**Universidad Simón Bolívar**
**Departamento de Computación y Tecnología de la Información**
**CI-2692 - Laboratorio de Algoritmos y Estructuras II**
**Trimestre Enero-Marzo 2016**

# Estudio experimental de algoritmos de ordenamiento

Alumnos: Edymar Mijares 12-10882
José Carmona 11-10156

## Introducción

El siguiente informe tiene como finalidad expresar los resultados del estudio de algoritmos de ordenamientos sobre distintos conjuntos de datos . Se estudiaron varias versiones del algoritmo Quicksort con distintas características, y se compararan con Mergesort iterativo y Heapsort. En el informe se explicara brevemente el funcionamiento de cada algoritmo y su tiempo teórico de corrida, se daran detalles sobre las implementaciones de prueba realizadas para estudiar los algoritmos así como los resultados de las mismas y se concluirá con el análisis de los casos de estudio.

*"La verdad necesita del poder. --En sí misma, la verdad no es una potencia, digan lo que digan los retóricos racionalistas. Por el contrario, necesita ponerse de su parte a la fuerza o ponerse ella del lado de la fuerza, pues de lo contrario perecerá siempre. Es cosa demostrada hasta la saciedad. "*
*Aurora, Federico Nietzsche.*

## Algoritmos Estudiados

**Mergesort**

Este algoritmo utiliza un enfoque de dividir y conquistar; divide el problema a resolver en subproblemas mas pequeños soluciona los subproblemas de forma recursiva y combinan las soluciones de los subproblemas en una solución al problema original. Nuestra versión de Mergesort, sin embargo , es una versión iterativa extraída del libro de Kaldewaij el cual contiene dos lazos while iterados que a su vez contienen un ciclo for y 4 ciclos while que en teoría deberían generar un tiempo de O(n log n).

El comportamiento del algoritmo depende directamente del tamaño de la entrada resultando ser tiempo O(n log n) para cualquier entrada.

**Heapsort**

Heapsort es un algoritmo de ordenamiento basado en comparaciones que corre en tiempo T (n) = Θ(n log n) para arreglos de n elementos. Es una algoritmo recursivo que depende de dos funciones importantes; Max-Heapify, funcion clave para mantener la propiedad de max-heap (corre en tiempo O(log n)), y Build-Max-Heap,construye un heap binario a partir de un arreglo en tiempo O(n) así Heapsort ordena un arreglo de n elementos “in-place” en tiempo O(n log n).

El comportamiento de este algoritmo depende el tipo de arreglo de entrada, el algoritmo es mas eficiente con arreglos desordenados, en cambio presenta el peor caso para arreglos ordenados y es muy poco eficiente para arreglos con elementos repetidos ya que no es un algoritmo estable.

**Quizksort randomizado**

al igual que el Mergesort es un algoritmo recursivo del tipo dividir y conquistar su rutina clave es el partition que corre en tiempo Θ(n) . El tiempo esperado de Quicksort es de O(n log n) y Θ( $n^2$ ) en el peor caso, el cual es muy poco probable debido que el pivote a utilizar en cada llamada a Partition se escoge de forma aleatoria.

El rendimiento del Quicksort depende de que tan buenas resultan las divisiones que crea el pivote sobre el arreglo de entrada,escoger el pivote al azar reduce la probabilidad de tener el peor caso. Si el pivote que genera el partition es un numero del arreglo por debajo del la media entre los elementos del arreglo va a dividir el arreglo de forma desigual, como la llamada a random escoge un elemento al azar los casos en los que es mas probable que esa elección sea desfavorable seria en el caso de los arreglos ordenados.

Las variaciones del Quicksort que vamos a estudiar son **Median-of-3 quicksort**, **Introsort** , **Quicksort with 3-way partitioning** y **Dual pivot Quicksort** tienen tiempo O(n log n) en el peor caso de igual forma su eficiencia depende de la escogencia del pivote.

## Experimentos

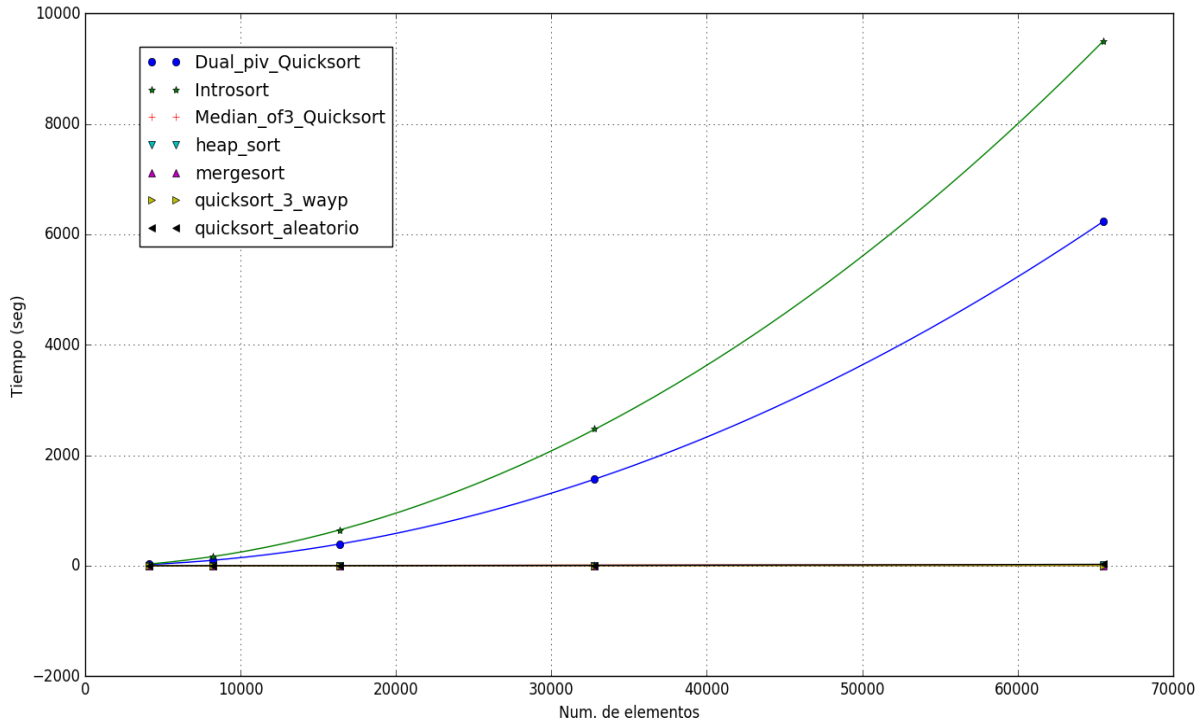
Los siguientes experimentos fueron realizados en Los siguientes equipos:
P1: SO Ubuntu 16.04 LTS, 64bits
Procesador: Intel® Pentium(R) CPU G860 @ 3.00GHz × 2.
RAM: 7.8 GiB.
P2: SO Ubuntu 16.04 LTS, 64bits
Procesador: Intel® Pentium(R) Dual CPU E2180@ 2.00GHz x 2.
RAM: 3,8 GiB.
P3: SO Ubuntu 14.04 LTS, 64bits
Procesador: Intel® Core™ i3 CPU M 370 @ 2.40GHz × 4.
RAM: 2,7 GiB.
P4: SO Ubuntu 14.04 LTS, 64bits
Procesador: Intel® Core™ i3 CPU M 370 @ 2.40GHz × 4.
RAM: 2,7 GiB.
P5: SO Ubuntu 16.04 LTS, 64bits
Procesador: Intel® Pentium(R) CPU G860 @ 3.00GHz × 2.
RAM: 7.8 GiB.
P6: SO Ubuntu 14.04 LTS, 64bits
Procesador: Intel® Core™ i3 CPU M 370 @ 2.40GHz × 4.
RAM: 2,7 GiB.
P7: SO Ubuntu 16.04 LTS, 64bits
Procesador: Intel® Pentium(R) CPU G860 @ 3.00GHz × 2.
RAM: 7.8 GiB**.**

