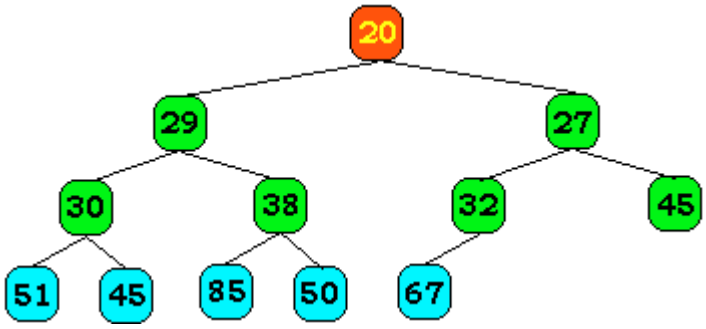
Exemple d'un arbre partiellement ordonné sur l'ensemble {20, 27,29, 30, 32, 38, 45, 45, 50, 51, 67 ,85 } d'entiers naturels :



Nous remarquons que **la racine d'un tel arbre est toujours l'élément de l'ensemble possédant la valeur minimum** (le plus petit élément de l'ensemble), car la valeur de ce noeud par construction est inférieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses descendants c'est le minimum.

Si donc nous arrivons à ranger une liste d'éléments dans un tel arbre le minimum de cette liste est atteignable immédiatement comme racine de l'arbre.

Exemple d'un autre arbre partiellement ordonné sur le même ensemble {20, 27,29, 30, 32, 38, 45, 45, 50, 51, 67 ,85 } d'entiers naturels (il n'y a pas unicité) :



Définition - 3 / Le tas :

On appelle **tas** un tableau représentant un **arbre parfait partiellement ordonné**.

Principe du tri par tas

C'est une variante de méthode de tri par sélection où l'on parcourt le tableau des éléments en sélectionnant et conservant les minimas successifs (plus petits éléments partiels) dans un arbre parfait partiellement ordonné.

A) Spécification abstraite

Nous supposons que les données $\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$ sont mises sous forme d'une liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$), le principe du tri par tas est de parcourir la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) en ajoutant chaque élément **a**k dans **un arbre parfait partiellement ordonné**.

- L'insertion d'un nouvel élément $\mathbf{a}_k$ dans l'arbre a lieu **dans la dernière feuille vide de l'arbre à partir de la gauche** (ou bien si le niveau est complet en recommençant un nouveau niveau par sa feuille la plus à gauche) et, en effectuant des échanges tant que la valeur de $\mathbf{a}_k$ est inférieur à celle de son pére.

- Lorsque tous les éléments de la liste seront placés dans l'arbre, l'élément minimum "$\mathbf{a}_i$" de la liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) se retrouve à la racine de l'arbre qui est alors partiellement ordonné.

- On recommence le travail sur la nouvelle liste ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) -{$\mathbf{a}_i$}(c'est la liste précédente privée de son minimum),

  pour cela on supprime l'élément minimum **a**i de l'arbre pour le mettre dans la liste triée puis,

  on prend l'élément de la dernière feuille du dernier niveau et on le place à la racine.

  On effectue ensuite des échanges de contenu avec le fils dont le contenu est inférieur, en partant de la racine, et en descendant vers le fils avec lequel on a fait un échange, ceci tant qu'il n'a pas un contenu inférieur à ceux de ses deux fils (ou de son seul fils) ou tant qu'il n'est pas à une feuille.

- On recommence l'opération de suppression et d'échanges éventuels **jusqu'à ce que l'arbre ne contienne plus aucun élément**.

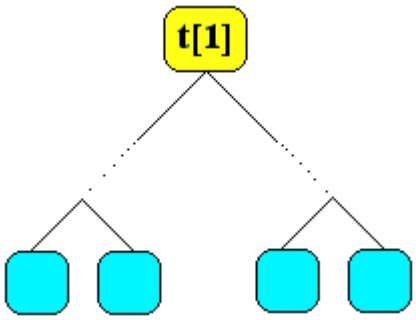
B) Spécification concrète

La suite ($\mathbf{a}_1$, $\mathbf{a}_2$, ... , $\mathbf{a}_n$) est rangée dans un tableau à une dimension T[...] correspondant au tableau d'initialisation. Puis les éléments de ce tableau sont ajoutés et traités un par un dans un arbre avant d'être ajoutés dans un tableau trié en ordre décroissant ou croissant, selon le choix de l'utilisateur.

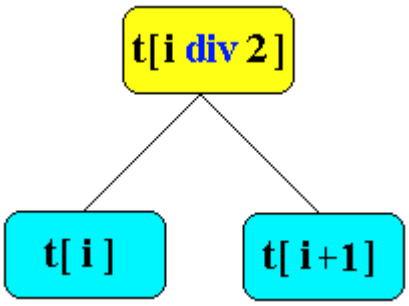
Signalons qu'un arbre binaire parfait se représente classiquement par un tableau :

Si t est ce tableau, on a les règles suivantes :

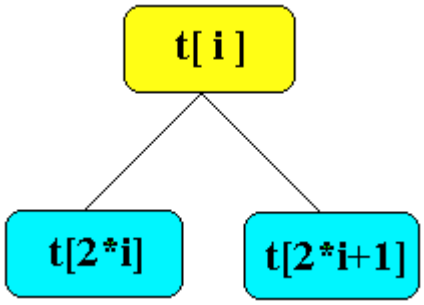
- t[1] est la racine :



- t[i **div** 2] est le père de t[i] pour i > 1 :



- t[2 * i] et t[2 * i + 1] sont les deux fils, s'ils existent, de t[i] :



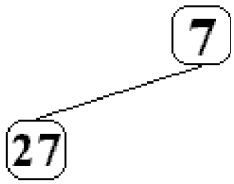
- si **p** est le nombre de noeuds de l'arbre et si 2 * i = p, t[i] n'a qu'un fils, t[p].
  si i est supérieur à p **div** 2, t[i] est une feuille.

Figures illustrant le placement des éléments de la liste dans l'arbre

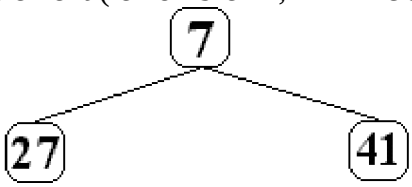
Exemple d'initialisation sur un tableau à 15 éléments :

Insertion du premier élément (le nombre **7**) à la racine de l'arbre :

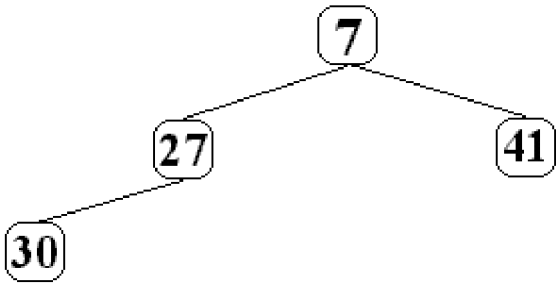
Insertion du second élément (le nombre **27**, 27 > 7 donc c'est un fils du noeud 7) :



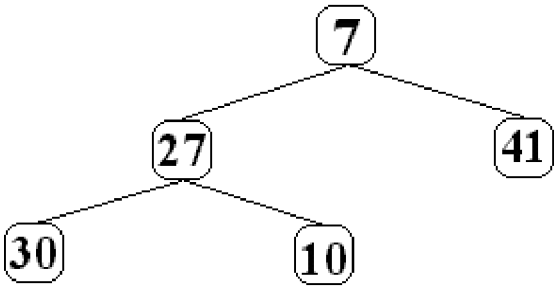
Insertion du troisième élément (le nombre **41**, 41 > 7 c'est un fils du noeud 7) :



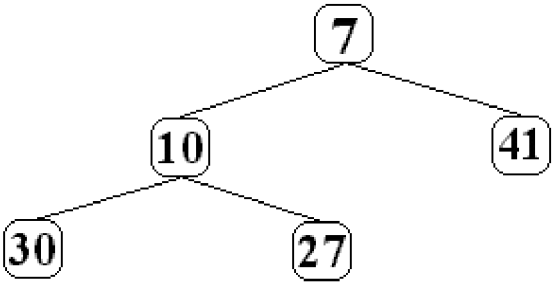
Insertion du quatrième élément (le nombre **30**, comme le niveau des fils du noeud 7 est complet, 30 est placé automatiquement sur une nouvelle feuille d'un nouveau niveau, puis il est comparé à son père 27, 30 > 27 c'est donc un fils du noeud 27 il n'y a pas d'échange) :



Insertion du cinquième élément (le nombre 10) : L'insertion du nouvel élément 10 dans l'arbre a lieu automatiquement dans la dernière feuille vide de l'arbre à partir de la gauche, ici le fils droit de 27 :



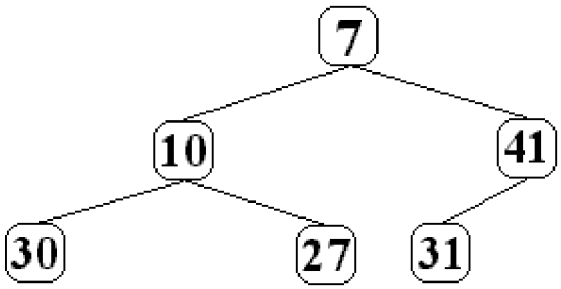
Puis 10 est comparé à son père 27, cette fois 10 est plus petit que 27, il y a donc échange des places entre 27 et 10, ce qui donne un nouvel arbre :



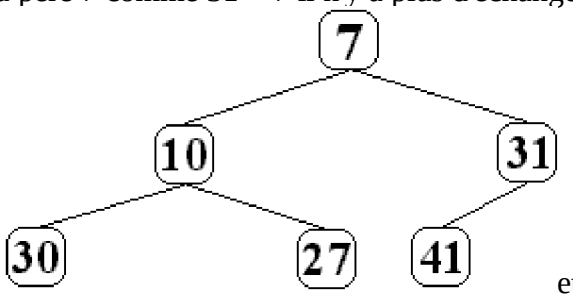
Puis 10 est comparé à son nouveau père 7, cette fois il n'y a pas d'échange car 10 est plus grand que 7.

Le processus continue avec l'élément suivant de la liste le nombre 31:

31 est automatiquement rangé sur la première feuille disponible à gauche soit le fils gauche de 41:



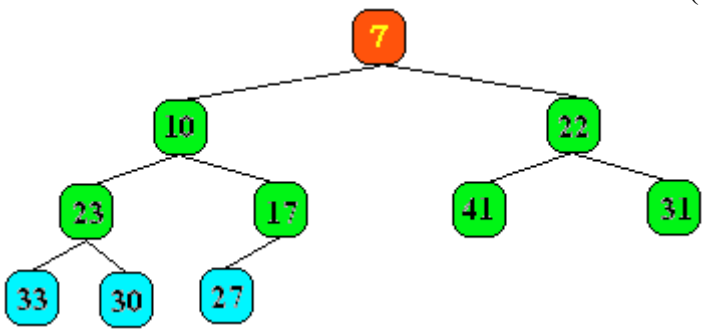
Puis 31 est comparé à son père, comme 31 < 41 il y a échange des valeurs, puis 31 est comparé à son nouveau père 7 comme 31 > 7 il n'y a plus d'échange :



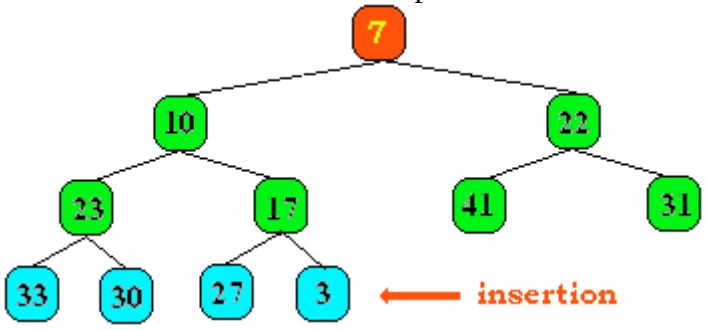
etc....

Supposons que l'arbre ait été construit sur les dix premiers éléments du tableau et observons maintenant comment l'élément minimum de la liste qui est le onzième élément, soit le nombre 3, est rangé dans l'arbre.

