Algorithmique et programmation : introduction

Résumé

Ce document décrit le module « Algorithmique et Programmation 1 ». Il introduit le domaine du développement logiciel, l'algorithmique et les raffinages. Il présente la méthode de travail qui sera utilisée.

Table des matières

1 Présentation générale du module

Il s'agit d'un module d'algorithmique et programmation. Les deux termes sont importants. Mais si l'un est plus important que l'autre, c'est certainement le premier !

Semaine 0 : Introduction au module
- Importance de l'algorithmique ;
- Quelques généralités sur le développement de programme ;
- Méthode de travail (comment construire un programme) ;
- Environnement de programmation Dev-C++ ;
- Exemple simple de C manipulant les entrées/sorties.

Semaine 1 : Éléments de base de l'algorithmique et du C
- les constantes, les types, les variables, les expressions ;
- les instructions d'entrée/sortie ;
- l'affectation ;
- les strctures de contrôle : conditionnelles et répétitions.

Semaine 2 : Les types utilisateurs
- les tableaux ;
- les enregistrements ;
- les types énumérés.

Semaine 3 : Les sous-programmes
- Procédures et fonctions ;
- Passage de paramètres ;
- Portée des variables.

Semaine 4 : Exercice de synthèse

Pourquoi utiliser un langage algorithmique. Nous utilisons **notre propre** langage algorithmique car ceci nous permet d'introduire des concepts intéressants empruntés à différents langages de programmation tels que Pascal, Ada, Eiffel, etc.

Pourquoi le langage C. Malgré tous les défauts de ce langage, il reste encore incontournable, en particulier lors de la programmation des couches basses, proches du matériel.

Attention cependant, ce langage n'aide guère le programmeur à structurer son développement. C'est donc à lui, et donc à vous, de faire attention.

Attention, le but de ce module n'est pas d'apprendre tous les éléments du langage C mais plutôt l'algorithmique. Le langage C n'est donc qu'un support et tous les aspects du C ne seront donc pas présentés.

2 Le développement de logiciel

2.1 Les différentes étapes d'un développement logiciel

Intuitivement, les étapes qu'on doit réaliser sont les suivantes (elles correspondent plus ou moins à celle du modèle en V) :
- analyser le cahier des charges ;
- concevoir (informellement) une solution de **type algorithmique** ;
- **raffiner** la solution si nécessaire ;
- coder dans un langage impératif ;
- tester sur divers jeux d'essais ;
- mettre en service ;
- maintenir le programme, si des erreurs apparaissent ou des évolutions sont demandées.

Remarque : Même si le test apparaît après le codage, il est vivement recommandé de vérifier vos algorithmes, par exemple en simulant une exécution à la main sur un jeu de tests. Les jeux de tests sont d'ailleurs souvent identifiés en début du développement, ceci permet de mieux comprendre le comportement du système à développer.

Attention : Cette démarche a des limites importantes. En particulier elle ne met pas l'accent sur la partie spécification (le QUOI ?) et se focalise essentiellement sur le COMMENT ? Ceci ne peut donc marcher que si le cahier des charges est correctement renseigné. Est-ce toujours le cas ?

Pour ces raisons, on parle de *Programming-in-the-small*.

2.2 Un modèle plus général du développement de logiciel : le modèle en V

En réalité, la construction d'un programme est une tâche plus compliquée, qui implique plusieurs personnes qui doivent travailler ensemble, se répartir le travail, se coordonner, se comprendre, utiliser les résultats les unes des autres, etc. On parle alors de *Programming-in-the-large*.

Il est donc nécessaire de décrire précisément le déroulement d'un projet. C'est l'objectif des modèles de cycle de vie (même s'ils restent à un niveau très abstrait). Ils ont comme intérêt de structurer le développement en identifiant des étapes clés.

Le plus célèbre est le modèle de cycle de vie en V. Il est présenté à la figure 1.

