# Rapport
# Projet IGI-2201

**Clouard Adam**

**Brenner Augustin**

2023
E2 - Groupe 3

# Index

## Exercice 1 :

Cette fonction glouton sert à calculer le nombre de pucerons mangés par la coccinelle en partant de n'importe quelle case de la première ligne. Après avoir fait les initialisations, on entre dans une boucle "while" avec comme condition d'arrêt l'arrivée de la coccinelle à la dernière ligne (la ligne du haut).

```
/**
* Calcule le nombre de pucerons mangés par une coccinelle
* @param G (int[][]) : la grille de pucerons
* @param d (int) : la colonne de départ
* @return : le nombre de pucerons qu'une coccinelle ayant atterri sur la case
(0, d) mangera sur son chemin glouton
*/
public int glouton(int[][] G, int d) {
   int L = G.length; // Nombre de lignes
   int C = G[0].length; // Nombre de colonnes
   int puceronsManges = 0; // Nombre de pucerons mangés
   int i = L - 1; // La coccinelle commence sur la dernière ligne
   int j = d; // Colonne de départ
   while (i > 0) { // Tant qu'on n'est pas sur la ligne du haut
       puceronsManges += G[i][j]; // Mange les pucerons sur la ligne actuelle
```

La coccinelle commence par manger les pucerons sur la case de départ, puis va comparer pour chacune des 3 cases possibles (nord-ouest, nord et nord-est) le nombre maximum de pucerons entre les 3. Une fois le maximum trouvé, on incrémente le nombre de pucerons mangés du nombre de pucerons sur la case avec le maximum et on déplace la coccinelle sur celle-ci.

```
    // Trouver la colonne avec le plus de pucerons parmi les 3 colonnes au-dessus
   int maxPucerons = -1; // Nombre de pucerons de la colonne avec le plus de pucerons
   int maxJ = j; // Colonne avec le plus de pucerons
   for (int k = -1; k <= 1; k++) { // Pour chaque colonne au-dessus
       int newJ = j + k;
       if (newJ >= 0 && newJ < C) { // Si la colonne est dans le tableau
            if (G[i - 1][newJ] > maxPucerons) { // Si la colonne a plus de pucerons que la colonne avec le plus de pucerons (maxPucerons)
               maxPucerons = G[i - 1][newJ]; // La colonne avec le plus de pucerons est la colonne actuelle
               maxJ = newJ;
           }
       }
   }
   // Déplacement
   i--; // Aller sur la ligne au-dessus
   j = maxJ; // Aller sur la colonne avec le plus de pucerons
}
```

Après avoir répété l'opération jusqu'à arriver sur la dernière ligne, on retourne le nombre de pucerons mangés.

```
   // Ajouter les pucerons de la première ligne (ligne 0)
   puceronsManges += G[0][j]; // Mange les pucerons sur la ligne actuelle
   return puceronsManges; }
```

## Exercice 2 :

Cette fonction glouton retourne un tableau (de une ligne et d colonnes) présentant les pucerons mangés durant le trajet pour chaque case (d) de la ligne de départ.

On appelle donc la fonction glouton construite précédemment dans une boucle allant de la première à la dernière case de la colonne. Finalement, on retourne le tableau.

```
/**
* Calcule le nombre de pucerons mangés par une coccinelle
* @param G (int[][]) : la grille de pucerons
* @return (int[]) : le nombre de pucerons qu'une coccinelle mangera sur son chemin glouton
*/
public int[] glouton(int[][] G) {
   int C = G[0].length; // Nombre de colonnes
     int[] Ng = new int[C]; // Tableau pour stocker le nombre de pucerons mangés pour chaque colonne
   for (int d = 0; d < C; d++) { // Pour chaque colonne d
          Ng[d] = glouton(G, d); // Appeler la fonction glouton pour chaque colonne d
   }
   return Ng;
}
```

## Exercice 3 :

Base :

- m(0, d) = G[0][d] => La valeur initiale de m à la première ligne est égale au nombre de pucerons dans la case où la coccinelle a atterri.
- m(0, c) = G[0][c] pour $c \neq d$ : La valeur initiale de m pour les autres cases de la première ligne est égale au nombre de pucerons dans ces cases.

Hérédité :

- m(l, c) = G[l][c] + max{m(l-1, c), m(l-1, c-1), m(l-1, c+1)} avec $1 \leq l < L$, $0 \leq c < C$

## Exercice 4 :

Ici, on souhaite trouver tous les chemins optimaux depuis n'importe quelle case de départ.

