Secteur Tertiaire Informatique
Filière étude - développement

ALGORITHMIQUE

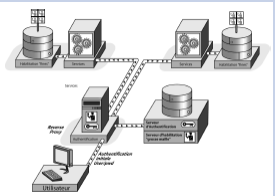
Support de formation

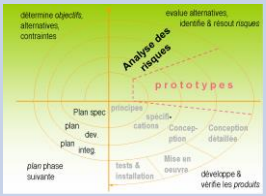
Accueil

Apprentissage

Période en entreprise

Evaluation





SOMMAIRE

Introduction :

- Avant de développer un composant informatique, il est parfois nécessaire de passer par une étape de résolution de problème. Cette étape s'appuie sur un formalisme qui per met l'analyse. Ce formalisme s'appelle l'algorithmique.
- De plus, lorsque l'on débute en programmation il est intéressant et souvent nécessaire de passer par cette étape afin de mieux maîtriser le codage du composant.

Les objectifs :

Vous serez capable, à l'issue de cette séquence :

- de décrire des algorithmes en pseudo-langage,

- de réaliser une analyse d'un problème de programmation avant son codage dans un langage informatique.

- de constituer un jeu d'essai unitaire (Le jeu d'essai d'un composant logiciel doit permettre de tester la structure et les traitements. Il permet dans un premier temps de tester "à la main" le pseudo-langage, puis après l'avoir entré en machine, de tester le composant logiciel en vérifiant que les résultats obtenus sont ceux que l'on attendait).

Notations :

Bibliographie : lecture et synthèse à effectuer sur les documents cités

Exercice de base : intégré dans le temps de formation moyen

Exercice supplémentaire : non compris dans la durée de formation moyenne

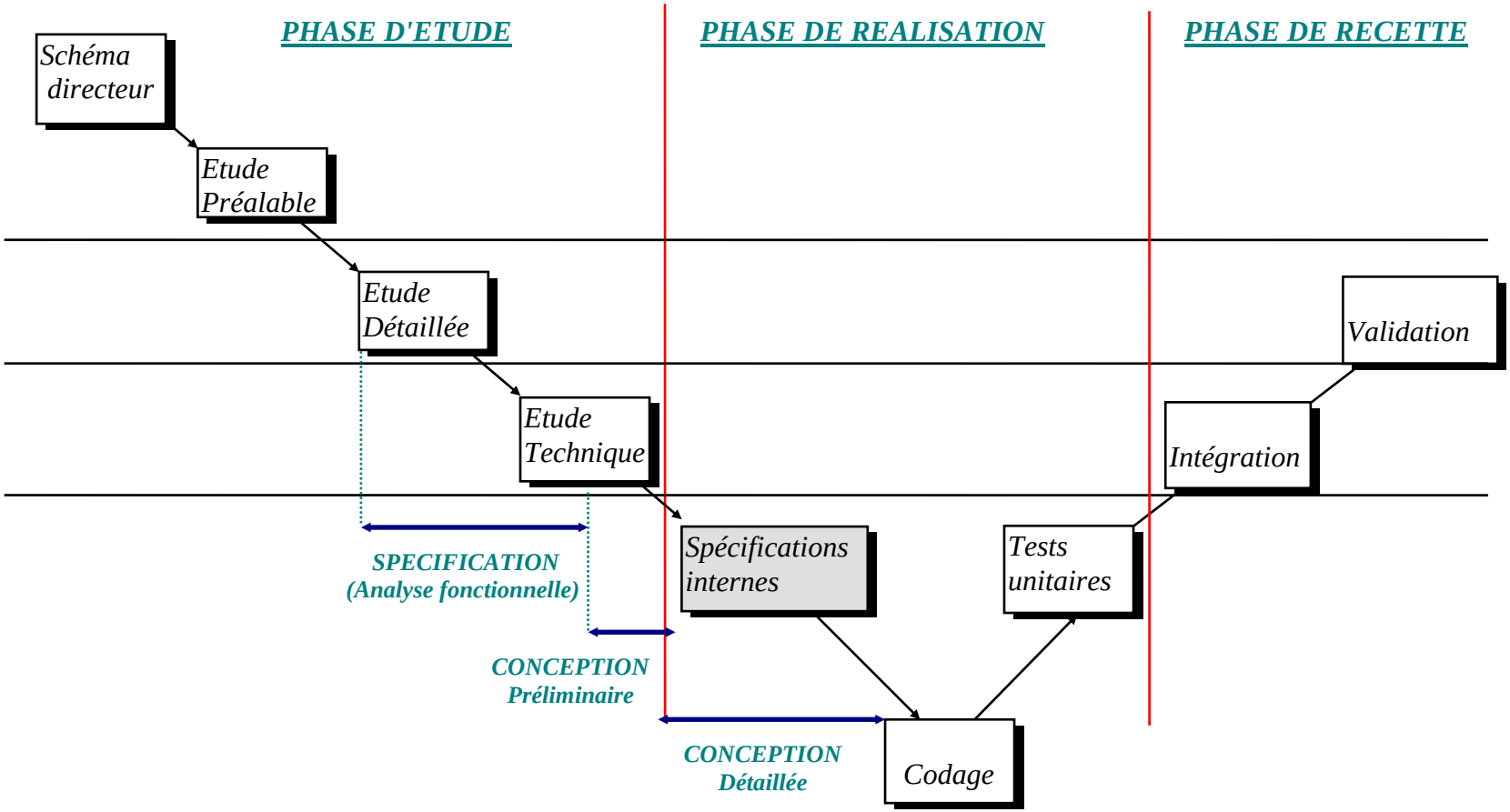
Les ressources :

- le présent support de formation ***« Algorithmique, Support de Formation »***, qui comprend un certain nombre d'exercices à effectuer,

- le support de cours ***« Algorithmique »***, qui détaille le langage algorithmique à utiliser.

1 - PRESENTATION GENERALE

1.1 Cycle de vie



1.2 Pourquoi une méthode de programmation

Les objectifs d'une méthode sont liés aux problèmes rencontrés dans le développement de logiciels depuis les débuts de l'informatique.

Les deux symptômes les plus évidents de cette crise du logiciel sont le coût de production et la qualité des produits finis. Le coût élevé du processus de programmation et la fiabilité médiocre du logiciel qui en découle sont en réalité deux aspects du même problème, étant donné que la faiblesse du rendement provient en partie du temps passé à corriger de précédentes erreurs.

Quelques chiffres ...

Le coût du logiciel occupe une part croissante du coût total d'un système informatique, et cette part dépasse maintenant celle du matériel.

Le coût de la maintenance constitue la part prépondérante du coût du logiciel (pouvant atteindre 70%, et s'étendant sur des années).

La part de l'écriture proprement dite, dans le développement du logiciel, ne représente qu'environ 20%, pour environ 30% d'analyse et de conception, et 50% de mise au point, tests et intégration.

Autres remarques

Une proportion considérable d'erreurs dans les programmes provient soit d'une mauvaise compréhension par le programmeur du problème tel qu'il a été posé, soit de spécifications défectueuses.

Il est donc primordial pour l'analyste programmeur de savoir définir, expliciter et valider la conception d'un programme (ou d'un composant logiciel), en cohérence avec le dossier de spécification fonctionnelle.

C'est ainsi que l'un des objectifs d'une méthode de programmation est de minimiser le nombre d'erreurs qui se produisent inévitablement et de rendre moins difficile la correction du code défectueux.

Un peu d'histoire

Quand au environ des années 50, il a fallu commencer à penser algorithme, on ne disposait que de ce qu'avait défini Türing, qui avait clairement défini ce qu'était un algorithme, mais avait omis de léguer la recette de fabrication : la programmation intuitive, au pas à pas, était née!

Ce fut, c'est et ce sera toujours passionnant, mieux qu'avec n'importe quelle méthode ou modèle. On parvient aux prouesses d'un minimum d'occupation mémoire ou d'un temps record d'exécution. On peut tout cela avec beaucoup d'astuce et de génie. Si le génie et l'astuce manquent, alors c'est la catastrophe : l'œuvre est un bricolage informe où l'expérience devient maîtresse, mais la réflexion scientifique absente. C'est l'impasse!

Au commencement des années 60, il y eut une première réaction importante contre la propagation des intuitions personnelles : la table de décision. Mais cette approche se révéla bien décevante :

- ce n'était que la concrétisation d'une forme d'algèbre assez réduite qu'est l'algèbre de Boole,
- la répétitivité, élément essentiel de l'algorithmique, est à peu près inexprimable avec une table de décision.

Vers la fin des années 60, alors que l'on ne croyait plus beaucoup aux tables de décision, sont apparus deux courants féconds :

- la méthode Warnier (LCP : Logique de Construction de Programmes),
- la programmation structurée.

Le premier s'appuyant sur les notions d'ensembles et de structures mettait tout un arsenal de règles et de lois au service d'une approche logique résolument globale. Le second mettait en lumière la nécessité d'écrire des programmes logiques, clairs et lisibles : en un mot, structurés.

1.3 Algorithme

Lorsque l'on a une tâche complexe à effectuer, il est nécessaire de la décomposer en étapes ou en opérations plus simples. La description de cette succession d'opérations, qui permet d'aboutir au résultat recherché, s'appelle un **algorithme**.

Tout algorithme est caractérisé par :

- un début et une fin,
- des étapes,
- des opérations à exécuter à l'intérieur de chaque étape,
- éventuellement, l'existence de conditions déterminant l'exécution ou non de certaines étapes.

