TD #4 — Tests

Exercice 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) ?
3. Réécrivez cette fonction avec un `else`.
4. Reprenez les deux premières questions pour cette nouvelle fonction.
5. Commentez le résultat.

Exercice 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;
}
```
2.
```
uint64_t a;
uint64_t b;
// ...
while ((a % 2 == 0) && (b % 2 == 0)) {
    a = a / 2;
    b = b / 2;
}
```
3.
```
uint64_t x;
// ...
while ((x % 4 != 0) && (x % 12 == 0)) {
    // ...
}
```
4.
```
uint64_t x;
uint64_t y;
// ...
while ((x > 0) && (y % x == 0) && (x % 2 == 0) && (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.
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.
5. Que fait ce programme ?

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])`.
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 ?

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`.

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 $[\![0, b]\!]$.
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.
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.
7. Proposez (à haut niveau) une approche de tests en boîte noire d'une fonction `shuffle` censée implémenter cet algorithme.

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 ?