On commence par faire les différentes initialisations (nombre de ligne, colonne et la création des tableaux M et A) :

```
/**
* Calcule les tableaux M et A. Le tableau M[0:L][0:C] de terme général
M[l][c] = m(l,c) et le tableau A[0:L][0:C] dont le terme général
A[l][c] = a(l,c) est l'indice de la colonne qui precède la case (l,c)
sur le chemin maximum.
* @param G (int[][]) : la grille de pucerons
* @param d (int) : la colonne de départ
* @return (int[][][]) : un tableau 3D contenant les tableaux M et A
*/
public static int[][][] calculerMA(int[][] G, int d) {
   int L = G.length; // Nombre de lignes
```

```
   int C = G[0].length; // Nombre de colonnes

   int[][] M = new int[L][C]; // Tableau pour stocker les valeurs de
m

   int[][] A = new int[L][C]; // Tableau pour stocker les indices des
colonnes précédentes
```

On commence par calculer le tableau M, un tableau 2D qui va stocker les valeurs de m. Tel que m(l,c) représentant le nombre maximum de pucerons qu'une coccinelle peut manger en partant de la case (l,c) et en suivant un chemin optimal jusqu'à la dernière ligne de la grille G.

Tout d'abord on initialise les valeurs de la dernière ligne (la ligne du bas) :

```
// Pour le tableau M



// Initialiser les valeurs de la dernière ligne (la ligne du bas)

for (int c = 0; c < C; c++) { // Pour chaque colonne c

   if (c == d) { // Si la colonne est la colonne de départ

       M[L - 1][c] = G[L - 1][c]; // Mettre la valeur de la case

   } else { // Si la colonne n'est pas la colonne de départ

         M[L - 1][c] = -1; // Mettre à -1 pour indiquer que la case
n'est pas accessible

   }

}
```

En se basant sur ce qu'on à trouver à la question 3, on peut calculer le maximum des valeurs en suivant l'équation de récurrence.

On vérifie d'abord si la colonne est accessible, et si ça n'est pas le cas on met sa valeur à -1.

```
// Calculer les valeurs de M en suivant l'équation de récurrence
for (int l = L - 2; l >= 0; l--) { // Pour chaque ligne l en partant de l'avant-dernière ligne (en bas)
   for (int c = 0; c < C; c++) { // Pour chaque colonne c
       if (c < d - (L - 1 - l) || c > d + (L - 1 - l)) { // Si la colonne n'est pas accessible
           M[l][c] = -1; // Mettre à -1 pour indiquer que la case n'est pas accessible
       }
```

Sinon on compare les valeurs des cases des colonnes de droite et de gauche et on met à jour la variable correspondant au maximum.

```
   } else { // Si la colonne est accessible
       // Calculer le maximum des valeurs suivantes selon l'équation de récurrence
       int maxSuivant = M[l + 1][c];
       int indiceMax = c; // Indice de la colonne avec le maximum
       if (c > 0 && M[l + 1][c - 1] > maxSuivant) { // Si la colonne à gauche est accessible et a une valeur plus grande que le maximum
           maxSuivant = M[l + 1][c - 1]; // Mettre à jour le maximum
           indiceMax = c - 1; // Mettre à jour l'indice de la colonne avec le maximum
       }
```

```
        if (c < C - 1 && M[l + 1][c + 1] > maxSuivant) { // Si la
colonne à droite est accessible et a une valeur plus grande que le
maximum

            maxSuivant = M[l + 1][c + 1]; // Mettre à jour le maximum

            indiceMax = c + 1; // Mettre à jour l'indice de la colonne
avec le maximum

        }

        // Mettre à jour les valeurs de M

        M[l][c] = G[l][c] + maxSuivant; // Mettre à jour la valeur de
M

    }

}
```

Ensuite pour le tableau A qui vas stocker le chemin de valeur maximum (les cases du chemins de valeurs maximum auront pour valeur 1, les autres auront 0 dans le tableau A), on commence par l'initialiser à la première ligne :

```
// Pour le tableau A



int max = M[0][0]; // Maximum de la première ligne

int indice = 0; // Indice de la colonne avec le maximum dans la première
ligne



// Trouver la colonne avec le maximum dans la première ligne

for (int k = 0; k < C; k++) {

    if (M[0][k] > max) { // Si la colonne a une valeur plus grande que le
maximum
```

```
      max = M[0][k];

      indice = k;

   }

}

A[0][indice] = 1; // Mettre à 1 la case avec le maximum dans la première ligne (la case de départ)
```