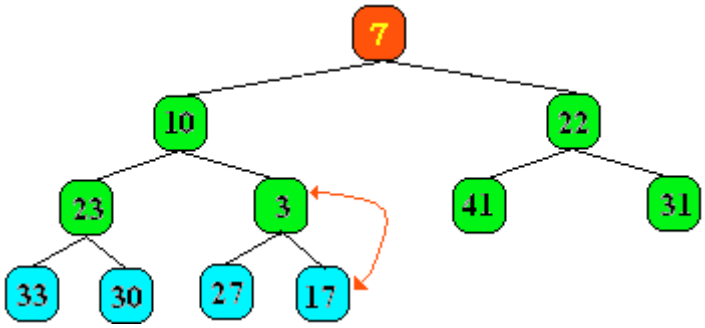
Voici l'état de l'arbre avant introduction du nombre 3 (quatre niveaux de nœuds) :



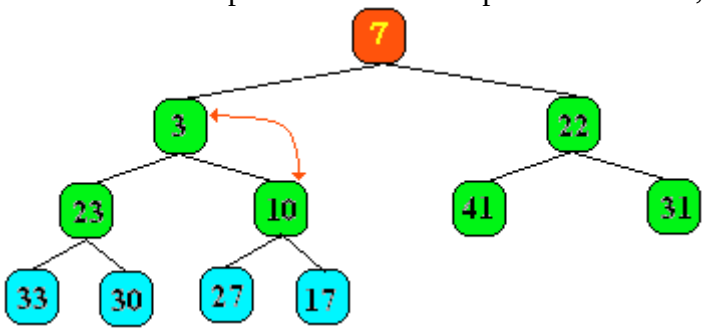
Le nombre 3 est introduit sur la première feuille libre du niveau quatre :



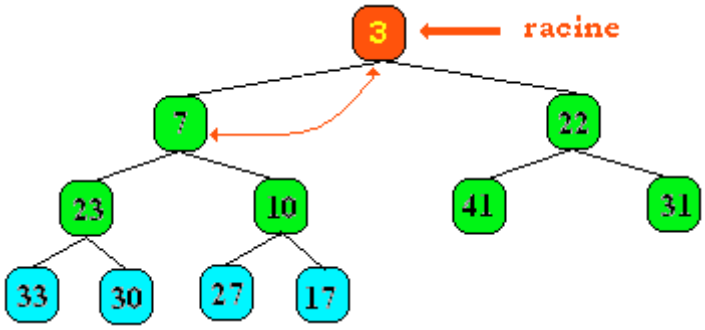
Il est comparé à son père le noeud 17, comme 3 < 17, il y a alors échange :



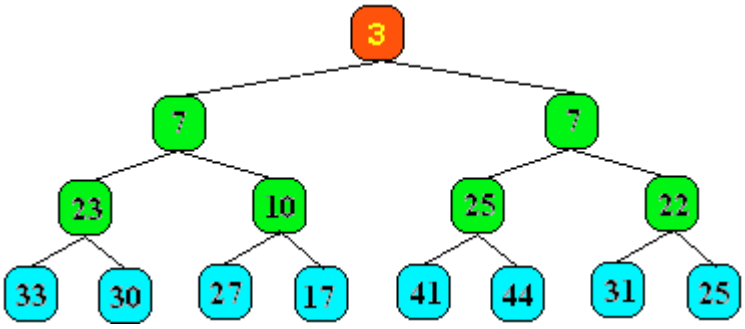
Il est ensuite comparé à son nouveau père le noeud 10, comme 3 < 10, il y a alors échange :



Il est enfin comparé à son dernier père (la racine de l'arbre), comme 3 < 7, il y a alors échange :



C'est l'élément **3** qui est maintenant **la racine** de l'arbre, comme les 4 éléments suivants sont plus grand que 3, ils seront rangés plus bas dans l'arbre (cf. figure ci-dessous) :

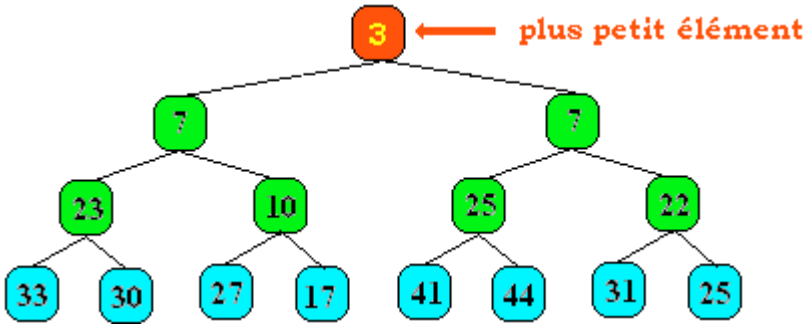


Conclusion sur le premier passage :

La liste initiale :

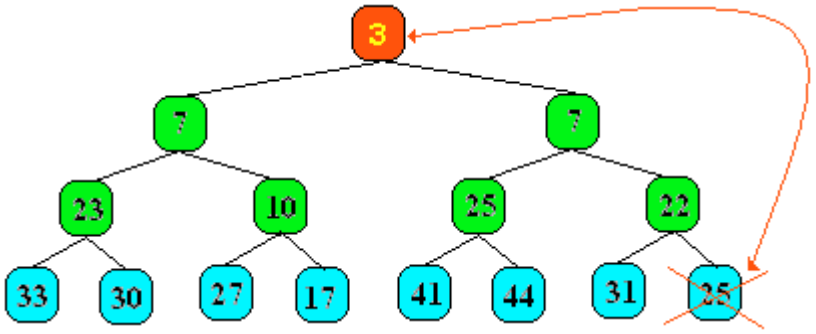
7	27	41	30	10	31	22	33	23	17	3	25	44	7	25

est finalement stockée ainsi :

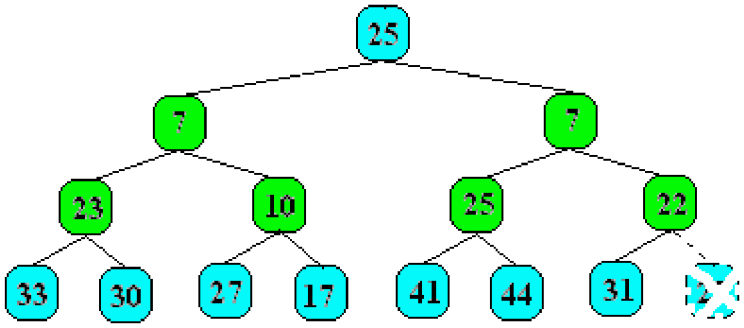


Figures illustrant la suppression de la racine

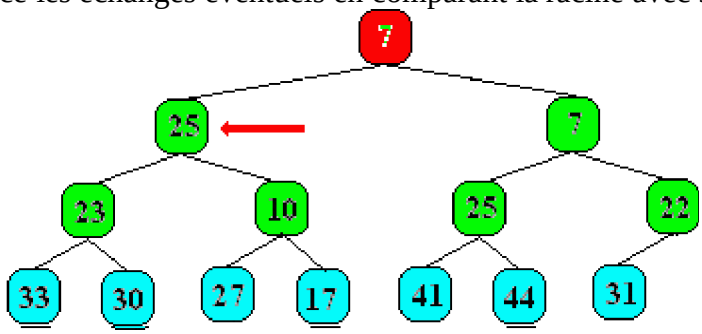
Le nombre **3** est le plus petit élément, on le supprime de l'arbre et l'on prend l'élément de la dernière feuille du dernier niveau (ici le nombre 25) et on le place à la racine (à la place qu'occupait le nombre 3)



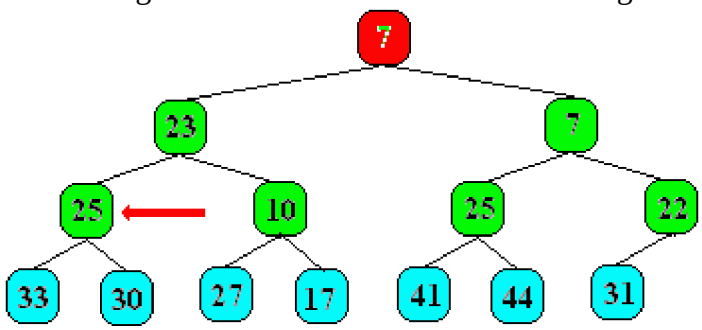
ce qui donne comme nouvelle disposition :



et l'on recommence les échanges éventuels en comparant la racine avec ses descendants :



le fils gauche 23 est inférieur à 25 => échange



Arrêt du processus (33 > 25 et 30 > 25)

On obtient le deuxième plus petit élément à la racine de l'arbre, ici le nombre 7.
Puis l' on continue à "vider" ainsi l'arbre et déplaçant les feuilles vers la racine et en échangeant les noeuds mal placés.

A la fin, lorsque l'arbre a été entièrement vidé, nous avons extrait à chaque étape le plus petit élément de chaque sous liste restante et nous obtenons les éléments de la liste classés par ordre croissant ou décroissant selon notre choix (dans notre exemple si nous stockons les minima successifs de gauche à droite dans une liste nous obtiendrons une liste classée par ordre croissant de gauche à droite).

En résumé notre version de tri par tas comporte les étapes suivantes :

- initialisation : ajouter successivement chacun des n éléments dans le tas t[1..p]; p augmente de 1 après chaque adjonction . A la fin on a un tas de taille p = n.
- tant que p est plus grand que 1, supprimer le minimum du tas (p décroît de 1), réorganiser le tas, ranger le minimum obtenue à la $(p + 1)^{ième}$ place.

On en déduit l'algorithme ci-dessous composé de 2 sous algorithmes Ajouter pour la première étape, et Supprimer pour la seconde.

C) Algorithme :

```
Algorithme Ajouter
Entrée  P : entier ; x : entier // P nombre d'éléments dans le tas, x élément à ajouter
          Tas[1..max]  : tableau d'entiers // le tas
Sortie  P : entier
          Tas[1..max] // le tas
Local  j, temp : entiers
début
  P ← P + 1 ; // incrémentation du nombre d'éléments du tas
  j ← P ; // initialisation de j à la longueur du tas (position de la dernière feuille)
  Tas[P] ← x ; // ajout l'élément x à la dernière feuille dans le tas
  Tantque (j > 1) et (Tas[j] < Tas[j div 2]) faire ; // tant que l'on est pas arrivé à la racine et
que le "fils" est inférieur à son "père", on permute les 2 valeurs
        temp ← Tas[j] ;
        Tas[j] ← Tas[j div 2] ;
        Tas[j div 2] ← temp ;
        j ← j div 2 ;
  finTant
FinAjouter

Algorithme Supprimer
Entrée : P : entier // P nombre d'éléments contenu dans l'arbre
          Tas[1..max]  : tableau d'entiers // le tas
Sortie : P : entier // P nombre d'éléments contenu dans l'arbre
          Tas[1..max]  : tableau d'entiers // le tas
          Lemini : entier  // le plus petit de tous les éléments du tas
Local i, j, temp : entiers ;
début
  Lemini ← Tas[1] ; // retourne  la racine (minimum actuel) pour stockage éventuel
  Tas[1] ←Tas[P] ; // la racine prend la valeur de la dernière feuille de l'arbre
  P ← P - 1 ;
  j ←1 ;
  Tantque j <= (P div 2) faire
     // recherche de l'indice i du plus petit des descendants de Tas[j]
     si (2 * j = P ) ou (Tas[2 * j] < Tas[2 * j + 1])
           alors i ← 2 * j ;
     sinon i ←2 * j +1 ;
     Fsi;
    // Echange éventuel entre Tas[j] et son fils Tas[i]
    si Tlab[j] > Tlab[i] alors
          temp ← Tlab[j] ;
          Tlab[j] ← Tlab[i] ;
          Tlab[i] ← temp ;
          j ← i ;
    sinon Sortir ;
    Fsi;
  finTant
FinSupprimer
```

```
Algorithme Tri_par_arbre
Entrée  Tas[1..max] // le tas
        TabInit[1..max] // les données initiales
Sortie   TabTrie[1..max]: tableaux d'entiers //  tableau une fois trié
Local    P  : Entier //  P le nombre d'éléments à trier
         Lemin : entier  // le plus petit de tous les éléments du tas
début
   P ← 0;
  Tantque P < max faire
     Ajouter(P,Tas,TabInit[P+1]); // appel de l'algorithme Ajouter
  finTant;
  Tantque P >= 1 faire
     Supprimer(P,Tas,Lemin) ; // appel de l'algorithme Supprimer
     TabTrie[max-P] ← Lemin ; // stockage du minimum dans le nouveau tableau
  finTant;
Fin Tri_par_arbre
```

D) Complexité :

Cette version de l'algorithme construit le tas par n appels de la procédure **Ajouter** et effectue les sélections par n - 1 appels de la procédure supprimer.

