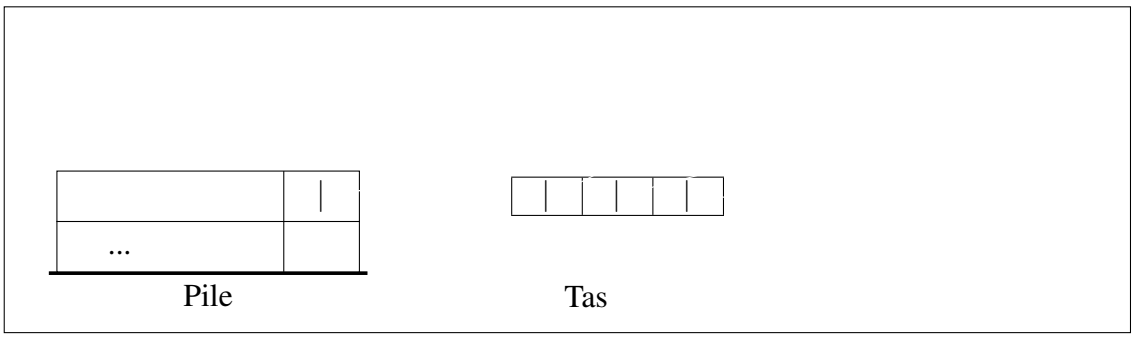
Faculté d'Orsay, Licence 1, Semestre 1, Programmation Impérative **Année 2020-2021**
Examen du 12 décembre 2020 (deux heures)

Calculatrices, téléphones mobiles et tout appareil électronique non autorisé doivent être éteints et déposés avec vos affaires personnelles.
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 partie du barème sur 100. Le barème est indicatif et sujet à ajustements.
Les réponses sont à donner, autant que possible, sur le sujet ; sinon, demander un intercalaire et mettre un renvoi.
Les enseignants collecteront votre copie à votre place.

Exercice 1 (Cours : tableaux à deux dimensions (15 points))**.**

(1) Illustrer les étapes de la construction d'un tableau à deux dimensions par un fragment de programme construisant un tableau avec trois lignes et quatre colonnes, initialisé de sorte que la valeur à la ligne i et colonne j soit $i + j$. Indiquer par des commentaires les étapes de la construction.

(2) En suivant les conventions du cours, complétez ci-dessous le dessin de la pile et du tas juste après l'exécution de votre fragment de programme.



Exercice 2 (J'aimerais tant voir Syracuse : fonctions, boucles, entrées-sorties (23 points))**.**
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 $N = 11$ est constituée par les termes

$$u_0 = 11, \qquad u_1 = f(11) = 3 \times 11 + 1 = 34, \qquad u_2 = f(34) = 34/2 = 17,$$
$$u_3 = f(17) = 3 \times 17 + 1 = 52, \qquad u_4 = f(52) = 52/2 = 26$$

les termes suivants étant $13,\ 40,\ 20,\ 10,\ 5,\ 16,\ 8,\ 4,\ 2,\ 1,\ 4,\ 2,\ 1,\ \ldots$

Nous admettons la « conjecture de Syracuse » : pour tout entier N strictement positif, la suite de Syracuse de N aboutit toujours aux termes 4, 2, 1 puis redonne indéfiniment ces termes 4, 2, 1.

(1) Implanter la fonction f dont la documentation est donnée ci dessous :
Indications : On rappelle que pour tous entiers a et b, le reste de la division euclidienne de a par b est donné en C++ par `a%b`. Ce reste est nul si et seulement si a est divisible par b.

```
/** Calcul du prochain terme de la suite de Syracuse
 * @param m: un entier
 * @return m/2 si m est pair, 3*m + 1 si m est impair
 **/
int f(int m) {




}
```

(2) Écrire un fragment de programme pour :
— faire saisir au clavier un entier N
— calculer (en utilisant la fonction `f` définie ci dessus) et afficher à l'écran tous les termes de la suite de Syracuse de N, jusqu'à ce que l'on obtienne un terme égal à 1.
Exemple d'exécution souhaitée pour ces instructions :

```
Veuillez saisir le premier terme: 20
Voici la suite: 20, 10, 5, 16, 8, 4, 2, 1
```

(3) Implanter une fonction `terme` (non récursive) 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. Par exemple, `terme(3, 11)` doit renvoyer 52, puisque le terme de rang 3 dans la suite de Syracuse de 11 est 52. Autre exemple : `terme(2, 13)` doit renvoyer 20.

```
/** Valeur du terme de rang n dans la suite de Syracuse de N
 * @param n: le rang du terme de la suite souhaité
 * @param N: le premier terme de la suite (terme de rang 0)
 * @return la valeur du terme de rang n dans la suite de Syracuse de N
   **/
```

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

(5) Implanter la fonction `tempsDeVol` dont la documentation et les tests sont donnés ci-dessous. Étant donné un entier N, cette fonction calcule et renvoie le « temps » nécessaire pour que la suite de Syracuse de N atteigne la valeur 1.