## Resultados de los experimentos (expresados en segundos)

**Llamada #1**
./cliente_ordenamiento.py -g -p 1 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:**Enteros aleatorios

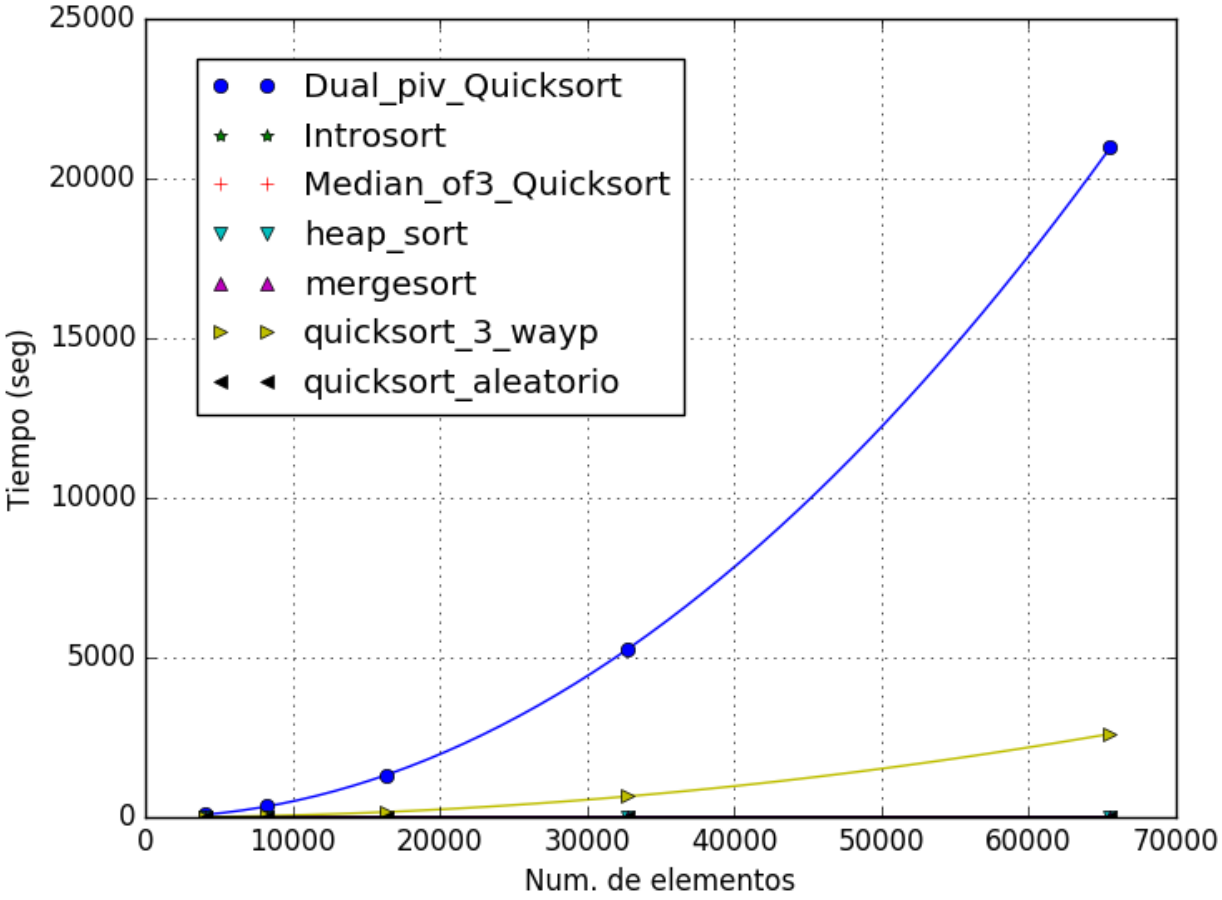
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.230 | 0.353 | 0.268 | 0.203 | 39.038 | 24.534 | 0.147 |
| **8192** | 0.482 | 0.765 | 0.756 | 0.944 | 159.715 | 98.905 | 0.248 |
| **16384** | 1.048 | 1.676 | 2.124 | 2.899 | 637.676 | 385.810 | 0.497 |
| **32768** | 2.254 | 3.611 | 6.908 | 8.068 | 2474.690 | 1569.829 | 0.973 |
| **65536** | 4.813 | 7.780 | 24.272 | 16.222 | 9502.261 | 6234.522 | 1.955 |




**Llamada #2**
./cliente_ordenamiento.py -g -p 2 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:**Orden inverso

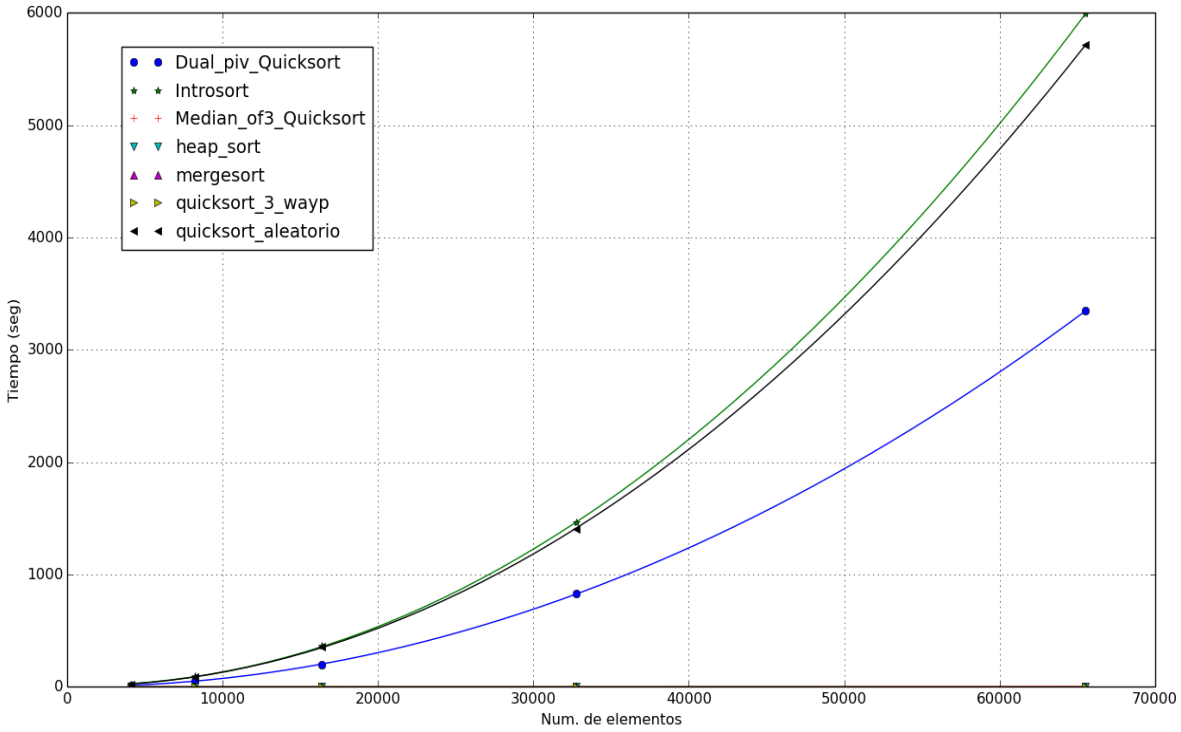
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.333 | 0.549 | 0.304 | 0.094 | 0.095 | 81.235 | 10.440 |
| **8192** | 0.718 | 1.226 | 0.758 | 0.195 | 0.195 | 340.506 | 41,486 |
| **16384** | 1.552 | 2.650 | 1.535 | 0.411 | 0.408 | 1310.417 | 164.168 |
| **32768** | 3.335 | 5.730 | 3.157 | 0.856 | 0.850 | 5272.375 | 653.066 |
| **65536** | 7.048 | 12.299 | 7.322 | 1.775 | 1.781 | 20965.157 | 2609.703 |




