Algorithmique et Programmation

—

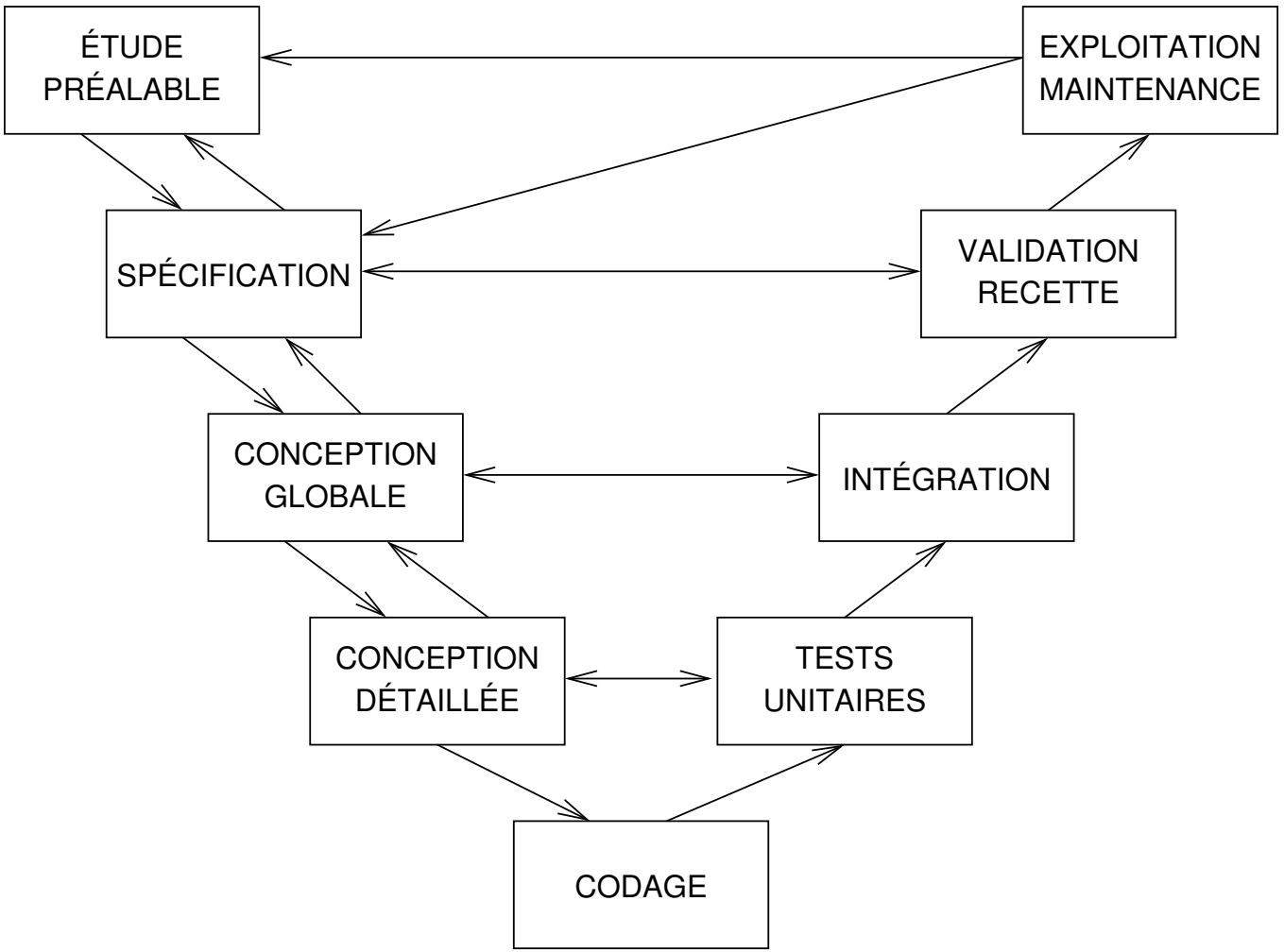
Introduction à l'algorithmique

Xavier Crégut, Marc Pantel
<Prénom.Nom@enseeiht.fr>

ENSEEIHT
Télécommunications & Réseaux
Informatique & Mathématiques Appliquées

Sommaire

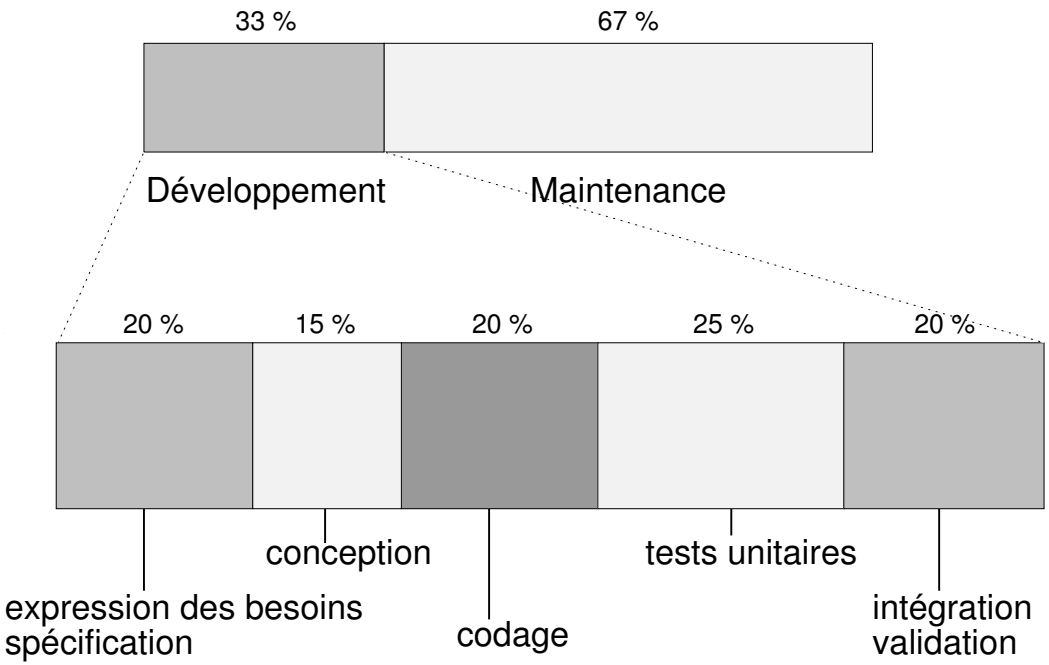
Le modèle en V



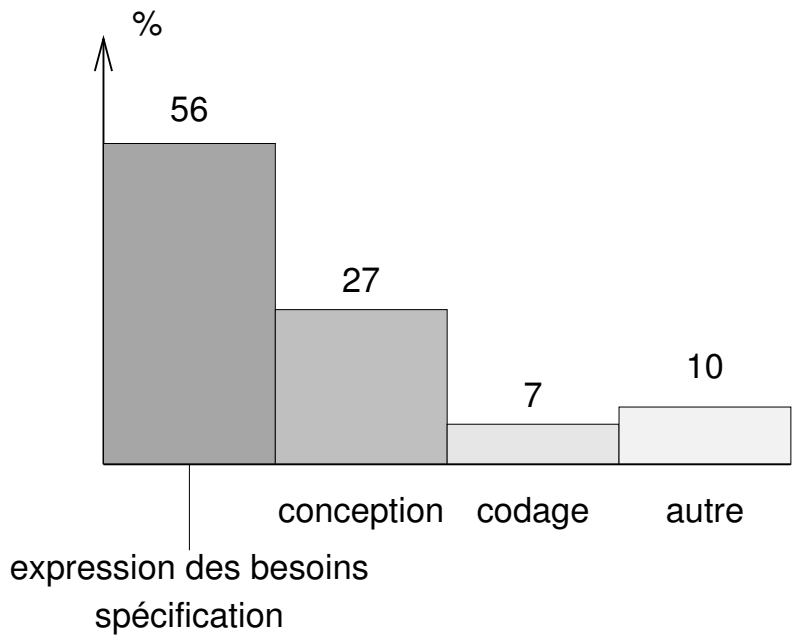
Description des étapes

- **Étude préalable :** vérifier la faisabilité du problème posé, l'intérêt.
- **Spécification :** décrire ce que doit faire le logiciel, le QUOI ;
  Elle inclut *l'analyse des besoins* : identifier le problème à résoudre ;
- **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 de chaque partie.
- **Codage :** traduire la conception détaillée dans le langage cible.
- **Test :** exécuter le programme sur des jeux de tests pour détecter des erreurs. Le test permet également d'évaluer les temps de réponse, l'utilisabilité, etc.
