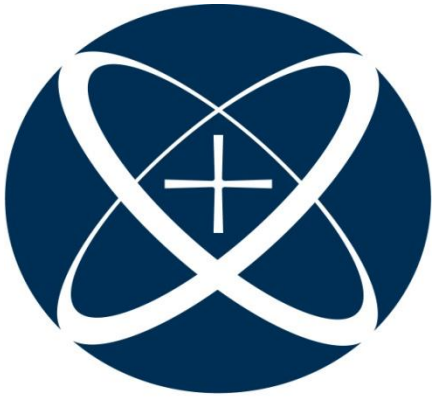
Instituto Tecnológico y de Estudios Superiores de Occidente – ITESO



ITESO

Universidad Jesuita de Guadalajara

Materia:	Estrategias Algorítmicas
Maestro:	Gilbert Pérez Diego Rodolfo
Actividad:	Proyecto 1
Fecha:	14/10/2024
Autor:	Cisneros Gutiérrez Carlos Arturo

Introducción

Se realizó el primer problema de la lista de dos problemas proporcionada por el profesor. Para esto se utilizo una variante del algoritmo del barrido angular y MergeSort. Este método consiste en encontrar el centro geométrico del conjunto de todos los puntos (centroide). Posteriormente se calculan los ángulos polares de todos los puntos con respecto a este centroide y se utilizan para ordenar los puntos en sentido horario o antihorario. Se utiliza el MergeSort para el ordenamiento debido a su alta eficiencia en estos casos.

Análisis a Priori:

- Método calcularCentroide:

```
public static Pair<Double, Double> calcularCentroide(List<Pair<Integer,
Integer>> puntos) {
        double sumaX=0, sumaY=0;
        for (Pair<Integer, Integer> punto:puntos) {
            sumaX+=punto.getKey();
            sumaY+=punto.getValue();
        }
        return new Pair<>(sumaX/puntos.size(), sumaY/puntos.size());
    }
```

Este método es llamado una sola vez en toda le ejecución. Recorre los N elementos de la lista y por cada uno realiza 2 operaciones. Simplificando, podría decirse que este método es de complejidad O(n).

- Método calcularAngulo:

```
public static double calcularAngulo(Pair<Double, Double> centroide,
Pair<Integer, Integer> punto) {
        double deltaX=punto.getKey() - centroide.getKey();
        double deltaY=punto.getValue() - centroide.getValue();
        return Math.atan2(deltaY, deltaX);
    }
```

Este método es llamado muchas veces durante la ejecución. Sin embargo, las operaciones que realiza en cada ejecución son constantes. Es de complejidad constante, O(1).

- Método merge:

```
public static List<Pair<Integer, Integer>> merge(List<Pair<Integer,
Integer>> izquierda, List<Pair<Integer, Integer>> derecha, Pair<Double,
Double> centroide) {
        List<Pair<Integer, Integer>> resultado=new ArrayList<>();
        int i=0, j=0;

        while (i<izquierda.size() && j<derecha.size()) {
            double anguloIzquierda=calcularAngulo(centroide,
izquierda.get(i));
            double anguloDerecha=calcularAngulo(centroide,
derecha.get(j));

            if (anguloIzquierda<=anguloDerecha) {
                resultado.add(izquierda.get(i));
                i++;
```

```
            } else {
                resultado.add(derecha.get(j));
                j++;
            }
        }

        //Añade los elementos restantes
        while (i<izquierda.size()) {
            resultado.add(izquierda.get(i));
            i++;
        }
        while (j<derecha.size()) {
            resultado.add(derecha.get(j));
            j++;
        }

        return resultado;
    }


    public static List<Pair<Integer, Integer>>
generarPuntosAleatorios(int n, int rangoX, int rangoY) {
        List<Pair<Integer, Integer>> puntos=new ArrayList<>();
        Random random=new Random();
        for (int i=0;i<n;i++) {
            int x=random.nextInt(rangoX);
            int y=random.nextInt(rangoY);
            puntos.add(new Pair<>(x, y));
        }
        return puntos;
    }
```

En este método todo es de ejecución constante excepto los bucles while, donde la complejidad es de O(n) para el primero y O(n/2) para los siguientes. Por lo que tiene una complejidad total O(n).

