Nom, Prénom : **Numéro d'étudiant :** ␣␣␣␣␣␣␣␣

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

Université Paris Sud, Licence MPI, Info 111 — Rattrapage du 24 juin 2019 (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. Le barème est donné à titre indicatif.

Exercice 1 (Cours (2 points))**.**
Rappeler la syntaxe (1 point) et la sémantique (1 point) de la boucle `for` en C++.

Correction : Syntaxe :

```
for ( initialisation ; condition ; incrementation ) {
    bloc d'instructions
}
```

Sémantique :

(1) Exécution de l'instruction d'initialisation
(2) Évaluation de la condition
(3) Si sa valeur est `true` :
  (a) Exécution du bloc d'instruction
  (b) Exécution de l'instruction d'incrémentation
  (c) On recommence en 2

Exercice 2 (Fonctions (3.5 points))**.**

(1) (1 point) Écrire une fonction `distance` qui prend en paramètres deux nombres réèls et renvoie leur distance, c'est-à-dire la valeur de la différence entre le plus grand des deux et le plus petit des deux. Par exemple, la distance entre 1.1 et 2 vaut .9 et celle entre 3.3 et 2 vaut 1.3.

Correction :

```
float distance(float a, float b) {
    if ( a > b ) {
        return a - b;
    } else {
        return b - a;
    }
}
```

(2) (1.5 points) Utilisation : écrire les lignes de code qui permettent de
- déclarer deux variables `x` et `y` ;
- les initialiser respectivement avec les valeurs 3.5 et -1.2 ;
- calculer la distance entre les deux avec **un appel à la fonction `distance`** ;
- affecter le résultat à une nouvelle variable `resultat`.

Correction :

```
    float x, y;
    x = 3.5;
    y = -1.2;
    float resultat = distance(x, y);
```

(3) (.75 point) Traduire les deux exemples du (1) sous forme de tests avec `ASSERT`.

Correction :

```
    float precision = 0.0001;
    ASSERT ( abs( distance(1.1, 2) -  .9 ) < precision );
    ASSERT ( abs( distance(3.3, 2) - 1.3 ) < precision );
    ASSERT ( abs( resultat         - 4.7 ) < precision );
```

(4) (.25 point) ♣ Si nécessaire, ajuster les tests pour tenir compte des possibles erreurs d'arrondis.

Exercice 3 (Triangle de Pascal (8.5 points))**.**

(1) (1 point) On rappelle que $n! = 1 \times 2 \times \cdots \times n$. Implanter la fonction `factorielle` dont la documentation et les tests sont donnés ci-dessous.

```
/** La fonction factorielle
 *  @param n un nombre entier positif
 *  @return n!
 **/
int factorielle(int n) {
    int resultat = 1;
    for ( int k = 1; k <= n; k++ ) {
        resultat = resultat * k;
    }
    return resultat;
}
```

```
void factorielleTest() {
    ASSERT( factorielle(0) == 1  );
    ASSERT( factorielle(1) == 1  );
    ASSERT( factorielle(2) == 2  );
    ASSERT( factorielle(3) == 6  );
    ASSERT( factorielle(4) == 24 );
}
```

(2) (1.5 points) On rappelle que le coefficient binomial est défini par $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Écrire la documentation d'une fonction `binomial` calculant le coefficient binomial $\binom{n}{k}$, puis implanter cette fonction.

Correction :

```
/** La fonction binomial
 *  @param n un nombre entier positif
 *  @param k un nombre entier positif
 *  @return le coefficient binomial n,k
 **/
int binomial(int n, int k) {
    return factorielle(n) / ( factorielle(k) * factorielle(n-k) );
}                                          //FinCorrection
```

(3) (.5 point) Écrire quelques tests pour la fonction `binomial`.

Correction :

```
    ASSERT( binomial(5, 2) == 10 );
    ASSERT( binomial(1, 1) == 1  );
    ASSERT( binomial(0, 0) == 1  );
```

(4) (1.5 points) Le triangle de Pascal est une présentation des coefficients binomiaux dans un triangle. Par exemple la ligne d'indice 2 d'un triangle de Pascal correspond aux trois coefficients binomiaux $\binom{2}{0}$, $\binom{2}{1}$ et $\binom{2}{2}$ qui valent, respectivement, $1, 2$ et 1. Ainsi, la ligne d'indice n correspond aux $n+1$ coefficients binomiaux

$$\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}.$$

Écrire un programme qui affiche les dix premières lignes du triangle de Pascal comme ci-dessous :

```
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
1       6       15      20      15      6       1
1       7       21      35      35      21      7       1
1       8       28      56      70      56      28      8       1
1       9       36      84      126     126     84      36      9       1
```

Pour bien aligner les nombres, on pourra les séparer par des tabulations (caractère "`\t`").

