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

Université Paris Sud, Licence MPI, Info 111 — Partiel du 22 octobre 2018 (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 mais font bien partie du barème sur 20.
Barème indicatif : en gros un point par question.
Les réponses sont à donner, autant que possible, sur le sujet ; sinon, mettre un renvoi.

Exercice 1 (Cours)**.**
Rappeler la syntaxe et la sémantique de la boucle `while` en C++.

Correction : Syntaxe :

```
while ( condition ) {
    bloc d'instructions;
}
```

Sémantique : tant que la condition est vraie, on répète le bloc d'instructions :
- La condition est évaluée
- Si sa valeur est `true`, le bloc d'instructions est exécuté
- On recommence

Exercice 2 (Cours)**.**

- 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 entier et une chaîne de caractères et renvoie un booléen.

  Correction :

  ```
  bool mystere(string t,int a);
  ```

- Donner un exemple d'appel à cette fonction.

  Correction :

  ```
  if ( mystere("coucou", 3) )
      cout << "C'est mystérieux" << endl;
  ```

Exercice 3 (Fonctions)**.**

(1) Écrire une fonction `aireRectangle` qui prend en paramètres deux entiers représentant la longueur et la largeur d'un rectangle, et renvoie son aire.

Correction :

```
int aireRectangle(int a, int b) {

    return a*b;

}
```

(2) Utilisation : écrire les lignes de code qui permettent de déclarer et initialiser deux variables avec les valeurs de votre choix pour représenter les dimensions d'un rectangle, puis de stocker son aire dans une nouvelle variable. Il est obligatoire d'utiliser un **appel** à votre fonction `aireRectangle`.

Correction :

```
    int p = 2;
    int q = 8;
    int resultat = aireRectangle(p,q);
```

Exercice 4 (Booléens)**.**

(1) Écrire la fonction dont la documentation et les tests sont donnés ci-dessous :

```
/** Fonction multipleAvecContrainte
 *  @param a un entier
 *  @param b un entier
 *  @return true si a est un multiple de b et l'un des
        deux entiers est supérieur ou égal à 10, false sinon
 **/
```

```
    ASSERT( multipleAvecContrainte(50, 5) );
    ASSERT( multipleAvecContrainte(370, 10) );
    ASSERT( not multipleAvecContrainte(8, 2) );
    ASSERT( not multipleAvecContrainte(7, 49) );
```

Correction :

```
bool multipleAvecContrainteNaif(int a, int b) {
    if ((a % b == 0) and (a >= 10 or b >= 10)) {
        return true;
    } else {
        return false;
    }
}
```

(2) Si ce n'est pas déjà le cas, ré-écrire votre fonction pour qu'elle ne contienne pas de `if` (ni aucune autre structure de contrôle bien sûr !).

Correction :

```
bool multipleAvecContrainte(int a, int b) {
    return ((a % b == 0) and (a >= 10 or b >= 10));
}
```

Exercice 5 (Boucles)**.**

(1) Écrire les lignes de code permettant d'afficher tous les nombres pairs compris entre 0 et 100 inclus.

Correction :

```
    for (int i = 0; i <= 100; i=i+2){
        cout << i << endl;
    }
```

(2) Écrire une fonction nommée `afficheSommePairs` qui prend en paramètre un entier m et qui affiche la somme de tous les nombres pairs compris entre 0 et m inclus.

Correction :

```
void afficheSommePairs(int m) {
    int somme = 0;
    for ( int i = 0; i <= m; i=i+2) {
        somme = somme + i;
    }
    cout << somme << endl;
}
```

(3) Écrire une fonction nommée `affichePuissances` qui prend en paramètres un entier m et un entier p, et qui affiche les k-ièmes puissances de m pour k compris entre 0 et p inclus (c'est-à-dire affiche les valeurs de $m^0, m^1, m^2, ..., m^p$). Vous devez utiliser un appel à la fonction `puissance` dont la documentation est donnée ci-dessous :

```
/** La fonction puissance
 *  @param x un nombre entier
 *  @param n un nombre entier positif
 *  @return la n-ième puissance x^n de x
 **/
```

(La fonction puissance est considérée déjà définie, vous n'avez pas à écrire son code.)