**Llamada #3**
./cliente_ordenamiento.py -g -p 3 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:** Cero-uno

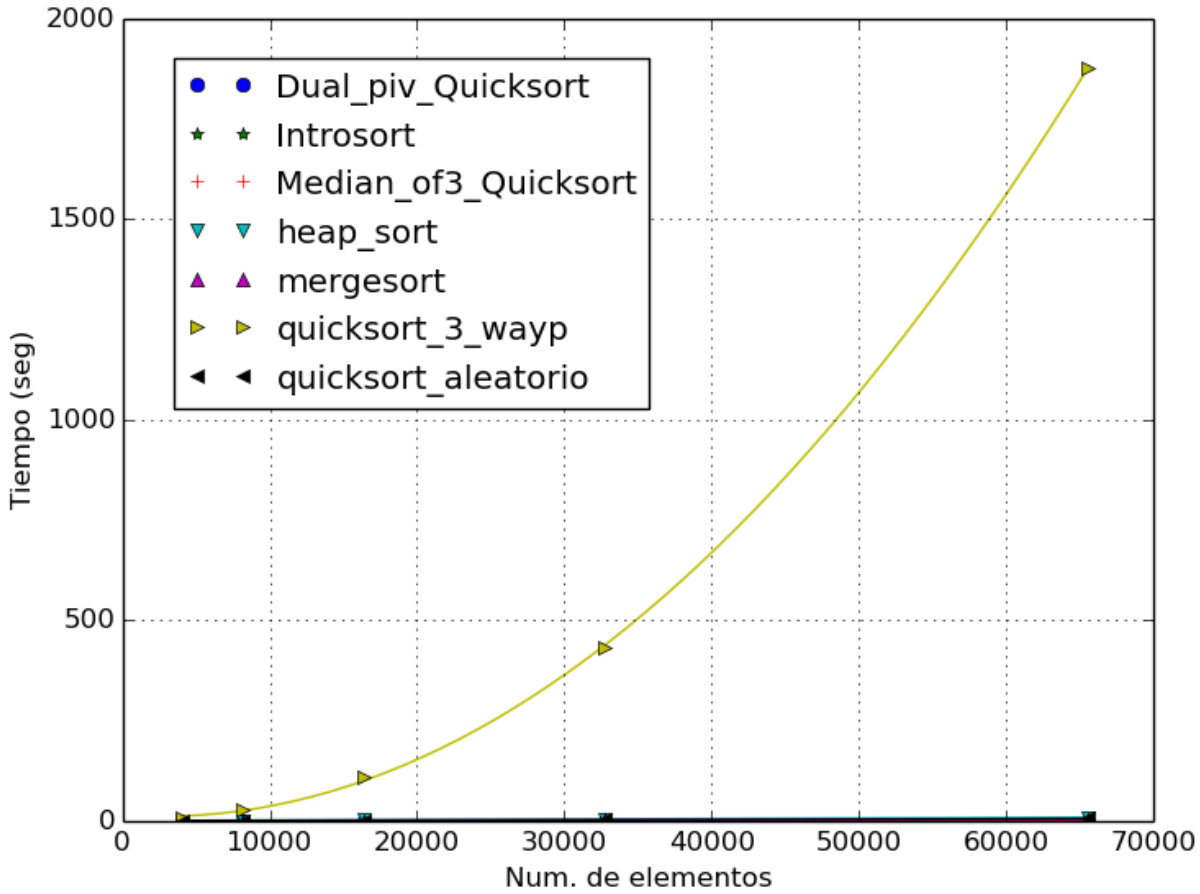
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.210 | 0.189 | 21.676 | 0.163 | 22.926 | 13.084 | 0.055 |
| **8192** | 0.460 | 0.407 | 88.173 | 0.335 | 91.811 | 50.870 | 0.112 |
| **16384** | 0.990 | 0.876 | 355.193 | 0.731 | 362.323 | 200.205 | 0.225 |
| **32768** | 2.113 | 1.972 | 1409.016 | 1.415 | 1463.105 | 827.174 | 0.449 |
| **65536** | 4.618 | 4.032 | 5714.230 | 3.620 | 5992.067 | 3345.874 | 0.908 |




**Llamada #4**
./cliente_ordenamiento.py -g -p 4 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:** Ordenado

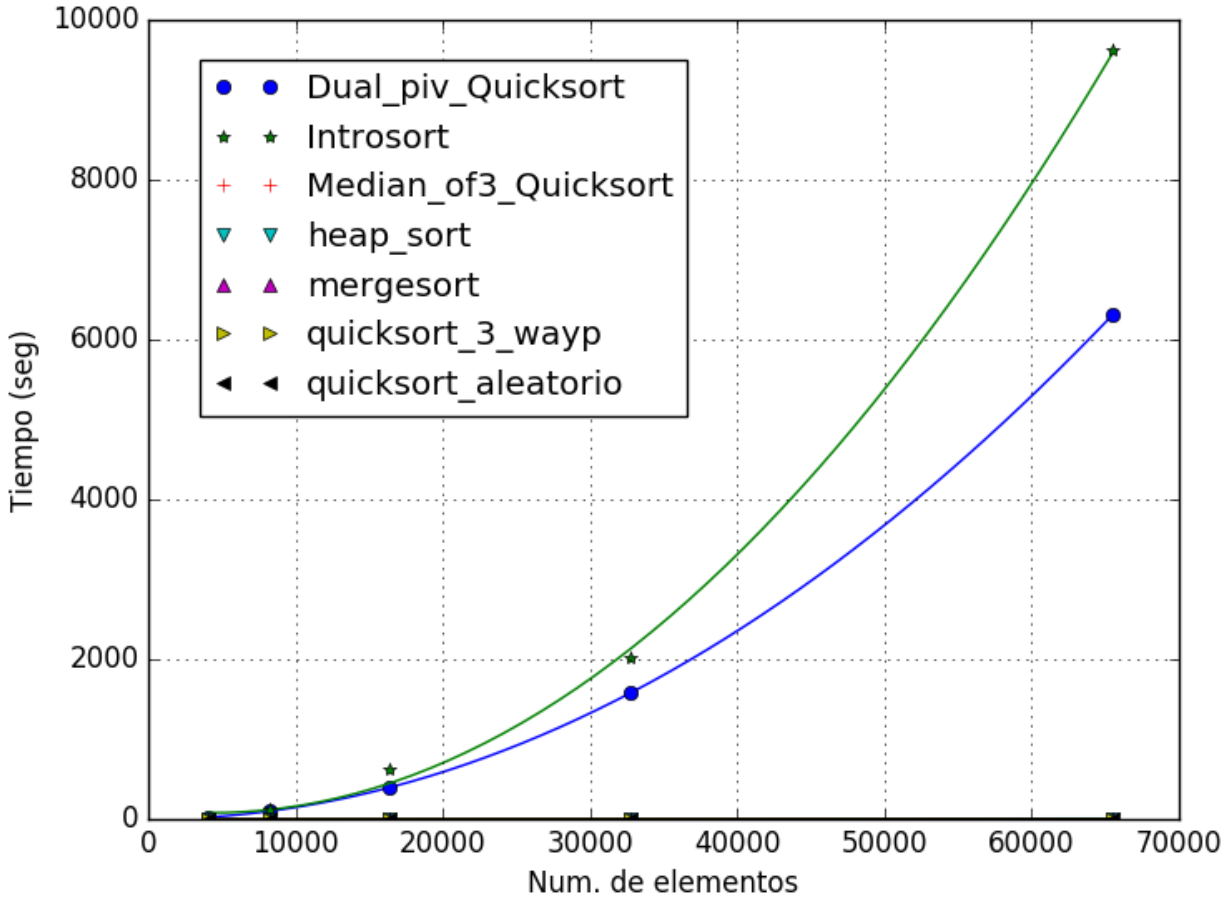
| **N** | **MergeSort** | **HeapSort** | **QS Random** | **Med-of-3 QS** | **Introsort** | **Dual pivot QS** | **QS 3-way** |
|---|---|---|---|---|---|---|---|
| **4096** | 0.183 | 0.396 | 0.233 | 0.047 | 0.047 | 0.014 | 7.338 |
| **8192** | 0.387 | 0.882 | 0.481 | 0.105 | 0.102 | 0.028 | 27.989 |
| **16384** | 0.856 | 1.888 | 1.146 | 0.216 | 0.221 | 0.055 | 112.055 |
| **32768** | 1.804 | 4.079 | 2.191 | 0.454 | 0.457 | 0.111 | 442.214 |
| **65536** | 3.916 | 8.682 | 4.520 | 0.982 | 0.987 | 0.222 | 1780.897 |




