TD 5 : Fonctions et tableaux

Exercice 1 (Échauffement)**.**
Qu'affiche le programme suivant :

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v;                          // Declaration
    v = vector<int>(5);                     // Allocation
    for ( int i = 0; i < v.size(); i++ ) // Initialisation
        v[i] = i*i;
    cout << v[0] << v[1] << v[2] << v[3] << v[4] << endl;
    return 0;
}
```

Correction Le programme affiche 014916 (ce sont les valeurs des éléments du tableau v).

Exercice 2 (Mystère)**.**
On considère le fragment de programme suivant :

```
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;
}

int main() {
    vector<int> t = { 5, -1, 3, 7, -4, 8, 4 };
    cout << mystere(t) << endl;
    return 0;
}
```

(1) Exécuter pas à pas ce programme, que fait-il ? Que fait la fonction mystere ?
Correction Ce programme affiche 8. La fonction mystère prend en argument un tableau d'entiers et renvoie le maximum du tableau.

(2) Modifier le programme pour qu'il affiche le minimum et le maximum d'un tableau.
Correction

```
#include <iostream>
#include <vector>
using namespace std;

/** Renvoie l'élément le plus grand du tableau **/
int max(vector<int> t) {
```

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

/** Renvoie l'élément le plus petit du tableau **/
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;
}

int main() {
    vector<int> tab = { 5, -1, 3, 7, -4, 8, 4 };
    cout << "Le max est "<< max(tab) <<endl;
    cout << "Le min est "<< min(tab) <<endl;
    return 0;
}
```

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

(1) Spécifier et écrire une fonction qui affiche les éléments d'un tableau d'entiers.

Correction

```
/** Affiche un tableau
 * @param tab un tableau d'entiers
 **/
void afficheTableau(vector<int> tab) {
    for(int i=0;i<tab.size();i++) {
        cout<<tab[i]<<'\t';
    }
    cout<<endl;
}
```

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

Correction

```
/** Teste si deux tableaux sont égaux
    (tailles égales, éléments de même indice égaux)
    * @param tab1 le premier tableau d'entiers
    * @param tab2 le deuxième tableau d'entiers
    * @return vrai si les deux tableaux sont égaux, faux sinon
    **/
bool testEgaux(vector<int> tab1, vector<int> tab2) {
    int taille = tab1.size();
    if(tab2.size() != taille) {
        return false;
    }
    else{
        for(int i=0; i<taille; i++) {
            if(tab1[i] != tab2[i]) {
```

```
                return false;
            }
        }
        return true;
    }
}
```

(3) Spécifier et écrire une fonction qui compte le nombre d'éléments plus grands que 10 dans un tableau d'entiers.

Correction

```
/** Compte le nombre d'éléments plus grands que 10 dans le tableau
 * @param tab un tableau d'entier
 * @return nombre d'éléments plus grand que 10
 */
int plusGrandQueDix(vector<int> tab) {
    int res = 0;
    for(int i=0; i<tab.size(); i++) {
        if(tab[i] > 10)
            res = res + 1;
    }
    return res;
}
```

(4) Spécifier et écrire 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'entier
 * @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;
}
```

Exercice 4.
On rappelle que 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 l'évolution d'un modèle simple de 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 d'implanter 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
 **/
int fibonacci(int n) {
    // À faire : Écrire le code ici !
```

```
}

void fibonacciTest() {
    ASSERT( fibonacci(1) == 1 );
    ASSERT( fibonacci(2) == 1 );
    ASSERT( fibonacci(3) == 2 );
    ASSERT( fibonacci(4) == 4 );
    ASSERT( fibonacci(5) == 5 );
    ASSERT( fibonacci(6) == 8 );
}
```

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

Correction

```
    ASSERT( fibonacciTableau(4) == 3 );
```

(2) Première implantation : en utilisant 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) Deuxième implantation : 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) Troisième implantation : en utilisant une fonction récursive, 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 ?

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;
}
```

(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.

(3) Écrire une fonction `palindrome` indépendamment des exercices précédents.

(4) Commenter les avantages et inconvénients de chacune.

Correction

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

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;
}

void palindromeTest() {
    ASSERT (     palindrome({ 1,2,3,4,5,4,3,2,1 }) );
    ASSERT (     palindrome({ 1,2,3,4,4,3,2,1   }) );
    ASSERT ( not palindrome({ 1,2,3,4,4,3,2,0   }) );
    ASSERT ( not palindrome({ 1,2,3,4,5,3,3,2,1 }) );
    ASSERT ( not palindrome({ 1,2,3,4,3,3,2,1   }) );
}

int main() {
    palindromeTest();
    return 0;
}
```

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