Le coût et de l'ordre de $\sum_{i=1}^{n} \log_2 i$ comparaisons, au pire.

La complexité au pire en nombre de comparaisons est en O(n log n).

Le nombre d'échanges dans le tas est majoré par le nombre de comparaisons et il est du même ordre de grandeur.

La complexité au pire en nombre de transferts du tri par tas est donc en O(n log n).

E) Programme Delphi (tableau d'entiers) :

```
program TriParArbre;
const Max =10; // nombre maximal d'éléments du tableau
type TTab=array [1..Max] of integer;  // TTab : Type Tableau
var Tas, TabInit, TabTrie : TTab; // Tas, tableau initial puis tableau
triéTableau
     P, Lemin : integer;

procedure Ajouter (var Tas : TTab; var P, x : integer);
  var j, temp : integer ;
```

```
begin
   P := P + 1 ;
   J := P ;
   Tas[P] := x ;
   if j>1 then
   While Tas[j] < Tas[j div 2] do
    begin
       temp := Tas[j] ;
       Tas[j] := Tas[j div 2] ;
       Tas[j div 2] := temp ;
       j := j div 2 ;
      if j<=1 then break;
    end
end; // Ajouter

procedure Supprimer (var Tas:TTab; var P, Lemin : integer);
 var i, j, temp : integer ;
 begin
   Lemin := Tas[1] ;
   Tas[1] := Tas[P] ;
   P := P - 1 ;
   j := 1 ;
   While j <= (P div 2) do
   begin
     if (2 * j = P ) or (Tas[2 * j] < Tas[2 * j + 1])
         then i := 2 * j
     else i := 2 * j +1 ;
     if Tas[j] > Tas[i] then
     begin
       temp := Tas[j] ;
       Tas[j] := Tas[i] ;
       Tas[i] := temp ;
       j := i ;
     end
     else break ;
   end
end; // Supprimer

procedure Initialisation(var Tab:TTab) ;
// Tirage aléatoire de Max nombres de 1 à 100
var i : integer;  // i : Indice de tableau
begin
  randomize;
  for i := 1 to Max do
     Tab[i] := random(100);
end;

procedure Impression(Tab:TTab) ;
// Affichage des Max nombres
var i : integer;
```

```
begin
  writeln('----------------------');
  for i:= 1 to Max do write(Tab[i] : 3, ' | ');
      writeln;
end;

begin // TriParArbre
  Initialisation(TabInit);
  writeln('TRI PAR ARBRE');
  writeln;
  Impression(TabInit);
  P:=0;
  while p < Max do
    Ajouter ( Tas, p, TabInit[p+1] );
  while p >= 1 do
  begin
     Supprimer ( Tas, P, Lemin ) ;
     TabTrie[Max-P]:=Lemin
  end;
  Impression(TabTrie);
  writeln('----------------------');
  readln
end. // TriParArbre
```

REMARQUE IMPORTANTE

Notons que dans la procédure nous avons traduit la condition de la boucle :

```
Tantque (j > 1) et (Tas[j] < Tas[j div 2]) faire
        temp ← Tas[j] ;
        Tas[j] ← Tas[j div 2] ;
        Tas[j div 2] ← temp ;
        j ← j div 2 ;
  finTant
```

par les lignes de programmes suivantes :

```
if j>1 then
   While Tas[j] < Tas[j div 2] do
    begin
       temp := Tas[j] ;
       Tas[j] := Tas[j div 2] ;
       Tas[j div 2] := temp ;
       j := j div 2 ;
      if j<=1 then break
    end
```

ceci afin d'éviter un incident dû à un effet de bord. Lorsque l'indice "j" prend par exemple la valeur 1, l'indice "j **div** 2" vaut alors **0** et cette valeur n'est pas valide puisque l'indice de tableau varie entre **1**..**Max**.

On peut pallier aussi d'une autre manière à cet inconvénient en ajoutant une **sentinelle** "à gauche dans le tableau" en étendant la borne inférieure à la valeur **0**, les indices pouvant alors varier entre **0**..**Max**. On obtient une autre écriture de la procédure "Ajouter" qui suit malgré tout l'algorithme de près :

```
type TTab=array [0..Max] of integer;   // 0 est l'indice sentinelle

     procedure Ajouter (var Tas : TTab; var P, x : integer);
       var j, temp : integer ;
     begin
        P := P + 1 ;
        J := P ;
        Tas[P] := x ;
        While (j > 1) et (Tas[j] < Tas[j div 2]) do
         begin
            temp := Tas[j] ;
            Tas[j] := Tas[j div 2] ;
            Tas[j div 2] := temp ;
            j := j div 2 ;
         end
     end; // Ajouter
```

Résultat de l'exécution du programme précédent :

```
TRI PAR ARBRE

-------------------------------------------------------------
 93 |  74 |  13 |   1 |  14 |  22 |  42 |  99 |  16 |  48 |
-------------------------------------------------------------
  1 |  13 |  14 |  16 |  22 |  42 |  48 |  74 |  93 |  99 |
-------------------------------------------------------------
```

3. Rechercher dans un tableau

Les tableaux sont des structures statiques contiguës, il est facile de rechercher un élément fixé dans cette structure. Nous exposons ci-après des algorithmes élémentaires de recherche utilisables dans des applications pratiques par le lecteur.

Essentiellement nous envisagerons la **recherche séquentielle** qui convient lorsqu'il y a peu d'éléments à consulter (quelques centaines), et la **recherche dichotomique** dans le cas où la liste est triée.

3.1 Recherche dans un tableau non trié

Algorithme de recherche séquentielle :

- Soit **t** un tableau d'entiers de **1..n** éléments non rangés.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**)
- Complexité en O(n)

Nous proposons au lecteur 4 versions d'un même algorithme de recherche séquentielle. Le lecteur adoptera une de ces versions en fonction des possibilités du langage avec lequel il compte programmer sa recherche.

Version **Tantque** avec "**et alors**" (si le langage dispose d'un opérateur **et** optimisé)	Version **Tantque** avec "**et**" (opérateur **et** non optimisé)
i ← 1 ; **Tantque** (i ≤ n) **et alors** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**	i ← 1 ; **Tantque** (i < n) **et** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi**

Version **Tantque** avec sentinelle en fin de tableau (rajout d'une cellule supplémentaire en fin de tableau contenant systématiquement l'élément à rechercher)

Version **Tantque** avec sentinelle avec "**et alors**"	Version **Pour** avec instruction de sortie (conseillée si le langage dispose d'un opérateur **Sortirsi**)
t[n+1] ← Elt ; *// sentinelle rajoutée* i ← 1 ; **Tantque** (i ≤ n) **et alors** (t[i] ≠ Elt) **faire** i ← i+1 **finTant;** **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**	**pour** i ← 1 **jusquà** n **faire** **Sortirsi** t[i] = Elt **fpour**; **si** i ≤ n **alors** rang ← i **sinon** rang ← -1 **Fsi**

3.2 Recherche dans un tableau trié

Spécifications d'un algorithme séquentiel

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**) ***On peut reprendre sans changement les versions de l'algorithme de recherche séquentielle précédent travaillant sur un tableau non trié.***
- Complexité moyenne en O(n)

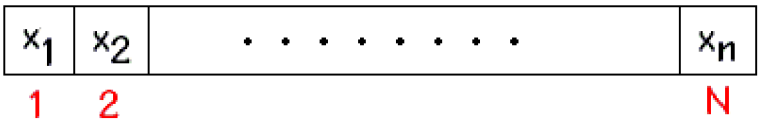
On peut aussi utiliser le fait que le dernier élément du tableau est **le plus grand élément** et s'en servir comme une sorte de **sentinelle**. Ci-dessous deux versions utilisant cette dernière remarque.

Version **Tantque**	Version **pour**
si t[n] < Elt **alors** rang ← -1 **sinon** i ← 1 ; **Tantque** t[i] < Elt **faire** i ← i+1; **finTant;** **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi** **Fsi**	**si** t[n] < Elt **alors** rang ← -1 **sinon** **pour** i ← 1 **jusquà** n-1 **faire** **Sortirsi** t[i] ≥ Elt *// sortie de la boucle* **fpour**; **si** t[i] = Elt **alors** rang ← i **sinon** rang ← -1 **Fsi** **Fsi**

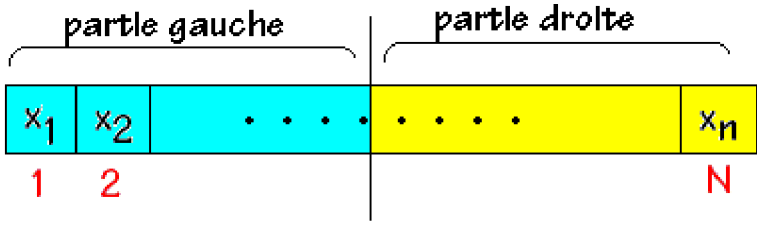
Spécifications d'un algorithme dichotomique

- Soit **t** un tableau d'entiers de **1..n** éléments rangés par ordre croissant par exemple.
- On recherche le rang (la place) de l'élément **Elt** dans ce tableau. L'algorithme renvoie le rang (la valeur -1 est renvoyée lorsque l'élément **Elt** n'est pas présent dans le tableau **t**). **Au lieu de rechercher séquentiellement du premier jusqu'au dernier, on compare l'élément Elt à chercher au contenu du milieu du tableau. Si c'est le même, on retourne le rang du milieu, sinon l'on recommence sur la première moitié (ou la deuxième) si l'élément recherché est plus petit (ou plus grand) que le contenu du milieu du tableau.**
- Complexité moyenne en O(log(n))

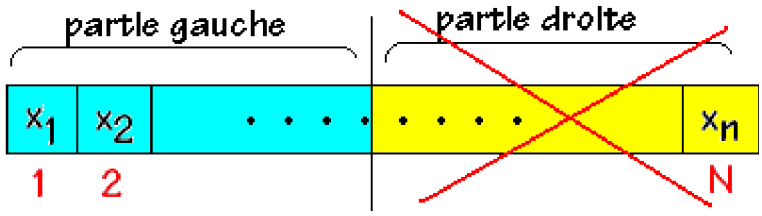
Soit un tableau au départ contenant les éléments (x_1, x_2, ... , x_n)



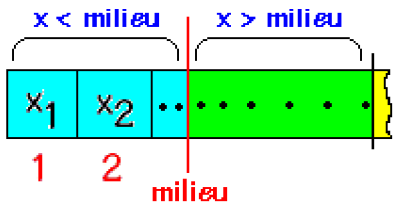
On recherche la présence de l'élément x dans ce tableau. On divise le tableau en 2 parties égales :



Soit x est inférieur à l' élément milieu et s'il est présent il ne peut être que dans la partie gauche, soit x est supérieur à l' élément milieu et s'il est présent il ne peut être que dans la partie droite. Supposons que x < milieu nous abandonnons la partie droite et nous recommençons la même division sur la partie gauche :



On divise à nouveau la partie gauche en deux parties égales avec un nouveau milieu :



Si x < milieu c'est la partie gauche qui est conservée sinon c'est la partie droite etc ...
Le principe de la division dichotomique aboutit à la fin à une dernière cellule dans laquelle soit x = milieu et x est présent dans la table, soit x ≠ milieu et x n'est pas présent dans la table.

Version itérative du corps de cet algorithme :

```
bas, milieu, haut, rang : entiers

bas ← 1;
haut ← N;
Rang ← -1;
repéter
  milieu ← (bas + haut) div 2;
  si x = t[milieu] alors
     Rang ← milieu
  sinon si t[milieu] < x alors
           bas ← milieu + 1
        sinon
           haut ← milieu-1
        fsi
  fsi
jusquà ( x = t[milieu] ) ou ( bas haut )
```

Voici en Delphi une procédure traduisant la version itérative de cet algorithme :

```
procedure dichoIter(x:Elmt; table:tableau; var rang:integer);
{recherche dichotomique par itération dans table rang =-1 si pas trouvé}
var
  milieu:1..max;
  g,d:0..max+1;
begin
 g := 1;
 d := max;
 rang := -1;
 while g <= d do
 begin
  milieu := (g+d) div 2;
  if x = table[milieu] then
  begin
    rang := milieu;
    exit
  end
  else
    if x < table[milieu] then d := milieu-1
    else g := milieu+1
 end;
end;{dichoIter}
```

Dans le cas de langage de programmation acceptant la récursivité (comme Delphi), il est possible d'écrire une version récursive de cet algorithme dichotomique.

