Nom, prénom, numéro d'étudiant :

Coller ou agrafer ici | Coller ou agrafer ici | Coller ou agrafer ici

Université Paris Sud, Licence MPI, Info 111 — Examen du 18 décembre 2017 (deux heures)

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Total
␣,␣␣	␣,␣␣	␣,␣␣	␣,␣␣	␣␣,␣

Calculatrices et autres gadgets électroniques interdits.
Seul document autorisé : une feuille au format A4 avec, au recto, le résumé de la syntaxe C++ de la fiche de TD 2 et, au verso, des notes manuscrites. Pour les étudiants inscrits en Français Langue Étrangère, un dictionnaire est autorisé.
Les exercices sont indépendants les uns des autres ; il n'est pas nécessaire de les faire dans l'ordre ; ceux marqués d'un ♣ sont plus difficiles mais font bien partie du barème sur 20.
Dans un exercice, vous pouvez utiliser les fonctions des questions précédentes même si vous n'avez pas réussi à les faire.
Les réponses sont à donner, autant que possible, sur le sujet ; sinon, mettre un renvoi.

Exercice 1 (Question de cours : Fichiers)**.**

(1) Rappeler la syntaxe pour déclarer et ouvrir un fichier en lecture en C++.

Correction :

```
ifstream identificateur("nom du fichier");
```

(2) Illustrer avec un fragment de programme qui ouvre le fichier `toto.txt` en lecture

Correction :

```
ifstream fichier("toto.txt");
```

(3) On suppose que `liste_de_nombres.txt` ne contient que des entiers, séparés par des espaces. Compléter le programme pour qu'il compte le nombre d'entiers dans le fichier. Bien fermer le fichier.

Correction :

```
int i, compteur;
while ( fichier >> i )
    compteur ++;
fichier.close();
```

Exercice 2 (Course Stellaire)**.**

(1) Question préliminaire : Donner l'implémentation d'une fonction `abs` calculant et renvoyant la valeur absolue d'un entier relatif.

Correction :

```
int abs(int x) {
    if (x < 0) {
        return -x;
    } else {
        return x;
    }
}
```

Anakin et Obi-Wan font la course à bord de leurs bolides, en ligne droite, depuis un point de l'espace jusqu'à l'atmosphère de la planète Tatooine.

— La vitesse d'Anakin, impétueux et compétitif, varie en fonction de la position de son maître. **Toutes les secondes**, son radar s'actualise et lui indique la position d'Obi-Wan. Anakin adapte alors instantanément sa vitesse :
si Obi-Wan se trouve à plus de 30km de lui, Anakin vole à 3km par seconde ;
si Obi-Wan est en vue (30km ou moins) et strictement devant lui, il fonce à 8km par seconde ;
enfin si Obi-Wan est en vue mais derrière lui, il adopte une vitesse de 5km par seconde.

— Obi-Wan, plus mesuré mais efficace, a toujours une vitesse de 6km par seconde.

On repérera les positions des bolides par leur distance au point de départ en kilomètres, dans deux variables de type `int`, `anakin` et `obiwan`.

(2) Écrire la documentation de la fonction suivante.

```
int position_obi_apres_1seconde(int obiwan) {
    return obiwan + 6;
}
```

Correction :

```
/** Position d'Obi-Wan au bout d'une seconde, en fonction de sa position actuelle
 * @param obiwan: la position actuelle d'Obi-Wan en km (entier)
 * @return la nouvelle position d'Obi-Wan en km (entier)
 **/
```

(3) En utilisant la fonction `abs`, donner une implémentation de la fonction `sont_proches` ci-dessous.

```
/** Fonction renvoyant true si Anakin et Obi-Wan sont à 30km ou moins, false sinon
 * @param anakin: la position d'Anakin en km (entier)
 * @param obiwan: la position d'Obi-Wan en km (entier)
 * @return un booléen valant vrai si et seulement s'il sont à 30 km ou moins
 **/
bool sont_proches(int anakin, int obiwan) {
```

Correction :

```
bool sont_proches(int anakin, int obiwan) {
    return abs(anakin - obiwan) <= 30;
}
```

Écrire quelques tests pour cette fonction (utilisez la macro ASSERT).

Correction :

```
    }
    return temps;
```

(4) En utilisant la fonction `sont_proches`, donner l'implémentation d'une fonction `position_ani_apres_1seconde`, qui prend en arguments 2 entiers `anakin` et `obiwan`, et renvoie la nouvelle position d'Anakin au bout d'une seconde lorsque `anakin` et `obiwan` sont les positions actuelles d'Anakin et d'Obi-Wan, respectivement.

Par exemple, si `anakin` est égal à 25 et `obiwan` est égal à 20, la fonction renvoie la nouvelle position `anakin+5`, c'est-à-dire 30.