Puis on compare parmi les 3 colonnes (nord-ouest, nord et nord-est) celle qui a la valeur maximale. Si c'est le cas, on met à jour le max et l'indice.

```
// Construire le chemin optimal depuis la deuxième ligne jusqu'à la dernière

for (int t = 1; t < L; t++) {

   max = -1;



     // Recherche de la colonne avec le maximum parmi les colonnes adjacentes

   for (int y = -1; y <= 1; y++) { // Pour chaque colonne adjacente

      int newIndice = indice + y; // Indice de la colonne adjacente

       if (newIndice >= 0 && newIndice < C && M[t][newIndice] > max) { // Si la colonne adjacente est dans le tableau et a une valeur plus grande que le maximum

          max = M[t][newIndice]; // Mettre à jour le maximum

             indice = newIndice; // Mettre à jour l'indice de la colonne avec le maximum

      }
```

```
    }

    A[t][indice] = 1; // Mettre à 1 la case avec le maximum dans la ligne t

}
```

Et enfin, on retourne le tableau final composé de A et M.

```
    // Retourner les tableaux M et A dans un tableau 3D

    return new int[][][]{M, A};

}
```

## Exercice 5 :

On s'intéresse  à présent à l'affichage d'un chemin à nombre de pucerons maximum, de la case d'atterrissage (0, d) jusqu'à la case (L – 1, cStar)

```
/**

* Affiche un chemin maximum

* @param A (int[][]) : le tableau A

*/

public void acnpm(int[][] A, int l, int c) {

    if (l >= 0 && c >= 0) { // Si la case existe

        if(A[l][c] == 1){ // Si la case est sur le chemin optimal

            System.out.print("(" + (A.length - 1 - l) + ", " + (c) + ")");
// Afficher la case

        }
```

Tout d'abord si la case précédente existe sur le chemin optimal, on rappelle la procédure sur la case du dessus (appel récursif).

```
if (A[l][c] == 1) { // Si la case est sur le chemin optimal
```

```
   acnpm(A, l - 1, c); // Appel récursif sur la case du dessus

}
```

Sinon si la case précédente est située sur la colonne d'avant alors on rappelle cette méthode sur la colonne de gauche (appelle récursif).

```
else if (c > 0 && A[l][c - 1] == 1) { // Si la case précédente est à
gauche

   acnpm(A, l, c - 1); // Appel récursif sur la case de gauche

}
```

Et enfin si elle est située sur la colonne d'après alors on rappelle cette méthode sur la colonne de droite (appelle récursif).

```
else if (c < A[0].length - 1 && A[l][c + 1] == 1) { // Si la case
précédente est à droite

               acnpm(A, l, c + 1); // Appel récursif sur la case de
droite

       }

   }

}
```

## Exercice 6 :

La méthode optimal prend en paramètre une grille G et une colonne de départ d. Elle utilise la méthode calculerMA pour calculer les tableaux M et A. Ensuite, la méthode optimal trouve la colonne avec le maximum dans la première ligne du tableau M et retourne cette valeur maximale. Cette valeur maximale représente le nombre maximum de pucerons qu'une coccinelle peut manger en partant de la case (0, d) et en suivant un chemin optimal.

```
/**
```

```java
/* Fonction qui retourne le nombre de pucerons qu'une coccinelle ayant atterri sur la case (0, d) mangera sur le chemin a nombre de pucerons maximum
* @param G (int[][]) : la grille de pucerons
* @param d (int) : la colonne de départ
* @return (int) : le nombre de pucerons mangés
*/
public int optimal(int[][] G, int d) {
   int[][][] MA = calculerMA(G, d); // Calculer les tableaux M et A
   int[][] M = MA[0]; // Tableau des valeurs de M
   int[][] A = MA[1]; // Tableau des indices des colonnes précédentes (tableau A)
   // Trouver la colonne avec le maximum dans la première ligne
   int max = M[0][0];
   int indiceMax = 0;
   for (int k = 1; k < M[0].length; k++) { // Pour chaque colonne k
       if (M[0][k] > max) {
           max = M[0][k];
           indiceMax = k;
       }
   }
   return max;
}
```