Voici en Delphi une procédure traduisant la version récursive de cet algorithme :

```
procedure dichoRecur(x:Elmt;table: tableau; g,d:integer; var rang:integer);
{ recherche dichotomique récursive dans table
   rang =-1 si pas trouvé.
   g , d: 0..max+1 }
var
   milieu:1..max;
begin
  if g<= d then
  begin
   milieu := (g+d) div 2;
   if x = table[milieu] then  rang := milieu
   else
     if x < table[milieu] then // la partie gauche est conservée
        dichoRecur( x, table, g, milieu-1, rang )
     else // la  partie droite est conservée
        dichoRecur( x, table, milieu+1, d, rang )
  end
  else rang := -1
end;{dichoRecur}
```

Exercices chapitre 3

Des exercices traités avec leur solution détaillée

Algorithmes et leur traduction en langage de programmation

- **Somme de 2 vecteurs**
- **Fonctions booléennes**
- **Variantes sur la factorielle**
- **PGCD , PPCM de deux entiers**
- **Nombres premiers**
- **Nombres parfaits**
- **Suite : racine carrée - Newton**
- **Inversion d'un tableau**

Des algorithmes

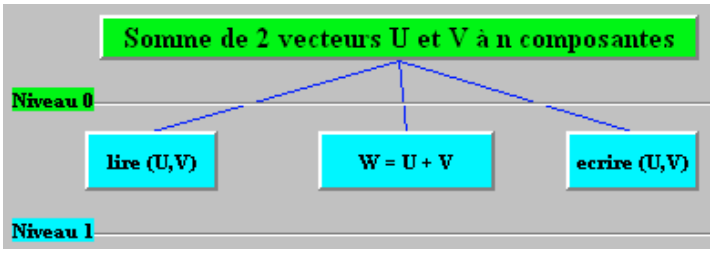
Somme de 2 vecteurs

Enoncé :

Programme simple d'utilisation des tableaux, on représente un vecteur de $\mathbf{Z}^n$ dans un tableau à un indice variant de 1 à n. Ecrire un programme LDFA additionnant 2 vecteurs de $\mathbf{Z}^n$ dont les composantes sont lues au clavier.

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé

Arbre



Algorithme

Niveau 1:

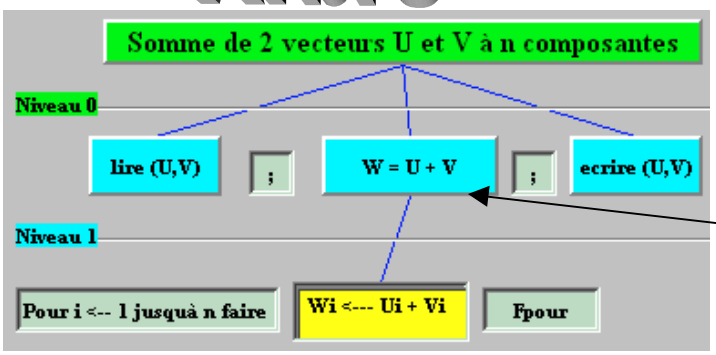
Algorithme Somme Vecteur
Entrée: U,V deux vecteurs de $\mathbf{Z}^n$
Sortie: W un vecteur de $\mathbf{Z}^n$
Local: i ∈ **N**

début
lire(U,V);
W ← U+V ;
ecrire (W)
FinSommeVecteur

Un vecteur de Z^n est défini par ses coordonnées : $U(U_1, U_2, \ldots, U_n)$; $V(V_1, V_2, \ldots, V_n)$ etc…

Action < **W = U+V** > :	Action < **Lire (U,V)** > :	Action < **Ecrire(W)** > :
$\forall i, 1 \le i \le n$ / $W_i = U_i + V_i$	$\forall i, 1 \le i \le n$ / lire U_i et V_i	$\forall i, 1 \le i \le n$ / ecrire W_i
Description :	*Description* :	*Description* :
Pour i ← 1 **jusqà** n **Faire** W_i ← $U_i + V_i$ **Fpour**;	**Pour** i ← 1 **jusqà** n **Faire** lire(U_i , V_i) **Fpour**;	**Pour** i ← 1 **jusqà** n **Faire** ecrire(W_i) **Fpour**;

Arbre



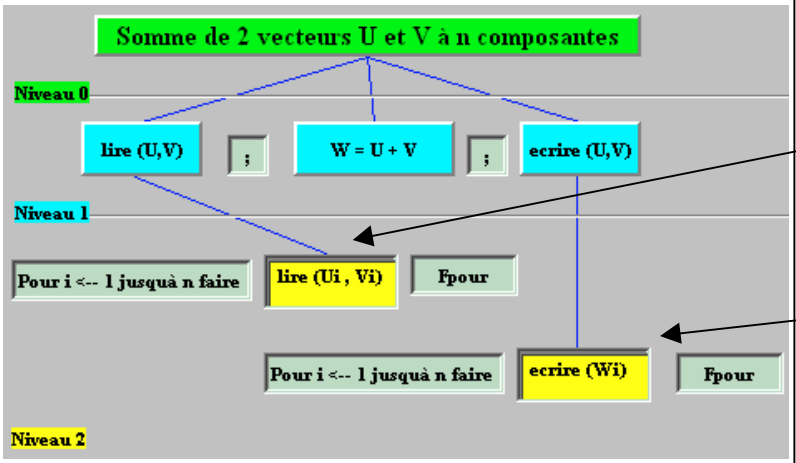
Algorithme

Niveau 2 :
début
< **lire(u,v)** >

Pour i ← 1 **jusqà** n **Faire**
W_i ← $U_i + V_i$
Fpour;

< **ecrire(w)** >
FinSommeVecteur

Arbre



Algorithme

début

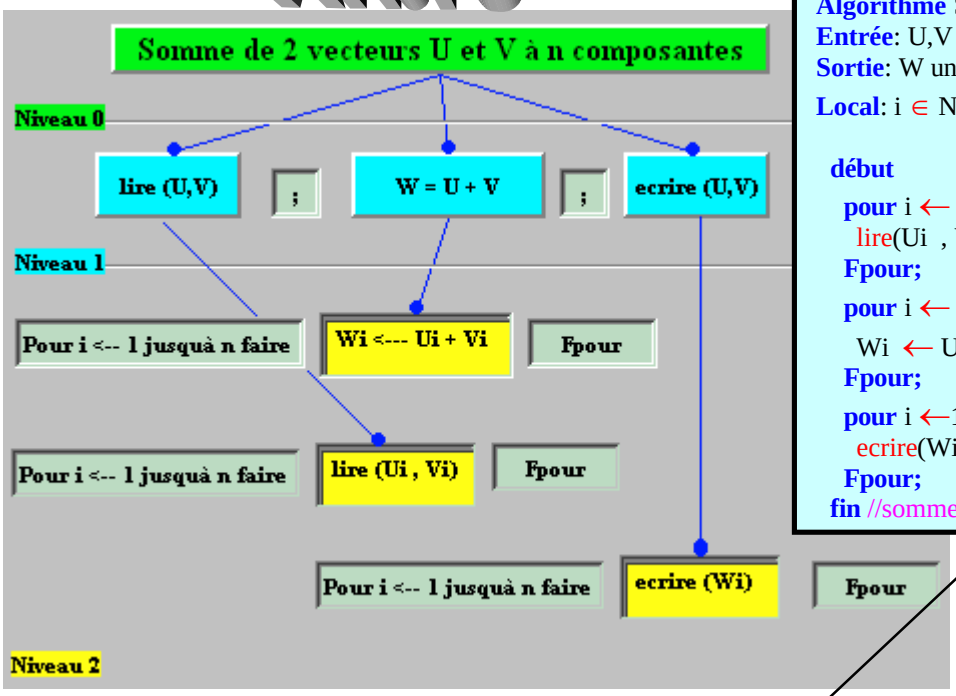
Pour i ← 1 **jusquà** n **Faire**
  lire(U_i , V_i)
Fpour;

W = U+V

Pour i ← 1 **jusquà** n **Faire**
  ecrire(W_i)
Fpour;

FinSommeVecteur

Arbre



Algorithme

Algorithme Somme Vecteur
Entrée: U,V deux vecteurs de Z^n
Sortie: W un vecteur de Z^n
Local: i ∈ N

début
  pour i ← 1 **jusquà** n **faire**
    lire(Ui , Vi)
  Fpour;
  pour i ← 1 **jusquà** n **faire**
    Wi ← Ui + Vi
  Fpour;
  pour i ←1 **jusquà** n **faire**
    ecrire(Wi)
  Fpour;
fin //sommevecteur

Delphi

```
program somme_vecteur;
const   n =3;
type
   intervalle=1..n;
   vecteur = array[intervalle] of integer;
var
   U,V,W:vecteur;
   i:intervalle;
begin
   for i:=1 to n do   readln (U[i],V[i]);
   for i:=1 to n do   W[i]:=U[i]+V[i];
   for i:=1 to n do   writeln(W[i]);
end.
```

Fonctions booléennes

Enoncé :

Programme simple d'écriture de deux fonctions de calcul des deux connecteurs logiques, le **et** booléen et le **ou** booléen par application des propriétés suivantes :

X **et** Faux = Faux X **et** Vrai = X	X **ou** Vrai = Vrai X **ou** Faux = X

Solution en Ldfa avec les traductions de chaque fonction en Delphi-Pascal et en Java

LDFA	**LDFA**
Algorithme LeEt // un Et logique **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens **debut** **si** x = **Faux alors** **resultat** ← **Faux** **sinon** **resultat** ← y **fsi** **fin** // LeEt	**Algorithme** LeOu // un Ou logique **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens **debut** **si** x =**Vrai alors** **resultat** ← **Vrai** **sinon** **resultat** ← y **fsi** **fin** // LeOu

DELPHI-Pascal
 	
function Et(x,y **:** Boolean)**:**Boolean**;**
begin
if x=faux **then**
result **:=** false
else
result **:=** y
end;
	DELPHI-Pascal
	
	function Ou(x,y **:** Boolean)**:**Boolean**;**
	begin
	if x=true **then**
	result **:=** true
	else
	result **:=** y
	end;

Java boolean Et **(** boolean x , boolean y **)** **{** **if (** x == false **)** **return** false **;** **else** **return** y **;** **}**	**Java** boolean Ou **(** boolean x , boolean y **)** **{** **if (** x == true **)** **return** true **;** **else** **return** y **;** **}**

On pourra utiliser les raccourcis d'écriture suivants pour un algorithme-fonction :

Algorithme LeEt **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens	**Algorithme** LeOu **Entrée**: x , y ∈ Booléens **Sortie**: **resultat** ∈ Booléens
	
Fonction LeEt (x , y: Booléens) : **resultat** Booléens	**Fonction** LeOu (x , y: Booléens) : **resultat** Booléens

Variantes sur la factorielle

Enoncé : Algorithme de calcul de la factorielle d'un entier **n! =n.(n-1)!** en utilisant différentes instructions de boucles.