La règle de calcul des racines d'une équation du second degré, une notice d'emploi d'un matériel, une recette de cuisine, un mode opératoire, sont des exemples d'algorithme.

Définition AFNOR : un algorithme est un ensemble de règles opératoires et de procédés définis en vue d'obtenir un résultat déterminé au moyen d'un nombre fini d'opérations.

Les deux notions de base de l'algorithmique et même de la programmation sont :

- l'alternative (ou choix - choix simple ou choix multiple)
- la répétitive (ou boucle).

1.4 Quelques généralités

Un programme se découpe en éléments. De manière classique on retrouve toujours : un début, un corps, une fin.

Chacun de ces éléments et en particulier le corps, peut, selon la complexité du programme se décomposer en un certain nombre de modules. Chacun de ces modules est lui-même composé d'un début, d'un corps et d'une fin.

La vie, le travail sont faits de tâches répétitives. L'ordinateur nous aide car, lorsque ces tâches sont "simples", et qu'il va très vite, il peut ***itérer*** de nombreuses opérations.

Ainsi, le corps d'un programme peut être une structure répétitive qui réalise un nombre contrôlé de fois la même opération, par exemple lire un enregistrement dans un fichier d'articles d'un stock.

L'expression d'un algorithme peut se faire de plusieurs manières.

On peut :

- **Ecrire une suite de courtes phrases claires et précises exprimant les actions et dans quelles conditions elles se réalisent. Ce procédé ne tient pas longtemps face à la complexité, et il n'est pas un standard de la profession.**

- **Exprimer de manière graphique, sous forme d'organigramme, la succession des opérations.**

- **Exprimer la même logique dans les conventions d'un pseudo-langage (ou pseudo code).**

2 - ALGORITHMES SIMPLES

A partir des spécifications d'une fonction informatique simple, nous devons être capable d'écrire comment réaliser l'application dans un formalisme commun à tous. Nous devons également prendre en compte les documents à réaliser pendant cette phase (définition de ce que l'on va faire, puis la réalisation), ainsi que les contraintes dues à la maintenance des applications informatiques.

Nous allons aborder les thèmes techniques suivants :

- Sensibilisation aux problèmes de l'algorithmique.
- Description d'algorithmes.
- Programmation par blocs, notion de procédure.
- Variables globales et locales.
- Vérification d'algorithmes et jeux d'essais.

2.1 Initiation à l'algorithmique

Nous devons prendre conscience de la difficulté que nous avons à exprimer de manière complète et non ambiguë la solution d'un problème donné.

Quand nous considérons un problème très simple, nous sommes dans la situation de résolution d'un problème dont l'analyse aurait déjà été faite (nous savons ce qu'il y a à faire, nous connaissons le « QUOI »).

Il nous faut alors exprimer comment nous y prendre pour résoudre le problème (nous allons exprimer le « COMMENT »).

Ce sont deux étapes bien distinctes, l'analyse précédant naturellement la conception. L'algorithme ainsi obtenu est un texte qui nous permettra de bâtir le programme compréhensible par la machine informatique.

Nous allons prendre conscience de la difficulté de formulation exacte de l'algorithme de réalisation d'un travail simple.

2.1.1 ANALYSE BIBLIOGRAPHIQUE :

Repérer la place de l'algorithme dans le cycle de vie du logiciel.

2.1.2 EXERCICES :

Nous voulons éplucher un nombre nécessaire et suffisant de pommes de terre. Nous prenons les pommes de terre, une à une, dans un panier. Ce panier peut être vide à un moment. Nous pouvons alors aller le remplir à la cave.

Nous savons qu'il y a assez de pommes de terre à la cave.

L'approche humaine d'un tel problème est globale, nous vous demandons d'en faire une approche détaillée, telle qu'en suivant une à une les recommandations données vous arriviez au résultat demandé quel qu'il soit. Vous vous exprimerez sous forme textuelle uniquement. Vous utiliserez les termes employés dans l'énoncé du problème sans plus entrer dans les détails (travail individuel d'une demi-heure maximum).

Nous exposerons et critiquerons de manière constructive quelques solutions, afin de vérifier si elles correspondent exactement au problème posé, quelque soient les cas de figure.

2.2 Formalisme de représentation algorithmique

Nous allons apprendre un formalisme de représentation d'une analyse détaillée. Pour cela il nous faut connaître le rôle des différents éléments composant ce formalisme, maîtriser les actions itératives et alternatives.

Il nous faut aussi connaître les différentes informations que l'on doit donner avant de commencer l'écriture d'un algorithme.

Nous allons utiliser une forme algorithmique pour exprimer le résultat de notre analyse détaillée. D'une part, c'est un formalisme proche de la langue naturelle (forme textuelle). D'autre part, de nombreux langages informatiques ont été conçus à partir de telles structures. Le passage de l'algorithme à la programmation s'en trouve ainsi facilité.

L'usage d'un formalisme permet de mieux communiquer au niveau de l'équipe de développement : le langage est commun. Cela n'est cependant pas suffisant. Il faut aussi des règles d'utilisation de ce langage qui soient communes (présentation des algorithmes, entête des procédures et des programmes ...).

2.2.1 ANALYSE BIBLIOGRAPHIQUE :

Vous étudierez dans le document sur notre formalisme 'Algorithmique de l'AFPA'. Ce formalisme sera à utiliser rigoureusement au cours de tous les exercices ou réalisations faites au cours du stage. Vous n'étudierez pas encore les chapitres 2.2.5 à 2.2.9, 2.5 et 2.6.

A l'issu de cette étude, exprimez un *tantque* avec un répéter et réciproquement.

2.2.2 EXERCICES :

- **calculer le nombre de jeunes.**
  Il s'agit de dénombrer toutes les personnes d'âge inférieur strictement à vingt ans parmi un échantillon donné de vingt personnes. Les personnes saisissent leur âge sur le clavier.

  Donnez l'algorithme correspondant.

- **calculer le nombre de jeunes, de moyens et de vieux.**
  Il s'agit de dénombrer les personnes d'âge inférieur strictement à 20 ans, les personnes d'âge supérieur strictement à 40 ans et celles dont l'âge est compris entre 20 ans et 40 ans (20 ans et 40 ans y compris).
  Le comptage est arrêté dès la saisie d'un centenaire.

  Donnez l'algorithme correspondant qui affiche les résultats.

2.3 Premiers algorithmes

Nous allons nous entraîner à assimiler les principes algorithmiques. Il nous faut :

- Maîtriser le parcours séquentiel d'une structure de donnée simple. (tableau)
- Maîtriser les principaux bornages de parcours.
- Maîtriser les instructions itératives pour les différents parcours.
- Comprendre la notion de procédure (définition, description, utilisation) et la notion de communication avec le reste du programme.
- Connaître la notion de variables locales et globales.

Le parcours séquentiel d'une structure de donnée est quelque chose de fondamental dans le traitement informatique. Nous allons donc nous y entraîner.

Mais les problèmes posés aux informaticiens peuvent être de différentes natures :
- soit le développement d'un programme complet.
- soit dans le cadre de développement d'un projet l'écriture d'une procédure (c'est à dire d'un morceau bien défini du programme qui s'intégrera parfaitement dans l'ensemble).

Nous allons donc traiter les parcours séquentiels dans ces deux cas, avec dans le deuxième cas la réalisation d'un programme nous permettant de tester notre procédure (tests unitaires).

2.3.1 ANALYSE BIBLIOGRAPHIQUE :

Vous étudierez, dans le document sur notre formalisme 'Algorithmique de l'AFPA', les chapitres 2.2.5, 2.5 et 2.6. (Les chapitres 2.2.6 à 2.2.9 ne sont pas à étudier)

2.3.2 EXERCICES

- **compter une lettre dans une phrase.**
  Soit une chaîne de caractères terminée par le caractère '.' . Donnez l'algorithme d'un programme qui compte le nombre occurrences d'une lettre donnée ('a' par exemple) dans cette chaîne.

- **compter le nombre de caractères dans une phrase.**
  Soit une chaîne de caractères terminée par le caractère '.' .
  Donnez l'algorithme d'une procédure qui compte la longueur de cette chaîne ('.' non compris).

- **compter les occurrences de deux lettres successives dans une phrase.**
  Donnez l'algorithme d'une procédure qui, pour une chaîne donnée terminée par un caractère terminateur donné et pour deux lettres données ('l', 'e' par exemple), compte le nombre occurrences de ces deux lettres successives dans la chaîne.

  Fournissez un programme de test de votre procédure.

- **rechercher une symétrie dans une chaîne de caractères.**
  Un palindrome est une chaîne de caractères que l'on peut lire identiquement de droite à gauche , et de gauche à droite.

  Par exemple :

  AA.
  38783.
  LAVAL.
  LAVAL A ETE A LAVAL.

  Soit une chaîne de caractères terminée par un point. Ecrivez l'algorithme d'une procédure permettant d'affirmer si cette phrase est ou non un palindrome.

