Algorithmes et structures de données I (420-V11-SF)

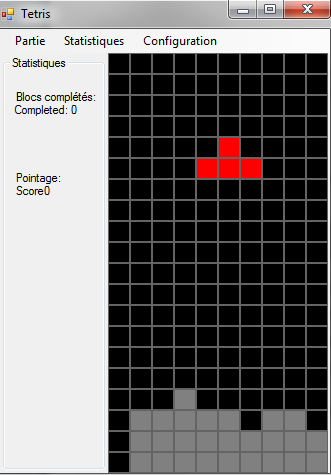
Travail pratique 3

Tetris

**Présentation sommaire du jeu**

Le jeu **Tetris** n’a plus vraiment besoin de présentation. Il est considéré comme un des plus grands classiques du jeu vidéo (aux côtés de Pong[[1]](#footnote-1)). Dans ce travail pratique, nous allons d’abord mettre en place les outils nécessaires à la réalisation du programme. Nous allons ensuite développer les algorithmes clés de celui-ci pour finalement élaborer quelques jeux d’essais pour s’assurer de la qualité de nos algorithmes.

Encore une fois, ce travail est fortement arrimé avec le Travail Pratique 3 du cours **Programmation de jeux vidéo 1**. Le visuel proposé dans ce cours pour le jeu de Tetris s’apparentera à celui-ci :



**Contexte de réalisation**

Ce travail pratique sera réalisé en équipe de 2 personnes. Il sera constitué d’un mandat commun (les 2 coéquipiers devront travailler en équipe sur ce mandat) et d’un mandat individuel (chaque coéquipier travaillera sur son propre mandat). Ce qui produira au final une évaluation différente pour les 2 coéquipiers. Il est à noter que les mandats individuels 1 et 2 représentent la même charge de travail et n’avantagent en aucun cas un des 2 coéquipiers. L’attribution des mandats est donc libre et doit se faire sous un commun accord entre les coéquipiers.

**Description sommaire des mandats :**

|  |  |
| --- | --- |
| **Mandat Commun (travail d’équipe) (55%)** | |
| * Analyse des variables partagées * Positionnement des blocs * Calculs des rotations * Algorithme de déplacement possible | |
| **Mandat Individuel 1 (45%)** | **Mandat Individuel 2 (45%)** |
| * Positionnement des blocs * Algorithme de ligne complétée * Jeux d’essais de déplacements | * Positionnement des blocs * Algorithme de décalage de lignes * Jeux d’essais de déplacements |

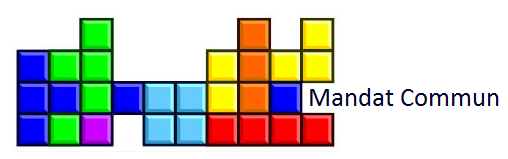
Les mandats individuels seront évalués selon la politique de travail individuel du département. Référez-vous aux règles départementales pour éviter d’être en situation de conflit avec ces règles.

**Détail des mandats**

Il y aura une seule remise par équipe, ce qui signifie que les coéquipiers devront assembler une copie finale de leur travail en s’assurant que les mandats sont dûment complétés en répondant aux questions en écrivant en **vert** sur **fond jaune.**

Ne sous-estimez pas cette opération de mise en commun du travail. Prenez-vous d’avance, une seule remise sera acceptée.

Les coéquipiers devront clairement indiquer quel mandat individuel ils ont réalisé. **Autrement, le professeur ne corrigera pas un mandat individuel non-identifié ne sachant pas à qui attribuer la note.**



|  |  |
| --- | --- |
| **Mandat réalisé par :** | **Les 2 coéquipiers** |

**Partie A - Analyse de la situation**

La création de ce programme nécessite l’utilisation de divers tableaux afin de représenter l’état du jeu, de calculer les différents déplacements (ex : descente, mouvements latéraux, rotations), d’afficher la planche de jeu, etc.

La première partie de ce mandat commun consistera à bien comprendre le rôle de ces variables partagées afin de pouvoir les utiliser adéquatement dans divers algorithmes.

