TD 5 : Fonctions et tableaux

Exercice 1 (Échauffement)**.**
Quelles sont les valeurs `v[0]`, `v[1]`, ... contenues dans le tableau `v` à la fin de l'exécution du programme suivant :

```
    vector<int> v;                              // Declaration
    v = vector<int>(5);                         // Allocation
    for ( int i = 0; i < v.size(); i++ ) { // Initialisation
        v[i] = i*i;
    }
```

Correction : Le tableau contient les éléments 0, 1, 4, 9, 16 (les premiers carrés).

Exercice 2 (Mystère)**.**
On considère la fonction mystère suivante :

```
int mystere(vector<int> t) {
    int m = t[0];
    for ( int i = 1; i < t.size(); i++ ){
        if ( m < t[i] ) {
            m = t[i];
        }
    }
    return m;
}
```

(1) Exécuter pas à pas le programme suivant :

```
    vector<int> t = { 5, -1, 3, 7, -4, 8, 4 };
    int resultat = mystere(t);
```

Quelle est la valeur de la variable `resultat` à la fin de l'exécution ? Que fait la fonction mystère ?

Correction : En exécutant pas à pas la fonction avec le paramètre donné, on a comme valeurs à chaque début de boucle :

i	m	t[i]
1	5	-1
2	5	3
3	5	7
4	7	-4
5	7	8
6	8	4
7		

Donc `resultat` vaut 8 (valeur renvoyée par la fonction `mystere` appliquée à `t`).

La fonction mystère prend en argument un tableau d'entiers et renvoie le maximum du tableau.

(2) Modifier la fonction pour qu'elle calcule le minimum d'un tableau.

Correction : Il suffit de changer le test du `if` :

```
int min(vector<int> t) {
    int m = t[0];
    for ( int i = 1; i < t.size(); i++ ){
        if ( t[i] < m )
            m = t[i];
    }
    return m;
}
```

Exercice 3 (Quelques fonctions sur les tableaux)**.**

(1) Spécifier (sous forme de documentation) et implanter (sous forme de code) une fonction qui renvoie vrai si tous les éléments d'un tableau d'entiers sont positifs, et faux sinon.

Correction :

```
/** Teste si tous les éléments d'un tableau sont positifs
 * @param tab un tableau d'entiers
 * @return true si tous les éléments de tab sont positifs, false sinon
 **/
bool positifs(vector<int> tab) {
    int taille = tab.size();
    for (int i = 0; i < taille; i++) {
        if (tab[i] < 0) return false;
    }
    return true;
}
```

(2) Spécifier et implanter une fonction qui incrémente de 1 tous les éléments d'un tableau d'entiers et renvoie le tableau.

Correction :

```
/** Incrémente de 1 tous les éléments d'un tableau et le renvoie
 * @param tab un tableau d'entiers
 * @return le tableau modifié
 **/
vector<int> incremente(vector<int> tab) {
    int taille = tab.size();
    for (int i = 0; i < taille; i++) {
        tab[i]++;
    }
    return tab;
}
```

(3) Spécifier et implanter une fonction qui teste si deux tableaux d'entiers sont égaux.

Correction :

```
/** Teste si deux tableaux sont égaux (c'est-à-dire tailles égales et
 * éléments de même indice égaux)
 * @param tab1 le premier tableau d'entiers
```

```
 * @param tab2 le deuxième tableau d'entiers
 * @return true si les deux tableaux sont égaux, false sinon
 **/
bool testEgaux(vector<int> tab1, vector<int> tab2) {
    if (tab1.size() != tab2.size()) {
        return false;
    }
    for (int i = 0; i < tab1.size(); i++) {
        if ( tab1[i] != tab2[i] ) {
            return false;
        }
    }
    return true;
}
```

(4) Spécifier et implanter une fonction qui compte le nombre d'éléments strictement plus grands qu'un seuil donné dans un tableau d'entiers.

Correction :

```
/** Compte les éléments du tableau strictement plus grands qu'un seuil donné
 * @param tab un tableau d'entiers
 * @param seuil l'entier que les éléments à dénombrer dépasse.
 * @return nombre d'éléments du tableau strictement plus grands que le seuil,
 * les doublons sont comptés plusieurs fois.
 **/
int nbElementsPlusQueSeuil(vector<int> tab, int seuil) {
    int cpt = 0;
    for (int i = 0; i < tab.size(); i++) {
        if (tab[i] > seuil)
            cpt = cpt + 1;
    }
    return cpt;
}
```

(5) Spécifier et implanter une fonction qui renvoie vrai si un tableau d'entiers est trié dans l'ordre croissant, et faux sinon.

Correction :

```
/** Teste si un tableau est trié en ordre croissant
 * @param tab un tableau d'entiers
 * @return true si le tableau est trié en ordre croissant, false sinon
 **/
bool estTrie(vector<int> tab) {
    for (int i = 0; i < tab.size() - 1; i++) {
        if (tab[i] > tab[i + 1])
            return false;
    }
    return true;
}
```

Remarquez la condition de boucle qui est ici `i < tab.size()-1` à cause de l'utilisation de `tab[i+1]`.

Notes pour les enseignants : L'exo Fibonacci suivant sera repris en TP. Ne pas forcément le corriger, mais donner des indications détaillées sur les trois versions afin qu'ils aient bien compris le principe et soient capables de les faire seuls en TP. Ils ont en général du mal à comprendre le principe (par exemple le fait que pour avoir u_n, il faut d'abord avoir calculé toutes les valeurs de u_1 à u_{n-1}, et les calculer dans cet ordre !).

