Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC3001 – Algoritmos y Estructuras de Datos

Comparación de Algoritmos de Ordenación

TAREA 6

Nombre Alumno : Gabriel Iturra Bocaz

Correo : gabrieliturrab@ug.uchile.cl

Profesor : Patricio Poblete

Profesor Auxiliar : Manuel Cáceres
Sergio Peñafiel

Fecha : 10 de Mayo de 2013

Índice

I. Introducción

El siguiente informe detalla el trabajo de la tarea 6 del curso de Algoritmos y Estructuras de Datos (CC3001). Como sexta de tarea del curso, se pretende que el alumno se interiorice con la implementación de los algoritmos de ordenación Megesort y Quicksort, juntos a las mejoras de Quicksort (Mediana de 3 y Mediana de 3 e Insertionsort para arreglos pequeños).

El problema propuesto es implementar dos algoritmos de ordenación que Mergesort y Quicksort, considerandos tres formas de implementar Quicksort, que son, el Método Clásico, Mediana de 3 y Mediana de 3 e Insertionsort para arreglos pequeños, todos bajo una interface llamada Ordenacion, con un método obligatorio dentro de la interface void ordenar(int[] a), comparando cuál de estas implementaciones es la más eficiente (midiendo los tiempos de ejecución en cada uno) para cada caso.

Como estrategia de solución al desafío se utilizó los conceptos vistos en clases para la implementación de void ordenar(int[] a) en las distintas clases, también el material visto en clases auxiliares que fue subido a material docente en donde se basó la elaboración del código de los algoritmos. En cuanto al análisis de cada algoritmo se utilizaron arreglos para distintos tamaños n, donde $n \in \{ 10^6, 2\,10^6, 4\,10^6, 6\,10^6, 8\,10^6, 10^7, 2\,10^7, 4\,10^7, 6\,10^7, 8\,10^7, 10^8 \}$, y también se usó, la función permutación entregada en enunciado para generar los arreglos permutados, midiendo el tiempo en milisegundos con el métodos *System.currentTimeMillis()*.

II. Análisis del Problema

II.1. Descripción del problema

Como se mencionó anteriormente el objetivo de esta tarea es comparar los tiempos de comparación de los algoritmos de ordenación, en particular:

- Diferentes Algoritmos O(n log n).
- Diferentes variantes del algoritmo Quicksort.

Que son Mergesort y 3 variantes de Quicksort (Todos de orden O(n log n)), aplicados sobre arreglos de enteros de n elementos que contienen una permutación al azar {1, 2, …, n}, cuya implementación se explica a continuación.

II.1.1 Mergesort

Divide un arreglo en dos mitades, ordena recursivamente éstas mitades, y la combina haciendo un "merge" de estas.

II.1.2 Quicksort

Elige un elemento al azar del arreglo denominado "pivote", generando sub-arreglos de elementos menores al pivote y otro con elementos mayores al pivote proceso llamado particionar, y para ordenar los elementos al final. Las implementaciones que se utilizaron fueron las siguientes.

- *Método Clásico:* Para ordenar el arreglo se elige un "pivote" al azar poniendo los elementos menores al pivote a la izquierda del arreglo y los mayores a la derecha, ordenando recursivamente cada uno de los sub-arreglos generados.

- *Mediana de Tres:* Análogo al método clásico, con la excepción con que el "pivote" es escoge como la mediana de tres elementos escogidos al azar.

- *Mediana de Tres e Insertionsort:* Análogo al método mediana de tres, excepto que la recursión del algoritmo se detiene cuando se quiere ordenar un arreglo de tamaño menor a M (en esta tarea usaremos M = 10). Finalmente, como este algoritmo no deja ordenado el arreglo, al final se aplica el algoritmo InsertionSort.

II.2. Supuestos Importantes

- Para tener una mayor cantidad de datos se amplió el rango desde 10^6 a 10^8 para cada valor n.
- Los datos almacenados en los arreglos son de la clase Integer.
- Las permutaciones en cada caso y para cada valor de n, fue hecha con la función permutacion(int[] a) que fue entregada en el enunciado, por lo que los elementos de los arreglos eran siempre entregados al azar, sin casos especiales.
- No se consideraron caso bases como arreglos vacíos para implementación de los códigos, debido a que el foco de la tarea en la comparación de los algoritmos.

