Universidad Nacional de San Agustín, Escuela de Ingenieria de Sistemas

ADA - Lab 05

Fernando Enrique Araoz Morales - 20173373

1 de diciembre de 2019

1. Introducción

En este documento se presenta la implementación de la estructura Heap, MinHeap, MaxHeap, así como la comparación del tiempo de ejecución de este algoritmo contra Merge sort y Quick sort usando diferentes elementos (aleatorio, ordenado, inverso, únicos).

2. Heap

La estructura Heap es una estructura sencilla de implementar. Consta de un array de elementos, un método siftDown que se encarga de tomar el elemento raiz y reubicarlo en su posicion adecuada, un método heapify que se encarga de crear el Heap a partir de un array cualquiera, y un método sort que usa estos métodos para ordenar el array 'in-place'.
El código en sí se encuentra organizado de la siguiente manera: Una clase abstracta BinaryHeap<T>, esta representa un Heap cualquiera, y contiene toda la implementación necesaria, excepto un método compare(). Este método es el que define si es un Max o Min heap, por lo tanto las clases que lo hereden son responsables de implementarlo.
Las clases MaxBinaryHeap y MinBinaryHeap simplemente heredan de la clase BinaryHeap e implementan el método compare() según sea adecuado.

```
public abstract class BinaryHeap<T extends Comparable<T>> {

    private final T[] arr;
```

```
public BinaryHeap(T[] arr) {
    this.arr = arr;
}

private void swap(int pos1, int pos2) {
    T buffer = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = buffer;
}

void heapify() {

    int lastPos = arr.length;
    int firstParent = (lastPos % 2 == 0)? (lastPos - 2) / 2: (lastPos - 1)
        / 2;

    for (int i = firstParent; i >= 0; i--) {
        siftDown(i, arr.length);
    }

}

abstract boolean compare(T a, T b);

private void siftDown(int pos, int maxPos) {
    int largestPos = pos;
    int posLeft = pos * 2 + 1;
    int posRight = pos * 2 + 2;

    if (posLeft < maxPos && compare(arr[posLeft], arr[largestPos])) {
        largestPos = posLeft;
    }

    if (posRight < maxPos && compare(arr[posRight], arr[largestPos])) {
        largestPos = posRight;
    }


    if (largestPos != pos) {
        swap(pos, largestPos);

        siftDown(largestPos, maxPos);
    }

}

void sort() {
    this.heapify();
    for (int max = arr.length; max > 0; --max) {
```

```
            swap(0, max - 1);
            this.siftDown(0, max - 1);
        }
    }

    @Override
    public String toString() {
        StringBuilder res = new StringBuilder("[");
        for (T i: arr) {
            res.append(i).append(", ");
        }
        res.append("]");

        return res.toString();
    }

}
```

```
public class MaxBinaryHeap<T extends Comparable<T>> extends BinaryHeap<T> {


    public MaxBinaryHeap(T[] arr) {
        super(arr);
    }

    @Override
    boolean compare(T a, T b) {
        return a.compareTo(b) > 0;
    }

}
```

```
public class MinBinaryHeap<T extends Comparable<T>> extends BinaryHeap<T> {

    public MinBinaryHeap(T[] arr) {
        super(arr);
    }

    @Override
        boolean compare(T a, T b) {
        return a.compareTo(b) < 0;
    }

}
```

