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 16 décembre 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.

Exercice 1 (Cours (2 points))**.**

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

```
type nom(type1 parametre1, type2 parametre2, ...);
```

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

```
string mystere(double d, int i);
```

— Donner un exemple d'appel à cette fonction.
Correction :

```
cout << mystere(3.14, 1) << endl;
```

Exercice 2 (Nombres triangulaires : fonctions, boucles, récursivité, complexité (5 points))**.**
On appelle « nombres triangulaires » les nombres entiers strictement positifs qui sont égaux à la somme de plusieurs entiers consécutifs à partir de 1. Par exemple :

21 est un nombre triangulaire car $21 = 1 + 2 + 3 + 4 + 5 + 6$
7 n'est pas un nombre triangulaire car $1 + 2 + 3 < 7 < 1 + 2 + 3 + 4$
Le premier nombre triangulaire est 1.
Le deuxième nombre triangulaire est 3 car 1+2=3.
Le troisième nombre triangulaire est 6 car 1+2+3=6. Etc

(1) On considère la fonction `nombreTriangulaire` dont la documentation est :

```
/** Fonction nombreTriangulaire
 *  @param un entier n
 *  @return le n-ième nombre triangulaire
 *          c'est à dire la somme des entiers entre 1 et n
 **/
```

Écrire trois tests pour cette fonction :

```
void nombreTriangulaireTest(){
    ASSERT( nombreTriangulaire(1) == 1 );
    ASSERT( nombreTriangulaire(2) == 3 );
    ASSERT( nombreTriangulaire(3) == 6 );
}
```

(2) Écrire un fragment de programme qui affiche à l'écran les 10 premiers nombres triangulaires.

```
    for ( int n = 1 ; n <= 10 ; n++ ) {
        cout << nombreTriangulaire(n) << " ";
    }
    cout << endl ;
```

(3) Implanter cette fonction

```
int nombreTriangulaire(int n) {
    int somme = 0;
    for (int k = 1 ; k <= n ; k++) {
        somme = somme + k;
    }
    return somme;
}
```

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

```
int nombreTriangulaireRecursif(int n) {
    if ( n == 1 ){
        return 1;
    } else {
        return n + nombreTriangulaireRecursif(n-1);
    }
}
```

(5) On cherche maintenant à déterminer si un nombre est triangulaire. Implanter la fonction dont la documentation est donnée ci dessous. **Cette fonction devra utiliser (appeler) la fonction `nombreTriangulaire` écrite précédemment**.

```
/** Fonction estTriangulaire1
 *  @param p un nombre entier
 *  @return true si p est un nombre triangulaire, false sinon.
 **/
bool estTriangulaire1(int p){
    for ( int n = 1; n <= p; n++ ) {
        if ( nombreTriangulaire(n) == p ) {
            return true;
        }
    }
    return false;
}
```

(6) On souhaite comparer cette implantation avec l'implantation alternative suivante :

```
bool estTriangulaire2(int p) {
    int nombreTriangulaire = 1;
    for ( int k = 2; nombreTriangulaire < p; k++ ) {
        nombreTriangulaire = nombreTriangulaire + k;
    }
    return nombreTriangulaire == p ;
}
```

Pour cela on va étudier leur complexité dans le modèle de calcul suivant :
— La taille du problème est donnée par p
— Les opérations élémentaires sont les additions (hors incrémentation)

Soit $c_1(p)$ la complexité de la fonction `estTriangulaire1`. Soit de même $c_2(p)$ la complexité de `estTriangulaire2(p)`. Le tableau suivant donne les premières valeurs de $c_1(p)$ (pour notre implantation ; cela peut changer un peu avec la vôtre) :

p	1	2	3	4	5	6	7	8
$c_1(p)$	2	6	6	20	30	12	56	72
$c_2(p)$								

Correction :

p	1	2	3	4	5	6	7	8
$c_1(p)$	2	6	6	20	30	12	56	72
$c_2(p)$	0	1	1	2	2	2	3	3

Compléter le tableau pour $c_2(p)$ en exécutant pas à pas `estTriangulaire2`.

(7) Laquelle des deux implantations `estTriangulaire1` et `estTriangulaire2` est la plus performante pour de petites valeurs de p ?

Correction : `estTriangulaire2` puisque $c_2(p) << c_1(p)$.