Dans ce modèle, le développement logiciel est décomposé en 6 grandes étapes :

1. l'analyse des besoins : identifier le problème à résoudre ;
2. la spécification : décrire ce que doit faire le logiciel, le QUOI ;
3. la conception : décrire indépendemment d'une cible particulière (langage, système opératoire, etc.) une solution au problème. La conception globale décrit l'architecture du système en « parties ». La conception détaillée précise les détails d'une partie.
4. le codage : traduire la conception détaillée dans un langage le programmation cible.
5. le test : exécuter le programme sur des jeux de tests pour détecter des erreurs. Un logiciel doit être correct (il fait ce qu'il est censé faire – cahier des charges) et robuste (il se

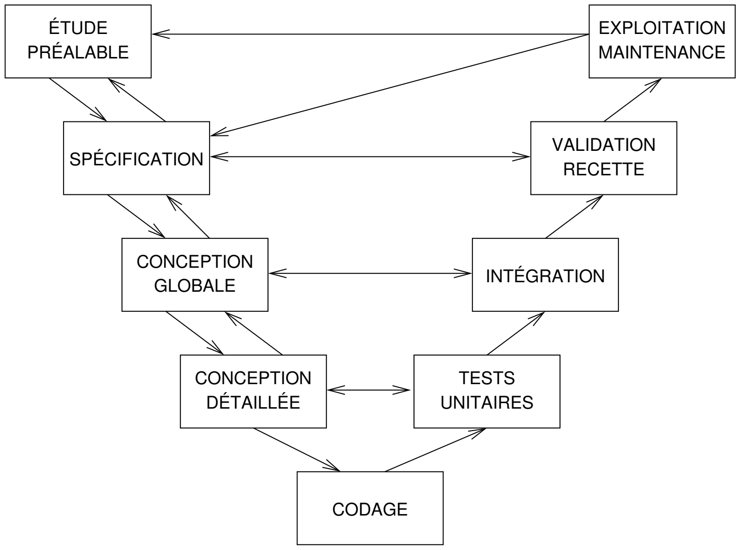

FIG. 1 – Le modèle en V

comporte bien dans les cas non prévus). Le test permet également d'évaluer les temps de réponse, l'utilisabilité, etc.

6. la maintenance : elle consiste à faire évoluer un logiciel pour lui adjoindre de nouvelles fonctions, l'adapter aux évolutions techniques (le système d'exploitation, par exemple) ou corriger des erreurs.

Remarque : Les deux branches du V ont des objectifs différents. La branche descendante consiste à construire un logiciel par étapes successives (flèches descendantes) et à prévoir les jeux de tests qui permettront de le vérifier (flèches horizontales). La branche montante correspond aux phases de vérification ou chaque élément de la construction est testé. Toute erreur détectée implique une remise en cause de la phase de construction correspondante (flèches horizontales).

2.3 L'algorithmique et la programmation dans le modèle en V

L'algorithmique intervient essentiellement dans la phase de conception (et même de conception détaillée) pour préparer l'opération de codage. En fait, l'algorithmique est bien adaptée pour les problèmes de petite taille qui peuvent être maîtrisés par une seule personne, donc dans le cas de *Programming-in-the-small*.

La programmation intervient dans la phase de codage. Elle ne constitue pas en générale une étape compliquée. Des études montrent que moins de 6% des erreurs de développement se produisent en phase de codage.

3 L'algorithmique

3.1 Définitions

Dans le *petit Robert*, dictionnaire de la langue française, mars 1995, l'algorithme est défini ainsi :

> **Algorithme** : Suite finie, séquentielle de règles qu'on applique à un nombre fini de données, permettant de résoudre des classes de problèmes semblables.
>
> Ensemble des règles opératoires propres à un calcul ou à traitement informatique. – Calcul, enchaînement des actions nécessaires à l'accomplissement d'une tâche.

L'Encyclopædia Univesalis, 1995, donne les définitions suivantes :

> L'objet de l'*algorithmique* est la conception, l'évaluation et l'optimisation des méthodes de calcul en mathématiques et en informatique. Un *algorithme* consiste en la spécification d'un schéma de calcul, sous forme d'une suite d'opérations élémentaires obéissant à un enchaînement déterminé.

En fait l'algorithmique est une branche de l'informatique théorique et l'écriture d'un algorithme correspond à la recherche d'une suite d'instructions dont l'exécution devra atteindre un but déterminé.

On connaît depuis l'antiquité des algorithmes sur les nombres (algorithme d'EUCLIDE pour le calcul du pgcd de deux nombres, par exemple), sur le traitement de l'information (algorithmes de tris, de recherche dans un ensemble, etc.)...

