[redacted]

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 12 décembre 2016 (deux heures)

Exercice 1	Exercice 2	Exercice 3	Exercice 4	Exercice 5	Exercice 6	Exercice 7	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.
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 (Cours)**.**

— Rappeler la syntaxe de la déclaration d'une fonction (entête de la fonction) en C++.

— Donner la déclaration (donc uniquement l'entête) d'une fonction qui prend en paramètres un entier et une chaîne de caractères et renvoie un booléen.

— Donner un exemple d'appel à cette fonction.

Exercice 2 (J'aimerais tant voir Syracuse)**.**
La suite de Syracuse d'un nombre entier $N > 0$ est définie de la manière suivante : $u_0 = N$ et $u_{n+1} = f(u_n)$ pour tout entier naturel $n > 0$, avec

$$f(m) = \begin{cases} \dfrac{m}{2} & \text{si } m \text{ est pair,} \\ 3 \times m + 1 & \text{si } m \text{ est impair.} \end{cases}$$

Par exemple, la suite de Syracuse du nombre $N = 14$ commence par $u_0 = 14$, $u_1 = 7$, $u_2 = 22$, $u_3 = 11$, puis continue par $34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...$

(1) Implanter en C++ la fonction f décrite ci-dessus. On rappelle que, pour tous entiers a et b, `a%b` est le reste de la division euclidienne de a par b ; il est nul si et seulement si b divise a.

```
int f(int m) {


}
```

(2) Implanter une fonction `terme` qui prend en argument deux entiers n et N et qui renvoie la valeur du terme u_n de la suite de Syracuse de N.

(3) Quelle est la complexité de votre fonction `terme` ? Vous préciserez le *modèle de complexité* : mesure de la taille du problème et choix des opérations élémentaires

(4) Implanter la fonction dont la documentation et les tests sont donnés ci-dessous. On pourra supposer que, pour tout entier $N > 0$, la suite de Syracuse de N atteint la valeur 1 (c'est-à-dire qu'il existe $n \geqslant 0$ tel que $u_n = 1$).

```
/** Temps de vol d'une suite de Syracuse
* @param N: le premier terme de la suite
* @return le plus petit entier n >= 0 tel que le n-ième terme de
*   la suite de Syracuse de N vaut 1
**/
int tempsDeVol(int N) {
|
|
|
|
|
|
|
}

void tempsDeVolTest() {
    ASSERT( tempsDeVol(1) == 0 );
    ASSERT( tempsDeVol(2) == 1 );
    ASSERT( tempsDeVol(4) == 2 );
    ASSERT( tempsDeVol(5) == 5 );
    ASSERT( tempsDeVol(14) == 17 );
    ASSERT( tempsDeVol(15) == 17 );
    ASSERT( tempsDeVol(127) == 46 );
}
```

(5) Implanter une fonction `maxTableau` qui prend en argument un tableau d'entiers et renvoie le plus grand élément du tableau.

(6) Implanter la fonction ci-dessous :

```
/** Écrit les éléments d'un tableau dans un fichier,
 *  séparés par des espaces.
 * @param t un tableau d'entiers
 * @param nomFichier le nom du fichier dans lequel écrire
**/
void ecritFichier(vector<int> t, string nomFichier) {



}
```

(7) On considère la fonction `mystere` dont le code et les tests sont donnés ci-dessous.

```
vector<int> mystere(int M) {
    vector<int> foo(M);
    foo[0] = 0;
    for ( int z = 1; z < M; z++ ) {
        foo[z] = tempsDeVol(z);
    }
    return foo;
}

void mystereTest() {
    ASSERT( mystere(16)[0] == 0 );
    ASSERT( mystere(16)[1] == 0 );
    ASSERT( mystere(16)[4] == 2 );
    ASSERT( mystere(16)[14] == 17 );
    ASSERT( mystere(16)[15] == 17 );
}
```

Écrire la documentation de cette fonction :

(8) En utilisant les fonctions des questions précédentes, implanter un fragment de programme qui écrit dans un fichier `"Syracuse.txt"` le temps de vol des suites de Syracuse des entiers de 1 à 100, puis affiche à l'écran le temps de vol maximal de ces suites.

Exercice 3 (Tableaux à deux dimension 5.5)**.**
On souhaite gérer les ventes de voitures d'une concession automobile. Ces ventes sont réalisées par plusieurs vendeurs. Dans la concession, il existe plusieurs modèles de voitures. Pour chaque vendeur, on comptabilise le nombre de voitures vendues pour chacun des modèles. Pour modéliser ce problème, on va utiliser un tableau à deux dimensions qui regroupe les informations relatives aux ventes de voitures dans une concession. Chaque ligne du tableau représente les ventes d'un vendeur (une ligne par vendeur). Chaque colonne représente les ventes d'un modèle par tous les vendeurs (une colonne par modèle). Chaque case contient le nombre de voitures d'un modèle M vendues par un vendeur V. Voici un exemple de tableau de ventes :

Vendeur	berline	4x4	électrique	van
André	0	3	2	0
Ingemar	2	3	0	1
Jean-Jérome	1	1	1	1
Cindy	5	1	0	0
Joey	1	1	2	0

Et le tableau C++ correspondant :

```
vector<vector<int>> ventes = {
    { 0, 3, 2, 0 },
    { 2, 3, 0, 1 },
    { 1, 1, 1, 1 },
    { 5, 1, 0, 0 },
    { 1, 1, 2, 0 }
};
```

On suppose que l'on dispose de deux tableaux 1D regroupant l'un les noms des vendeurs et l'autre celui des modèles, comme ceci :

```
vector<string> vendeurs = {"André","Ingemar","Jean-Jérôme","Cindy","Joey"};
vector<string> modeles = {"berline", "4x4", "électrique", "van"};
```

(1) 1 Implanter, avec sa documentation, une fonction qui prend en argument un tableau de ventes et un numéro de modèle (entier), et qui renvoie le nombre total d'exemplaires vendus pour ce modèle.

