BERNARD Vincent Groupe 3 et BONNEAULT Timothé :

Rapport projet d'Algorithmique

Introduction

Le projet d'algorithmique consistait à rechercher les solutions de cinq problèmes en identifiant d'une part les solutions optimales et d'autre part les approches gloutonnes possibles. Une fois ce travail effectué, la comparaison des deux approches était évaluée en mesurant l'écart relatif entre l'approche optimale et l'approche gloutonne sur une population de 5000 échantillons. Pour cette comparaison, le projet demandait de fournir la moyenne, l'écart type et l'histogramme de l'écart relatif.

Chaque exercice possède sa propre problématique. A chaque fois, l'identification de l'algorithme glouton et de l'algorithme optimal sont différentes.

Pour des raisons de simplicité, la création des histogrammes a été faite en Java en utilisant des librairies disponibles sur internet.

Exercice 1 Robot :

Dans cet exercice, la partie optimale a été traitée en td. Nous devions par conséquent uniquement faire la partie gloutonne ainsi que comparer ces deux méthodes. Nous devions juste faire l'algorithme glouton et changer l'attribution des coûts de déplacement par des valeurs aléatoires.

Dans un premier temps nous avons changé le coût de déplacement. Pour ce faire, nous avons créé trois tableaux à 2 dimensions qui correspondent au coût Nord, Est et Nord-Est. Nous affectons à chaque case un coût de déplacement aléatoire. Voici la méthode utilisée :

```
int[][] coutN = new int[L][C];
int[][] coutNE = new int[L][C];
int[][] coutE = new int[L][C];

for(int i = 0; i < L; i++) {
    for(int j = 0; j< C; j++) {
        coutN[i][j]= vRandom.nextInt(100);
        coutNE[i][j]= vRandom.nextInt(100);
        coutE[i][j]= vRandom.nextInt(100);
    }
}
```

Ainsi, à chaque fois que nous appelons les fonctions `calculerM`, `accm` et la fonction `glouton`, expliquée ci-dessous, nous mettons en paramètre de ces fonctions/procédures, les tableaux de coût de déplacement. Les fonctions, n, ne et e ont été modifiées de la manière suivante pour s'adapter à la nouvelle situation avec les coûts de déplacement aléatoire :

```
static int n(int l, int c, int L, int C, int[][] coutN){
    if (l==L-1) return Integer.MAX_VALUE;
    return coutN[l][c];
}
static int ne(int l, int c, int L, int C, int[][] coutNE){
    if (l == L-1 || c == C-1) return Integer.MAX_VALUE;
    return coutNE[l][c];
}
static int e(int l, int c, int L, int C, int[][] coutE){
    if (c == C-1) return Integer.MAX_VALUE;
    return coutE[l][c];
```

Nous avons donc créé la fonction `glouton` qui prend en paramètre L et C qui sont les dimensions du tableau en ligne et colonne et qui renvoie comme entier la valeur du chemin glouton.

Pour cela, nous déclarons dans un premier temps plusieurs variables que nous déclarons toutes à 0. Elles prendront respectivement le prix pour aller au nord-est : m1, le prix pour aller à l'est : m2, puis le prix pour aller au nord : m3. Le minimum de ces 3 prix est affecté à une variable : minimum, puis on ajoute ce coût dans la sommeGloutonne, déplacement après déplacement. On a également 2 variables c et l qui parcourent le tableau en ligne et en colonne.

Nous avons donc fait une boucle while qui s'arrête uniquement lorsque le robot à atteint la dernière case du tableau. On utilise les conditions suivantes :

```
(c != C-1 || l != L-1)
```

Comme expliqué précédemment, on affecte aux variables m1, m2 et m3 le coût de déplacement dans les différentes directions en fonction de l'endroit où se trouve le robot :

```
m1 = ne(l, c, L, C, coutNE);
m2 = e(l, c, L, C, coutE);
m3 =  n(l, c, L, C, coutN);
minimum = min(m1,m2,m3);
```

Ensuite, si le coût minimum est le coût pour aller au nord-est, on ajoute 1 aux variables l et c pour se déplacer dans la diagonale et on ajoute le coût de déplacement `coutNE(l,c)` dans la variable sommeGloutonne. Dans le cas où le coût minimum est celui pour aller à l'est on ajoute 1 uniquement à la variable c pour se déplacer dans la colonne suivante et on ajoute aussi le coût de déplacement `coutE(l,c)` à la variable sommeGloutonne. A l'inverse, si le coût de déplacement `coutN(l,c)` est minimum dans la direction du nord, on ajoute cette fois-ci uniquement 1 à la variable l pour aller à la ligne suivante. Ce n'est pas précisé ici mais si les différents coûts de déplacement sont égaux, le déplacement dans la diagonale est privilégié afin de gagner un déplacement.

Enfin, lorsque la dernière case du tableau est atteinte, on sort de la boucle et la fonction renvoie la somme gloutonne totale.

```
static int glouton(int L, int C, int[][] coutN, int[][] coutNE, int[][] coutE) {
    //int[][]M = new int [L][C];//toutes les vals sont à 0
    int m1 = 0;//ne(l, c, L, C);
    int m2 = 0;//e(l,c,L,C);
    int m3 = 0;//n(l,c,L,C);
    int minimum = 0;
    int sommeGloutonne = 0;
    int c = 0;
    int l = 0;
    //System.out.printf("(%d,%d)",l,c);
    while (c != C-1 || l != L-1) {
        m1 = ne(l, c, L, C, coutNE);
        m2 = e(l, c, L, C, coutE);
        m3 =  n(l, c, L, C, coutN);
        minimum = min(m1,m2,m3);

        if (m1 == minimum) {
            sommeGloutonne += minimum;
            l++;
            c++;
            //System.out.printf("-> %d, (%d,%d)",minimum,l,c);
        }
        else if (m2 == minimum) {
            sommeGloutonne += minimum;
            c++;
            //System.out.printf("-> %d, (%d,%d)",minimum,l,c);
        }
        else {
            sommeGloutonne += minimum;
            l++;
            //System.out.printf("-> %d, (%d,%d)",minimum,l,c);
        }
    }
    System.out.println();
    return sommeGloutonne;
}
```

