UFR IM²AG Université Grenoble Alpes

L3 Informatique
Année 2021 – 2022

UE ALGO5 — TD2 — Séance 2 : Validation expérimentale de programmes

Objectifs À la fin de cette séance, vous devriez être capable de :
- construire et décrire un jeu de tests adapté pour valider un programme ;
- décrire la spécification d'un programme sous forme d'un oracle (programme de vérification) ;
- réaliser l'APNÉE 2.

---

*La **validation d'un programme** (ou d'un algorithme) correspond au fait de **vérifier** que ce qu'il fait **correspond à sa spécification**.*

Concernant Algo5 (INF351) et Algo6 (INF363) :

- *en cours/TD : la validation se fait sous la forme de preuves formelles (invariants/variants d'itération, logique de Hoare)*
- *en TD2 et surtout en Apnée : travail orienté «mise en oeuvre» ; validation expérimentale sur l'algorithme implémenté (programme). Dans notre cas cette validation se réduit au test mais il faut garder à l'esprit que d'autres méthodes sont possibles (model-checking, analyse statique...)*

Question : à quoi sert-il de tester un programme ?

Réponse : à «trouver des erreurs» plus que «vérifier le bon fonctionnement». Un (ou même plusieurs) test(s) positif(s) n'est pas une preuve de validité (sauf si on peut couvrir tout le domaine, ce qui dans la pratique est très rarement possible).

On cherche ici à identifier quelques «bonnes pratiques» pour tester un programme. Normalement plusieurs de ces pratiques ne devraient être que des rappels de bon sens :
- *partir de la spécification, y identifier (i) les données d'entrée du programme, (ii) le domaine de valeurs sur lequel on souhaite tester le programme ;*
- *écrire un jeu d'essai (i.e., **plusieurs** tests). Les critères à avoir en tête sont :*
  - *la pertinence de chaque test du jeu d'essai,*
  - *la couverture du jeu d'essai, soit par rapport au domaine testé, soit par rapport au programme testé (boite blanche) ;*
- *éventuellement, un jeu d'essai peut être généré (partiellement) de manière automatique : cf Apnée Prog5 ;*
- *sauf pour les tests de robustesse, il n'est pas utile ni très pertinent de tester des valeurs en-dehors du domaine spécifié ; notamment, avec des valeurs ne remplissant pas les préconditions*
- *à partir du jeu d'essai :*
  - *exécuter et vérifier visuellement la validité du résultat*
  - *ou bien, écrire un oracle : un programme vérifiant lui-même que le résultat est conforme à la spécification.*

Exercice 1.

Corrigé —

Le but est ici à la fois de réfléchir au contenu d'un jeu d'essai à partir d'une spécification, et d'écrire quelques algorithmes simples sous la forme d'oracles.

—Concernant les jeux d'essais : le principe n'est pas de donner une liste de valeurs, mais les propriétés de ces valeurs (les différents cas possibles, les cas limites).
—concernant les oracles : on se rend assez vite compte que l'oracle peut être plus complexe (en terme de coût et de réflexion) que le programme à tester. Parfois le plus simple est de réécrire le programme pour comparer les résultats. Noter que c'est une pratique admise :
  —on peut comparer le résultat d'un algorithme à une version plus coûteuse mais moins compliquée à programmer
  —dans certains domaines (systèmes critiques), on fait réaliser la même fonction par deux équipes indépendantes : parfois ces deux fonctions sont utilisées en production, pour minimiser la probabilité de défaillance (tolérance aux fautes)

Si l'on ne souhaite pas réécrire la fonction (ce qu'on ne fera pas dans ce TD...), il peut être nécessaire de sélectionner, dans la spécification, ce que l'on veut tester. On en revient à ce qui était dit en introduction : le test ne permet «que» de trouver des erreurs et n'est pas une preuve de validité...

Pour chacune des fonctions ou procédures suivantes :

Q 1. décrire un jeu d'essai permettant de tester la fonction ;

Q 2. écrire un oracle permettant de vérifier que le résultat donné par la fonction correspond à sa spécification.

```
RacineCarrée(X)
{ donnée : X : un entier ≥ 0
  résultat : un entier, renvoie la partie entière de la racine carrée de X
}
```

Corrigé —

Jeu d'essai :
—valeurs particulières : 0, 1
—valeurs quelconques :
  —Au moins un carré : 4, 9, 16, 25 ...
  —Au moins un non-carré : 3, 5, 6, 7, 15, 352, ...
Oracle :

```
OracleRacineCarree(X) :
    Y <- RacineCarree(X)
    renvoyer (Y*Y <= X) et ((Y+1)*(Y+1) > X)
```

```
Trier(T)
{ donnée-résultat : T : un tableau sur [1..n] d'entiers
  post-condition : trie le tableau T
}
```

Corrigé —

Jeu d'essai :
—tableaux de tailles différentes (dont valeurs particulières : tableau de taille 0, 1)
—pour des tableaux de petite taille, on peut tester toutes les permutations d'un tableau donné ;
—valeurs particulières : tableau déjà trié, trié par ordre croissant/décroissant, valeurs identiques...
Oracle : il faut vérifier que

1. le tableau résultat est ordonné (facile),
2. le tableau résultat est une permutation du tableau initial (plus difficile).