Remarque : Le terme d'*algorithme* tire lui-même son origine du nom du mathématicien persan Al Khzārizmī (env. 820) dont le traité d'arithmétique servit à transmettre à l'Occident les règles de calcul sur la représentation décimale des nombres antérieurement découvertes par les mathématiciens de l'Inde. Cependant, cette notion est connue depuis l'Antiquité, comme en témoignent les écrits de Diophante d'Alexandrie et d'Euclide datant du IV^{e} siècle av. J.-C.

3.2 Pourquoi l'algorithmique

L'objectif de l'algorithmique est de définir une étape intermédiaire entre le problème à résoudre et sa solution. En effet, la solution informatique est généralement à un niveau de détail trop faible pour en comprendre le principe et la justesse. La définition du problème est trop abstraite pour être exécutée directement. L'algorithme structure le problème et décrit la solution envisagée.

3.3 Les différents types d'algorithmiques

Il existe plusieurs types d'algorithmique. Elles dépendent généralement du type de langage qui sera utiliser pour coder la solution. En voici quelques types illustrés avec l'algorithme d'Euclide pour le calcul du pgcd :

- fonctionnel

```
Pgcd (a, b) =
    Si a = b Alors
        a
    SinonSi a > b Alors
        pgcd (a-b, b)
    Sinon
        pgcd (a, b-a)
    FinSi
```

– impératif

```
Pgcd (a, b) =
    TantQue a <> b Faire
        Si a > b Alors
            a ← a - b
        Sinon
            b ← b - a
        FinSi
    FinTQ
    Résultat ← a
```

– déclaratif (logique)

```
Pgcd (a, a) :- a;
Pgcd (a, b) :- Pgcd (a-b, b) Si a > b;
Pgcd (a, b) :- Pgcd (a, b-a) Si b > a;
```

Remarque : Dans la suite de ce cours, nous ne nous intéresserons qu'à l'algorithmique impérative.
Remarque : L'algorithmique fonctionnelle n'est pas forcément incompatible avec l'algorithmique impérative.

3.4 Un exemple : le robot

Exercice 1 : Déplacement d'un robot
Écrire un algorithme qui permet à un robot sachant avancer d'un pas et pivoter de 90^o vers la droite de se déplacer de la salle de cours (position initale du robot) jusqu'au secrétariat (voir figure 2).

4 Raffinages et algorithme

4.1 Définition

La méthode des raffinages est une méthode d'analyse descendante dont l'objectif est de construire un algorithme en décomposant le problème posé en sous-problèmes (étapes de haut niveau) qui seront à leurs tours décomposées (raffinées). Une structure arborescente est ainsi construite : les nœuds sont des instructions de haut niveau, les feuilles sont des instructions élémentaires (ou des étapes sans difficultés à traduire dans le langage de programmation).

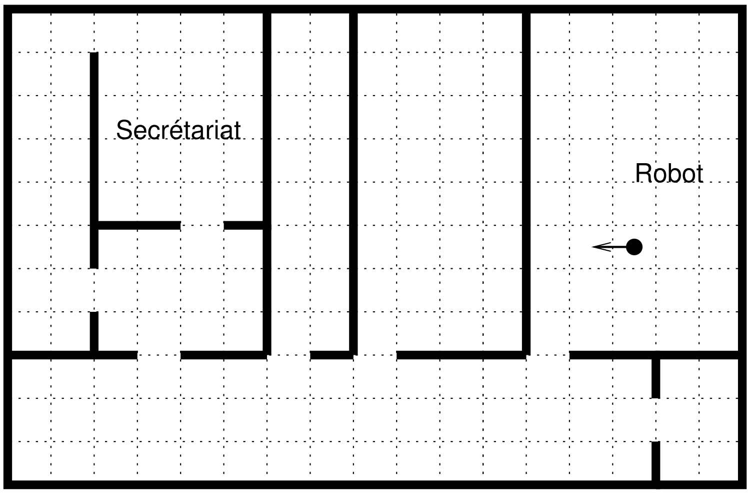

FIG. 2 – Plan des lieux

L'algorithme est alors obtenu en « aplatissant » l'arbre des raffinages : les étapes intermédiaires (nœuds de l'arbre) sont remplacées par leur raffinage. Ces étapes intermédiaires **doivent** être conservées sous forme de commentaires dans l'algorithme (et dans le programme final).