- Método mergeSort:

```
public static List<Pair<Integer, Integer>> mergeSort(List<Pair<Integer,
Integer>> puntos, Pair<Double, Double> centroide) {
        if (puntos.size()<=1) {
            return puntos;
        }

        //Divide la lista en dos mitades
        int mitad=puntos.size()/2;
        List<Pair<Integer, Integer>> izquierda=new
ArrayList<>(puntos.subList(0, mitad));
        List<Pair<Integer, Integer>> derecha=new
ArrayList<>(puntos.subList(mitad, puntos.size()));

        //Ordena ambas mitades
        izquierda=mergeSort(izquierda, centroide);
        derecha=mergeSort(derecha, centroide);

        //Mezcla ambas mitades ordenadas
        return merge(izquierda, derecha, centroide);
```

```
        }
```

Si el tamaño de puntos es igual o menor a uno retorna el conjunto, con un costo O(1). Al dividir la lista en dos se tiene un costo operacional de O(n/2), u O(n).

Por último, se recurre a la recursión, por lo que el algoritmo se divide recursivamente en dos sublistas de tamaño aproximadamente n/2, lo que lleva a una recurrencia de la forma T(n) = 2T(n/2) + O(n), donde O(n) es el costo de la combinación.

Debido a que la lista se va dividiendo en 2 hasta frenar la recurrencia podemos obtener una expresión para la cantidad de llamadas recursivas:

$$\frac{n}{2^k} = 1 \quad \rightarrow \quad k = \log_2(n)$$

Además de esto, en cada llamada recursiva se tiene un costo O(n) al juntar las listas, por lo que la complejidad total es O(n log n).

En resumen

La complejidad teórica de este algoritmo de ordenamiento es de O(n log n), siendo bastante eficiente ya que es recursivo.

Análisis a Posteriori:

Para el análisis a posteriori se crearon 100 listas, cada una con 1000 elementos más que la anterior y se tomó el tiempo que se tardaba en ordenar cada lista, así como la cantidad de llamadas a métodos que hacía. Los resultados fueron los siguientes:

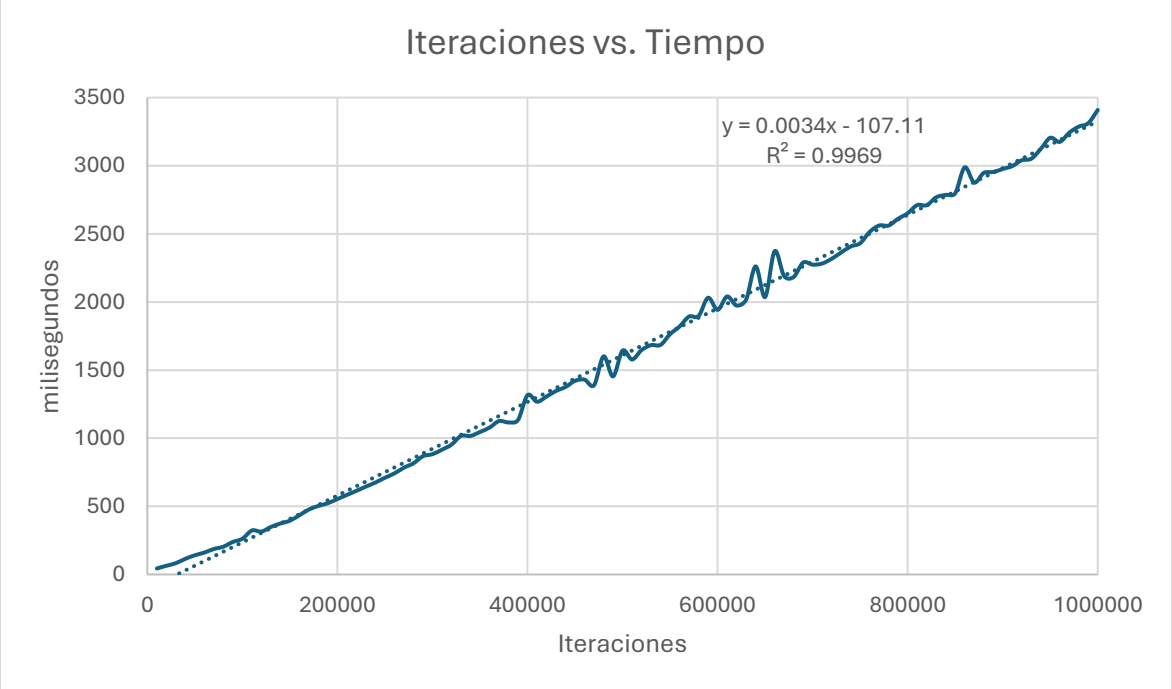


Figura 1. Gráfico de iteraciones contra tiempo de ejecución.