Solution en Ldfa

par boucle tantque..ftant

```
Algorithme Factor
  Entrée: n ∈ Entier*
  Sortie: fact ∈ Entier*
  Local: i ∈ N

début
 lire(n) ;
 fact ← 1;
 i ← 2 ;
 Tantque i <= n Faire
  fact ← fact * i ;
  i ← i + 1 ;
 Ftant;
 Ecrire (n ,! = , fact );
Fin // Factor
```

par boucle pour…jusquà

```
Algorithme Factor
   Entrée: n ∈ Entier*
   Sortie:   fact ∈ Entier*
   Local:    i ∈ N

début
 lire(n) ;
 fact ← 1;
 Pour i ← 2 jusquà n Faire
  fact ← fact*i ;
 Fpour;
 Ecrire (n ,! = , fact );
Fin // Factor
```

par boucle repeter.... jusqua

```
Algorithme Factor
Entrée: n ∈ Entier*
Sortie: fact ∈ Entier*
Local: i ∈ Entier
début
  lire(n) ;
  fact ← 1;
  i ← 2 ;
  Repeter
     fact ← fact * i ;
     i ← i + 1 ;
  jusquà i > n ;
  ecrire(n ,! = , fact );
fin // Factor
```

par récursivité à partir de la formule : n! =n.(n-1)!

```
Algorithme fact_recur

utilise  Fonction fact
 debut
  ecrire ('5! =',fact(5))
 fin // fact_recur


Fonction fact (n:entier) : resultat entier
debut
    si n = 0 alors   resultat ← 1
    sinon  resultat ← fact (n - 1) * n
    fsi
fin // fact
```

```
program Factor;
 var n , fact , i :integer ;
 begin
    readln(n);
    fact:=1;
    i:=2;
    repeat
       fact:=fact*i;
       i:=i+1
    until i > n ;
    writeln(n ,'! = ', fact)
 end.
```


```
program Factor;
var n:integer;

 function fact  (n : integer ) : integer ;
  begin
     if n=0 then   result:=1
     else   result:=fact(n-1)*n
  end;// fact

begin
 readln(n);
 writeln(n ,'! = ', fact(n));
end.
```

PGCD , PPCM de deux entiers

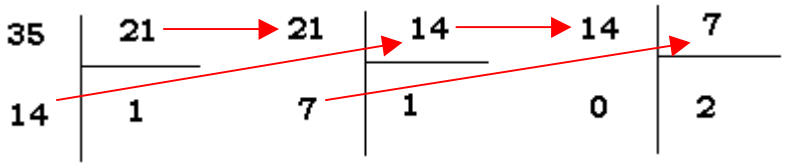
Enoncé : Ecrire des programmes LDFA donnant le pgcd et le ppcm de 2 entiers naturels.

Le pgcd de 2 entiers non nuls est le plus grand diviseur commun à ces deux entiers.
Exemple : 35 et 21 ont pour pgcd 7, car 35 = 7*5 et 21 = 7*3,

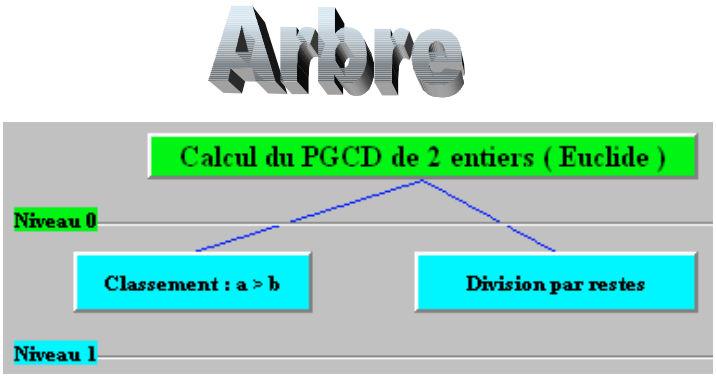
Le ppcm de 2 entiers non nuls est le plus petit commun multiple à ces deux entiers.
Exemple : 35 et 21 ont pour ppcm 105, car 105 =3*35 et 105=5*21.

Solution du pgcd faisant apparaître les niveaux de décomposition et l'algorithme associé

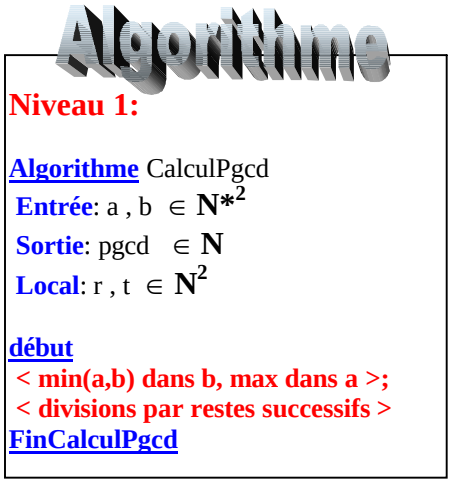
La méthode employée pour évaluer le pgcd de 2 entiers, se dénomme l'algorithme d'Euclide ou encore la division par les restes successifs. Soit le calcul du pgcd de 35 et 21 : on divise 35 par 21, puis 21 par le reste 14, puis 14 par le nouveau reste 7etc…Le processus s'arrête lorsque le dernier reste est nul, le pgcd est alors le précédent reste non nul



Le dernier reste non nul étant 7, le pgcd de 35 et 21 est donc 7. D'une manière générale, pour calculer le pgcd de deux entiers a et b nous divisons le plus grand par le plus petit, chacun de ces deux entiers est représenté par une variable a $\in$ **N** et b $\in$ **N**, nous classons systématiquement les contenus de ces variables en mettant dans **a** le plus grand des deux entiers et dans **b** le plus petit des deux.



Deux actions sont utilisées pour calculer le pgcd de 2 entiers.

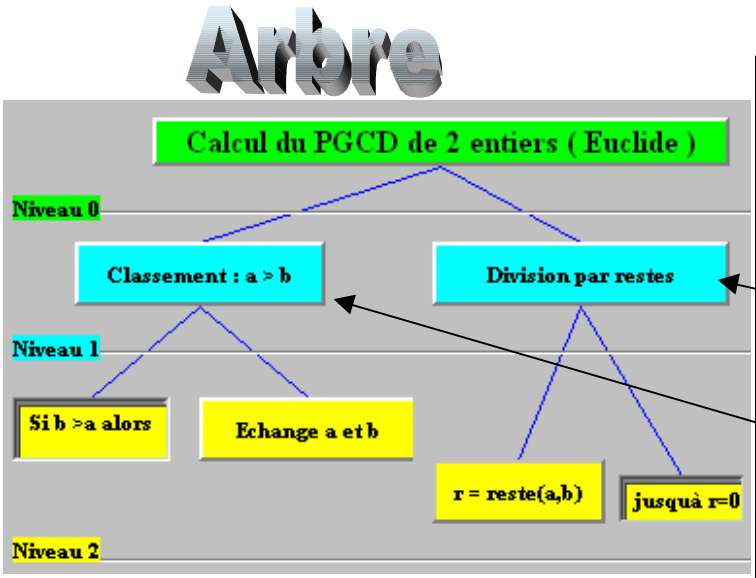


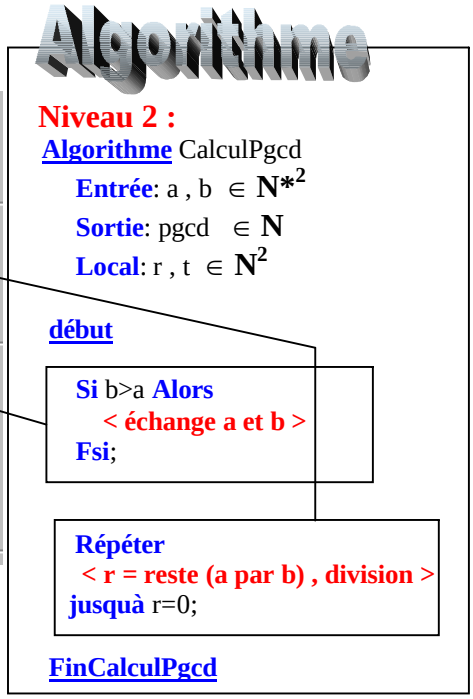
Niveau 1:

Algorithme CalculPgcd
Entrée: a , b $\in$ **N*²**
Sortie: pgcd $\in$ **N**
Local: r , t $\in$ **N²**

début
< min(a,b) dans b, max dans a >;
< divisions par restes successifs >
FinCalculPgcd

Action **< min(a,b) dans b, max dans a >:**	Action **< divisions par restes successifs >:**
Description :	*Description* :
Si b>a **Alors** **< échange a et b >** **Fsi**;	**Répéter** **< r = reste (a par b) , division >** **jusqà** r=0; pgcd ← a;





Action **< échange a et b >**

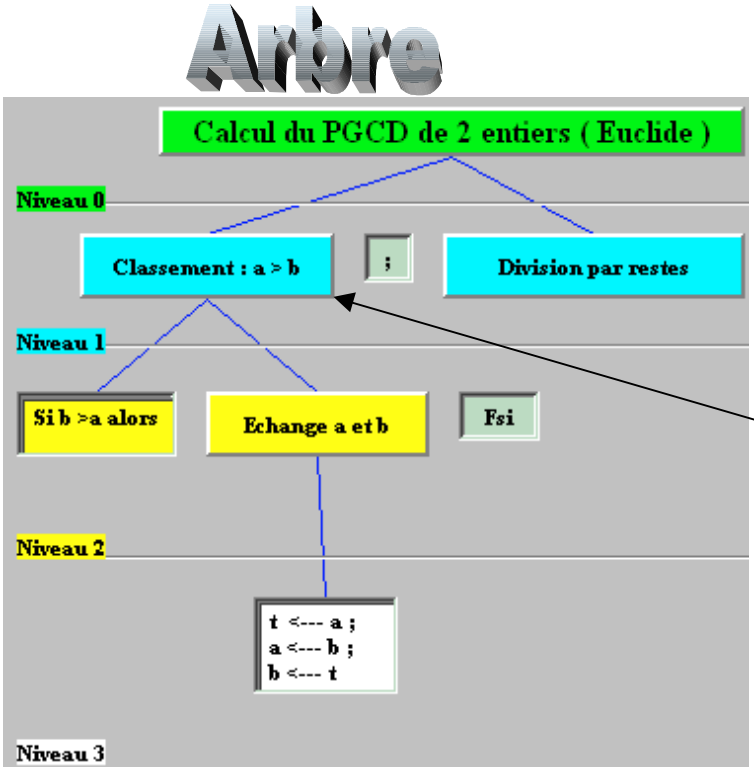
Description :

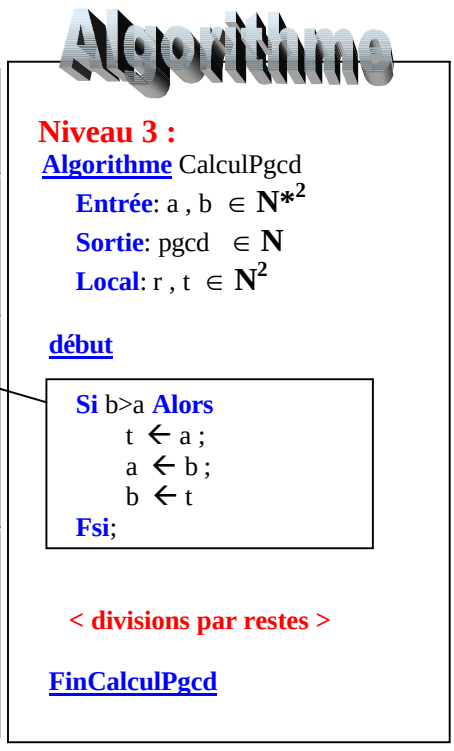
t ← a ;
a ← b ;
b ← t

Action **< r = reste (a par b) , division >**

Description :

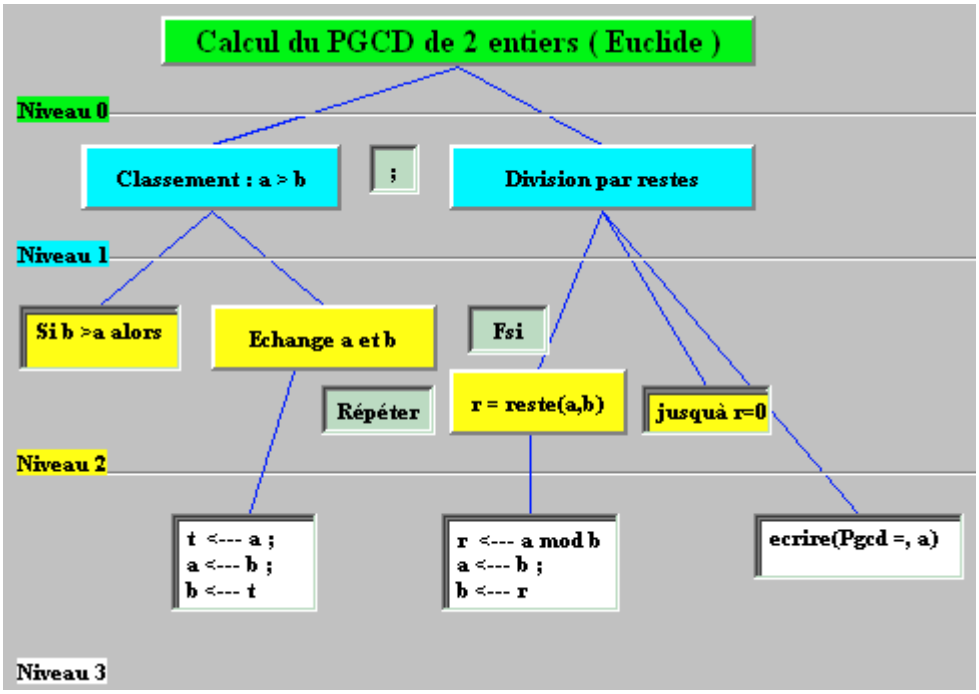
r ← a mod b ;
a ← b ;
b ← r





Ce qui donne finalement en développant les branches de l'arbre jusqu'au niveau 3 :