2.4 Démarche de construction d'algorithmes

En commençant par définir notre algorithme en pseudo-langage, nous évitons d'affronter en même temps la réflexion de fond sur le problème et les difficultés spécifiques à un langage informatique (problèmes de compilation, « pièges » du langage…)

Mais, lorsque les problèmes posés deviennent plus complexes, la phase de réflexion sur papier doit être soutenue par une « démarche » systématique : un bon algorithme se « construit » étape par étape. Pour illustrer cette démarche, nous nous appuierons sur un cahier des charges simple :

> *A partir d'une phrase terminée par un point, où les mots sont séparés par un ou plusieurs blancs, nous voulons constituer une copie « compressée » qui élimine les blancs superflus : la phrase « compressée » commencera directement par le premier mot, et se terminera par un point ; chaque mot sera séparé de son suivant par un seul blanc ; le point final sera accolé au dernier mot.*

Etape 1 : définir un « jeu d'essai »

- Représenter l'état initial (ou les données initiales) du problème à résoudre, puis l'état final (ou le résultat final) du problème résolu. Un bon jeu d'essai doit prévoir le cas général, mais aussi tous les cas particuliers.

1) Une suite d'espaces au début et à la fin du texte

␣␣␣S|A|L|U|T| | |L|E|S| | |C|O|P|A|I|N|S| |A| |B|I|E|N|T|O|T| ␣␣␣␣.|

donne

S|A|L|U|T| |L|E|S| |C|O|P|A|I|N|S| |A| |B|I|E|N|T|O|T|.|

2) Uniquement le terminateur

.|

donne

.|

3) Une suite d'espace et le terminateur

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣.|

donne

.|

4) Une suite d'espaces à la fin du texte

S|A|L|U|T| | |L|E|S| | |C|O|P|A|I|N|S| |A| |B|I|E|N|T|O|T| ␣␣␣␣ ␣.|

donne

S|A|L|U|T| |L|E|S| |C|O|P|A|I|N|S| |A| |B|I|E|N|T|O|T|.|

5) Une suite d'espaces au début du texte

␣␣␣␣␣␣␣S|A|L|U|T| | |L|E|S| | |C|O|P|A|I|N|S| |A| |B|I|E|N|T|O|T|.|

donne

S|A|L|U|T| |L|E|S| |C|O|P|A|I|N|S| |A| |B|I|E|N|T|O|T|.|

Etape 2 : définir l'interface de l'algorithme

- En partant du jeu d'essai, posons-nous les questions : que doit-on fournir à l'algorithme pour qu'il puisse effectuer le traitement demandé ? Que nous renvoie-t-il ?

```
Procédure Compresser (Entrée original : chaîne ; Sortie copie : chaîne)
// La procédure permet de compresser une phrase en éliminant tous les
// espaces superflus.
// original est la phrase à compresser
// copie est la phrase épurée des espaces superflus
```

Nous n'oublions pas de commenter le rôle de la procédure, ainsi que le rôle de chacun des paramètres. Ceci nous permet d'utiliser la procédure indépendamment de son écriture.

- Remarque : cette interface exige que l'original soit délimité par un caractère terminateur, et que ce terminateur soit conservé dans la copie. S'il n'existait pas de terminateur, il faudrait préciser le nombre de caractères utiles dans la phrase d'origine et dans sa copie, ce qui donnerait l'interface :

```
Procédure Compresser (Entrée original : chaîne ; Entrée nbCarOriginal : entier ;
                      Sortie copie : chaîne ; Sortie nbCarCopie : entier)
// La procédure permet de compresser une phrase en éliminant tous les
// espaces superflus.
// original est la phrase à compresser
// nbCarOriginal est le nombre de caractères de la phrase originale
// copie est la phrase épurée des espaces superflus
// nbCarCopie est le nombre de caractères de la phrase copie
```

Etape 3 : définir le programme de test

- Chaque étape complète et valide les précédentes : il ne sert à rien de décrire le détail de notre algorithme, si la « boîte noire » que nous venons de définir est inutilisable !

- Le programme de test doit donc montrer comment on va de l'état initial à l'état final du jeu d'essai, en appelant la « boîte noire » définie par son interface.

```
programme      TestCompresser

// Ce programme effectue une copie d'une phrase saisie au clavier, en retirant tous les blancs superflus

Constantes     taille = 100                  // longueur maximum d'un texte
               carterm = '.'                 // caractère terminateur : marque la fin d'un texte

Types
               chaîne = tableau [taille] de caractères    // un texte terminé par le caractère carterm

Variables      phrase      : chaîne          // le texte saisi au clavier
               compressee : chaîne           // le texte filtré, sans les blancs
```

Commentaires obligatoires !

<u>Procédure</u> Compresser (**<u>Entrée</u>** original : chaîne ; **<u>Entrée</u>** nbCarOriginal : entier)

// La procédure permet de compresser une phrase en éliminant tous les
// espaces superflus.
// original est la phrase à compresser

// copie est la phrase épurée des espaces superflus

Cahier des charges
du test du composant

<u>Début</u> *// début du programme*
Ecrire ('Entrez votre phrase')
Lire (phrase)
Compresser (phrase, compressee)
Ecrire ('La phrase filtrée est', compressee)
<u>fin</u>

<u>Etape 4 : détailler l'algorithme</u>

- Les trois premières étapes sont obligatoires et doivent être effectuées dans l'ordre indiqué : nous savons maintenant ce que <u>doit faire</u> notre algorithme, et comment il est utilisé dans le programme principal.

- Il reste à « ouvrir » la boîte noire, et à expliquer <u>comment</u> l'algorithme doit procéder pour parvenir au résultat.

- Toutes les opérations qui suivent devront être réalisées pour construire l'algorithme définitif, mais elles pourront l'être dans un ordre différent selon le problème : elles sont complémentaires et doivent « s'éclairer » mutuellement.

4.1 Visualiser la structure de données

- Représenter graphiquement la structure de données utilisée : succession de cases mémoires, graphe arborescent, tableau, succession de tableaux. Ne pas hésiter à « inventer » des formalismes ou graphismes, si cela peut éclairer le problème ! Pour notre problème, nous utiliserons un tableau de caractères :

| |S|A|L|U|T| | |L|E|S| | |C|O|P|A|I|N|S| |A|
|B|I|E|N|T|O|T| |.

4.2 Faire tourner l'algorithme à la main, en recherchant les régularités

- Il faut partir d'un cas intermédiaire : l'algorithmique est une discipline « optimiste » où l'on suppose toujours qu'une partie du problème a déjà été traitée.

  En supposant ici que l'on a su compresser les I premières cases de notre tableau, comment faut-il faire pour traiter la case I + 1 ?

  Trouver un bon algorithme revient à formuler une hypothèse de récurrence, même si on n'a pas besoin du formalisme mathématique.

| | |S|A|L|U|T| | **|L|E|S| | |C|O|P|A|I|N|S| |A|**
|B|I|E|N|T|O|T| |.

S|A|L|U|T| **| | | | | | | | | | | | | | | | | | | | | | | | | | | |**

- Pour poursuivre le traitement, il faudra :

    - recopier les lettres « LES »
    - recopier également le premier BLANC après le S
    - sauter tous les BLANCS jusqu'à trouver le mot suivant ou le terminateur…

- Le caractère situé à la case I +1 doit donc être recopié :

    - dans tous les cas, si ce n'est pas un BLANC
    - si c'est un BLANC, uniquement si le caractère précédent n'est pas un BLANC

- L'action à effectuer (**recopier** ou **sauter**) peut être résumée par une table de vérité :

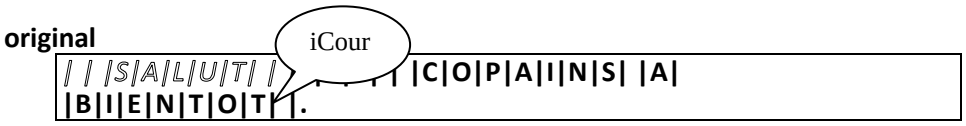
Car. courant *Car. précédent*	**Non Blanc**	**Blanc**
Non Blanc	recopier	recopier
Blanc	recopier	sauter

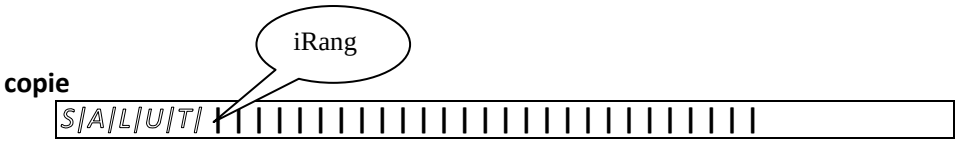
- On ne sautera donc le caractère courant que dans un cas : si c'est un BLANC précédé d'un BLANC. On recopie le caractère courant si ce n'est pas un BLANC ou si le caractère précédent n'est pas un BLANC.

4.3 Prendre des conventions, et écrire le cas général de l'algorithme

- L'étape précédente nous permet de repérer les « invariants » de boucle : ce qui est commun entre le traitement du ième et du (i+1) ème caractère.

- Appelons **iCour** l'indice du prochain caractère qui doit être traité dans le tableau **original** (tous les caractères précédents dans le tableau ont déjà été traités). **iRang** l'indice de la première case libre dans le tableau **copie**





- Le cas général peut se formuler comme suit :

```
// Si le caractère courant n'est pas un blanc ou que le caractère précédent n'est pas un blanc…

Si  original[iCour] <> BLANC ou original[iCour-1] <> BLANC alors

      // recopie du caractère courant dans la copie compressée
       copie[iRang] := original[iCour]
      iRang := iRang +1
Finsi

// avance dans la phrase
```

```
iCour := iCour +1
```

4.4 Préciser la condition d'arrêt

- L'algorithme s'arrête lorsque l'on rencontre le caractère terminateur, d'où la condition d'arrêt :

```
original[iCour] = carterm
```

- Réécriture de l'algorithme avec la boucle principale :

```
Tantque original[iCour] <> carterm faire

    Si  original[iCour] <> BLANC ou original[iCour-1] <> BLANC alors
       copie[iRang] := original[iCour]
     iRang := iRang +1
    Finsi

    iCour := iCour +1
Fintantque
```

4.5 Initialisation

- Nous connaissons maintenant le cas général : comme dans une vraie récurrence en mathématiques, il faut maintenant préciser le premier terme (U0), en revenant sur nos conventions.

