Leçon 3 : Tests de programme et inspection de code

Niveau : MPII Pré-requis : Notion de fonction
Base d'arithmétique

I - Introduction

I-1) Qu'est-ce qu'un test ?

Tester est un anglicisme pour le mot français essayer (ou éprouver). Me soumettant à la folie anglomane ambiante, je garderai ce mot, soucieuse de ma cohérence avec le monde extérieur.

Définition 1 : Tester un programme consiste à essayer d'y trouver des erreurs

Remarque 2 : On ne cherche pas ici à prouver directement que le programme est correct, mais à prouver qu'on n'arrive pas à se rendre compte qu'il est incorrect

I-2) Données de tests

Définition 3 : Une donnée de test est un couple (valeur d'entrée, valeur de sortie) où, à l'évidence, le deuxième élément représente la valeur de sortie attendue de la fonction appelée sur le premier élément.

Définition 4 : Un jeu de données de test (ou jeu de tests) est alors un ensemble de tels couples, permettant de vérifier la validité du programme sur certaines entrées

Remarque 5 Certaines sorties (attendues) peuvent être des erreurs.

Exemple 6 Un jeu de tests pour une fonction calculant le pgcd de deux nombres peut-être { ((1,2),1), ((-3,6),3), ((0,0),0), ((2,2.45), Erreur de type) }

I-3) Types de tests

Il existe deux types de tests :
- Les tests en boîtes noires. On ne connaît pas le code de la fonction, on peut simplement l'appeler
- Les tests en boîtes blanches. On connaît le code et on génère un jeu de tests en fonction

II - Tests en boîtes noires

II-1) Caractéristiques

Pour un test en boîtes noires, comme on ne connaît pas le code, il faut tester beaucoup de données. Idéalement, toutes, mais cela se trouve souvent impossible

Exemple 7 : On peut tester toutes les valeurs d'une fonction qui implémente une fonction booléenne mais pas celle de notre fonction calculant le pgcd

Viennent alors deux problèmes : Générer les données d'entrée et faire les tests sur ces données.

II-2) Générer des données d'entrée

Dans de nombreux cas, ne pouvant pas essayer toutes les données d'entrée, on va devoir faire des choix.

La première approche consisterait à générer des valeurs aléatoires dans un domaine et espérer en prendre suffisamment pour détecter les éventuels bogues.

Principe 8 : Une approche plus maline serait de partitionner le domaine puis appliquer l'approche naïve sur chaque domaine.

Exemple 9 Pour le calcul de pgcd (a, b), on peut partitionner le domaine d'entrée suivant les positions relatives de a, b et

0 (donc $a \leq 0 \leq b$, $0 \leq a \leq b$, $0 \leq b \leq a$, $a \leq b \leq 0$, $b \leq a \leq 0$ et $b \leq 0 \leq a$)

Remarque 10: On se contente souvent de prendre un seul test par classe.

Remarque 11: Le choix du partitionement est totalement arbitraire et doit donc être fait selon les différentes natures de traiter un problème.

Une fois cela fait, il est très commun que les erreurs puissent venir des cas limites.

Principe 12: On essaye de se placer au limite des domaines, pour vérifier ces cas là.

Exemple 13: Pour l'exemple précédent, on testerai les cas d'égalité: $0 \leq a = b$, $a = 0 = b$, $a = 0 \leq b$ etc... (et en effet le cas (0,0) semble très spécifique).

II-3) Utiliser les valeurs de sorties efficacement

Néanmoins, maintenant que l'on a des valeurs d'entrées, il faut, pour avoir notre jeu de tests également les valeurs de sortie.

Exemple 14: Si on veut générer la sortie du pgcd sur des entrées que l'on a pris au hasard, il nous faudrait déjà un programme qui calcule le pgcd

On a alors plusieurs méthodes:

1) Générer un jeu de tests à la main

2) Utiliser un programme moins performant mais que l'on sait correct.

Exemple 15: Si on calcule le pgcd par soustraction successives on peut tester en calculant le pgcd en prenant le plus grand nombre positif divisant a et b, en les testant tous.

3) Ne pas calculer la réponse mais simplement vérifier que la réponse fourni est correcte

Exemple 16: Si on a un programme qui nous donne la décomposition en facteurs premier d'un nombre, il nous suffit de tester la primalité de chaque sortie et de vérifier que leur produit fait l'entrée.

> Développement 1: Méthode probabiliste pour vérifier le produit de matrices en $O(n^2)$

III- Tests en boîtes blanches

III-1) Graphe de flot de contrôle

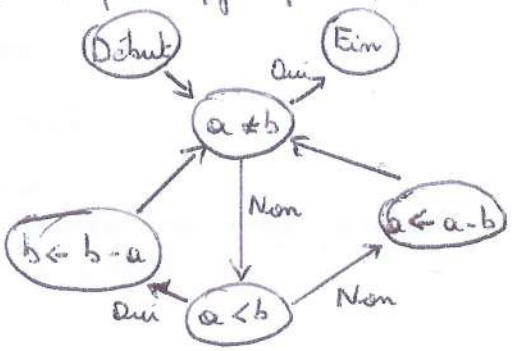
Pour un test en boîtes blanches, on connaît le code et on veut générer des données d'entrées en fonction de ce code là.

Pour cela on extrait du code le graphe de flot de contrôle

Définition 17: Le graphe de flot de contrôle est un graphe où chaque boîte contient des lignes du code, et les boîtes sont reliés si on peut exécuter l'une puis l'autre.

Exemple 18: On prend l'exemple du pgcd pour $a, b \in \mathbb{N}^*$

```
Tant que a ≠ b .
    Si a < b alors
        b ← b - a
    Sinon
        a ← a - b
Renvoyer a
```



Ne rien écrire hors du cadre.

III-2) Utilisation du graphe

On essaye alors de générer un jeu de données de test qui parcourt une bonne partie du graphe.

Par exemple, un jeu couvrant :

- tous les noeuds (On veut un jeu de tests tels que au cours de tous les tests pris ensemble, chaque noeud du graphe est parcouru au moins une fois)
- tous les arcs
- tous les chemins.

Exemple 19 : Sur l'exemple 18, $\{((1,1),1),((1,3),3)\}$ ne teste pas tous les noeuds quand $\{((1,3),3),((3,1),3)\}$ teste tous les noeuds et tous les arcs mais pas tous les chemins

Remarque 20 : Quand il y a une boucle, tous les chemins peut être un critère infini. On peut alors se limiter aux chemins d'une certaine taille

Remarque 21 : Parfois les critères sont insatisfiables

Remarque 22 : Aucun de ces critères ne garantissent la validité d'un algorithme

Développement 2 : Intérêts et insuffisances de ces critères

III-3) Test exhaustif de condition

Une autre approche consiste à avoir un jeu qui satisfait ou invalide toutes les conditions de toutes les manières possibles

Exemple 23 : On voit l'utilité sur l'exemple suivant :

```
int max (int a, int b){
    if (a>b || a==500)
        return a;
    else
        return b;
}
```

où pour détecter le problème (que max n'est pas le max) il faut des tests où est dans le premier if à cause de a == 500, donc des tests où a vaut 500.

IV- Pratiques pour éviter d'avoir à déboguer

* Compiler avec -Wall permettant d'activer tous les avertissements, évitant beaucoup de bogues nuls.

* Respecter les normes d'édition rendant le code beaucoup plus facile à relire par quelqu'un d'autre (par exemple pour le nommage des variables, la taille des indentations, etc...) On peut même utiliser un linter pour vérifier que les normes sont respectées.

* Faire de la programmation défensive en utilisant assert par exemple.