EFAYONG Anthony
DUONG Hoang-Duc

Compte rendu du projet d'algorithmique

Partie 1:

Dans l'exercice 1, le but du programme à créer était, à partir d'une grille en 2 dimensions remplies d'une quantitée variée de pucerons, de pouvoir trouver le chemin optimal permettant de manger le plus de pucerons depuis la ligne la plus au sud vers la ligne la plus au nord. La coccinelle ne pouvait seulement se déplacer vers le nord, le nord-est et le nord-ouest.

La première étape était de créer un tableau M de terme général M[l][c]=m(l,c), avec m(l,c) le nombre maximum de pucerons que la coccinelle peut manger pour arriver sur la case de coordonnées (l,c).

Pour cela il a fallu utiliser un algorithme ressemblant à celui vu précédemment lors du TD sur le calcul du chemin de coût minimum pour qu'un robot se déplace de la case (0,0) à la case (l,c).

```
public static void main(String[] args){

    int M[][]={
            {2,4,3,9,6},
            {1,10,15,1,2},
            {2,4,11,26,66},
            {36,34,1,13,30},
            {46,2,8,7,15},
            {89,27,10,12,3},
            {1,72,3,6,6},
            {3,1,2,4,5},
    };


    int Mp[][]= copyOfM(M);

    //M et son calcul sont gardés dans Ms, une liste a 3 dimensions
    int[][][] Ms= new int[2][M.length][M[0].length];
    Ms[0]= Mp;
    Ms[1]=calculerM(M);

    afficherM(Ms);

}
```

Ci dessus se trouve la fonction Main de notre programme. On donne dans la variable M la grille 2D de pucerons, ensuite on crée une copie Mp de cette grille. On calcule la grille de terme général M[l][c]=m(l,c) (avec m(l,c) le nombre maximum de pucerons que la coccinelle peut manger pour arriver sur la case de coordonnées (l,c)) et on insère Mp et la grille calculée dans la variable 3D Ms. On affiche ensuite le résultat du programme.

```java
static int[][] copyOfM(int[][] M){
    int[][] mFin= new int[M.length][M[0].length];
    for(int i=0; i<M.length; i++){
        for(int j=0; j<M[0].length; j++){
            mFin[i][j]= M[i][j];
        }
    }
    return mFin;
}
```

Pour copier la matrice originale on utilise la fonction copyOfM. Cette fonction parcours linéairement la Matrice M donnée et renvoie une copie mFin de la Matrice M donnée.

```java
34    static int[][] calculerM(int[][] M){
35        int L= M.length, C=M[0].length;
36        int mNE=0, mN=0, mNO=0;
37        for(int pL=1; pL<L; pL++){
38            for(int pC=0; pC<C; pC++){
39                mN= M[pL][pC] + valCase(pL-1, pC, M);
40                mNE= M[pL][pC] + valCase(pL-1, pC-1,M);
41                mNO= M[pL][pC] + valCase(pL-1, pC+1,M);
42                //calcul max
43                M[pL][pC]= Math.max(mN, Math.max(mNE, mNO));
44            }
45        }
46        return M;
47    }
```

Notre fonction “calculerM” est ici faite pour calculer la grille de pucerons maximum pouvant être mangés par case. Elle prend en argument une liste 2D ‘M’.

La fonction parcourt linéairement la liste 2D M donnée, en remplaçant la case (l,c) par la somme de cette case et du maximum entre la case du sud, sud-est et sud-ouest, et retourne la grille 2 dimensions calculée.

Ensuite il reste à trouver le chemin contenant le plus de pucerons dans la grille. Avant d'aborder l'explication de "trouverChemin" nous allons expliquer quelques fonctions faites pour le fonctionnement de celle ci.

```java
static int maxi(int[] List){
    int max=List[0];
    for(int i : List){
        if(i>max){
            max=i;
        }
    }
    return max;
}
```

```java
static int valCase(int pL, int pC, int[][] M){
    int L= M.length, C=M[0].length;
    if (pL>L-1|| pL<0) return 0;
    if(pC>C-1 || pC<0) return 0;
    else return M[pL][pC];
}
```

```java
static int trouverIndice(int[] M, int val){
    for(int i=0; i<M.length; i++){
        if(M[i]==val){
            return i;
        }
    }
    return -1;
}
```

