TD #4 — Tests (avec solutions)

Exercise 1. *Plus petit* `if` (exercice repris de Vladislav Tempez)

On considère la fonction suivante :

```
int branch(int cond, int x) {
    if (cond > 0) {
        x = x + 1;
    }
    if (cond <= 0) {
        x = x - 1;
    }
    return x;
}
```

1. Dessinez son graphe de flot de contrôle.
2. Caractérisez l'ensemble des chemins possibles dans ce graphe. Y a-t'il des chemins infaisables (qui ne sont réalisés par aucune exécution du programme) ?

   Parmi tous les chemins possibles, les seuls faisables sont ceux qui passent par exactement l'un des deux sommets correspondant aux deux premiers blocs de base. Il y a donc des chemins infaisables.

3. Réécrivez cette fonction avec un `else`.

```
int branch2(int cond, int x) {
    if (cond > 0) {
        x = x + 1;
    }
    else // cond <= 0
        x = x - 1;
    }
    return x;
}
```

4. Reprenez les deux premières questions pour cette nouvelle fonction.

   Tous les (deux) chemins sont à présent faisables.

5. Commentez le résultat.

   Les deux versions de la fonction renvoient exactement les mêmes résultats pour toutes leurs entrées, mais la seconde a un graphe de flot de contrôle « plus simple » : il possède moins de chemins que celui de la première et tous ses chemins sont possibles. Ceci permet de dire que d'une certaine façon la seconde version de la fonction est « meilleure » que la première.
   Remarque : écrire des programmes dont les graphes de flot de contrôle sont les plus simples possible n'est *pas une priorité absolue*, mais peut néanmoins contribuer à améliorer la lisibilité des programmes.

Exercise 2. *Test exhaustif de conditions*

Pour chacune des conditions suivantes, proposez un jeu de test permettant d'en tester exhaustivement toutes les possibilités d'évaluation (c'est à dire que l'on veut un cas de test pour toutes les « façons » dont la condition peut s'évaluer à vrai ou à faux). Proposez une correction le cas échéant, sur la base du contexte fourni et du code exécuté ou non en fonction de la condition.

1.
```
assert(pn > 0);
assert(qn > pn);
int *pa = malloc(pn * sizeof(int));
int *qa = malloc(qn * sizeof(int));
```

```
// ...
while (i < pn || i < qn) {
    pa[i] = qa[i];
    i = i + 1;
}
```

On propose : `i = 0` ; `i = pn` ; `i = qn`, ce qui par la contrainte `qn > pn` est exhaustif. Dans les deux premiers cas la condition s'évalue à `true` et l'on rentre dans le `while`. Le second de ces cas provoque un accès hors-borne dans la zone mémoire pointée par `pa`, et une condition `i < pn && i < qn` semble donc plus appropriée.

2.
```
uint64_t a;
uint64_t b;
// ...
while ((a % 2 == 0) && (b % 2 == 0)) {
    a = a / 2;
    b = b / 2;
}
```

On propose : `a = 0, b = 0` ; `a = 1, b = 0` ; `a = 0, b = 1` ; `a = 1, b = 1`. Seul le premier cas fera s'évaluer à la condition à `true`, et l'on peut remarquer que dans ce cas la boucle sera infinie. Ainsi, on pourrait éventuellement ajouter une condition `... && !(a == 0 && b == 0)`, ce qui ferait que tous les cas ci-dessus font s'évaluer la condition à `false`. On rajoute alors un nouveau cas : `a = 2, b = 2` pour qu'elle s'évalue à `true`.

3.
```
uint64_t x;
// ...
while ((x % 4 != 0) && (x % 12 == 0)) {
    // ...
}
```

Si `x` est divisible par 12 alors il l'est nécessairement aussi par 4 : aucune valeur de `x` ne permet donc de satisfaire cette condition, et le corps du `while` ne sera jamais exécuté. Le contexte ne permet pas de deviner quelle modification apporter, mais en tant que telle cette boucle est inutile.

4.
```
uint64_t x;
uint64_t y;
// ...
while ((x > 0) && (y % x == 0) && (x % 2 == 0) && (y % 2 == 0)) {
    // ...
}
```

On propose :

- `x = 0, y = 0` $\rightsquigarrow (F, U, T, T) \rightsquigarrow$ `false`
- `x = 0, y = 1` $\rightsquigarrow (F, U, T, F) \rightsquigarrow$ `false`
- `x = 3, y = 1` $\rightsquigarrow (T, F, F, F) \rightsquigarrow$ `false`
- `x = 3, y = 2` $\rightsquigarrow (T, F, F, T) \rightsquigarrow$ `false`
- `x = 2, y = 1` $\rightsquigarrow (T, F, T, F) \rightsquigarrow$ `false`
- `x = 4, y = 2` $\rightsquigarrow (T, F, T, T) \rightsquigarrow$ `false`
- `x = 3, y = 3` $\rightsquigarrow (T, T, F, F) \rightsquigarrow$ `false`
- `x = 3, y = 6` $\rightsquigarrow (T, T, F, T) \rightsquigarrow$ `false`
- `x = 2, y = 2` $\rightsquigarrow (T, T, T, T) \rightsquigarrow$ `true`