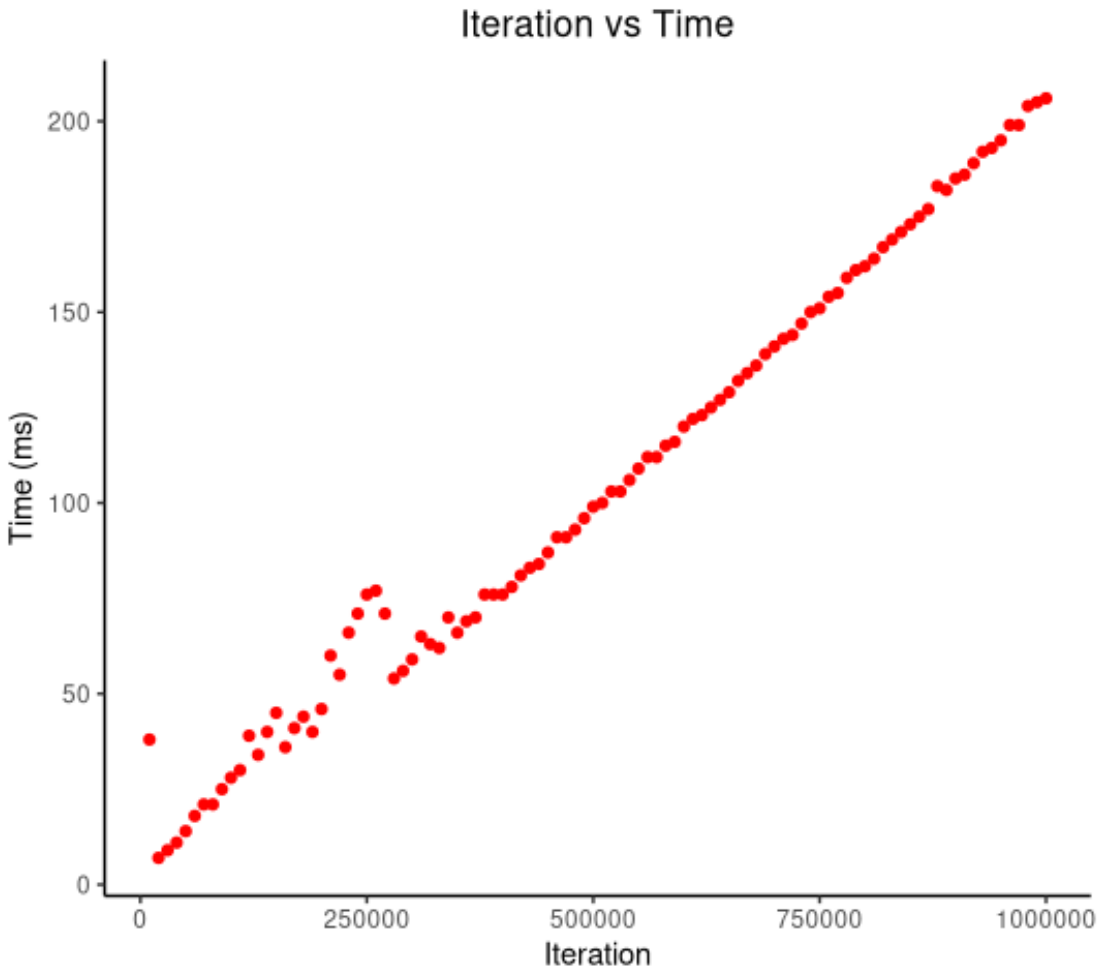

Figura 3.1: Heapsort con datos invertidos.

3. Comparaciones

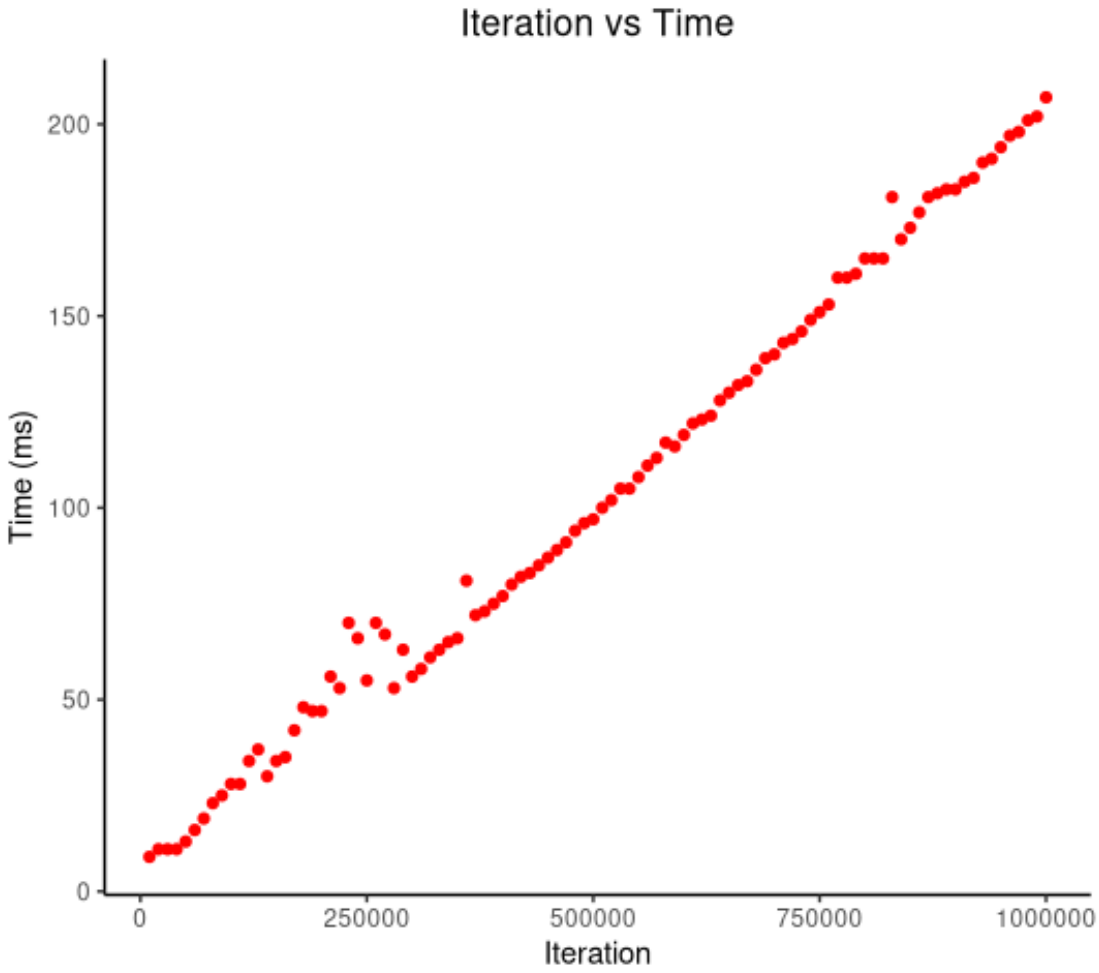

Figura 3.2: Heapsort con datos ordenados.