- La fonction maxi prend en paramètre une liste 1D et renvoie la valeur maximum de cette liste.
- La fonction valCase renvoie la valeur de la case dans la matrice M au indices pL pC. Si ces indices sont en dehors de la Matrice, la fonction renvoie 0 (Pour que cette case ne soit pas prise pour la case de coût maximum dans "trouverChemin", sinon une erreur se ferait)
- La fonction trouverIndice prend en paramètre une Matrice et une valeur et renvoie l'indice de la première occurence de cette valeur.

```java
static void trouverChemin(int[][] M, int ligne, int colonne, List<String> chemin){
    int mN,mNE, mNO;
    int valMax, indiceChemin;

    //Cas de base (Si il n'y a plus de valeurs sous la case)
    if(valCase(ligne, colonne,M)==0) return;
    if(valCase( pL: ligne-1, colonne, M)==0){
        String pos= "("+ligne+","+colonne+")";
        System.out.print(pos+" ");
        chemin.add(pos);
        return;
    };

    //Calcul max
    mN=  valCase( pL: ligne-1, colonne, M);
    mNE= valCase( pL: ligne-1,  pC: colonne-1,M);
    mNO= valCase( pL: ligne-1,  pC: colonne+1,M);

    valMax= Math.max(mN, Math.max(mNE, mNO));
    indiceChemin= trouverIndice(M[ligne-1], valMax);


    //Recursion
    trouverChemin(M,  ligne: ligne-1, indiceChemin, chemin);
    String pos= "("+ligne+","+colonne+")";
    System.out.print(pos+" ");
    chemin.add(pos);

    return;
}
```

Notre fonction trouverChemin marche comme ceci :

- **Sa condition d'arrêt :**
- Si la valeur de la case actuelle est égale à 0 (Si on sort de la Matrice)

- **Initialisation:**

```java
static void trouverChemin(int[][] M){
    int valMax= maxi(M[M.length-1]);
    int indiceFin= trouverIndice(M[M.length-1], valMax);
    trouverChemin(M,  ligne: M.length-1, indiceFin, chemin);
}
```

-
- On part du maximum de la dernière ligne pour retracer le chemin inverse contenant le nombre de pucerons maximum.

- **La récursion :**
- Détermination de la case du dessous de plus haute valeur afin de s'assurer de manger le plus de pucerons possible par étape. Exécution de la même fonction ensuite sur cette case.

```java
static void afficherM(int[][][] Ms){
    int[][] M= Ms[0];
    int[][] m= Ms[1];


    int L= m.length, C=m[0].length;
    int valMax= maxi(m[L-1]);


    //Affichage de la grille des pucerons
    System.out.println("Grille des pucerons: ");
    printM(M);

    //Affichage tableau du nombre max de pucerons par case
    System.out.println("Tableau M[L][C] de terme général M[l][c] = m(l,c) :");
    printM(m);

    System.out.print("\nLa coccinelle a mangé "+ valMax +" pucerons !\nElle a suivi le chemin suivant : ");
    trouverChemin(m);


    System.out.println(
                    "\nCase d'atterisage = "+chemin.get(0)+
                    "\nCase d'interview = "+chemin.get(M.length-1)
    );
}
```

Ensuite se fait l'affichage du résultat (Se composant de la valeur de la Matrice de départ, de la Matrice calculée, du chemin ainsi que de la case de départ et d'arrivée.

Voici l'exemple de calcul du sujet du projet avec notre programme :

```
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Grille des pucerons:
3 1 2 4 5
1 72 3 6 6
89 27 10 12 3
46 2 8 7 15
36 34 1 13 30
2 4 11 26 66
1 10 15 1 2
2 4 3 9 6

Tableau M[L][C] de terme général M[l][c] = m(l,c) :
279 277 278 149 145
205 276 145 140 140
204 142 124 134 125
115 71 98 114 122
64 69 51 90 107
16 28 35 50 77
5 14 24 10 11
2 4 3 9 6


La coccinelle a mangé 279 pucerons !
Elle a suivi le chemin suivant : (0,3) (1,2) (2,2) (3,1) (4,0) (5,0) (6,1) (7,0)
Case d'atterisage = (0,3)
Case d'interview = (7,0)

Process finished with exit code 0
```

Partie 2:

```
import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.awt.Color;
```