Ici, il devient presque raisonnable de programmer un tri simple et de comparer les résultats (sauf si existence de valeurs identiques !).

```
OracleTrier(T) :
    Tinit : tableau sur [1..n] d'entiers
    P : tableau sur [1..n] de booléens
    Tinit <- T
    Trier(T)
    // T est ordonné
    i <- 2
    tant que (i <= n) et puis (T[i] >= T[i-1])
        i <- i + 1
    Tordonné <- (i > n)

    // T permutation de Tinit
    pour i de 1 à n
```

```
14          P[i] <- faux
    i <- 1
16  EstPermutation <- vrai
    tant que (i <= n) et EstPermutation
18      j <- 1
        tant que (j <= n) et puis (P[j] ou (T[i] <> Tinit[j]))
20          j <- j + 1
        si (j <= n) alors
22          P[j] <- vrai
        sinon
24          EstPermutation <- faux
    renvoyer (Tordonné et EstPermutation)
```

```
1 RechercherChaine(T,P)
  { données : T : un tableau sur [1..n] de caractères, P : un tableau
3  sur [1..m] de caractères, m ≤ n
   résultat : un entier : si P est une sous−chaîne de T, alors renvoie une
5  position de cette sous−chaîne dans T, 0 sinon
  }
```

Corrigé —

Bien décomposer le problème en deux cas (souvent seul le cas positif est testé !) :

(1) si la fonction trouve la chaîne, il faut vérifier que l'indice donné correspond à la sous-chaîne P (la spécification ne dit rien concernant la possibilité que P apparaisse plusieurs fois : il n'y a pas besoin par exemple, de tester que c'est la première occurrence)

Jeu d'essai : chaînes et sous-chaînes de différentes longueurs (cas particuliers : longueur de 0 et 1), sous-chaînes à différents endroits de la chaîne, préfixes de la sous-chaîne apparaissant à d'autres endroits dans la chaîne.

(2) si la fonction ne trouve pas la chaîne, il faut vérifier que T ne contient effectivement pas P

Jeu d'essai : chaînes et sous-chaînes de différentes longueurs, préfixes de la sous-chaîne apparaissant à différents endroits dans la chaîne.

Vérifier (2) revient à programmer un algo de recherche de sous-chaîne : la version naïve peut suffire.

Oracle :

```
OracleRechercherChaine(T,P):
2   x <- RechercherChaine(T,P)

4   si x > 0 alors
        { vérifier la présence de P dans T à l'indice x }
6       i <- x
        j <- 1
8       tant que (i <= n) et puis (j <= m) et puis (T[i] = P[j])
            i <- i + 1
10          j <- j + 1
        renvoyer (j > m)
12  sinon
        { vérifier que T ne contient pas P }
14      i <- 1
        tant que (i <= (n − m + 1)) et puis (T[i..(i+m−1)] <> P[1..m])
16          i <- i + 1
        renvoyer (i > (n−m+1))
```

Exercice 2.

Soit la fonction de recherche ci-dessous :

```
RechercheDichotomique(T,X)
{ donnée : T : un tableau sur [1..n] d'entiers, X : un entier
  résultat : un entier ; renvoie l'indice de X dans T si T contient la valeur X, 0 sinon
  pré-condition : T est trié par ordre croissant
}
  g,d,i : entiers

  g <− 1
  d <− n
  i <− (g + d)/2
  tant que (T[i] ≠ X) et (g ≤ d)
     si T[i] < X alors
        d <− i−1
     sinon
        g <− i+1
     i <− (g + d)/2
  si T[i] = X alors
     renvoyer i
  sinon
     renvoyer 0
```

Q 1. Proposer un protocole permettant d'automatiser le test de cette fonction.

Corrigé —

1. On commence par écrire l'oracle : ici encore, décomposer en deux cas
   —la recherche aboutit : on vérifie que le résultat renvoyé correspond à l'indice d'une valeur de X
   —la recherche échoue : on vérifie que X n'appartient pas au tableau

```
OracleRechercheDicho(T,X):
   Y <− RechercheDichotomique(T,X)
   si Y > 0 alors
      renvoyer (Y <= n) et puis (T[Y] = X) // bien penser à tester (Y <= n) !
   sinon
      i <− 1
      tant que (i <= n) et puis (T[i] <> X)
         i <− i + 1
      renvoyer (Y=0) et (i > n)
```

2. On peut ensuite chercher à générer des tests : comme pour un jeu d'essai, il faut identifier correctement le domaine de valeurs (ici un tableau **et** un entier). Bien avoir en tête qu'on ne teste pas séparément les domaines de valeurs (en faisant varier les valeurs pour T sans faire varier X, et inversement), le domaine de test sera généralement un produit de domaines :

```
pour chaque valeur de T à tester
   pour chaque valeur de X à tester { ici X peut dépendre de T }
      si non OracleRechercheDicho(T,X) alors ...
```

   Ne pas oublier la spécification : génération de tableaux triés...

3. On peut enfin instrumenter le programme à tester : ajouter des assertions (concernant les invariants ou variants d'itération par exemple), tester la valeur de ces assertions.