Par convention, on note R_0 la définition (succincte et précise) du problème à résoudre, R_1 la décomposition de R_0, premier niveau de raffinage et plus généralement R_i un raffinage de niveau i. Il faut alors préciser l'étape qui est raffinée soit en donnant son nom (ce que nous utiliserons), soit en utilisant une codification (par exemple à plusieurs entiers).

Notons que les étapes intermédiaires d'un raffinage peuvent être décrites sous la forme de sous-programmes (instructions utilisateurs).

Remarque : Le terme « affinage » est parfois employé comme un synonyme de « raffinage ».

4.2 Qualités d'un raffinage

Voici quelques qualités que l'on peut attendre d'un raffinage.

- Un raffinage doit être bien présenté (indentation).
- Le vocabulaire utilisé doit être précis et clair.
- Chaque niveau de raffinage doit apporter suffisamment d'information (mais pas trop). Il faut trouver le bon équilibre !
- Le raffinage d'une étape (l'ensemble des sous-étapes) doit décrire complètement cette étape.
- Le raffinage d'une étape ne doit décrire que cette étape.
- Les différentes étapes introduites doivent avoir un niveau d'abstraction homogène.
- La séquence des étapes doit pouvoir s'exécuter logiquement.
- Il est interdit d'employer des structures de contrôle déguisées (conditionnelles ou répétitions).

Remarque : Certaines de ces règles sont subjectives. C'est vrai. Mais ce qui est important c'est que vous puissiez expliquer et justifier les choix que vous avez faits.

Conseil : Pour identifier les étapes d'un raffinage, on peut se poser la question « comment je fais cette étape ? ». Pour vérifier le niveau de détail d'une étape ou savoir si elle correspond à l'étape raffinée, on peut se poser la question « pourquoi je fais cette étape ? ».
Remarque : Pour contrôler un raffinage, on peut également s'intéresser aux données qui sont utilisées, modifiées ou élaborées par une étape du raffinage et vérifier ce qu'en font les autres étapes ou l'étape supérieure.

4.3 Comment construire un algorithme ?

La construction d'un algorithme est en soit un algorithme. On peut donc essayer de le décrire sous la forme d'un algorithme. Voici une proposition.

```
R0 : Construire un algorithme (pour un problème posé)

R1 : Comment « Construire un algorithme »
    | Comprendre le problème               in énoncé; out R0, tests
    | Identifier une solution informelle   in R0, tests; out principe
    | Structurer cette solution            in principe, R0, tests; out raffinages
    | Produire l'algorithme                in raffinages; out algorithme
    | Tester le programme                  in algorithme, in tests; out erreurs

R2 : Comment « Comprendre le problème »
    | Reformuler le problème (R0)
    | Identifier des jeux de tests

R2 : Comment « Structurer cette solution »
    | Construire R1
    | Vérifier R1
    | Raffiner les étapes non élémentaires
    | Vérifier l'ensemble de l'algorithme

R3 : Comment « Identifier des jeux de tests »
    | Identifier des jeux de tests représentatifs des cas nominaux
    | Identifier des jeux de tests correspondant aux cas limites
    | Identifier des jeux de tests correspondant aux cas hors limites

R3 : Comment « Construire R1 »
    | Lister les étapes
    | Ordonner les étapes
    | Regrouper les étapes
    | Identifier les flots de données
    | Compléter le dictionnaire des données

R3 : Comment « Vérifier R1 »
    | Vérifier que les étapes introduites font R0
    | Vérifier que les étapes introduites ne font que R0
    | Vérifier le niveau d'abstraction des étapes
    | Vérifier l'enchaînement des étapes
    | Vérifier la cohérence du flôt de donnée
    | Vérifier la précision du vocabulaire
```

```
    | ...

R3 : Comment « Raffiner les étapes non élémentaires »
    | TantQue il y a des étapes non élémentaires Faire
    |    | Choisir l'étape la moins bien comprise
    |    | Construire le raffinage de cette étape
    |    | Vérifier ce raffinage
    | FinTQ
```

Si cet algorithme donne une vision assez claire des différentes étapes à effectuer pour construire un algorithme, il comporte néanmoins un certain nombre d'erreurs par rapport aux critères de qualité énoncés ci-dessus.

Principalement, il n'y a pas de prise en compte d'une erreur détectée lors d'une vérification. Le terme « vérifier » ne devrait jamais être utilisé dans un raffinage. En effet, si on vérifie c'est pour prendre une décision en cas d'échec de la vérification. Il faudrait donc faire apparaître une structure de contrôle.