Pour pouvoir réussir convenablement cette seconde partie, il nous faut tout d'abord importer plusieurs bibliothèques nécessaires aux traitement d'images: ImageIO pour la lecture d'image et File pour les fichiers, IOException pour utiliser les fonctions d'imageIO, BufferedImage pour traiter les images et Color pour manipuler les différents RGB.

```java
/**
 * Affiche la BufferedImage passée en paramètre.
 * @param image image a afficher.
 */
private static void afficherImage(BufferedImage image) {
    JFrame frame = new JFrame("Image");
    frame.getContentPane().setLayout(new FlowLayout());
    frame.getContentPane().add(new JLabel(new ImageIcon(image)));
    frame.pack();
    frame.setVisible(true);
}

/**
 * Affiche le tableau 2D passé en paramètre.
 * @param tab Tableau 2D d'entiers a afficher.
 */
private static void afficher2D(int[][] tab){
    for(int x=tab.length-1; x>=0; x--){
        for(int y= 0; y<tab[0].length; y++){
            System.out.print(tab[x][y]+" ");
        }
        System.out.println();
    }
}
```

Ci dessus se trouvent des fonctions permettant respectivement de faire une affichage d'une BufferedImage et d'afficher dans le terminal un tableau 2D.

```java
/**
 * Retourne la transposée de la Matrice passée en paramètre.
 * @param M Matrice 2D Double a transposer.
 * @return Transposée de M.
 */
private static double[][] transpoz(double[][] M){
    double[][] Mt= new double[M[0].length][M.length];
    for(int i=0; i< M[0].length ; i++){
        for(int j=0; j<M.length; j++){
            Mt[i][j]=M[j][i];
        }
    }
    return Mt;
}
```

Cette fonction permet de faire la transposée de la matrice correspondante à l'image. La fonction utilisée pour trouver le chemin étant basé sur celui de la partie 1, elle trouvera le chemin d'énergie minimum

vertical. On a donc créé une fonction pouvant transposer une matrice, afin de lire et repérer les seams horizontaux.

```java
/**
 * Retourne la valeur située en M[pL][pC]. Si la valeur n'est pas dans la Matrice M, retourne l'infini.
 * @param pL ligne
 * @param pC colonne
 * @param M Matrice 2D M.
 * @return la valeur de la case si elle existe, l'infini sinon.
 */
private static double valCase(int pL, int pC, double[][] M){
    int L= M.length, C=M[0].length;
    if (pL>L-1|| pL<0) return Double.POSITIVE_INFINITY;
    if(pC>C-1 || pC<0) return Double.POSITIVE_INFINITY;
    else return M[pL][pC];
}

/**
 * Retourne l'indice de la première occurence du double donné dans le tableau 1D donné.
 * @param M tableau Double.
 * @param val valeur dont l'indice est a déterminer.
 * @return indice de val.
 */
private static int trouverIndice(double[] M, double val){
    for(int i=0; i<M.length; i++){
        if(M[i]==val){
            return i;
        }
    }
    return 0;
}
```

La fonction valCase retourne la valeur d'une case en fonction de sa position dans la matrice. Ici la valeur dépend de la position colonne et la position ligne. Si la valeur de la case que l'on cherche se trouve en dehors de la matrice M donnée, la fonction renvoie la valeur +infini. Elle renvoie sinon la valeur de la Matrice M[pL][pC].

La fonction trouverIndice permet de trouver l'indice de la première occurrence du double val donné dans une liste de double 1D M.

```java
/**
 * Retourne le minimum de la liste de Double.
 * @param List liste de Double.
 * @return minimum de la liste Double donnée.
 */
private static double mini(double[] List){
    double min=List[0];
    for(double i : List){
        if(i<min){
            min=i;
        }
    }
    return min;
}
```

Cette fonction retourne le minimum de la liste. Ici on s'intéresse au minimum d'une ligne ou d'une colonne.

```java
/**
 * Arrondi la valeur de sortie avec un nombre de décimal choisi
 * @param value Valeur en entrée non arrondie
 * @param nb Nombre de décimal
 * @return
 */
public static double round(double value, int nb) {
    if (nb < 0) throw new IllegalArgumentException();

    long factor = (long) Math.pow(10, nb);
    value = value * factor;
    long reduc = Math.round(value);
    return (double) reduc / factor;
}
```

On utilise une fonction round qui permettra d'arrondi à la décimale souhaitée le temps de calcul de redimension de l'image.

```
private static double getPixEnergy(int pix1, int pix2){
    Color a= new Color(pix1);
    Color b= new Color(pix2);
    double pixEnergy;

    double red2= b.getRed();
    double green2= b.getGreen();
    double blue2= b.getBlue();
```

