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 17 décembre 2018 (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 (Cours : tableaux à deux dimensions)**.**
Quelles sont les étapes pour construire un tableau à deux dimensions en C++ ?

Illustrer votre réponse avec la construction d'un tableau à deux dimensions avec 7 lignes et 6 colonnes, initialisé de sorte que la valeur à la ligne i et colonne j soit $i + j$.

Exercice 2 (Exponentiation rapide)**.**
Notions : fonctions, boucles, récursivité, complexité

(1) Implanter la fonction `puissanceNaive` dont la documentation est donnée.

```
/** Fonction puissanceNaive
 *  @param x un nombre entier
 *  @param n un nombre entier positif
 *  @return n-ième puissance x^n de x
 **/
int puissanceNaive( int x, int n ) {




}
```

(2) Écrire trois tests pour cette fonction :

```
void puissanceNaiveTest() {




}
```

(3) ♣ Implanter une version récursive de cette fonction :

```
int puissanceNaiveRecursive( int x, int n ) {



}
```

(4) Quelles sont les complexités respectives de ces deux fonctions ? Bien préciser le modèle de calcul : taille du problème, opérations élémentaires ; pour ces dernières, on pourra par exemple prendre les multiplications.

(5) En fait on peut faire mieux ; vous vous rappelez de l'algorithme vu en amphi permettant de calculer x^4 en deux multiplications ? On va généraliser cette idée pour tout n.

Exécutez pas à pas la fonction suivante pour $x = 2$ et pour toutes les valeurs de n suivantes : $0, 1, 2, 4, 5, 8$:

```
int puissanceRapide( int x, int n ) {
    int y = 1; int c;

    while ( n > 0 ){
        c++;
        if ( n % 2 == 1 ) {
            y = y * x;
            n = n - 1;
        } else {
            x = x * x;
            n = n / 2;
        }
    }

    return y;
}
```

Noter le résultat et le nombre $c(n)$ de mutiplications effectuées dans le tableau qui suit :

n	0	1	2	4	5	8
2^n						
$c(n)$						

Proposer une conjecture pour $c(n)$ lorsque n est une puissance de 2.

(6) Compter à la main le nombre de multiplications devient vite fastidieux. Implanter une variante de la fonction `puissanceRapide` qui renvoie $c(n)$ au lieu de x^n.

```
int puissanceRapideComplexite( int x, int n ) {




}
```

(7) ♣ Implanter une version récursive de `puissanceRapide`.

```
int puissanceRapideRecursive( int x, int n ) {




}
```

Exercice 3 (Le parking : tableaux, fonctions, boucles)**.**
Dans cet exercice, on modélise par un parking dont les places sont numérotées de 0 à n-1 par un tableau de type `vector<int>`. Les places vides contiennent 0 et les places occupées contiennent un nombre entier identifiant une voiture. Par exemple, le tableau suivant

```
{0,0,2,0,1,0,0,3}
```

contient 8 places dont 3 sont occupées par des voitures.

(1) Complétez la fonction suivante (dont les tests sont donnés) qui prend en paramètre un tableau représentant un parking et renvoie l'indice de la première place libre. Si le parking est plein, la fonction renvoie -1.

```
/** Renvoie la premiere place vide
 * @param parking un tableau d'entier
 * @return l'indice du premier emplacement vide
 **/
int premierVide(vector<int> parking) {




}

void premierVideTest() {
    ASSERT(premierVide({0,0,2,0,1,0,0,3}) == 0);
    ASSERT(premierVide({4,5,2,0,1,0,0,3}) == 3);
    ASSERT(premierVide({4,5,2,9,1,8,7,0}) == 7);
    ASSERT(premierVide({4,5,2,9,1,8,7,3}) == -1);
}
```

(2) On suppose à présent que chaque voiture à une **place préférée**. La voiture avance jusqu'à sa place préférée et se gare ensuite dans la première place libre qu'elle trouve. La voie est en sens unique, la voiture ne peut pas faire demi-tour. Si toutes les places après la sienne sont prises, alors elle ne se gare pas dans le parking.