- **Maintenance :** 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.

Quelques chiffres



Coût relatif des phases

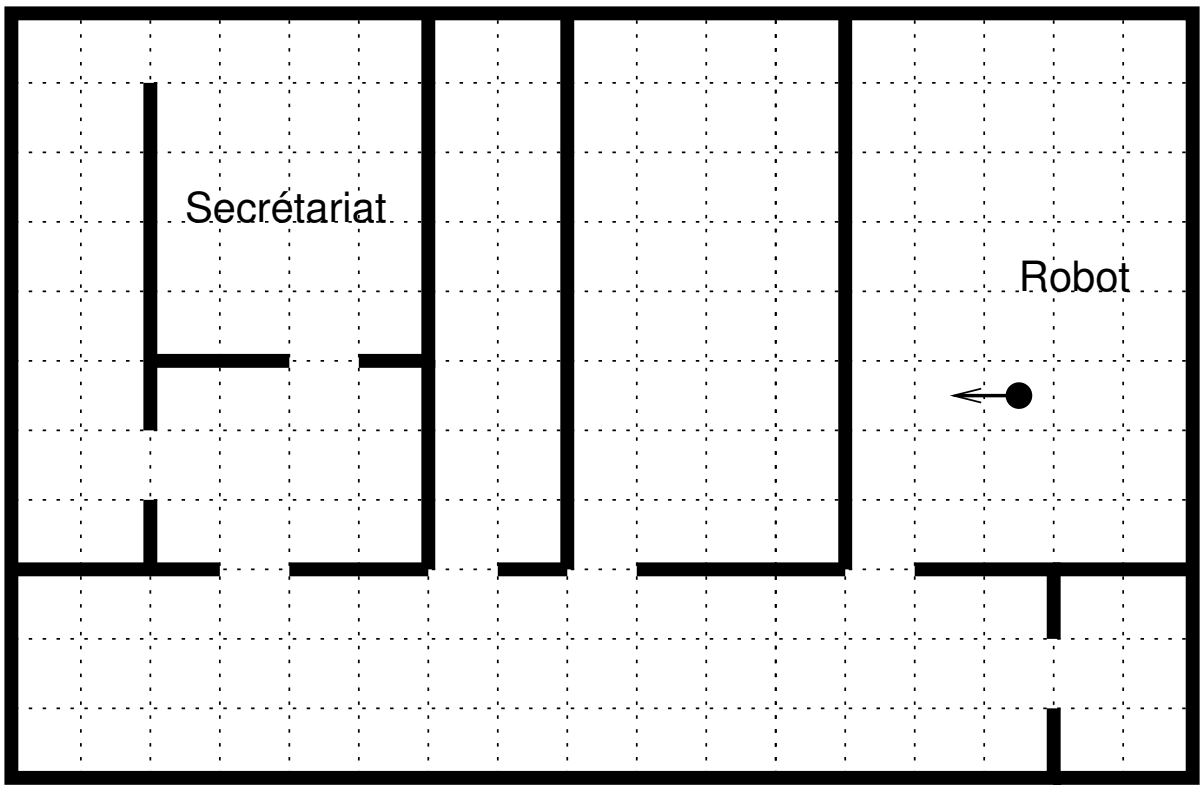


Origine des erreurs

Sommaire

Guider un robot

Exercice : Écrire un programme qui permet à un robot de se déplacer de la salle de cours (position initiale du robot) jusqu'au secrétariat sachant qu'il est capable d'avancer d'une case et de pivoter de 90° vers la droite.



Analyse du problème

- Le programme s'adresse au robot qui est notre **processeur**.
- Le robot ne comprend que deux ordres :
  - avancer d'une case ;
  - pivoter de 90° à droite.

Ce sont les **instructions élémentaires**.

Attention : Chaque instruction a un effet qu'il est important de bien comprendre pour savoir comment l'utiliser !

Processeur

Élément (machine, homme, etc.) qui exécute un programme.

Instruction élémentaire

Opération qu'un processeur sait exécuter.