(8) ♣ Est-ce que cela reste vrai pour de grandes valeurs de p ? Justifier empiriquement.

Correction : Oui car dans `estTriangulaire1` le n-ième nombre triangulaire est obtenu en recalculant la somme $1+\cdots+n$ (donc $n-1$ opérations) alors que dans `estTriangulaire2` il est obtenu en une opération à partir du $n-1$-ième nombre triangulaire.

Exercice 3 (Gestion hôtelière : tableaux, fonctions, boucles (6 points))**.**
On réalise un programme de gestion des réservations d'un hôtel. Les chambres sont numérotées de 0 à $n-1$. On représente les réservations par un tableau de type `vector<int>`. Il contient 0 pour les chambres libres et un nombre entier identifiant un client pour les chambres réservées. Voici par exemple un tableau de réservations pour un hôtel de dix chambres dont 4 sont réservées :

```
vector<int> reservations = {0,2,3,0,1,0,0,4,0,0};
```

De plus, chaque chambre est caractérisée par une capacité qui représente le nombre de lits. Ces capacités sont enregistrées dans un deuxième tableau de type `vector<int>`. Par exemple, avec le tableau de capacités suivant, les chambres numéro 3 et 9 contiennent toutes les deux 5 lits :

```
vector<int> capacites    = {2,3,3,5,4,2,3,4,3,5};
```

(1) Implanter une fonction qui annule toutes les réservations d'un client

```
/** Annule la reservation d'un client
 * @param un tableau: les réservations des chambres
 * @param un entier: le client
 * @return le tableau des réservations mis à jour
 **/
vector<int> annulation(vector<int> reservations,
                       int client) {
    for ( int i = 0; i < reservations.size(); i++ ) {
        if ( reservations[i] == client ) {
            reservations[i] = 0;
        }
    }
    return reservations;
```

(2) Proposer une documentation pour la fonction suivante

```
/** Renvoie la première chambre disponible satisfaisant le besoin du client
 * @param un tableau: les réservations des chambres
 * @param un tableau: les capacités des chambres
 * @param un entier: le nombre de lits à réserver
 * @return le numéro de la chambre
 **/
int mystere(vector<int> H , vector<int> C, int N) {
    for ( int i = 0; i< H.size(); i++ ){
        if( H[i] == 0 and C[i] >= N )
            return i;
    }
    return -1;
}
```

(3) Donner la complexité au pire de cette fonction (préciser le modèle de calcul) :

Correction : $2n$ opérations, où n est la taille de l'hôtel et où l'on compte les accès aux tableaux.

(4) Pour réserver une chambre, le client doit préciser le nombre de lits dont il a besoin. Implantez la fonction `reservation` qui réserve la première chambre disponible avec un nombre de lits suffisants. Si aucune chambre ne convient, aucune réservation n'est faite.

```
/** Réserve une chambre pour un client
 * @param un tableau: les chambres réservées
```

```
 * @param un tableau: les capacités des chambres
 * @param un entier: le client
 * @param un entier: le nombre de lits à réserver
 * @return le tableau des réservations mis à jour
 **/

vector<int> reservation(vector<int> reservations,
                        vector<int> capacites,
                        int client, int nbLits) {
    int i = mystere(reservations, capacites, nbLits);
    if ( i >= 0 ) {
        reservations[i] = client;
    }
```

(5) Afin d'optimiser l'affectation des chambres, l'hôtel souhaite attribuer une chambre **de capacité minimale** mais suffisante pour les besoins du client. La fonction `chambreMin` renverra une telle chambre **de numéro minimal**, ou -1 si aucun chambre ne convient. Écrire les tests pour des besoins de 3, 5 et 6 lits respectivement en utilisant les tableaux `reservations` et `capacites` donnés plus haut :

```
void chambreMinTest() {

    ASSERT( chambreMin(reservations, capacites, 6) == -1 );
    ASSERT( chambreMin(reservations, capacites, 5) == 3 );
```

(6) ♣ Implanter la fonction `chambreMin`.

```
 **/

int chambreMin(vector<int> reservations,
               vector<int> capacites, int nbLits){
    int chambre = -1 ;
    int minCapacite = -1;
    for ( int i = 0; i < reservations.size(); i++ ) {
        if (     reservations[i] == 0
             and capacites[i] >= nbLits
             and (chambre == -1 or capacites[i] < minCapacite) ) {
            chambre = i;
            minCapacite = capacites[i];
        }
    }
    return chambre;
```