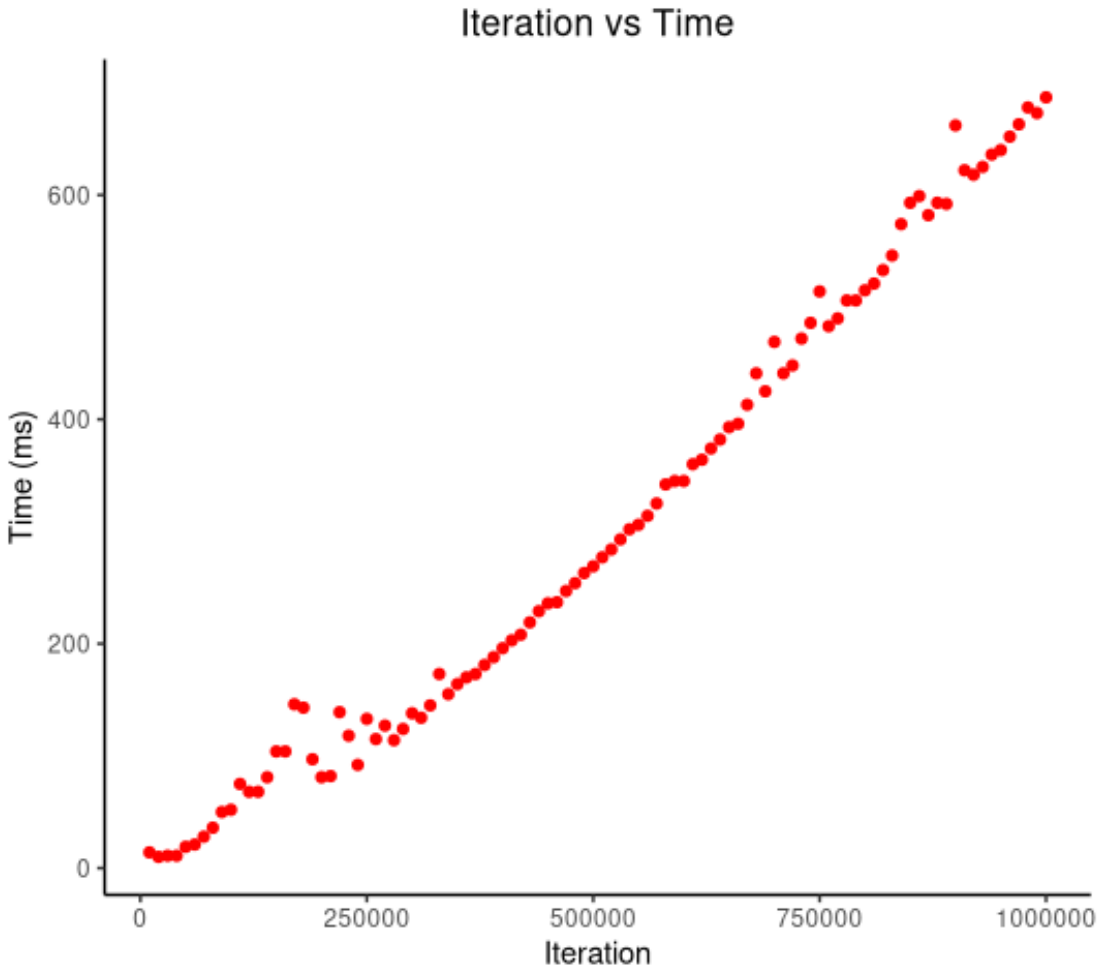

Figura 3.3: Heapsort con datos úticos.

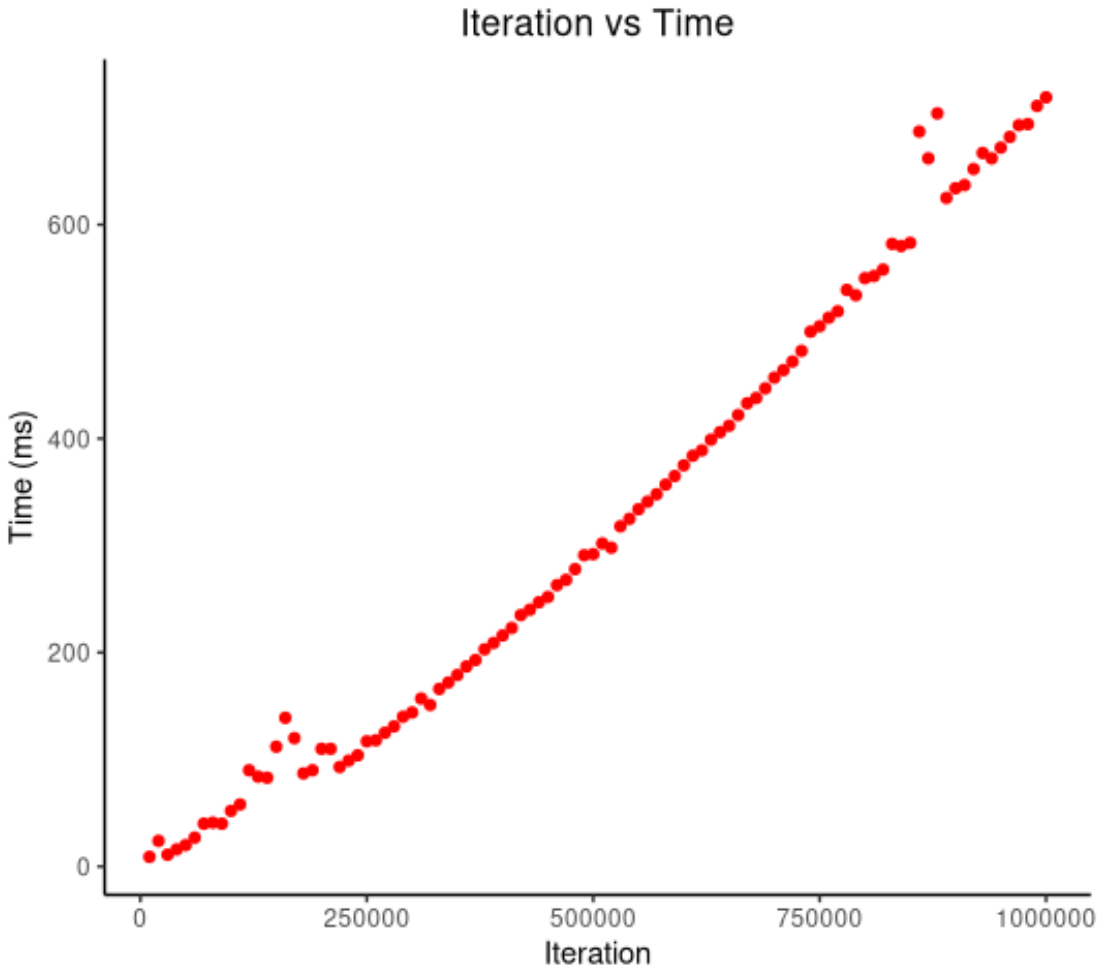

Figura 3.4: Heapsort con datos aleatorios.