- Notre algorithme revient donc à déplacer un CURSEUR constitué par les cases d'indices **iCour** et **iCour-1** dans le tableau **original.** Nos conventions supposent donc qu'il existe une case **iCour – 1** : l'algorithme devra donc commencer sur la deuxième case, avec iCour = 2.

- La première case doit être traitée à part, dans l'initialisation : elle ne sera recopiée que si elle n'est pas égale à BLANC (voir cahier des charges *: le texte compressé doit commencer sur le premier mot*).

```
    iRang := 1
    iCour := 1

    Si  original[iCour] <> BLANC  alors
       copie[iRang] := original[iCour]
     iRang := iRang +1
    Finsi

    iCour := iCour + 1
```

4.5 Traitement final

- Le Terminateur n'est pas recopié (condition de sortie de la boucle). Il faut donc le recopier à la fin du tableau : à l'indice **iRang** si aucun caractère n'a été stocké ou si le dernier caractère stocké n'est pas un BLANC, à l'indice **iRang-1** si le dernier caractère stocké est un BLANC (voir cahier des charges : *le point doit toujours être accolé au dernier mot dans la phrase compressée*)

```
    Si  (iRang > 1) et (copie[iRang] = BLANC)  alors
```

```
        iRang := iRang - 1
    Finsi
    copie[iRang] := carterm
```

4.6 Cas particuliers

- Il est temps de rejeter un coup d'œil aux jeux d'essais : avons-nous vraiment prévu tous les cas ?

⇨ Si le texte original se limite à un point, l'algorithme ne fonctionne pas : il stocke le point (différent de BLANC) dans le tableau **copie** et boucle indéfiniment (recherche le terminateur à partir du deuxième caractère). Il faudra donc rajouter ce cas particulier dans la version finale de l'algorithme.

4.7 Réécriture finale de l'algorithme

- Cette démarche peut paraître longue, mais nous sommes maintenant bien armés pour écrire l'algorithme défnitif, en « collant les morceaux ». On en profitera pour prendre du recul, et faire une dernière relecture critique.

(voir algo page suivante)

```
Procédure Compresser (Entrée original : chaîne ; Sortie copie : chaîne)
// La procédure permet de compresser une phrase en éliminant tous les
// espaces superflus.
// original est la phrase à compresser
// copie est la phrase épurée des espaces superflus

Constantes      BLANC = ' '

Variables       iCour : entier          // indice du prochain caractère à traiter dans le tableau original
                iRang : entier          // indice de la première case libre dans le tableau copie
début
 iCour := 1
 iRang := 1

  // cas particulier du texte vide
 Si original[iCour] = carterm alors
     copie[iRang] := carterm
 sinon
        // initialisation : recopie du premier caractère si non blanc
        Si  original[iCour] <> BLANC  alors
           copie[iRang] := original[iCour]
           iRang := iRang +1
        Finsi
        iCour := iCour + 1

        //  cas général
        Tantque original[iCour] <> carterm faire

                Si  original[iCour] <> BLANC ou original[iCour-1] <> BLANC alors
                   copie[iRang] := original[iCour]
                   iRang := iRang +1
                Finsi
                iCour := iCour +1
        Fintantque

        //  traitement final
        Si  (iRang > 1) et (copie[iRang-1] = BLANC)  alors
           iRang := iRang - 1
        Finsi
        copie[iRang] := carterm
 Finsi
Fin
```

Etape 5 : optimiser l'algorithme

- En suivant les étapes, nous obtenons une solution fiable, mais qui n'est pas forcément la meilleure. En règle générale, l'étape finale va consister à rechercher les « optimisations » possibles, en terme de rapidité de traitement ou de simplification de l'algorithme (parfois les deux en même temps).

- Une piste : essayer de supprimer les cas particuliers inutiles qui compliquent l'algorithme. Le plus souvent, ces cas particuliers ne viennent pas du problème lui-même, mais de notre manière de l'appréhender !

- Par exemple, l'algorithme de la page précédente est trop compliqué, par rapport au problème posé. Nous devrons le reprendre en changeant d'hypothèses, pour aboutir à une solution plus simple *(voir chapitre 3.1.2)*

2.4.1 EXERCICE : Validez le jeu d'essai, en faisant tourner l'algorithme à la main

2.5 Algorithmes plus complexes

Nous allons maintenant apprendre à :

- Maîtriser les outils algorithmiques.

- Savoir mettre en œuvre les outils de vérification des algorithmes.
  - valeurs initiales.
  - conditions de sortie des instructions itératives.
  - résultats attendus.

Les algorithmes de tri et de recherche sont des grands classiques de l'algorithmique. On les trouve dans tous les bons ouvrages traitant du sujet. Cependant il est bon d'en avoir traité quelques exemples, car ils demandent pour leur réalisation une grande attention.

Ces algorithmes représentent le niveau maximum de difficulté que peut présenter un algorithme, et nous amènent à faire des vérifications pour valider leur fonctionnement.
La seule façon de vérifier un algorithme est de le faire tourner à la main. Il faut alors noter toutes les modifications des variables sur un échantillon de données soigneusement choisies. Ces jeux d'essais sont définis dans la phase de conception détaillée, avant l'écriture des algorithmes.

Nous allons également faire des algorithmes de calcul sur les bits et sur les codes ascii, qui nous demanderons de découvrir un mode calculatoire puis de le transcrire sous forme algorithmique.

Trois remarques pour finir :

- Pour réaliser un bon test d'un algorithme, il faut être aussi rigoureux qu'un ordinateur. Ce n'est pas si facile.

- La personne qui a écrit un algorithme n'a qu'une envie : qu'il marche.

- La personne qui teste un algorithme ne doit avoir qu'une envie: qu'il ne marche pas.

Tirez-en les conclusions.

2.5.1 ANALYSE BIBLIOGRAPHIQUE :

N'accéder à la littérature qu'à la fin de la séquence. Vous trouverez dans les ouvrages traitant de l'algorithmique la solution de problèmes de tri et de recherche.

2.5.2 EXERCICES :

- **trier un tableau d'entiers par la méthode de tri par remontée des bulles.**
  Donnez l'algorithme de la procédure paramétrée correspondant à la méthode de tri par remontée des bulles (voir la méthode ci-dessous).
  Faites votre algorithme le plus simple possible pour qu'il fasse le traitement tel qu'il est proposé.

  Trouvez ensuite un algorithme qui respecte cette méthode de tri, mais qui l'optimise. Il ne va trier que ce qui est nécessaire, mais toujours selon la même méthode.
  Optimisez progressivement, il y a 3 optimisations possibles.

Principe du tri par remontée des bulles : on parcourt le tableau de valeurs en comparant les éléments deux à deux. Si le plus grand est avant le suivant, on inverse les éléments dans le tableau.

On fait autant de parcours du tableau que nécessaire pour que le tableau soit trié (et qu'on s'en rende compte).

Exemple : Etat du tableau pendant le parcours :

premier parcours

9	1	1	1
1	**9**	4	4
4	**4**	**9**	2
2	2	**2**	9

en grisé éléments comparés deux à deux

il y a eu des inversions pendant le premier parcours

deuxième parcours

1	1	1	1
4	**4**	2	2
2	**2**	**4**	4
9	9	**9**	9

en grisé éléments comparés deux à deux

il y a eu des inversions pendant le deuxième parcours

troisième parcours

1	1	1	1
2	**2**	2	2
4	**4**	**4**	4
9	9	**9**	9

en grisé éléments comparés deux à deux

il n'y a pas eu d'inversions pendant le troisième parcours

Le tableau est donc trié. Cette méthode est utilisée pour trier des tableaux peu désordonnés. Il existe bien d'autres méthodes de tri, plus performantes sur des grands tableaux et sur des tableaux « très désordonnés ».

- **rechercher par dichotomie un élément d'une table classée.**
  Soit une table contenant des prénoms, classés par ordre alphabétique.

  Nous désirons chercher l'indice de la case du tableau où se trouve un prénom dans le tableau, si il s'y trouve.

  Pour cela nous utiliserons la méthode de dichotomie (voir ci-dessous la méthode).

  Donnez l'algorithme de la procédure qui recherche, par dichotomie, le numéro du prénom cherché ou zéro s'il n'y est pas. On suppose que l'on sait comparer des chaînes de caractères (= > < >= <= <>).

  Il peut être instructif de refaire le même algorithme, en tenant compte du fait que l'on ne sait pas comparer des chaînes de caractères.

Principes de la recherche par dichotomie :

Exemple table des prénoms

1 ->	agathe	Les prénoms sont classés par ordre alphabétique ;
2 ->	berthe	On connaît la taille de la table (7 ici)
3 ->	cellulite	
4 ->	cunegonde	<- M0
5 ->	olga	<- M2
6 ->	raymonde	<- M1
7 ->	sidonie	

On cherche 'olga' dans la table.

