Algorithmique – 1
Notions fondamentales

Bertrand LIAUDET

SOMMAIRE

Remarque : tout ce qui est précédé par « ++ » est à bien savoir !!!

Edition octobre 2017

INTRODUCTION

1. Algorithme et algorithmique en général

Algorithmique

Nom commun : science des algorithmes. Etude des objets et des méthodes permettant de produire des algorithmes.

Adjectif : relatif aux algorithmes.

Algorithme

Définition 1

Texte décrivant sans ambiguïté une méthode pour résoudre un problème.

Définition 2

Texte décrivant sans ambiguïté une suite ordonnée d'actions (ou opérations ou traitements) à effectuer pour arriver au(x) résultat(s) attendu(s). En général, certains éléments sont fournis au départ pour effectuer les opérations.

Bilan

Un algorithme est une recette !

Domaine d'application

Un algorithme peut s'appliquer à n'importe quel domaine.

Mathématique

Algorithme du calcul du PGCD de deux nombres (Euclide au 3ème siècle avant JC).

Eléments de départ : 2 entiers

Résultat : le PGCD

Opérations : des opérations mathématiques.

Informatique

Algorithme d'affichage d'un fond d'écran animé.

Elément de départ : aucun.

Résultat : l'affichage du fond d'écran.

Opération : des opérations mathématiques et informatiques.

Cuisine

Algorithme (recette) du gâteau au chocolat

Recette du gâteau au chocolat

Séparer le blanc des jaunes.

Mélanger le sucre aux jaunes.

Etc.

Faire cuire

Fin

Eléments de départ : les ingrédients.

Résultat : le gâteau.

Opérations : les opérations de cuisine.

Le document de montage d'un meuble en kit

Eléments de départ : les planches, les vis, etc.

Résultat : le meuble (une armoire, par exemple).

Opérations : les opérations de montage.

REMARQUE :

C'est plus ou moins bien fait ! Ca vaut pour la description d'une recette comme pour les algorithmes informatiques !

Notion d'entrée, de sortie et d'instruction

Les actions décrites dans un algorithme manipulent des objets fournis au départ pour permettre de produire le ou les résultats.

- Les ENTREES sont les éléments de départ.
- Les SORTIES sont les résultats produits.
- Les INSTRUCTIONS sont les actions qui sont décrites dans l'algorithme.

Distinction entre les types d'algorithme

Trois éléments distinguent les différents types d'algorithme :

- Le type des SORTIES : produire un gâteau ou produire un nombre.
- Le type des ENTREES : des œufs ou des tableaux de nombres.
- Le type des INSTRUCTIONS : mélanger les œufs et le sucre ou comparer la valeur de deux données.

2. Algorithme informatique

Informatique

L'informatique est la science du traitement automatique de l'information. Les anglo-saxons parlent de « computeur science ».

Est informatique ce qui est relatif à cette science.

Algorithme informatique

Les algorithmes informatiques sont ceux qui décrivent des traitements automatisées d'informations qui ont lieu dans des ordinateurs.

++ Ecran et clavier

L'écran et le clavier d'un ordinateur sont les deux périphériques de base qui permettront de fournir des entrées et d'afficher les résultats.

Toutefois, les entrées peuvent provenir de divers périphériques ou de fichiers. De même, les sorties peuvent être des fichiers ou des informations pour divers périphériques.

3. Langages algorithmiques

Les 3 sortes de langages algorithmiques

- Les langages textuels.
- Les langages graphiques (organigrammes, diagramme d'activités).
- Les langages symboliques.

Ces trois sortes de langage ont les mêmes objectifs et ont la même sémantique. La différence se situe uniquement au niveau de la présentation (graphie et syntaxe).

Les langages textuels (pseudo-code)

Ce sont des langages qui décrivent l'algorithme :

- avec un langage alphabétique
- en écrivant les instructions les unes en dessous des autres
- en utilisant un principe d'indentation.

De ce fait, il y a une représentation spatiale à deux dimensions de l'algorithme.

Les langages textuels sont les langages utilisés le plus communément pour écrire des algorithmes : on parle aussi de **pseudo-code**.

Les langages graphiques (organigrammes, diagramme d'activités)

Ce sont des langages qui utilisent des symboles graphiques à deux dimensions pour écrire l'algorithme.

On retrouve les organigrammes avec les diagrammes d'activités d'UML.

Les langages graphiques (organigrammes) ne sont plus utilisés pour décrire précisément les algorithmes informatiques de type procédure ou fonction car le formalisme prend trop de place. On les utilise généralement pour des descriptions plus générales (par exemple pour décrire le déroulement d'un cas d'utilisation avec ses différents scénarios ou décrire l'organisation générale d'une entreprise).

Les langages symboliques

Ce sont des langages qui décrivent l'algorithme :

- avec un langage symbolique
- en écrivant les instructions les unes à la suite des autres sur la même ligne
- donc, sans utiliser de principe d'indentation.

Les langages symboliques ne sont utilisés que par quelques théoriciens. Leur principal intérêt est de pouvoir écrire très rapidement des algorithmes sur un ticket de métro !

Les différentes approches du pseudo-code (des langages textuels)

Il y a plusieurs approches dans l'usage du pseudo-code :

- Pseudo-code avec déclaration ou non des types.
- Pseudo-code avec déclaration ou non des variables locales.
- Pseudo-code avec usage ou non de débranchements structurés.
- Pseudo-code avec duplication ou non des débuts de bloc.
- Pseudo-code avec marquage des blocs ou non (numérotation par exemple)

FORMALISME CHOISI

Le choix de ce cours suivra le principe suivant : **RIEN DE TROP !** L'écriture algorithmique se limitera au strict nécessaire tout en s'assurant de la clarté de l'algorithme.

Ce qui conduit à choisir les options suivantes :

- Pas de déclaration des types des paramètres formels grâce à l'utilisation d'une convention de nommage en fonction des types.
- Usage des flèches pour préciser le mode de passage des paramètres formels.
- Commentaire facultatif de l'en-tête (sous l'en-tête, avant le corps), précisant la signification des variables et leurs types
- En cas de fonction, précision du type renvoyé en fin d'en-tête avec commentaire précisant la signification de ce qui est renvoyé.
- Déclaration facultatives des variables locales dans les procédures et les fonctions.
- Usage de débranchements structurés.
- Pas de duplication des débuts de bloc.
- Marquage des blocs par numérotation de type numérotation de paragraphe facultatif, mais à savoir faire !

4. ++ Les paradigmes de programmation, d'après Bjarnes Stroustrup (C++)

Paradigme 0 : pdg. du débutant : « Tout dans le main »

Codez tout dans le programme principal.
Faîtes des tests jusqu'à ce que ça marche !

Ce paradigme est celui du débutant. C'est ce qu'on fait quand on commence la programmation.

Paradigme 1 : pdg. de la programmation procédurale

Choisissez les procédures (=fonctions).
Utiliser les meilleurs algorithmes que vous pourrez trouver.

Le problème est celui du bon découpage du main en procédures.

Il faut définir les entrées et les sorties pour chaque procédure.