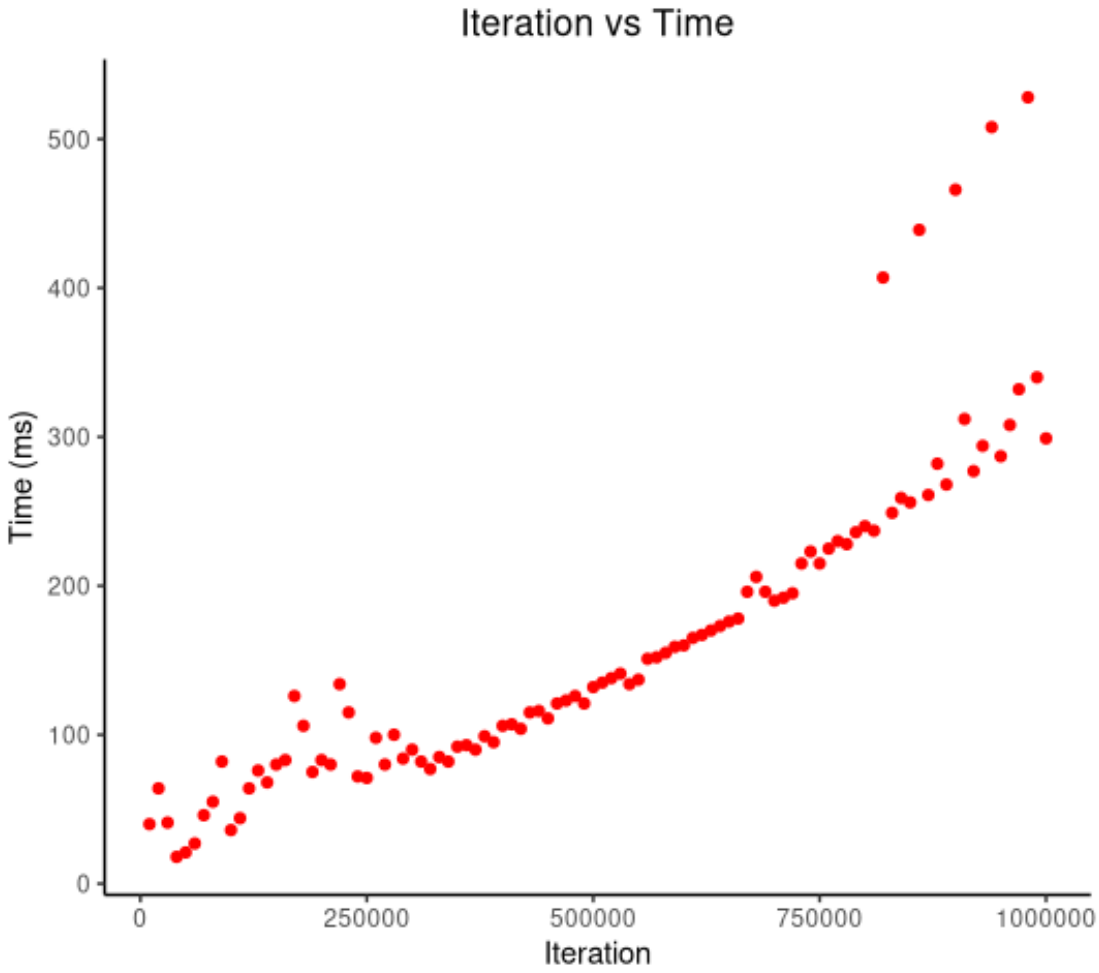

Figura 3.5: Mergesort con datos invertidos.