Lorsque vous recevrez le code de départ de ce TP, vous verrez la déclaration de quelques variables **partagées** dans le programme :

|  |
| --- |
| //Nombre de colonnes dans le jeu  int nbColonnesJeu = 10; |
| //Nombre de lignes dans le jeu  int nbLignesJeu = 20; |
| //Énumération des différents types de blocs dans le jeu  enum TypeBloc {None, Gelé, Carré, Ligne, T, L, J, S, Z} |
| //Représentation visuelle du jeu  PictureBox[,] toutesImagesVisuelles = null; |

Bien que nous possédions un tableau de PictureBox pour l’affichage du jeu, nous n’allons pas effectuer nos calculs à partir de celui-ci.

*Exemple de calcul : Est-ce que le bloc peut descendre d’un étage ? (autrement dit, sommes-nous rendus au plancher du jeu ? Sommes-nous appuyés sur un autre bloc gelé dans le jeu ?, Etc.)*

**Question A1**

Pour pouvoir effectuer ces calculs, nous allons travailler sur un tableau 2D partagé autre que celui des PictureBox. En considérant les variables partagées données précédemment, expliquez en vos mots quel genre de tableau allez-vous créer pour pouvoir faire les calculs (voir exemple) dans le jeu. Soyez précis sur son contenu et son utilisation.

|  |
| --- |
| Le tableau 2D créé sera de type *TypeBloc* qui contient l’énumération des différents types de blocs dans le jeu. Par exemple, lors de l’absence de pièce du jeu, la valeur *None* sera attribuée*.*  La taille du tableau est alors selon les valeurs des variables *nbColonnesJeu* et *nbLignesJeu* partagées. |

**Question A2**

Donnez la déclaration pseudocode du tableau décrit dans la question A1 :

|  |
| --- |
| Déclarer tableauDeJeu tableau 2D de nbLignesJeu \* nbColonnesJeu TypeBloc |

**Question A3**

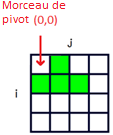
Au chargement du jeu, nous allons remplir ce tableau avec une valeur appropriée. Donnez le pseudocode nécessaire pour effectuer le remplissage :

|  |
| --- |
| Pour i de 0 à Taille(tableauDeJeu) -1  Pour j de 0 à Taille(tableauDeJeu[0]) – 1  tableauDeJeu[i][j] 🡨 TypeBloc.None  FinPour  FinPour |

**Partie B - Algorithmes**

Les calculs seront essentiellement effectués sur le bloc en descente (celui contrôlé par le joueur) que nous allons appeler le bloc actif.

Afin de faciliter les calculs, nous allons adopter une stratégie dans laquelle nous allons représenter un bloc actif en mémoire de cette façon (exemple avec le bloc en « T ») :



Et afin de simplifier la syntaxe de nos algorithmes, nous allons stocker les informations du bloc actif dans 2 tableaux de 1 dimension **partagés** qui contiennent respectivement les valeurs **i** et les valeurs **j** des 4 morceaux du bloc (le coin supérieur gauche de chaque morceau). Pour ceux et celles qui s’y retrouvent mieux dans un espace cartésien, on parlera de **Y et X respectivement.** Ces valeurs représentent donc une position relative par rapport au bloc de rotation (0,0).

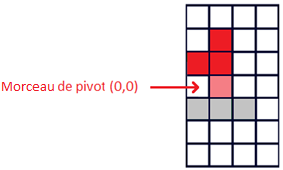
|  |
| --- |
| // Positions relatives des morceaux d’un bloc actif en forme de « T »  // Ces tableaux sont partagés dans le programme :  blocActifY[0] 🡨 0  blocActifY[1] 🡨 1  blocActifY[2] 🡨 1  blocActifY[3] 🡨 1  blocActifX[0] 🡨 1  blocActifX[1] 🡨 0  blocActifX[2] 🡨 1  blocActifX[3] 🡨 2  //Donc les 4 positions relatives des morceaux sont : [0][1], [1][0], [1][1] et [1][2]  //Voir illustration du bloc en « T » pour visualiser ces coordonnées |

Les blocs pivoteront autour du morceau (0,0). Ceci rendra uniforme les calculs de rotation de tous les blocs. Autrement dit, le morceau situé en (0,0) tournera sur lui-même, ce qui revient à dire que visuellement dans le jeu on aura l’impression qu’il ne bouge pas lors de la rotation.

Nous nous serviront donc des tableaux « blocActifY » et « blocActifX » pour représenter l’état du bloc selon la rotation qui lui a été appliquée (quelles sont ses positions relatives courantes).