Un principe général est de distinguer entre l'interface utilisateur (la saisie et l'affichage) et le calcul.

Paradigme 2 : pdg. de la programmation modulaire : masquer l'information

Choisissez vos modules (fichiers avec des fonctions et regroupements de fichiers)
Découpez le programme de telle sorte
que les données soient masquées par les modules

Le paradigme de la programmation modulaire est un aboutissement de la programmation procédurale.

Il consiste à regrouper les fonctions dans des fichiers et à organiser ces fichiers en modules (un module est un regroupement de fichiers) qui permettent de masquer les données du programme.

Techniquement, il conduit à l'utilisation de variables globales, de static, d'extern, de compilation conditionnelle, d'espace des noms, d'organisation du type : outils.c, outils.h (interface), utilisation.c.

La partie « masquage de l'information » du paradigme est rendue obsolète par l'usage de la programmation objet. La partie « organisation des fichiers » reste présente.

Paradigme 3 : pdg. de l'abstraction des données : type structurés = classe

Choisissez les types dont vous avez besoin.
Fournissez un ensemble complet d'opérations pour chaque type.

Une classe est un type, en général structuré, auquel on ajoute des procédures.

Fabriquer un type structuré consiste à regrouper des types (simples ou déjà structurés) dans un même type structuré : c'est déjà un mécanisme d'abstraction.

Une classe en tant que type qui peut donner lieu à la fabrication d'une variable (d'un objet) est un type est tout à fait concret (et non pas abstrait » : il est aussi réel que int et float.

Principe d'encapsulation

Le principe d'encapsulation est le principe de fonctionnement du paradigme d'abstraction.

Y sont associées les notions de : **visibilité** des attributs et des méthodes, **constructeur**, **destructeur**, **surcharge**.

Paradigme 4 : pdg. de l'héritage

Choisissez vos classes.
Fournissez un ensemble complet d'opérations pour chaque classe.
Rendez toute similitude explicite à l'aide de l'héritage.

Principe de substitution

Le principe de substitution est le principe de fonctionnement de l'héritage.

Y sont associées les notions de : **polymorphisme**, **redéfinition**, **interface**, **implémentation**, **classe et méthode abstraites**.

Paradigme 5 : pdg. de la généricité (type variable)

Choisissez vos algorithmes.
Paramétrez-les de façon qu'ils soient en mesure de fonctionner
avec une variété de types et de structures de donnée.

La généricité au sens stricte consiste à ce que le type des paramètres des algorithmes devienne un paramètre lui aussi. Elle s'appuie souvent sur l'utilisation d'interfaces.

Paradigme final : pdg. de la productivité : factoriser encore plus le code

- **Les interfaces :** elles permettent d'élargir la possibilité de coder de la généricité stricte et sont largement utilisées dans les design patterns. Une interface est un type abstrait (tandis que la classe est un type concret). Ce type abstrait est utilisé de façon générique dans le code. Seule l'instanciation concrète différencie ensuite les comportements.
- **Les design patterns :** ce sont des solutions classiques à des petits problèmes de codage. Ils s'appuient souvent sur les interfaces.
- **Les patterns d'architecture :** ce sont des solutions classiques d'architecture. Le MVC est un pattern d'architecture.
- **Les bibliothèques :** elles offrent des méthodes permettant d'éviter de les réécrire et organisant même la façon de réfléchir à la solution des problèmes à résoudre.
- **Les framework :** ce sont des architecture semi-finies qu'on peut va ensuite paramétrer et compléter en fonction des spécificités du produit à réaliser.

NOTIONS FONDAMENTALES

1. Programme, afficher, instruction, bloc, indentation

Premier exemple

```
Programme AfficherBonjour
    ecrire (« Bonjour ») ;
Fin
```

Mots-clés

Les mots-clés sont les mots constants du langage algorithmique. Dans l'exemple proposé, ils sont soulignés. Dans la pratique, on ne les souligne pas. On a ici 3 mots-clés : Programme, Ecrire et Fin.

Programme

Un programme est un algorithme. On lui donne un nom (« afficherBonjour «)

Ecrire

Il n'y a qu'une seule instruction : l'instruction ecrire( ) (ou afficher() ou out( )).

Ecrire est une instruction du langage algorithmique. C'est l'instruction qui permet d'écrire quelque chose à l'écran.

Instruction

« ecrire » est une instruction. En général, et écrit les instructions les unes en dessous des autres et on termine chaque instruction par un point-virgule. C'est toutefois facultatif.

Notion de bloc

L'algorithme commence avec « Programme » et finit avec « Fin ». Tout ce qui se trouve entre « Programme » et « Fin » constitue un bloc d'instructions, c'est-à-dire un ensemble d'instructions. « Programme » marque le début du bloc. « Fin » marque la fin du bloc.

Indentation : question de présentation

Le bloc d'instructions est décalé (d'une tabulation) par rapport au début du bloc qui la contient. La fin du bloc est mise juste en dessous du début du bloc.

Programme AfficherBonjour

La première ligne consiste à donner un nom au programme juste après le mot-clé : Programme.

Forme générale d'un programme

Programme *NomDuProgramme*

    Instructions ;

Fin

2. Lire, variable, affectation, expression, évaluation

Deuxième exemple

```
Programme Fahrenheit
    lire (celsius)
    fahr ← celsius * 9 / 5 + 32
    ecrire (fahr) ;
Fin
```

Variables

On a deux variables dans l'algorithme : Fahr et Celsius.

Affectation

« fahr ← 9 / 5 celsius + 32 ; » est une affectation.

Une affectation comporte trois parties :

- A gauche : le nom d'une variable
- Au milieu : le symbole d'affectation : « ← ». On utilise aussi le symbole « := » ou aussi « = » mais ce dernier ne doit alors pas être confondu avec le l'opérateur booléen d'égalité.
- A droite : une expression mathématique à évaluer

Une affectation est une instruction. C'est l'instruction de base de toute la programmation.

L'affectation est l'instruction qui permet de donner la valeur de l'expression de droite à la variable de gauche (appelée parfois « left value »).

Expression et évaluation

Les algorithmes contiennent souvent des expressions mathématiques. Ici : « Celsius * 9 / 5 + 3 »

Ces expressions sont évaluées selon les règles d'évaluation classiques des mathématiques.

L'évaluation d'une expression retourne une valeur qui sera utilisée dans l'instruction où l'expression se l'expression se trouve.

++ Affectation et évaluation

Evaluation et Affectation sont les deux actions élémentaires fondamentales de tout algorithme.

Evaluer consiste à trouver la valeur d'une expression.

Affecter consiste à mettre une valeur dans une variable.

++ Lire

La fonction de lecture permet de récupérer la valeur d'une variable

A cette fonction, on passe la ou les variables à lire.

Les valeurs saisies au clavier sont affectées dans les variables de la fonction lecture.

```
lire (Celsius)
```

Est équivalent à :

```
celsius ← lire ( )
```

Est équivalent à :

```
celsius ← Clavier
```

On affecte le clavier (ce qui est saisi au clavier) dans la variable Celsius.