où l'on note T, F, U les valeurs de vérité des différentes sous-conditions, où U dénote une condition ne pouvant être évaluée.
On peut notamment remarquer que le cas (T, T, T, F) ne peut pas se produire : si `y` est divisible par `x` et que `x` est divisible par 2, alors `y` est nécessairement aussi divisible par 2 ; la dernière sous-condition est donc redondante et il vaut mieux la supprimer, quitte à utiliser un `assert` pour mettre l'invariant en évidence :

```
while ((x > 0) && (y % x == 0) && (x % 2 == 0)) {
    assert(y % 2 == 0);
    // ...
}
```

Exercice 3. *Couverture de CFG* (exercice adapté de Vladislav Tempez)

On considère la fonction suivante :

```
uint64_t fun(int i) {
    uint64_t j = 1;
    for ( ; i < 63 && i >= 0 ; i++) {
        j = j * 2;
    }
    return j;
}
```

1. Dessinez son graphe de flot de contrôle.
2. Caractérisez l'ensemble des chemins faisables dans ce graphe.
3. Donnez des tests permettant de couvrir tous les sommets du graphe ; tous les arcs.

   Dans les deux cas, `i = 10` (par exemple) suffit à couvrir tous les sommets et tous les arcs.

4. Est-il raisonnable de vouloir couvrir tous les chemins du graphe ? Tous les chemins à nombre de répétition de *cycles* près ? Si oui, donnez des cas de tests correspondant.

   On peut couvrir tous les chemins possibles en testant le programme pour son argument prenant toutes les valeurs dans $[\![0, 62]\!]$, et une valeur en dehors de cet intervalle : c'est donc raisonnablement envisageable (et facilement réalisable automatiquement). On peut néanmoins remarquer qu'en général tester *tous* les nombres possibles d'itération d'une boucle peut être compliqué et coûteux.

5. Que fait ce programme ?

   Il renvoie `1` pour un argument i hors de l'intervalle $[\![0, 62]\!]$, et sinon 2^{63-i}

Exercice 4. *Génération de cas de test pour une fonction de tri*

1. Proposez des spécifications (raisonnablement) précises que doit satisfaire une fonction de tri (par ordre croissant) dont la signature serait `void sort(size_t n, int a[n])`.

   On propose : soit a la valeur initiale du tableau fourni en argument à `sort` et a' sa valeur à l'issue de l'exécution de cette dernière, on doit avoir que a' est trié (par ordre croissant) et est une permutation de a.

2. Pourquoi n'est il pas suffisant pour tester `sort` de vérifier que les éléments de son argument sont triés (par ordre croissant) après appel ?

   (En reprenant les notations ci-dessus,) ceci ne permet pas de vérifier que a' est une permutation de a, ce qui est exigé par les spécifications.

On suppose disposer d'une fonction de signature `void shuffle(size_t n, int a[n])` qui modifie son argument `a` en lui appliquant une permutation aléatoire (suivant une distribution *a priori* quelconque). On suppose également disposer d'une fonction de signature `bool sorted(size_t n, int a[n])` qui renvoie `true` si son argument `a` est trié (pour l'ordre croissant), et `false` sinon.

3. Proposez (et implémentez) une approche effectuant (au moins) un test non trivial de `sort`.

   On peut générer un tableau trié à des valeurs connues, le permuter avec `shuffle`, le trier avec `sort`, et vérifier que le résultat est trié et contient les mêmes valeurs qu'initialement. Par exemple :

```
bool test_sort2000(void)
{
    int a[2000];
    for (int i = 0; i < 2000; i++)
    {
        a[i] = i;
    }
    shuffle(2000, a);
    sort(2000, a);
```

```
    if (!sorted(2000, a))
    {
        return false;
    }
    for (int i = 0; i < 2000; i++)
    {
        if (a[i] != i)
        {
            return false;
        }
    }
    return true;
}
```

Pour obtenir des cas plus variés, on pourrait initialiser le tableau à des valeurs pseudo-aléatoires (il serait alors probablement nécessaire de le copier avant l'appel à `sort` afin de pouvoir effectuer la seconde partie de la vérification).

Il existe de nombreuses façons d'appliquer une permutation (pseudo-)aléatoire à un tableau ; un algorithme du folklore est le suivant : soit N le nombre d'éléments du tableau (indicé à partir de 0), pour i allant de 0 à $N-2$, on échange l'élément i du tableau avec un élément choisi uniformément et indépendamment parmi ceux d'indices $[\![i, N-1]\!]$.

