Nino COILLARD (G10), Alexis ROLLAND (G11)

Projet du cours d'algorithmique

Question 1 :

D'après l'énoncé, on a la fonction **glouton(int[][] G, int d)** qui retourne le nombre de pucerons qu'une coccinelle atterrit en case (0, d) mangera sur son chemin glouton.

Ici, l'initialisation est ; nbM(0,d) = G(0,d).

Ici, l'équation d'hérédité est ; nbM(k,d), $1 \leq k < n$, $0 \leq p < c$.
nbM(k, d) = nbM(k-1,d) + max(G(k − 1, p+1), G(k − 1,p+1),G(k-1,p))

pour obtenir cette équation d'hérédité il faut juste réfléchir à comment appliquer la consigne, ici on cherche a trouver le chemin le plus long mais de manière naïve, on vas donc, pour trouver le nombre de moucheron maximal sur la case suivant il faut seulement additionner le nombre de moucherons calculé précédemment et prendre la case maximal sur les 3 case au dessus.

On obtient alors la fonction glouton(int[][] G, int d) suivante :

```
public static int glouton(int[][] G, int d){int n = G.length; int
c = G[0].length;
      int nbM = G[0][d];
      int p = d;
         for(int i = 1 ; i < n ; i++){
            int val = G[i][p];
            if(p+1<c){
                  val = max(val,G[i][p+1]);
                  }
                  if(p-1>=0){
                        val = max(val,G[i][p-1]);
                  }
                  if(p-1>=0 && val == G[i][p-1]){
                      p-=1;
                  }else if (p+1<c && val == G[i][p+1]){
                      p+=1;
                  }
            nbM = nbM+val;
         }
         return nbM;
}
```

Nino COILLARD (G10), Alexis ROLLAND (G11)

Question 2 :

En utilisant le code de la question précédente et d'après la consigne
On a **glouton(int[][] G)** qui retourne un tableau de terme général Ng[d] = glouton(G,d) :

Ici, l'équation d'hérédité est ; N(k), 0 ≤ k < d. avec d la taille du tableau G[0]
N(k) = glouton(G,k);

```
public int[] glouton(int[][] G){int d = G[0].length;
      int[] N = new int[d];
      for(int i = 0 ; i < d ; i++){
            N[i] = glouton(G,i);
      }
      return N;
}
```

Question 3 :

D'après l'énoncé du sujet, on obtient l'**équation de récurrence** des valeurs m(l,c) :

Ici, l'initialisation est ; m(0,d) = G(0,d); le nombre de pucerons sur la case 0, d. Pour tout c tel que 0≤c<C et c!=d on a m(0,c)=-1

Ici, l'équation d'hérédité est ; m(l,c), 1 ≤ l < L, 0 ≤ c < C.
m(l,c) = G(l,c) + max(m(l−1, c), m(l−1,c+1),m(l-1,c)).

Pour obtenir cette équation c'est simple, il faut réfléchir à l'envers, en partant du résultat pour descendre au début, on a alors le nombre de pucerons consommé jusqu'à la case actuelle est égale aux nombre de pucerons sur la case actuelle plus le max entre le nombre de pucerons consommé dans les trois case précédentes (droite gauche et milieu) l'équation devient donc très simple à déterminer.

Question 4 :

On calcule les 2 tableaux **M[0:L][0:C]** et **A[0:L][0:C]**, dont le terme général est l'indice de la colonne qui précède la case (l,c), sur le chemin maximum.
En utilisant l'équation de récurrence de la question précédente ;

```
public static int[][][] calculerMA(int[][] G, int d){ int n =
G.length; int c = G[0].length;
      int[][] M = new int[n][c];
      int[][] A = new int[n][c];
      //base :
      for(int i = 0 ; i < c ; i++){
            if(i==d){M[0][i] = G[0][i]; A[0][i] = 0;}
            else {M[0][i] = -1; A[0][i] = -1;}
      }
      //hérédité ;
      for(int i = 1 ; i<n ; i++){
             for(int j = 0 ; j < c ; j++){
                 int val = M[i-1][j];
                 if(j+1<c){
                           val = max(val,M[i-1][j+1]);
                 }
                 if(j-1>=0){
                           val = max(val,M[i-1][j-1]);
                 }
                 if(val == -1){
                       M[i][j] = -1;
                       A[i][j] = -1;
                 }else{
                       M[i][j] = val + G[i][j];
                       if(j-1>=0 && val == M[i-1][j-1]){
                             A[i][j] = j-1;
                       }else if (j+1<c && val == M[i-1][j+1]){
                             A[i][j] = j+1;
                       }else{
                             A[i][j] = j;
                       }
                       }
            }

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

Question 5 :

On cherche à afficher un **chemin à nombre de pucerons maximum**, de la case d'atterrissage (0,d) jusqu'à la case (L-1,cStar) :