II.3. Implementación

Como ya se dijo el trabajo consistió en la implementación de los cuatro algoritmos que fueron explicados el punto anterior, las cuales debían seguir la siguiente interface que se puede ver en la figura 1.

```
1 public interface Ordenacion {
2   public void ordenar(int[] a);
3 }
```

Figura 1: Interface Ordenacion.

II.4. Experimentación

En la etapa de experimentación, se siguieron las siguientes características que se detallan a continuación:

- Solo se midió el tiempo de ordenamiento.
- Los largos de los arreglos, son n ϵ $\{ 10^6, 2\,10^6, 4\,10^6, 6\,10^6, 8\,10^6, 10^7, 2\,10^7, 4\,10^7, 6\,10^7, 8\,10^7, 10^8 \}$, y los tiempos se calcularon separadamente para cada valor de n.
- Para tener una mejor *performance* de los algoritmos, se establecieron estos criterios:

    1. Comportamiento asintótico de algoritmo.
    2. Que puedan diferenciar:
        - QuickSort de MergeSort
        - Las versiones de QuickSort.

III. Solución del Problema

III.1. Desarrollo

La implementación de los cuatro algoritmos es similar en cuanto al método que debió seguir, que es, void ordenar(int[] a), para evitar la extensión del informe, en ésta sección se explican las diferencias más importantes de Mergesort y Quicksort, y las de cada una de las implementaciones Quicksort.

Como se dijo en la sección pasada, cada algoritmo de ordenación debía seguir la interface, ya que en el método se implementa el algoritmo, llamándolos a todos de la misma forma, a continuación, se explica el detalle de cada algoritmo.

III.2. Mergesort

La implementación de éste algoritmo, consto de tres métodos que son merge, sort y ordenar, que se explican a continuación:

- *void merge(int[] arreglo, int inicio, int centro, int termino):* se encarga de separar el arreglo en dos, que son arregloIzq y arregloDer, y mediante dos ciclos for se insertar los elementos del arreglo original correspondientes, luego, los elementos dentro de cada arreglo son ordenados a través de 3 ciclo while mediante las comparaciones respectivas en cada caso, en la figura 2, se puede apreciar el código orden en el merge.

```
while (i < largo1){
   arreglo[k] = arregloIzq[i];
   ++i;
   ++k;
}
```

Figura 2: Comparación de elementos.

- *void sort(int[] arreglo, int inicio, int termino):* Este método, se encarga de realizar la llamada recursiva a la primera y segunda mitad del arreglo, además de llamar al merge para ordenar los elementos, mientras el inicio a ordenar sea menor al termino, en la figura 3 se aprecia este método.

```
public static void sort(int[] arreglo, int inicio, int termino){
   if (inicio < termino){
      int medio = (inicio + termino)/2;
      sort(arreglo, inicio, medio);
      sort(arreglo, medio + 1, termino);

      merge(arreglo, inicio, medio, termino);
   }
```

Figura 3: Mergesort

- void ordenar(int[] a): Es método llama al método sort, como inicio el 0 y de termino el largo del arreglo menos 1.

III.3. QuickSort

La implementación de Quicksort consto de tres formas, que se basaron en los código de la clase auxiliar número 10, realizando los cambios respetivos a cada implementación, los detalles se explican en los puntos siguientes.

III.3.1 Método Clásico

Como ya se dijo, está implementación se basó en el código del auxiliar números 10, utilizando cuatro métodos inportantes, que fueron:

- void swap(int[] a, int i, int j): Intercambia el elemento de la posición i a la posición j.
- int particionar(int[] a, int lo, int hi): devuelve el índice del pivote, una vez que los elementos menores al pivote se encuentran a su izquierda, y los mayores a la derecha. Dentro del arreglo se elige un elemento al azar, utilizando la clase Random predefinida en JAVA, luego se un “swap” entre el elemento que está al inicio con el pivote, para que este quede al inicio, ya que mediante un ciclo while, se recorre el arreglo, primero al inicio del arreglo hasta encontrar un elemento mayor al pivote, y luego se recorre al final de arreglo hasta encontrar un elemento mayor, y luego se realiza un swap con el elemento menor y mayor al pivote, y así sucesivamente hasta que la diferencia entre hi y lo es un número negativo. En figura 4, se puede apreciar un estracto de este código.

```
//Eleccion del pivote
int ip = new Random().nextInt(hi-lo+1) + lo;
swap(a,lo,ip);

int p = a[lo];
while (true) {
   // Avanzar i
   while (a[++i] < p)
      if (i == hi) break;
```

Figura 4: Método particionar.