Arbre



Algorithme

```
Algorithme CalculPgcd
    Entrée: a ,b  ∈ N*²
    Sortie: pgcd  ∈ N
    Local: r , t  ∈ N²

début
  lire(a,b);
  Si b>a Alors
      t ← a ;
      a ← b ;
      b ← t
  Fsi;
  Répéter
      r ← a mod b ;
      a ← b ;
      b ← r
  jusquà r=0;
  pgcd ← a;
  ecrire(pgcd)
FinCalculPgcd
```

Delphi

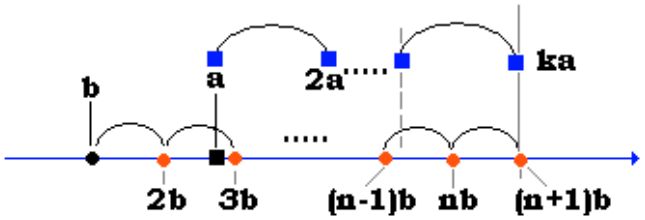
```
program calcul_Pgcd;
var
  a , b : integer;
  pgcd, r, t : integer;

begin
   readln(a,b);
   if b>a then
   begin
       t := a ;
       a := b ;
       b := t
   end;
   repeat
       r := a mod b ;
       a := b ;
       b := r
   until r=0;
   pgcd:= a;
   writeln(pgcd)
end.
```

Solution du calcul du ppcm de deux entiers sous forme d'un algorithme-fonction

La méthode employée pour évaluer le ppcm de 2 entiers a et b (b <a)est simple :

Nous évaluons tous les multiples de b par ordre croissant (2b, 3b, …) et nous arrêtons le calcul dès qu'un multiple est divisible par a (si a et b sont premiers entre eux le ppcm est égal à leur produit) :

Algorithme




```
Fonction ppcm (a,b:entier) résultat entier;
local : n , p ∈ entier

debut
   p ← b;
   n ← 1;
   tantque (n <= a) et (p Mod a<>0) faire
        p ← b * n;
        n ← n + 1;
   ftant
   résultat ← p
Fin // ppcm
```

Delphi

```
program  calcul_ppcm;
var
 a, b, p:integer;

function ppcm (a,b:integer):integer;
var
 k,p:integer;
begin
   p:=b;
   k:=1;
   while (k<= a)and(p mod a<>0) do
   begin
      p:= b * k;
      k:= k + 1;
   end;
   result:=p
end;

begin
   a:= 125;
   b:= 45;
   p:= ppcm(a, b);
   writeln('Calcul du ppcm :');
   writeln('a=', a,' b=', b,' => ppcm=', p)
end.
```

Java

```
class ApplicationPpcm {
 public static void main(String[ ] args) {
  int a,b,p;
  a=125;
  b=45;
  p=ppcm(a,b);
  System.out.println("Calcul du ppcm :");
  System.out.println("a="+a+" b="+b+" => ppcm="+p);
}

    static int ppcm (int a , int b)
    {
        int n=1 , p=b;
        while((n <= a)&(p % a !=0))
        {
            p= b * n;
            n ++;
        }
        return p ;
    }
/* Autre version avec un for sans aucun corps. Le code est
    plus compact, mais il est moins lisible !
*/
    static int ppcm (int a , int b)
    {
        int p=b;
        for(int n=1; (n<=a)&(p%a!=0); n++,p=b*n);
         return p ;
    }

}
```

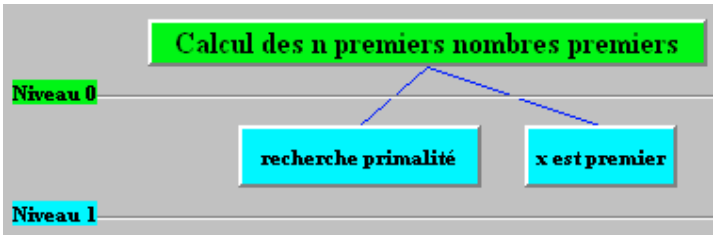
Nombres premiers

Enoncé : Un nombre entier est premier s'il n'est divisible que par 1 et par lui-même.
Exemple : 17 , 19, 31 sont des nombres premiers.

Ecrire un programme LDFA donnant les **n** premiers nombres premiers.

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé

Arbre



Deux actions sont utilisées pour calculer les nombres premiers, elles correspondent chacune à une branche de l'arbre.

Algorithme

Niveau 1:

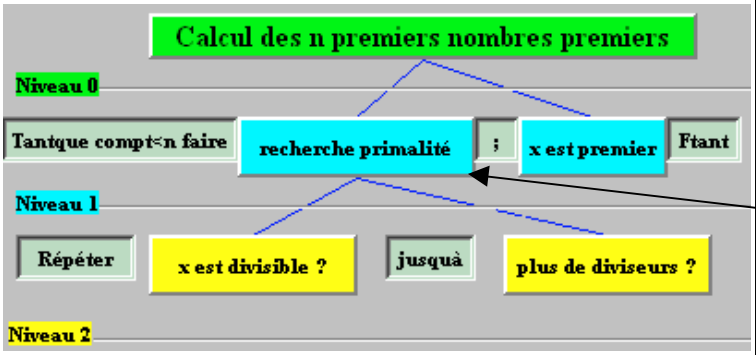
Algorithme Premier
Entrée: n ∈ **N**
Sortie: x ∈ **N**
Local: Est_premier ∈ {**Vrai** , **Faux**}
Divis , compt ∈ $\mathbf{N}^2$

début
lire(n);
Tantque(compt < n) **Faire**
< recherche primalité de x >;
< comptage si x est premier >
Ftant
FinPremier

Action **< recherche primalité de x >**	Action **< comptage si x est premier >**
Description :	*Description* :
Répéter **< x est-il divisible ?>** **jusquà < non premier, ou plus de diviseurs>**	**Si** Est_premier =**Vrai Alors** **<on ecrit x et on le compte>** **Fsi**; **<on passe au x suivant>**

Etude de la branche gauche de l'arbre au niveau 2 :

Arbre



Algorithme

Niveau 2 :

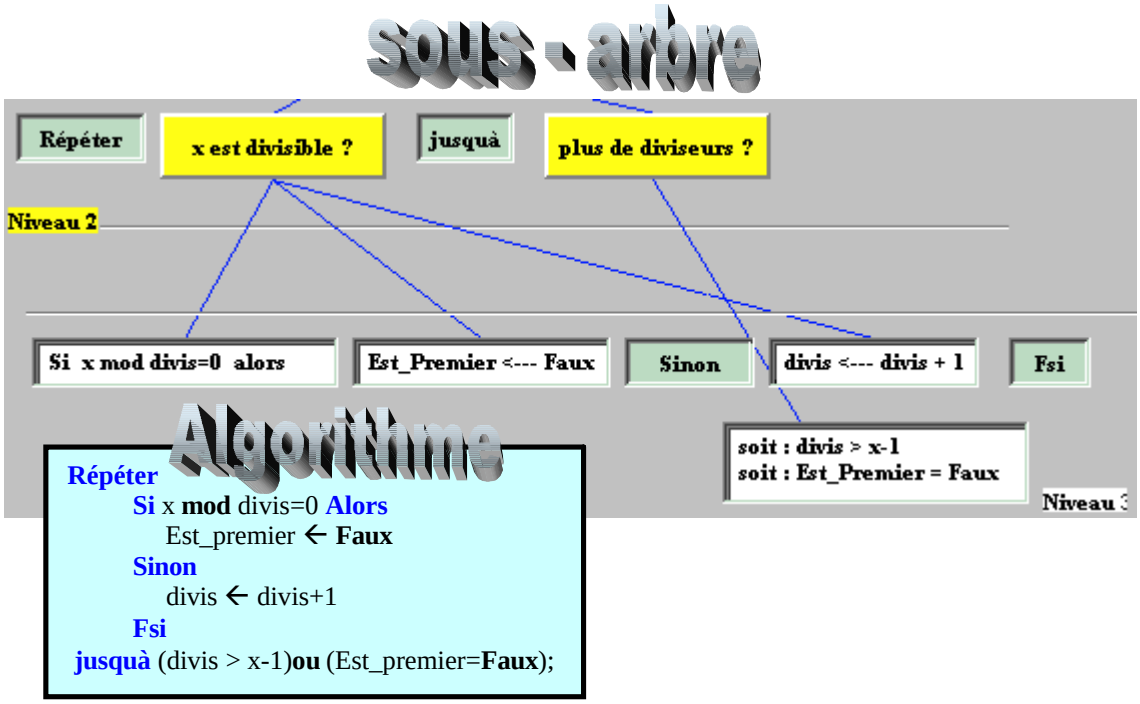
début
lire(n);
Tantque (compt < n) **Faire**

Répéter
< x est-il divisible ?>
jusquà < plus de diviseurs>

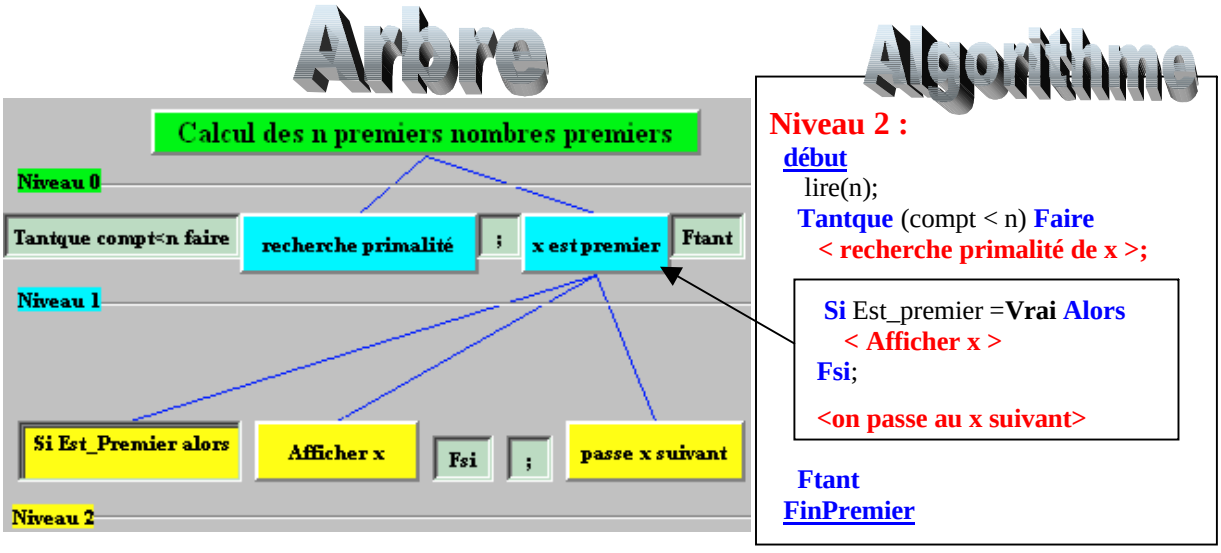
< comptage si x est premier >
Ftant
FinPremier

Etude de la branche gauche de l'arbre au niveau 3 :

Action < **x est-il divisible ?**>	Action < **non premier, ou plus de diviseurs**>
Description : **Si** x mod divis=0 **Alors** Est_premier ← **Faux** **Sinon** divis ← divis+1 **Fsi**	*Description* : (divis > n-1) **ou** (Est_premier=**Faux**)