Une solution élémentaire

```
Algorithme Guider_robot

Début
    AVANCER
    AVANCER
    PIVOTER
    PIVOTER
    PIVOTER
    AVANCER
    AVANCER
    AVANCER
    PIVOTER
    AVANCER
    AVANCER
    AVANCER
    AVANCER
    AVANCER
    AVANCER
    AVANCER
    PIVOTER
    AVANCER
    PIVOTER
    PIVOTER
    PIVOTER
    AVANCER
Fin.
```

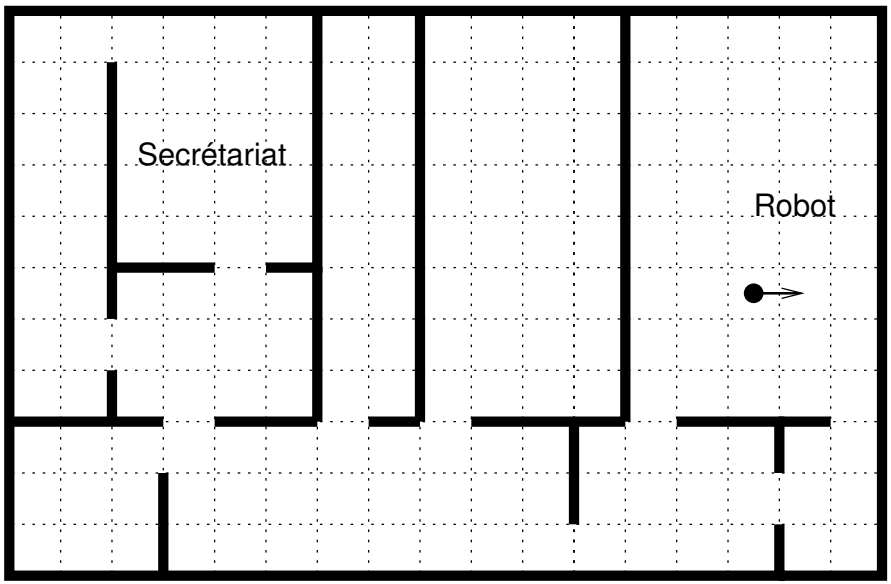
Remarque : C'est aussi le programme !

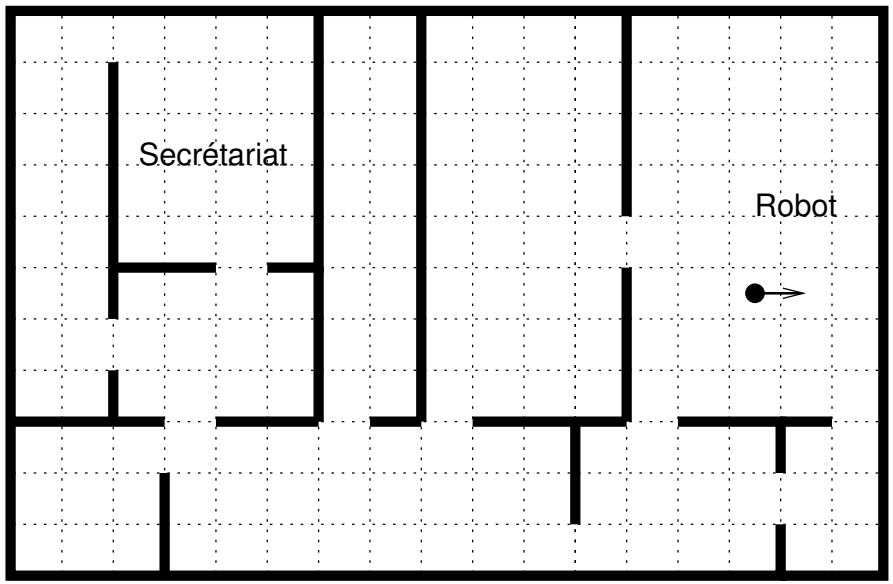
Évaluation de la solution proposée

Les questions suivantes doivent permettre d'évaluer la solution proposée :

1. Est-ce que le processeur (le robot) comprend le programme écrit ?
2. Est-ce qu'un lecteur humain comprend le programme écrit ?
   - par où passe le robot ?
   - quand on considère une instruction, sait-on ce que fait le robot ? Connaît-on son but ?
3. Est-il facile de faire évoluer le programme ?
   - Si on change la topologie des lieux (voir transparent 11) ?
   - Si on change les caractéristiques techniques du robot ?
4. Quel est le chemin suivi ?
   - Est-ce le plus court ?
   - Est-ce le plus rapide ?
5. ...

Modification de l'exercice sur le robot





Exercice : Indiquer, pour chaque nouvelle configuration, les modifications à apporter au programme déjà écrit.

Critique de la solution proposée

- Le programme fonctionne... mais ce n'est pas évident d'en être certain.
- Le programme est difficile à comprendre.
- On ne sait pas *a priori* quel est le chemin suivi par le robot. On peut donc difficilement répondre aux questions suivantes :
  - Est-ce le chemin le plus court ? le plus rapide ?
  - Que faut-il modifier si on bloque une partie du couloir ?
  - Que faut-il modifier si on change de type de robot ?

Conclusion : En plus du programme, il est nécessaire de conserver des informations qui expliquent le programme, facilitent sa compréhension et permettent ses évolutions.

Ce que nous apprend cet exercice

Vous êtes capables d'écrire un programme !
mais :

- Le programme est-il facile à comprendre ? Par vous... et par d'autres ?
- Est-il correct ?
- Est-il facile de le modifier (changements dans le problème posé) ?

Remarque : Pour des programmes simples (comme tous ceux que vous ferez dans le cadre de votre formation), il est possible de trouver une solution par *tâtonnements successifs* : essais/erreurs. Cependant :

- vous risquez d'y passer plus (trop) de temps ;
- vous ne serez peut-être pas sûrs d'avoir écrit un programme correct ;
- vous ne pourrez pas facilement le faire évoluer.

$\Longrightarrow$ S'appuyer sur une **méthode** systématique : **les RAFFINAGES**

Solution pour guider le robot (raffinages)

Remarque : On peut commencer par nommer les pièces : salle de cours, couloir, vestibule, secrétariat, escalier, bureau du chef, cafétéria, salle de cours 2.

```
R0 : Guider le robot de la salle de cours vers le secrétariat

R1 : Comment « Guider le robot de la salle de cours vers le secrétariat »
       | Sortir de la salle de cours
       | Longer le couloir
       | { Le robot est devant la porte du vestibule }
       | Traverser le vestibule
       | { Le robot est dans le secrétariat }
```

Solution pour guider le robot (raffinages) (2)

```
R2 : Comment « Sortir de la salle de cours »
       | Progresser de 2 cases
       | Tourner à gauche
       | Progresser de 3 cases

R2 : Comment « longer le couloir »
       | Tourner à droite
       | Progresser de 9 cases
       | Tourner à droite

R2 : Comment « traverser le vestibule »
       | Progresser de 3 cases
       | Tourner à droite
       | Progresser de 1 case
       | Tourner à gauche
       | Progresser de 1 case
```

Solution pour guider le robot (raffinages) (3)

```
R3 : Comment « Tourner à gauche »
       | PIVOTER
       | PIVOTER
       | PIVOTER

R3 : Comment « Tourner à droite »
       | PIVOTER

R3 : Comment « Progresser de 2 cases »
       | AVANCER
       | AVANCER

R3 : Comment « Progresser de 3 cases »
       | AVANCER
       | AVANCER
       | AVANCER
```