Exercice 4 (Sudoku (7 points))**.**
Le but du Sudoku est de compléter une grille de 81 cases (9x9) avec des chiffres de 1 à 9. Cette grille est divisée en 9 blocs de 3x3. Le joueur doit compléter la grille avec la contrainte de ne jamais avoir deux fois le même chiffre sur une ligne, une colonne ou un bloc de 3x3.
Au départ, la grille de Sudoku est partiellement remplie. La figure 1 présente un exemple de Sudoku à remplir.

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

FIGURE 1. Un exemple de Sudoku partiellement rempli en début de jeu

Dans ce problème, on s'intéresse à l'écriture d'un programme permettant de compléter une grille de Sudoku en respecant les règles de jeu. On représente une grille de Sudoku par un tableau 2D d'entiers, les cases vides étant encodées par l'entier 0.

(1) **Complétion d'une grille** : Implanter une fonction `estRemplie` qui prend en paramètre d'entrée une grille de Sudoku quelconque et renvoie `true` si une grille est complètement remplie et `false` sinon, sans vérifier sa validité.

```
bool estRemplie(Grille grille) {
    for (int ligne = 0; ligne < 9; ligne++) {
        for (int colonne = 0; colonne < 9; colonne++) {
            if (grille[ligne][colonne] == 0) {
                return false;
            }
        }
    }
    return true;
}
```

(2) **Validité d'une grille** :

(a) Implanter une fonction `elementsDeBloc` qui prend en paramètre d'entrée une grille de sudoku, ainsi que la ligne et la colonne d'un bloc de 3x3 (0, 1 ou 2) et qui renvoie les entiers **non-nuls** contenus dans ce bloc. Si l'on prend l'exemple de la figure 1, l'appel de `elementsDeBloc(grille, 2, 0)` doit renvoyer le tableau $\{7, 9, 3\}$.

```
/** Renvoie les valeurs non-nulles d'un bloc d'indice i et j
 * @param grille une grille de sudoku
 * @param indice de la ligne du bloc (entre 0 et 2)
 * @param indice de la colonne du bloc (entre 0 et 2)
 * @return un tableau d'entier
 **/
vector<int> elementsDeBloc(Grille G, int ligne, int colonne) {
    vector<int> res;
    int element;
```

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            element = G[ligne * 3 + i][colonne * 3 + j];
            if (element != 0) {
                res.push_back(element);
            }
        }
    }
    return res;
}
```

(b) On considère qu'il existe une fonction `contient` qui prend en paramètre d'entrée un entier et un tableau d'entiers et renvoie `true` si l'entier est présent dans le tableau, `false` sinon. Vous avez déjà implanté une fonction très similaire pendant le semestre, lors du TP sur le jeu de Yams. Implanter la fonction `sansDoublon` ci-dessous, (en utilisant la fonction `contient` si vous le souhaitez) :

```
/** Teste si un tableau d'entiers n'a pas de doublons entre 1 et 9,
 * NB : on ne considère pas les doublons de 0
 * @param un tableau d'entier
 * @return blooléen: True si contient un doublon, False sinon
 **/
bool sansDoublon(vector<int> t) {
    vector<int> vus = {};
    for (int i = 0; i < t.size(); i++) {
        if (contient(t[i], vus)) {
            return false;
        }
        if (t[i] != 0) {
            vus.push_back(t[i]);
        }
    }
    return true;
}
```

(c) Écrire au moins trois tests pertinents de la fonction `sansDoublon` en utilisant la fonction `ASSERT`.

```
void sansDoublonTest() {
    ASSERT(sansDoublon({0, 1, 2, 3, 0}));
    ASSERT(not sansDoublon({0, 1, 0, 2, 2, 0}));
    ASSERT(sansDoublon({0, 0, 0}));
    ASSERT(sansDoublon({}));
}
```

(d) De manière analogue à la fonction `elementsDeBloc`, on suppose que l'on a les fonctions `elementsDeLigne` et `elementsDeColonne` qui extraient les entiers non-nuls contenus dans une ligne (resp. une colonne) d'indice donné. En réutilisant les fonctions des questions (a) et (b), écrire une fonction `estValide` qui prend en paramètre d'entrée une grille quelconque (remplie ou partiellement remplie) et renvoie `true` si la grille est valide c'est à dire qu'elle ne présente pas de doublons sur ses lignes, ses colonnes et ses blocs.