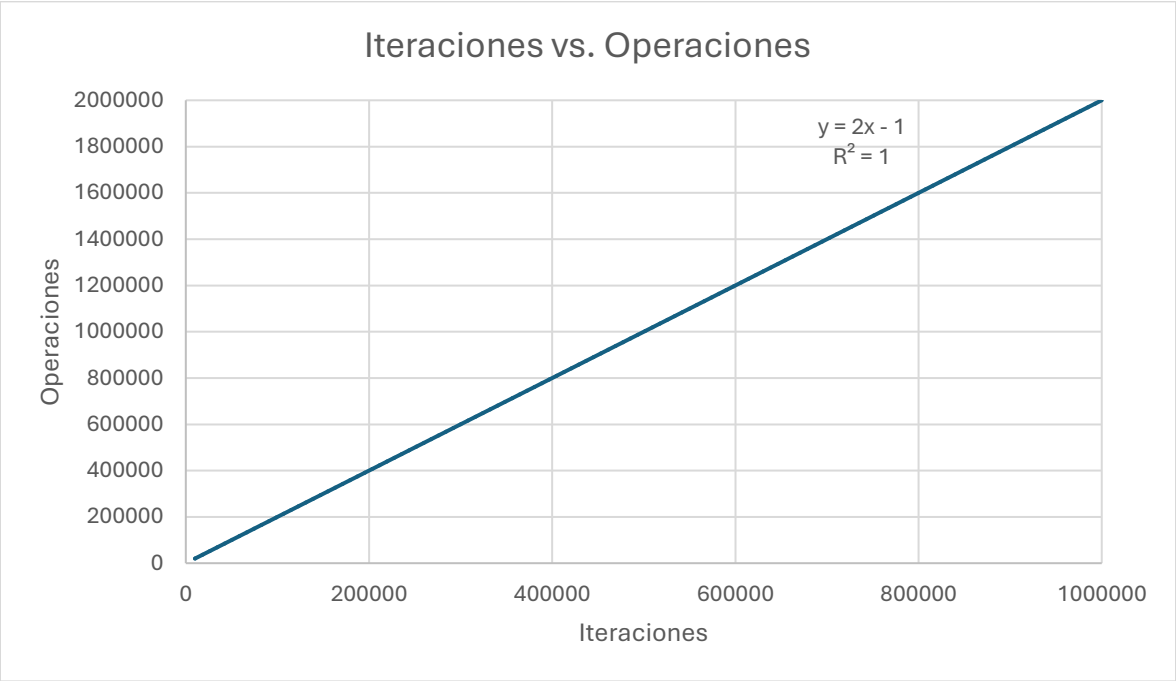


Figura 2. Gráfico de iteraciones contra operaciones.

Se puede apreciar en las graficas un comportamiento casi perfectamente lineal. Esto cuadra muy bien con el resultado cuasilineal que arrojó el análisis a priori. Un grupo más grande de puntos es posible que mostrara un comportamiento menos lineal en las gráficas.

En resumen

Se obtuvo el resultado esperado en el análisis a priori. De la misma forma, los resultados obtenidos fueron bastante buenos, demostrando la eficiencia del algoritmo MergeSort en este tipo de casos.

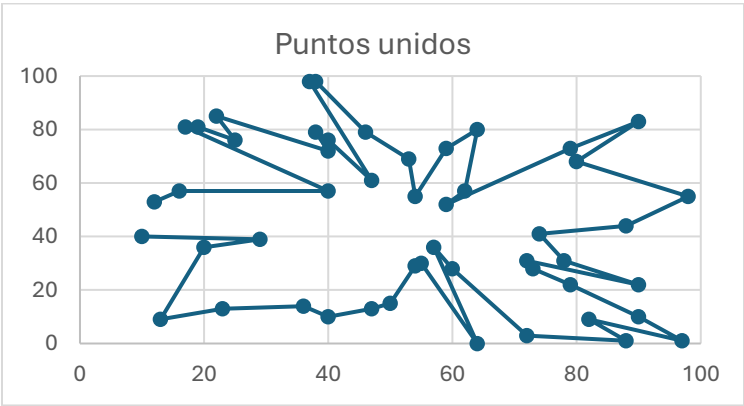


Figura 3. Caso promedio de curva formada.