Principe : On partitionne la table en deux sous-tables et un élément médian, et, suivant le résultat de la comparaison de l'élément médian et du prénom recherché (plus grand, plus petit, ou égal) on recommence sur une des deux "sous-table" de la table la recherche, jusqu'à avoir trouvé ou obtenir une sous-table vide (le prénom est alors absent de la table).

```
chercher('olga') : milieu c'est l'élément n°4                    (sur M0 )
                   'olga' > élément n°4
                   on traite le tableau du dessous ( 5 à 7 )
                            milieu c'est l'élément n°6           (sur M1 )
                            'olga'  < élément n°6
                            on traite le tableau au dessus ( 5 à 5 )
                                     milieu c'est l'élément n°5  (sur M2)
                                     'olga' = élément n°5
```

Le numéro de la case contenant 'olga' est donc 5.

Ainsi on pourrait trouver un prénom dans une table de 1000 prénoms en maximum 11 comparaisons, car à chaque comparaison on divise l'espace de recherche par 2.

- **donner la valeur du bit numéro x d'un entier.**

  Donnez l'algorithme d'une procédure qui, pour un entier, donne la valeur du bit numéro x de cet entier. Cette valeur sera donnée sous la forme d'un booléen : vrai si la valeur est 1, faux si elle est 0.

  Le LSB (Lower Significant Bit) a pour numéro 0.

  Rappel : il existe deux opérateurs sur les entiers qui sont MOD et DIV utilisables de la manière suivante:
  7 MOD 3 = 1 reste de la division entière de 7 par 3
  7 DIV 3 = 2 résultat de la division entière de 7 par 3.

- **Coder une phrase en une autre en code ascii.**
  On désire enregistrer des informations en caractères ASCII étendu sur des étiquettes magnétiques. Hélas le protocole de dialogue avec le programmeur d'étiquettes n'autorise pas d'envoyer de tels caractères. Il n'accepte de transmettre que des caractères imprimables du code ASCII (<80 hexa). Pour pouvoir imprimer correctement les étiquettes, il faut opérer de la sorte :

  Pour transmettre 'A' (code ascii 41 hexa)
  - Envoyer les caractères '4' et '1' (code ascii 34 et 31 hexa)
  - Le programmeur d'étiquettes reçoit '4' et '1', associe les 2 caractères et imprime 'A' .

  Donnez l'algorithme de la procédure qui, pour une chaîne de caractères de longueur donnée, constitue la chaîne de caractères à envoyer au programmeur d'étiquettes.

3 - CONCEPTION DESCENDANTE D'ALGORITHMES

A partir des spécifications d'une fonction informatique complexe, nous devons être capable de faire la conception préliminaire du problème, puis d'en faire les différents algorithmes. Nous devons ensuite prendre en compte les documents que nous allons réaliser pendant cette phase (définition de ce que l'on va faire, puis la réalisation). Enfin nous prendrons en compte les contraintes dues à la maintenance.

Nous allons aborder les thèmes techniques suivants :
- Sensibilisation à la réalisation d'une application de manière arborescente.
- Variables locales et globales.
- Sensibilisation au découpage propre du logiciel.
- Intégration de procédure et tests d'intégration.

3.1 Algorithmes simples à plusieurs niveaux

Nous allons mettre en œuvre le traitement séquentiel dans une analyse à plusieurs niveaux.

Nous devons être capable, dans le traitement d'un exemple plus complexe, de nous ramener à des traitements simples en décomposant le travail en procédures. Ceci nous oblige à mettre en oeuvre la phase de conception préliminaire.

Jusqu'à maintenant notre analyse a porté sur une suite d'éléments simples (caractères ou noms que l'on peut comparer). Hélas les choses ne sont pas toujours aussi simples. Un livre par exemple peut être vu comme un ensemble de chapitres, un chapitre comme un ensemble de paragraphes, un paragraphe comme un ensemble de phrases, une phrase comme un ensemble de mots et enfin un mot comme un ensemble de caractères. Un livre est aussi un ensemble de caractères. A nous de voir laquelle de ces manières de considérer un livre nous intéresse pour traiter le problème posé.

Exemple : le problème de recherche d'un nom dans une table. A priori on ne dispose pas des outils de comparaison de noms. Il faut alors créer ces outils qui travaillent au niveau des caractères. Le programme de recherche, qui lui utilise ces outils, travaille au niveau des noms et ignore superbement toute considération de caractères.

Quand le problème devient complexe, il nous faut introduire ces niveaux de raisonnement. Cela nous permet de sérier les problèmes, ainsi que de partager le travail. Avec un peu de chance cela permet aussi de récupérer des outils existants.

3.1.1 ANALYSE BIBLIOGRAPHIQUE :

Voir les livres de la bibliothèque. Certains principes sont bien analysés dans "Algorithmique et représentation des données", bien qu'un peu théoriques.

3.1.2 REPRISE DE L'EXERCICE 2.4

- Pour illustrer la conception d'algorithmes à plusieurs niveaux, nous allons reprendre l'analyse de l'algorithme d'élimination des espaces superflus.

- La démarche va être modifiée, à partir de l'étape **4** : avant de réaliser l'algorithme détaillé, nous allons écrire un « algorithme de principe » qui va mettre en évidence les actions principales de l'algorithme. Ces actions deviendront souvent des procédures ou des fonctions.

1° Ecrire l'algorithme de principe

- Le bon algorithme va progressivement du général au particulier. Plutôt que de passer directement de la phrase tout entière (ensemble de caractères) au caractère, nous allons isoler des sous-ensembles : mots, et suites d'espaces.

- Tout algorithme a une « forme » propre qu'il faut respecter, sans quoi on invente des cas particuliers inutiles. En repartant du jeu d'essai, décomposons notre algorithme en actions élémentaires sur les mots et les suites d'espaces (noté #) :

#####MOT1#MOT2###MOT3#####.
Donne
MOT1#MOT2#MOT3.

⇨ *A partir de 3 mots, 4 suites d'espaces et un point : on copie 3 mots, 2 espaces intermédiaires et un point*

MOT1###MOT2##MOT3####.
Donne
MOT1#MOT2#MOT3.

⇨ *A partir de 3 mots, 3 suites d'espaces et un point : on copie 3 mots, 2 espaces intermédiaires et un point*

####MOT1###MOT2#MOT3.
Donne
MOT1#MOT2#MOT3.

⇨ *A partir de 3 mots, 3 suites d'espaces et un point : on copie 3 mots, 2 espaces intermédiaires et un point*

MOT1###MOT2##MOT3.
Donne
MOT1#MOT2#MOT3.

⇨ *A partir de 3 mots, 2 suites d'espaces et un point : on copie 3 mots, 2 espaces intermédiaires et un point*

- **Conclusions** :

⇨ On recopie toujours le même nombre de mots, et d'espaces intermédiaires, indépendamment des suites d'espaces avant, après et entre les mots. Le nombre de mots et d'espaces intermédiaires à copier ne dépend que du nombre de mots présents dans le texte initial.

⇨ Il faut copier un espace après chaque mot, sauf pour le dernier

⇨ On termine toujours l'algorithme en copiant le caractère terminateur.

Pour trouver le « bon algorithme », il faudra donc traiter indépendamment ces quatre points :

- la copie des mots
- la copie des espaces intermédiaires
- le saut des espaces
- la copie du terminateur

Algorithme de principe

```
sauter les espaces

Tantque non fini faire
   copier un mot
   sauter les espaces
   Si non fini alors
      copier espace
   Finsi
FinTantQue

copier terminateur
```

- Avant de détailler l'algorithme, toujours valider l'algorithme de principe avec les jeux d'essai : rien ne sert de détailler un algorithme qui ne tourne pas !

2° Lister les interfaces des sous-procédures et fonctions

Les actions principales donnent lieu à l'écriture de procédures ou fonctions, dont il faut d'abord préciser l'interface :

Procédure StockerMot (**Entrée** original : chaîne ; **Entrée-sortie** iCour : **entier**
Sortie copie : chaîne ; **Entrée-sortie** iRang : **entier**)

*// Cette procédure recopie un mot du tableau **original** à partir de l'indice **iCour** dans le*
*// tableau **copie** à partir de l'indice **iRang**.*

*// Elle fait progresser les indices **iCour** et **iRang** : en sortie,*
*// - **iCour** pointe sur le premier espace après le mot ou sur le terminateur*
*// - **iRang** pointe sur la première case libre du tableau **copie***

Procédure SauterEspaces (**Entrée** original : chaîne ; **Entrée-sortie** iCour : **entier ;**
Sortie fini : **booléen)**

*// Cette procédure saute tous les espaces rencontrés dans le tableau **original** à partir de l'indice*
*// **iCour**. Elle fait progresser l'indice **iCour***

*// En sortie, **iCour** pointe sur la première lettre du mot suivant ou sur le terminateur*
*// **fini** vaut VRAI si on a atteint le terminateur*

3° Détailler l'algorithme principal à l'aide des interfaces

- Dans le cas d'algorithmes à plusieurs niveaux, il faut s'assurer au fur et à mesure de la validité des algorithmes de principe et des interfaces proposés, en détaillant l'algorithme principal : il ne sert à rien de commencer à analyser le « Comment » d'interfaces qui ne « s'emboîtent » pas correctement.

```
Procédure Compresser (Entrée original : chaîne ; Sortie copie : chaîne)

Constantes   BLANC = ' '
             Carterm = '.'

Variables  iCour : entier      // indice du prochain caractère à traiter dans le tableau original
           iRang : entier      // indice de la première case libre dans le tableau copie
           fini : booleen      // vrai si on a atteint le terminateur

début
    iCour := 1                          // traitement du premier caractère dans original
    iRang := 1                          // indice de la première case libre dans copie

    SauterEspaces (original, iCour, fini)        // saut des espaces en début du texte

    Tantque non fini faire
        StockerMot (original, iCour, copie, iRang)   // copie d'un mot
        SauterEspaces (original, iCour, fini)        // saut des espaces intermédiaires
        Si non fini alors                            // si non fin texte, copie d'un espace
            copie[iRang] := BLANC
            iRang := iRang + 1
        Finsi
    FinTantQue

    copie[iRang] := Carterm                  // copie du terminateur
Fin
```

4° Pour chaque procédure ou fonction, donner un jeu de test et détailler son algorithme

procédure StockerMot

…… **jeux d'essai**……..