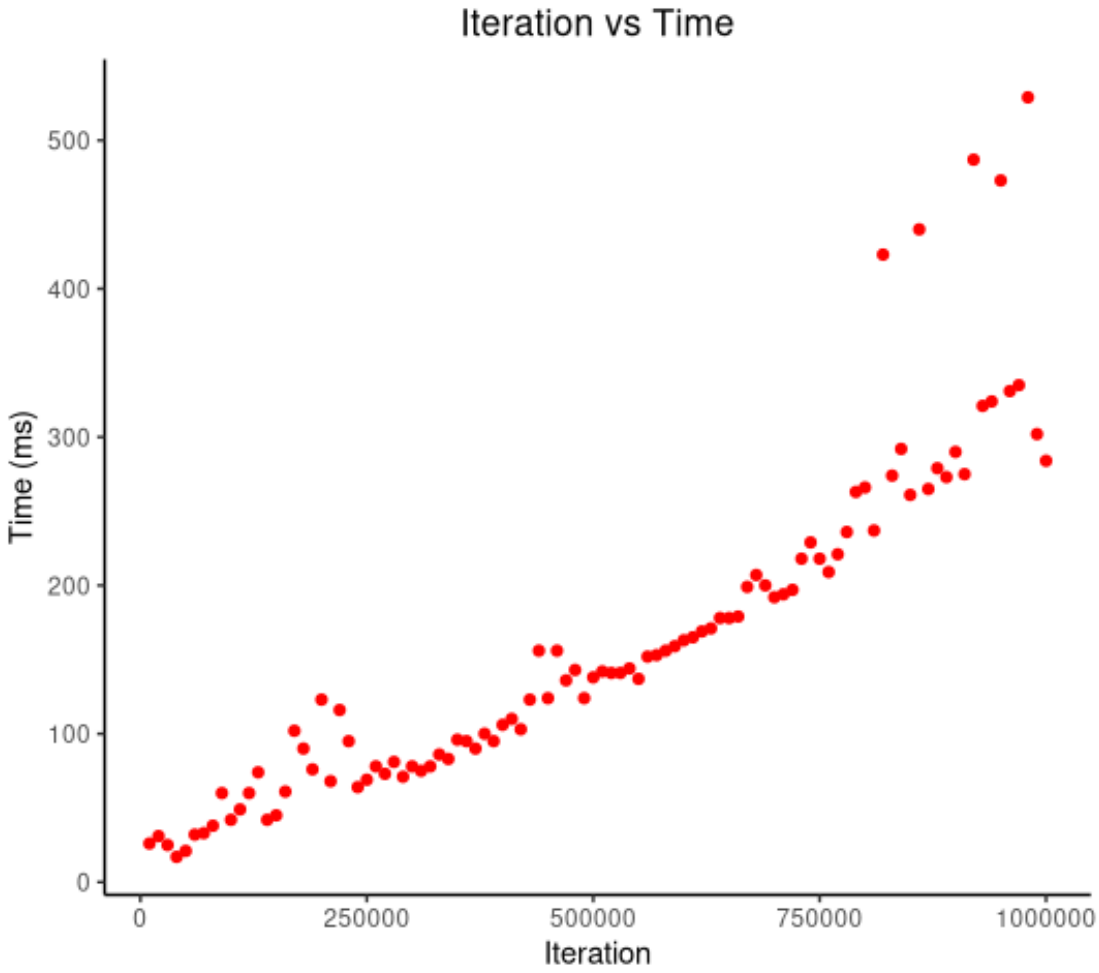

Figura 3.6: Mergesort con datos ordenados.

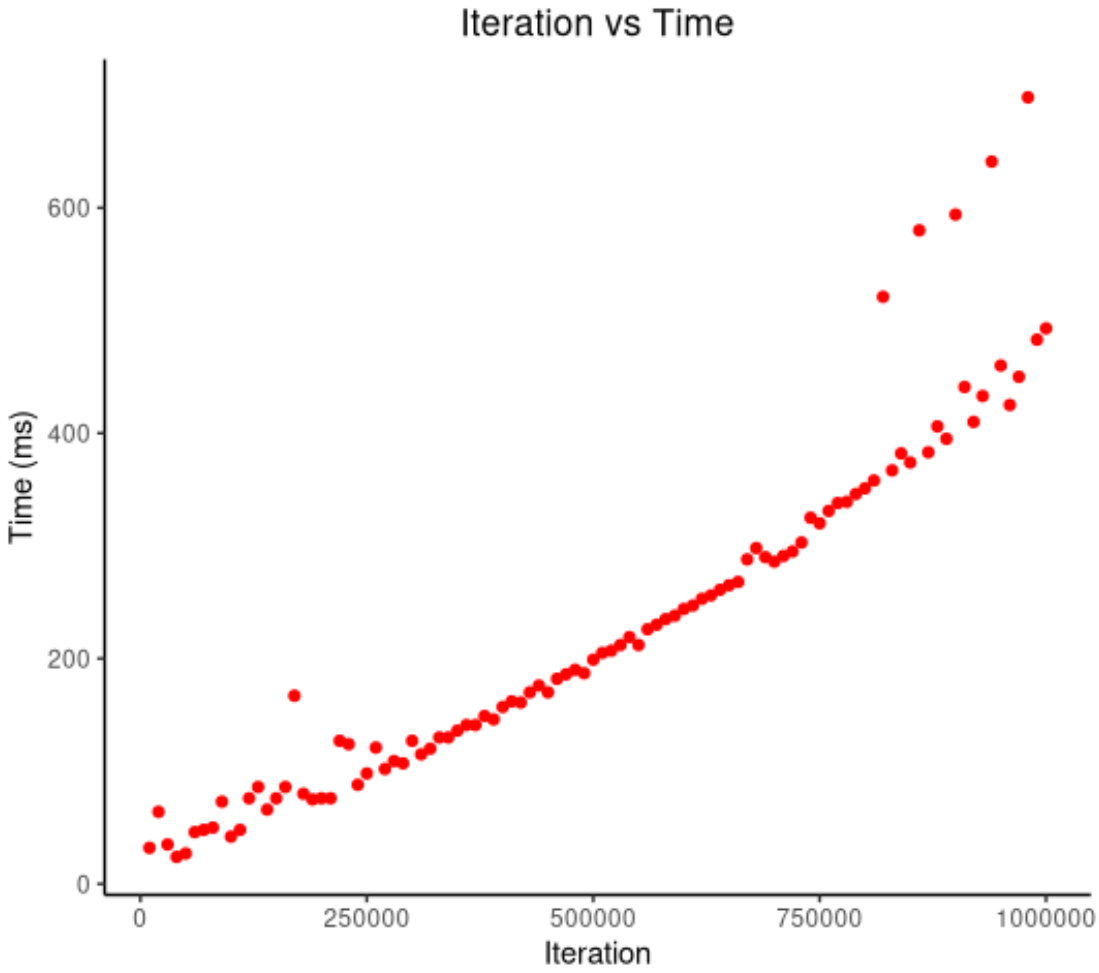

Figura 3.7: Mergesort con datos úticos.