La première étape a été de pouvoir créer une liste 2D de gradients RGB sde chaques pixels se trouvant sur l'image. Pour cela on a créé une fonction qui calcule l'énergie d'un pixel qui équivaut au gradient de la somme des pixels rouge, bleu et vert. Pour çela on compare l'énergie des 2 pixels de coordonnées supérieurs et inférieurs du pixel concerné. Les commandes getRed, getBlue et getGreen détermine le degré de rouge, de bleu et de vert du pixel.

```
pixEnergy= Math.pow((red1-red2), 2) + Math.pow((green1-green2), 2) + Math.pow((blue1-blue2), 2);

return pixEnergy;
```

```
private static double[][] calculerMatEnergie(BufferedImage image){
    int imageLargeur= image.getWidth();
    int imageHauteur= image.getHeight();
    double[][] grilleEnergie= new double[imageLargeur][imageHauteur];
```

Par la suite, il faut donc conserver chaque énergie de chaque pixel dans un tableau organisé selon la position du pixel sur l'image. Nous avons donc pour cela crée une matrice qui stocke les valeurs en énergie des différents pixels de l'image. On regarde ici en fonction de la largeur et de la hauteur de l'image en utilisant getWidth et getHeight. On va ensuite comparer l'énergie avec les pixels autour de lui (gauche droite haut et bas). En fonction de la position du pixel, on ajuste les paramètres des autres pixels. En appliquant cela à l'horizontale et à la verticale, on trouve donc une grille d'énergie de l'image que l'on insère dans la matrice d'énergie.

```
if(x==0){
    //colonne de gauche
    pixelG= image.getRGB(x,y);
    pixelD= image.getRGB(x+1,y);
}
```

```
//Energie horizontale
energieHorizontale= getPixEnergy(pixelG, pixelD);
```

Nous avons ensuite créés les chemins adéquats à parcourir en fonction de l'orientation. Si le chemin est horizontal, on laisse le programme parcourir la matrice normalement. Si le chemin suivi est vertical, on a demandé au programme à ce qu'il suive le chemin de la matrice d'énergie Transposée pour aller dans une direction opposée tout en utilisant la même fonction findSeam et par la suite on inverse le résultat vertical pour obtenir les valeurs dans les bonnes normes.

```
if(orientation.equals("Vertical")){
    double[][] Mt= transpoz(M);
    chemin = findSeam(Mt);
    chemin= inverse2D(chemin);
}
return chemin;
```

Nous avons séparé la fonction findSeam en 2 parties: une où l'on va chercher le Seam de coût minimal et puis le transférer récursivement vers la findSeam et l'autre qui détaille tous les calculs pour retrouver le chemin de la Seam de coût minimal.

```java
/**
 * @param M Matrice dans lequel trouver le Seam de coût minimal.
 * @return chemin du Seam de coût minimal.
 */
private static int[][] findSeam(double[][] M){
    int[][] chemin= new int[M.length][2];
    double valMin= mini(M[M.length-1]);
    int indiceFin= trouverIndice(M[M.length-1], valMin);
    findSeam(M, ligne: M.length-1, indiceFin, chemin);
    return chemin;
}
```

Dans cette première partie on reprend les différentes fonctions de calculs présentées précédemment, ici mini et trouverIndice. On cherche tout d'abord la taille de la matrice puis des valeurs se trouvant dans la matrice à savoir ici le minimum. Puis on prend tous les paramètres nécessaires dans la fonction findSeam.

```java
private static void findSeam(double[][] M, int ligne, int colonne, int[][] chemin){
    double mN,mNE, mNO;
    double valMin;
    int indiceChemin=0;
```

```
//Cas de base (Si il n'y a plus de valeurs sous la case)
if(valCase(ligne, colonne,M)==Double.POSITIVE_INFINITY) return;
if(valCase( pL: ligne-1, colonne, M)==Double.POSITIVE_INFINITY){
    chemin[ligne][0]= ligne;
    chemin[ligne][1]= colonne;
    return;
}
```

```
//Calcul max
mN=  valCase( pL: ligne-1, colonne, M);
mNE= valCase( pL: ligne-1,  pC: colonne-1,M);
mNO= valCase( pL: ligne-1,  pC: colonne+1,M);

valMin= Math.min(mN, Math.min(mNE, mNO));

if(valMin==mN)  indiceChemin= colonne;
if(valMin==mNE) indiceChemin= colonne-1;
if(valMin==mNO) indiceChemin= colonne+1;
```