Pour calculer la distance relative entre ces deux méthodes sur 5000 runs, nous avons créé une fonction nommée `init()` qui renvoie comme float la distance relative entre la méthode optimale et la

méthode gloutonne dans laquelle on génère 2 valeurs aléatoires qui nous donne le nombre de lignes et de colonnes du tableau. On appelle ensuite les méthodes optimale et gloutonne afin de les comparer à l'aide de la formule suivante :

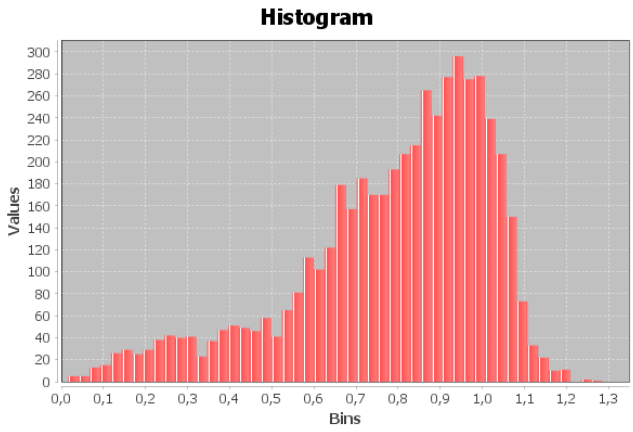
```
return (float)Math.abs(sommeOptimale - sommeGloutonne)/sommeOptimale;
```

Par la suite, on crée une méthode `evalStat()` dans laquelle on initialise un tableau de float dans lequel seront stockées toutes les distances relatives des 5000 runs. Pour cela, on fait une boucle qui affecte, à notre tableau de distances, la valeur renvoyée par la fonction `init()`.

Enfin, on appelle une fonction qui s'appelle `sortMoyMedEcart(vDistance)` qui prend en paramètre le tableau et qui affiche la moyenne, la médiane et l'écart Type.

```
sommeOptimale : 1, sommeGloutonne : 485distance relative : 484,000000
moyenne : 498,353607, mediane : 496,000000, EcartType : 288,070343
```

On peut voir ci-dessus un exemple de valeur optimale. Dans cet exercice, toutes les valeurs optimales valent 1 d'après la définition des fonctions n, e et en. Ci-dessous, on voit la valeur moyenne, la médiane et l'écart type sur les 5000 runs. Voici l'histogramme :



Dans le main on appelle `evalStat()` pour obtenir cela.

Exercice 2 Sac Max :

Dans cet exercice, la partie optimale a été traitée en cours. Nous devions par conséquent uniquement faire la partie gloutonne. Il y avait ici 2 manières de réaliser la méthode gloutonne : par densité et par valeur. Après, il fallait comparer ces différentes méthodes afin de déterminer la plus efficace.

Il y a plusieurs manières de réaliser les fonctions `gloutonVal` et `gloutonDens`. La première consiste à réutiliser l'un des algorithmes de tri vu en cours, trier par ordre décroissant les valeurs du tableau tout en déplaçant les tailles des objets pour qu'il se situent au même indice que leur valeur (pour `gloutonVal`). Puis, refaire la même chose mais cette fois trier par ratio/densité, rapport entre valeur de l'objet et sa taille.
La seconde consiste à utiliser une autre méthode disponible directement dans le langage JAVA mais que l'on n'a pas vu en cours. C'est pour cette raison que nous avons décidé de l'adopter, en utilisant des objets afin d'expérimenter de nouvelles solutions et méthodes.

Nous avons créé la classe MatchTailleVal dans laquelle nous avons importé les Lists. Nous avons implémenté ensuite l'interface Comparable<MatchTailleVal>.

```
1 package exo2_sacMax;
2⊕import java.util.List;
6
7 public class MatchTailleVal implements Comparable<MatchTailleVal>{
8⊖    public static void main(String[] args) {
```

Dans cette classe, il y a 2 attributs. L'un qui correspond à la taille de l'objet et l'autre correspond à sa valeur. Cette classe contient un constructeur qui permet de donner les différentes valeurs à ces attributs. Nous avons créé 3 getters, l'un pour la valeur, l'autre pour la taille et le dernier pour le ratio. Cela afin de réaliser la méthode gloutonne par ratio qui va être expliquée plus loin. Il y a également une méthode héritée `compareTo` qui compare les valeurs.

```
   private int aTaille = 0;
   private int aValeur = 0;

     // other getters and setters omitted

   public MatchTailleVal(int pTaille, int pValeur) {
       this.setTaille(pTaille);
       this.setValeur(pValeur);
   }

   public int getValeur() {
       return this.aValeur;
   }

   public int getTaille() {
       return this.aTaille;
   }

   public float getRatio() {
       return (float)this.aValeur/this.aTaille;
   }

   public void setValeur(int pValeur) {
       this.aValeur = pValeur;
   }

   public void setTaille(int pTaille) {
       this.aTaille = pTaille;
   }

   @Override public int compareTo(MatchTailleVal object) {
       if (this.getValeur() == 0 || object.getValeur() == 0) {
           return 0;
         }
       return this.getValeur() - object.getValeur();
     }
```

Méthode glouton par valeur :

On avait à disposition 2 tableaux, l'un qui est la taille de chaque objet et l'autre qui est la valeur des différents objets. Nous avons donc créé une fonction `gloutonVal` qui prend en paramètre ces 2 tableaux. La fonction prend également en paramètre la taille du sac et renvoie un entier qui est la valeur maximale que l'on peut mettre dans le sac via cette méthode.

```
static int gloutonVal(int[]T, int []V, int tailleSac) {
```

Nous avons donc créé une variable entière qui est la valeur maximale qu'on peut avoir en remplissant le sac avec cette méthode. Puis nous avons créé une liste de type MatchTailleVal à l'aide de la commande suivante :

```
int valeurMax =0;
List<MatchTailleVal> tableau = new ArrayList<MatchTailleVal>();
```

Nous avons ensuite fait une boucle qui permet de relier la taille des objets avec leur valeurs et les ajouter dans la liste.

```
for(int i = 0; i < T.length; i++) {
    MatchTailleVal objetSac = new MatchTailleVal(T[i],V[i]);
    tableau.add(objetSac);
}
```