On peut passer plusieurs paramètres à la fonction de lecture :

```
lire (a, b, c)
```

++ Ecrire

Pour afficher la valeur d'une variable, on la passe en paramètre à la fonction afficher. A noter qu'il n'y a plus de guillemets : on utilisait les guillemets pour distinguer les textes qu'on veut afficher des variables qu'on veut aussi afficher.

```
ecrire (fahr) ;
```

Est équivalent à :

```
Ecran ← fahr
```

On affecte l'écran (on affiche à l'écran) la valeur de la variable Fahr.

++ Forme générale d'un programme

```
Programme
        Lecture
        Traitement
        Affichage
Fin
```

Cette forme est très importante à retenir.

++ Circulation de l'information

Un programme consiste en une circulation d'information.

Notre algorithme peut être décrit ainsi :

- L'utilisateur choisit une valeur dans sa tête
- Cette information est transmise à ses mains
- Ses mains transmettent l'information au clavier
- Le clavier transmet l'information à la variable « Celsius »
- La variable « Celsius » transmet l'information à la variable « Fahr »
- La variable « Fahr » transmet l'information à l'écran
- L'écran transmet l'information aux yeux de l'utilisateur
- Les yeux de l'utilisateur transmettent l'information au cerveau de l'utilisateur.

Retour sur la notion d'instruction

++ Toute instruction peut se ramener à une ou plusieurs affectations.

- instructions-affectations qui permettent de récupérer de l'information de l'extérieur : « lire ».
- instructions-affectations qui permettent d'envoyer de l'information à l'extérieur : « écrire ».

- instructions-affectations qui permettent de modifier les informations dans le programme : « affectation » et « appel de procédures » (les appels de procédure seront abordés un peu plus tard).

Bilan du vocabulaire d'un algorithme

Dans un algorithme on trouve :

- Des mots-clés (noms fixes) : Programme, Afficher, Lire, ←
- Des noms variables : Fahrenheit, Celsius, Fahr, etc.
- Des valeurs : 9, 32, « bonjour », etc.
- Des expressions : 9 / 5 celsius + 32, etc. Une expression est constituée de valeurs, de variables et d'opérateurs (+, /, etc.).

++ Vérification et simulation

Principe

Pour vérifier un algorithme, une méthode consiste à simuler son exécution et à regarder le contenu des variables après chaque instruction.

Mise en œuvre

Pour mettre en œuvre la simulation, il faut faire un tableau. Chaque colonne contient une variable. Chaque ligne correspond à une instruction. On remplit au fur et à mesure la valeur des variables dans le tableau.

Exemple

➢ ***Algorithme***

```
Programme Fahrenheit
    lire (celsius)
    fahr ← celsius * 9 / 5 + 32
    ecrire (fahr) ;
Fin
```

➢ ***Simulation***

	Celsius	Fahr	Ecran
lire (celsius)	20		
fahr ← celsius * 9 / 5 + 32		68	
ecrire (fahr) ;	-17,6		68

3. Variables, types et expression

Distinction entre variable mathématique et variable informatique

En mathématique, la variable est un symbole, généralement une lettre, défini de telle sorte que ce symbole peut être remplacé par **0, 1 ou plusieurs valeurs**. Les valeurs possibles d'une variable dans une situation donnée (x dans une équation) ne sont **pas forcément connues**. Les exercices d'algèbres consistent souvent à mettre au jour les valeurs possibles pour les variables. L'ensemble des valeurs possible pour une variable n'est **pas modifiable**.

En informatique, une variable a toujours **1 et 1 seule valeur**. Cette valeur est toujours **connue**. Cette valeur **peut être modifiée**.

++ Description d'une variable informatique

++ Caractéristiques d'une variable

- un nom : celsius : c'est l'identificateur de la variable. Le nom peut faire référence à la valeur ou au contenant selon son usage.
- une valeur (ou contenu): 3 : c'est la traduction en fonction du type de l'état physique réel de la variable.
- un contenant : c'est le bout de mémoire dans lequel la valeur est stockée. Il est symbolise par un carré.
- une adresse : ad_celsius, c'est la localisation du contenant.
- un type : Il permet de définir l'ensemble des valeurs possibles pour la variable (entier, réel, caractère, booléen, etc.).
- une signification (ou sens) : par exemple, celsius c'est la température en °C. La signification est conventionnelle (c'est le programmeur qui la choisit). Mais elle est très importante.

Représentation schématique des variables

Il faut se représenter clairement ce qu'est une variable C'est quelque chose où je peux mettre une information (un tiroir, une page blanche, une ardoise, un contenant). Cette chose est fixe : elle a une position géographique (c'est son adresse : ad_n) et un nom (n) qui me facilite la vie (c'est pour cela qu'il vaut mieux qu'il soit significatif, sinon on pourrait la nommer par son adresse!). Le nom, l'adresse et le contenant ne sont pas modifiables.

Une variable a aussi un contenu : c'est sa valeur, c'est-à-dire l'information qu'elle contient. On peut modifier la valeur. Une variable a toujours un contenu.

Du bon usage : bien nommer les variables

Il faut nommer significativement les variables !

En effet, dans le programme précédent, on pourrait appeler fahr et celsius x et y, ou même celsius et fahr ! Comme ça, on aurait toutes les chances de ne plus rien y comprendre ! Pensez à la relecture!

++ Double usage du nom d'une variable

Le nom d'une variable fait référence soit la valeur, soit le contenant.

- Quand le **nom** "celsius" est utilisé dans une expression à évaluer, "celsius" c'est la **valeur** de "celsius". "celsius" est utilisé en entrée de l'affectation : il est utilisé mais pas modifié.
- Quand le **nom** fahr est utilisé à gauche de l'affectation. fahr c'est le **contenant** de fahr. On parle aussi de "lvalue" ("left value", "valeur à gauche" de l'affectation, ou "valeur_g"). fahr est utilisé en

++ Les 4 + 1 types élémentaires

Le type permet de définir l'ensemble des valeurs possibles pour la variable.

Il y a **4 types élémentaires :**

- Entier : domaine de définition : IN
- Réel : domaine de définition : IR
- Caractère : domaine de définition : l'alphabet, les chiffres, la ponctuation, les symboles, etc. Tout ce qu'on veut entre apostrophes. 'a' est un caractère. ' »' est un caractère. 'F12' est un caractère (celui correspondant à la touche F12 du clavier), etc.
- Booléen : domaine de définition : vrai et faux.

Auxquels on peut ajouter **un pseudo type élémentaire** :

- Chaîne de caractères

Type et convention algorithmique

Dans les algorithmes, on précisera le type que quand ce sera nécessaire.

On choisira les noms de variables de telle sorte qu'ils correspondent au type.

Voici les noms habituellement associés aux types élémentaires :

- Entier : i, j, k, n, m
- Réel: x, y, z
- Caractère : c, c1, c2
- Chaîne de caractères : ch, ch1, ch2
- Booléen : flag, fg, fg1, drapeau, dp, dp1, bool, ok
  et les symboles  pour un drapeau à vrai et pour un drapeau à faux.