```
static void acnpm(int[][] M, int[][] A){ int L = M.length;
      int cStar = argMax(M[L-1]) ; // colonne d'arrivée du chemin
      //max. d'origine (0,d)
      acnpm(A, L-1, cStar); // affichage du chemin maximum de (0,d)
      //a (L-1, cStar)
      System.out.print(" Valeur : " + valMax(M[L-1]) + "\n");
}

static void acnpm(int[][] A, int l, int c){
      if(l == 0){
            System.out.printf("un chemin maximum : (%d,%d)",0,c);
            return;
      }
      acnpm(A,l-1,A[l][c]);
      System.out.printf("(%d,%d)",l,c);
}
```

avec :

```
  static int argMax(int[] table){
        int valMax = table[0];
        int index = 0;
        for(int i = 1 ; i < table.length ; i++){
            if(valMax<table[i]){
                valMax = table[i];
                index = i;
            }

        }
        return index;
    }

   static int valMax(int[] table){
        int valMax = table[0];
        for(int i = 1 ; i < table.length ; i++){
            if(valMax<table[i]){
                valMax = table[i];
            }
        }
        return valMax;
    }
```

Question 6 :

On a **optimal(int[][] G, int d)** qui retourne le nombre de pucerons qu'une coccinelle ayant atterri sur la case (0,d) mangera sur le chemin à nombre de pucerons maximum :

```
static int optimal(int[][] G, int d){int[][] M =
calculerMA(G,d)[0];
        return valMax(M[M.length-1]); //Cherche la valeur max du
//tableau sur la dernière ligne
}
```

Question 7 :

On a **optimal(int[][] G)** qui retourne le tableau de terme général Nmax[d], le tableau des valeurs maximum obtenus après que la coccinelle ait atterri sur les différentes cases de départ.

```
static int[] optimal(int[][] G){int C = G[0].length;
        int[] N = new int[C];

        for(int i = 0 ; i<C ; i++){
            N[i] =optimal(G,i);
        }
        return N;
}
```

Question 8 :

On a **gainRelatif(int[] Nmax, int[] Ng)** qui retourne un tableau float de terme général gain(d)=(nmax(d)-ng(d))/ng(d) :

Ici, l'équation d'hérédité est ; gain(d), 0 ≤ d < D,
gain(d) = (Nmax(d)-Ng(d))/(Ng(d))