Une fois cela réalisé, il ne reste plus qu'à trier cette liste par valeurs, ce que l'on peut faire par l'intermédiaire de cette ligne de commande :

```
//Collections.sort(tableau);
tableau.sort(Comparator.comparing(MatchTailleVal::getValeur));
```

Cette ligne trie la liste du plus petit au plus grand, par conséquent lorsque l'on parcourt la liste pour mettre, si cela est possible, les objets dans le sac, il faut partir des indices les plus élevés. On met successivement les objets de plus grandes valeurs dans le sac, si possible jusqu'à ce que l'on ne puisse plus rentrer d'objets. Pour savoir quelle taille il reste dans le sac, à chaque fois que l'on ajoute un objet on déduit sa taille à la taille du sac afin de connaître la taille restante dans le sac. Et on ajoute successivement la valeur de cet objet à la variable valeurMax qui nous permettra à la fin de renvoyer la valeur totale du sac à la fin de la boucle.

```
for(int i = 0; i < T.length; i++) {
    if(tailleSac - tableau.get(T.length - 1-i).getTaille() > 0) {
        tailleSac -= tableau.get(T.length - 1-i).getTaille();
        valeurMax += tableau.get(T.length - 1-i).getValeur();
        //System.out.printf("gloutonVal : tailleSac : %d, valeurMax : %d\n",tailleSac,valeurMax);
    }
}
System.out.printf("resultat gloutonVal: tailleSacRestant : %d, valeurMax : %d\n",tailleSac,valeurMax);
return valeurMax;
```

Méthode glouton par densité :

Comme tout à l'heure, cette fonction renvoie un entier qui est la valeur maximale avec laquelle on a rempli le sac. Elle reçoit également les tableaux T et V en paramètre ainsi que la taille du sac. On crée également une liste de MatchTailleVal que l'on remplit avec les tailles et valeurs du sac. Cette fois-ci en revanche, on trie le tableau par ratio à l'aide de la ligne suivante :

```
tableau.sort(Comparator.comparing(MatchTailleVal::getRatio));
```

De nouveau, ce ratio est dans l'ordre croissant or ce qui nous intéresse est dans l'ordre décroissant. Par conséquent, on parcourt le tableau dans le sens inverse. A chaque fois que l'on essaie d'ajouter un objet dans le sac, on vérifie qu'il suisse rentrer selon sa taille. Si c'est le cas, on retire sa taille à la taille restante du sac et bien sûr on ajoute sa valeur à la valeur mise dans le sac de la manière suivante :

```
for(int i = 0; i < T.length; i++) {
    if(tailleSac - tableau.get(T.length - 1-i).getTaille() > 0) {
        tailleSac -= tableau.get(T.length - 1-i).getTaille();
        valeurMax += tableau.get(T.length - 1-i).getValeur();
        //System.out.printf("gloutonDens: tailleSac : %d, valeurMax :
    }
}
System.out.printf("resultat gloutonDens: tailleSacRest : %d, valeurMax
return valeurMax;
```

Evaluation statistique :

Une fois ces 2 méthodes réalisées, il faut comparer les différentes méthodes. Pour cela, nous avons créé une fonction `calculerRelative` qui renvoie un tableau de float. Dans cette fonction on initialise dans un 1er temps les valeurs des différentes cases des tableaux T et V en appelant les fonctions `initTabV` et `initTabT` et on affecte à ces cases des valeurs aléatoires. On a également une variable à laquelle on affecte une valeur random/aléatoire, qui est la taille du sac. Ensuite, on appelle la fonction `calculerM` qui prend en paramètre les tableaux T et V ainsi que la taille globale du sac. On détermine la valeur optimale en prenant la dernière case de ce tableau à 2 dimensions. Puis on affecte à deux autres variables les valeurs maximales des sacs avec la méthode gloutonne par valeur et la méthode gloutonne par densité. Ensuite, on vérifie que la valeur optimale ne soit pas nulle pour s'assurer que la division soit possible. Si c'est le cas, on applique la formule de la densité relative

entre la méthode optimale et la méthode gloutonne par valeur, on met le résultat dans le 1er indice du tableau de float. Puis, on effectue ce même calcul mais cette fois-ci, avec la méthode gloutonne par densité que l'on met au second indice du tableau de float. Enfin, on retourne ce tableau.

```
static float[] calculerRelative(int vNbCases) {
    float[]D = new float[2];
    Random vRandom = new Random();

    int[] T = initTabT(vNbCases,100);
    int[] V = initTabV(vNbCases,100);

    int n = V.length;
    //int tailleSac = 18;
    int tailleSac = vRandom.nextInt(100);;

    int[][] M = calculerM(V,T,tailleSac);
    int valOptimale = M[V.length][tailleSac];
    int maxGloutVal = gloutonVal(T,V,tailleSac);
    int maxGloutDens = gloutonDens(T,V,tailleSac);
    //afficher(M);

    //System.out.println("V = " + Arrays.toString(V));
    //System.out.println("T = " + Arrays.toString(T));
    //System.out.printf("M[%d][%d] = %d\n",V.length, C,M[V.length][C]);

    if (valOptimale != 0) {
    //System.out.printf("valOptimale = %d, maxGloutVal = %d, maxGloutDens = %d ",val
        D[0] = (float)(valOptimale - maxGloutVal)/valOptimale;
        D[1] = (float)(valOptimale - maxGloutDens)/valOptimale;
    }
    else {
        D[0]=0;
        D[1]=0;
    }
    return D;
}
```

La fonction `calculerRelative` est elle-même appelée par la fonction `evalDistRelative` dans une boucle de 5000 tours. Au fur et à mesure de ces tours, on affecte les valeurs calculées précédemment dans un tableau à 2 dimensions dont le nombre de colonnes correspond au nombre de tours et dont le nombre de lignes correspond aux 2 différents calculs de la distance relative. La fonction `evalDistRelative` renvoie donc un tableau à 2 dimensions.

```
static float[][] evalDistRelative(int pNbRuns) {
    Random vRandom = new Random();
     float [] vDistanceRes = new float[2];
     //float [] vDistanceDens = new float [pNbRuns];
     float[][] R = new float[2][pNbRuns];
     int vNbVals = 0;
        for (int i=0; i<pNbRuns; i++){
            vNbVals = vRandom.nextInt(2,1000);
            vDistanceRes = calculerRelative(vNbVals);
            R[0][i] = vDistanceRes[0];
            R[1][i] = vDistanceRes[1];

            //System.out.printf("distance relative Valeur: %f\n",vDistanceVal[i]);
            //System.out.printf("distance relative Densité: %f\n",vDistanceVal[i]);
        }

    return R;
}
```

Cette fonction est elle-même appelée dans la méthode `evalStat` avec les moyennes, les médianes et les écart-types. Cette fonction est appelée dans le main.