**Exemple :**

Si nous effectuons une rotation antihoraire de 90 degrés sur le bloc illustré plus haut autour du bloc de pivot, nous serons dans cette nouvelle position :



La partie en rouge représentant la position finale après une rotation antihoraire

Nous allons forcément nous retrouver avec certaines positions relatives négatives, ce qui ne causera pas problème outre mesure dans le jeu.

**Question B1**

Considérant l’exemple de rotation précédent, indiquez les nouvelles positions du bloc actif :

|  |
| --- |
| **//Positions relatives du bloc en forme de « T » une fois la rotation antihoraire de 90 degrés**  **//effectuée depuis la position initiale**  blocActifY[0] 🡨 0  blocActifY[1] 🡨 -1  blocActifY[2] 🡨 -1  blocActifY[3] 🡨 -2  blocActifX[0] 🡨 1  blocActifX[1] 🡨 0  blocActifX[2] 🡨 1  blocActifX[3] 🡨 1  //Donc les nouvelles positions relatives des morceaux sont : [0][1], [-1][0], [-1][1] et [-2][1] |

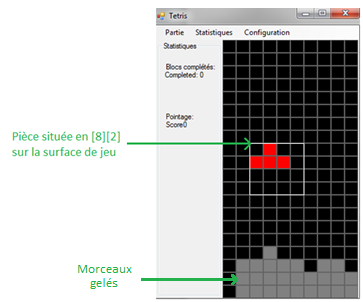
**Question B2**

Tel que mentionné, cette technique nous permettra de calculer les nouvelles positions de façon uniforme (en une seule formule) peu importe le type de bloc qui subit une rotation. Écrivez les formules de rotation pour le sens horaire et antihoraire (truc : dessinez la rotation de plusieurs types de bloc et déduisez les formules) :

|  |
| --- |
| **//Rotation de 90 degrés Antihoraire**  blocActifNouveauY[0] 🡨 -1 \* blocActifX[0]  blocActifNouveauY[1] 🡨 -1 \* blocActifX[1]  blocActifNouveauY[2] 🡨 -1 \* blocActifX[2]  blocActifNouveauY[3] 🡨 -1 \* blocActifX[3]  blocActifNouveauX[0] 🡨 blocACtifY[0]  blocActifNouveauX[1] 🡨 blocACtifY[1]  blocActifNouveauX[2] 🡨 blocACtifY[2]  blocActifNouveauX[3] 🡨 blocACtifY[3]  **//Rotation de 90 degrés Horaire**  blocActifNouveauY[0] 🡨 -1 \* blocActifX[0]  blocActifNouveauY[1] 🡨 -1 \* blocActifX[1]  blocActifNouveauY[2] 🡨 -1 \* blocActifX[2]  blocActifNouveauY[3] 🡨 -1 \* blocActifX[3]  blocActifNouveauX[0] 🡨 blocACtifY[0]  blocActifNouveauX[1] 🡨 blocACtifY[1]  blocActifNouveauX[2] 🡨 blocACtifY[2]  blocActifNouveauX[3] 🡨 blocACtifY[3] |

**Question B3**

Nous savons maintenant comment la pièce sera représentée dans l’espace mais il nous reste à savoir où elle se situe dans le plan de jeu. Afin de pallier au problème, nous allons maintenir 2 variables partagées qui nous aideront à la situer : **colonneCourante** et **ligneCourante**.

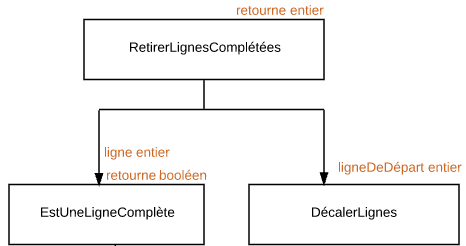


Nous avons maintenant tout ce qu’il faut (voir Sections A et B) pour faire les différents calculs sur les déplacements possibles. Complétez le pseudocode de la fonction **BlocPeutBouger** qui prend en paramètre un sens de déplacement valide (descente, gauche, droite, rotationHoraire, rotationAntihoraire) et qui retourne un booléen. On devra considérer tous les cas (attention si vous êtes à proximité d’un morceau qui a été gelé).