Correction :

```
void affichePuissances(int m, int k) {
    for(int p = 0; p <= k; p++) {
        cout << puissance(m,p) << endl;
    }
}
```

Exercice 6 (Tableaux)**.**

(1) Écrire une fonction nommée `appartient` qui prend en paramètres un tableau d'entiers `t` et un entier `i` et renvoie `true` si `i` appartient à `t` et `false` sinon. La fonction devra passer les tests suivants :

```
    ASSERT(     appartient({2, 8, 6}, 2) );
    ASSERT(     appartient({2, 8, 6}, 8) );
    ASSERT( not appartient({2, 8, 6}, 3) );
```

Correction :

```
bool appartient(vector<int> t, int v) {
    for ( int i = 0; i < t.size(); i++ )
        if ( t[i] == v )
            return true;
    return false;
}
```

(2) ♣ Si ce n'est déjà fait, réécrivez la fonction précédente en utilisant une boucle *for each* : `for (int v:t){...}`.

Correction :

```
bool appartient_foreach(vector<int> t, int v) {
    for ( int w: t )
        if ( w == v )
            return true;
    return false;
}
```

(3) Écrire une fonction nommée `positionPair` qui prend en paramètre un tableau d'entiers `t`, et renvoie l'indice du premier nombre pair dans `t`, ou -1 si `t` ne contient pas de nombre pair. La fonction devra passer les tests suivants :

```
    ASSERT( positionPair({1, 5, 3}) == -1 );
    ASSERT( positionPair({2, 4, 3}) ==  0 );
    ASSERT( positionPair({1, 4, 3}) ==  1 );
    ASSERT( positionPair({1, 3,-2}) ==  2 );
```

Correction :

```
int positionPair(vector<int> t) {
    for ( int i = 0; i < t.size(); i++ )
        if ( t[i] % 2 == 0 )
            return i;
    return -1;
}
```

Exercice 7 (Jeu de Nim)**.**
Le jeu de Nim est un jeu à deux joueurs qui se joue avec des jetons. Avant de commencer la partie, on répartit les jetons en plusieurs tas (on peut en faire autant que l'on veut). Les joueurs jouent à tour de rôle. Chaque joueur doit à son tour choisir un tas qui contient des jetons et peut décider d'en retirer autant qu'il veut de ce tas (au moins un jeton à chaque fois). Le joueur qui retire le dernier jeton a gagné.

On représente un état du jeu de Nim par un tableau d'entiers, chaque case du tableau représentant le nombre de jetons dans un tas (on numérote donc les tas en commençant à zéro).

(1) Implanter la fonction `afficheNim` qui prend en paramètre un état du jeu de Nim et affiche le nombre de jetons dans chaque tas, selon l'exemple ci-dessous.

```
afficheNim({2, 0, 3, 4});
```

```
2
0
3
4
```

Correction :

```
void afficheNim(vector<int> etat) {
    for (int i = 0; i < etat.size(); i++) {
        cout << etat[i] << endl;
    }
}
```

(2) La partie est finie lorsque tous les tas sont vides. On veut écrire une fonction `fini` qui prend en paramètre un état du jeu de Nim et qui renvoie `true` si le jeu est fini et `false` sinon. Écrire quelques tests pour la fonction `fini`, puis l'implanter.

Correction :

```
ASSERT( fini({0, 0, 0}) );
ASSERT( not fini({0, 1, 3, 0}));
```

```
bool fini(vector<int> etat) {
    for (int i = 0; i < etat.size(); i++) {
        if (etat[i] > 0) {
            return false;
        }
    }
    return true;
}
```

(3) Un tour de jeu correspond à choisir un tas et à retirer au moins un jeton de ce tas. Implanter la fonction `tour` correspondant à la documentation et aux tests suivants. Si le coup n'est pas valide (numéro de tas ou nombre de jetons à retirer invalide), on ne modifie pas l'état du jeu.

```
/** Jouer un tour au jeu de Nim
 * @param etat un tableau d'entiers représentant l'état actuel du jeu
 * @param tas le numéro du tas à modifier
 * @param nbJetons le nombre de jetons à retirer du tas
 * @return l'état après avoir retiré les jetons du tas correspondant
 */
```

```
    ASSERT( tour({2,1,3,4}, 2, 2) == {2,1,1,4} );
    ASSERT( tour({1,2,3}, 0, 1) == {0,2,3} );
    ASSERT( tour({0,0}, 0, 1) == {0,0} );
```

Correction :

```
vector<int> tour(vector<int> etat, int tas, int nbJetons) {
    if ( tas >= etat.size() or tas < 0
            or nbJetons <= 0 or etat[tas] < nbJetons) {
        return etat;
    }
    etat[tas] -= nbJetons;
    return etat;
}
```