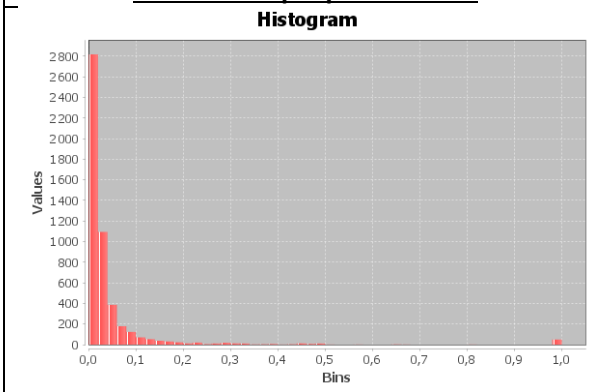
```java
static void evalStat(int pNbRuns) {
    float[] DistDens = new float[pNbRuns];
    float[] DistVal = new float[pNbRuns];
    Arrays.sort(DistDens);
    Arrays.sort(DistVal);
    float moyDistVal = 0;
    float moyDistDens = 0;
    float medianeVal = 0;
    float medianeDens = 0;
    float varianceVal = 0;
    float varianceDens = 0;
    float ecartTypeVal = 0;
    float ecartTypeDens = 0;
    float[][] tabRelatives = evalDistRelative(pNbRuns);
    for(int j =0; j < pNbRuns; j++) {
        DistVal[j] = tabRelatives[0][j];//evalDistRelative(pNbRuns)[0][j];
        DistDens[j] = tabRelatives[1][j];//evalDistRelative(pNbRuns)[1][j];
    }
    for(int i = 0; i < pNbRuns; i++) {
        moyDistVal += DistVal[i];
        moyDistDens += DistDens[i];
    }
    moyDistVal = moyDistVal/pNbRuns;
    moyDistDens = moyDistDens/pNbRuns;

    for(int i=0; i< pNbRuns;i++) {
        varianceVal += (DistVal[i] - moyDistVal)*(DistVal[i] - moyDistVal);
        varianceDens += (DistDens[i] - moyDistDens)*(DistDens[i] - moyDistDens);
    }
    ecartTypeDens = (float) Math.sqrt(varianceDens);
    ecartTypeVal = (float) Math.sqrt(varianceVal);

    if(pNbRuns%2 == 0) {
        medianeDens = (DistDens[pNbRuns/2] + DistDens[pNbRuns/2 +1])/2;
        medianeVal = (DistVal[pNbRuns/2] + DistVal[pNbRuns/2 +1])/2;
    }
    else {
        medianeDens = DistDens[pNbRuns/2];
        medianeVal = DistVal[pNbRuns/2];
    }
    System.out.printf("moyDistVal : %f, medianeVal : %f, ecartTypeVal : %f\n", moyDi
    System.out.printf("moyDistDens : %f, medianeDens : %f, ecartTypeDens : %f\n", mo
}
```

L'affichage est le suivant :

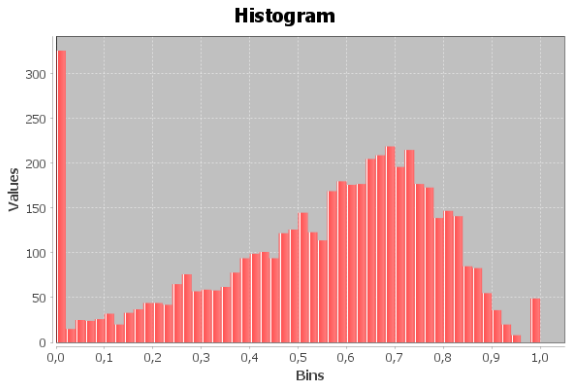
```
moyDistVal : 0,538349, medianeVal : 0,269140, ecartTypeVal : 2,605015
moyDistDens : 0,042153, medianeDens : 0,014134, ecartTypeDens : 1,126891
```

On peut conclure que la méthode gloutonne par densité est plus efficace que la méthode gloutonne par valeur car la distance relative avec la méthode de la densité est nettement inférieure. Par conséquent, elle se rapproche plus de la méthode optimale.

Eval statistique par densité :



Eval statistique par valeur :



Exercice 3 répartition de stocks :

Dans cet exercice, la version optimale avait été effectuée en td. Nous devions juste faire la version gloutonne.

Nous avons donc créé une fonction `glouton` qui prend en paramètre un tableau à 2 dimensions qui donne les différents gains obtenus en fonction du nombre de stocks déposés dans les différents

entrepôts. Cette fonction renvoie un tableau des stocks déposés dans les différents entrepôts par la méthode gloutonne.

On définit dans un premier temps la variable n qui est le nombre d'entrepôts et la variable entrepotsMax, le nombre de stocks. On définit 2 tableaux. L'un qui est le nombre de stocks déposés par entrepôts et l'autre est le gain effectué dans chacun des entrepôts.

On a également défini la variable diffMax qui regarde dans quel entrepôt la différence de gain est maximale si l'on dépose ou non un stock en plus. Et une variable indiceDiffMax qui permet de retenir le nombre de stocks déposés dans chacun des entrepôts.