++ Expression et évaluation

Constituants d'une expression

Une expression est constituée par :

- Des constantes littérales : 3, -4.5, 'C', « bonjour »
- Des noms de variables
- Des opérateurs : +, -, *, /, etc.
- Des noms de fonctions : sin( ), cos( ), racine( )
- Des parenthèses

➢ ***Remarques :***

On ne doit pas utiliser le symbole de mise au carré dans une expression algorithmique (5^2), mais soit faire l'opération (5*5), soit se doter d'une fonction qui fait l'opération : carré(5).

De même, on ne doit pas écrire : $\frac{4*x}{5+y}$ mais (4 * x) / (5+y)

Dans tous les cas, les expressions doivent être ramenées à une expression sur une seule ligne en utilisant des opérateurs standards, des fonctions et des parenthèses.

Evaluation d'une expression

Pour trouver la valeur d'une expression, on évalue l'expression, ce qui consiste à faire les opérations contenue dans l'expression en suivant l'ordre de priorité des opérateurs, des fonctions et des parenthèses.

Valeur et type d'une expression

Une expression à une valeur qui est d'un certain type parmi les types élémentaire.

On verra que les expressions peuvent aussi avoir un type structuré (ce qui sera d'autant plus vrai avec l'utilisation de la « surcharge » des opérateurs).

++ L'affectation

L'affectation ou assignation est l'opération qui permet de donner une valeur à une variable.

« x ← 3 » veut dire que x prend la valeur 3 (on met 3 dans x).

« x ← 3 * y » veut dire qu'on met le résultat de l'évaluation de 3 * y dans x.

« x ← x + 1 » veut dire qu'on met le résultat de l'évaluation de x * 1 dans x, autrement dit, x est incrémenté de 1.

« 5 ← x + 1 » ne veut rien dire ! On ne peut pas mettre x+1 dans 5 !

Exercices – Série 1 - Affectation

Exercice 1

Ecrire programme qui calcule le double d'un entier.

Exercice 2

Ecrire programme qui calcule le carré d'un réel.

Exercice 3

Écrire un programme qui transforme des degrés Celsius en degrés Fahrenheit. Le zéro Celsius correspond à 32 °F; 100 °C égale 212 °F.

Exercice 4

Écrire un programme qui calcule le prix TTC.

Exercice 5

Écrire un programme qui calcule la circonférence d'un cercle et l'aire du disque délimité par ce cercle.

Exercice 6

Trouvez ce que fait le programme suivant grâce à un exemple, puis prouvez-le.

```
Programme exo4
    lire(a, b)
    a ← b-a
    b ← b-a
    a ← a+b
    afficher (a, b)
Fin
```

Exercice 7

Trouvez ce que fait le programme suivant grâce à un exemple, puis prouvez-le.

```
Programme exo4
    lire(a, b, c)
    a ← a + b + c ;
    b ← b + c ;
    c ← a - c ;
    a ← a - c ;
    b ← c - b + a;
    c ← c - b;
    afficher (a, b, c)
Fin
```

4. ++ Test

Troisième exemple

```
Programme racineCarré
        Lire (x)
        si x  < 0 alors
                afficher(« Pas de racine »)
        sinon
                racine = racineCarrée(x)
                afficher (racine)
        finsi
Fin
```

si – sinon – finsi

Pour pouvoir effectuer une instruction sous condition, on utilise le « si ».

Si une condition est réalisée, alors, on exécute la série d'instructions du bloc « si ». Le début du bloc « si » est marqué par le « si », la fin du bloc « si » est marquée par le « sinon », ou bien, s'il n'y a pas de « sinon », par le « finsi ».

Le sinon correspond à l'alternative du si : ici le cas sinon correspond à x >= 0.

Le finsi vient fermer le bloc ouvert par le sinon.

Evaluation de la condition

La condition (x <0 dans l'exemple) est une expression de type booléen qui est évaluée. Elle vaut soit VRAI, soit FAUX. Si elle vaut vrai, c'est le bloc « si » qui sera exécuté. Si elle vaut faux, c'est le bloc « sinon ». A noter que FAUX est équivalent à 0 et VRAI à tout sauf 0.

Bloc d'instructions

Un « si » ouvre un bloc d'instructions : les instructions dans le bloc sont donc décalées d'une tabulation par rapport au « si ».

Le « alors » est facultatif.

Le « sinon » ferme le bloc d'instructions du « si » : il est donc placé sous le « si ».

Le « sinon » ouvre aussi un nouveau bloc d'instructions : les instructions du cas « sinon » sont donc décalées d'une tabulation par rapport au « sinon ».

Le « finsi » ferme le bloc d'instruction du « sinon » ou celui du « si » quand il n'y a pas de « sinon ». Il est donc placé sous le « sinon » ou sous le « si ».

++ Principe d'imbrication

Dans un bloc instruction, on peut mettre n'importe quel type d'instruction. On peut donc imbriquer les instructions les unes dans les autres, les tests dans les tests, dans les boucles, etc.

En cas d'imbrications multiples et d'algorithme long, on peut numéroter les débuts et fin des blocs d'instructions dans une logique de paragraphe : B1, B1.1, B1.2, B1.1.1, etc.

On peut aussi tirer un trait entre le début et la fin du bloc.

Fonction racine

Dans l'exercice, il ne s'agit pas de trouver une méthode pour calculer la racine carrée. On peut donc utiliser la fonction racineCarrée. C'est un principe général. On peut se donner toutes les fonctions mathématiques dont on a besoin, sauf quand l'exercice précise qu'il faut la fabriquer.

Remarque sur l'écriture algébrique dans les algorithmes

Dans les algorithmes, on n'utilise jamais de barres de fraction, ni de symboles comme racine carrée, carré, ou autre.

Les expressions algébriques doivent être formées avec les 4 opérateurs de base : +, -, *, / et avec les parenthèses en respectant l'ordre de priorité des opérateurs, et avec des fonctions du genre de racineCarrée(x), carré(x), puissance(x, n), log(x), ln(x), factoriel(n), etc.

Remarque sur la structure générale du programme

On retrouve le « lire » en premier.

Ensuite, traitements et affichage sont un peu mixés.

Malgré cela, la structure : « lire, traiter, afficher » reste pertinente.

++ Les 3 différents types de tests

si – sinon - finsi

```
si condition alors
        bloc1
sinon
        bloc2
finsi
```

Le sinon est facultatif :

```
si condition alors
        bloc1
finsi
```

si – sinonsi – etc. – sinon - finsi

```
si condition alors
        bloc1
sinonsi
        bloc2
sinonsi
        bloc3
…
sinon
        bloc4
finsi
```

Le sinon est facultatif.

Cas où – cas1 - cas2 … - défaut - fincas

```
cas où variable vaut
cas 1 : expression ou liste d'expression
        bloc1
cas 2 : expression ou liste d'expression
        bloc2
…
défaut
        blocDefaut
finsi
```

L'instruction « cas où » fait une comparaison d'égalité entre la variable et l'expression ou une liste d'expressions.

Le « défaut » est facultatif.

++ Expression booléenne

La condition est une expression booléenne.

Une expression booléenne est une comparaison (égalité ou inégalité). « a >0 » est une expression booléenne dont l'évaluation renverra « vrai » ou « faux » selon la valeur de « a ».