**Llamada #5**
./cliente_ordenamiento.py -g -p 5 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:**Reales aleatorios

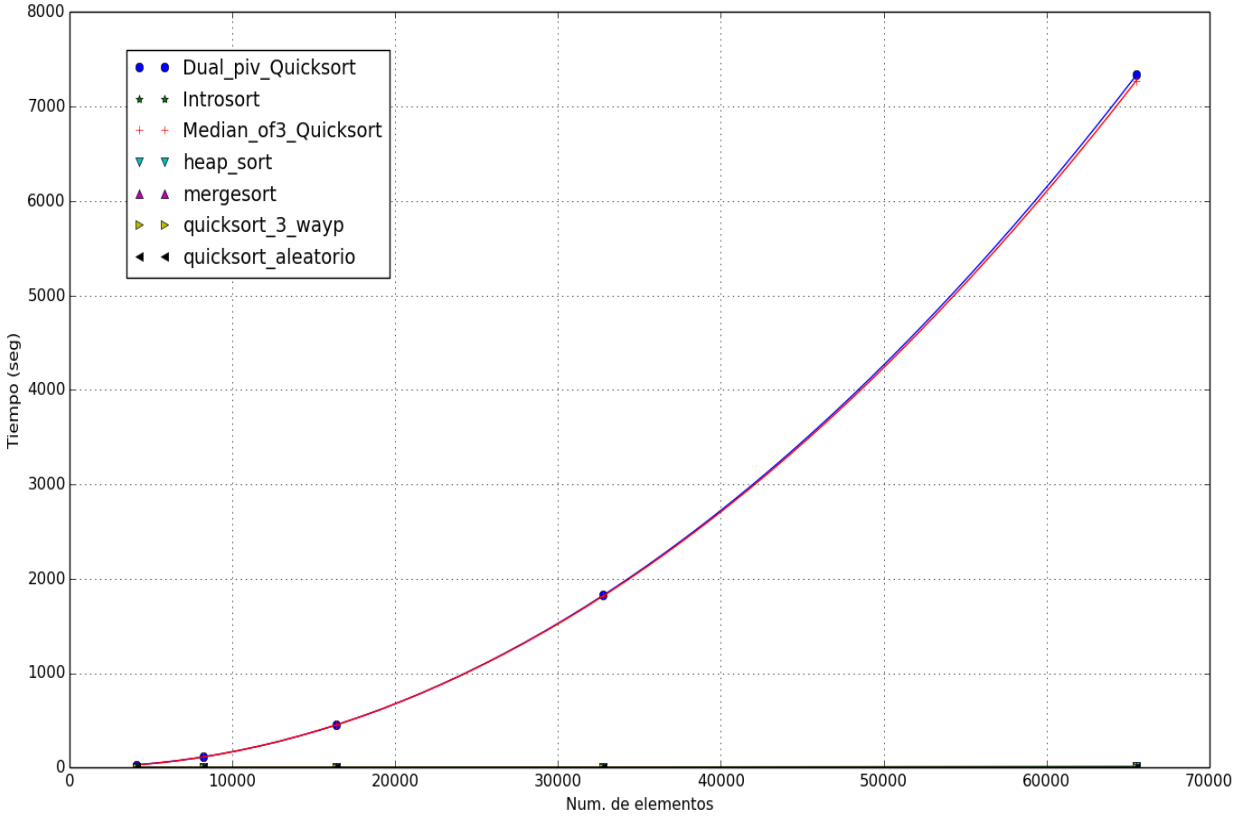
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.232 | 0.381 | 0.215 | 0.184 | 0.183 | 24.953 | 0.210 |
| **8192** | 0.481 | 0.765 | 0.420 | 0.392 | 117,902 | 98.938 | 0.440 |
| **16384** | 1.053 | 1.680 | 0.930 | 0.896 | 617.641 | 390.497 | 0.924 |
| **32768** | 2.243 | 3.575 | 1.960 | 1.800 | 2023.516 | 1583.011 | 1.899 |
| **65536** | 4.823 | 7.701 | 4.382 | 4.018 | 9615.841 | 6316.888 | 3.926 |




**Llamada #6**
./cliente_ordenamiento.py -g -p 6 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:**Mitad

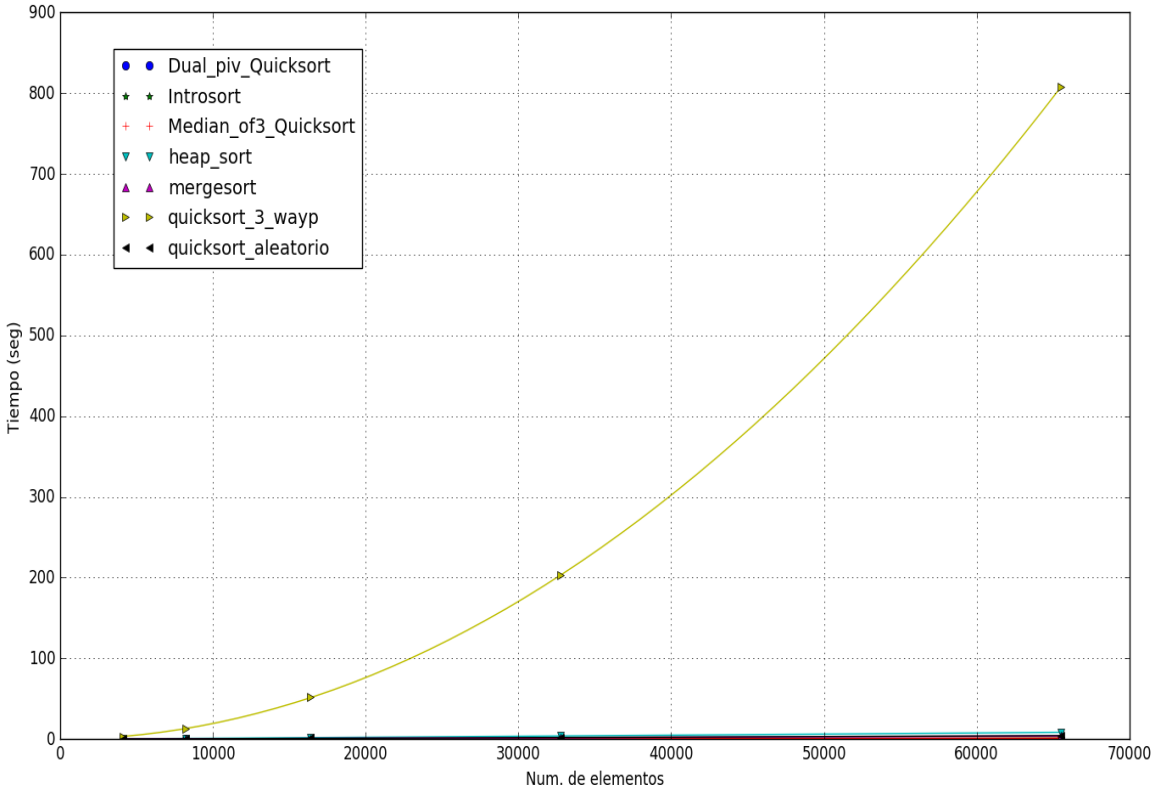
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.209 | 0.400 | 0.223 | 27.444 | 0.561 | 28.523 | 0.259 |
| **8192** | 0.445 | 0.856 | 0.447 | 111.602 | 1.213 | 111.738 | 0.561 |
| **16384** | 0.963 | 1.856 | 1.093 | 450.958 | 2.586 | 453.322 | 1.235 |
| **32768** | 2.090 | 4.030 | 2.069 | 1816.460 | 5.534 | 1822.586 | 2.749 |
| **65536** | 4.404 | 8.595 | 4.960 | 7275.229 | 11.630 | 7333.588 | 5.830 |