La seconde partie dépend de plusieurs fonctions comme valCase qui détermine la position du pixel et voir si elle est toujours dans la bonne dimension d'image. On va ensuite de haut en bas la bonne valeur pour le Seam en regardant les valeurs situées au nord, nord-est et nord-ouest. Au final, on compare les trois valeurs possibles et cela détermine la valeur de coût minimum qui s'intègre dans le chemin optimal. On répète ensuite la fonction récursivement pour aller sur l'intégralité de l'image, ici la verticale.

```java
private static BufferedImage enleverSeam(BufferedImage image, int[][] chemin, String direction){
    boolean decaler=false;
    int couleur;
    int largeur= image.getWidth();
    int hauteur= image.getHeight();
    BufferedImage newImage= null;
```

```java
//Chemin horizontal a enlever
else if(direction.equals("Horizontal")){
    for(int x=0; x<largeur; x++){
        for(int y=0; y<hauteur-1;y++){
            if(chemin[x][0]==x && chemin[x][1]==y){
                decaler= true;
            }

            if(decaler) couleur= image.getRGB(x,y+1);
            else couleur= image.getRGB(x,y);

            newImage.setRGB(x,y,couleur);
        }
        decaler=false;
    }
}
return newImage;
```

Après avoir trouvé les Seams, il fait maintenant trouver comment les retirer et donc recadrer l'image proportionnellement. On a instancié ici un boolean decaler qui vérifie si le pixel en question doit être retiré et donc décaler les pixels d'à côté. Si c'est le cas, alors la variable couleur qui définit le RGB du pixel change donc de valeur en paramètre, un décalage de coordonnées verticale ou horizontale.

```java
 */
private static BufferedImage seamCarv(BufferedImage image, int pourcentH, int pourcentV){
    System.out.println("Calculs en cours ...");

    int nbIterationHorizontal= pourcentH* image.getWidth() /100;
    int nbIterationVertical= pourcentV*   image.getHeight()/100 ;
```

```
for(int x= 0; x<Math.max(nbIterationHorizontal, nbIterationVertical); x++){
    if(x<nbIterationHorizontal){
        double[][] grille= calculerMatEnergie(image);
        int[][] cheminV=findSeam(grille, orientation: "Vertical");
        image= enleverSeam(image, cheminV, direction: "Vertical");
    }
```

La fonction seamCarv regroupe une grande majorité de toutes les fonctions écrites jusqu'à présent. En lançant le programmant dans le terminal l'utilisateur va entrer le pourcentage de l'horizontal et de la verticale qu'il souhaite garder, qu'on notera nbIteration.

On a ensuite utiliser un point x de coordonnées inférieures au deux itérations verticale et horizontale. Tant que x reste inférieur, il va continuer de s'incrémenter et selon sa valeur calculer son énergie, déterminer le nombre de seam optimal à trouver et ensuite les retirer pour la nouvelle image.

```
public static void main(String[] args){

    //start
    double lStartTime = System.nanoTime();

    BufferedImage image;
    int pourcentH, pourcentV;
```

```
image= seamCarv(image, pourcentH, pourcentV);
String nomFichier = param.substring(0,param.length()-4)+"_resized_"+pourcentH+'_'+pourcentV+".png";
try{
    ImageIO.write(image, "png", new File("../Images/"+nomFichier));
}
catch(Exception e){
    System.out.println(e);
}
//afficherImage(image);
System.out.println("\nCalculs termines. \nUn fichier redimensionne a ete cree !");
```

La fonction Main de notre programme est à découper en plusieurs parties. On lance le temps de calcul dès le début du programme.

Tout d'abord une première partie sert à savoir si les bons arguments ont étés donnés dans la ligne de commande. On regarde si l'image donnée existe bien et peut être ouverte, sinon le programme s'arrête en renvoyant un message d'erreur. Ensuite les arguments donnés en position 1 et 2 dans le tableau Args (qui

correspondent aux pourcentages de réduction de l'image) sont testés : On regarde s'ils sont inférieurs à 100 et supérieurs à 0 (L'image ne peut pas être agrandie).

Un SOP notifie l'utilisateur du bon fonctionnement du programme et ensuite l'algorithme est lancé (Avec la ligne image= seamCarv(image, pourcentH, pourcentV)).