(2) .5 On considère la fonction documentée et déclarée ci-dessous :

```
/** Nombre d'exemplaires par vendeur
 * @param C: le tableau des ventes
 * @param v: le numéro d'un vendeur
 * @return le nombre d'exemplaires vendus par le vendeur v
 **/
int nbexemplairesParVendeur(vector<vector<int>> C, int v) {
```

Compléter les tests suivants avec trois ASSERTS choisis de manière pertinente :

```
void nbexemplairesParVendeurTest() {



}
```

(3) 1.5 On suppose qu'il y a M modèles de voitures et V vendeurs. Compléter l'implantation de la fonction suivante :

```
/** Construit et renvoie un tableau de ventes
 *  en lisant les données à partir du clavier
 * @param V : un entier représentant le nombre de vendeurs
 * @param M : un entier représentant le nombre de modèles de voitures
 * @return un tableau d'entiers a deux dimensions V x M
**/
vector<vector<int>> tableauVentes(int V, int M) {
}
```

(4) 1 On suppose que les données des ventes sont stockées dans un fichier ; pour l'exemple que nous avons vu plus haut, le fichier contiendrait :

```
5 4
0 3 2 0
2 3 0 1
1 1 1 1
5 1 0 0
1 1 2 0
```

Implanter la fonction suivante :

```
/** Construit et renvoie un tableau de ventes
 *   en lisant les données à partir d'un fichier
 * @param filename : string nom de fichier qui contient les ventes
 * @format fichier : la première ligne contient 2 entiers : le
 *         nombre de vendeurs (nbV) et le nombre de modèles (nbM).
 *         La suite du fichier contient le tableau des ventes
 * @return un tableau d'entiers a deux dimensions V x M
**/
vector<vector<int>> tableauVentesFichier(string filename) {
|
|
|
|
|
|
|
|
|
|
|
}
```

(5) 1.5 Implanter un fragment de programme – utilisant certaines des fonctions précédentes – qui lit le tableau de ventes depuis le fichier `vector2D-concession.txt` et affiche le nom du modèle le plus vendu.

Exercice ♣ 4 (Tri par insertion)**.**
Le tri par insertion d'un tableau `t` de taille n procède en n étapes. Au début de l'étape i, les i premières cases sont triées, on insère alors l'élément `t[i]` dans le sous tableau `t[0]`, ..., `t[i-1]` comme suit :

— calcul de la position `pos` où insérer `t[i]` ;
— décalage de tous les éléments de `t[pos]` à `t[i-1]` de une case vers la droite ;
— insertion de `t[i]` en position `pos`.

Exemple de tri par insertion du tableau contenant les éléments $6, 3, 1, 2, 3$:

6	**3**	1	2	3	on insère 3 dans [6] ;
3	6	**1**	2	3	on insère 1 dans [3,6] ;
1	3	6	**2**	3	on insère 2 dans [1,3,6] ;
1	2	3	6	**3**	on insère 3 dans [1,2,3,6] ;
1	2	3	3	6	le tableau est trié.

(1) On commence par la fonction `mystere` suivante :

```
vector<int> mystere(vector<int> foo, int truc, int abc) {
    int bla = foo[abc];
    for ( int w = abc - 1; w >= truc; w-- )
        foo[w+1] = foo[w];
    foo[truc] = bla;
    return foo;
}
```

Deviner ce que cette fonction est sensée faire, puis réécrire la fonction avec sa documentation, en choisissant des noms informatifs pour elle-même et ses variables.

(2) Compléter les tests suivants avec au moins deux appels pertinents à `ASSERT` :

```
void decalageInsereTest() {
    ASSERT( mystere({2,4,6,8}, 1, 1) == vector<int>({2,4,6,8}) );

}
```

(3) Quelle est la complexité de la fonction `mystere` ?

(4) Implanter la fonction dont le prototype, la documentation et les tests sont donnés ci-dessous :

```
/** Renvoie la position à laquelle insérer un élément dans un tableau
 * dans lequel les i premiers éléments sont triés par ordre croissant.
 * @param tab un tableau d'entiers
 * @param x un entier à insérer dans tab
 * @param i un entier tel que tab[0], ..., tab[i-1] soit trié
 * @return la position <= i à laquelle insérer x
 **/
int position(vector<int> tab, int x, int i) {

}
```

(5) La complexité de votre fonction `position` est-elle optimale ? Si non, décrivez en quelques mots un algorithme permettant d'améliorer la vitesse d'exécution.

(6) Implanter le tri par insertion tel que décrit dans l'entête de l'exercice. Écrire des tests.

```
vector<int> triInsertion(vector<int> tab) {



}

void triInsertionTest() {



}
```

(7) Quelle est la complexité globale du tri par insertion ?