Prenons par exemple la situation suivante :

```
{0,0,3,0,1,0,0,2}
```

On suppose que la voiture 4 veut se garer à la place 3. Elle est libre, donc elle se gare. Le parking est maintenant :

```
{0,0,3,4,1,0,0,2}
```

La place préférée de la voiture 5 est la numéro 2 qui est prise par la voiture 3, la première place libre après la numéro 2 dans le parking est la 5 et on obtient donc :

```
{0,0,3,4,1,5,0,2}.
```

Implantez la fonction suivante dont la documentation et les tests sont donnés. On n'utilisera PAS la fonction précédente.

```
/** Gare une voiture en fonction de sa place préférée et du parking
 * @param parking un tableau d'entier
 * @param voiture un entier représentant une voiture
 * @param place, un entier représentant la place préférée de la voiture
 *        dans le parking
 * @return le parking modifié si la voiture peut se garer
 *        et le parking non modifié sinon
 **/
vector<int> gareVoiture(vector<int> parking, int voiture, int place) {









}

void gareVoitureTest() {
    ASSERT(gareVoiture({0,0,3,0,1,0,0,2}, 4, 3) == vector<int>({0,0,3,4,1,0,0,2}));
    ASSERT(gareVoiture({0,0,3,4,1,0,0,2}, 5, 2) == vector<int>({0,0,3,4,1,5,0,2}));
    ASSERT(gareVoiture({0,0,3,4,1,5,0,2}, 6, 4) == vector<int>({0,0,3,4,1,5,6,2}));
    ASSERT(gareVoiture({0,0,3,4,1,5,6,2}, 7, 3) == vector<int>({0,0,3,4,1,5,6,2}));
}
```

(3) Dans la fonction suivante, le vecteur `voitures` contient les places préférées des voitures (la place préférée de la voiture i est à l'indice $i-1$). Observez le code et complétez la documentation et les tests : ajoutez au moins deux tests dans lesquels la fonction renvoie respectivement `true` et `false`.

```
bool mystere(vector<int> voitures) {
    vector<int> parking = vector<int>(voitures.size(), 0);
    for(int i=0; i< voitures.size(); i++) {
        parking = gareVoiture(parking, i+1, voitures[i]);
    }
    return premierVide(parking) == -1;
}

void mystereTest() {
    ASSERT(mystere({3,2,1}));
    ASSERT(not mystere({3,3,3}));
```

Exercice 4 (tableaux 2D)**.**
On souhaite gérer les notes des étudiants aux examens. Ces notes concernent plusieurs matières. Pour modéliser ce problème, on va utiliser un tableau à deux dimensions. Chaque ligne du tableau représente les notes de tous les étudiants dans une matière (une ligne par matière). Chaque colonne représente les notes d'un étudiant dans toutes les matières (une colonne par étudiant). Chaque case contient la note d'un étudiant E pour une matière M. Voici un exemple de tableau de notes, où 120, 121.....,129 représentent les numeros des étudiants :

Matière/Numéro	120	121	122	123	124	125	126	127	128	129
Informatique	20	18	18	11	12	13	14	15	17	16
Math	18	15	15	11	12	11	14	15	12	13
Physique	19	12	10	11	13	11	14	15	18	16
Chimie	18	12	12	11	13	11	13	15	12	13
Sciences naturelles	18	16	12	11	13	11	14	15	18	16

On suppose que l'on dispose de deux tableaux 1D regroupant l'un les numéros des étudiants et l'autre les noms des matières, comme ceci :

```
vector<int> numerosEtudiants = {120, 121, 122, 123, 124, 125, 126, 127, 128, 129};
vector<string> matieres = {"Informatique", "Math", "Physique", "Chimie",
                           "Sciences naturelles"};
```

Pour alléger les notations, on définit un raccourci Notes pour les tableaux bidimensionnels des notes :

```
typedef vector<vector<int>> Notes;
```