L'algorithme correspondant

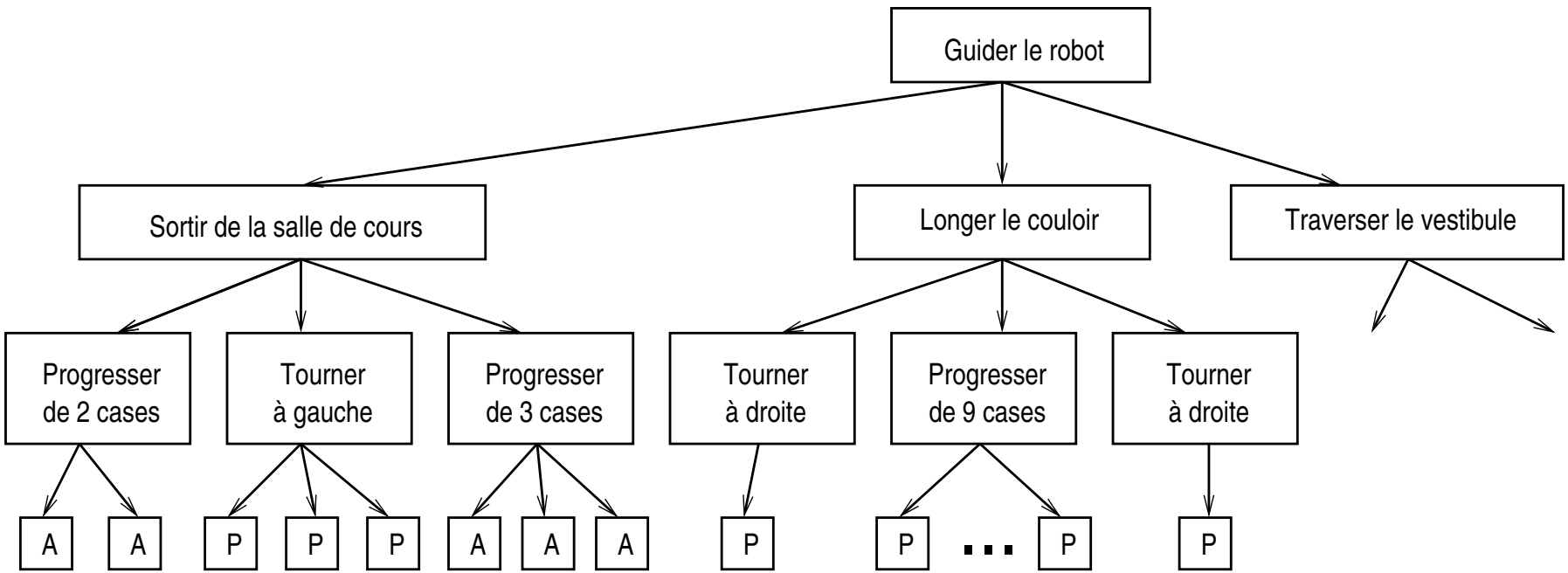
```
Algorithme Guider_robot

    -- Guider le robot de la salle de cours vers le secrétariat
Début
    -- Sortir de la salle de cours
    --    Progresser de 2 cases
    AVANCER
    AVANCER
    --    Tourner à gauche
    PIVOTER
    PIVOTER
    PIVOTER
    --    Progresser de 3 cases
    AVANCER
    AVANCER
    AVANCER

    -- Longer le couloir (jusqu'à la porte du vestibule)
    --    Tourner à droite
    PIVOTER
    ...
Fin.
```

Que retirer de cet exemple ?

1. Le problème est décomposé en étapes de haut niveau, abstraites.
   - Elles sont nécessaires pour comprendre la solution finale.
   - Elles ne sont pas comprises par le robot et doivent donc lui être détaillées.
2. Les étapes abstraites (non élémentaires) sont la trace de notre raisonnement.
3. On peut mettre autant de niveaux de raffinage que nécessaire ou utile.
4. On s'arrête quand toutes les étapes identifiées sont élémentaires.
   - Ici AVANCER et PIVOTER, seules instructions du robot
5. La décomposition en raffinage produit un arbre de racine R0 dont les feuilles contiennent les instructions élémentaires.
6. Le programme s'obtient en mettant l'arbre à plat, les étapes non élémentaires étant conservées sous forme de commentaire.

Arbre des raffinages

- On peut dessiner l'arbre des raffinages :
  - La racine, en haut, est le R0.
  - Les feuilles sont des étapes élémentaires.
  - Les nœuds sont des étapes abstraites.

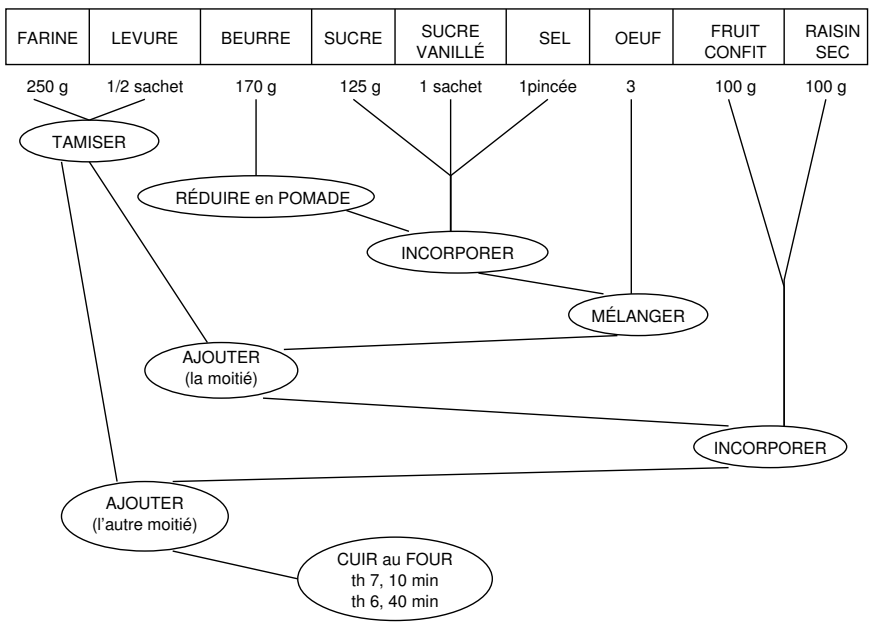


- Mais dessiner l'arbre n'est pas toujours simple !