Nous avons donc fait une boucle while qui parcourt le nombre de stocks, on initialise/réinitialise les variables diffMax et indiceDiffMax à 0. Dans cette boucle, il y a une seconde boucle for qui parcourt tous les entrepôts. Dans un premier temps, elle vérifie que si diffMax est inférieur au gain on puisse vendre 1 stock de plus à tel ou tel entrepôt. Si c'est le cas, cela signifie qu'il ne s'agit pas du gain maximal que l'on peut effectuer, alors on remplace diffMax par cette même valeur. On définit donc qu'il s'agit de l'entrepôt i dans lequel on va déposer un stock de plus. Une fois sorti de cette boucle for, on ajoute 1 à l'entrepôt d'indice diffMax car c'est à lui que l'on va vendre le stock en train d'être traité. Puis, on passe au stock suivant et ainsi de suite jusqu'à ce que tous les stocks soient vendus.

```
static int[] glouton(int[][] G){//recoit en parametre un tableau à 2 dimensions (le prix des stocks e
    System.out.printf("on est au glouton\n");
    int nbEntrepot =G.length;
    int nbStocks = G[0].length;
    int h = 0;
    int k = 0;
    int[] listeNbStockParEntrepot = new int[nbEntrepot];
    int[] listeGainStockParEntrepot = new int[nbEntrepot];
    int diffMax = 0;
    int indiceDiffMax = 0;
    int diffEntrepot = 0;
    while(h < nbStocks-1) {//on parcours entrepotsMax (le nombre de stocks)
        indiceDiffMax = 0;
        diffMax = 0;
        //on veut augmenter les bénéfices de manière gloutonne
        //donc on cherche simplement à augmenter la marge faite par le vendeur stocks après stocks
        //on veut donc avoir la différence de prix qu'on a entre vendre k et k+1 stocks
        for(int i = 0; i < nbEntrepot; i++) {//parcours tous les entrepots
            //on recherche quel est la meilleur augmentation de profit et à quel entrepot
            //on retient cette différence max(diffMax) et à quel endroit (indiceDiffMax)

            if(listeNbStockParEntrepot[i] == 0 ) {
                diffEntrepot = G[i][listeNbStockParEntrepot[i]];
            }
            else {
                diffEntrepot = G[i][listeNbStockParEntrepot[i]] - G[i][listeNbStockParEntrepot[i]-1];
            }

            if(diffEntrepot > diffMax) {
                diffMax = diffEntrepot;
                indiceDiffMax = i;
            }
        }
```

Pour finir, on fait une boucle sur les différents entrepôts et on affecte au tableau listeGainStockParEntrepot le gain que l'on réalise dans chacun des entrepôts en fonction du nombre de stocks vendu. Enfin, on renvoie la liste des gains dans chacun des entrepôts. Cependant, il a fallu faire -1 sur la liste du nombre de stocks à chaque entrepôt car en effet si on vend 1 stock, ce stock vendu se situe à l'indice 0 du tableau des prix de stocks et ainsi de suite.

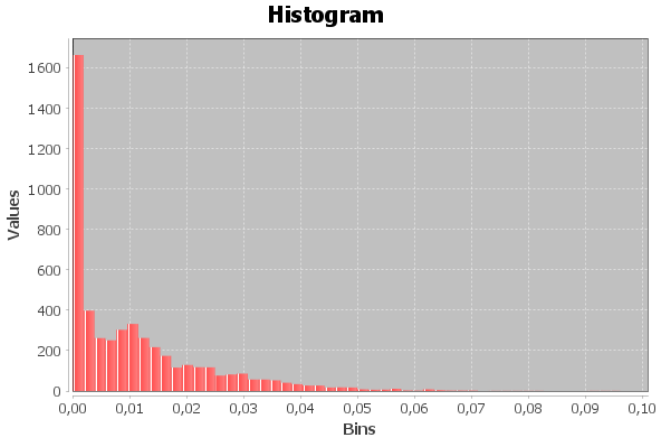
```
            if(diffEntrepot > diffMax) {
                diffMax = diffEntrepot;
                indiceDiffMax = i;
            }
        }
        //System.out.printf("diffMax = %d, indiceDiffMax = %d\n",diffMax, indiceDiffMax);
        listeNbStockParEntrepot[indiceDiffMax] += 1;//une fois qu'on est sur de l'endroit où il y a la meilleu
        h++;//puis on passe au stock suivant
    }
    //afficheTab(listeNbStockParEntrepot, "listeNbStockParEntrepot");
    System.out.printf("\n");
    while(k < nbEntrepot) {
        //on fait un tableau dans lequel il y a les différents gains effetues dans chacuns des entrepots
        listeNbStockParEntrepot[k] += -1;
        if(listeNbStockParEntrepot[k] == -1) {
            listeGainStockParEntrepot[k] = 0;
            k++;
        }
        else{
            listeGainStockParEntrepot[k] = G[k][listeNbStockParEntrepot[k]];
            k++;
        }
    }
    /*for(int j = 0; j < nbEntrepot; j++) {
        System.out.printf("listeGainStockParEntrepot[%d] = %d\n",j,listeGainStockParEntrepot[j]);
    }*/
    return listeGainStockParEntrepot;//on renvoie ce tableau des différents gains
}
```

Nous avons ensuite créé une fonction `chaqueTour` qui prend en paramètre le nombre d'entrepôts, le nombre de stocks et le numéro du tour (numéro du run). Elle retourne un float qui est la distance relative entre la valeur optimale et la valeur gloutonne. Dans cette fonction, on génère G, on en calcule la valeur optimale puis la valeur gloutonne en utilisant le tableau renvoyé par la fonction `glouton`. On utilise une fonction `somme` qui additionne toutes les valeurs fournies par `glouton(G)`. Puis on en déduit la distance relative.

On crée une fonction `tabDistRelative`, qui renvoie un tableau de float composé de toutes les distances relatives générées lors des 5000 runs. A chaque tour on génère un prixMaxStock aléatoire et un nombre d'entrepôts aléatoire et on affecte à l'indice i la valeur fournie par la fonction `chaqueTour`.

Le tableau retourné est ensuite analysé par la fonction `evalStat` qui comme vue précédemment affiche la moyenne, la médiane et l'écart-type de ce tableau que l'on peut voir ci-dessous.

```
moyenneRelative = 0,010930, MedianeRelative = 0,007288, EcartTypeRelatif = 0,012868
```



Exercice 4 Temps de travail :

Cet exercice ressemble au précédent. On a un élève qui estime les notes qu'il aurait en fonction de son temps de travail dans la matière. L'algorithme optimal a été fait en cours, par conséquent il faut

simplement faire la méthode gloutonne pour résoudre ce problème. Pour résoudre ce problème, nous avons repris l'algorithme glouton de l'exercice précédent et fait quelques modifications pour l'adapter à la nouvelle situation.

