Школа Аналізу Даних

Курс: "Алгоритми і структури даних"

Домашня робота №1

Підготував: Руслан Сакевич

12 лютого 2016 р.

Зміст

1 Умова задачі

На площині задано N точок. Чи можна побудувати замкнену ламану без самоперетинів та самодотиків з вершинами тільки в заданих точках, яка з'єднує всі N точок?

Вхідні дані

Перший рядок вхідного файлу містить натуральне число $N(1 \leq N \leq 10^5)$. Кожний з наступних N рядків містить пару цілих чисел x_i, y_i – координати i-ї точки$(-10^5 \leq x_i, y_i \leq 10^5)$.

Вихідні дані

В першому рядку вихідного файлу виведіть **Yes** або **No** в залежності від того, чи існує шукана ламана. Якщо ламана існує виведіть в другому рядку N цілих чисел через пробіл – номера вершин ломаної в порядку обходу. Нумерація вершин починається з одиниці.

Приклади

4 0 0 0 1 1 0 1 1	Yes 1 3 4 2
3 0 0 0 1 0 2	No

2 Алгоритм розв'язання

Нехай M - множина вхідних точок, $|M| = N$.
Опишемо алгоритм з *низьким рівнем деталізації*.

1. Серед точок з M виберемо дві точки A та B.
   Введемо позначення $M' = M \backslash \{A, B\}$.

2. Проведемо уявну пряму AB, і розіб'ємо множину M' на три підмножини:

   (a) M_{up} – множина точок з M', що знаходяться над прямою AB.

   (b) M_{on} – множина точок з M', що знаходяться на прямій AB.

   (c) M_{down} – множина точок з M', що знаходяться під прямою AB.

3. В залежності від того які множини порожні маємо декілька варіантів:

   (a) $M_{up} = M_{down} = \varnothing$
   Тоді зрозуміло що всі точки з M знаходяться на прямій AB.
   І в такому випадку неможливо побудувати шукану ламану.

   (b) $M_{up} = \varnothing$ або $M_{down} = \varnothing$
   Не порушуючи загальності візьмемо, що
   $M_{down} = \varnothing, M_{up} \neq \varnothing$
   Тоді включимо точки з M_{on} до порожньої множини M_{down}
   $M'_{down} = M_{down} \cup M_{on} = \varnothing \cup M_{on} = M_{on}$
   $M'_{up} = M_{up}$

   (c) $M_{up} \neq \varnothing, M_{down} \neq \varnothing$
   Тоді не важливо до якої множини включати точки з M_{on}.
   Нехай це буде множина M_{up}, маємо:
   $M'_{down} = M_{down}$
   $M'_{up} = M_{up} \cup M_{on}$

4. Побудуємо верхню ламану U на точках з M'_{up}, і окремо нижню ламану D на точках з M'_{down}.

5. Обєднаємо ламані U та D з вершинами A та B, і отримаємо *шукану замкнену ламану*.

3 Детелізація алгоритму

Спробуємо деталізувати наш алгоритм, а саме відповісти на наступні запитання:

- за яким прицином обирати *крайні точки* A, B?
- як будувати ламані U та D?
- чому при обєднанні ламаних U та D утвориться ламана без самоперетинів та самодотиків?

Для того, щоб зрозуміти принцип вибору точок A та B, потрібно спочатку зрозуміти алгоритм побудови ламаних U та D.

3.1 Побудова верхньої ламаної

Розглянемо алгоритм побудови верхньої ламаної U(алгоритм побудови нижньої ламаної D, буде аналогічний, за виключенням декількох деталей, які будуть зазначені).

1. Відсортуємо точки з M'_{up} в порядку зростання значення їх координат(спочатку за першою, потім за другою).
2. З'єднаємо точки в такому порядку, і отримаємо ламану U.

Зрозуміло, що отримана ламана U не буде містити самоперетинів. Це легко довести від супротивного, припустивши що самоперетин існує, і розглянувши можливі варіанти розташування кінців ланок ламаної, що перетинаються.

3.2 Вибір точок A та B

На даний момент, ми вже можемо дати відповідь на перше запитання, а саме. Оскільки в результаті ми маємо отримати верхню ламану з кінцями в точках A та B, то ці точки мають бути на початку і в кінці відсортованого списку. Для визначеності вважатимемо, що точка A – на початку, а точка B – в кінці. Отже точка A повинна мати мінімальні координати, а точка B – максимальні.

