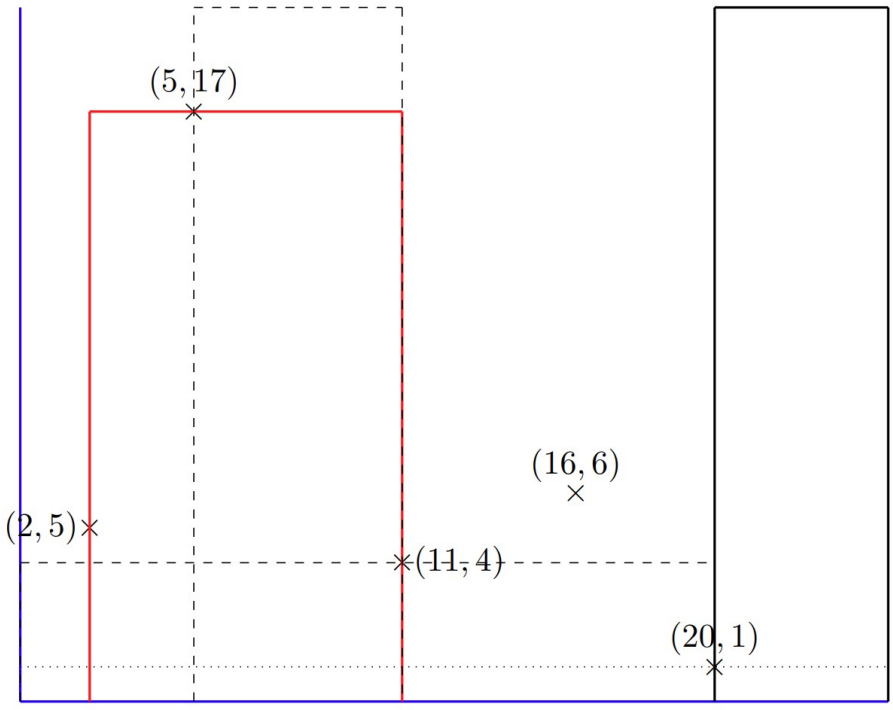
Rapport de TP

Algorithmes et Complexité

Le plus grand rectangle



Hamani Rayane
Nguyen Kévin

Master 1 d'Informatique
2020 - 2021

Introduction

Objectif pédagogique

Le TP aborde un problème d'algorithme simple. L'objectif pédagogique est de réfléchir à différentes solutions algorithmiques tout en se posant les questions de complexité et de correction.

Le problème

Soit la région rectangulaire du plan déterminée par le rectangle (0,0) (0,h) (l,h) (l,0), h et l étant deux entiers positifs et soient n points à coordonnées entières à l'intérieur de cette région. On souhaite dessiner un rectangle dont la base est sur l'axe des abscisses, dont l'intérieur ne contienne aucun des n points et qui soient de surface maximale.

Question 1 : Une première approche

Il figure parmi les contraintes que le rectangle doit avoir sa base sur l'axe des abscisses. C'est pourquoi deux des quatre sommets doivent avoir 0 comme ordonnée.

La hauteur z est la hauteur maximale possible entre plusieurs points permettant de délimiter la surface du rectangle et forcément les 2 sommets sur l'axe des abscisses,
La surface maximal est de (xj – xi) * z

Avec une complexité en O(n^3), nous ne pensons pas que la contrainte d'exécution en moins d'1 seconde sera respectée. Avec une complexité en O(n^2), il est possible que notre algorithme s'exécute en moins d'1 seconde pour n = 100 000 (il passe le test 8) mais ce n'est pas une garantie. Nous devons donc le rendre plus efficace.

Parmi les contraintes, il y a que le rectangle doit avoir sa base sur l'axe des x et pour avoir le rectangle de surface maximale il faut nécessairement avoir 2 sommets du rectangle, sur l'axe des abscisses. et les 2 sommets restant du nuage de points.

On constate que les algorithmes en O(n^2) et O(n^3) sont extrêmement coûteux en temps sur des jeux de données très grand.

Dans l'état actuel de l'algorithme non, il ne sera pas possible de réaliser la tâche en moins de 1 seconde pour un jeu de données de 100 000 lignes.

Question 2 : Diviser pour régner

L'algorithme est plus efficace que l'algorithme précédent. Cependant, une observation faite sur le tout premier exemple avec les 5 points, la majorité des points se retrouvent dans un des 2 tableaux, le rendant juste aussi performant que l'ancienne algorithme en O(n^2). Cependant lorsque la répartition des points dans les 2 tableaux est pratiquement équilibré, l'algorithme diviser pour régner devient beaucoup plus performant, avec une complexité en O(n*log(n)). Ce qu'il faut éviter avec cet algorithme donc, ce sont les séries de points croissants en abscisse et décroissants en ordonnée.