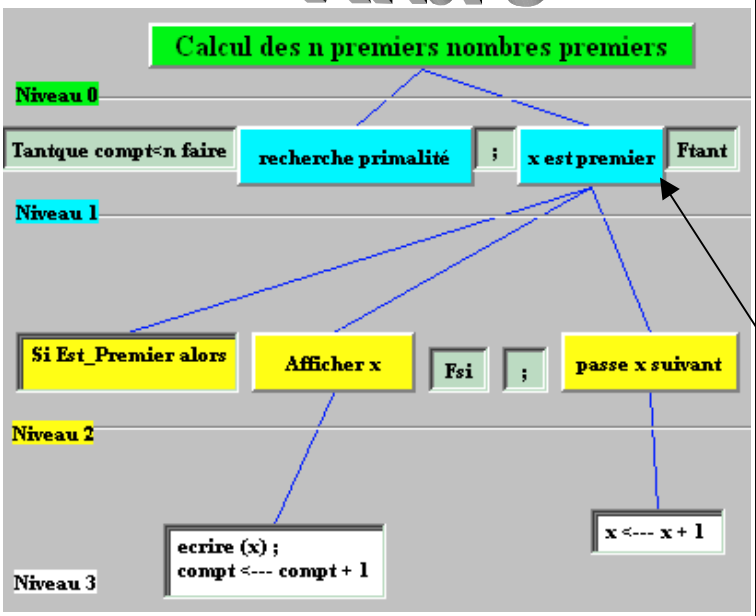
Etude de la branche droite de l'arbre au niveau 2 :



Etude de la branche droite de l'arbre au niveau 3 :

Action < **Afficher x** >	Action **<on passe au x suivant>**
Description : ecrire(x); compt ← compt+1	*Description* : x ← x+1

Arbre



Algorithme

Niveau 3 :

Algorithme Premier
Entrée: n ∈ **N**
Sortie: x ∈ **N**
Local: Est_premier ∈ {**Vrai** , **Faux**}
Divis , compt ∈ $\mathbf{N}^2$
début
lire(n);
Tantque (compt < n) **Faire**

< recherche primalité de x >;

Si Est_premier =**Vrai** **Alors**
ecrire(x);
compt ← compt+1
Fsi;
x ← x+1

Ftant
FinPremier

Version finale complète de l'algorithme → **sa traduction en Delphi**

Algorithme Premier
Entrée: n ∈ **N**
Sortie: x ∈ **N**
Local: Est_premier ∈ {**Vrai** , **Faux**}
Divis , compt ∈ $\mathbf{N}^2$
début
lire(n);
compt ← 1;
ecrire(2);
x ← 3;
Tantque(compt < n) **Faire**
divis ← 2;
Est_premier ← **Vrai**;
Répéter
Si x mod divis=0 **Alors** Est_premier ← **Faux**
Sinon divis ← divis+1
Fsi
jusqà (divis > x-1)**ou** (Est_premier=**Faux**);
Si Est_premier =**Vrai** **Alors**
ecrire(x);
compt ← compt+1
Fsi;
x ← x+1
Ftant
FinPremier

Algorithme

Delphi

```
program Premier;
 var
  n,nbr,divis,compt:integer;
  Est_premier:boolean;
 begin
  write('Combien de nombres premiers :  ');
  readln(n);
  compt:=1;
  writeln(2);
  nbr:= 3;
  while (compt < n) do
  begin
   divis:= 2;
   Est_premier:= true;
   repeat
     if nbr mod divis=0 then    Est_premier:= false
     else     divis:= divis+1
   until   (divis > nbr div 2)or (est_premier=false);
   if Est_premier=true then
   begin
    writeln(nbr);
    compt:= compt+1
   end;
   nbr:= nbr+1
  end
 end.
```

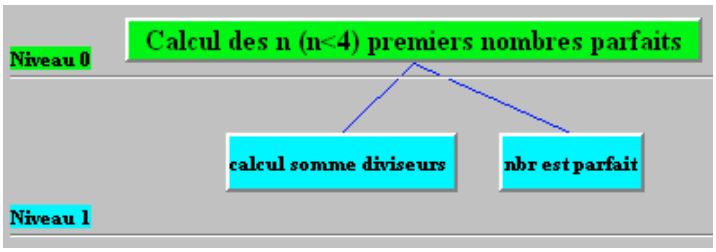
Nombres parfaits

Enoncé : Un nombre est dit parfait s'il est égal à la somme de ses diviseurs 1 compris.
Exemple : 6 = 1+2+3 , est parfait

Ecrire un programme LDFA donnant les **n** premiers nombres parfaits.

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé

Arbre



Deux actions sont utilisées pour calculer les nombres parfaits, elles correspondent chacune à une branche de l'arbre.

Algorithme

Niveau 1:

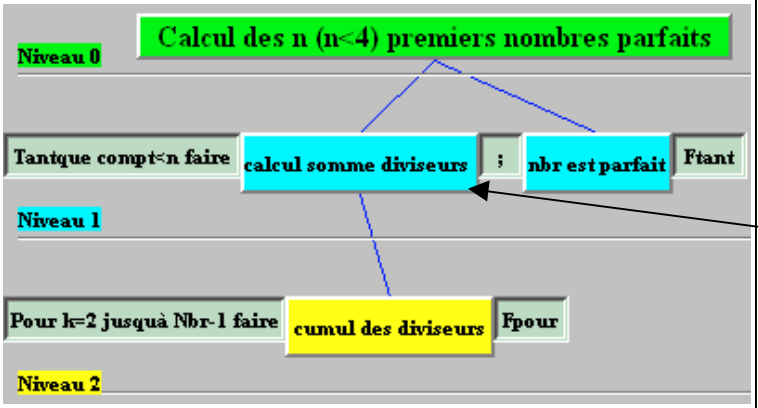
Algorithme Parfait
Entrée: n ∈ **N**
Sortie: nbr ∈ **N**
Local: som, k, compt ∈ $\mathbf{N}^3$

début
Tantque (compt < n) **Faire**
< somme des diviseurs de nbr >;
< nbr est parfait >
Ftant
FinParfait

Action **< somme des diviseurs de nbr >**	Action **< nbr est parfait >**
Description : *calcul de la somme des diviseurs du nombre :* ***nbr***	*Description* : *lorsque le nombre «* ***nbr*** *» est reconnu comme parfait, il doit être compté, puis affiché à l'écran*
Pour k ← 2 **jusquà** nbr-1 **Faire** **< cumul, si k divise nbr >** **Fpour**	**< nbr est parfait si nbr = som >** **< comptage >**

Etude de la branche gauche de l'arbre au niveau 2 :

Arbre



Algorithme

Niveau 2 :

Début

Tantque (compt < n) **Faire**

pour k ← 2 **jusquà** nbr-1 **Faire**
< cumul des diviseurs >
Fpour;

< nbr est parfait si nbr=som >
< comptage >
Ftant
FinParfait

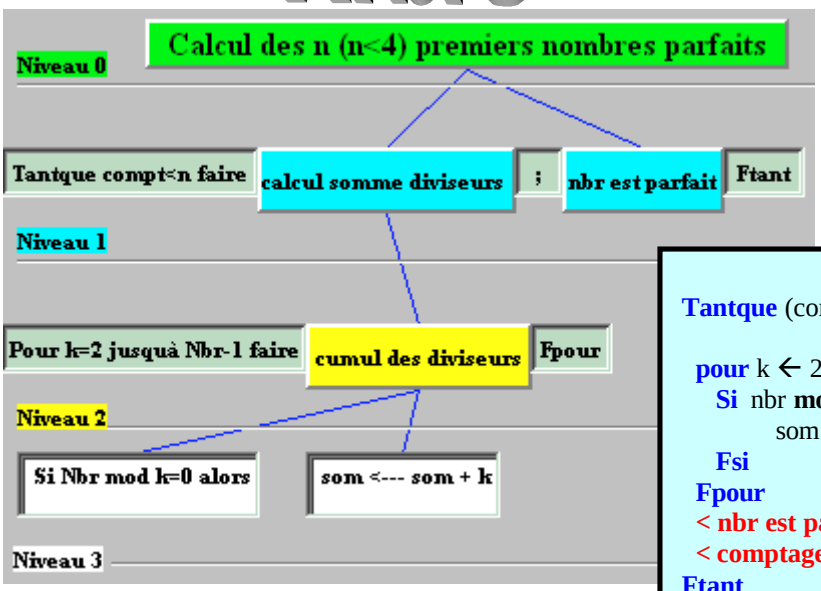
Etude de la branche gauche de l'arbre au niveau 3 :

Action < **cumul des diviseurs, (si k divise nbr)** >

*on cumule **k** dans la variable **som** (somme des diviseurs) du nombre **nbr** lorsque **k** divise effectivement **nbr**.*

Si nbr **mod** k = 0 **Alors**
som ← som + k
Fsi

Arbre



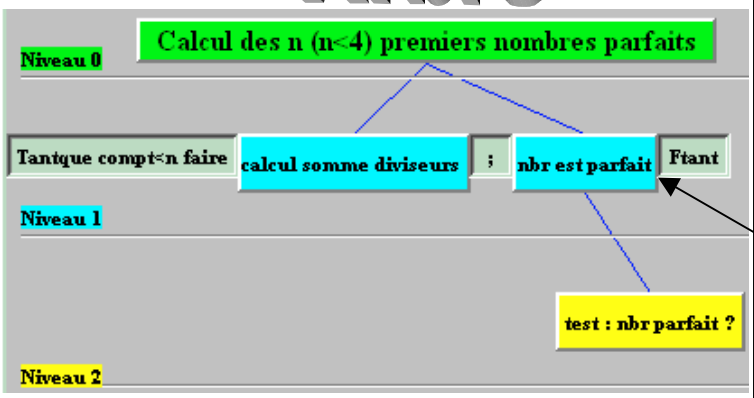
Algorithme

Tantque (compt < n) **Faire**

pour k ← 2 **jusquà** nbr-1 **Faire**
Si nbr **mod** k = 0 **Alors**
som ← som + k
Fsi
Fpour
< nbr est parfait si nbr=som >
< comptage >
Ftant

Etude de la branche droite de l'arbre au niveau 2 :

Arbre



Algorithme

Niveau 2 :
début
Tantque (compt < n) **Faire**

< somme des diviseurs de nbr >;

< test nbr parfait ? >
< comptage >

Ftant

FinPremier

Action **< test nbr parfait ? >**

Description :
*le nombre **nbr** est parfait s'il est égal à la somme calculée :*

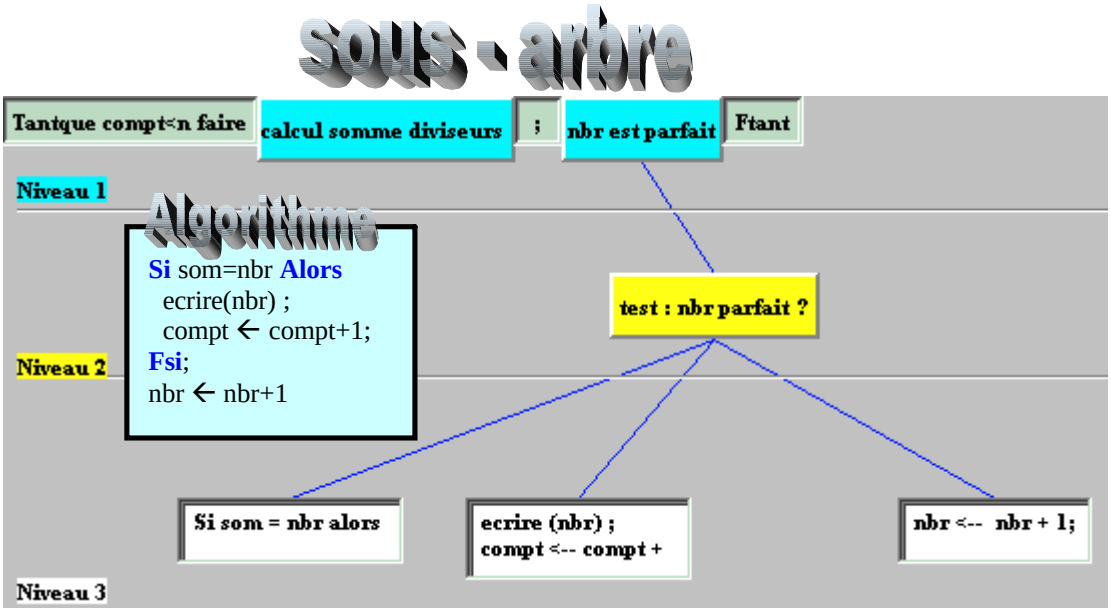
Si som=nbr **Alors**
ecrire(nbr) ;
compt ← compt+1;
Fsi;