Оскільки алгоритм має властивість детермінованості, то будемо вибирати точку A, не тільки з найменшою першою координатою, але й з найменшою другою. Аналогічно B – точка з найбільшою першою координатою, і найбільшою другою. З точки зору геометрії, точка A – найнижча з найлівіших точок, а точка B - найвища, з найправіших.

3.3 Модифікації для нижньої ламаної

Врато зазначити, як алгоритм побудови нижньої ламаної буде відрізнятися від алгоритму для верхньої. Виявляється, що різниця буде лише в тому, що сортувати точки потрібно по спаданню їх координат. Тоді перша точка в відсортованому списку буде точка B, а остання – точка A. Завдяки цьому нам буде легко з'єднувати ламані, оскільки не прийдеться змінювати порядок вершин.

3.4 Чому алгоритм правильний?

Залишається відкритим лише третє питання, щодо того, чому взагалі описаний алгоритм буде давати правильну відповідь на задачу. А саме, потрібно показати, що утворена замкнена ламана, не буде містити самоперетинів та самодотиків. Зрозуміло, що якщо верхня ламана не має самоперетинів, і нижня ламана не має самоперетинів, то і утворена ламана не буде мати самоперетинів.

Для того, щоб показати, що утворена ламана не буде мати самодотиків, необхідно повернутися до побудови множин M'_{up} та M'_{down}. Зрозуміло що самодотики можуть утворитися тільки в точках, що лежать на прямій AB. Але з побудови M'_{up} та M_{down} випливає, що точки що лежать на прямій завжди будуть відноситися до однії з ламаних, і не буде такого, що одна точка буде в верхній ламаній, а інша в нижній.

Особливу увагу потрбіно приділити, випадку коли якась з множин M_{up} чи M_{down} буде порожньою. Тоді за алгоритмом побудови "розширених"множин, точки на прямій потраплять до порожньої ламаної, і ламана виродиться у відрізок AB. Важливо зрозуміти, що якби не було такого строгого розподілу точок на прямій, то можливий був би варіант, коли верхня ламана виродилася у відрізок, а нижня ламана містила б точку на прямій AB, і тоді б утворився самодотик. Але цей варіант

неможливий, завдяки детермінованому алгоритму побудови "розширених"множин M'_{up} та M'_{down}.

3.5 Часткові випадки

Дамо відповідь і на наступне запитання. Чи існують які небудь часткові випадки в яких алгоритм працювати не буде. Виявляється, що існують. А саме, випадок, коли *не можливо* вибрати точки A та B. Це можливо тільки тоді, коли $N = 1$. Оскільки задачею не регламентовано, то існування ламаної випливає з того, чи вважати замкненою ламаною одну точку. Для цього звернемося до *Вікіпедії*:

> A closed polygonal chain is one in which the first vertex coincides with the last one.

Отже будемо вважати одну точку, замкненою ламаною, що задовольняє умову.

4 Складність алгоритму

Для того, щоб оцінити алгоритм достаньо дати оцінку кожному з його пунктів, оскільки він вже розбитий на декілька незалежних частин.

Крок алгоритму	**Часова складність**
Вибір точок A та B	$O(1)$
Розбиття точок на множини M'_{up}, M'_{down}	$O(N)$
Сортування точок в множинах M'_{up}, M'_{down}	$O(NlogN)$[1]
Побудова шуканої ламаної	$O(1)$
Загальна складність алгоритму	$O(NlogN)$

[1] *Вважаючи, що ми використовуємо сортування, що в найгіршому випадку мають часову складність* $O(NlogN)$.

З таблиці видно, що загальна часова складність алгоритму становить $O(NlogN)$.

Говорячи про використану пам'ять, можна стверджувати, що алгоритм потребує $O(N)$ додаткової пам'яті, для побудови множин M'_{up} та M'_{down}.