Après étude de la complexité de l'algorithme diviser pour régner :

Dans le pire des cas, il y a un énorme déséquilibre lors de la distribution des points dans les 2 tableaux lors des multiples récursivités, dans ce cas on est en O(n^2).

Dans le meilleur des cas, la répartition des points entre les 2 tableaux lors des multiples récursivités est pratiquement équilibrée, dans ce cas on est en O(n*log(n)).

Question 3 : Linéaire ?

Pour réaliser l'algorithme en complexité linéaire, on a inclut une pile. Celle-ci sert à stocker les nouveaux points parcourus dont l'ordonnée augmente par rapport à la tête de pile. Concrètement, à chaque fois qu'on descend en ordonnée par rapport à la tête de pile, on dépile la pile et on calcule les rectangles avec le point fraîchement dépilé, le point courant et la nouvelle tête de pile.

L'avantage avec cette approche est que l'on construit le rectangle maximal alors connu de part les points déjà parcourus.

Annexes

Code de la question 1 : complexité quadratique, en O(n^3)

```
import java.util.ArrayList;
import java.util.Scanner;

public class Theta_N3 {
   public static class Point {
      private final int x;
      private final int y;

      public Point(int x, int y) {
         this.x = x;
         this.y = y;
      }

      public int getX() {
         return x;
      }

      public int getY() {
         return y;
      }
   }

   public static void main(String[] args) {
      /* on lit les inputs pour le programme */
      Scanner sc = new Scanner(System.in);

      int lengthGrid = sc.nextInt();
      int heightGrid = sc.nextInt();
      int numberOfPoints = sc.nextInt();
      ArrayList<Point> listOfPoints = new ArrayList<Point>();

      /* on ajoute les points à prendre en compte */
      listOfPoints.add(new Point(0, 0));
      for(int i = 0; i < numberOfPoints; i++) {
         int x = sc.nextInt();
         int y = sc.nextInt();
         Point point = new Point(x, y);
         listOfPoints.add(point);
      }
      listOfPoints.add(new Point(lengthGrid, 0));
      numberOfPoints += 2; /* on met à jour le nombre de point dans la liste */

      /* Calcule en O(n^3), pour calculer la surface de rectangle la plus grande possible */
      int surfaceMax = 0;
       /* on itére sur tout les points du graphe qu'on nommera i */
      for(int i = 0; i < numberOfPoints-1; i++) {
            /* Pour chacun de ces points i on va calculer la surface maximale possible avec tout les points j ce trouvant
après le point j. */
         for(int j = i+1; j < numberOfPoints; j++) {
            int hauteurMinimale = heightGrid;
                /* on va itérer sur tout les points nommés k, entre les points i et j afin de trouver la hauteur minimale
parmi ces points k
                car normalement certains points intermédaire peuvent ce trouver a l'intérieur des rectangles,
interdisant normalement le calcules de surface de ces
                rectangles, mais a la place on va minimiser au maximum leur surface pour eviter de trouver une
surface maximale érronée. */
                if ((j - i) > 1) {
                   for(int k = i+1; k < j; k++){
```

```
                    if (listOfPoints.get(k).getY() < hauteurMinimale) {
                                hauteurMinimale = listOfPoints.get(k).getY();
                    }
                        }
                     }
                    /*On regarde si la surface calculée est plus grande que la surface maximale actuelle trouvée*/
                     int surface = (listOfPoints.get(j).getX() - listOfPoints.get(i).getX()) * hauteurMinimale;
                     if ( surfaceMax < surface ) {
                    surfaceMax = surface;
                     }
        }
    }

    /* on affiche la surface */
    System.out.println(surfaceMax);

    sc.close();
  }
}
```