On peut aussi utiliser les opérateurs ET et OU dans l'expression. « a > 0 ET b < 3 ». L'évaluation de cette expression renvoie à la table de vérité de l'opérateur ET.

++ Méthode d'analyse d'un algorithme avec tests : l'arbre dichotomique

Toute situation avec des tests pourra être analysée en construisant un arbre de décision dichotomique.

++ Vérification et simulation

Principe

Pour vérifier un algorithme, une méthode consiste à simuler son exécution et à regarder le contenu des variables après chaque instruction.

Mise en œuvre

Pour mettre en œuvre la simulation, il faut faire un tableau. Chaque colonne contient une variable. Chaque ligne correspond à une instruction. On remplit au fur et à mesure la valeur des variables dans le tableau.

Exemple

- ***Algorithme***

```
Programme RacineCarré
    Lire (x)
    si x  < 0 alors
        afficher(« Pas de racine »)
    sinon
        racine = racineCarrée(x)
        afficher (racine)
    finsi
Fin
```

- ***Simulation***

	x	x < 0	racineCarrée(x)	racine	écran
Lire (x)	4				
x < 0		faux			
racineCarrée(x)			2		
racine = racineCarrée(x)				2	
Afficher (racine)					2

- ***Simulation***

	x	x < 0	écran
Lire (x)	-4		
x < 0		vrai	
Afficher(« Pas de racine »)			Pas de racine

Exercices – Série 2 - Tests

Exercice 1

Écrire un programme qui calcule la valeur absolue de la différence de deux nombres.

Exercice 2

Écrire un programme qui dit si un entier est pair ou pas.

Exercice 3

Écrire un programme qui calcule le maximum de deux nombres.

Exercice 4

Écrire un programme qui calcule le maximum de trois nombres.

Exercice 5

Écrire un programme qui inverse les valeurs de deux variables

Exercice 6

Écrire un programme qui permet de redéfinir 3 variables, X, Y, Z de façon à obtenir : X < Y < Z

Exercice 7

Ecrire une fonction qui permet de calculer le prix d'un certain nombre de photocopies en appliquant le tarif suivant : les 10 premières coûtent 0,10 E pièce, les 20 suivantes 0,08 E pièce et toutes les autres 0,07 E pièce.

Exercice 8

Ecrire un programme qui affiche le nombre de jours d'un mois d'une année donnée. On rappelle que les années bissextiles sont celles qui sont divisibles par 4. Toutefois, les années divisibles par 100 ne sont pas bissextiles. Mais les années divisibles par 400 sont quand même bissextiles. Par exemple : 2008 est bissextile, 2010 n'est pas bissextile, 2000 est bissextile, 1900 n'est pas bissextile.

Exercice 9 (branché maths !)

Écrire un programme qui résout dans IR une équation du premier degré en traitant tous les cas possibles (si a=0, etc.)

Exercice 10 (branché maths !)

Écrire un programme qui résout dans IR une équation du second degré en traitant tous les cas possibles (si a=0, etc.)

5. ++ Boucle

Quatrième exemple

On veut afficher tableau de conversion des fahrenheits en celsius pour des fahrenheits allant de 0 à 300 de 20 en 20.

```
Programme listeFahr
    fahr ← 0
    Tant que fahr < = 300
        celsius ←  5 * (fahr – 32) / 9
        afficher (fahr, celsius)
        fahr ← fahr + 20
    Fin tant que
Fin
```

Boucle tant que

La boucle tant que ouvre un bloc d'instructions.

Ce bloc d'instructions est répété tant que la condition est vérifiée.

++ Les 4 différents types de boucle

Boucle « Tant que »

```
Tant que condition
    Instructions
Fin tant que
```

Boucle « Répéter »

Il y a deux types de boucle « Répéter » :

- ***avec tant que final***

```
Répéter
    Instructions
Tant que condition
```

- ***avec jusqu'à ce que final***

```
Répéter
    Instructions
Jusqu'à ce que condition
```

Boucle avec compteur : pour i de 1 à n par pas de 1

```
Pour i de 1 à N / pas
        Instructions
Fin pour
```

Par défaut le pas vaut 1 et n'est pas écrit.

Boucle sans fin

```
Boucle
        Instructions
Fin de boucle
```

++ Les débranchements de boucle : débranchements structurés

Break

Le break permet de quitter la boucle.

Next

Le suivant permet de sauter à la fin de la boucle mais sans quitter la boucle.

Notion de rupture de séquence

Les débranchements sont des ruptures de séquences.

++ Le goto : débranchement non-structuré : INTERDIT !!!

Le GOTO

Le GOTO est une rupture de séquence qui permet de sauter le déroulement continu des instructions (la séquence) en allant directement à la borne précisée par le goto.

Le GOTO permet d'aller n'importe où : on n'utilisera jamais le GOTO.

CF : « GO TO statement considered harmful » de Dijkstra (1968).

Notion de rupture de séquence

Les tests et les boucles sont des ruptures de séquences : les instructions ne vont pas s'exécuter l'une après l'autre.

Le test fait sauter en avant.

La boucle fait sauter en arrière, et éventuellement en avant.

Le test et la boucle sont des GOTO structurés.

Traduction d'une boucle tant que avec un GOTO structuré

```
Programme listeFahr
        Fahr ← 0
        borne tantque
```

```
        si fahr > 300
                goto borne fintantque
        finsi
        Celsius ← 5 * (fahr – 32) / 9
        Afficher (fahr, celsius)
        Fahr ← fahr + 20
        goto borne tantque
        borne fintantque
Fin
```

++ Méthode d'analyse d'un algorithme avec boucle

- Dès qu'un traitement doit être répété, on fera appel à une boucle.
- La boucle de base est la boucle tant que. Toute répétition doit donc être traitée en première hypothèse par un tant que.
- Si la première occurrence du traitement est nécessairement effectuée, on aura intérêt à utiliser un « répéter ».
- Si on sait combien de répétitions on doit effectuer, on utilisera une boucle « pour ».

++ Vérification et simulation

Principe

Pour vérifier un algorithme, une méthode consiste à simuler son exécution et à regarder le contenu des variables après chaque instruction.

Mise en œuvre

Pour mettre en œuvre la simulation, il faut faire un tableau. Chaque colonne contient une variable. Chaque ligne correspond à une instruction. On remplit au fur et à mesure la valeur des variables dans le tableau.

Exemple

➢ ***Algorithme***

```
Programme listeFahr
        Fahr ← 0
        Tant que fahr < = 300
                Celsius ← 5 * (fahr – 32) / 9
                Afficher (fahr, celsius)
                Fahr ← fahr + 20
        Fin tant que
Fin
```

- ***Simulation***

	Fahr	Fahr <=300	Celsius
Fahr ← 0	0		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			-17,6
Fahr ← fahr + 20	20		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			-6,7
Fahr ← fahr + 20	40		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			4,4
	Etc.		
Fahr ← fahr + 20	280		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			137,8
Fahr ← fahr + 20	300		
		Vrai	
Celsius ← 5 * (fahr – 32) / 9			148,9
Fahr ← fahr + 20	320		
		faux	

Exercices – Série 3 - Boucle

Exercice 1

Écrire un programme qui affiche la table de multiplication par 7.