Autres exemples de raffinages et/ou algorithmes

- Document, rapport, article, etc.
  - plan, souvent explicité par la table des matières
    au moins annoncé dans l'introduction !
- Construction d'un bâtiment
  - l'architecte commence par faire des plans
- Mode d'emploi
  - description des procédures à suivre sous forme de listes numérotées d'actions à réaliser par l'utilisateur
- Recette de cuisine
  - décrit les étapes à suivre
  - des étapes considérées comme connues
    - préparer une pâte brisée
    - abaisser la pâte...

    (décrites en début d'ouvrage !)
  - plusieurs présentations



Sommaire

Critères de qualité d'un programme (1/2)

... ou **Comment évaluer son programme ?**

- Est-ce qu'il est **lisible/compréhensible** ?
  - mise en évidence de sa structure (sauts de ligne, indentation) ;
  - assez de commentaires (informatifs !) ? Trace du raffinage ?
- Est-ce qu'il est **correct** ?
  Un programme est correct si toutes ses exécutions possibles fournissent les résultats attendus.
  Il faut « prouver » l'algorithme !
- Est-ce qu'il est **valide** ?
  Valide : répond au cahier des charges.
  « Est-ce que j'ai écrit le bon programme ? »
- Est-ce qu'il est **robuste** ?
  Le programme se « comporte bien » dans les cas non prévus par le CdC.

Critères de qualité d'un programme (2/2)

- Est-ce que le programme est **efficace** ?
  Est-ce que le programme utilise de manière optimale les ressources : quantité de mémoire, vitesse du CPU, accès aux disques, etc.
- Est-ce qu'il pourra être utilisé dans d'autres contextes (**réutilisabilité**) ?
  Intérêt : Diminuer le coût des développements actuels et futurs.
- Est-ce qu'il sera facile de le modifier, le faire évoluer (**extensibilité**) ?
  Justification : Les besoins changent (*maintenance évolutive*),
  des erreurs sont trouvées (*maintenance corrective*) !

Sommaire

Qu'appelle-t-on « algorithme » ?

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.*

- suite finie, nombre fini : un **programme doit se terminer** ;
- séquentielle, enchaînement : ordre d'exécution ;
- régles : instructions ;
- classes de problèmes semblables ;

Exemple : La *multiplication* de deux entiers : 124 * 735 = ?

Multiplication de deux entiers

- Trois exemples :

```
  124     |           1  2  4     |     124     735      124
* 735     |       *   7  3  5     |     248     367      248
-----     |       -----------     |     496     183      496
  620     |          05 10 20     |     992      91      992
 372      |       03 06 12        |    1984      45     1984
868       |    07 14 28           |    3968      22
-----     |    --------------     |    7936      11     7936
91140     |    07 17 39 22 20     |   15872       5    15872
          |    --------------     |   31744       2
          |     9  1  1  4  0     |   63488       1    63488
          |                       |                    -----
          |                       |                    91140
```

- Comment ça marche ?

Algorithme vs programme

- **Intérêt :** Représentation de la *solution à un problème posé* avec une solution opératoire de manière indépendante d'une réalisation effective dans un langage de programmation particulier.
  $\Longrightarrow$ répond au problème posé mais plus abstrait que le langage considéré.
- **Algorithme et programme sont souvent synonymes.**
  La différence est que le programme est souvent plus formel (peut être interprété par une machine, en fait par un compilateur ou un interpréteur) alors que l'algorithme est plus abstrait (plus libre) et s'adresse à un être humain.
- L'algorithmique utilisée est souvent fonction du langage de programmation considéré !

Les différents types d'algorithmique

Algorithmique fonctionnelle

```
Pgcd(a, b) =

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

Algorithmique impérative

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

Algorithmique déclarative (logique)

```
Pgcd(a, a, a).
Pgcd(a, b, r) :- a > b, a1 is a - b, Pgcd(a1, b, r).
Pgcd(a, b, r) :- a < b, Pgcd(b, a, r).
```

Principales caractéristiques des différentes approches ?

- **Fonctionnelle**
  - Pas de modification de données existantes (pas d'effet de bord)
    $\Longrightarrow$ création de nouvelles données.
  - Proche des mathématiques.
  - Exemple : Caml.
  - Abordée au travers de la récursivité mais attention aux effets de bord !
- **Impérative :**
  - Modification de l'état du programme (affectation : `<-`)
  - C'est celle que nous allons traiter essentiellement.
- **Déclarative :**
  - L'ordre des instructions (règles) n'a pas d'importance.
  - Abordée avec la commande `make` et les fichiers `Makefile` (Algo-Prog 1TR)

Sommaire

Les étapes dans le développement d'une application

- analyser le cahier des charges ;
- concevoir (informellement) une solution de **type algorithmique** ;
- **raffiner** la solution ;
- coder dans un langage impératif ;
- tester sur divers jeux d'essais (jeux de test) ;
- mettre en service ;
- maintenir le programme, si des erreurs apparaissent ou des évolutions sont demandées ;
- jeter le programme.

Attention : Ceci est une démarche simplifiée qui marche bien dans le cas de petits problèmes qui peuvent être maîtrisés par une personne et dont on connait déjà la solution.

Exemple fil rouge

Exercice : Pgcd de deux nombres.
Écrire un programme qui affiche le pgcd de deux entiers *strictement positifs* dont la valeur a été saisie au clavier.

Comprendre le problème posé

Il est essentiel de bien comprendre le problème posé pour avoir des chances d'écrire le **bon** programme !

Moyens :

- Reformuler le problème en rédigeant R0.
- Lister des « exemples d'utilisation » du programme. Il s'agit de préciser :
  - les données en entrées ;
  - **et** les résultats attendus.

Remarque : Les exemples d'utilisation donneront les jeux de test fonctionnels qui permettront de tester le programme.

Comprendre le problème posé : exemple sur le pgcd

R0 : Afficher le pgcd de deux entiers strictement positifs

Remarque : La reformulation doit être relativement concise et précise, au risque d'être un peu incomplète. Il s'agit de synthétiser le problème posé.
Les **exemples d'utilisation** (ou jeux de tests) pourraient être les suivants :