## Exercice 7 :

La fonction optimal de cet exercice est utilisée pour calculer le nombre maximum de pucerons qu'une coccinelle peut manger en partant de chaque case (0, d) et en suivant un chemin optimal jusqu'à la dernière ligne de la grille G. La méthode prend en entrée une grille G. Pour chaque colonne d de la grille, elle appelle la méthode optimal(G, d) pour calculer le nombre maximum de pucerons qu'une coccinelle peut manger en partant de la case (0, d). Elle stocke ces valeurs dans un tableau Nmax et le retourne. Chaque élément Nmax[d] du tableau Nmax représente le nombre maximum de pucerons qu'une coccinelle peut manger en partant de la case (0, d) et en suivant un chemin optimal.

```
/**
* Fonction qui retourne le tableau Nmax[0 : C] de terme genéral
Nmax[d] = nmax (d), nombre de pucerons que la coccinelle qui a atteri
sur case (0,d) mangera sur le chemin a nombre de pucerons maximum
* @param G (int[][]) : la grille de pucerons
* @return (int[]) : le nombre de pucerons mangés
*/
public int[] optimal(int[][] G) {
   int C = G[0].length; // Nombre de colonnes
    int[] Nmax = new int[C]; // Tableau pour stocker le nombre de
pucerons maximum pour chaque colonne
   for (int d = 0; d < C; d++) { // Pour chaque colonne d
        Nmax[d] = optimal(G, d); // Appeler la fonction optimal pour
```

```
chaque colonne d

   }

   return Nmax;

}
```

## Exercice 8 :

Cette fonction nommée gainRelatif retourne le gain relatif de la stratégie optimale sur la stratégie gloutonne, et ce peu importe la case de départ de la première ligne.

La méthode gainRelatif calcule le gain relatif de la stratégie optimale par rapport à la stratégie gloutonne pour chaque colonne de départ dans la grille de pucerons. Elle prend en entrée deux tableaux d'entiers : Nmax et Ng. Le tableau Nmax contient le nombre maximum de pucerons qu'une coccinelle peut manger en partant de chaque case (0, d) et en suivant un chemin optimal jusqu'à la dernière ligne de la grille. Le tableau Ng contient le nombre de pucerons qu'une coccinelle mangera en partant de chaque case (0, d) et en suivant un chemin glouton. La méthode calcule le gain relatif pour chaque colonne d en utilisant la formule suivante : $gain(d) = \frac{n_{\max}(d) - n_{g(}(d)}{n_g(d)}$ soit :

```
(Nmax[d] - Ng[d]) / Ng[d]
```

Si le nombre de pucerons mangés par la stratégie gloutonne est zéro (c'est-à-dire si Ng[d] est zéro), alors le gain relatif est défini à zéro pour éviter une division par zéro.

La méthode retourne un tableau de float, nommé Gain où chaque élément Gain[d] représente le gain relatif pour la colonne d.

```
/**

* Fonction qui retourne un tableau float Gain[0:C] contenant, pour
toute case (0, d) de depart, le gain relatif de la stratégie optimale
sur la strategie gloutonne.

* @param Nmax (int[]) : le tableau Nmax
```

```
* @param Ng (int[]) : le tableau Ng
* @return (float[]) : le gain relatif
*/
public float[] gainRelatif(int[] Nmax, int[] Ng) {
   int C = Nmax.length; // Nombre de colonnes
     float[] Gain = new float[C]; // Tableau pour stocker le gain relatif pour chaque colonne
   for (int d = 0; d < C; d++) { // Pour chaque colonne d
         if (Ng[d] != 0) { // Si le nombre de pucerons mangés par la stratégie gloutonne est différent de zéro
            Gain[d] = (float) (Nmax[d] - Ng[d]) / Ng[d]; // Calculer le gain relatif
       } else {
            // En cas de division par zéro, définir le gain relatif à zéro
           Gain[d] = 0;
       }
   }
   return Gain;
}
```

## Exercice 9 :

La méthode permutationAleatoire est utilisée pour générer une permutation aléatoire des éléments d'un tableau d'entiers T. Elle prend en paramètre un tableau d'entiers T et retourne ce même tableau après avoir permuté aléatoirement ses éléments. Voici

comment elle fonctionne :