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

}
```

```
    ASSERT( tempsDeVol( 1) ==  0 );    ASSERT( tempsDeVol(2) == 1 );
    ASSERT( tempsDeVol( 4) ==  2 );    ASSERT( tempsDeVol(5) == 5 );
    ASSERT( tempsDeVol(11) == 14 );
```

Exercice 3 (tableaux, fichiers, fonctions, boucles (20 points))**.**
On réalise un programme permettant de déterminer le temps de trajet entre deux stations de train sur une ligne.

On représente une ligne de train par un fichier avec une ligne pour chaque tronçon (espace entre deux stations). Pour chaque tronçon, on écrit sa **longueur** en kilomètres (donc la distance séparant les deux stations), puis la **vitesse** à laquelle vont les trains sur ce tronçon. Cette vitesse est exprimée en *nombre de minutes nécessaires pour parcourir un kilomètre*.

Pour un fichier contenant n tronçons, il y a $n + 1$ stations que l'on numérote de 0 à n.

Le fichier suivant représente une ligne avec 4 stations. Les deux premières y sont séparées de 30 km. Les trains mettent 3 minutes pour parcourir un kilomètre sur le tronçon entre les deux dernières stations.

train.txt

```
30 5
40 2
50 1
5 3
```

(1) Complétez le fragment de programme suivant pour construire et initaliser les tableaux `distances` et `vitesses` en ouvrant et lisant `"train.txt"`, de manière à faire passer les tests. Le fichier est au format indiqué ci-dessus, mais pourrait contenir d'autres valeurs, auquel cas votre code doit rester correct en adaptant juste les tests.

```
    vector<int> distances;
    vector<int> vitesses;
```

```
    ASSERT( distances == vector<int>({30, 40, 50, 5}) );
    ASSERT( vitesses  == vector<int>({5, 2, 1, 3})    );
```

(2) Implantez la fonction `tempsTotal` dont la documentation et les tests sont donnés :

```
/** Renvoie le temps total nécessaire pour parcourir toute la ligne
 * @param distances : tableau contenant la longueur de chaque tronçon
 * @param vitesses : tableau contenant le nombre de minutes nécessaires
 *     pour parcourir un kilomètre pour chaque tronçon
 * @return : le temps total en minutes pour parcourir tous les tronçons
 *     en prenant en compte les vitesses
 **/
int tempsTotal(vector<int> distances, vector<int> vitesses) {


}
```

```
    ASSERT( tempsTotal(vector<int>({100, 50}),
                       vector<int>({1, 2})) == 200 );
    ASSERT( tempsTotal(vector<int>({30, 40, 50, 5}),
                       vector<int>({5, 2, 1, 3})) == 295 );
```

(3) Implantez la fonction `sousTableau` dont la documentation et les tests sont donnés :

```
/** Renvoie un sous-tableau de tab aux bornes données
 * @param tab : un tableau d'entiers
 * @param debut : un indice de tab
 * @param fin : un indice de tab strictement supérieur à debut
 * @return : un tableau contenant
 *   les éléments {tab[debut], tab[debut+1], ..., tab[fin]}
 **/
vector<int> sousTableau(vector<int> tab, int debut, int fin) {
```

```
    ASSERT( sousTableau(vector<int>({1, 2, 3, 4}), 0, 3)
            == vector<int>({1, 2, 3, 4}) );
    ASSERT( sousTableau(vector<int>({1, 2, 3, 4}), 1, 2)
            == vector<int>({2, 3}) );
```

(4) Implantez la fonction `tempsTrajet` dont la documentation et les tests sont donnés ci-dessous. **Rappel** : s'il y a n tronçons, il y a $n + 1$ stations.