- void QuickSortMC(int[] a, int lo, int hi): Realiza la llamada recursiva, a la mitad del arreglo justo antes del pivote y a la mitad después, justo antes de particionar el arreglo.

```
static void QuicksortMC(int[] a, int lo, int hi) {
    if (hi <= lo) return;
    int k = particionar(a, lo, hi);
    QuicksortMC(a, lo, k-1);
    QuicksortMC(a, k+1, hi);
```

Figura 5: Quicksort Método Clásico.

- void ordenar (int[] a): análogo al método de igual nombre en Mergesort.

III.3.2 Mediana de Tres

Su implementación es análoga a la de Quicksort en su método clásico, pero se agrega un método adicional para calcular la mediana de un muestreo de tres elementos del arreglo.

- int medianaDe3(int[] a, int lo, int hi): Como se toma el supuesto de que los arreglos eran entregados con los elementos al azar. Se elige el elemento del dentro como lo + hi / 2 definido en una variable ce, realizándose un swap, sí el elemento del inicio es mayor el elemento del final, retornando finalmente el índice del penúltimo de arreglo justo antes de realizar un último swap del elemento central y el penúltimo. En la figura 6, se aprecia el método fue descrito.

```
public static int medianaDe3(int[] a, int lo, int hi) {
    int ce = (lo + hi) / 2;

    if (a[lo] > a[ce])
      swap(a, lo, hi);
```

```
swap(a, ce, hi - 1);
return hi - 1;
```

Figura 6: medianade3

****El resto de la implementación es análoga al Método Clásico.*

III.3.3 Mediana de Tres e InsertionSort

La implementación es análoga a Mediana de Tres, pero cuando la cantidad de elementos a ordenar es menor a cierto M (en la tarea se usó M = 10), en lugar de ordenar con Mediana de 3, se utiliza el algoritmo de ordenación InsertionSort, que se explica en el punto siguiente.

- InsertionSort: Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay ***k*** elementos ordenados de menor a mayor, se toma el elemento ***k+1*** y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se *inserta* el elemento ***k+1*** debiendo desplazarse los demás elementos. En la figura 7, se puede ver el código de este algoritmo.

```
private static void insertionSort( int[ ] a, int low, int high ) {
  for( int p = low + 1; p <= high; p++ ) {
    int tmp = a[ p ];
    int j;

    for( j = p; j > low && less(a[p], a[j - 1]); j-- )
      a[ j ] = a[ j - 1 ];
    a[ j ] = tmp;
```

Figura 7: InsertionSort.

- void sort(int[] a, int lo, int hi): Se añade una condición para ordenar los arreglos con insertionSort, en el cual hi – lo + 1 es mejor a M, se ordenar por insertionSort.

```
int M = 10;
if (hi - lo + 1 < M){
   insertionSort(a, lo, hi);
```

Figura 8: Condición.

****El resto de la implementación es igual a la de Mediana de Tres.*

III.4. Toma de datos

La toma de datos se hizo separadamente para cada valor de n y algoritmo, definiendo un objeto que dependía del algoritmo, y se llamaba al método ordenar para utilizar el algoritmo respectivo, en la figura 9, se detalla un ejemplo de esto.

```
static public void main(String[] args){
    int[] a = permutacion((int) Math.pow(10,6));
    Ordenacion m = new MedianaDeTresEInsertion();
    long start = System.currentTimeMillis();
    m.ordenar(a);
    long finish = System.currentTimeMillis() - start;
    System.out.println("El algoritmo demora "+finish+" milisegundos");
    //System.out.println(m.print());
```

Figura 9: Toma de datos.

Una vez finalizado la ejecución del programa, se generaba un print como el de la figura 10.

```
El algoritmo demora 147 milisegundos
```

Figura 10: Tiempo de ejecutado.