Correction :

```
int position_ani_apres_1seconde(int anakin, int obiwan) {
    bool proches = sont_proches(anakin, obiwan);
    if (not proches) {
        return anakin + 3;
    } else if (anakin >= obiwan) {
        return anakin + 5;
    } else {
        return anakin + 8;
    }
}
```

(5) Décrire ce que fait la fonction mystère ci-dessous.

```
int mystere(int foo) {
    int hop = 0;
    int buh = 0;
    for (int i = 0; i < foo; i++) {
        hop = position_ani_apres_1seconde(hop, buh);
        buh = position_obi_apres_1seconde(buh);
    }
    return hop;
}
```

Réécrire la fonction en choisissant des noms informatifs pour elle-même et pour ses variables.

Correction :

```
int position_anakin(int temps_ecoule) {
    int anakin = 0;
    int obiwan = 0;
    for (int s = 0; s < temps_ecoule ; s++) {
        anakin = position_ani_apres_1seconde(anakin, obiwan);
        obiwan = position_obi_apres_1seconde(obiwan);
    }
    return anakin;
}
```

(6) Implémenter une fonction `qui_gagne`, dont la documentation et les tests sont donnés ci-dessous, en appelant les fonctions précédentes.
Remarque : En cas d'égalité, Obi-Wan déclare gentiment Anakin vainqueur.

```
/** Fonction donnant le nom du gagnant selon la distance à l'arrivée
 * @param distance_arrivee: la distance entre le point de départ et l'arrivée en km
 * @return le nom du premier arrivé (chaîne de caractères)
 **/
string qui_gagne(int distance_arrivee) {
```

Tests :

```
    ASSERT(qui_gagne(114) == "Obi-Wan");
    ASSERT(qui_gagne(50)  == "Anakin" );
    ASSERT(qui_gagne(713) == "Anakin" );
```

Correction :

```
string qui_gagne(int distance_arrivee) {
    int anakin = 0;
    int obiwan = 0;
    while (anakin < distance_arrivee and obiwan < distance_arrivee) {
        anakin = position_ani_apres_1seconde(anakin, obiwan);
        obiwan = position_obi_apres_1seconde(obiwan);
    }
    if (anakin >= distance_arrivee) {
        return "Anakin";
    } else {
        return "Obi-Wan";
    }
}
```

(7) Modifier votre fonction pour qu'elle renvoie plutôt le temps mis par le gagnant pour atteindre l'arrivée (ne pas oublier de modifier l'entête !).

Correction :

```
int meilleur_temps(int arrivee) {
    int anakin = 0;
    int obiwan = 0;
    int temps = 0;
    while (anakin < arrivee and obiwan < arrivee) {
        anakin = position_ani_apres_1seconde(anakin, obiwan);
        obiwan = position_obi_apres_1seconde(obiwan);
```

```
        temps = temps + 1;
    }
    return temps;
}
```

Exercice 3 (tableaux 1D, complexité, tableaux de tableaux)**.**
Toute suite finie d'entiers peut être décomposée de manière unique en une suite de séquences strictement croissantes maximales. En représentant une suite dans un tableau t, le tableau

[1,2,5,7,2,6,0,5,2,4,6,7,8,9,3,4,6,1,2,7,8,9,4,2,3,1,5,9,7,1,6,6,3]

se décompose ainsi en le tableau de 13 tableaux d'entiers suivant :

[[1,2,5,7], [2,6], [0,5], [2,4,6,7,8,9], [3,4,6], [1,2,7,8,9], [4], [2,3], [1,5,9], [7], [1,6], [6], [3]]

Les premiers éléments de ces séquences sont d'abord le premier élément du tableau, puis les éléments qui sont inférieurs ou égaux à leur prédécesseur dans le tableau : $t[i] \leq t[i-1]$. Ils sont soulignés dans l'exemple ci-dessus.

(1) Implantez la fonction suivante, dont la documentation et les tests sont donnés et qui permet de compter les *descentes* du tableau, c'est-à-dire le nombre d'indices tels que $t[i] \leq t[i-1]$. Sur l'exemple, la fonction doit renvoyer 12.

```
/** Nombre de descentes
 * Renvoie le nombre d'indices dans le tableau tel que t[i] <= t[i-1]
 * @param t un tableau d'entier
 * @return le nombre de descentes du tableau
 **/
int nombreDescentes(vector<int> t) {
    int nb = 0;
    for ( int i = 1; i < t.size(); i++ ) {
        if ( t[i] <= t[i-1] ) {
            nb += 1;
        }
    }
    return nb;
}

void nombreDescentesTest() {
    ASSERT(nombreDescentes({1,2,3,1}) == 1);
    ASSERT(nombreDescentes({1,3,5,6,8}) == 0);
    ASSERT(nombreDescentes({3,5,5,2,6,7,7,1}) == 4 );
}
```

(2) Donner la complexité de cette fonction, en précisant bien le modèle de calcul.