Faire 4 versions : avec une boucle à compteur, avec une boucle tant que, avec une boucle jusqu'à ce que, avec un boucle sans fin.

Exercice 2

Écrire un programme qui calcule la somme des n premiers nombres (on utilisera une boucle).

Écrire une variante qui calcule la somme des n premiers nombres pairs.

Exercice 3

Réécrire le programme précédent en effaçant l'écran au début et en donnant la possibilité à l'utilisateur de recommencer le programme ou de quitter le programme.

Exercice 4

Écrire un programme qui calcule X puissance n, avec n entier relatif. On traitera tous les cas particuliers.

Exercice 5

Ecrire un programme pour déterminer si un nombre est premier.

Rappel : un nombre premier n'a pas de diviseurs.

Exercice 6

Ecrire un programme qui permet de calculer factorielle N (N !).

Rappels : 0 ! =1, N ! = 1*2*3*…*N

Exercice 7

On désire calculer le terme d'ordre n de la suite suivante définie par :

S(0)=1, S(n)=*2F(n-1)+1

Écrire un programme qui permet de calculer F(n).

Exercice 8

On désire calculer la racine carrée par la méthode de Newton.

Racine(A) est donnée par le calcul de la suite suivante :

S(0)=0, S(n+1) = 0,5 * (S(n) + A / S(n))

Quand deux valeurs successives de la suite sont identiques, on a trouvé la racine.

Écrire un programme qui permet de calculer racine(A) avec cette méthode.

Exercice 9

On désire calculer le terme d'ordre n de la suite de Fibonacci définie par :

F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2).

Écrire un programme qui permet de calculer F(n).

Exercice 10

Ecrire un programme qui permet de calculer le développement limité de sin x.

$\sin(x) = x - x^3 / 3! + x^5 / 5 ! - x^7 / 7 !$ etc.

La condition d'arrêt du calcul sera que le terme $x^n/n !$ soit plus petit qu'une constante EPSILON donnée. (Epsilon est la plus petite valeur possible pour un réel).

Exercice 11

Un nombre est dit parfait s'il est égal à la somme de ses diviseurs excepté lui-même. Par exemple, 6 est parfait, 6=1+2+3. 28 est parfait : 28=1+2+4+7+14

Ecrire un programme qui permet d'afficher tous les nombres parfaits plus petit qu'une valeur N.

Exercice 12

Un triplé d'entiers naturels (x, y, z) est dit pythagoricien si et seulement si on a : x2 +y2=z2.

Ecrire un programme qui, pour une valeur de n donnée, affiche tous les triplets pythagoriciens tels que x2 + y2 <=n.

Exercice 13

Ecrire un programme qui permet d'afficher le triangle de Pascal le niveau étant fourni au départ.

Triangle de pascal :

N0 : 1

N1 : 1 1

N2 : 1 2 1

N3 : 1 3 3 1

N4 : 1 4 6 4 1

Etc.

Principe par l'exemple : pour N4 :

- Le premier vaut toujours 1,
- 4 = 3 + 1 (les valeurs sont prises en N3 au-dessus et au-dessus à gauche)
- 6 = 3 + 3
- 4 = 1 + 3
- Le dernier vaut toujours 1

Exercice 13

Ecrire un programme qui permet d'afficher une pyramide d'étoiles : en fonction du nombre de niveaux fournis.

Exemple pour 4 niveaux :

```
        *
     * * *
  * * * * *
* * * * * * *
```

Exercice 14

Un plongeur avec bouteille quand il remonte à la surface doit marquer des arrêts pour décompresser : ce sont les paliers de décompression.

Le principe des paliers est que le plongeur doit faire un palier de 100 secondes à 30 m. De 30 m à 0, il y a un palier tous les 3 m, le temps diminuant de 10 secondes à chaque fois. Au-dessus de 30 m, ses paliers sont augmentés de 25 secondes tous les 10 m, (ainsi p.ex.: 150 s à 50 m).

On souhaite connaître, pour une profondeur donnée, le temps de remontée minimum, sachant que la vitesse de remontée est fixée à 1 m/s.

6. ++ Fonction et procédure

++ La notion de fonction

Première définition

Une fonction est algorithme qui retourne une valeur. L'algorithme a un nom et est utilisable dans une expression.

Cinquième exemple

Ecrire une fonction qui calcule la surface d'un rectangle et un programme qui permet de l'utiliser.

```
Réel : surface (larg , long)
// E : larg : Réel // la largeur
// E : long : Réel // la longueur
Début
        return  larg * long;
Fin

Programme afficheSurface
        Lire (larg, long)
        Surf = surface(larg, long) ;
        Afficher (surf) ;
Fin
```

En-tête de la fonction

```
Réel surfaceRectangle (larg , long) : réel  //  surface d'un rectangle
```

L'en-tête de la fonction commence par le type de la valeur retourné.

Ensuite vient le nom de la fonction.

Puis les variables utilisées par l'algorithme de la fonction entre parenthèses (« larg , long »). On appelle ces variables : « paramètres formels ».

Au bout, de l'entête, un commentaire pour préciser la signification du résultat renvoyé (« // surface d'un rectangle»).

La sortie standard

La valeur renvoyée par la fonction peut être appelé : sortie standard.

Paramètres formels de la fonction

Les variables de l'en-tête de la fonction sont appelés : paramètres formels.

Dans cet exemple, ces paramètres sont passés en entrée.

Une variable en entrée est une variable dont on utilise la valeur pour le calcul de l'algorithme mais dont la valeur ne sera pas modifiée.

Commentaires de l'en-tête

```
// E : larg : réel // la largeur
// E : long : réel // la longueur
```

Le commentaire de l'en-tête consiste à préciser le mode de passage (E), le type et la signification des variables de l'en-tête.

Corps de la fonction

```
Début
        return  larg * long;
Fin
```

Le corps de la fonction décrit l'algorithme de la fonction.

Les instructions sont écrites entre les mots-clés « Début » et « Fin ».

Le mot-clé « Début » est facultatif : en général, on ne le mettra que si l'on met des commentaires.

L'instruction return permet de renvoyer le résultat de la fonction sur la sortie standard.

Version compacte

```
Réel : surfaceRectangle (larg , long) {
        return  larg * long;
}
```

On ne commente pas les paramètres : tout est évident par le nom.

On utilise des accolades à la place de Début et Fin.

Sortie standard de la fonction : le propre d'une fonction

```
Fonction surface (larg , long) : réel  //  la surface
```

Une fonction renvoie un résultat sur sa sortie standard. C'est ce qui distingue une fonction d'une procédure.

Le type du résultat fourni par la fonction est précisé dans l'en-tête de la fonction : réel dans l'exemple.

Débranchement de fonction : le return

```
        return  larg * long;
```

L'instruction return permet de renvoyer le résultat de la fonction sur la sortie standard.

Cette instruction permet aussi de quitter la fonction à tout moment.