4. Écrivez une fonction `shuffle` comme ci-dessus qui utilise cet algorithme. On supposera pour cela l'existence d'une fonction de signature `size_t urand(size_t b)` qui renvoie à chaque appel un nombre tiré uniformément et indépendamment dans l'intervalle $[\![0, b]\!]$.

On propose :

```
uint64_t shuffle(size_t n, int a[n])
{
    assert(n > 0);
    for (size_t i = 0; i < n - 1; i++)
    {
        size_t ni = i + urand(n - 1 - i);
        uint64_t t = a[i];
        a[i] = a[ni];
        a[ni] = t;
    }
}
```

5. Exprimez l'algorithme précédent de façon *récursive* ; c'est à dire, expliquez comment procéder pour générer une permutation de $N > 1$ éléments en fonction d'un moyen pour générer une permutation de $N-1$ éléments.

On obtient l'algorithme suivant : si $N = 1$ il n'y a rien à faire, sinon on choisit uniformément un élément parmi ceux d'indices $[\![0, N-1]\!]$, on l'échange avec celui en début de tableau, et l'on applique récursivement l'algorithme aux éléments (nouvellement) d'indices $[\![1, N-1]\!]$ (ou dit autrement, on appelle récursivement l'algorithme sur le sous-tableau des éléments d'indice $[\![1, N-1]\!]$).

6. Montrez que pour tout $N > 0$, cet algorithme permet de générer toutes les permutations de N éléments possibles. On pourra pour cela assimiler le tableau produit en résultat à sa représentation par extension : l'élément à l'indice i indique l'image de i par la permutation.

On prouve cela par récurrence *via* un argument de comptage. On pose $\mathcal{P}(n)$: pour un tableau de taille $n \in \mathbb{N} \setminus \{0\}$, l'algorithme génère toutes les permutations de n éléments.
Initialisation ($\mathcal{P}(1)$) : l'algorithme termine immédiatement sans avoir modifié son entrée, déjà égale à l'unique permutation possible de 1 élément.

Hérédité ($\mathcal{P}(n) \Rightarrow \mathcal{P}(n+1)$) : Il existe $(n+1)!$ permutations de $n+1$ éléments, qui se partitionnent en $n+1$ sous-ensembles de $n!$ permutations : pour chaque $i \in [\![0, n]\!]$, il existe $n!$ permutations (dont on note σ_i le sous-ensemble) dont l'image de 0 est i.

Par construction l'algorithme produit des permutations dans chaque σ_i : pour tout $k \in [\![0, n]\!]$, la première étape peut échanger l'élément d'indice k avec celui d'indice 0 dans le tableau, et ne modifie plus celui-ci pour le reste de l'exécution.
Toujours par construction, l'algorithme est ensuite appelé récursivement sur les n éléments restants. Par hypothèse de récurrence, il peut ainsi produire toutes les $n!$ permutations possibles de ces éléments ; il produit ainsi $(n+1)!$ permutations distinctes au total, soit toutes les permutations de $(n+1)!$ éléments.

7. Proposez (à haut niveau) une approche de tests en boîte noire d'une fonction `shuffle` censée implémenter cet algorithme.

Puisque l'algorithme est censé produire toutes les permutations possibles, on peut envisager de vérifier que ceci est bien le cas, pour de petites valeurs de n. Ceci ne peut raisonnablement s'envisager que pour $n \lessapprox 12$, et implique d'être capable de décider efficacement si une permutation a déjà été observée ou non. Cette dernière tâche peut se réaliser en utilisant une (implémentation efficace d'une) *structure de donnée d'ensemble*, dont nous parlerons au second semestre.

Digression

On peut montrer (comment ?) que l'algorithme précédent permute uniformément les éléments de son argument a (à condition que les indices soient *réellement* tirés uniformément et indépendamment dans les intervalles requis, ce qui n'est essentiellement jamais le cas en pratique) ; ce n'est pas le cas (pourquoi ?) de la variante suivante : pour i allant de 0 à $N-1$, on échange l'élément i du tableau avec un élément choisi uniformément parmi ceux d'indices $[\![0, N-1]\!]$. Pourtant, c'est cette dernière variante qui est utilisée (avec $N = 256$) dans le cœur de l'algorithme de chiffrement RC4 (alors même qu'en cryptographie, on préfère quand les choses sont uniformes…).

8. À votre avis, en quoi ce second algorithme peut-il être supérieur au premier dans le contexte d'utilisation de RC4 ?

Il est bien plus efficace de tirer uniformément un nombre aléatoire dans l'intervalle (fixe) $[\![0, 255]\!]$ (il suffit par exemple de tirer huit bits uniformément et indépendamment) qu'un nombre aléatoire dans la suite d'intervalles (variables) $[\![0, 255]\!]$, $[\![1, 255]\!]$, etc. : sur une architecture classique, la plupart de ces tirages nécessiteront l'utilisation d'une division ou d'une multiplication et « consommeront » plus de bits aléatoires que pour l'intervalle $[\![0, 255]\!]$ (pourtant plus grand !)