Code de la question 1 : complexité quadratique, en O(n^2)

```
import java.util.ArrayList;
import java.util.Scanner;

public class Theta_N2 {
    public static class Point {
        private final int x;
        private final int y;

        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public int getX() {
            return x;
        }

        public int getY() {
            return y;
        }
    }

    /**
     * Calcule en O(n^2) l'aire du plus grand rectangle parmi les points donnés
     *
     * @param maxHauteur la hauteur de l'axe des ordonnées
     * @param points la liste des points
     * @param nombreDePoints le nombre de points dans la liste
     * @return l'aire du plus grand rectangle vide et ayant sa base sur l'axe des abscisses
     */
    public static int calculDuPlusGrandRectangle(int maxHauteur, ArrayList<Point> points, int nombreDePoints) {
        int maxAire = 0;

        /* pour chaque bordure gauche */
        for(int i = 0; i < nombreDePoints-1; i++) {
            /* on initialise la hauteur minimale à la hauteur de l'axe des ordonnées */
            int minHauteur = maxHauteur;

            /* pour chaque bordure droite de la bordure gauche */
            for(int j = i+1; j < nombreDePoints; j++) {
                /* si l'arête droite est plus haute que la hauteur minimale */
                if(minHauteur < points.get(j).getY()) {
                    continue; /* on ignore le point et on passe au suivant */
                }
                /* sinon si l'arête droite est plus basse que la hauteur minimale */
                else {
                    /* on calcule l'aire du rectangle entre les points d'index i et j et de hauteur minimale */
                    int newAire = (points.get(j).getX() - points.get(i).getX()) * minHauteur;

                    /* on met à jour l'aire maximale */
                    maxAire = Math.max(maxAire, newAire);

                    /* on met à jour la nouvelle hauteur minimale pour la prochaine bordure droite */
                    minHauteur = Math.min(minHauteur, points.get(j).getY());
                }
            }
        }

        return maxAire;
    }

    public static void main(String[] args) {
        /* on lit les inputs pour le programme */
        Scanner sc = new Scanner(System.in);
```

```
        final int maxLongueur = sc.nextInt();
        final int maxHauteur = sc.nextInt();

        int nombreDePoints = sc.nextInt() + 2;
        ArrayList<Point> points = new ArrayList<Point>();

        /* on liste les points à prendre en compte */
        points.add(new Point(0, 0));
        for(int i = 0; i < nombreDePoints-2; i++) {
            int x = sc.nextInt();
            int y = sc.nextInt();
            points.add(new Point(x, y));
        }
        points.add(new Point(maxLongueur, 0));

        /* on n'oublie pas de fermer le scanner ! */
        sc.close();

        /* on affiche l'aire du plus grand rectangle après l'avoir calculé ! */
        System.out.println(calculDuPlusGrandRectangle(maxHauteur, points, nombreDePoints));
    }
}
```

Code de la question 2 : diviser pour régner

```
import java.util.ArrayList;
import java.util.Scanner;

public class DiviserPourRegner {
   public static class Point {
      private final int x;
      private final int y;

      public Point(int x, int y) {
         this.x = x;
         this.y = y;
      }

      public int getX() {
         return x;
      }

      public int getY() {
         return y;
      }
   }

   /**
    * Calcule en O(n.log(n)) l'aire du plus grand rectangle parmi les points donnés
    *
    * @param maxHauteur la hauteur de l'axe des ordonnées
    * @param points la liste des points
    * @param nombreDePoints le nombre de points dans la liste
    * @return l'aire du plus grand rectangle vide et ayant sa base sur l'axe des abscisses
    */
   public static int calculDuPlusGrandRectangle(int maxHauteur, ArrayList<Point> points, int nombreDePoints) {

         /* s'il n'y a pas assez de points pour former un rectangle */
         if(nombreDePoints  < 2) return 0; /* retourne une valeur qui sera ignorée */
      /* s'il y a tout juste assez de points pour former un rectangle */
       else if(nombreDePoints == 2) return (points.get(1).getX() - points.get(0).getX()) * maxHauteur; /* retourne l'aire du
seul rectangle construisable */
      /* sinon s'il y a plus deux points */
      else {
         /* on cherche l'indice de la hauteur minimale parmi les points autres que le premier et le dernier */
         int minHauteurIndex = 1;
         for(int i = 2; i < nombreDePoints-1; i++) {
            if(points.get(i).getY() < points.get(minHauteurIndex).getY()) minHauteurIndex = i;
         }

         /* on calcule l'aire du rectangle aux arêtes gauche et droite les plus éloignés */
         int maxAire = (points.get(nombreDePoints-1).getX() - points.get(0).getX()) * points.get(minHauteurIndex).getY();

         /* on divise la liste des points en deux, le point à la hauteur minimale étant le point d'intersection */
         ArrayList<Point> pointsAGauche = new ArrayList<Point>(points.subList(0, minHauteurIndex+1));
         ArrayList<Point> pointsADroite = new ArrayList<Point>(points.subList(minHauteurIndex, nombreDePoints));

         /* on met à jour l'aire maximale */
                        maxAire = Math.max(maxAire, calculDuPlusGrandRectangle(maxHauteur, pointsAGauche,
minHauteurIndex+1));
            maxAire = Math.max(maxAire, calculDuPlusGrandRectangle(maxHauteur, pointsADroite, nombreDePoints-
minHauteurIndex));

         return maxAire;
      }
   }

   public static void main(String[] args) {
      /* on lit les inputs pour le programme */
```

```
        Scanner sc = new Scanner(System.in);

        final int maxLongueur = sc.nextInt();
        final int maxHauteur = sc.nextInt();

        int nombreDePoints = sc.nextInt() + 2;
        ArrayList<Point> points = new ArrayList<Point>();

        /* on liste les points à prendre en compte */
        points.add(new Point(0, 0));
        for(int i = 0; i < nombreDePoints-2; i++) {
            int x = sc.nextInt();
            int y = sc.nextInt();
            points.add(new Point(x, y));
        }
        points.add(new Point(maxLongueur, 0));

        /* on n'oublie pas de fermer le scanner ! */
        sc.close();

        /* on affiche l'aire du plus grand rectangle après l'avoir calculé ! */
        System.out.println(calculDuPlusGrandRectangle(maxHauteur, points, nombreDePoints));
    }
}
```