|  |
| --- |
| **Fonction BlocPeutBouger(sens Déplacement) retourne booléen**  **Déclarer peutBouger booléen**  **peutBouger 🡨 true**  **Si (sens == DESCENTE) alors**  Pour i de 0 à 3  Si tableauDeJeu[blocActifY[i] + 1][blocActifX[i]] == TypeBloc.None  peutBouger 🡨 true  Sinon  peutBouger 🡨 false  FinSi  FinPour  **SinonSi (sens == GAUCHE) alors**  Pour i de 0 à 3  Si tableauDeJeu[blocActifY[i]][blocActifX[i] - 1] == TypeBloc.None  peutBouger 🡨 true  Sinon  peutBouger 🡨 false  FinSi  FinPour  **SinonSi (sens == DROITE) alors**  Pour i de 0 à 3  Si tableauDeJeu[blocActifY[i]][blocActifX[i] + 1] == TypeBloc.None  peutBouger 🡨 true  Sinon  peutBouger 🡨 false  FinSi  FinPour  **SinonSi (sens == ROTATION\_ANTIHORAIRE)**  Pour i de 0 à 3  Si tableauDeJeu[blocActifNouveauY[i]][blocActifNouveauX[i]] == TypeBloc.None  peutBouer 🡨 true  Sinon  peutBouger 🡨 false  FinSi  FinPour  **Sinon //ROTATION\_HORAIRE**  Pour i de 0 à 3  Si tableauDeJeu[blocActifNouveauY[i]][blocActifNouveauX[i]] == TypeBloc.None  peutBouger 🡨 true  Sinon  peutBouger 🡨 false  FinSi  FinPour  **FinSi**  **Retourner peutBouger**  **FinFonction** |

Une des fonctionnalités principales du jeu consiste à retirer les lignes complétées. Voici le diagramme des fonctions responsables de cette fonctionnalité :



**Question B4**

Donnez la documentation, la déclaration, ainsi que le pseudocode de la fonction **RetirerLignesComplétées.**

**Note** : On ne demande pas de donner le code des 2 autres fonctions, seulement de les appeler au moment opportun dans la fonction :

|  |
| --- |
| //Rôle : Retirer les lignes complétées et décaler ensuite les lignes gelées.  //Paramètre : Aucun  //Retour : Retourne un entier qui représente le nombre de lignes complétées.  //Déclaration et pseudocode de la fonction ici  Fonction RetirerLignesComplétées() retourne entier  Déclarer nbLignesComplétées entier  Déclarer ligneComplète booléen  nbLignesComplétées 🡨 0    Pour i de 0 à nbLignesJeu  ligneComplète 🡨 estUneLigneComplète(i)  Si ligneComplète == true alors  Pour j de 0 à nbColonnesJeu  tableauDeJeu[i][j] 🡨 TypeBloc.None  FinPour  nbLignesComplétées++  FinSi  FinPour  DécalerLignes(nbLignesJeu - nbLignesComplétées)  Retourner nbLignesComplétées  FinFonction |

**FIN DU MANDAT COMMUN**



|  |  |
| --- | --- |
| **Mandat réalisé par :** | Kevin R. Leclaire |

**Partie A - Analyse**

**Question A1**

En s’inspirant de l’exemple donné dans le mandat commun, donnez les positions relatives des blocs suivants selon cette disposition :



|  |
| --- |
| **//Positions relatives de la pièce carrée :**  blocActifY[0] 🡨  blocActifY[1] 🡨  blocActifY[2] 🡨  blocActifY[3] 🡨  blocActifX[0] 🡨  blocActifX[1] 🡨  blocActifX[2] 🡨  blocActifX[3] 🡨  **//Positions relatives de la pièce « L » :**  blocActifY[0] 🡨  blocActifY[1] 🡨  blocActifY[2] 🡨  blocActifY[3] 🡨  blocActifX[0] 🡨  blocActifX[1] 🡨  blocActifX[2] 🡨  blocActifX[3] 🡨  **//Positions relatives de la pièce « S » :**  blocActifY[0] 🡨  blocActifY[1] 🡨  blocActifY[2] 🡨  blocActifY[3] 🡨  blocActifX[0] 🡨  blocActifX[1] 🡨  blocActifX[2] 🡨  blocActifX[3] 🡨 |

**Partie B – Algorithmes**

**Question B1**

Écrivez la documentation, la déclaration et le pseudocode de la fonction **EstUneLigneComplète** qui est appelée dans l’algorithme du mandat commun :

|  |
| --- |
| //Rôle :  //Paramètre :  //Retour :  //Déclaration et pseudocode de la fonction ici |

**Partie C – Jeux d’essais**

Dans cette section, nous allons tester la fonction **BlocPeutBouger**élaborée dans le mandat commun. Cette fonction est cruciale dans le jeu car nous voulons à tout prix éviter des anomalies au niveau des mouvements et rotations (bloc qui sort de la surface de jeu, qui passe au travers d’un morceau gelé, etc.). Parmi le choix des jeux d’essais à effectuer sur le bloc actif, il faudra donc considérer la proximité des limites du jeu et morceaux qui sont gelés pour bâtir de bons tests.