Ici aussi, on a une boucle while qui parcourt le nombre d'heures de travail qu'on a à disposition. Dans cette boucle, il y a une boucle for qui parcourt les unités/matières. On cherche à avoir une augmentation de la moyenne de l'élève, la plus forte. Ici, toutes les matières ont le même coefficient. Par conséquent, il faut que pour chaque heure de travail mise à disposition il y ait une augmentation de la note la plus forte possible. Exemple, si on a 3 matières et 3h de travail à disposition :

- Maths : 0h = 7, 1h = 10, 2h = 12, 3h =12
- Physique : 0h = 7, 1h = 15, 2h = 15, 3h =17
- Algorithmique : 0h = 6, 1h = 10, 2h = 14, 3h =15

On a regardé la différence de note entre aucune heure de travail et 1h de travail, pour la matière 1, la différence est de 3. Pour la matière 2, la différence est de 8 et pour la matière 3 la différence de note est de 4. Par conséquent il vaut mieux mettre la 1ère heure de travail au profit de la Physique.

Pour la 2ème heure, dans la matière 1 on a toujours 3, pour la matière 2, cette fois ci on a une différence de note de 0, et dans le cadre de la matière 3, on a toujours une différence de note de 4. On met donc à profit la 2ème heure de travail pour la matière 3.

Pour la 3ème heure de travail, on a toujours une différence de note pour la matière 1 de 3 points, pour la matière 2 de 0 points et cette fois-ci, pour la matière 3, une différence de 4 points. On met donc à profit cette 3ème heure de travail pour la matière 3.

Au final, on aura travail aucune heure pour la matière 1, 1h la matière 2 et 2h la matière 3.

C'est sensiblement ce qu'on a fait dans le cadre de la répartition des stocks dans les différents entrepôts mais au lieu de faire des différences de points gagnés, on avait une différence de gains.

Pour finir, on dresse une liste des différentes notes obtenues dans chacune des matières, l'indice correspond au numéro de la matière. La fonction renvoie donc le « bulletin » scolaire.

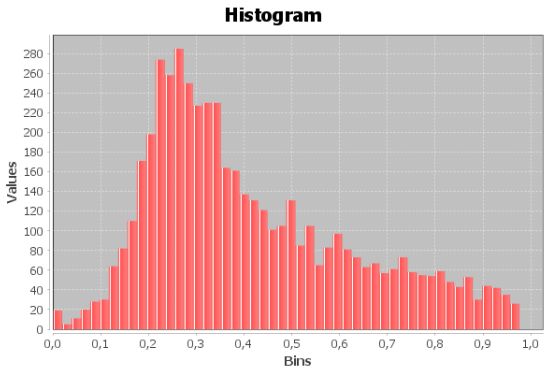
Voici le code de l'algorithme glouton ci-dessous :

```
static int[] glouton(int[][] G){//recoit en parametre un tableau à 2 dimensions (le prix des stocks et
    System.out.printf("on est au glouton\n");
    int nbEntrepot =G.length;
    int nbStocks = G[0].length;
    int h = 0;
    int k = 0;
    int[] listeNbStockParEntrepot = new int[nbEntrepot];
    int[] listeGainStockParEntrepot = new int[nbEntrepot];
    int diffMax = 0;
    int indiceDiffMax = 0;
    int diffEntrepot = 0;
    while(h < nbStocks-1) {//on parcours entrepotsMax (le nombre de stocks)
        indiceDiffMax = 0;
        diffMax = 0;
        //on veut augmenter les bénéfices de manière gloutonne
        //donc on cherche simplement à augmenter la marge faite par le vendeur stocks après stocks
        //on veut donc avoir la différence de prix qu'on a entre vendre k et k+1 stocks
        for(int i = 0; i < nbEntrepot; i++) {//parcours tous les entrepots
            //on recherche quel est la meilleur augmentation de profit et à quel entrepot
            //on retient cette différence max(diffMax) et à quel endroit (indiceDiffMax)

            if(listeNbStockParEntrepot[i] == 0 ) {
                diffEntrepot = G[i][listeNbStockParEntrepot[i]];
            }
            else {
                diffEntrepot = G[i][listeNbStockParEntrepot[i]] - G[i][listeNbStockParEntrepot[i]-1];
            }

            if(diffEntrepot > diffMax) {
                diffMax = diffEntrepot;
                indiceDiffMax = i;
            }
        }
        //System.out.printf("diffMax = %d, indiceDiffMax = %d\n",diffMax, indiceDiffMax);
        listeNbStockParEntrepot[indiceDiffMax] += 1;//une fois qu'on est sur de l'endroit où il y a la meilleur
        h++;//puis on passe au stock suivant
    }
    //afficheTab(listeNbStockParEntrepot, "listeNbStockParEntrepot");
    System.out.printf("\n");
    while(k < nbEntrepot) {
        //on fait un tableau dans lequel il y a les différents gains effetues dans chacuns des entrepots
        listeNbStockParEntrepot[k] += -1;
        if(listeNbStockParEntrepot[k] == -1) {
            listeGainStockParEntrepot[k] = 0;
            k++;
        }
        else{
            listeGainStockParEntrepot[k] = G[k][listeNbStockParEntrepot[k]];
            k++;
        }
    }
    /*for(int j = 0; j < nbEntrepot; j++) {
        System.out.printf("listeGainStockParEntrepot[%d] = %d\n",j,listeGainStockParEntrepot[j]);
    }*/
    return listeGainStockParEntrepot;//on renvoie ce tableau des différents gains
}
```

On fait la même chose que lors de l'exercice précédent pour faire l'évaluation statistique. On fait 5000 runs avec un nombre d'heures de travail disponible de manière aléatoire, un nombre de matières aléatoire et bien évidemment des notes aléatoires.

```
moyenneRelative = 0,011322, MedianeRelative = 0,007740, EcartTypeRelatif = 0,013032
```

On peut donc voir que cet algorithme glouton permet d'avoir des moyennes assez proches de ce qu'on aurait en faisant un algorithme optimal.



J'ai changé les valeurs au lieu de mettre jusqu'à 20 les valeurs des stocks maximum, elles sont jusqu'à 100. Ce qui crée des disparités plus importantes.