**Llamada #7**
./cliente_ordenamiento.py -g -p 7 -t 3 4096 8192 16384 32768 65536

**Conjunto de prueba:** Casi ordenado

| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.182 | 0.391 | 0.204 | 0.066 | 0.065 | 0.047 | 3.385 |
| **8192** | 0.395 | 0.855 | 0.506 | 0.138 | 0.138 | 0.095 | 12.866 |
| **16384** | 0.850 | 1.847 | 0.972 | 0.283 | 0.284 | 0.186 | 51.735 |
| **32768** | 1.838 | 3.997 | 2.047 | 0.594 | 0.593 | 0.378 | 203.085 |
| **65536** | 3.877 | 8.478 | 4.375 | 1.232 | 1.228 | 0.752 | 807.791 |

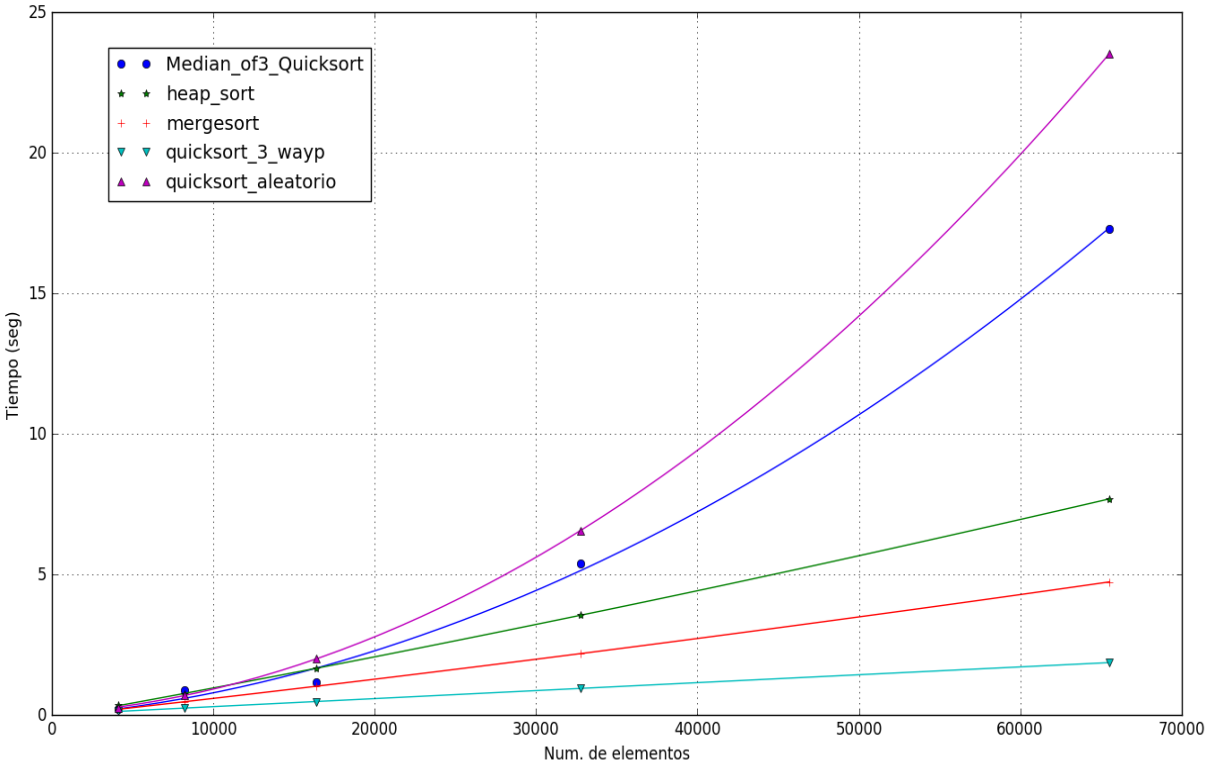



**Segunda corrida de los algoritmos que presentaron tiempo lineal o cuasi-lineal**

#1

**Conjunto de prueba:**Enteros aleatorios sin funciones cuadráticas

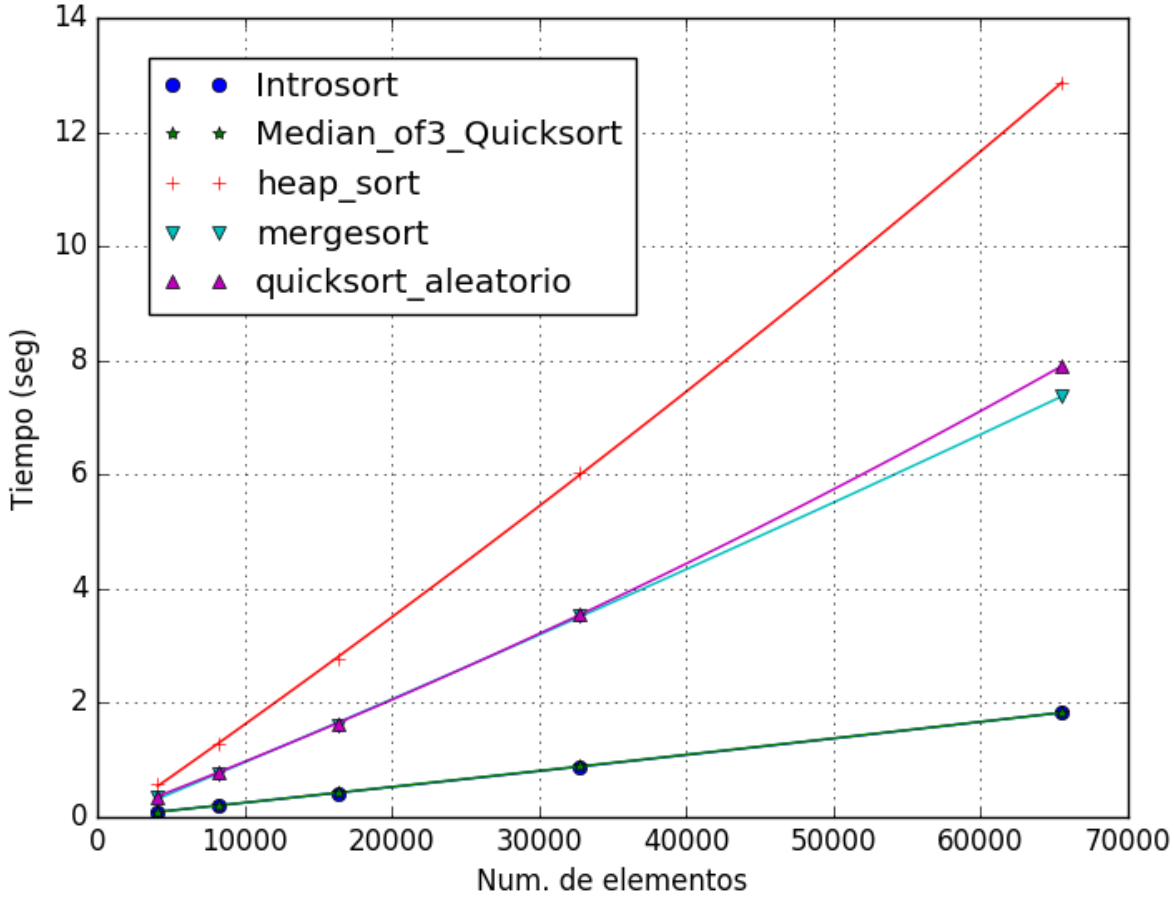
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.218 | 0.348 | 0.257 | 0.258 | x | x | 0.139 |
| **8192** | 0.469 | 0.761 | 0.726 | 0.785 | x | x | 0.245 |
| **16384** | 1.020 | 1.663 | 2.016 | 2.817 | x | x | 0.482 |
| **32768** | 2.199 | 3.562 | 6.559 | 4.276 | x | x | 0.956 |
| **65536** | 4.739 | 7.692 | 23.537 | 48.303 | x | x | 1.869 |