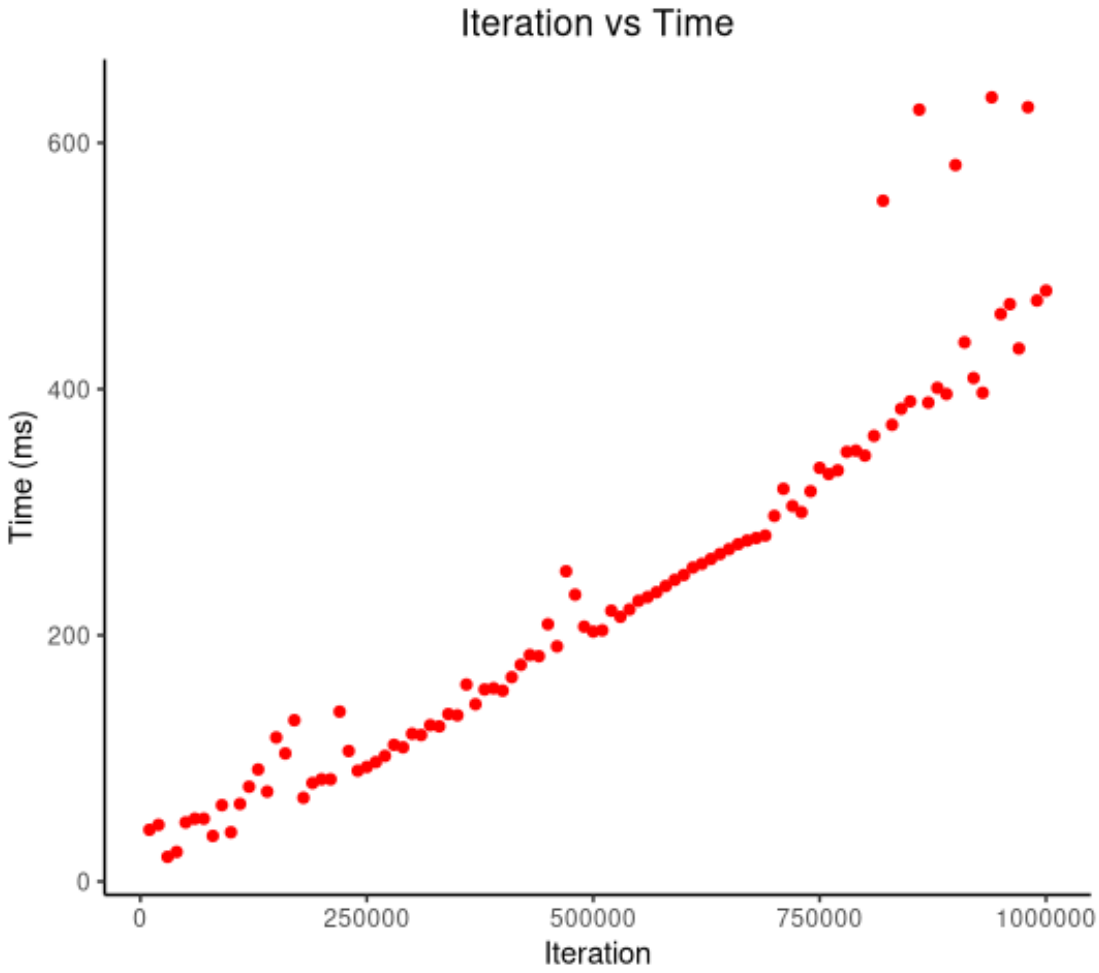

Figura 3.8: Mergesort con datos aleatorios.

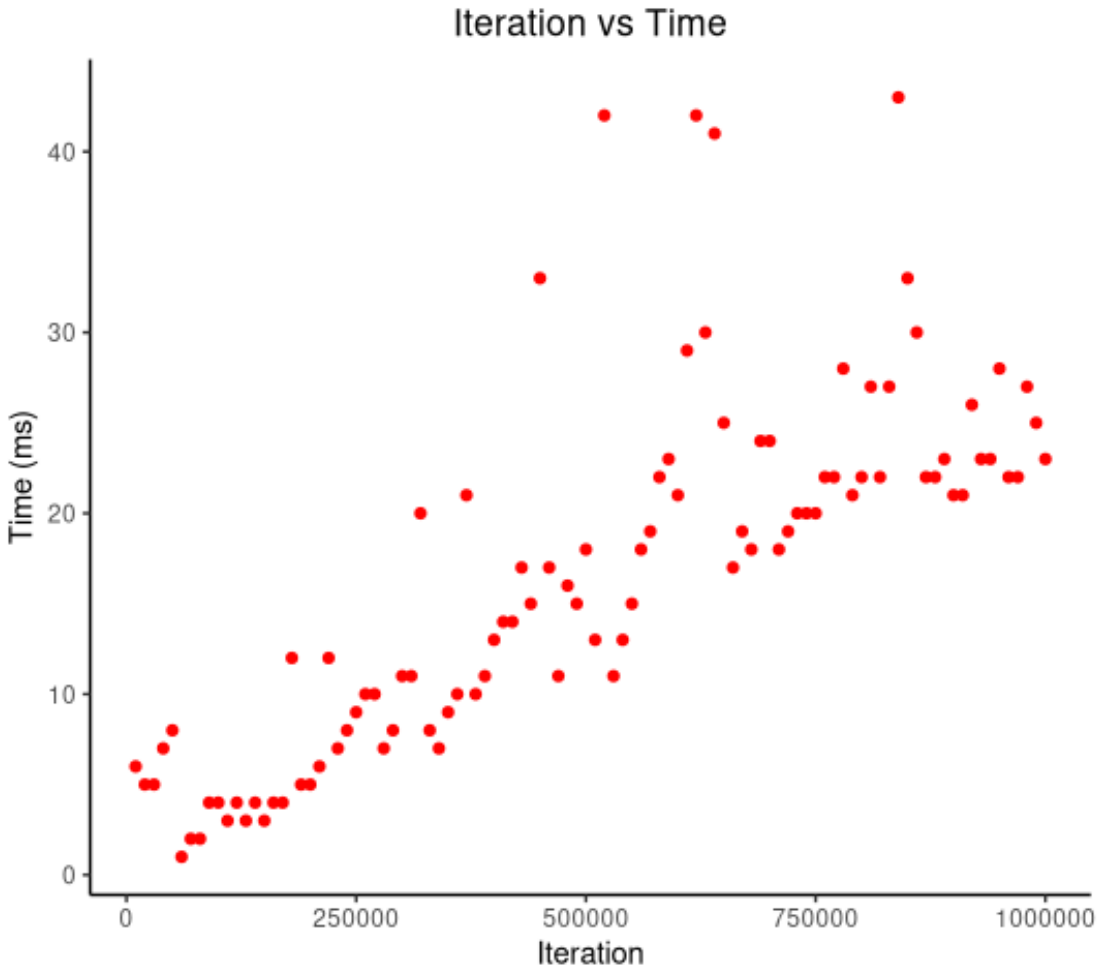

Figura 3.9: Quicksort con datos invertidos.