ENTREES	SORTIES
1) original = 'Le###**g**rand##méchant##loup.'	copie = 'Le#grand'
iCour = 6	iCour = 11
iRang = 4	iRang = 9
2) original = 'Le###grand##méchant##**l**oup.'	copie = 'Le#grand#méchant#loup'
iCour = 22	iCour = 26
iRang = 18	iRang = 22

……**algorithme détaillé**……

procédure StockerMot (**Entrée** original : chaîne ; **Entrée-sortie** iCour : **entier**
Sortie copie : chaîne ; **Entrée-sortie** iRang : **entier**)

Début
// parcours et stockage du mot : jusqu'à espace ou terminateur

Tantque (original[iCour] <> BLANC) **et** (original[iCour] <> carterm) **faire**
copie[iRang] := original[iCour]
iRang := iRang + 1
iCour := iCour + 1
FinTantque
Fin

2° procédure SauterEspaces

…… **jeux d'essai**……..

	ENTREES	SORTIES
1)	original = 'Le**#**##grand##méchant##loup.' iCour = 3	iCour = 6 fini = faux
2)	original = 'Le###grand##méchant##loup**.**' iCour = 26	iCour = 26 fini = vrai

……**algorithme détaillé**……

Procédure SauterEspaces (**Entrée** original : chaîne ; **Entrée-sortie** iCour : **entier ;**
Sortie fini : **booléen)**

début

// Saut des espaces : jusqu'au début du mot suivant, ou jusqu'au terminateur du texte
Tantque (original[iCour] = BLANC) **faire**
iCour := iCour + 1
FinTantque

// retourne VRAI si le texte n'est pas fini (terminateur non rencontré)
Si original[iCour] = Carterm **alors**
fini := VRAI
sinon
fini := FAUX
Finsi

Fin

3.1.3 EXERCICES

Pour réaliser ces exercices, adoptez la démarche proposée pour les algorithmes à plusieurs niveaux

- **chercher un mot dans une phrase.**

Soit une phrase terminée par un point. Donnez l'algorithme de la procédure qui pour un mot donné (mot et longueur du mot) détermine si le mot est dans la phrase.

- **justifier une phrase.**

Soit une phrase de 80 caractères terminée par un point.
Donnez l'algorithme du programme qui justifie cette phrase. La justification d'une phrase consiste à répartir les mots sur la totalité de la phrase en répartissant équitablement les espaces entre les mots. S'il reste des espaces, ils seront rajoutés aux premiers intervalles.

- **Recopier une phrase en inversant chaque mot.**

Soit une phrase terminée par un point.

Donnez l'algorithme de la procédure qui inverse chacun des mots de cette phrase et qui rend le résultat dans une deuxième phrase. Cette deuxième phrase comportera un espace entre deux mots et le point final sera cadré le plus à gauche possible.

Exemple :

le#chat##est###gris#####.

Donnera

el#tahc#tse#sirg. (un seul espace entre les mots)

3.2 Algorithmes complexes à plusieurs niveaux

Nous allons maîtriser la conception préliminaire de problèmes informatiques complexes.

Ceci implique que nous ayons la maîtrise de la découpe d'un problème complexe en problèmes simples, en gérant les communications entre les différents problèmes simples.

Nous allons être confrontés à des problèmes inextricables sans une analyse rigoureuse du problème, un choix de stratégie de résolution, un découpage modulaire et descendant et des algorithmes clairs, commentés et paramétrés.

Il est quelquefois difficile de dire si sa conception préliminaire est correcte ou non quand on débute. Mais à posteriori, si l'écriture des algorithmes pose des problèmes majeurs, ou dévoile des zones d'ombres, il est alors important de remettre en cause sa conception préliminaire.

Tout retour arrière dans le développement d'un projet étant une perte de temps importante, il est nécessaire de prendre le temps de faire une analyse organique générale approfondie avant de se lancer dans l'écriture des algorithmes.

Nous allons donc mettre en œuvre toutes nos facultés d'analyse, d'abstraction et de réflexion au cours de ces exercices.

3.2.1 ANALYSE BIBLIOGRAPHIQUE :

Voir les livres de la bibliothèque. Certains principes sont bien analysés dans "Algorithmique et représentation des données", bien qu'un peu théoriques..

3.2.2 EXERCICES

- **calculer la somme de nombres en base quelconque.**

  Soit une chaîne de caractères représentant des nombres dans une base quelconque comprise entre 2 et 16.
  Soit b la base dans laquelle ces nombres sont écrits.

  Le premier caractère zéro rencontré en début de nombre indique la fin de la chaîne de caractères à traiter.

  Donnez l'algorithme de la procédure qui, à partir de ces informations, donne l'entier représentant la somme des nombres contenus dans cette chaîne.

Exemple :

1	7		2	0	3			9		0	

base b = 13

Le calcul donnera un entier de valeur 1*13+7 + 2*13*13+0*13+3 + 9 = 20 + 341 + 9 = 370

Note :

- On sait que les informations contenues dans la chaîne de caractères sont correctes
- Les nombres sont non signés.
- Les débordements ne seront pas traités.

4 ALGORITHMES ET STRUCTURES DE DONNEES

- Nous devons être capables de définir les structures de données les mieux adaptées à la réalisation d'une fonction informatique. Nous devons également savoir exploiter les principales structures de fichiers. Alors nous serons capables d'écrire les algorithmes mettant en œuvre ces structures de données et fichiers.

- Les thèmes techniques abordés sont les suivants :

    - Sensibilisation à la structuration des données.
    - Choix critique entre diverses solutions pour un même problème.

4.1 Algorithmes et représentation des données

- Nous allons apprendre à structurer de façon descendante les données à traiter dans un problème informatique.

- Nous allons également apprendre à structurer les données en prévision des traitements à effectuer.

- L'informatique est la science du traitement de l'information. Nous avons beaucoup parlé de traitement jusqu'à présent, nous allons maintenant nous intéresser à la représentation de l'information.

- Pour cela nous aurons deux approches complémentaires :

    - la première est l'analyse descendante de notre information (décomposition successive de notre information en informations élémentaires).

    - la deuxième approche viendra y greffer une structure liée aux traitements à effectuer. Nous définirons les traitements à effectuer sur une donnée, puis nous choisirons la structure de données qui permet de simplifier le traitement des données (au risque de compliquer la structure de données).

4.1.1 ANALYSE BIBLIOGRAPHIQUE :

Tous les livres d'algorithmique de la bibliothèque.

Vous étudierez particulièrement les chapitre 2.2.6, 2.2.7 et 2.2.8 de 'algorithme' de l'AFPA.
Vous y trouverez des nouveaux types de données.

4.1.2 EXERCICES

- **structurer des données associées à des pièces.**

  Une usine fabrique des pièces : sphériques, cubiques et cylindriques. Pour chaque pièce produite on veut connaître ses dimensions, sa couleur (jaune, vert, bleu, rouge, orange, mauve), son numéro de série et sa date de fabrication.

  Donnez une structure de donnée permettant de stocker toutes les informations sur les pièces produites par l'usine.

- **manipuler une pile d'entiers gérée avec un tableau et un indice.**

  Une pile d'entiers est une structure de données qui permet de stocker des entiers et de les restituer dans un ordre bien précis.

  Ces données seront restituées une à une sur demande selon la règle : dernière entrée, première sortie (LIFO: Last In First Out). Ceci correspond à l'usage habituel d'une pile d'assiettes ou de torchons.

  Cette pile est réalisée à l'aide d'un tableau d'entier et d'un indice de parcours.

1) Donnez le principe de fonctionnement avec les structures de données de cette pile (ajout et retrait d'un élément) en étudiant particulièrement ce qui se passe quand le tableau est vide et quand il est plein.

2) Donnez les interfaces des procédures «init_pile», «empiler», «dépiler».

3) Donnez les algorithmes des trois procédures.

- **gérer une liste de noms classés alphabétiquement.**

Il s'agit de constituer une liste de noms classés par ordre alphabétique. Cette liste peut évoluer dans le temps, un nom pouvant être ajouté ou retranché à cette liste, tout en conservant l'ordre alphabétique.