Ensuite le fichier contenant l'image redimensionnée est créé dans le même dossier que l'image originale.

On pose ensuite une capture du temps à la fin de l'exécution du programme. La différence de cette valeur avec celle du début nous donneras le temps en nanoseconde de la conversion que l'on divisera par 10^9 pour obtenir un temps en seconde. Par la suite, on appliquera la fonction round pour donner une meilleur valeur estimée.

Un message de fin s'affiche, suivi du temps d'exécution du programme.

```
C:\Users\dduon\Documents\ESIEE\E2\Algos\file>javac SeamCarved.java

C:\Users\dduon\Documents\ESIEE\E2\Algos\file>java SeamCarved ../Images/canyon.png 50 50

Recuperation image OK
Reduction de l'image de : 50% x 50%. (Horizontal x Vertical)
Calculs en cours ...

Calculs termines.
Un fichier redimensionne a ete cree !

Temps de calcul: 393.907 secondes
```

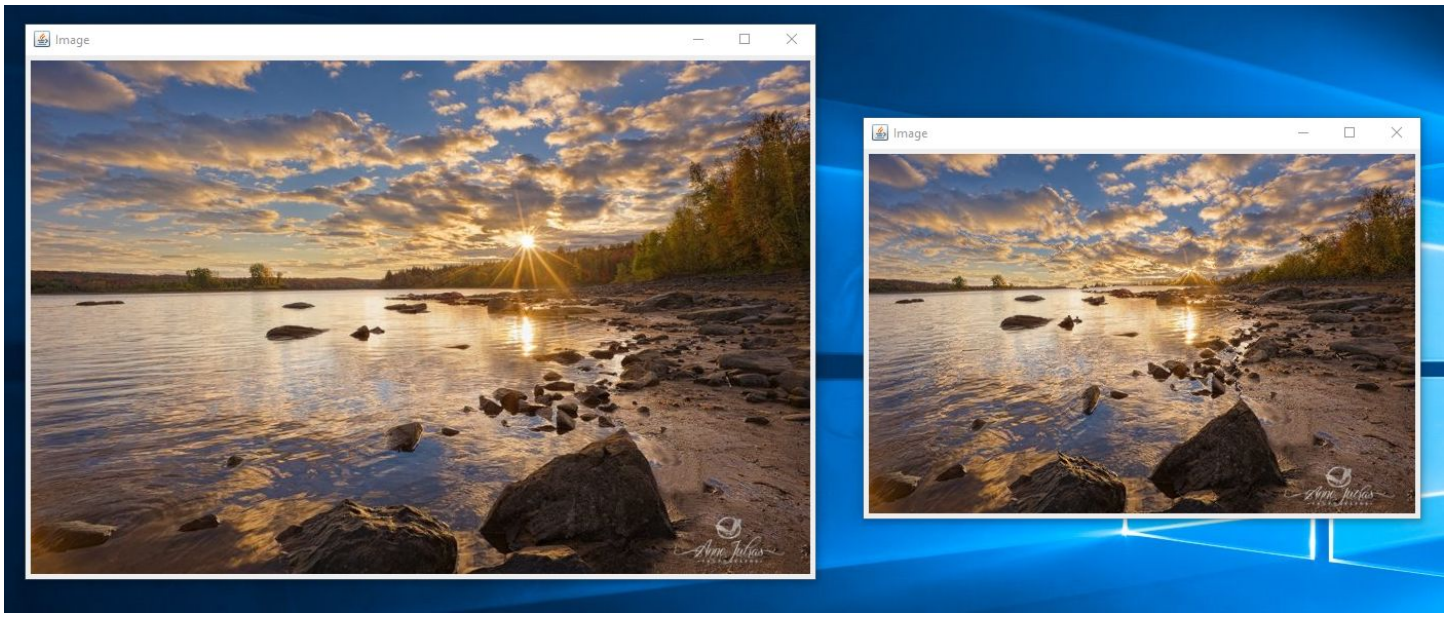
Tests de notre programme:

```
C:\Users\dduon\Documents\ESIEE\E2\Algos\file>set path=%path%;C:\Program Files\Java\jdk1.8.0_251\bin

C:\Users\dduon\Documents\ESIEE\E2\Algos\file>javac SeamCarved.java

C:\Users\dduon\Documents\ESIEE\E2\Algos\file>java SeamCarved ../Images/paysage.png 30 30

Recuperation image OK
Reduction de l'image de : 30% x 30%. (Horizontal x Vertical)
Calculs en cours ...

Calculs termines.
Un fichier redimensionne a ete cree !

Temps de calcul: 10.3 secondes
```