(3) Implantez la fonction `rupture` suivante, dont la documentation et les tests sont donnés et qui, étant donné un tableau t d'entiers, renvoie un tableau contenant 0 en premier élément et les indices i des éléments de t tels que $t[i] \leq t[i-1]$. Pour le tableau donné en exemple, la fonction renvoie le tableau : [0 4 6 8 14 17 22 23 25 28 29 31 32]. **Attention !** Notez bien que le nouveau tableau contient les **indices** et non les **valeurs** du tableau de départ.

```
/** Point de rupture
 * Renvoie les indices de départ des sous-suites croissantes du tableau t
```

```
 * @param t un tableau d'entier
 * @return un tableau contenant 0 puis les indices i tels que t[i] <= t[i-1]
 **/
vector<int> rupture(vector<int> t){
    vector<int> tt = vector<int>();
    tt.push_back(0);
    for ( int i = 1; i < t.size(); i++ ) {
        if ( t[i] <= t[i-1] ) {
            tt.push_back(i);
        }
    }
    return tt;
}

void ruptureTest() {
    ASSERT(rupture({1,2,3,1}) == vector<int>({0,3}));
    ASSERT(rupture({1,3,5,6,8}) == vector<int>({0}));
    ASSERT(rupture({3,5,5,2,6,7,7,1}) ==  vector<int>({0,2,3,6,7}));
}
```

(4) Écrire une fonction `sousTableau` qui, étant donné un tableau t d'entiers et une position x, renvoie la plus longue sous-suite croissante commençant à cette position. Pour le tableau donné en exemple et la position 9, la fonction retourne le tableau [4 6 7 8 9].

```
vector<int> sousTableau(vector<int> t, int x) {
    int taille = t.size();
    vector<int> tt =  vector<int> (taille) ;
    int n = 1;
    int i = x-1;
    tt[0]= t[i];

    while ( (i<taille) and (t[i+1] > t[i]) ) {
        tt[n] = t[i+1];
        n++;
        i++;
    }

    vector<int> res = vector<int>(n);
    for ( int j = 0; j< n; j++ )
        res[j] = tt[j];

    return res;
}
```

(5) ♣ Écrire une fonction `factorisation` qui, étant donné un tableau t d'entiers, renvoie un tableau bidimensionnel d'entiers, c'est-à-dire un tableau de tableaux d'entiers, dont la i-ème ligne contient la i-ème séquence croissante maximale de nombres adjacents dans le tableau t (résultat tel que celui donné dans l'exemple). **Indication** : il est utile d'utiliser les fonctions écrites dans les questions précédentes.

```
vector<vector<int>> factorisation(vector<int> t) {

    vector<int> r = rupture(t);
    vector<vector<int>> res = vector<vector<int>> (r.size());

    for ( int i = 0; i < r.size() - 1; i++ ) {
        res[i] = vector<int>(r[i+1] - r[i]);

        for( int j = 0; j < res[i].size(); j++ )
            res[i][j] = t[r[i] + j];
    }

    res[r.size() - 1] = vector<int>(t.size() - r[r.size()-1]);

    for ( int j = 0; j < res[r.size() - 1].size(); j++ )
        res[r.size() - 1][j] = t[r[r.size()-1] + j];

    return res;
}
```

Exercice 4 (Carré magique)**.**
Un *carré magique* est une grille carrée contenant des entiers telle que la somme de chaque ligne, de chaque colonne et de chaque diagonale ait la même valeur. Un carré magique de n lignes est dit *normal* s'il contient chaque entier compris entre 1 et n^2 exactement une fois. Par exemple, le tableau suivant est un carré magique normal :

6	7	2
1	5	9
8	3	4

Dans la suite de l'exercice, il s'agit d'écrire étape par étape les fonctions utiles pour tester si un tableau bidimensionnel est un carré magique normal. Vous pouvez utiliser les fonctions des étapes précédentes, même si vous ne les avez pas implantées. Pour alléger les notations, on définit un raccourci `Grille` pour les tableaux bidimensionnels d'entiers :

```
typedef vector<vector<int>>  Grille;
```

Dans les tests, on pourra supposer que l'on a à disposition la variable suivante :

```
Grille carreMagique = { { 6, 7, 2}, {1, 5, 9}, {8, 3, 4} };
```

(1) Définir deux autres variables contenant des tableaux d'entiers à deux dimensions qui ne sont pas des carrés magiques.