Correction :

```
    for (int n = 0; n < nmax; n++) {
        for (int k = 0; k <= n; k++) {
            cout << binomial(n, k) << "\t";
        }
        cout << endl;
    }
```

(5) (2 points) Quelles sont les complexités
— de la fonction `factorielle`,
— de la fonction `binomial`,
— du programme.
Bien préciser le *modèle de calcul* et *justifier* les réponses.

Correction : Modèle de calcul : taille du problème : n ; opérations élémentaires : multiplications et divisions

Complexité de la fonction factorielle : $O(n)$

Complexité de la fonction binomial : $O(n)$ (3 appels à factorielle, pour des valeurs inférieures à n).

Complexité du programme : $O(n^3)$: n lignes de au plus n coefficient dont le calcul est de complexité n.

(6) (1 point) Quelle(s) ligne(s) faut-il changer dans votre programme, et comment, pour que le triangle de Pascal soit écrit dans un fichier `pascal.txt` ?

Correction :

```
    ofstream fichier("pascal.txt");
    for (int n=0; n < nmax; n++) {
        for (int k=0; k <= n; k++) {
            fichier << binomial(n, k) << "\t";
        }
        fichier << endl;
    }
    fichier.close();
```

(7) (1 point) ♣ On rappelle que les coefficients binomiaux satisfont la formule récursive suivante :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Cela se traduit, sur le triangle de Pascal, par le fait que chaque entrée est la somme de l'entrée située au-dessus à gauche et de celle située au-dessus dans la même colonne.

En utilisant cette propriété, calculer et afficher les dix premières lignes du triangle de Pascal. On pourra, au choix, modifier la fonction `binomial` ou le programme.

Correction :

```
int binomial_recursif(int n, int k) {
    if ( k > n ) {
        return 0;
    }
    if ( n <= 1 or k == 0 ) {
        return 1;
    } else {
        return binomial_recursif(n-1,k) + binomial_recursif(n-1,k-1);
    }
}                                                    //FinCorrection
```

Exercice 4 (Tableaux à deux dimension (5.5 points))**.**
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 point) 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.

```
/** Nombre d'exemplaires par modèle
 * @param C: le tableau des ventes
 * @param m: le numéro d'un modèle
 * @return le nombre d'exemplaires vendus du modèle m
 **/
int nbexemplairesParModele(vector<vector<int>> C, int m) {
    int nbex = 0;
    for (int i=0; i<C.size(); i++)
        nbex = nbex + C[i][m];
    return nbex;
}
```

(2) (.5 point) 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() {
    ASSERT( nbexemplairesParVendeur(ventes, 0) == 5 );
    ASSERT( nbexemplairesParVendeur(ventes, 1) == 6 );
    ASSERT( nbexemplairesParVendeur(ventes, 2) == 4 );
}
```

(3) (1.5 points) 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) {
    vector<vector<int>> ventes(V);
    for (int i = 0; i < ventes.size(); i++ ) {
        ventes[i] = vector<int> (M);
        for ( int j = 0; j < ventes[0].size(); j++ ) {
            cout << "Nombre de ventes du vendeur "
            << vendeurs[i] << " pour le modèle "
            << modeles[j] << " : ";
            cin >> ventes[i][j];
        }
    }
    return ventes;
}
```

(4) (1 point) 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) {
    ifstream f(filename);
    int V, M;
    f >> V >> M;
    vector<vector<int>> ventes(V);
    for ( int i = 0; i < ventes.size(); i++ ) {
        ventes[i] = vector<int> (M);
        for ( int j = 0; j < ventes[0].size(); j++ )
          f >> ventes[i][j];
    }
    f.close();
    return ventes;
}
```

(5) (1.5 points) 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.

```
    vector<vector<int>> ventes;
    ventes = tableauVentesFichier("vector2D-concession.txt");
    int vente;
    int maxVente = 0;
    int modele = 0;
    for ( int i = 0; i < ventes[0].size(); i++ ) {
        vente = nbexemplairesParModele(ventes, i);
        if ( vente >= maxVente ) {
            maxVente = vente;
            modele = i;
        }
    }
    cout << "meilleur modele : " << modeles[modele] << endl;
```