(4) On vous fournit une fonction `strategie` correspondant à la documentation suivante, qui étant donné un état du jeu vous renvoie un coup valide à jouer.

```
/** strategie
 * @param etat un tableau d'entiers représentant l'état actuel du jeu
 * @return un tableau d'entiers de taille 2 dont la première case est
 *         le numéro du tas à jouer et la deuxième case le nombre de
 *         jetons à retirer de ce tas
 */
```

Écrire la documentation et implanter une fonction `partie` qui, partant d'un état du jeu, joue les coups donnés par `strategie` jusqu'à ce que la partie soit finie, et qui renvoie un entier qui est le numéro du joueur gagnant (1 ou 2). Le joueur qui commence est le joueur 1. (Écrire votre réponse sur la page suivante.)

Correction :

```
/** Une partie complète
 * @param etat un tableau d'entiers représentant l'état de départ du jeu
 * @return numéro du joueur qui gagne la partie
 */
int partie(vector<int> etat) {
    int joueur = 2;
    while (not fini(etat)) {
        vector<int> strat = strategie(etat);
        etat = tour(etat, strat[0], strat[1]);
        if (joueur == 1) {
            joueur = 2;
        } else {
            joueur = 1;
        }
    }
    return joueur;
}
```

(5) Une première stratégie pour ce jeu consiste à choisir le premier tas non vide et retirer un seul jeton de ce tas. Implanter la fonction `strategie` en utilisant cette méthode. Votre fonction doit respecter la documentation donnée à la question précédente et les tests suivants :

```
    ASSERT( strategie({3, 2, 1})[0] == 0 );
    ASSERT( strategie({0, 2, 1})[0] == 1 );
    ASSERT( strategie({3, 2, 1})[1] == 1 );
```

Correction :

```
vector<int> strategie(vector<int> etat) {
    for(int i = 0; i < etat.size(); i++) {
        if (etat[i] != 0) {
            return {i, 1};
        }
    }
    return {0, 0};
}

vector<int> strategieOptimale(vector<int> etat) {
```

(6) Écrire des tests pour la fonction `partie` utilisant la stratégie décrite dans la question précédente.

Correction :

```
    ASSERT( partie({1}) == 1 );
    ASSERT( partie({2}) == 2 );
    ASSERT( partie({1, 1}) == 2);
```

(7) ♣ Implanter une fonction `afficheNimEtoiles` qui prend en paramètre un état du jeu de Nim et l'affiche en représentant les jetons par des astérisques et un tas par ligne, selon l'exemple ci-dessous.

```
afficheNimEtoiles({2, 0, 3, 4});
```

```
**

***
****
```

Correction :

```
void afficheNimEtoiles(vector<int> etat) {
    for (int i = 0; i < etat.size(); i++) {
        for (int j = 0; j < etat[i]; j++) {
            cout << "*";
        }
        cout << endl;
    }
}
```

(8) ♣ La stratégie optimale pour ce jeu est connue. Elle nécessite une opération arithmétique de base : le xor (“ou exclusif bit à bit”). Cette opération a pour symbole l’accent circonflexe ^ en C++ :

```
3 ^ 2 ^ 5 // se lit "3 xor 2 xor 5"
```

La stratégie optimale est la suivante :

(a) calculer le xor des nombres de jetons des tas (on appelle X cette quantité) ;

(b) si $X = 0$, on a perdu si l’adversaire joue parfaitement et on joue n’importe quoi ;

(c) sinon, on choisit un tas tel que le xor de X et du nombre de jeton du tas (appelons cette quantité Y) est strictement plus petit que le nombre de jetons du tas ;

(d) on retire de ce tas un certain nombre de jetons de manière à ce que le nouveau nombre de jetons du tas soit égal à Y.

Implanter une fonction `strategieOptimale` sur le modèle de la fonction `strategie` qui renvoie le coup optimal selon cette méthode. Dans le cas où X vaut zéro, on doit quand même jouer quelque chose : utiliser la fonction `strategie`.

Correction :

```
vector<int> strategieOptimale(vector<int> etat) {
    if (etat.size() == 0)
        return {0, 0};
    int X = etat[0];
    for (int i = 1; i < etat.size(); i++) {
        X = X ^ etat[i];
    }
    if (X == 0) { // Perdu :/
        return strategie(etat);
    }
    int j = 0;
    while (j < etat.size() and (etat[j] ^ X) >= etat[j]) {
        j++;
    }
    return {j, etat[j] - (etat[j] ^ X)};
}
```