Correction :

```
Grille carre = { { 6, 7, 2}, {1, 6, 9}, {8, 3, 4} };
Grille rectangle = { { 6, 7}, {1, 6}, {8, 3} };
```

(2) Spécifier (par une documentation) et implanter une fonction `estCarre` qui teste si une grille donnée en paramètre est une grille carrée (c'est-à-dire un tableau dont toutes les lignes et les colonnes ont la même longueur).

Correction :

```
/** teste si une grille est carrée
 * @param g de type Grille
 * @return le booléen true si la grille g est carrée, false sinon
 **/
bool estCarre(Grille g) {
    for ( int i = 0 ; i < g.size() ; i++ ) {
        if (g[i].size() != g.size()) {
            return false;
        }
    }
    return true;
}
```

(3) Proposer des tests pour cette fonction (en utilisant `ASSERT`).

Correction :

```
bool estCarreTest() {
    ASSERT( estCarre(carreMagique) );
    ASSERT( estCarre(carre) );
    ASSERT( not estCarre(rectangle) );
}
```

(4) Implanter une fonction `sommeLigne` qui prend en paramètres un entier l et une grille et qui renvoie la somme des éléments de la l-ième ligne de la grille. Par exemple, l'appel sur l'exemple `sommeLigne(carreMagique, 1)` renvoie $1 + 5 + 9 = 15$.

Dans la suite, on supposera que l'on dispose aussi d'une fonction `sommeColonne` analogue.

Correction :

```
int sommeLigne(Grille g, int l) {
    int somme = 0;
    for (int j = 0 ; j < g.size() ; j++)
        somme = somme + g[l][j];
    return somme;
}
```

(5) Spécifier et implanter une fonction `sommeDiagonaleMajeure` qui prend en paramètre une grille carrée et renvoie la somme des éléments de la diagonale majeure de la grille (la *diagonale majeure* est celle qui commence en haut à gauche et termine en bas à droite ; dans l'exemple ci-dessus, elle contient 6, 5 et 4).

Correction :

```
/** renvoie la somme des éléments de la diagonale majeure d'une grille
 * @param g de type Grille
 * @return la somme des éléments de la diagonale majeure de g
 **/
int sommeDiagonaleMajeure(Grille g) {
    int somme = 0;
    for (int i = 0 ; i < g.size() ; i++)
        somme = somme + g[i][i];
    return somme;
}
```

(6) On suppose que l'on dispose d'une fonction `sommeDiagonaleMineure` qui prend en paramètre une grille carrée et renvoie la somme des éléments de la diagonale mineure de la grille (la *diagonale mineure* est celle qui commence en bas à gauche et termine en haut à droite ; dans l'exemple ci-dessus, elle contient 8, 5 et 2). Écrire un test pour la fonction `sommeDiagonaleMineure` (ne pas écrire la fonction, seulement le test).

Correction :

```
void sommeDiagonaleMineureTest() {
    ASSERT( sommeDiagonaleMineure(carreMagique) == 15 );
}
```

(7) Implanter une fonction `estCarreMagique` qui prend en paramètre une grille, et renvoie `true` s'il s'agit d'un carré magique (pas forcément normal), `false` sinon.

Correction :

```
bool estCarreMagique(Grille g) {
    int somme = sommeLigne(g, 0);
    if ( !estCarre(g))
        return false;
    for (int i = 0 ; i < g.size() ; i++)
        if (somme != sommeLigne(g, i) || somme != sommeColonne(g, i))
            return false;
    if (somme != sommeDiagonaleMajeure(g) || somme != sommeDiagonaleMineure(g))
        return false;

    return true;
}
```

(8) ♣ Implanter une fonction `histogramme` qui prend en paramètre une grille `g` et qui renvoie son histogramme, c'est-à-dire un tableau `H` de n^2 entiers (avec n le nombre de lignes de `g`) tel que pour tout `v` entre 1 et n^2, `H[v-1]` contient le nombre d'occurrences de la valeur `v` dans la grille `g`.

Correction :

```
vector <int> histogramme(Grille g) {
    int n = g.size();
    vector <int> H(n*n);
    int i,j;
    for (i = 0 ; i < n*n ; i++)
        H[i]=0;
    for (i = 0 ; i < n ; i++)
        for (j = 0 ; j < g[i].size() ; j++)
            H[g[i][j]-1] += 1;
    return H;
}
```

(9) ♣ Implanter une fonction `estCarreMagiqueNormal` qui prend en paramètre une grille, et renvoie `true` s'il s'agit d'un carré magique normal, `false` sinon.
Indication : on pourra par exemple construire l'histogramme de la grille.

Correction :

```
bool estCarreMagiqueNormal(Grille g) {
    vector <int> H;
    if ( !estCarreMagique(g))
        return false;
    H = histogramme(g);
    for (int i = 0 ; i < H.size() ; i++)
        if (H[i] !=1) return false;
    return true;
}
```