Exercice 4.
La suite de Fibonacci est définie récursivement par la relation : $U_n = U_{n-1} + U_{n-2}$. Cette définition doit être complétée par une condition d'initialisation, en l'occurrence : $U_1 = U_2 = 1$. Introduite par Léonard de Pise (surnommé Fibonacci), cette suite décrit un modèle simplifié de l'évolution d'une population de lapins. Les premiers termes sont donnés par : 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ... Voir aussi `http://oeis.org/A000045`.

L'objectif est de donner trois implantations de la fonction dont la documentation et les tests sont donnés ci-dessous :

```
/** Suite de Fibonacci
 *  @param n un entier strictement positif
 *  @return le n-ième terme de la suite de Fibonacci
 **/
```

```
    CHECK( fibonacci(1) == 1 );
    CHECK( fibonacci(2) == 1 );
    CHECK( fibonacci(3) == 2 );
    CHECK( fibonacci(4) == 4 );
    CHECK( fibonacci(5) == 5 );
    CHECK( fibonacci(6) == 8 );
```

(1) Vérifier que les tests sont cohérents avec la documentation. Corriger si nécessaire.

Correction : Le 4e test n'est pas correct. Version corrigée :

```
    CHECK( fibonacci(1) == 1 );
    CHECK( fibonacci(2) == 1 );
    CHECK( fibonacci(3) == 2 );
    CHECK( fibonacci(4) == 3 );
    CHECK( fibonacci(5) == 5 );
    CHECK( fibonacci(6) == 8 );
```

(2) Implanter la fonction fibonacci avec une boucle `for` et un tableau de taille n pour stocker tous les éléments de la suite $U_1, \ldots, U_n$.

Correction :

```
int fibonacciTableau(int n) {
    if(n == 1 or n == 2) {
        return 1;
    } else {
        vector<int> u(n);
        u[0] = 1;
        u[1] = 1;
        for (int i = 2; i < n; i++) {
            u[i] = u[i-1] + u[i-2];
        }
        return u[n-1];
    }
}
```

(3) Implanter la fonction fibonacci sans tableau, en utilisant une boucle `for` et trois variables dont deux qui, au début de chaque tour de boucle, contiennent les valeurs de U_{k-1} et U_{k-2}.

Correction :

```
int fibonacciVariables(int n) {
    int u1 = 1;
    int u2 = 1;
    int tmp = 0;
    if(n == 1 or n == 2) {
        return 1;
    } else {
        for (int i = 3; i <= n; i++) {
            tmp = u2;
            u2 = u1 + u2;
            u1 = tmp;
        }
        return u2;
    }
}
```

(4) Implanter une version récursive de la fonction fibonacci, c'est-à-dire qui s'appelle elle même.

Correction :

```
int fibonacciRecursive(int n) {
    if(n == 1 or n == 2) {
        return 1;
    } else {
        return fibonacciRecursive(n-2) + fibonacciRecursive(n-1);
    }
}
```

(5) Laquelle de ces trois implantations est la plus expressive ?

Correction : La fonction récursive : elle correspond à la définition mathématique de la suite.

(6) ♣ Comparer l'efficacité des trois implantations. Quel phénomène a lieu pour la fonction récursive ? Comment pourrait-on le corriger ?

Correction : Les deux premières implantations calculent la valeur en temps linéaire bien que la première stocke inutilement les valeurs intermédiaires. La troisième implantation, bien que plus expressive, montre un nombre d'appel récursif qui croit de manière exponentielle. De plus, les mêmes valeurs vont être calculées plusieurs fois.
NB : il est possible de dessiner un arbre dont les noeuds sont des appels de la fonction récursive afin d'illustrer la croissance exponentielle. On pourra également entourer les noeuds similaires pour montrer que la même valeur est calculée plusieurs fois.
Un moyen de corriger cela est la mémoïsation : voir `https://fr.wikipedia.org/wiki/M%C3%A9mo%C3%AFsation#Exemple` pour un exemple sur la version récursive de la suite de Fibonacci.

Exercice ♣ 5.

(1) Que fait la fonction suivante ?

```
vector<int> mystere(vector<int> t) {
    int n = t.size();
```

```
    vector<int> r(n);
    for (int i=0; i<n; i++) {
        r[i] = t[n-1-i];
    }
    return r;
}
```

Correction : La fonction mystere prend en paramètre un tableau t et renvoie un nouveau tableau r qui est le miroir de t(le premier élément de r est le dernier élément de t, etc.).

(2) Un *palindrome* est un tableau comme $\{2, 8, -1, 6, -1, 8, 2\}$ qui peut se lire de la même façon dans les deux sens. Écrire une fonction qui teste si un tableau est un palindrome en utilisant les fonctions déjà vues dans le TD.

Correction :

```
bool palindrome(vector<int> t) {
    return testEgaux(t, mystere(t));
}
```

(3) Réécrire une fonction `palindrome`, testant si un tableau est un palindrome, sans utiliser les fonctions déjà vues dans les exercices précédents.

Correction :

```
bool palindromeIndependant(vector<int> t) {
    int n = t.size();
    for ( int i = 0; i < n/2; i++) {
        if (t[i] != t[n-i-1])
            return false;
    }
    return true;
}
```

(4) Commenter les avantages et inconvénients des deux implantations.

Correction : La première est plus concise et expressive (proche de la définition mathématique). La deuxième est plus optimisée.