5 Реалізація на Java

```
/**
 * Created by lionell on 11/1/15.
 *
 * @author Ruslan Sakevych
 */

import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.util.StringTokenizer;

public class ClosedPolyline {
    public static void main(String[] args) {
        Reader in = new Reader();
        int n = in.nextInt();
        Point[] m = new Point[n];
        for (int i = 0; i < n; i++) {
            int x = in.nextInt();
            int y = in.nextInt();
            m[i] = new Point(x, y, i + 1);
        }
        in.close();
        // Build desired polyline
        int[] order = PolylineBuilder.build(m);
        // Print polyline
        PrintWriter out = new PrintWriter(new BufferedOutputStream(Syst
        if (order.length == 0) {
            out.println("No");
        } else {
            out.println("Yes");
```

```
            for (int index : order) {
                out.print(index);
                out.print("␣");
            }
        }
        out.close();
    }

    private static class Point implements Comparable<Point> {
        private int x;
        private int y;
        private int i;

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

        public boolean lessThan(Point o) {
            return compareTo(o) < 0;
        }

        @Override
        public int compareTo(Point o) {
            if (equals(o)) {
                return 0;
            }
            return getX() < o.getX() ||
                    getX() == o.getX() && getY() < o.getY()
                    ? -1 : 1;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) {
                return true;
            }
            if (o == null || getClass() != o.getClass()) {
                return false;
            }
```

```
            Point point = (Point) o;

            return getX() == point.getX() && getY() == point.getY();
        }

        @Override
        public int hashCode() {
            final int PRIME = 31;
            int result = getX();
            result = PRIME * result + getY();
            return result;
        }

        public int getX() {
            return x;
        }

        public int getY() {
            return y;
        }

        public int getI() {
            return i;
        }
    }

    private static class PolylineBuilder {
        private static final Comparator<Point> DESCENDING_ORDER =
                new Comparator<Point>() {
                    @Override
                    public int compare(Point o1, Point o2) {
                        return o2.compareTo(o1);
                    }
                };

        private static long direction(Point a, Point b, Point c) {
            return (long) (b.getX() - a.getX()) * (c.getY() - a.getY()
                    (long) (b.getY() - a.getY()) * (c.getX() - a.getX(
        }
```

```
public static int[] build(Point[] points) {
    // Find points a and b
    // Size of points is greater than 1
    Point a = points[0];
    Point b = points[1];
    for (Point p : points) {
        if (p.lessThan(a)) {
            a = p;
        }
        if (b.lessThan(p)) {
            b = p;
        }
    }
    // Generate arrays mUp, mOn and mDown
    List<Point> mUp = new ArrayList<>();
    List<Point> mOn = new ArrayList<>();
    List<Point> mDown = new ArrayList<>();
    for (Point p : points) {
        long wedgeProduct = direction(a, b, p);
        if (wedgeProduct > 0) {
            mUp.add(p);
        } else if (wedgeProduct < 0) {
            mDown.add(p);
        } else if (!p.equals(a) && !p.equals(b)) {
            mOn.add(p);
        }
    }
    // Check if we can build polyline
    if (mUp.isEmpty() && mDown.isEmpty()) {
        return new int[0];
    }
    // Extend arrays mUp and mDown
    if (mDown.size() == 0) {
        mDown.addAll(mOn);
    } else {
        mUp.addAll(mOn);
    }
    // Sort points in arrays mUp and mDown
    Collections.sort(mUp);
    Collections.sort(mDown, DESCENDING_ORDER);
    // Combine points to required polyline
```

```
        ArrayList<Point> polyline = new ArrayList<>();
        polyline.add(a);
        polyline.addAll(mUp);
        polyline.add(b);
        polyline.addAll(mDown);
        // Generate polyline vertexes order
        int[] order = new int[polyline.size()];
        for (int i = 0; i < polyline.size(); i++) {
            order[i] = polyline.get(i).getI();
        }
        return order;
    }
}

private static class Reader {
    private BufferedReader br;
    private StringTokenizer st;

    public Reader() {
        br = new BufferedReader(new InputStreamReader(System.in));
    }

    public Reader(String s) {
        try {
            br = new BufferedReader(new FileReader(s));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    private String nextToken() {
        while (st == null || !st.hasMoreElements()) {
            try {
                st = new StringTokenizer(br.readLine());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return st.nextToken();
    }
```

```
        public int nextInt() {
            return Integer.parseInt(nextToken());
        }

        public void close() {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```