IV. Resultados y Conclusiones

En ésta última sección se explica el detalle de la toma de datos, y los resultados en bases a los tiempos de ejecución de las distintas implementaciones de los algoritmos. El tiempo de ejecución fue medido usando el método *System.currentTimeMillis()*, por lo que la unidad de tiempo utilizada fueron los milisegundos [ms], los arreglos "desordenados" eran generados por la función *permutacion(int n)*, donde n, determina el largo del arreglo (función entregada por el enunciado).

Se cuidó que los resultados midieran solo los tiempos de ordenamiento, comenzando el punto 0, cuando terminaba de retornarse los arreglos desordenados. Luego se midió para cada valor n con los distintos algoritmos, como se mostró en la sección anterior, los datos obtenidos pueden verse en la tabla 1.

Cantidad de llaves	Mergesort Tiempo [ms]	Quicksort (Metodo clásico)	Mediana de Tres Tiempo [ms]	Mediana de 3 e InsertionSort
10^6	266	224	175	144
$2\ 10^6$	533	443	312	304
$4\ 10^6$	1042	826	613	606
$6\ 10^6$	1597	1252	953	918
$8\ 10^6$	2181	1747	1207	1169
10^7	2688	2165	1560	1436
$2\ 10^7$	5553	4300	3184	3121

$4\ 10^7$	11489	8932	6677	6592
$6\ 10^7$	17757	12745	10911	9857
$8\ 10^7$	23838	18532	13696	13033
10^8	29694	20844	17931	16910

Tabla 1: Tiempos de ejecución.

IV.1. Comparación MergeSort vs QuickSort

De acuerdo a los datos, se puede apreciar que el algoritmo que demora menos en ordenar los elementos es Mediana de Tres e InsertionSort, seguido de Mediana de Tres, QuickSort Mediana de Tres e InsertionSortMétodo Clásico, y en último lugar MergeSort. Sin embargo, para visualizar una mejor comparación entre algoritmos de ordenamiento se graficaron los datos de MergeSort y QuickSort (Método Clásico) la diferencia puede verse en el gráfico 1.