Exercice 5 Chemin somme Max, le Triangle :

Ici, nous devions mettre en place la situation nous-même puis résoudre ce problème en créant un algorithme optimal et un algorithme glouton.

Dans un premier temps, il a fallu créer un tableau à une dimension qui correspond au triangle. Nous avons donc créé une méthode qui s'appelle `initTabValeurs`, elle prend en paramètre un entier qui correspond au nombre de valeurs voulues dans le tableau. Cette méthode met des nombres aléatoires dans chacune des cases du tableau comprises entre 0 et 100 et ne renvoie rien car on a décidé dans cet exercice d'utiliser des variables globales pour optimiser la vitesse. Donc vTabValeurs est accessible partout.

```
void initTabValeurs(int pNbValeurs){//initialise de manière aléatoire un tableau
    int vNbRandom;

    for (int i=0;i<pNbValeurs; i++){
        vNbRandom = vRandom.nextInt(Vmax);
        vTabValeurs[i]=vNbRandom;
    }
}
```

Ensuite, il a fallu créer une méthode qui permet d'associer à chaque indice du tableau un rang, qui correspond à l'étage que cet indice occupe dans la pyramide. Nous avons donc créé la méthode `initTabRAng`. Elle est composée d'une boucle while qui parcourt le nombre de rangs maximums présents dans le tableau. Dans cette boucle il y a également une boucle for qui associe à chaque case identifiée par son indice un numéro de rang. Une fois que la boucle for est finie, on considère le rang suivant et on associe de nouveau aux valeurs suivantes un numéro de rang. Globalement, la boucle for met la même valeur partout dans le rang et la boucle while permet de passer au rang suivant pour qu'à son tour on mette la même valeur dans ce rang et ainsi de suite jusqu'au dernier rang. vTabRang est aussi une variable globale.

```
void initTabRang(){//permet d'établir un rang à chacun des indices du tableau
    //pour un rang n, il y a n valeurs or tot = n(n+1)/2 => 2tot =~n^2
    int i=0;
    int rang=0;
    while (i < Nmax) {
        for (int j=0; j < (rang +1); j++){
            vTabRang[i+j]= rang;
        }
        rang += 1;
        i += rang;
    }
}
```

Ensuite, il a fallu faire 2 fonctions :d pour la valeur en dessous à droite et g pour la valeur en dessous à gauche. On s'est aperçu que pour aller à la valeur en dessous à gauche sachant qu'on a un rang 0, il faut connaître l'indice auquel on se trouve, et ajouter le rang de l'indice auquel on se trouve +1 (car on a un rang 0). Pour aller à l'indice en dessous à droite, il faut procéder de la même manière, c'est-à-dire, connaître l'indice auquel on se trouve et ajouter le rang de cet indice puis ajouter 2 pour avoir l'indice situé après celui qui est à gauche.

```
int g(int pIndice){//retourne l'indice gauche suivant
    int a = pIndice + vTabRang[pIndice] + 1;
    //System.out.printf("indice de la val gauche %d\n",a);
    return a;//nouvel indice (si est le max)
}

int d(int pIndice){// retourne l'indice droit suivant
    int a = pIndice + vTabRang[pIndice] + 2;
    //System.out.printf("indice de la val droite %d\n",a);
    return a;//nouvel indice (si est le max)
}
```

Pour résoudre le problème suivant, nous avons commencé par la méthode gloutonne qui nous paraissait plus simple. Nous avons donc créé une fonction `glouton` qui prend en paramètre l'indice de départ (qui est ici tout le temps 0) et qui renvoie la somme gloutonne.

On commence par définir une variable indice qu'on instancie à 0 et une variable sommeGloutonne qu'on initialise à la valeur de la case 0 du tableau (1ère case). On fait ensuite une boucle for qui parcourt tous les rangs du triangle en partant du rang 1 car le rang 0 est déjà fait puisque l'on initialise la valeur de la somme gloutonne avec la 1ère valeur du tableau qui correspond au rang 0. Dans cette boucle, on affecte à la variable indice l'indice auquel il y a la valeur la plus élevée entre les valeurs situées en dessous à gauche et en dessous à droite, en utilisant la fonction créée `maxGetD`. Et après, on ajoute à la somme gloutonne, la valeur présente dans le tableau à cet indice.

```
int glouton(int pIndice) {
    int indice = 0;
    int vSommeGloutonne=vTabValeurs[0];
    for (int i=1; i<vNbRangs; i++) {
        indice = maxGetD(pIndice);
        vSommeGloutonne += vTabValeurs[indice];
        //System.out.printf("cet indice est: %d, Somme gloutonne :%d\n"
        pIndice = indice;
    }
    return vSommeGloutonne;
}
```

Avec la fonction `maxGetD` définie comme ceci, on vérifie que les indices en bas à droite et à gauche existent et si c'est le cas on regarde à quel indice la valeur est la plus grande entre celle en bas à droite et celle en bas à gauche :

```
int maxGetD(int pIndice){//permet de regader quel est la valeur max parmis ces 2 là
    if (g(pIndice)<=vNbVal && d(pIndice)<=vNbVal){
        if(vTabValeurs[g(pIndice)]>=vTabValeurs[d(pIndice)]){
            //System.out.printf("le max est à gauche\n");
            return g(pIndice);
        }
        else{
            //System.out.printf("le max est à droite\n");
            return d(pIndice);
        }
    }
    return 0;
}
```

Pour résoudre le problème de façon optimale, nous avons vu 2 solutions, l'une définie dans la fonction `m1` et l'autre dans la fonction `m`. Lorsque l'on lance l'algorithme nous lançons la solution `m` car celle-ci est plus rapide.

Pour m1 :

C'est fonction qui prend en paramètre un tableau M vide, dans lequel nous allons mettre les valeurs optimales dans chacune des cases du tableau. Ce tableau à la même taille que le tableau T qui est également mis en paramètre de cette fonction, il contient d'ailleurs toutes les valeurs de la pyramide. Cette fonction prend également en paramètre le tableau des rangs de chaque indice, ainsi qu'un entier l'indice i auquel on se trouve.

Dans un premier temps, on vérifie que l'indice auquel on se trouve ne se situe pas sur le dernier rang sinon cela signifie que le tableau M est entièrement complété.