```
   a    b       pgcd
-------------------------------------------------------
   2     4  ==>  2                    -- cas nominal
  20    15  ==>  5                    -- cas nominal
  20    20  ==>  20                   -- cas limite
  20     1  ==>  1                    -- cas limite
   1     1  ==>  1                    -- cas limite
   0     4  ==>  Erreur : a <= 0      -- cas d'erreur (robustesse)
   4    -4  ==>  Erreur : b <= 0      -- cas d'erreur (robustesse)
-------------------------------------------------------
```

Identifier une solution informelle

- Identifier une manière de résoudre le problème.
- Il s'agit d'avoir l'idée, l'intuition de comment traiter le problème.
- Comment trouver l'idée ? **C'est le point difficile !**
- *Exemple :* Pour calculer le pgcd d'un nombre, on peut appliquer l'algorithme d'Euclide :

  Il s'agit de soustraire au plus grand nombre le plus petit. Quand les deux nombres sont égaux, ils correspondent au pgcd des deux nombres initiaux.

- **Remarque :** On peut vérifier son idée sur les exemples d'utilisation identifiés.

Raffinage

Idée : Décomposer un problème « compliqué » en sous-problèmes plus simples que l'on sait résoudre (que l'on espère savoir résoudre !).
Remarque : Il s'agit de formaliser la solution informelle.
Notation : On note R1 la décomposition/le **raffinage** de R0.
Définition : Un raffinage est la décomposition d'une étape en sous-étapes qui, réalisées dans l'ordre indiqué, réalisent exactement le même objectif.
Exemple :

```
R1 : Raffinage De « Afficher le pgcd de deux entiers positifs »
  | Saisir deux entiers
  | { les deux entiers sont strictements positifs }
  | Déterminer le pgcd des deux entiers
  | Afficher le pgcd
```

Important : Entre accolades est exprimée une propriété du programme, vraie à cet endroit.
Remarque : On peut utiliser des structures de contrôle dans un raffinage.

Raffinages et flot de données

Un raffinage décrit un enchaînement d'étapes qui échangent (généralement) des données (des informations).
Il est alors important de caractériser ce qu'une étape fait d'une donnée :

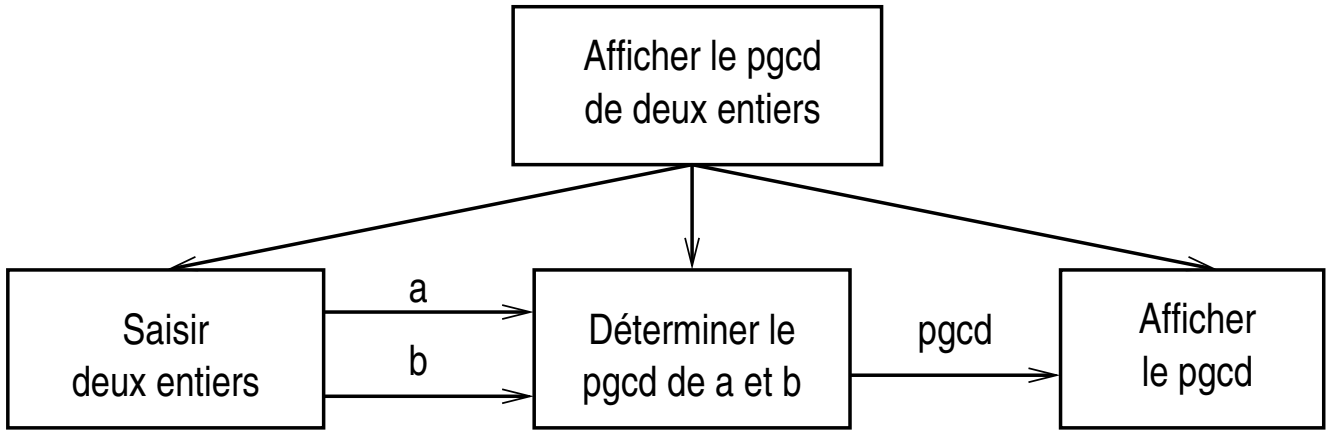
- **in** : elle l'utilise sans la modifier ;
- **out** : elle la produit (modifie) cette donnée sans accéder à sa valeur ;
- **in out** : elle l'utilise, puis la modifie.

```
R1 : Raffinage De « Afficher le pgcd de deux entiers positifs »
  | Saisir deux entiers                a, b: out
  | { (a > 0) Et (b > 0) }
  | Déterminer le pgcd de a et b       a, b: in; pgcd: out
  | Afficher le pgcd                   pgcd: in
```

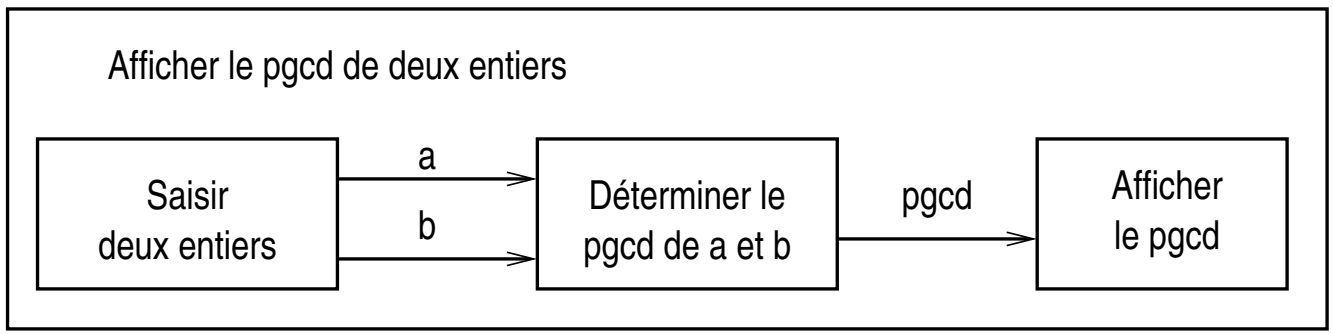
Avantage 1 : Expliciter les données permet de les référencer dans les autres étapes et d'écrire plus formellement les propriétés du programme.
Avantage 2 : Contrôler la chronologie d'une étape : une variable ne peut être en **in** que si elle apparaît avant en **out**.

Représentation graphique

Représentation sous forme d'arbre



Représentation sous forme de boîtes imbriquées



Remarque : Ces représentations permettent de « visualiser » un raffinage mais ne seront pas utilisées car elles sont peu adaptées si les flots de données sont complexes ou les sous-étapes non séquentielles.

Continuer à raffiner

- Si un raffinage introduit des étapes non élémentaires, elles doivent être à leur tour raffinées.
- **Exemple :** Les 3 sous-étapes introduites dans la décomposition de R0 sont non élémentaires et doivent donc être raffinées.
- **Notation :** On note R2 le raffinage des étapes introduites dans R1.
  Attention, il faut préciser l'étape qui correspond au raffinage.
- **Généralisation :** R_i est le raffinage d'une des étapes introduites dans un raffinage de niveau R_{i-1}.
- **Raffinages :** L'ensemble des raffinages constitue un arbre dont la racine est R0 et les feuilles sont des étapes élémentaires.
  R_i indique un raffinage de profondeur i dans l'arbre des raffinages.
- **Remarque :** On peut arrêter de raffiner si une étape est déjà connue ou si sa réalisation ne pose pas de problème (subjectif, dépend de la cible).

Continuer le raffinage du pgcd

