c)

Debido al tamaño del código, no podemos adjuntarlo en este documento. Sin embargo, puede verificar el repositorio de esta entrega en el siguiente enlace para leer el código fuente:

[**https://github.com/Arcargotte/Examen-2-Lenguajes-de-Programacion**](https://github.com/Arcargotte/Examen-2-Lenguajes-de-Programacion)

En una primera instancia del desarrollo de este programa, se creó una versión sencilla del programa que utilizaba **BubbleSort** como algoritmo implícito de ordenamiento realizado durante cada repetición del iterador. En líneas generales, una vez recibida la lista por el iterador, éste iba por cada repetición a buscar el mínimo de la lista. Una vez encontrado, lo eliminaba de la misma y lo **yielda** al usuario, imprimiendo en la salida estándar el mínimo de la lista. Y así por cada uno de los elementos. Al final, le devolvía a uno los elementos de la lista en orden. Sin embargo, la complejidad de este algoritmo inicial era . Por este motivo se prefirió implementar otra versión que utilizara el principio de **MergeSort** para reducir considerablemente la complejidad del algoritmo resultante.

Lo que hace esta nueva versión del iterador, al que nos referiremos como **MergeSortIterator**, es crear un árbol binario cuyos nodos pueden interpretarse como la parte izquierda y la parte derecha del arreglo original. Dado un nodo, su hijo izquierdo correspondía a la parte izquierda del arreglo; su hijo derecho, el derecho. Así, por cada iteración, el algoritmo recorre el árbol binario buscando el elemento más pequeño en el subárbol izquierdo y el subárbol derecho desde la raíz. Una vez encontrados estos mínimos, se comparan entre sí, es retornado el menor y se actualiza el árbol eliminando el mínimo encontrado para así, en la siguiente iteración, buscar el siguiente mínimo.

Ilustramos el comportamiento de **MergeSortIterator** con el arreglo

[1 3 3 2 1]

Creamos el árbol, generando un nodo por cada sección de la lista: la lista completa, la parte izquierda y la parte derecha

[1 3 3 2 1]

[1 3 3]

[2 1]

[1 3]

[3]

[2]

[1]

[1]

[3]

Todo nodo hoja tiene un valor, que es el contenido del arreglo de tamaño 1. En cambio, todo nodo que no sea hoja tiene dos valores adicionales: el mínimo del árbol izquierdo y el mínimo del árbol derecho.

**min\_left = 1**

**min\_right = 1**

**min\_left = 1**

**min\_right = 3**

**min\_left = 1**

**min\_right = 3**

**value = 1**

**min\_left = 2**

**min\_right = 1**

**value = 3**

**value = 3**

**value = 2**

**value = 1**

[1 3 3 2 1]

[1 3 3]

[2 1]

[1 3]

[3]

[2]

[1]

[1]

[3]

Asimismo, implementamos dos métodos: **peek** y **pop**. **peek** se encarga de buscar recursivamente el mínimo del subárbol izquierdo y el mínimo del subárbol derecho. Es decir, busca el valor del nodo hoja en cada subárbol y determina si se trata de un valor mínimo. Dependiendo de dónde encontró el mínimo (si en el subárbol izquierdo o en el subárbol derecho), actualiza los valores **min\_left** y **min\_right**. **pop** se encarga de eliminar el mínimo del árbol. Hace esto buscando el valor mínimo del árbol, guardándolo y actualizando los valores del nodo donde lo encontró a **None**. Para hacer esto, es necesario que sepa en qué subárbol está el mínimo, y lo hace verificando en sí mismo si **min\_left** o **min\_right** son nulos. Si uno de ellos lo es, va a buscar recursivamente en el subárbol opuesto; si resulta que ninguno es nulo, simplemente compara cuál es el mínimo de ambos y busca en ese subárbol.

En el árbol que presentamos ilustra con verde los nodos hoja que son el mínimo del subárbol izquierdo y el subárbol derecho que, para la raíz, se tratan de los dos valores más pequeños del arreglo. El método **peek** se encarga de actualizar estos valores para cada nodo y en el iterador utilizamos el método **pop** para recuperar el mínimo y borrarlo del árbol.

class MergeSortIterator:

    def \_\_init\_\_(self, data):

        self.root = build\_tree(data)

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self):

        if self.root.peek() is None:

            raise StopIteration

        v = self.root.pop()

        return v

**Python**

Inicializamos el iterador cuando le pasamos un arreglo de enteros. Por definición, el método **\_\_iter\_\_** debe retornar a la instancia de la clase para cada iteración y el método **\_\_next\_\_** contiene la lógica de aquello que debe **retornar** el iterador por cada repetición. Aquí vemos que retorna el mínimo del arreglo, para luego eliminarlo y actualizarlo en la siguiente iteración.

Al final, obtenemos un iterador que retorna una lista ordenada con orden , que resulta en una solución preferible a la originalmente planteada.