```
static float[] gainRelatif(int[] Nmax, int[] Ng){int D
=Nmax.length;
        float[] gain = new float[D];
        for(int d = 0; d<D ; d++){
            gain[d] = ((float)(Nmax[d]-Ng[d]))/((float)Ng[d]);
        }
        return gain;
}
```

Avec le tableau donné on obtient donc un tableau de gain équivalent à ;
[0.0, 1.4159292, 1.4285715, 1.3559322, 0.8380282]

Question 9 :

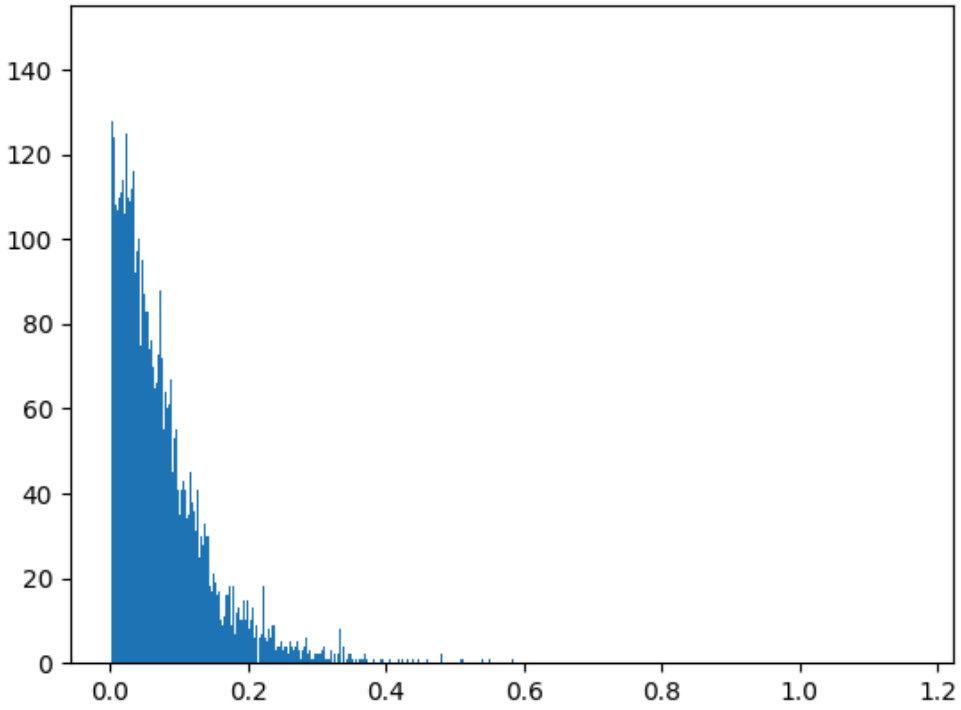
Ici on va utiliser des tableaux à taille aléatoire comprise entre 5 et 16 dont leurs valeurs sont elles même des permutation aléatoires d'un tableau dont les valeurs sont aléatoires entre 0 et LxC.

Tout d'abord, nous allons faire cela dans le main de la classe pour le faire automatiquement sans utiliser une fonction.

Pour obtenir 10000 résultats statistique on vas donc devoir utiliser une boucle for, puis l'on vas calculer les gain relatifs de tous les tableaux aléatoires pour après les mettre dans une chaîne de caractères, ses ret enfin les enregistrer dans un fichier CSV (en utiliser la classe PrintWriter de java) afin d'analyser tous les résultats et les représenter dans un graphique (le code pour obtenir ce dernier est donné).

```
int n = 10000;
String listeDesGains="";
for(int i = 0 ; i < n ; i++){
      G = new
int[(int)(Math.random()*11)+5][(int)(Math.random()*11)+5];
            int L = G.length;
            int C = G[0].length;
            for(int v = 0 ; v < L ; v++){
                for(int j = 0 ; j < C ; j++){
                          G[v][j] = (int)(Math.random()*(L*C));
                }
                G[v] =permutationAleatoire(G[v]);
      }
      gain = gainRelatif(optimal(G),glouton(G));
            for(float value : gain)
                      listeDesGains += value + "\n";
         }
try (PrintWriter writer = new PrintWriter(new
File("gainsRelatifs_nruns="+n+"_Linf=5_Lsup=16_Cinf=5_Csup=16.csv"
))) {
            writer.write(listeDesGains);
            writer.close();
} catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
}
```

On a alors, après avoir exécuté le code .py :



Conclusion :

En conclusion, lors de ce projet d'algorithmique nous avons appris à utiliser les notions apprises en cours pour résoudre un problème concret, ici, trouver le nombre de moucherons maximum que pouvait manger la coccinelle. On a aussi pu observer la différence entre un programme "mal" pensé et un programme réfléchi par, justement, le graphique ci dessus. Enfin, l'utilisation des validations statistiques peuvent être utiles même en dehors d'un projet de ce type et permet donc d'apprendre à optimiser son code intelligemment.