1. Elle récupère la longueur du tableau T et la stocke dans la variable n.
2. Elle crée une instance de la classe Random de Java pour générer des nombres aléatoires.
3. Elle parcourt le tableau T de la fin vers le début. Pour chaque position i dans le tableau :
    - Elle génère un nombre entier aléatoire r entre 0 (inclus) et i (exclu).
    - Elle permute l'élément à la position r avec l'élément à la position i-1 en utilisant la méthode permuter(T,r,i-1).
4. Une fois tous les éléments du tableau T permutés de manière aléatoire, elle retourne le tableau T.

```
/**
* fonction qui calcule une permutation aleatoire des valeurs d'un tableau.
* @param T (int[]) : le tableau
* @return (int[]) : le tableau avec les valeurs permutées
*/
static int[] permutationAleatoire(int[] T){ int n = T.length;
// Calcule dans T une permutation aléatoire de T et retourne T
   Random rand = new Random(); // bibliothèque java.util.Random
   for (int i = n; i > 0; i--){
       int r = rand.nextInt(i); // r est au hasard dans [0:i]
       permuter(T,r,i-1);
   }
   return T;
}
```

La méthode permuter est une méthode statique qui permute les valeurs de deux cases prises en paramètres d'un tableau d'entiers T.

```java
/**
* Fonction qui permute les valeurs de deux cases d'un tableau
* @param T (int[]) : le tableau
* @param i (int) : l'indice de la première case
* @param j (int) : l'indice de la deuxième case
*/
static void permuter(int[] T, int i, int j){
   int ti = T[i];
   T[i] = T[j];
   T[j] = ti;
}
```

## Annexe :

Voici le code complet, y compris la fonction main qui est automatiquement appelée lors de l'exécution de la commande : `java Projet`. Cette fonction permet l'exécution et l'affichage des résultats des différentes méthodes de la classe Projet :

Github :

Lien du fichier : https://github.com/Augustin-Br/Projet-IGI/blob/master/src/Projet.java

Raw : https://raw.githubusercontent.com/Augustin-Br/Projet-IGI/master/src/Projet.java

Sur mon espace personnel ESIEE :

https://perso.esiee.fr/~brennera/Projet-IGI/Code/src/

Voici ce qu'affiche notre programme :

```
PS C:\Users\augbr\Desktop\Projet-IGI\src> java Projet
Grille G:
G[2] : [1, 1, 10, 1, 1]
G[1] : [6, 5, 1, 2, 8]
G[0] : [2, 2, 3, 4, 2]
Valeurs des chemins gloutons depuis les cases (0, d) : Ng = [9, 9, 18, 13,
11]

Programmation dynamique, case de départ, (0, 0)
Grille M :
M[2] : [9, 9, 17, -1, -1]
M[1] : [8, 7, -1, -1, -1]
M[0] : [2, -1, -1, -1, -1]
un chemin maximum : (0, 0)(1, 1)(2, 2) Valeur : 17

Programmation dynamique, case de départ, (0, 1)
Grille M :
M[2] : [9, 9, 17, 4, -1]
M[1] : [8, 7, 3, -1, -1]
M[0] : [-1, 2, -1, -1, -1]
un chemin maximum : (0, 1)(1, 1)(2, 2) Valeur : 17

un chemin maximum : (0, 0)(1, 1)(2, 2) Valeur : 17
un chemin maximum : (0, 1)(1, 1)(2, 2) Valeur : 17
un chemin maximum : (0, 2)(1, 1)(2, 2) Valeur : 18
un chemin maximum : (0, 3)(1, 3)(2, 2) Valeur : 16
un chemin maximum : (0, 4)(1, 3)(2, 2) Valeur : 14

Ng = [9, 9, 18, 13, 11]
Nmax = [17, 17, 18, 16, 14]
Gain relatifs = [0.8888889, 0.8888889, 0.0, 0.23076923, 0.27272728]
```

```
TAB. 1 (Exercice 8):
Ng = [267, 113, 112, 118, 142]
Gain relatifs = [0.0, 1.4247788, 1.4375, 1.3644068, 0.8450704]

VALIDATION STATISTIQUE

nruns = 10000
L au hasard dans [ 5, 16 ]
C au hasard dans [ 5, 16 ]

run 1/10000, (L,C) = (8,10)
run 2/10000, (L,C) = (7,8)
run 3/10000, (L,C) = (13,6)
...
...
...
run 9999/10000, (L,C) = (5,15)
run 10000/10000, (L,C) = (10,6)
GAINS.length = 100029, min = 0.0, max = 0.8936170339584351, mean =
0.04402928797055766, med = 0.03257589787244797
```