****Es importante mencionar que se utilizó una escala logarítmica en el eje x de los datos graficados****

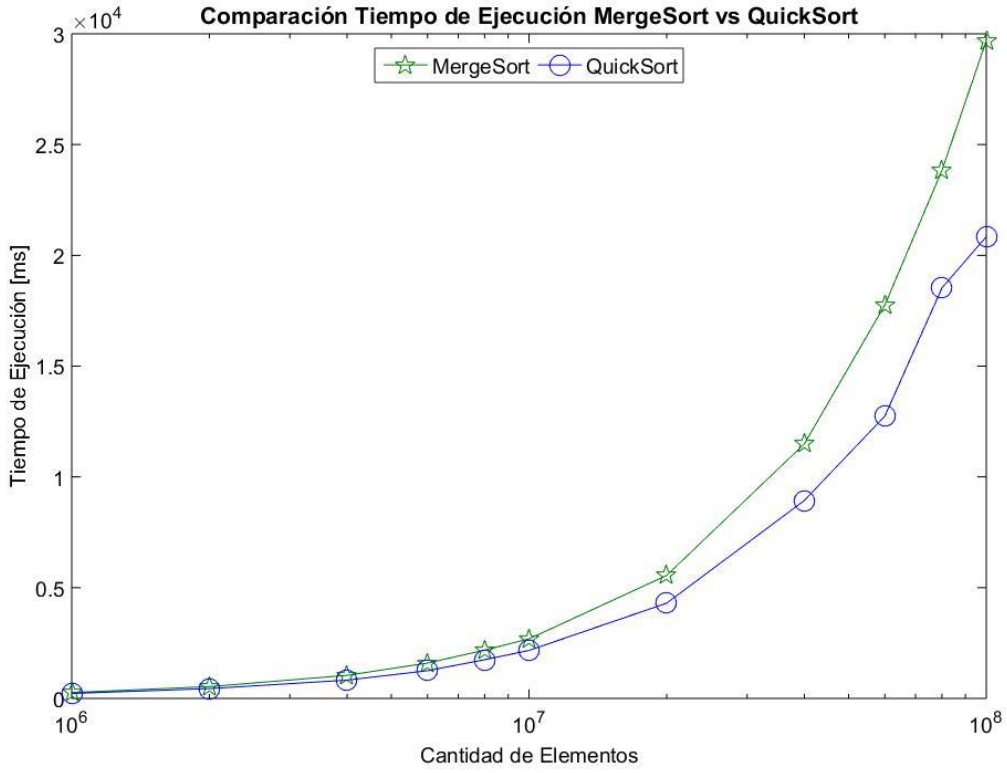

Gráfico 1: MergeSort vs QuickSort

De acuerdo al gráfico 1, a pesar de que los algoritmos O(n log n) la implementación de QuickSort (línea azul) es más eficiente y demora menos que la de MergSort (línea verde).

IV.2. Comparación Implementaciones de QuickSort

Como los algoritmos Mediana de Tres y Mediana de Tres e InsertioSort son versiones más óptimas de QuickSort que mejoran el tiempo de ejecución, pero ¿Cuál de ellas es mejor?, la respuesta se puso obtener gracias a los datos que se pueden ver claramente en el gráfico 2, donde:

- El método Clásico se representa en la línea azul.
- Mediana de Tres por la línea roja.
- Y Mediana de Tres e InsertionSort por la línea Negra.

Sin embargo, como las gráficas de las mejoras de QuickSort son muy parecidas en este muestreo de datos, se midieron comparaciones por separadas de cada una estas con el método clásico, y una comparación entre las dos implementaciones, como se puede ver en los gráficos 3, 4 y 5.

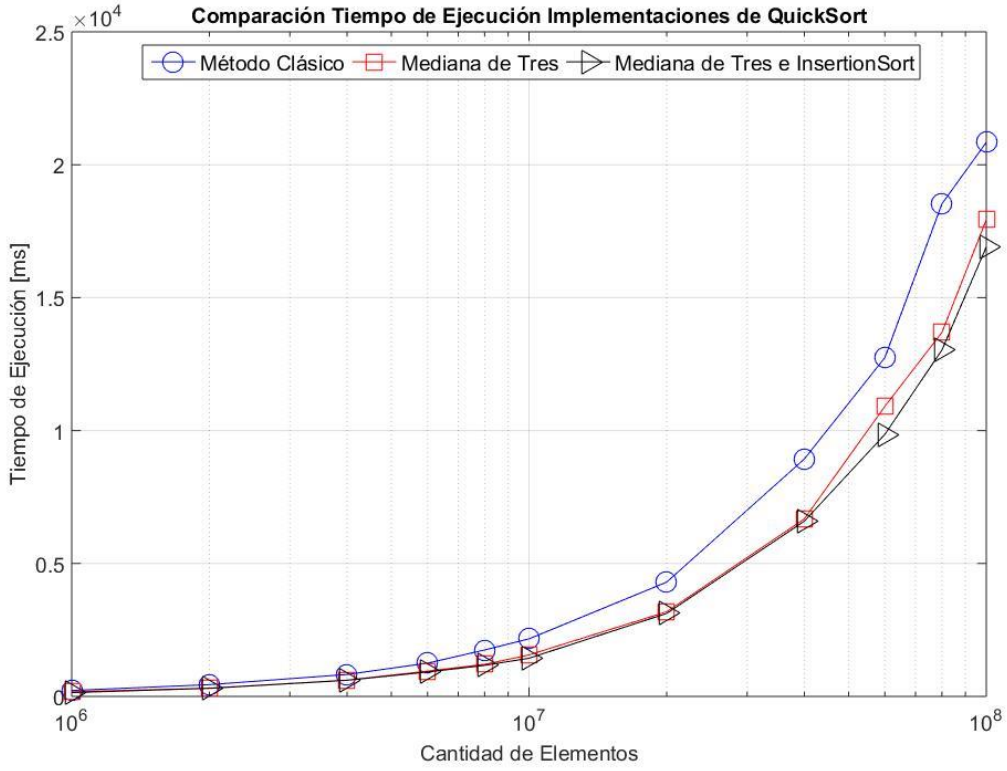

Gráfico 3: Implementaciones de QuickSort.