5 Programme

L'algorithme étant défini, un programme est sa traduction dans un langage de programmation donné. Cette traduction est quasi-automatique.

Les étapes intermédiaires de l'arbre de raffinage peuvent éventuellement être décrites sous la forme de sous-programmes (instructions utilisateurs).

6 Rappel sur le développement de programme

Les principales étapes du développement d'un programme sont présentées à la figure 3.

Considérons le problème du calcul du pgcd donné à l'exercice 2. En appliquant, les étapes ci-dessus, on obtient un raffinage sur plusieurs niveaux (listing 1), on en déduit un algorithme (listing 2) que l'on peut ensuite traduire en programme C (listing 3). Les autres étapes sont décrites dans la section suivante.

Exercice 2 : Pgcd par l'algorithme d'EUCLIDE
Écrire un programme qui affiche le pgcd de deux entiers strictement positifs saisis au clavier. Le pgcd sera calculé suivant l'algorithme d'EUCLIDE :

$$pgcd(a,b) = \begin{cases} a & \text{si } a = b \\ pgcd(a-b, b) & \text{si } a > b \\ pgcd(a, b-a) & \text{si } a < b \end{cases}$$

7 Recommandations lors de l'écriture d'un programme

Les noms Donnez des noms significatifs aux constantes, types, variables, procédures, fonctions... et plus généralement à tout identificateur.

Listing 1 – Raffinages possibles pour le calcul du pgcd

```
R0 : Calculer le pgcd de deux nombres entiers strictement positifs

R1 : Raffinage De « Calculer le pgcd de deux nombres entiers strictement positifs »
        Saisir les deux entiers          a, b: out Entier
        Déterminer leur pgcd             a, b: in Entier; pgcd: out Entier
        Afficher le pgcd                 pgcd: in Entier

R2 : Raffinage De « Déterminer leur pgcd »
        TantQue a /= b Faire
            Si a > b Alors
                a ← a -b
            Sinon
                b ← b - a
            FinSi
        FinTQ
        pgcd ← a
```

Listing 2 – Algorithme déduit des raffinage du listing 1

```
Algorithme pgcd_euclide

    -- Calculer le pgcd de deux nombres entiers strictement positifs

Variables
    a, b: Entier          -- Deux entiers saisis au clavier
    pgcd: Entier          -- le pgcd de a et b

Début
    -- Saisir les deux entiers
    Écrire("Donner␣deux␣entiers␣strictement␣positifs␣(pas␣de␣contrôle)␣:␣");
    Lire(a, b);

    -- Déterminer leur pgcd
    TantQue a /= b Faire
        Si a > b Alors
            a ← a -b
        Sinon
            b ← b - a
        FinSi
    FinTQ
    pgcd ← a

    -- Afficher le pgcd
    Écrire(pgcd)
Fin.
```

Listing 3 – Le fichier pgcd.c

```c
/**********************************************************************
 *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
 *  Version  : 1.3
 *  Objectif :
 *      Calculer le pgcd de deux nombres entiers strictement positifs
 *********************************************************************/

#include <stdio.h>
#include <stdlib.h>

/*
 * Calcul du pgcd de deux entiers strictement positifs saisis au
 * clavier (pas de vérification des entrées).  Le résutat est affiché
 * sur l'écran.
 */
int main()
{
    int a, b;   /* deux entiers dont on veut déterminer le pgcd */

    /* saisir les valeurs de a et b     */
    printf("Donnez deux valeurs entières strictement positives.\n");
    printf("A = ");
    scanf("%i", &a);
    printf("B = ");
    scanf("%i", &b);

    /* calculer le pgcd (algorithme d'Euclide)  */
    while (a != b) {
        if (a > b) {
            a = a - b;
        }
        else {
            b = b - a;
        }
    }

    /* afficher le résultat     */
    printf("pgcd = %i\n", a);

    return EXIT_SUCCESS;
}

/* Tests effectués
  10                15         ->      5
   2                 3         ->      1
   1        2000000000         ->      1 (très, très long !)
  -1                 0         ->      boucle !
*/
```