```
bool estValide(Grille G) {
    for (int i = 0; i < 9; i++) {
        if (sansDoublon(elementsDeLigne(G, i)) == false or
```

```
            sansDoublon(elementsDeColonne(G, i)) == false) {
            return false;
        }
    }

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (sansDoublon(elementsDeBloc(G, i, j)) == false) {
                return false;
            }
        }
    }
    return true;
}
```

(3) **Vers la recherche automatique de solution** :

Nous nous intéressons à présent au processus de remplissage des cases en vue d'automatiser la recherche de solution.

(a) Implanter une fonction `valeursPossibles` qui prend en paramètre d'entrée une grille et les indices d'une case de la grille et renvoie les valeurs compatibles pour cette case c'est à dire celles ne créant pas de doublons avec la ligne, la colonne et le bloc considéré. **Indice :** pensez à réutiliser les fonctions `contient`, `elementsDeLigne`, `elementsDeColonne` et `elementsDeBloc`.

```
vector<int> valeursPossibles(Grille G, int i, int j) {
    vector<int> res;
    for (int k = 1; k <= 9; k++) {
        if (not (contient(k, elementsDeLigne(G, i)) or
                 contient(k, elementsDeColonne(G, j)) or
                 contient(k, elementsDeBloc(G, i / 3, j / 3)) ) ) {
            res.push_back(k);
        }

    }
    return res;
}
```

(b) On considère la fonction `resolutionNaive` qui, à partir d'une grille, remplit tour à tour les cases vides n'ayant qu'une seule valeur possible et renvoie une grille vide si la complétion n'est pas possible. Autrement dit, la fonction `resolutionNaive` renvoie un Sudoku complété au maximum permis par l'algorithme ou une grille vide si le Sudoku n'est pas solvable. La fonction `resolutionNaiveBoguee` suivante implante cet algorithme, mais contient trois bogues.

Pour chaque bogue, **identifier le numéro de la ligne fautive, décrire le problème et proposer une correction**.

```
1  Grille resolutionNaiveBoguee(Grille grille) {
2      bool changement = false;
3      while ( changement ) {
4          changement = false;
5          for (int ligne = 0; ligne < 9; ligne++) {
6              for (int colonne = 0; colonne < 9; colonne++) {
7                  if ( grille[ligne][colonne] = 0 ) {
8                      vector<int> vals =
9                          valeursPossibles(grille, ligne, colonne);
10                     if ( vals.size() == 0 ) {
11                         return grilleVide();
12                     }
13                     if ( vals.size() == 1 ) {
14                         grille[ligne][colonne] == vals[0];
15                         changement = true;
16                     }
17                 }
18             }
19         }
20     }
21     return grille;
22 }
```

(i)

Correction : Ligne 2 : on n'entre jamais dans la boucle `bool changement = true;`

(ii)

Correction : Ligne 7 : affectation au lieu d'un test d'égalité `if (grille[ligne][colonn`

(iii)

Correction : Ligne 14 : test d'égalité au lieu d'affectation `grille[ligne][colonne] =`

(c) ♣ Déterminer une majoration de la complexité au pire de la fonction `estValide` puis de `resolutionNaive`. Bien préciser le modèle de calcul choisi.

Correction : Prenons $n = 9$ la taille de la grille, et comptons le nombre d'accès à la grille. `elementsDeLigne` et consorts, ainsi que contient sont de complexité au pire n. Donc `valeursPossibles` est de complexité au pire $v(n) = n(3n + 3n) = 6n^2 = 486$.

À chaque tour de la boucle while, on lance `valeursPossible` au plus sur toutes les cases en faisant éventuellement une affectation ; cela fait $n^2(v(n) + 1)$. À chaque tour de la boucle while sauf la dernière, on remplit au moins une case, donc cela fait au maximum n^2 tours de boucles. Au total, on obtient une complexité au pire bornée par $n^4(v(n) + 1)$, soit 3195207 opérations.

(d) ♣ Proposer un algorithme permettant de déterminer toutes les solutions d'un problème de sudoku. On pourra se contenter d'en donner les grandes lignes.