#2

**Conjunto de prueba:**Orden inverso sin funciones cuadráticas

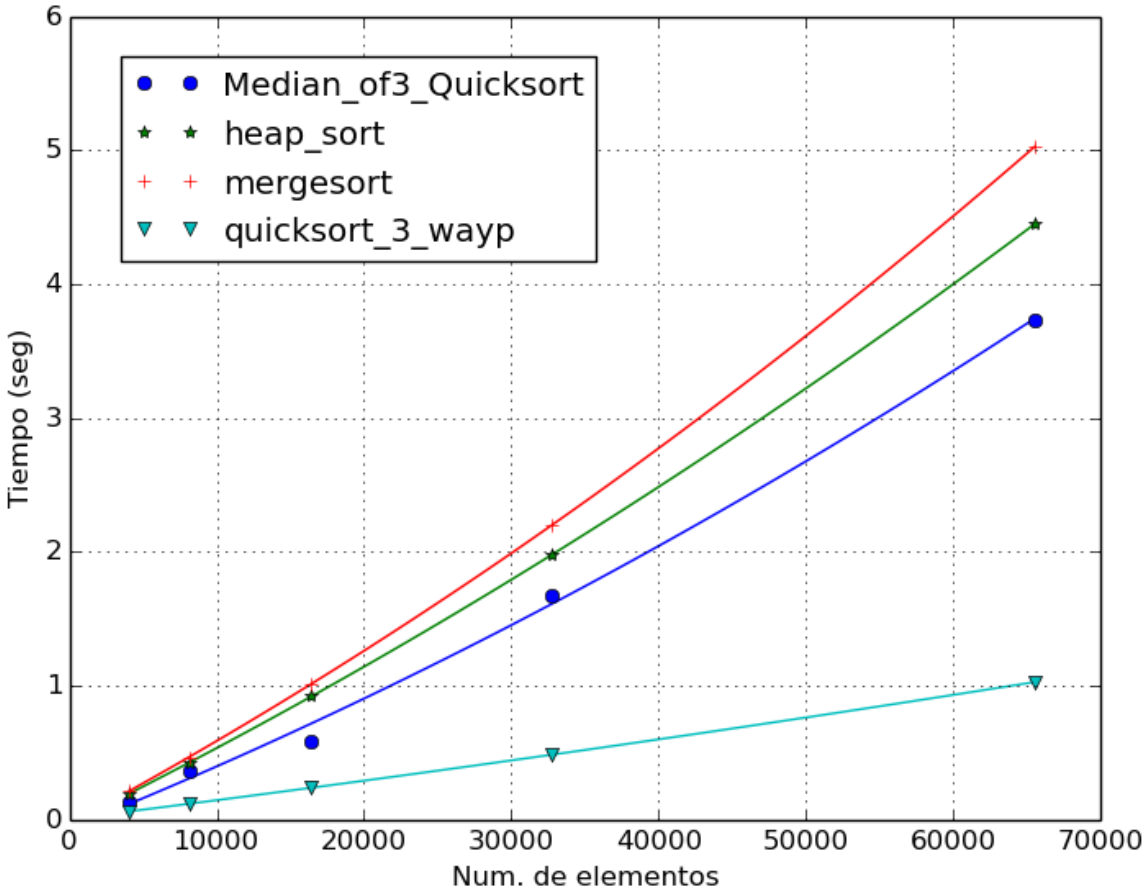
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.344 | 0.574 | 0.361 | 0.096 | 0.097 | | |
| **8192** | 0.755 | 1.269 | 0.789 | 0.198 | 0.199 | | |
| **16384** | 1.615 | 2.776 | 1.646 | 0.423 | 0.422 | | |
| **32768** | 3.539 | 6.024 | 3.547 | 0.894 | 0.888 | | |
| **65536** | 7.369 | 12.868 | 7.905 | 1.830 | 1.832 | | |




#3

**Conjunto de prueba:** Cero-uno

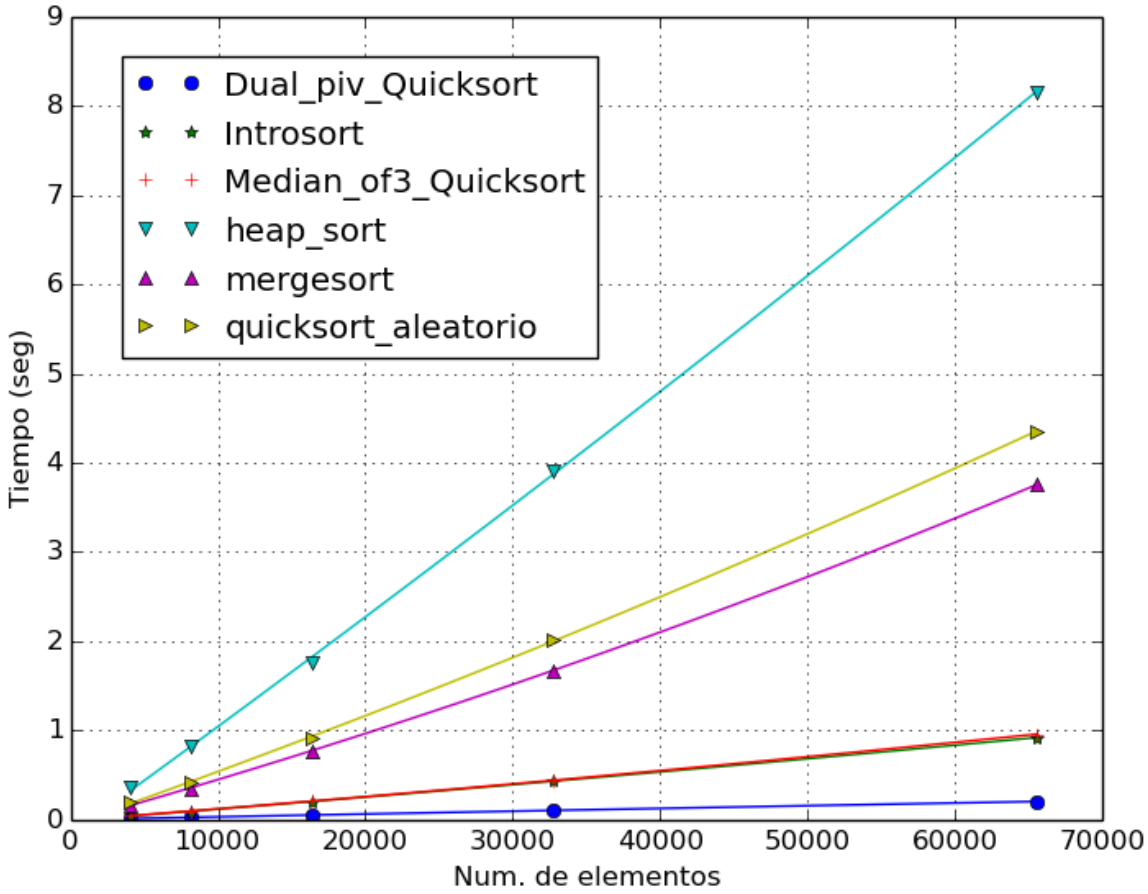
| N | **MergeSort** | **HeapSort** | **Med-of-3 QS** | **QS 3-way** |
|---|---|---|---|---|
| **4096** | 0,219 | 0,198 | 0,130 | 0,059 |
| **8192** | 0,467 | 0,424 | 0,368 | 0,118 |
| **16384** | 1,013 | 0,924 | 0,589 | 0,245 |
| **32768** | 2,202 | 1,978 | 1,680 | 0,481 |
| **65536** | 5,029 | 4,452 | 3,736 | 1,029 |