```
/** Renvoie le temps nécessaire à aller d'une gare à l'autre
 * @param distances : tableau contenant la longueur de chaque tronçon
 * @param vitesses : tableau contenant le nombre de minutes nécessaires
 *     pour parcourir un kilomètre pour chaque tronçon
 * @param depart : numéro de gare
 * @param arrivee : numéro de gare supérieur à arrivee
 * @return temps nécessaire pour aller de la gare depart
 *     à la gare arrivee
 */
int tempsTrajet(vector<int> distances, vector<int> vitesses,
                int depart, int arrivee) {



}
```

```
    ASSERT( tempsTrajet(vector<int>({30, 40, 50, 5}),
                        vector<int>({5, 2, 1, 3}), 0, 4) == 295 );
    ASSERT( tempsTrajet(vector<int>({30, 40, 50, 5}),
                        vector<int>({5, 2, 1, 3}), 1, 2) == 80  );
```

Tous les exercices et toutes les questions qui suivent sont indépendantes et concernent des manipulations de tableaux à 1 ou 2 dimensions : le jeu Puissance 4 ne sert qu'à donner le contexte. Vous n'avez besoin ni de savoir jouer ni de connaître le détail des règles.

`Puissance 4` est un jeu où deux joueurs font tomber chacun à son tour un pion dans une grille verticale comptant 6 rangées et 7 colonnes, dans le but d'aligner 4 pions.

Dans les exercices suivants, on s'intéresse à l'écriture d'un programme pour jouer à Puissance 4. Par convention, on représentera par 1 le joueur 1 ou un de ses pions, par 2 le joueur 2 ou un de ses pions et par 0 une case vide. La grille de jeu sera représentée par un `vector<vector<int>>`, de sorte que `grille[3]` représente la quatrième rangée de la grille en partant du haut.

Pour tester les fonctions, on définit la variable globale suivante :

```
vector<vector<int>> grille = {
    {1,0,0,0,0,2,1},
    {1,1,0,2,0,2,2},
    {2,1,1,2,0,2,2},
    {2,1,1,1,1,2,1},
    {2,2,1,2,2,1,2},
    {1,2,2,1,2,1,1} };
```

Exercice 4 (Affichage et sauvegarde : tableaux 2d, entrées-sorties, fichiers (12 points))**.**
La fonction suivante a été proposée pour afficher la grille de jeu :

```
/** Affiche la grille de puissance 4 sur le terminal
 * @param grille une grille de puissance 4
 **/
void afficheGrille(vector<vector<int>> grille) {
    for ( int i = 0; i < grille.size(); i++ ) {
        for ( int j = 0; j < grille[i].size(); j++ ) {
            cout << grille[i][j];
        }
    }
}
```

Lorsqu'on effectue l'appel suivant :

```
afficheGrille(grille);
```

on obtient l'affichage :

```
1000021110202221120222111121221221212211
```

(1) Retouchez le code ci-dessus (vous pouvez corriger directement sur le code) pour que la grille soit affichée ligne à ligne en séparant les colonnes par un espace. L'appel ci-dessus devrait donc donner :

```
1 0 0 0 0 2 1
1 1 0 2 0 2 2
2 1 1 2 0 2 2
2 1 1 1 1 2 1
2 2 1 2 2 1 2
1 2 2 1 2 1 1
```

(2) On voudrait un système pour pouvoir sauvegarder / charger des parties en cours. On utilise un format de fichier très simple : le fichier contient le mot « Puissance4 » (pour identifier le jeu) sur la première ligne puis une grille de jeu écrite ligne à ligne comme pour l'affichage. Ainsi, notre exemple `grille` sera représenté par le fichier :

```
Puissance4
1 0 0 0 0 2 1
1 1 0 2 0 2 2
2 1 1 2 0 2 2
2 1 1 1 1 2 1
2 2 1 2 2 1 2
1 2 2 1 2 1 1
```

Écrivez la fonction `chargePartie` qui permet de charger une partie précédemment sauvegardée dans un fichier suivant le format décrit. La fonction prend en paramètre le nom du fichier et renvoie la grille correspondante lue dans le fichier.