Utilisation d'une fonction : les paramètres d'appels de la fonction

```
        surf = surface(larg, long) ;
```

Quand un programme fait appel à une fonction, les paramètres passés à la fonction sont appelés paramètres d'appel.

Ces paramètres sont des expressions qui sont évaluées. Il peuvent donc être : des constantes littérales (3, 'A', «Bonjour »), des noms de variables ou des expressions. (Quand on a des variables de type complexe, ce sont en général des adresses qui sont passées en paramètre.)

C'est la valeur de l'évaluation qui est passée en paramètre à la fonction (passage par valeur).

Cette valeur est affectée à la variable du paramètre formel correspondant.

La correspondance entre les paramètres d'appels et les paramètres formels se fait par l'ordre des paramètres : le premier paramètre formel prend la valeur du premier paramètre d'appel, et ainsi de suite avec les suivants.

++ Notion de procédure : paramètres en sortie

Première définition

Une procédure est algorithme qui ne retourne pas valeur.

L'algorithme est instruction et ne peut pas être utilisé dans une expression.

Entête et corps

Une procédure est constituée, comme une fonction, d'un en-tête et d'un corps.

Paramètre en sortie

A la différence d'une fonction, une procédure a des paramètres en sortie dans son en-tête.

Il faut donc préciser obligatoirement le mode de passage des paramètres : on utilisera les flèches.

Ainsi, la fonction précédente pourrait s'écrire :

```
                                  ↓    ↓    ↑
Procédure surfaceRectangle (larg, long, surf)
// E : larg : Reel // largeur
// E : larg : Reel // longeur
// S : surf : Reel // surface
Début
        surf = larg * long;
Fin
```

On a alors affaire à une procédure.

Auquel cas le programme principal aurait été :

```
Programme afficheSurface
        Lire (larg, long)
        surface(larg, long, surf)
        Afficher (surf)
Fin
```

Une procédure peut avoir plusieurs paramètres en sortie.

Pas de sortie standard

Une procédure, par définition, n'a pas de sortie standard. C'est ce qui la distingue des fonctions.

Remarque :

Une fonction, qui a une sortie standard, peut aussi avoir des paramètres en sortie. Dans ce cas, il faudra préciser le mode de passage des paramètres.

Paramètres d'appels d'une procédure : passage par adresse

Le principe est le même pour une procédure et pour une fonction pour les paramètres en entrée.

Par contre, pour les paramètres en sortie, le paramètre d'appel ne peut pas être une expression : c'est obligatoirement une variable. On parle alors de passage par référence ou par adresse.

Débranchement de procédure : le return

L'instruction return permet de quitter la procédure à tout moment.

Comme il n'y a pas de sortie standard, le return n'est jamais accompagné d'une expression.

Ecriture simplifiée

Dans l'algorithme traité, le type des paramètres est évident, leur nom suffit à dire ce qu'ils sont, on peut donc éviter les commentaires :

```
                                    ↓    ↓    ↑
Procédure surfaceRectangle (larg, long, surf) {
        surf = larg * long;
}
```

++ Fonction mixte

Une fonction mixte possède une sortie standard et des paramètres en sortie.

Exercices – Série 4

Exercice 1

Ecrire une procédure ou une fonction renvoie le double d'un entier.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 2

Écrire une procédure ou une fonction qui calcule le prix TTC.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 3

Écrire une procédure ou une fonction qui calcule le maximum de deux nombres.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 4

Écrire une procédure ou une fonction qui calcule le maximum de trois nombres.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 5

Ecrire une procédure ou une fonction qui permet de calculer le prix d'un certain nombre de photocopies en appliquant le tarif suivant : les 10 premières coûtent 0,50 F pièce, les 20 suivantes 0,30 F pièce et toutes les autres 0,20 F pièce.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 6

Ecrire une procédure ou une fonction qui renvoie le nombre de jours d'un mois d'une année donnée.

On rappelle qu'une année est bissextile si le millésime est divisible par 4 et n'est pas divisible par 100, ou si le millésime est divisible par 400.

Exemple : 2000 et 2008 étaient des années bissextiles, 1900 n'était pas une année bissextile.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 7 (branché maths !)

Écrire une procédure ou une fonction résout dans IR une équation du premier degré en traitant tous les cas possibles (si a=0, etc.)

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 8 (branché maths !)

Écrire une procédure ou une fonction qui résout dans IR une équation du second degré en traitant tous les cas possibles (si a=0, etc.)

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 9

Ecrire une procédure ou une fonction qui affiche la table de multiplication par 5.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 10

Écrire une procédure ou une fonction qui calcule la somme des n premiers nombres (on utilisera une boucle).

Écrire une variante qui calcule la somme des n premiers nombres pairs.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 12

Écrire une procédure ou une fonction (c'est-à-dire pas récursive) qui calcule X puissance n, avec n entier positif ou nul. On traitera tous les cas particuliers.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 13

On désire calculer le terme d'ordre n de la suite de Fibonacci définie par :

F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2).

Écrire une procédure ou une fonction itérative (c'est-à-dire pas récursive) qui permet de calculer F(n) et un programme pour la tester.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 14

Ecrire une procédure ou une fonction pour déterminer si un nombre est premier. Un nombre premier est un nombre qui ne se divise que par 1 et par lui même. Autrement dit, il n'a pas de diviseur. 17 est un nombre premier. 15 est divisible par 3 et n'est donc pas un nombre premier.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 15

Ecrire une procédure ou une fonction itérative qui permet de calculer N ! (N ! se lit « factorielle N) N ! = 1*2*3… * (N-1)*N. Par définition, 0 ! = 1. La factorielle d'un nombre négatif n'est pas définie.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 17

Un nombre est dit parfait s'il est égal à la somme de ses diviseurs sans compter lui-même. Par exemple, 6 est parfait car 6=1+2+3 et que 1, 2 et 3 sont les diviseurs de 6 (sans compter lui-même, 6, qui est aussi un diviseur de 6)

1) Ecrire une procédure ou fonction qui dit si un nombre est parfait ou pas.

2) Ecrire une procédure ou une fonction qui, pour une valeur donnée de n, affiche les nombres parfaits < n

3) Ecrire un programme qui utilise cette fonction.

Exercice 16 (branché maths !)

Ecrire une procédure ou une fonction qui permet de calculer le développement limité de sin x avec une précision Epsilon.

Rappel : $\sin(x) = x - x^3 / 3! + x^5/5 ! \ldots$

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 18

Ecrire une procédure ou une fonction qui calcule le temps de remontée d'un plongeur à partir d'une profondeur donnée. Le principe de la remontée pour une profondeur inférieure ou égale à 100 m est le suivant :

le plongeur doit faire un palier de 100 secondes à 30 m puis de 30 m à 0, des paliers tous les 3 m, le temps diminuant de 10 secondes à chaque fois.

Les paliers au-dessus de 30 m sont par contre augmentés de 25 secondes tous les 10 m, (ainsi p.ex.: 150 s à 50 m).

La vitesse de remontée est fixée à 1 m/s.

Ecrire un programme qui teste la procédure ou la fonction écrite.