Dans les tests, on pourra supposer que l'on a à disposition le tableau des notes suivant qui correspond l'exemple montré ci-dessus :

```
Notes notes = {
    {20, 18, 18, 11, 12, 13, 14, 15, 17, 16},
    {18, 15, 15, 11, 12, 11, 14, 15, 12, 13},
    {19, 12, 10, 11, 13, 11, 14, 15, 18, 16},
    {18, 12, 12, 11, 13, 11, 13, 15, 12, 13},
    {18, 16, 12, 11, 13, 11, 14, 15, 18, 16}
};
```

(1) On suppose qu'il y a N étudiants et M matières. Compléter l'implantation de la fonction suivante :

```
/** Construit et renvoie un tableau de notes
 * en lisant les données à partir du clavier
 * @param N : un entier représentant le nombre d'étudiants
 * @param M : un entier représentant le nombre de matières
 * @return un tableau d'entiers a deux dimensions M x N
 **/
Notes tableauNotes(int N, int M) {









}
```

(2) On suppose que les notes des matières sont stockées dans un fichier dont l'entête contient le nombre de matières et d'étudiants ; pour l'exemple que nous avons vu plus haut, le fichier contiendrait :

```
5 10
20 18 18 11 12 13 14 15 17 16
18 15 15 11 12 11 14 15 12 13
19 12 10 11 13 11 14 15 18 16
18 12 12 11 13 11 13 15 12 13
18 16 12 11 13 11 14 15 18 16
```

Implanter la fonction suivante :

```
/** Construit et renvoie un tableau de notes
 * en lisant les données à partir d'un fichier
 * @param filename : string nom de fichier qui contient
 * les notes des matiéres
 * @format fichier : la première ligne contient 2 entiers : le
 * nombre de matières (M) et le nombre de étudiants (N).
 * La suite du fichier contient le tableau des notes
 * @return un tableau d'entiers a deux dimensions M x N
 **/
Notes tableauNotesFichier(string filename) {




}
```

(3) Implanter, avec sa documentation, la fonction moyenneMatière qui prend en argument un tableau de notes et un numéro de matière (entier), et qui renvoie la moyenne pour cette matière.

```
float moyenneMatiere(Notes notes, int m) {

}
```

(4) En utilisant la fonction moyenneMatière de la question précédente, implanter une fonction qui prend en argument un tableau de notes et qui renvoie un tableau 1D, dont chaque case contient la moyenne pour chaque matière.

```
/** Calcule la moyenne de toutes les matières
 * @param notes un tableau d'entiers contenant les notes
 * @return un tableau d'entiers à N élèment (N étant le nombre d'étudiants)
 **/
vector<float> moyenneToutesMatieres(Notes notes) {

}
```

(5) On suppose que l'on dispose d'une fonction maxToutesMatières qui prend en paramètre un tableau des notes et renvoie un tableau 1D, dont chaque case contient la note maximale pour chaque matière. En s'appuyant sur le tableau tabDenotes pour l'exemple donné, écrire un test pour la fonction maxToutesMatières (ne pas écrire la fonction, seulement le test).

```
void maxToutesMatieresTest() {                    // Correction
    ASSERT(maxToutesMatieres(notes) == 20 );      // FinCorrection
}
```

(6) On suppose que l'on dispose d'un tableau 1D représentant les coefficients des matières, comme ceci :

```
vector<int> coefficients = {2, 4, 3, 1, 2} ;
```

Tout en prenant en considération ces coefficients, implanter, avec sa documentation, la fonction moyenneEtudiant qui prend en arguments un tableau des notes, un tableau des coefficients et un numéro d'étudiant (entier) et qui renvoie la moyenne générale de cet étudiant.

```
float moyenneEtudiant(Notes notes, vector<int> coeffs, int numero ) {









}
```

(7) Implanter la fonction moyennesGenerales qui prend en arguments un tableau des notes, un tableau des coefficients et le nombre d'étudiants N et qui renvoie un tableau 1D, dont chaque case contient la moyenne générale pour chaque étudiant.

```
vector<float> moyennesGenerales(Notes notes, vector<int> coeffs,
                                int nombreEtudiants) {









}
```

(8) En utilisant certaines des fonctions des questions précédentes, écrire un fragment de programme qui affiche la moyenne générale pour chaque étudiant de la façon suivante :

```
      Numero étudiant: 123, Moyenne générale: 18
```

```
```