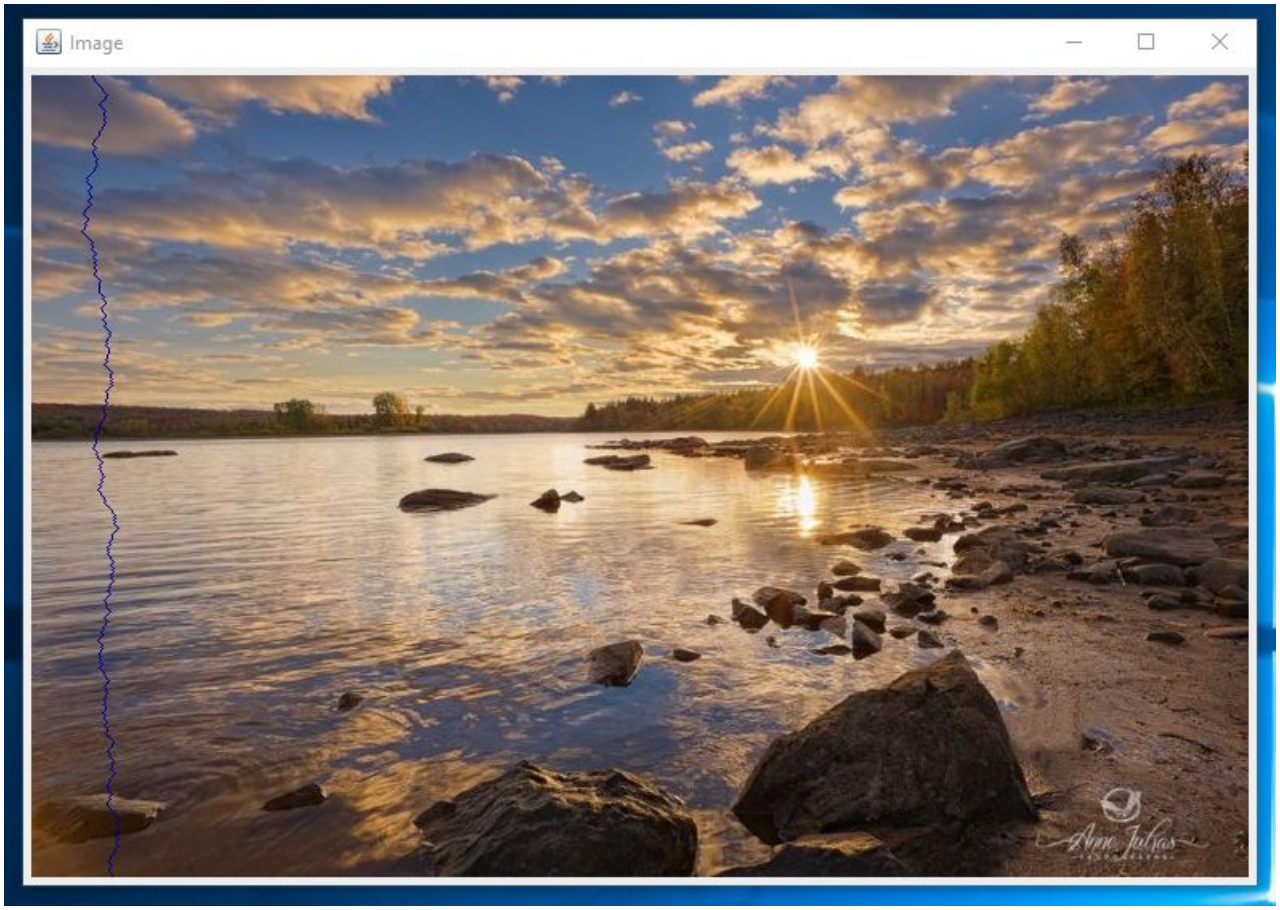
paysage

paysage_resized_30_30

Un fichier paysage_resized_30_30.png a bien été créé dans le même dossier que paysage.png.



Voici les résultats côte à côte de paysage.png et de son redimensionnement.



Voici ci-dessus l'exécution d’un seul tour de “seamCarv”.