```
R2 : Raffinage De « Déterminer le pgcd a et b »
  | na <- a      -- variables auxiliaires car a et b sont en in
  | nb <- b      -- et ne peuvent donc pas être modifiées.
  | TantQue na et nb différents Faire                    na, nb: in
  |   | Soustraire au plus grand le plus petit           na, nb: in out
  | FinTQ
  | pgcd <- na  -- pgcd était en out, il doit être initialisé.
```

```
R2 : Raffinage De « Afficher le pgcd »
  | Écrire("pgcd = ")
  | Écrire(pgcd)
```

```
R2 : Raffinage De « Saisir deux entiers »
  |    ...
```

NB : Un raffinage peut utiliser des structures de contrôle (**Si**, **TantQue**...).

Continuer le raffinage du pgcd

```
R3 : Raffinage De « na et nb différents »
  | Résultat <- na <> nb

R3 : Raffinage De « Soustraire au plus grand le plus petit »
  | Si na > nb Alors
  |    | na <- na - nb
  | Sinon
  |    | nb <- nb - na
  | FinSi
```

Remarque : Quand on raffine une expression (premier R3), on indique la *valeur* de cette expression en initialisant **Résultat**.

Qualités 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 étapes introduites doivent avoir un niveau d'abstraction homogène.
- La séquence des étapes doit pouvoir s'exécuter logiquement.
- Ne pas employer de structures de contrôle déguisées (si, jusqu'à) **mais** on peut (doit) utiliser des structures de contrôle (**`Si`**, **`TantQue`**, **`Répéter`**...) !
- N'utiliser qu'une seule structure de contrôle par raffinage.

Remarque : Certaines de ces règles sont subjectives !
L'important est que vous puissiez expliquer et justifier les choix faits.

Vérification d'un raffinage

- A-t-on utilisé des verbes à l'infinitif pour les étapes ?
- Pour être correct, un raffinage doit répondre à la question COMMENT !
- Pour vérifier l'appartenance d'une étape *e* au raffinage d'une étape *s*, se demander POURQUOI on fait cette étape *e*.
  $\Longrightarrow$ Ceci permet d'identifier :
  - soit une étape intermédiaire entre *e* et *s* ;
  - soit que *e* n'est pas une sous-étape de *s* (donc pas à sa place).
- Utiliser les flots de données :
  - pour vérifier les communications entre niveaux :
    - les sous-étapes doivent produire les résultats de l'étape ;
    - les sous-étapes peuvent (doivent) utiliser les entrées de l'étape.
  - au sein d'un niveau : enchaînement des **`in`**, **`out`** et **`in out`**. On ne peut utiliser une donnée (**`in`**) que si elle a été produite (**`out`**) avant.

Dictionnaire des données

La liste des données (variables) utilisées avec leur signification.

Algorithme

Un *algorithme* est la mise à plat du raffinage :

- R0 devient le commentaire général de l'algorithme.
- En faisant un parcours en profondeur d'abord, de la gauche vers la droite,
  - les étapes élémentaires sont les instructions ;
  - les étapes intermédiaires deviennent des commentaires.

Remarque : L'obtention de l'algorithme est directe si les derniers niveaux du raffinage ne contiennent que des étapes élémentaires.
Sinon, il faut expliciter ces étapes non raffinées !

Le programme

C'est la traduction de l'algorithme dans un langage de programmation.

Algorithme du pgcd

```
1   Algorithme pgcd_euclide
2       -- Afficher le pgcd de deux entiers strictement positifs
3   Variables
4       a, b: Entier          -- deux entiers saisis au clavier
5       pgcd: Entier          -- le pgcd de a et b
6       na, nb: Integer       -- utilisées pour le calcul du pgcd
7   Début
8       -- Saisir deux entiers
9       ...
10      { (a > 0) Et (b > 0) }
11
12      -- Déterminer le pgcd de a et b
13      na <- a     -- variables auxiliaires car a et b sont en in
14      nb <- b     -- et ne peuvent donc pas être modifiées.
15      TantQue na <> nb Faire      -- na et nb différents
16          -- Soustraire au plus grand le plus petit
17          Si na > nb Alors
18              na <- na - nb
19          Sinon
20              nb <- nb - na
21          FinSi
22      FinTQ
23      pgcd <- na  -- pgcd était en out, il doit être initialisé.
24
25      -- Afficher le pgcd
26      Écrire("pgcd = ")
27      Écrire(pgcd)
28  Fin
```

Programme du pgcd en Pascal

```
1   program pgcd_euclide;
2       (* Afficher le pgcd de deux entiers strictement positifs *)
3   var
4       a, b: Integer;      (* deux entiers saisis au clavier *)
5       pgcd: Integer;      (* le pgcd de a et b *)
6       na, nb: Integer;    (* utilisées pour le calcul du pgcd *)
7   begin
8       (* Saisir deux entiers *)
9       ...
10      { (a > 0) Et (b > 0) }
11
12      (* Déterminer le pgcd de a et b *)
13      na := a;    (* variables auxiliaires : ceci permet *)
14      nb := b;    (* de conserver les valeurs saisies *)
15      while na <> nb do           (* na et nb différents *)
16          (* Soustraire au plus grand le plus petit *)
17          if na > nb then
18              na := na - nb
19          else
20              nb := nb - na;
21      pgcd := na;
22
23      (* Afficher le pgcd *)
24      write('pgcd = ');
25      writeln(pgcd);
26  end.
```

Programme du pgcd en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* Afficher le pgcd de deux entiers strictement positifs */
int main() {
    /* Saisir deux entiers */
    int a, b;   /* deux entiers saisis au clavier */
    int pgcd;   /* le pgcd de a et b */
    // ...
    assert(a > 0 && b > 0);

    /* Déterminer le pgcd de a et b */
    int na = a; /* variables auxiliaires : ceci permet */
    int nb = b; /* de conserver les valeurs saisies */
    while (na != nb) {          /* na et nb différents */
        /* Soustraire au plus grand le plus petit */
        if (na > nb) {
            na = na - nb;
        }
        else {
            nb = nb - na;
        }
    }
    pgcd = na;

    /* Afficher le pgcd */
    printf("pgcd = %d\n", pgcd);

    return EXIT_SUCCESS;
}
```

Programme du pgcd en Java

```
1   /** Afficher le pgcd de deux entiers strictement positifs */
2   class Pgcd {
3       public static void main(String[] args) {
4           // Saisir deux entiers
5           int a, b;       // deux entiers saisis au clavier
6           ...
7           assert a > 0 && b > 0;
8
9           // Déterminer le pgcd de a et b
10          int na = a;     // variables auxiliaires : ceci permet
11          int nb = b;     // de conserver les valeurs saisies
12          while (na != nb) {              // na et nb différents
13              // Soustraire au plus grand le plus petit
14              if (na > nb) {
15                  na = na - nb;
16              } else {
17                  nb = nb - na;
18              }
19          }
20          int pgcd = na;          // le pgcd de a et b
21
22          // Afficher le pgcd
23          System.out.println("pgcd = " + pgcd);
24      }
25  }
```

Attention : Ce n'est pas un programme Java représentatif !