Pour cela, et pour optimiser les temps de traitement, utilisez la structure de données suivante :

```
Constantes      tailtable = 100          // taille du tableau contenant les noms
                tailnom = 10  // les noms ont dix caractères maximum
                final = 0                // indicateur de fin de liste

Types           chaînenom = tableau [ tailnom ] de caractère
                                         // type des noms rangés dans le tableau
                élément = Enregistrement
                                // type des éléments de la table des noms
                  nom : chaînenom        // nom est le nom contenu dans la table
                  suivant : entier       // indice du nom suivant dans la table
                Finenregistrement

                tablenom = Enregistrement
                              // type des tables de noms
                  table : tableau [ tailtable ] de élément
                                         // table permet de ranger les informations
                  libre : entier         // libre est le premier élément de la liste libre
                  premier : entier       // premier est le premier élément de la liste
                                // des noms
                Finenregistrement
```

Questions

1) donnez une procédure permettant d'initialiser la table des noms à l'origine (vide).

2) donnez une procédure permettant d'ajouter un nom à la table des noms.

3) donnez une procédure permettant d'enlever un nom de la table des noms.

Exemple de table de noms classée alphabétiquement :

1	Cunégonde	7	
2	Berthe	6	
3	Sidonie	0	
4		8	
5	Agathe	2	premier = 5
6	Cellulite	1	libre = 4
7	Raymonde	3	
8		9	
9		10	

- **créer une liste dynamique gérée en FIFO.**

Une liste dynamique est une liste à laquelle on peut rajouter un nombre quelconque d'éléments sans restriction de taille (sauf celle de la mémoire). Les éléments de la liste sont chaînés entre eux.

Structure d'un élément de la liste:

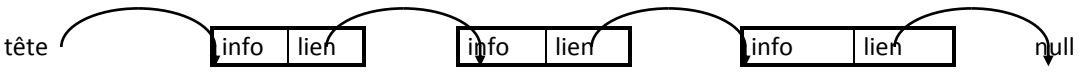
type
référence = **pointeur de** élément
élément = **enregistrement**
valeur : info // contient l'information que l'on désire conserver
suivant : référence // pointeur vers l'élément suivant de la liste
finenregistrement

La liste sera repérée par un pointeur sur son premier élément (la tête de la liste).
Elle se termine par le premier élément trouvé dont le lien est à **null** ou avec le pointeur de tête à **null** (en cas de liste vide).

Soit une liste dynamique d'entiers fonctionnant en FIFO (First In First Out). Le dernier élément de la liste est le dernier élément rentré dans la liste. Le premier élément de la liste est le premier élément rentré dans la liste et le premier élément qui sortira de la liste.

Ecrire une procédure qui rajoute un entier à la liste.
Ecrire une procédure qui récupère un élément de la liste.

Structure d'une liste dynamique :



- **gérer une liste dynamiquement.**

Il s'agit de faire le parallèle entre l'utilisation des pointeurs et les utilisations des indices pour désigner le nom suivant de la liste dans l'exercice des noms classés alphabétiquement.

Refaites l'exercice des noms classés alphabétiquement à l'aide d'une liste dynamique.

4.2 Les fichiers

Nous allons apprendre à traiter les fichiers de données. Toutes les données que nous traitions jusqu'à maintenant avaient une durée de vie qui ne pouvait pas dépasser celle du programme ou de la procédure où elles avaient été créées.

Quand nous désirons conserver des données sur des références de produits, sur des clients, ou sur des mesures, nous aimerions quelquefois que ces données aient une existence propre, c'est-à-dire qu'elles existent en dehors de tout programme. Elles pourraient alors être utilisées par un autre programme, ou être complétées ou modifiées par le même programme lors d'une exécution ultérieure.

Les fichiers vont nous permettre de conserver des données d'un programme. Ces données pourront être traitées ultérieurement. Il existe plusieurs modes de conservation de données dans un fichier. Nous en verrons deux : le mode séquentiel, et le mode indexé.

4.2.1 Rappels sur les fichiers

C'est un chapitre crucial de tout langage de développement. La finalité d'un programme est en effet d'établir un lien entre l'extérieur et l'espace de stockage des données.

Les données sont enregistrées sur support magnétique ou optique sous forme d'un ou plusieurs enregistrements, regroupés par nature en *fichiers*.

En gestion, il est courant que les enregistrements soient constitués de champs, (ou attributs, ou propriétés) se présentant toujours dans le même ordre au fil des enregistrements.

On parle d'Entrées-Sorties (E/S) ou I/O en anglais. Clarifions un point de vocabulaire :

- Entrée (*Input*) exprime la lecture de la donnée depuis le support (fichier disque) vers le programme
- Sortie (*Output*) exprime l'écriture du programme vers le support.

Les supports se divisent en supports à *accès direct* ou sélectif (disques, disquettes, CD-ROM) et support à *accès séquentie*l (cartouches, streamers).

Si les fichiers

qui sont sur les supports à accès séquentiel ne peuvent avoir qu'une *organisation séquentielle* et une *méthode d'accès* également *séquentielle*,

ceux qui sont sur disque, peuvent être organisés indifféremment séquentiellement ou en direct.

Qui peut le plus, peut le moins : un fichier organisé en accès direct, pourra être lu en direct, mais aussi séquentiellement, du début à la fin, selon le problème à traiter.

On aura compris qu'un fichier, outre son nom, est caractérisé par :

◊ Son organisation : séquentielle, indexée, relative, binaire, ou table de base de donnée. C'est une caractéristique permanente.

◊ Sa méthode d'accès : séquentielle, indexée ou relative.
Cela dépend du type de traitement et du programmeur

4.2.2 Les Organisations

Organisation séquentielle

Les enregistrements du fichier sont rangés l'un derrière l'autre.
Un champ sert parfois *d'identifiant* (unique ou avec synonymes).
Le fichier est en désordre, ou trié (classé) selon l'ordre croissant (ou décroissant) d'un de ses champs.
On ne peut accéder à un enregistrement qu'après avoir lu ceux qui le précèdent.

Organisation indexée

Les enregistrements du fichier sont rangés l'un derrière l'autre souvent dans l'ordre de saisie (quelconque).

Un champ sert *toujours* d'*identifiant unique*. On l'appelle alors *clé*.
Dans certaines organisations, il peut y avoir *en plus* d'autres champs servant de *clés secondaires* et le plus souvent avec *synonymes*.

Une zone particulière du fichier, ou d'un fichier conjoint, appelé *index*, contient le répertoire des clés avec l'adresse de l'enregistrement qui lui correspond.

On peut donc accéder à un tel fichier, soit dans son ordre de saisie, soit à travers l'index.
Cette organisation est très fréquente sur les minis et mainframes et moins sur les micros.

Organisation relative

Les enregistrements du fichier sont rangés l'un derrière l'autre, la place qui leur est affectée l'ayant été par un programme, à un endroit bien précis.

Il n'y a pas d'index.

Il y a tout de même une clé : c'est le numéro d'ordre de l'enregistrement dans le fichier.

Cela implique que :

- un enregistrement ne change jamais de place
- Il y a donc des trous dans le fichier. Places qui ne correspondent à aucun enregistrement pour le moment.
- On ne peut bien sûr pas "tasser" un fichier qui a des trous
- Il faut avoir un moyen logique de retrouver le rang de l'enregistrement pour pouvoir y accéder.

Organisation binaire

Il n'y a qu'un seul enregistrement dans tout le fichier. C'est une suite d'octets non structurée. Il en est ainsi de tous les fichiers textes, images, acquisition de mesures, programmes exécutables.
On se promène dans ces fichiers de manière très particulière et ... artisanale.

Pas pour débutant !

Organisation Bases de données

Un ou plusieurs fichiers, selon les bases (Oracle, Ingres, Access, Paradox, Informix etc...)
Cette organisation est vue comme un ensemble de tables (des fichiers) qui sont composées d'enregistrements (des tuples) eux-mêmes composés d'attributs (les champs ou propriétés).
Chaque table possède ses clés.
On fait plutôt de l'accès direct, mais ... sur plusieurs tables à la fois!

4.2.3 Fichiers séquentiels

L'organisation séquentielle est faite en vue d'un traitement complet du fichier, c'est à dire que le programme va ouvrir le fichier pour en faire une lecture complète des enregistrements, du premier au dernier.

Il faut ouvrir le fichier avant toute action sur ses enregistrements. En général, l'ouverture peut se faire en vue :

- d'une lecture seule (mode **INPUT**),
- d'une écriture seule (mode **OUTPUT**) : on ne peut que rajouter de nouveaux enregistrements, en écrasant le contenu du fichier déjà existant,
- d'un ajout d'enregistrement (mode **EXTEND**) : on ne peut que rajouter de nouveaux enregistrements, en fin de fichier,
- ou enfin en vue de modifier des enregistrements (mode **I-O**).

L'ajout d'un enregistrement peut donc s'effectuer de deux manières : soit en fin de fichier, soit en écrasant le fichier déjà existant à l'ouverture.

Il faut fermer le fichier lorsque le travail est terminé.

4.2.4 Fichiers séquentiels indexés

L'organisation séquentielle indexée donne la possibilité d'accéder aux enregistrements d'un fichier de manière sélective ou séquentielle.

Chaque enregistrement d'un fichier indexé est individualisé par la valeur d'un indicatif (ou clé), à l'intérieur de cet enregistrement. Cette clé identifie de manière unique l'enregistrement et est associée à un index qui fournit un chemin logique vers l'enregistrement.