#4

**Conjunto de prueba:** Ordenado

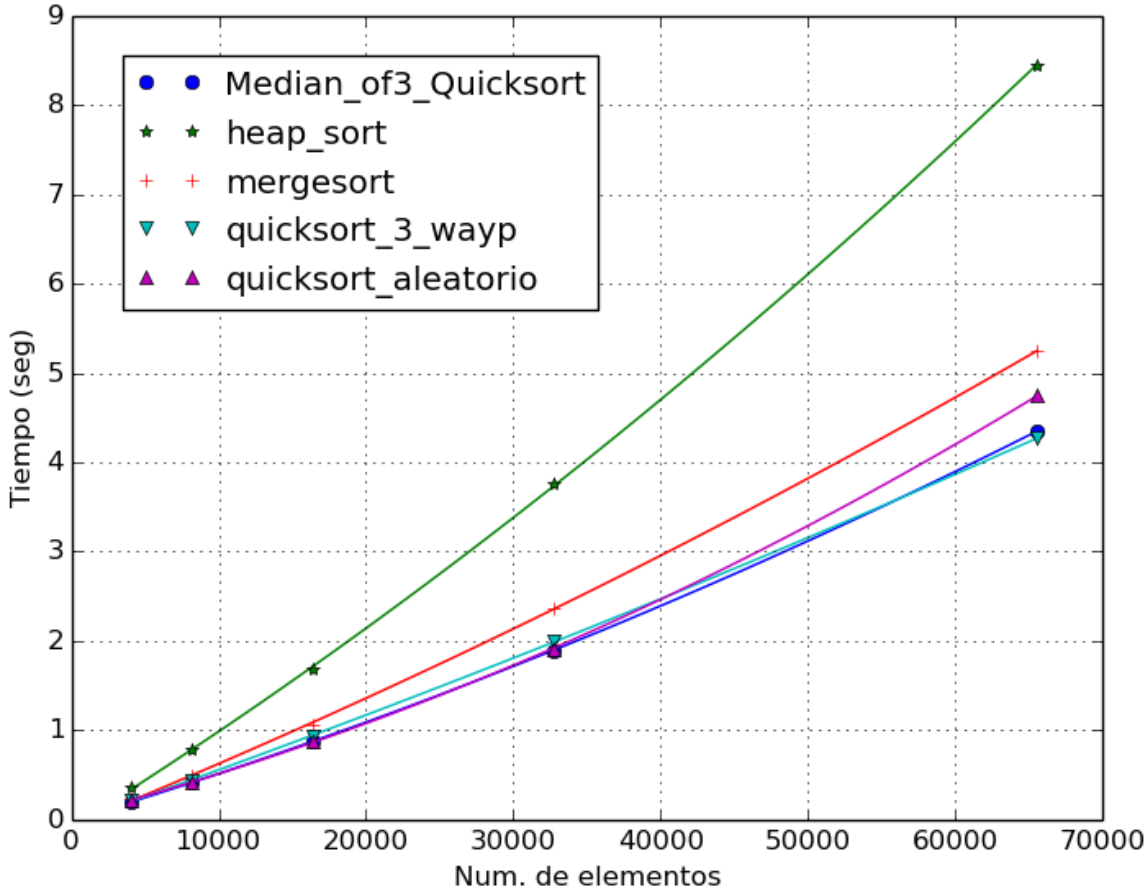
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.182 | 0.400 | 0.209 | 0.049 | 0.049 | 0.014 | x |
| **8192** | 0.396 | 0.871 | 0.475 | 0.103 | 0.103 | 0.028 | x |
| **16384** | 0.882 | 1.910 | 1.005 | 0.231 | 0.223 | 0.057 | x |
| **32768** | 1.888 | 4.069 | 2.175 | 0.469 | 0.469 | 0.114 | x |
| **65536** | 4.046 | 8.804 | 4.513 | 1.017 | 1.018 | 0.227 | x |




#5

**Conjunto de prueba:**Reales aleatorios

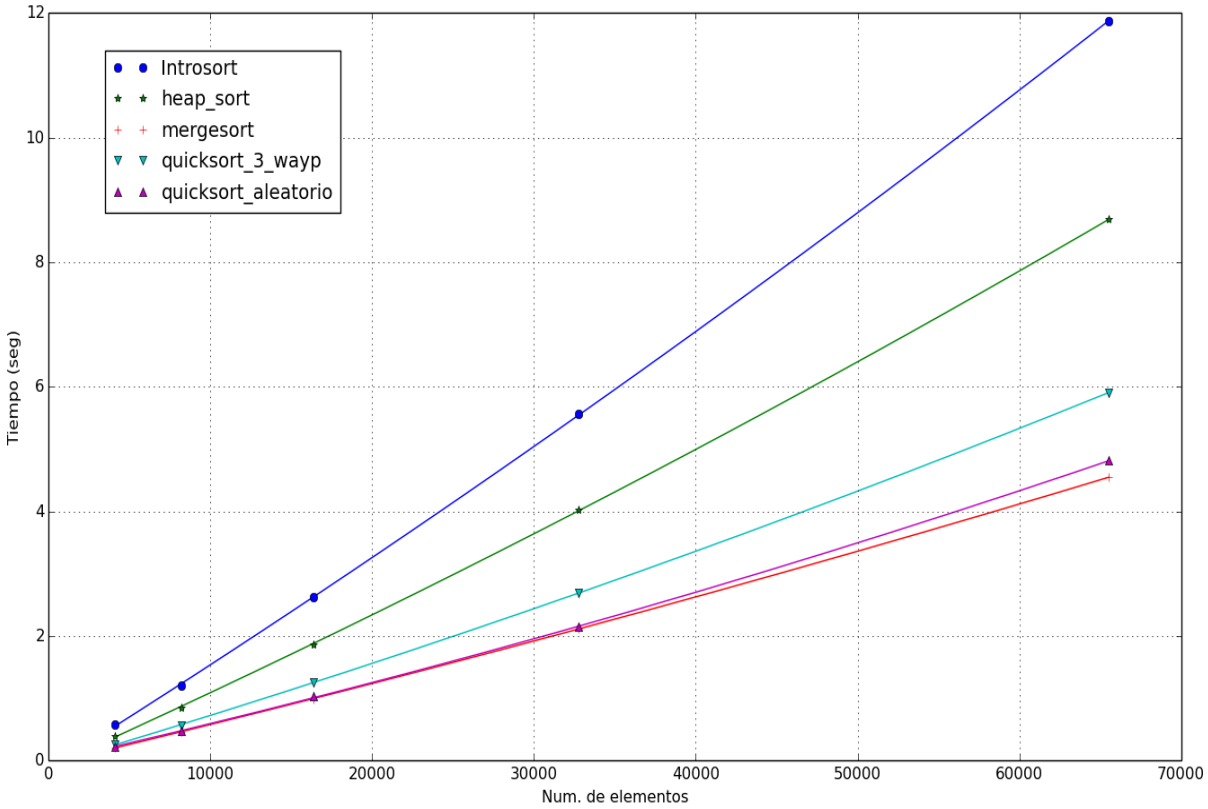
| N | **MergeSort** | **HeapSort** | **Med-of-3 QS** | **QS 3-way** | **Random QS** |
|---|---|---|---|---|---|
| **4096** | 0,228 | 0,357 | 0,194 | 0,216 | 0,213 |
| **8192** | 0,490 | 0,779 | 0,413 | 0,442 | 0,420 |
| **16384** | 1,067 | 1,692 | 0,886 | 0,931 | 0,877 |
| **32768** | 2,374 | 3,753 | 1,892 | 2,000 | 1,913 |
| **65536** | 5,245 | 8,451 | 4,349 | 4,268 | 4,744 |