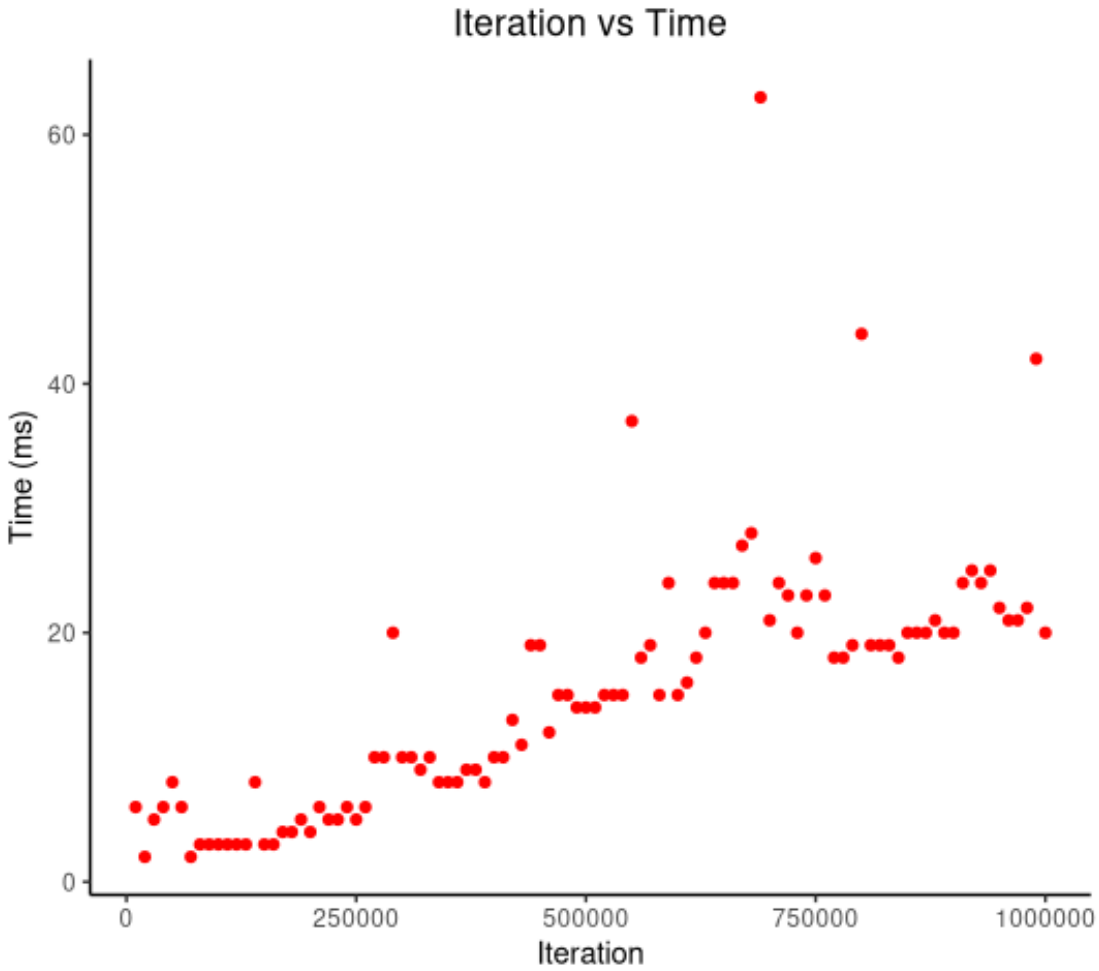

Figura 3.10: Quicksort con datos ordenados.