Il faut ouvrir le fichier avant toute action sur ses enregistrements. En général, l'ouverture peut se faire en vue :

- d'une lecture seule (mode **INPUT**),
- d'une écriture seule (mode **OUTPUT**) : on ne peut que rajouter de nouveaux enregistrements,
- ou enfin en vue de modifier ou de supprimer des enregistrements (mode **I-O**).

Il faut aussi fermer le fichier lorsque le travail est terminé.

La lecture peut s'effectuer de manière séquentielle (du début à la fin), ou pour un enregistrement précis, dont on a donné la valeur de la clé.

L'écriture s'effectue après avoir rempli les valeurs de l'enregistrement (et notamment la valeur de la clé).

La suppression s'effectue après avoir donné la valeur de la clé.

Enfin la modification s'effectue après avoir lu l'enregistrement, et modifié les valeurs de ses attributs (sauf la clé).

ANALYSE BIBLIOGRAPHIQUE

Etudiez l'utilisation des fichiers telle que définie au chapitre 2.4.2 de 'ALGORITHMIQUE' de l'AFPA.

EXERCICES

- **Fichier séquentiel de nombres conservés par ordre croissant.**

  Ecrire une procédure qui permet d'ajouter un nombre à un fichier séquentiel de nombres classés par ordre croissants, le fichier résultant étant lui aussi classé par ordre croissant.

- **Gestion d'un fichier de salariés.**

  Soit un fichier de salariés, où chaque fiche comporte un matricule (unique pour chaque salarié), un nom, et un salaire.

  Ecrire l'algorithme d'un programme qui permet de créer une nouvelle fiche, de détruire une fiche, de voir une fiche, et de lister l'ensemble des fiches, par numéro de fiche croissant.

  Le fonctionnement se fera en mode rouleau, sans souci de présentation de l'écran.

4.3 Ruptures - Appareillage de fichiers

Considérons le fichier des commandes constitué de la manière suivante :

Client	*N° commande*	*Code Produit*	*Quantité*	*Prix total*
Durant	98001	123	20	1 002,50
Durant	98001	988	1	544,00
.....	.....	.....	.....	.....
Durant	98001	344	20	22 176,20
Durant	98002	872	5	552,00
Durant	98002	123	12	623,30
.....	.....	.....	.....	.....
Durant	98002	222	20	66 762,00
Durant	98003	100	1	727,50
Durant	98003	121	10	7 798,00
Dupont	98005	665	13	4 267,50
Dupont	98005	333	1	334,00
.....	.....	.....	.....	.....
Dupont	98005	344	20	22 176,20
Dupont	98007	223	33	21 534,00
Dupont	98007	123	12	623,30
.....	.....	.....	.....	.....
Dupont	98007	323	10	1 862,00

Ce fichier est trié de la manière suivante :
Pour un client donné, on a la liste de toutes les commandes qui le concernent (triées par numéro croissant). Pour chacune de ces commandes on a l'ensemble des produits concernés par cette commande. On a ce que l'on appelle plusieurs niveaux de rupture :

- une rupture principale sur le client
- une rupture secondaire sur le n° de commande
- une rupture tertiaire sur le code produit.

Ces notions de rupture sont très importantes, car on est souvent amené, au cours de la lecture séquentielle d'un tel fichier, à effectuer des traitements particuliers en début de rupture ou en fin de rupture (par exemple, calcul des sous-totaux par N° de commande, et des totaux par client).

4.3.1 EXERCICES

- **Calculs de sous-totaux.**
  Sur le fichier séquentiel ci-dessus, écrire l'algorithme qui calcule les sous-totaux (prix) par numéro de commande, et les totaux par client, et qui affiche ces totaux sous la forme :

```
client1   N°Commande1   Ss total
client1   N°Commande2   Ss total
client1                 Total
client2   N°commande3   Ss total
client2   N°commande4   Ss total
client2                 Total
.......   .........     .....
```

On peut être aussi amené à traiter des programmes qui travaillent avec plusieurs fichiers à la fois, et dont le traitement de l'un des fichiers est conditionné par le résultat d'un traitement sur l'autre fichier. Considérons par exemple les deux fichiers suivants :

le fichier commandes :

Code Client	*N° commande*	*Code Produit*	*Quantité*	*Prix total*
154	98001	123	20	1 002,50
154	98001	988	1	544,00
.....	.....	.....	.....	.....
154	98001	344	20	22 176,20
154	98002	872	5	552,00
154	98002	123	12	623,30
.....	.....	.....	.....	.....

et le fichier client :

code Client	*Nom*	*adresse*	*N° tel*
.....	.....	.....	.....
123	Dupond	123 rue de la paix - 38000 Grenoble	04.76.76.76.76
124	Dupont	12 av. V. Hugo - 75002 Paris	01.82.82.82.82
.....	.....	.....	.....
154	Durant	24 Place Mozart - 34000 Montpellier	04.22.22.22.22
.....	.....	.....	.....

4.3.2 EXERCICES :

- **Appareillage de fichiers.**

  Sur les fichiers ci-dessus, écrire l'algorithme qui balaye le fichier commandes et qui va chercher le nom du client chaque fois que c'est nécessaire .

5 REGLES PRATIQUES DE PROGRAMMATION STRUCTUREE

5.1 Les variables

- **Donner des noms significatifs aux variables**

Le nom des variables doit être significatif, à chaque fois que cela est possible.
Cela permet de rendre le code plus lisible.
Cela évite les définitions multiples de variables (une variable globale x, une variable locale x) ce qui peut provoquer des erreurs à l'exécution et complique le débuggage.

- **Limiter le nombre de variables globales**

Une variable ne doit être déclarée globale que s'il est impossible de faire autrement. C'est le cas lorsque une valeur doit être conservée durant toute l'exécution du programme et qu'elle est utilisée par plusieurs fonctions et/ou procédures.
Ces variables ont toujours un rôle précis et il est donc toujours possible de leur donner un nom significatif.

5.2 Les fonctions

- **fonction ou procédure ?**

Chaque fois qu'il sera possible de faire une fonction, on fera une fonction.
En particulier, lorsque la procédure peut détecter une erreur (fichier non ouvert, enregistrement non trouvé, paramètres erronés, etc...) il faut faire une fonction qui retourne un code indiquant si l'opération s'est bien déroulée. De la sorte, on peut traiter l'erreur au niveau de l'appelant.

- les fonctions et procédures doivent avoir un nom significatif

Ceci a pour but de rendre plus lisibles les fonctions appelantes.

5.3 Les fichiers

- **ouvrir et fermer les fichiers**

Ces conseils sont valables pour des programmes destinés à fonctionner sur des systèmes d'exploitation peu performants en gestion de fichiers (ex: Dos, Windows, unix, OS2) et pour des fichiers séquentiels, directs ou séquentiels indexés purs (c'est-à-dire pas pour les SGBDR)
Les fichiers doivent rester ouverts le moins longtemps possible pour assurer un maximum de sécurité aux données. L'ouverture et la fermeture étant très rapides, il est inutile de s'en priver.
Pour ce faire, on a intérêt à faire des fonctions pour chaque opération sur les fichiers (lecture, écriture, ajout, suppression).
Dans ce cas, le fichier sera ouvert en début de fonction et fermé en fin de fonction.

- **consulter les enregistrements**

Quel que soit le langage, quel que soit le système d'exploitation et quelle que soit l'application, la consultation des données doit permettre à l'utilisateur :

  - de retrouver une information par son code
  - d'avoir une liste des informations existantes, lui permettant de retrouver un code (sous forme de liste déroulante, de tableau ou autre)
  - de changer d'enregistrement sans revenir au menu principal (ressaisie de la clé ou bouton suivant/précédent ou autre)

- **modifier les enregistrements**

Pour que l'utilisateur puisse modifier un enregistrement, il faudra d'abord que celui-ci soit affiché.
Un écran de modification ne doit jamais permettre que l'on modifie la clé de l'enregistrement.
La modification doit être (sauf exception rare) une action délibérée de l'utilisateur, c'est-à-dire qu'elle ne sera effective que lorsqu'il appuiera sur un bouton (ou une touche de fonction) ou après validation.

- **créer des enregistrements**

Un écran de création doit toujours obliger l'utilisateur à saisir la clé de l'enregistrement.
Le programme doit alors obligatoirement vérifier que la clé n'existe pas déjà, et interdire la création si elle existe.
Attention à mettre à jour dans ce cas, la liste des enregistrements éventuellement affichée quelque part (liste déroulante notamment)

- **supprimer des enregistrements**

Lorsqu'un utilisateur demande la suppression d'un enregistrement, l'enregistrement doit toujours être affiché et le programme doit demander une confirmation « êtes-vous sûr de vouloir supprimer l'enregistrement de clé ... ?». De la sorte, l'action peut être annulée.
Attention à mettre à jour dans ce cas, la liste des enregistrements éventuellement affichée quelque part (liste déroulante notamment)

Suppression logique :
Dans le cas de fichier séquentiel ou direct, il est possible de n'effectuer qu'une suppression logique. Cela évite de restructurer le fichier et cela permet éventuellement de récupérer des enregistrements supprimés.
Elle consiste à ajouter un champ booléen aux enregistrements indiquant s'ils sont supprimés ou effectifs.

Etablissement référent
Direction de l'ingénierie Neuilly
Centre afpa - Grenoble

Equipe de conception

Groupe d'étude de la filière étude - développement

Remerciements :

Date de mise à jour 14/04/2014
afpa © Date de dépôt légal avril 14