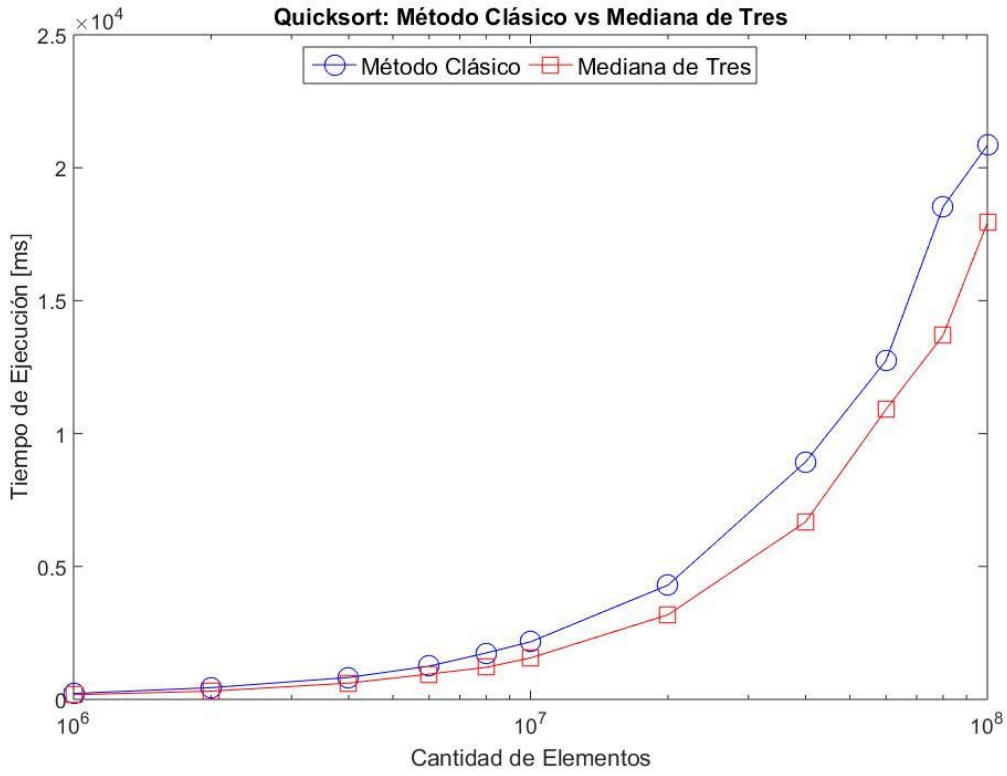

Gráfico 3: Método Clásico vs Mediana de Tres.