Exercice 19

Ecrire un programme qui permette d'utiliser les fonctions ou procédures des exercices 5, 8 et 11 (max de trois nombres, équation du second degré et calcul de la puissance). Le programme permettra de choisir parmi ces trois possibilités. Il permettra aussi de recommencer ou de choisir d'arrêter. A chaque nouveau choix, on repartira avec un écran effacé en utilisant la procédure effaceEcran( ).

7. ++ Complexité et optimisation

++ Les 3 types de complexité : apparente, spatiale et temporelle

On distingue 3 types de complexité : apparente (de structure), spatiale (de mémoire) et temporelle (de durée).

La complexité temporelle se divise en complexité temporelle théorique et pratique.

La complexité apparente (de structure)

Elle concerne la difficulté qu'éprouve le lecteur à comprendre l'algorithme. C'est une notion assez subjective. Toutefois, on prendra en compte plusieurs éléments :

1) La dénomination significative des variables
2) Usage standard des minuscules, majuscules, camel case.
3) La modularité de l'algorithme (utilisation de procédures et de fonctions)
4) Le bon usage des indentations.
5) La limitation des niveaux d'imbrication. Les débranchements structurés et la modularité permettent de le limiter.
6) La distinction, autant que possible, entre l'interface utilisateur et les traitements.
7) La distinction entre le traitement du cas général et le traitement des cas particuliers.
8) Le bon usage des commentaires.

La complexité spatiale (de mémoire)

La complexité spatiale correspond à l'encombrement mémoire de l'algorithme.

Elle peut être calculée théoriquement à partir de la connaissance des données mises en œuvre dans l'algorithme.

La complexité temporelle théorique (de durée d'exécution)

- ***Présentation***

  C'est un ordre de grandeur théorique du temps d'exécution de l'algorithme.

  Ce calcul concerne toujours des algorithmes avec des boucles.

  La mesure de la complexité est donnée par : « O ».

  « O » signifie : « complexité de l'ordre de ».

  On écrit : O(N), ou O(N/2) ou O(N^2) pour une complexité de l'ordre de N, de N/2, de N*N.

- ***Valeurs possibles***

  1, N, N/2, 2N, 3N, N^2, $2N^2$, $N^2/2$, racine(N), racine(N)/2, etc.

- ***Approximations***

  $N+1 = N$, $N^2+N = N^2$, etc.

- ***Exemple : complexité de la recherche du plus petit diviseur***

Algo 1 :	**Algo 2 :**
fonctionPPD (N) : entier // plus petit diviseur	fonctionPPD (N) : entier // plus petit diviseur

```
    Pour i de 2 à N-1
        Si N mod i = 0
            Return i
        Finsi
    Finpour
    Return 1
FIN
```

Complexité: O(N)
N=100 : O(100)

```
    si N mod 2 = 0
        return 2
    finsi
    pour i de 3 à racine(N) par pas de 2
        si N mod i = 0
            return i
        finsi
    Finpour
    Return 1
FIN
```

Complexité : O (racine(N)/2)
N=100 : O(5)

- ***Remarques***
  - On voit dans l'exemple qu'on ne calcule pas la complexité à l'unité près. On n'écrit pas : O(N-2) mais O(N). O n'est qu'une approximation. Si le temps d'exécution n'est pas fonction de N, la complexité vaut : O (1)
  - Il faudrait aussi prendre en compte le temps de calcul de racine(N).
- ***La différentes complexités temporelles théoriques***
  - Temps constant : O(1)
  - Temps linéaire : O(N)
  - Temps proportionnel : O(N/2)
  - Temps quadratique : O(N*N)
  - Temps quadratique amélioré : O(N*N/2)
  - Temps logarithmique : O(log2(N)) : c'est le cas de la recherche dichotomique.

La complexité temporelle pratique

C'est la mesure réelle du temps d'exécution de l'algorithme.

Elle est calculée en faisant des tests avec différents jeu de données.

Par exemple, on produit les résultats pour les algorithmes de tris selon la taille des tableaux.

Exemple de résultats :

Nombre de lignes :	10	100	1000	10000	50000
BULLE	0,16	20	2400		
EXTRACTION	0,12	7,3	680		
INSERTION	0,12	6,7	610		
SHELL	0,07	2	37	600	4200

ARBRE	0,2	3,5	50	660	3960
RAPIDE		2	28	365	2140

(d'après C. BAUDOIN et B. MEYER, cité dans GUYOT et VIAL)

Les 3 types d'optimisation

Corrélativement à la complexité, on peut distinguer 3 types d'optimisation : apparente, spatiale et temporelle.

L'optimisation temporelle se divise en optimisation temporelle théorique ou pratique

L'optimisation apparente (d'écriture)

L'optimisation d'écriture consiste à améliorer tous les points abordées dans la complexité apparente.

L'optimisation spatiale

Elle consiste à choisir des structures de données qui prennent moins de place et à éviter certaines méthodes algorithmique coûteuses en place (comme la récursivité, par exemple).

L'optimisation temporelle théorique

Elle consiste à diminuer la valeur de la complexité « O »,

L'optimisation temporelle pratique

Elle consiste à faire des tests de performance parmi plusieurs algorithmes en concurrence.

8. ++ Méthode pour écrire un algorithme

Méthode générale

1. **Comprendre le problème :** bien lire le sujet et bien comprendre ce qu'il y a à faire, c'est à dire le cas général correspondant au problème posé.
2. **Lister les entrées-sorties :** ce dont on a besoin pour résoudre le problème (les données) et ce qu'on va produire (les résultats).
3. **Lister les cas particuliers de départ** : c'est-à-dire les valeurs et situations particulières des données pour la résolution le problème.
4. **Trouver un principe de résolution pour les cas particuliers de départ.** Souvent, le principe de résolution des cas particuliers de départ est simple, mais ce n'est pas forcément le cas.
5. **Trouver un principe de résolution pour le cas général** : se donner les grandes lignes, c'est-à-dire les grandes actions, en français, de la méthode de résolution. Eventuellement, on peut se doter d'actions très générales, mais dans ce cas, on a intérêt à préciser ce dont on a besoin pour ces actions et ce qu'elles produisent (comme pour tout algorithme). Autrement dit, on peut se donner des procédures et des fonctions dans le principe de résolutions.
6. **Chercher des alternatives dans le cas général.** Trouver un principe de résolution pour ces alternatives. Souvent, le principe de résolution des alternatives au cas général est simple, mais ce n'est pas forcément le cas.
7. **Ecrire l'algorithme en détail, avec ou sans actions générales.** Ecrire en détail veut dire lister les opérations dans l'ordre et préciser ce qu'utilisent et ce que produisent les actions générales.

Méthode pour le principe de résolution

Pour trouver un principe de résolution, on peut essayer de trouver une solution « manuelle » : on imagine les données concrètement (dans des boîtes, sur des cartes, sur un schéma sur le papier, etc.) et on essaye de résoudre le problème « à la main ». La méthode qu'on utilise pour résoudre le problème « à la main » doit nous servir de fil conducteur pour écrire l'algorithme. On peut penser à la façon de trier un jeu de cartes par exemple.

Méthode résumé

1. Entrées-Sortie,
2. Cas particuliers de départ,
3. Cas général,
4. Alternatives