**Rappel pour écrire un bon jeu d’essai** :

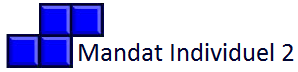
1. Choisir un cas pertinent selon la catégorie demandée (général, particulier)
2. Préparer les données adéquatement à l’aide des variables partagées
3. Appeler la fonction à tester
4. Effectuer la ou les affirmations

**Question C1**

Écrire le pseudocode des jeux d’essais de descente, mouvement à droite et rotation horaire **avec un bloc actif de type « L »** en considérant les 4 étapes mentionnées ci-haut. Relire le pseudocode de la fonction développée pour connaître les bons paramètres à passer et revoir le mandat commun pour les variables partagées disponibles. Revoir aussi les notions de rotation afin de bien cibler les cas intéressants dans ce genre de mouvement.

|  |
| --- |
| **Fonction Tester\_BlocPeutBouger()**      **//1. Test de la descente du bloc actif :**    **//Jeu d’essai général pour la descente :**  **//Jeu d’essai particulier #1 pour la descente :**  **//Jeu d’essai particulier #2 pour la descente :**  **//2. Test du mouvement à droite du bloc actif :**    **//Jeu d’essai général pour le mouvement à droite :**  **//Jeu d’essai particulier #1 pour le mouvement à droite :**  **//Jeu d’essai particulier #2 pour le mouvement à droite :**  **//3. Test de la rotation horaire du bloc actif :**    **//Jeu d’essai général pour rotation horaire :**  **//Jeu d’essai particulier #1 pour la rotation horaire :**  **//Jeu d’essai particulier #2 pour la rotation horaire :**  **//Jeu d’essai particulier #3 pour la rotation horaire :**  **FinFonction** |

**FIN DU MANDAT INDIVIDUEL #1**

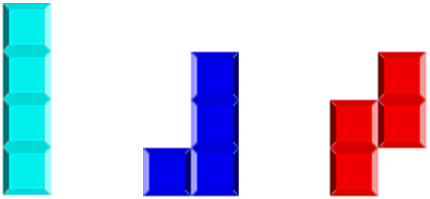


|  |  |
| --- | --- |
| **Mandat réalisé par :** | Jonathan Roy Noël |

**Partie A - Analyse**

**Question A1**

En s’inspirant de l’exemple donné dans le mandat commun, donnez les positions relatives des blocs suivants selon cette disposition :



|  |
| --- |
| **//Positions relatives de la pièce ligne :**  blocActifY[0] 🡨 0  blocActifY[1] 🡨 1  blocActifY[2] 🡨 2  blocActifY[3] 🡨 3  blocActifX[0] 🡨 1  blocActifX[1] 🡨 1  blocActifX[2] 🡨 1  blocActifX[3] 🡨 1  **//Positions relatives de la pièce « J » :**  blocActifY[0] 🡨 1  blocActifY[1] 🡨 2  blocActifY[2] 🡨 3  blocActifY[3] 🡨 3  blocActifX[0] 🡨 2  blocActifX[1] 🡨 2  blocActifX[2] 🡨 2  blocActifX[3] 🡨 1  **//Positions relatives de la pièce « Z » :**  blocActifY[0] 🡨 1  blocActifY[1] 🡨 2  blocActifY[2] 🡨 2  blocActifY[3] 🡨 3  blocActifX[0] 🡨 2  blocActifX[1] 🡨 2  blocActifX[2] 🡨 1  blocActifX[3] 🡨 1 |