Action **< comptage >**

Description :
Puis l'on passe au nombre suivant

nbr ← nbr+1

Etude de la branche droite de l'arbre au niveau 3 :



Version finale complète de l'algorithme → **sa traduction en Delphi**

Algorithme Parfait
Entrée: n ∈ **N**
Sortie: nbr ∈ **N**
Local: som, k, compt ∈ $\mathbf{N}^3$

début
lire(n);
compt ← 0;
nbr ← 2;
Tantque (compt < n) **Faire**
som ← 1;
Pour k ← 2 **jusquà** nbr-1 **Faire**
Si nbr mod k = 0 **Alors**
som ← som + k
Fsi
Fpour ;
Si som=nbr **Alors**
ecrire(nbr) ;
compt ← compt+1;
Fsi;
nbr ← nbr+1
Ftant

FinParfait

```
program Parfait;
 var
  n, nbr : integer;
  som, k, compt : integer;

 begin
   readln(n);
   compt := 0;
   nbr := 2;
   while (compt < n) do
   begin
     som:= 1;
     for k:= 2 to nbr-1 do
      if nbr mod k=0 then
        som := som + k ;
     if som=nbr then
     begin
       writeln(nbr);
       compt:= compt+1;
     end ;
     nbr:= nbr+1
   end
end.
```

Suite : racine carrée - Newton

Enoncé : Etude d'une suite convergente de la forme $U_n = f(U_{n-1})$

Ecrire un programme LDFA proposant une étude simple de la suite U_n suivante (méthode de Newton) :

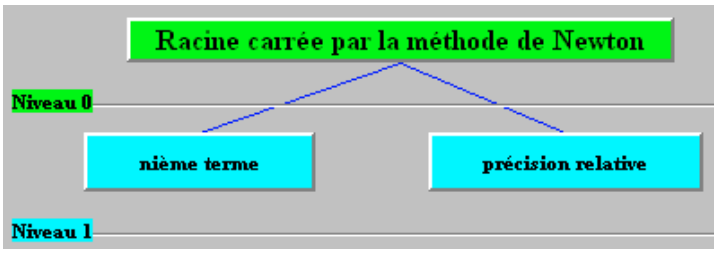
$$U_n = 1/2(U_{n-1} + X/ U_{n-1}) \quad X>0$$

La suite U_n converge vers le nombre $\sqrt{X}$ (la racine carré de X), le programme doit effectuer :

1° le calcul du terme de rang n donné par l'utilisateur,
2° le calcul jusqu'à une précision relative Epsilon fixée

Solution faisant apparaître les niveaux de décomposition et l'algorithme associé

Arbre



Deux actions sont utilisées pour effectuer les calculs demandés, elles correspondent chacune à une branche de l'arbre.

Algorithme

Niveau 1:

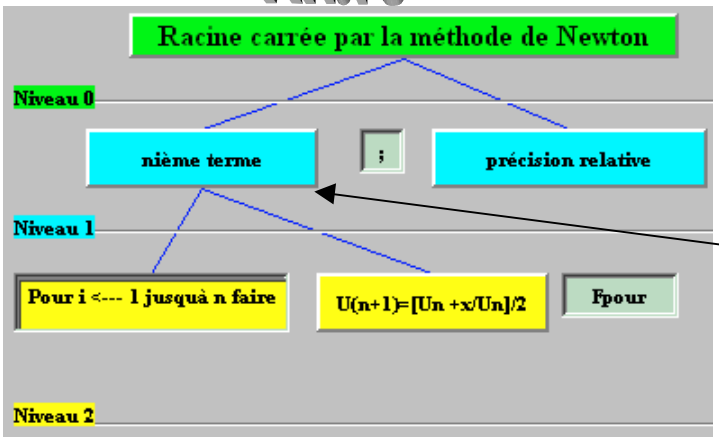
Algorithme Newton
Entrée: n ∈ **N***
x ∈ **R***
ε ∈ **R** (ε ∈ [0,1])
Sortie: u ∈ **R**
Local: v ∈ **R**
i ∈ **N**
début

<calcul du terme de rang n>;
<calcul de la limite à la précision ε >

FinNewton

Action **<calcul du terme de rang n>**	Action **<calcul de la limite à la précision ε >**
Description : **Pour** i ← 1 **jusqà** n **Faire** **< u(n+1) ← [u(n)+x/u(n)]/2 >** **Fpour**;	*Description* : **Répéter** **<* u(n+1)=[u(n)+x/u(n)]/2 , précision *>** **jusqà <* précison < ε *>**

Etude de la branche Arbre gauche de l'arbre au niveau 2 :



Algorithme

Niveau 2:

début
Eps ← 10^{-4};
n ←10;
lire(x);
Pour i ← 1 **jusqà** n **Faire**
< u(n+1) ← [u(n)+x/u(n)]/2 >
Fpour;
ecrire(u);

<calcul de la limite à la précision ε >
FinNewton

Etude de la branche gauche de l'arbre au niveau 3 :

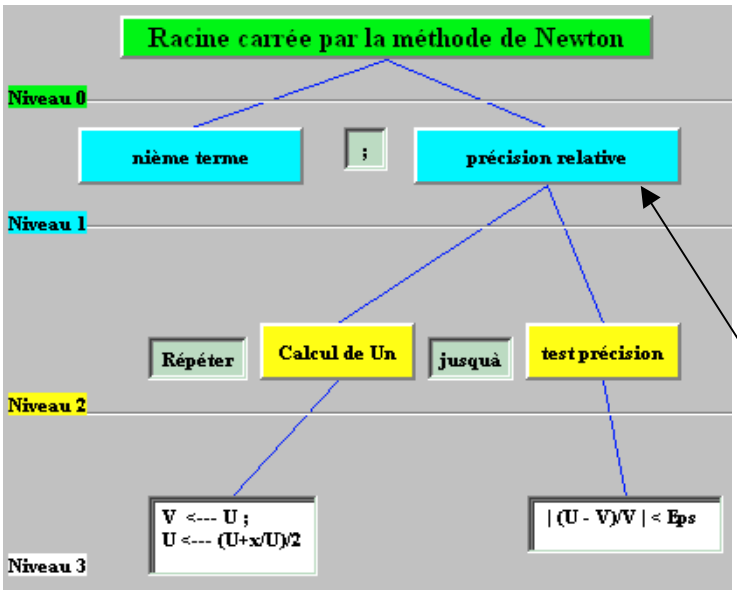
```
Action  < u(n+1) ← [u(n)+x/u(n) ]/2  >
    v ← u;
    u ←  (u + x/u)/2 ;
```

Algorithme

```
Niveau 3:
  début
    Eps ←  10^-4;
    n ←10;
    lire(x);
    Pour i ← 1 jusqà n Faire
        v ← u;
        u ←  (u + x/u)/2 ;
    Fpour;
    ecrire(u);
  <calcul de la limite à la précision ε >
FinNewton
```

Développement de la branche droite de l'arbre jusqu'au niveau 3 :

Arbre



Algorithme

```
Niveau 3:
début
  Eps ←  10^-4;
  n ←10;
  lire(x);
  Pour i ← 1 jusqà n Faire
      v ← u;
      u ←  (u + x/u)/2 ;
  Fpour;
  ecrire(u);
  Répéter
    v ← u;
    u ←  (u + x/u)/2 ;
  jusqà | (u-v)/v | < Eps;

   ecrire(u)

FinNewton
```

Version finale complète de l'algorithme → **sa traduction en Delphi**

```
Algorithme Newton
 Entrée:
    n ∈N* , x ∈ R* , ε ∈ R  (ε ∈ [0,1] )
 Sortie: u  ∈ R
 Local:  v  ∈ R , i  ∈ N

début
 Eps ←  10^-4;
 n ←10;
 lire(x);
 // calcul du terme de rang n donné:
 u ←  (1 + x)/2 ;
 Pour i ← 1 jusqà n Faire
  u ←  (u + x/u)/2 ;
 Fpour;
 ecrire(u);
```

Algorithme

Delphi

```
program Newton;

 const
   Eps=1E-4;
   n=10;
 var
   u,v,x : real ;
   i : integer ;

 begin
    readln(x);
   // calcul du terme de rang n donné:
    u:=(1+x)/2;
  for i:= 1 to n do
     u:= (u + x/u)/2 ;
  writeln(u);
```

Algorithme *suite* **Delphi**

```
//calcul jusqu'à une précision Epsilon fixée

u ← (1 + x)/2 ; { réinitialisation }
Répéter
 v ← u;
 u ← (u + x/u)/2 ;
jusquà | (u-v)/v | < Eps;
ecrire(u)

FinNewton
```

```
//calcul jusqu'à une précision Epsilon fixée

u:=(1+x)/2;
 repeat
   v:= u;
   u:= (u+x/u)/2
 until   abs((u-v)/v ) < eps;
 writeln(u)

 end.
```

Java

```
import Readln;

class Newton
{
 public static void main (String [ ] arg) {

  final double Eps=1E-4;
  int n=10;
  double u,v,x;
  x = Readln.undouble();
  // calcul du terme de rang n donné:
  u = (1+x)/2;
  for(int i=1;i<=n;i++)
   u = (u + x/u)/2 ;
  System.out.println("1°) après "+n+" termes: "+u);

   //calcul jusqu'à une précision Epsilon fixée:
  u=(1+x)/2;
  do
   {
    v = u;
    u = (u+x/u)/2;
   }
  while(Math.abs((u-v)/v ) >= Eps);
  System.out.println("2°) à la précision "+Eps+": "+u);
 }
}
```

Inversion d'un tableau

Enoncé : Autre programme simple d'utilisation des tableaux, écrire un programme LDFA inversant le contenu d'un tableau à n éléments entiers déjà rempli, on écrira le contenu du tableau avant inversion, puis son contenu après inversion.

Solution en Ldfa avec les traductions de chaque fonction en Delphi-Pascal et en Java

Algorithme

```
Algorithme InverseTable
  Global: Table; vecteur de N^n
  Local: temp ∈ N , i ∈ N
         ( i ∈ [1 , Max] )
début
{remplissage aléatoire du tableau}

 Pour i ← 1 jusquà Max Faire
   ecrire (Table_i )
 Fpour;
 Pour i ← 1 jusquà Ent[Max/2] Faire
      Temp ← Table_i ;
      Table_i ← Table_Max - i + 1 ;
      Table_Max - i + 1 ← Temp
 Fpour;
 Pour i ← 1 jusquà Max Faire
   ecrire (Table_i )
 Fpour;

FinInverseTable
Ent[p] représente la partie entière de p
```

Delphi

```
program inverse_tableau;
 const
   Max=10;
 type
   intervalle=1..Max;
   Tableau=array[intervalle] of integer;
 var
   Table:Tableau;
   i:intervalle;
   Temp:integer;
 begin
  {remplissage aléatoire du tableau}
  for i:=1 to Max do
   Table[i]:=random(1000);

  {voir le contenu du tableau  avant opération :  }
  for i:=1 to Max do
   writeln('i=',i:2,':',Table[i]:4);

  for i:=1 to Max div 2 do
  begin
   Temp:=Table[i];
   Table[i]:=Table[Max-i+1];
   Table[Max-i+1]:=Temp
  end;
  writeln('-----------------');
  {voir le contenu du tableau  après opération :  }
  for i:=1 to Max do
   writeln('i=',i:2,':',Table[i]:4);
 end.
```

```
class InvTab{
 public static void main (String [ ] arg)   {
  final int Max=6;
  long[ ]table= new long[Max+1];
  //remplissage aléatoire du tableau
  for(int i=0;i<=Max;i++)
   table[i] = Math.round(Math.random( )*100);

  //voir le contenu du tableau avant opération
  for(int i=0;i<=Max;i++)
   System.out.println("table["+i+"] =  "+
table[i]);

   for(int i=0;i<=Max/2;i++)  {
     long Temp=table[i];
     table[i]=table[Max-i];
    table[Max-i]=Temp;
   }
 }
}
```