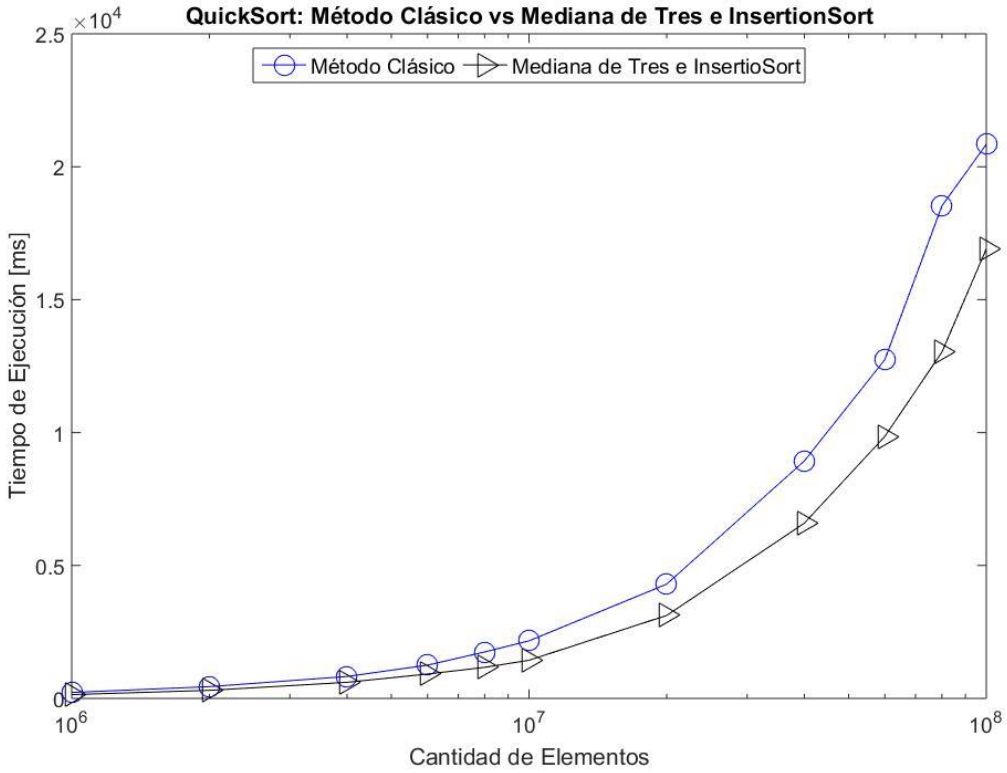

Gráfico 4: Método Clásico vs Mediana de Tres e InsertionSort.

De acuerdo a los graficamos anteriores se muestra claramente que las mejoras de QuickSort también tiene complejidad O(n log n) y demoran menos tiempo que la implementación clásica.

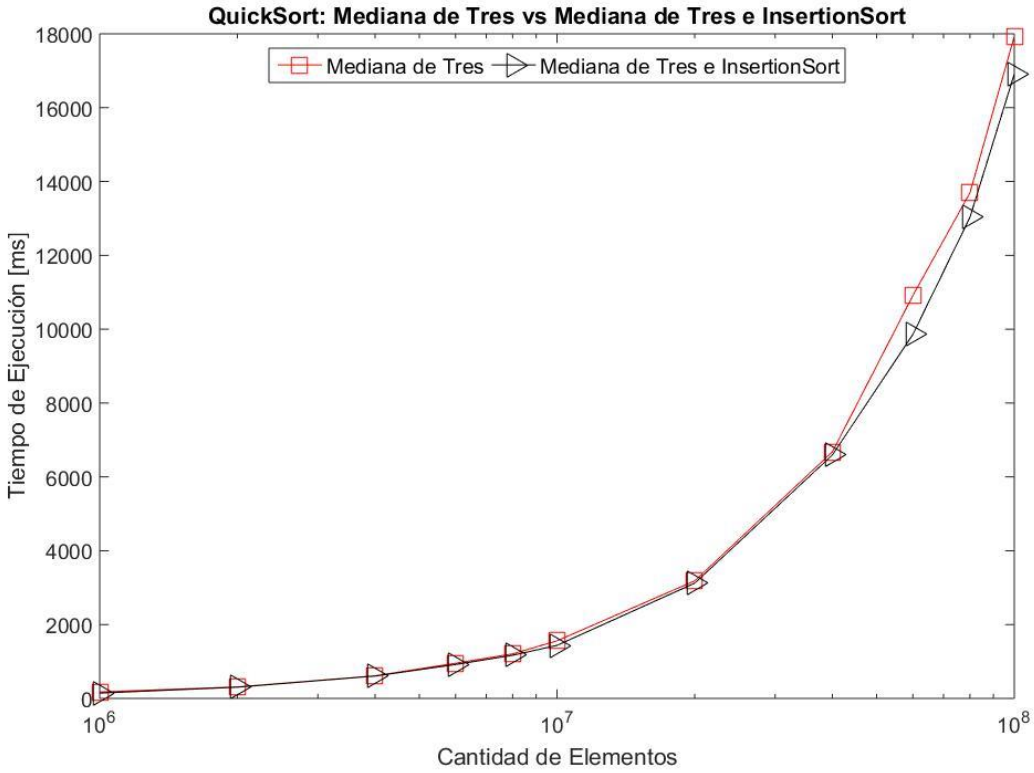

Gráfico 5: Mediana de Tres vs Mediana de Tres e InsertionSort.

Finalmente, en el último gráfico se la implementación Mediana de Tres e InsertionSort, demora un menor tiempo a medida que el arreglo debe ordenar elementos mayores a 10^7 cuando se comienza a llegar a 10^8 es más visible la eficiencia de está en implementación en comparación a Mediana de Tres, aunque es ligeramente menor.