Si ce n'est pas le cas, on appelle cette même fonction sur l'indice gauche et sur l'indice droit à la place de l'indice i. Puis on met dans la case i du tableau M la valeur présente dans le tableau T additionné à la plus grande valeur retrouvée en dessous à droite ou à gauche. Puis on renvoie cette valeur du tableau M[i].

On parcourt dans ce cas-là le tableau du haut vers le bas.

```
int m1(int []M,int []T, int [] rang, int rangMax, int i){//première version trop couteuse en temps
    System.out.printf("indice : %d et rangMax :%d \n",i,rangMax);
    if (rang[i] == rangMax){
        M[i]= T[i];
        return M[i];
    }
    else {
        int gauche = m1(M,T,rang,rangMax,g(i));
        int droite = m1(M,T,rang,rangMax,d(i));

        if(gauche < droite){
            M[i] = droite + T[i];
            return M[i];
        }
        else{
            M[i] = gauche + T[i];
            return M[i];
        }
    }
```

Pour m :

Nous faisons l'inverse, c'est-à-dire que nous parcourons le tableau du bas vers le haut. Cette fonction renvoie directement le tableau M complété au lieu de renvoyer case par case la valeur du tableau M.

Dans un premier temps, nous mettons dans toutes la dernière ligne du triangle (optimal, tableau M) la dernière ligne du triangle T. Si on ne se situe pas dans la dernière ligne du triangle, alors la valeur optimale à l'indice k (M[k]) est la somme de la valeur à l'indice k du tableau T (T[k]) additionnée à la plus grande des 2 valeurs situées en dessous à droite ou en dessous à gauche de la case M[k] soit les cases M[rang[k] + k +1] pour la gauche ou M[rang[k] + k +2] pour la droite. De cette manière, on remonte jusqu'au sommet de la pyramide avec la plus grande valeur, la valeur optimale de résolution à l'indice 0 du tableau M.

```
int[] m(int []M,int []T, int [] rang, int nbMaxElements, int i){
    for (int k = nbMaxElements; k >= i; k--) {
        if (rang[k] == rang[nbMaxElements]){
            M[k]= T[k];
        }
        else {
            M[k] = max(M[rang[k]+k+1],M[rang[k]+k+2]) + T[k];
        }
    }
    return M;
}
```

La fonction m est appelée par la fonction `calculerM` qui renvoie le tableau M construit dans la fonction m.

Nous avons également créé la fonction `acsm` qui prend en paramètre le tableau optimale, M et le tableau de valeurs T, ainsi que i, l'indice de départ (qui est en général 0) et n l'indice d'arrivée.

A partir de ces 2 indices i et n on récupère les rangs de départ et d'arrivée. On en définit donc la taille du chemin que l'algorithme va devoir parcourir. On crée un tableau qui a la taille du chemin que l'on va devoir parcourir (le nombre de rangs à passer). On crée des variables droite et gauche que l'on instancie à 0 pour le moment.

L'endroit de départ fait partie du chemin par conséquent on affecte à chemin[0] la valeur de i (de départ).

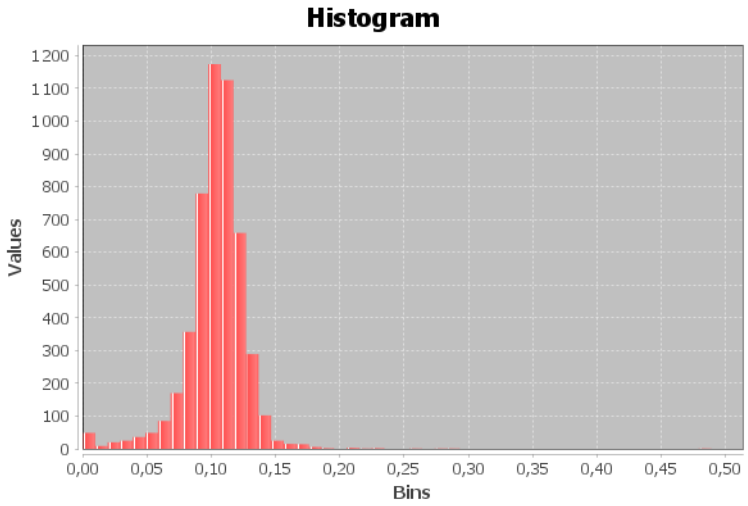
On crée ensuite une boucle while qui continue tant qu'on n'a pas réalisé le chemin total. On affecte la valeur des fonctions `g(i)` et `d(i)` aux variables droite et gauche créées précédemment. Puis on regarde quelle est la plus grande valeur entre M[droite] et M[gauche]. Sachant que M est le tableau optimal donc on a juste à descendre prendre les endroits où les valeurs sont les plus grandes, une à

une comme pour l'algorithme glouton. On prend la plus grande des 2 valeurs et on sait donc qu'on est passé par là donc on met son indice dans le tableau chemin à l'indice k.

```
void acsm(int[] M, int[] T, int i, int n) {
    int rangDepart = vTabRang[i];
    int rangArrivee = vTabRang[n-1]; //taille chemin arrivee-depart
    int tailleChemin = rangArrivee - rangDepart + 1;
    int [] chemin = new int [tailleChemin];
    int k = 0;
    int gauche = 0;
    int droite = 0;
    chemin[0] = i;
    while (k < tailleChemin-1) {
        gauche = g(i);
        droite = d(i);
        if (M[gauche]<= M[droite]) {
            i = droite;
        }
        else {
            i = gauche;
        }
        k++;
        chemin[k] = i;
    }
    for(k = 0; k < tailleChemin; k++) {
        System.out.printf("case %d, T[%d],M[%d]\n", chemin[k], T[chemin[k]], M[chemin[k]]);
    }
}
```

On réalise ensuite l'analyse statistique :

```
moyenne : 0,103630, mediane : 0,105423, EcartType : 0,023061
```



Conclusion

L'algorithme optimal donne toujours une meilleure solution que l'algorithme glouton. Cependant, l'algorithme glouton a des avantages : Il est plus rapide et plus facile à développer. Dans certains cas, le résultat obtenu est assez proche de l'algorithme optimal. Il est donc intéressant de se poser les bonnes questions pratiques avant de résoudre un problème pour bien comprendre les enjeux de la meilleure solution.