```
/** Charge une partie sauvegardée dans un fichier
 * La première ligne du fichier contient uniquement "Puissance4"
 * Le fichier contient ensuite la grille ligne par ligne en représentant
 * les pions par les numéros des joueurs
 * @param nomFichier le nom du fichier
 * @return la grille de puissance 4 du fichier (6 lignes et 7 cols)
 **/
vector<vector<int>> chargePartie(string nomFichier) {

}
```

Exercice 5 (Trouver le gagnant : tableaux 1d et 2d (30 points))**.**
Dans cet exercice, nous allons écrire différentes fonctions permettant de tester si un joueur a gagné en alignant 4 pions sur une ligne, colonne ou diagonale. **Les questions sont indépendantes.**

(1) Complétez les tests de la fonction ci-dessous pour :
- un tableau vide
- un tableau de taille 7 contenant une série de quatre 2 alignés
- un tableau de taille 7 contenant quatre 1 mais non alignés

```
/** Teste si 'tableau' contient 'valeur' dans au moins 4 cases consécutives
 *  @param tableau un tableau d'entiers à une dimension
 *  @param valeur  un entier
 *  @return un booléen
**/
bool estGagnant1D(vector<int> tableau, int valeur) {
    int compteur = 0;
    for ( int i = 0; i < tableau.size(); i++ ) {
        if ( tableau[i] == valeur ) {
            compteur = compteur + 1;
            if ( compteur == 4 )
                return true;
        } else {
            compteur = 0;
        }
    }
    return false;
}
```

```
    ASSERT(     estGagnant1D({2,2,1,1,1,1,0}, 1));
    ASSERT( not estGagnant1D({2,2,1,1,1,1,0}, 2));
```

(2) Écrivez la fonction `ligneGagnante` documentée et testée ci-dessous. **Vous pouvez utiliser la fonction précédente si vous le souhaitez.**

```
/** Teste si au moins une ligne de la grille contient
    au moins 4 pions alignés du 'joueur'
 * @param grille une grille de puissance 4
 * @param joueur un entier 1 ou 2 désignant le joueur
 * @return un booleen
 **/
```

```
    ASSERT(     ligneGagnante(grille, 1) ); // j1 gagne ligne 3
    ASSERT( not ligneGagnante(grille, 2) ); // j2 n'a pas de ligne gagnante
```

(3) Pour tester les colonnes, on écrit d'abord une fonction permettant d'extraire une colonne de la grille sous forme d'un tableau à une dimension. Complétez la fonction ci-dessous dont on vous fournit la documentation, la signature et les tests.

```
/** Extraction d'une colonne
 * Renvoie la colonne j sous forme d'un tableau 1D
 * @param grille une grille de puissance 4
 * @return la colonne d'indice j
 **/
vector<int> extractionColonne(vector<vector<int>> grille, int j) {




}
```

```
    ASSERT( extractionColonne(grille,0) == vector<int>({1,1,2,2,2,1}));
    ASSERT( extractionColonne(grille,1) == vector<int>({0,1,1,1,2,2}));
    ASSERT( extractionColonne(grille,2) == vector<int>({0,0,1,1,1,2}));
```

(4) Écrivez la fonction `colonneGagnante` qui prend en paramètre une grille et un numéro de joueur renvoie vrai si le joueur a aligné 4 pions sur au moins une colonne de la grille. On vous fourni la documentation et les tests. **Vous pouvez utiliser les fonctions précédentes si vous le souhaitez.**

```
/** Teste si 4 pions de 'joueur' sont alignés sur une colonne
 * @param grille une grille de puissance 4
 * @param joueur un entier 1 ou 2 désignant le joueur
 * @return un booleen
 **/
```

```
    ASSERT(not colonneGagnante(grille,1));
    ASSERT(colonneGagnante(grille,2));
```

(5) Les tests des alignements en diagonale sont plus difficile. On écrit une fonction simplifiée qui ne vérifie que la diagonale *principale*, c'est-à-dire celle-ci :

```
x . . . . . .
. x . . . . .
. . x . . . .
. . . x . . .
. . . . x . .
. . . . . x .
```

Complétez la fonction dont on vous fournit la documentation, la signature et les tests.

```
/** Teste 4 pions de 'joueur' sont alignés sur la diagonale principale
 * @param grille une grille de puissance 4
 * @param joueur un entier 1 ou 2 désignant le joueur
 * @return un booleen
 **/
bool diagonalePrincipaleGagnante(vector<vector<int>> grille, int joueur) {









}
```

```
    ASSERT(diagonalePrincipaleGagnante(grille,1));
    ASSERT(not diagonalePrincipaleGagnante(grille,2));
```

(6) ♣ Adaptez votre fonction pour tester les alignements sur toutes les *diagonales gauches* :

```
c d e f . . .
b c d e f . .
a b c d e f .
. a b c d e f
. . a b c d e
. . . a b c d
```

```
/** Teste si 4 pions de 'joueur' sont alignés sur au moins
 *  une des diagonales gauche.
 * @param grille une grille de puissance 4
 * @param joueur un entier 1 ou 2 désignant le joueur
 * @return un booleen
 **/
```