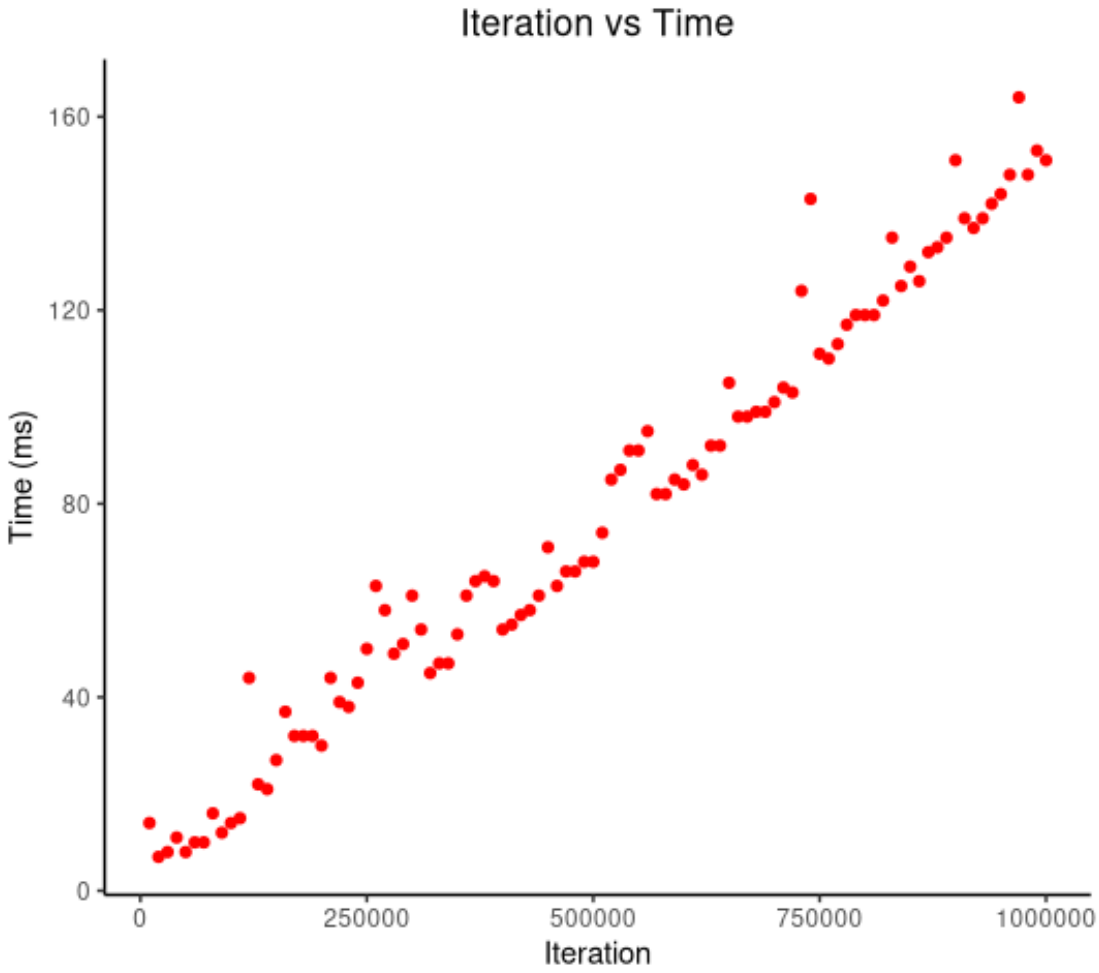

Figura 3.11: Quicksort con datos úticos.

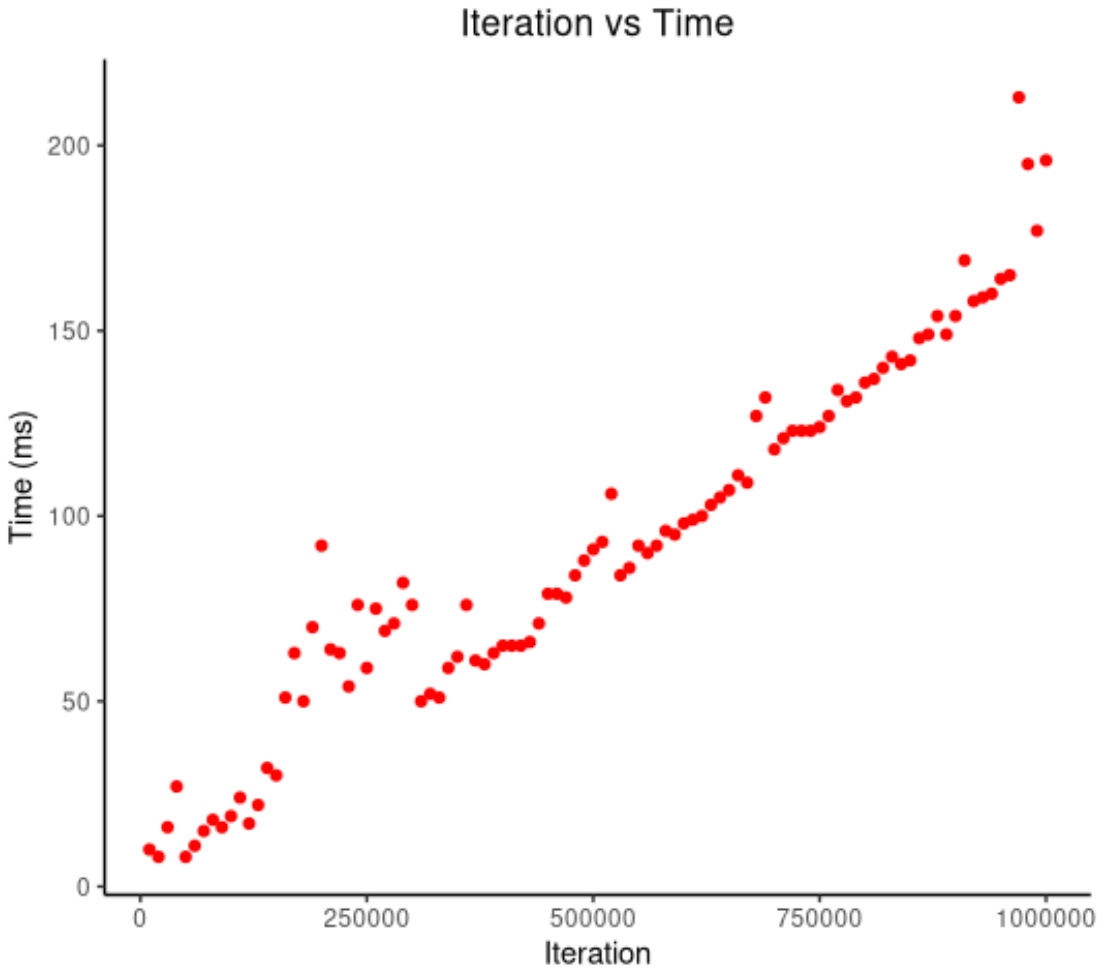

Figura 3.12: Quicksort con datos aleatorios.