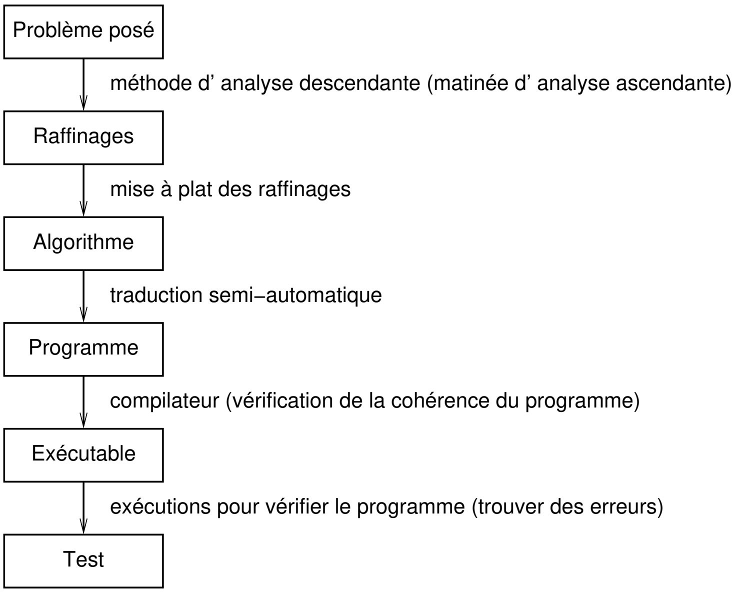

FIG. 3 – Le développement de programme (vision simplifiée)

Présentation du texte Pour mettre en évidence la structure du programme, **il faut** indenter (décaler vers la droite) les blocs d'instructions contrôlées par une structure de contrôle. Ceci facilite la compréhension du programme, la recherche des erreurs et la relecture (pensez que vos programmes seront lus par vous et par d'autres !).

N'hésitez pas à sauter une ligne pour mettre en évidence un groupe d'instructions reflétant le raffinage du programme. Un commentaire (trace du raffinage) devra indiquer l'objectif de chacun de ces groupes d'instructions. Le programme de calcul du pgcd donné au listing 3 définit 3 blocs : saisir les entrées, calculer le pgcd et afficher le résultat.

Les commentaires L'objectif des commentaires est d'expliquer le code du programme, les choix que vous avez faits, les algorithmes utilisés...

Pour être efficaces, ils doivent être clairs, concis et apporter une information. Le commentaire suivant est inutile :

```
i: Entier;  -- déclaration d'une variable i entière
```

En effet, toutes les informations apportées par le commentaire sont contenues dans le code !

Les commentaires doivent être placés partout où ils apportent quelque chose. En particulier, vous devez faire apparaître les commentaires liés aux raffinages : ils indiquent ce qui va être fait par les instructions suivantes. Ceci permet d'expliquer l'algorithme utilisé.

Vous devez également mettre des commentaires dans les cas suivants :

– déclaration des variables : ne déclarez qu'une seule variable par ligne et indiquez à quoi elle sert, quel est son rôle. En lui donnant un nom significatif, vous pourrez diminuer la taille du commentaire. Évitez `bool, test, cond: Booléen` ou `tab: Tableau ...` qui n'apportent rien !

– entête des fonctions et procédures : pour chaque fonction et procédure indiquez son rôle, la signification des paramètres et les contraintes qu'ils doivent respecter. Pour les fonctions indiquez les contraintes sur le type de retour.

Le test des programmes Le but des exercices n'est pas d'écrire du C « compilable », mais d'écrire un programme C qui fait ce qui est demandé dans l'énoncé ! Faire des tests ne permet évidemment pas de prouver que le programme est correct. Surtout si vous ne faites qu'un seul test ! Cependant les tests permettent de vérifier que le programme marche bien dans les cas testés et, surtout, ils permettent de découvrir des erreurs qui devront être corrigées.

Comme la correction d'une erreur peut introduire de nouvelles erreurs, il est nécessaire de refaire (rejouer) tous les tests. Il faut donc conserver une trace des jeux de tests utilisés pour les rejouer et vérifier qu'ils donnent toujours des résultats corrects (tests de non régressivité).

Tout comme les étapes de raffinage, les tests font donc partie intégrante de l'écriture d'un programme. Ainsi, tout comme la trace des raffinages est conservée sous forme de commentaire dans le code C, nous vous proposons de mettre en commentaire, à la fin du fichier C, les tests à effectuer (et donc effectués) et les résultats attendus.