Code de la question 3 : complexité linéaire, en O(n)

```
import java.util.ArrayList;
import java.util.Scanner;
import java.util.Stack;

public class Theta_N {
   public static class Point {
      private final int x;
      private final int y;

      public Point(int x, int y) {
         this.x = x;
         this.y = y;
      }

      public int getX() {
         return x;
      }

      public int getY() {
         return y;
      }
   }

   /**
    * Calcule en O(n) l'aire du plus grand rectangle parmi les points donnés
    *
    * @param longueurAbscisse la longueur de l'axe des abscisses
    * @param hauteurOrdonnee la hauteur de l'axe des ordonnées
    * @param points la liste des points
    * @param nombreDePoints le nombre de points dans la liste
    * @return l'aire du plus grand rectangle vide et ayant sa base sur l'axe des abscisses
    */
   public static int calculDuPlusGrandRectangle(int maxHauteur, ArrayList<Point> points, int nombreDePoints) {
      int maxAire = 0;
      Stack<Integer> pile = new Stack<Integer>(); /* la pile des indices des points dans la liste */

      /* pour chaque point sauf (0,0) */
      for(int i = 1; i < nombreDePoints; i++) {
            /* si la pile est vide ou que l'ordonnée du point actuel est supérieure ou égale à celle du point en tête de pile
*/
            /* on calcule l'aire de hauteur de l'axe des ordonnées entre le point actuel et le point précédent */
            /* on met à jour l'aire maximale */
            /* on empile le point */
            if(pile.isEmpty() || points.get(i).getY() >= points.get(pile.peek()).getY()) {
               int newAire = (points.get(i).getX() - points.get(i-1).getX()) * maxHauteur;
               maxAire = Math.max(maxAire, newAire);
                  if(!pile.isEmpty() && points.get(pile.peek()).getY() == points.get(i).getY()) pile.pop(); /* si l'ordonnée de la
tête de la pile = l'ordonnée du point actuel, on enlève la tête de pile */
               pile.push(i);
            }

            /* sinon si la pile n'est pas vide et que l'ordonnée du point actuel est inférieure à celui de la tête de liste */
            else {
               /* on dépile les points de la pile jusqu'à obtenir un point dont l'ordonnée est inférieure */
                  /* on calcule l'aire de hauteur du point dépilé entre le point actuel et soit la tête de pile si la pile n'est pas
vide, soit 0 si la pile est vide */
               /* on met à jour l'aire maximale */
               while(!pile.isEmpty() && points.get(i).getY() <= points.get(pile.peek()).getY()) {
                  int iPointDepile = pile.pop();
                  int newAire = 0;
                                    if(!pile.isEmpty())  newAire  =  points.get(iPointDepile).getY()  *  (points.get(i).getX()  -
points.get(pile.peek()).getX());
                  else              newAire = points.get(iPointDepile).getY() *  points.get(i).getX();
                  maxAire = Math.max(maxAire, newAire);
```

```
                }

                /* on ajoute le point actuel en tête de pile */
                pile.push(i);
            }
        }

        /* on dépile le dernier point de la liste */
        /* on calcule l'aire de hauteur de l'axe des ordonnées entre la tête de la pile et le point qui le précède dans la liste des points */
        /* on met à jour l'aire maximale */
        int iPointDepile = pile.pop();
        int newAire = (points.get(iPointDepile).getX() - points.get(iPointDepile-1).getX()) * maxHauteur;
        maxAire = Math.max(maxAire, newAire);

        return maxAire;
    }

    public static void main(String[] args) {
        /* on lit les inputs pour le programme */
        Scanner sc = new Scanner(System.in);

        final int maxLongueur = sc.nextInt();
        final int maxHauteur = sc.nextInt();

        int nombreDePoints = sc.nextInt() + 2;
        ArrayList<Point> points = new ArrayList<Point>();

        /* on liste les points à prendre en compte */
        points.add(new Point(0, 0));
        for(int i = 0; i < nombreDePoints-2; i++) {
            int x = sc.nextInt();
            int y = sc.nextInt();
            points.add(new Point(x, y));
        }
        points.add(new Point(maxLongueur, 0));

        /* on n'oublie pas de fermer le scanner ! */
        sc.close();

        /* on affiche l'aire du plus grand rectangle après l'avoir calculé ! */
        System.out.println(calculDuPlusGrandRectangle(maxHauteur, points, nombreDePoints));
    }
}
```