**Partie B – Algorithmes**

**Question B1**

Écrivez la documentation, la déclaration et le pseudocode de la fonction **DécalerLignes** qui est appelée dans l’algorithme du mandat commun :

|  |
| --- |
| //Rôle : Décale les lignes afin d’enlever l’espace créer à la suite de la //complétion des lignes.  //Paramètre : ligneDeDépart entier, représente la ligne de départ à décaler.  //Retour : Aucun  //Déclaration et pseudocode de la fonction ici  Fonction DécalerLignes (lignedeDépart entier)  Pour i de ligneDeDépart à 0 + 1  Pour j de 0 à Taille(tableauDeJeu[0]) – 1  Si tableauDeJeu[i][j] == TypeBloc.Gelé  tableauDeJeu[i][j] 🡨 tableauDeJeu[i + 1][j]  FinSi  tableauDeJeu[i + 1][j] 🡨 TypeBloc.None  FinPour  FinPour |

**Partie C – Jeux d’essais**

Dans cette section, nous allons tester la fonction **BlocPeutBouger**élaborée dans le mandat commun. Cette fonction est cruciale dans le jeu car nous voulons à tout prix éviter des anomalies au niveau des mouvements et rotations (bloc qui sort de la surface de jeu, qui passe au travers d’un morceau gelé, etc.). Parmi le choix des jeux d’essais à effectuer sur le bloc actif, il faudra donc considérer la proximité des limites du jeu et morceaux qui sont gelés pour bâtir de bons tests.

**Rappel pour écrire un bon jeu d’essai** :

1. Choisir un cas pertinent selon la catégorie demandée (général, particulier)
2. Préparer les données adéquatement à l’aide des variables partagées
3. Appeler la fonction à tester
4. Effectuer la ou les affirmations

**Question C1**

Écrire le pseudocode des jeux d’essais de mouvement à gauche et rotation antihoraire **avec un bloc actif de type « ligne »** en considérant les 4 étapes mentionnées ci-haut. Relire le pseudocode de la fonction pour connaître les bons paramètres à passer et revoir le mandat commun pour les variables partagées disponibles. Revoir aussi les notions de rotation afin de bien cibler les cas intéressants dans ce genre de mouvement.

|  |
| --- |
| **Fonction Tester\_BlocPeutBouger()**      **//1. Test mouvement à gauche du bloc actif :**    **//Jeu d’essai général pour le mouvement à gauche :**  **//Jeu d’essai particulier #1 pour le mouvement à gauche :**  **//Jeu d’essai particulier #2 le mouvement à gauche :**  **//2. Test de la rotation antihoraire du bloc actif :**    **//Jeu d’essai général pour la rotation antihoraire :**  **//Jeu d’essai particulier #1 pour la rotation antihoraire :**  **//Jeu d’essai particulier #2 pour la rotation antihoraire :**    **//Jeu d’essai particulier #3 pour la rotation antihoraire :**  **//Jeu d’essai particulier #4 pour la rotation antihoraire :**  **FinFonction** |

**FIN DU MANDAT INDIVIDUEL #2**

**Modalités de remise**

**Travail pratique**

* Complétez les espaces jaunes avec les réponses aux questions. Les réponses doivent être en vert sur fond jaune tel que formaté dans le document original.
* Identifiez clairement les auteurs des mandats individuels.
* Remettre **UN SEUL DOCUMENT .docx** par équipe via Omnivox selon la modalité suivante :
  + Nommez votre fichier XXXXXXX\_YYYYYYY\_TP3.docx où « XXXXXXX » et « YYYYYYY » représentent les matricules des 2 coéquipiers

**Auto-Évaluation**

Chaque coéquipier devra remplir le document « AutoÉvaluation.docx » afin de donner son évaluation du travail d’équipe dans le mandat commun. Ce document est **strictement confidentiel** et fera l’objet d’une remise **individuelle**. Le professeur pourra pondérer à la baisse la note du mandat commun pour un étudiant dont il a jugé la participation insatisfaisante (par souci d’équité).

**Dates de remise  (TP et Évaluations) :**

**Rappels :**

* **1 SEUL DOCUMENT DU TP3** par équipe à remettre pour l’énoncé prévu sur Omnivox
* **TOUS LES ÉTUDIANTS** devront remettre leur Auto-Évaluation pour l’énoncé prévu sur Omnivox

**Remises du Groupe 00002** : **Lundi 28 novembre 8h00**

**Remises du Groupe 00001** : **Mardi 29 novembre 8h00**

 **Note importante :** Toutes ces modalités sont sujettes à des pénalités si non respectées.

BON TRAVAIL !

1. Réf : Wikipédia [↑](#footnote-ref-1)