Programme du pgcd en Python

```
1   #  Auteur   : Xavier Crégut <nom@enseeiht.fr>
2   #  Version  : 1.5
3   #  Objectif :
4   #       Calculer le pgcd de deux nombres entiers strictement positifs
5
6   """
7   Calcul du pgcd de deux entiers strictement positifs saisis au
8   clavier (pas de vérification des entrées).  Le résutat est affiché
9   sur l'écran.
10  """
11  # saisir les valeurs de a et b
12  # Attention, aucun contrôle n'est fait que les entiers saisis !
13  print('Donnez deux valeurs entières strictement positives.')
14  a = int(input('A = '))
15  b = int(input('B = '))
16
17  # calculer le pgcd (algorithme d'Euclide)
18  while a != b:
19          if a > b:
20                  a = a - b
21          else:
22                  b = b - a
23
24  # afficher le résultat
25  print('pgcd = ', a);
26
27  # Tests effectués
28  # 10                15          ->      5
29  #  2                 3          ->      1
30  #  1        2000000000          ->      1 (très, très long !)
31  # -1                 0          ->      boucle !
```

Tester le programme

Tester : processus qui permet de confirmer que le programme est correct et ne contient pas d'erreur !

Tester : processus d'exécution d'un programme avec l'intention de découvrir des erreurs !

Que pensez-vous de ces deux affirmations ?

Tester le programme (suite)

Deux grands types de tests :

- *tests fonctionnels :* le programme fait-il ce qu'on veut qu'il fasse ? (programme vu comme une « boîte noire »)
  - chez le fournisseur : par rapport à la spécification
  - chez le client : par rapport au besoin initial
- *tests structurels :* toutes les parties du code ont-elles été exercées ? (programme vu comme une « boîte blanche »)
  - test unitaire : feuille dans la conception détaillée
  - test intégration : nœud dans la conception détaillée

Tester le programme (suite)

Principe : Échantillonnage qui repose sur l'hypothèse implicite que si le programme testé fournit des résultats corrects pour l'échantillon choisi, il fournira aussi des résultats corrects pour toutes les autres données.
Problème : Comment choisir l'échantillon ?
Dans le cas de tests structurels, on peut s'appuyer sur la notion de taux de couverture. Par exemple, a-t-on exécuté au moins une fois toutes les instructions ? Est-on passé au moins une fois par tous les enchaînements.
Pb de l'oracle : Comment savoir que le résultat du programme est correct ?
Conception Dirigée par les Tests : Ecriture des tests durant la spécification, la conception et la programmation. Les tests font partis explicitement des éléments de spécification, conception et programmation.
Attention : On peut commencer à tester dès l'écriture d'un raffinage (test d'intégration avec des composants bouchons (mockup)).

Assistance à la conception dirigée par les tests

Intégration des tests dans les programmes : Automatisation de l'exécution des tests

- module *doctest :* les tests sont écrits comme des commentaires du programme (paramètres et résultats attendus)
- module *unittest :* les tests sont écrits comme des méthodes particulières des classes avec une convention de nommage (inspirée de *junit* en *Java*)

Principe : Intégration continue (approche agile). Exécution systématique des tests lors de l'intégration des programmes dans le gestionnaire de version.
Principe : Non régression. Chaque modification du fichier conduit à une ré-exécution des tests.
Pb des programmes interactifs : Comment automatiser leur tests ?
Introduire lors du raffinage une séparation entre les interactions et les calculs. Test automatique des parties calculs. Test manuel des parties interactions.

Contrat et Programmation défensive

Objectif : Maitriser l'utilisation de composants prédéfinis.
Contrat : Contraintes sur les paramètres (préconditions) et les résultats (postconditions) d'une partie d'un programme (usuellement une fonction ou une méthode) pour assurer son intégration correcte dans le reste du programme.
Programmation défensive : Les éléments du contrat sont intégrés dans la partie de programme en début et fin (intérieur de la fonction ou la méthode).
Programmation offensive : Les éléments du contrat sont intégrés dans les contextes d'utilisation (avant et après l'appel de la fonction ou la méthode).
Preuve par analyse statique : Les éléments du contrat sont intégrés comme annotations utilisées par des outils de preuve automatique
Génération des harnais de programmation défensive : Outil *PyContracts* module *contracts*

Exemple de programme prouvé automatiquement

```
parameter p : int ref
parameter q : int ref

logic pgcd : int , int -> int
axiom maxeq : forall a : int. max(a,a) = a
axiom maxlt : forall a : int. forall b : int. a < b -> max(a,b) = b
axiom maxgt : forall a : int. forall b : int. a > b -> max(a,b) = a

logic max : int , int -> int
axiom pgcdeq : forall a : int. pgcd(a,a) = a
axiom pgcdlt : forall a : int. forall b : int. a < b -> pgcd(a,b) = pgcd(a,b-a)
axiom pgcdgt : forall a : int. forall b : int. a > b -> pgcd(a,b) = pgcd(a-b,b)

let main (a,b:int) = { a > 0 and b > 0 }
p :=  a;
q := b;
while (!p <> !q) do
{ invariant pgcd(p,q) = pgcd(a,b) and p > 0 and q > 0 variant max(p,q) }
    if (!p <= !q) then
        q := !q - !p
    else
        p := !p - !q
done
{ p = q and p = pgcd(a,b) }
```

Exceptions et Gestion des erreurs

Objectif : Séparer la gestion des erreurs du fonctionnement correct du programme
Signaler une erreur : instruction `raise NomErreur`
Traiter une erreur : instruction

```
try :
    # partie qui peut signaler une erreur
except NomErreur :
    # traitement des erreurs
except (Erreur1, Erreur2, ...) :
    # traitement des erreurs
else :
    # aucune erreur ne s'est produite
finally :
    # dans tous les cas
```

Comment construire un algorithme ?

Exercice :
Écrire un algorithme qui structure les étapes nécessaires pour construire un algorithme. On fera apparaître les flots de données au moins pour les premiers niveaux de raffinage.