#6

**Conjunto de prueba:**Mitad

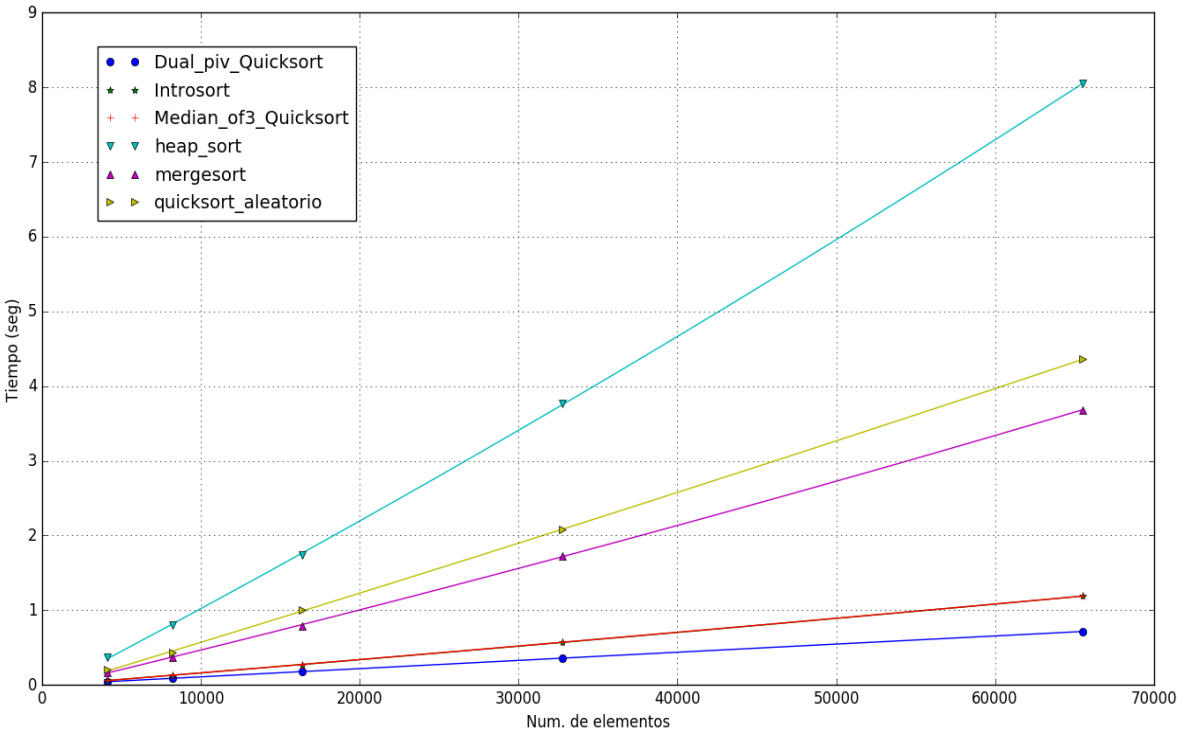
| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.217 | 0.351 | 0.197 | 0.183 | x | x | 0.207 |
| **8192** | 0.473 | 0.768 | 0.417 | 0.405 | x | x | 0.434 |
| **16384** | 1.024 | 1.673 | 0.931 | 0.854 | x | x | 0.896 |
| **32768** | 2.235 | 3.636 | 2.002 | 1.863 | x | x | 1.899 |
| **65536** | 4.718 | 7.798 | 4.193 | 3.858 | x | x | 3.952 |




#7

#**Conjunto de prueba:** Casi ordenado sin funciones cuadráticas

| N | MergeSort | HeapSort | QS Random | Med-of-3 QS | Introsort | Dual pivot QS | QS 3-way |
|---|---|---|---|---|---|---|---|
| **4096** | 0.171 | 0.369 | 0.200 | 0.062 | 0.062 | 0.045 | x |
| **8192** | 0.368 | 0.805 | 0.434 | 0.128 | 0.128 | 0.088 | x |
| **16384** | 0.795 | 1.744 | 0.999 | 0.271 | 0.271 | 0.178 | x |
| **32768** | 1.728 | 3.771 | 2.085 | 0.575 | 0.572 | 0.360 | x |
| **65536** | 3.683 | 8.052 | 4.359 | 1.189 | 1.189 | 0.716 | x |


Dual_piv_Quicksort
Introsort
Median_of3_Quicksort
heap_sort
mergesort
quicksort_aleatorio
Tiempo (seg)
Num. de elementos
0
1
2
3
4
5
6
7
8
9
0
10000
20000
30000
40000
50000
60000
70000

## Conclusiones

Se analizaron siete algoritmos de ordenamiento con siete tipo de pruebas distinto, nuestro objetivo es analizar la eficiencia de cada algoritmo tanto por el tamaño de la entrada como por el tipo de entrada y comparar eso con la teoría respecto al algoritmo

| **Algoritmo** | **Mejores casos** | **Peores casos** |
|---|---|---|
| **Merge Sort**<br>**O(n log n)**<br><br>**En general su comportamiento es casi lineal en todas las pruebas como es de esperarse según la teoría** | A pesar de que teóricamente su tiempo de ordenamiento depende directamente del tamaño del arreglo y no se el tipo de arreglo el MergeSort presento los mejores tiempos en el caso de arreglos ordenados o casi ordenados | Los peores casos para Mergesort están representados por las entradas de arreglos aleatorios tanto de números enteros como de números reales |
| **Heapsort**<br><br>**O(n log n)**<br><br>**En general el heapsort fue un algoritmo eficiente en todas las pruebas y sus tiempos fueron mas o menos acordes** | El heapsort tuvo un comportamiento mas eficiente en el arreglo de ceros y unos aleatorios con respecto a sus resultados con el resto de las pruebas. | El peor caso es el esperado , cuando el arreglo esta totalmente ordenado seguido del caso en el que esta casi ordenado |
| **Quicksort randomizado** | Su mejor caso fue la prueba con numeros reales aleatorios sin embargo en general sus tiempos de corridas son buenos en casi todos los casos probados | Su peor caso fue la prueba con el arreglo de ceros y unos |
| **Med of 3 QS**<br><br>**los tiempos de corrida de este algoritmo fueron mas variados que los algoritmos anteriores respecto a las entradas recibidas** | Su mejor caso en las pruebas realizadas fue con el arreglo ordenado | Su peor caso en las pruebas realizadas fue con el arreglo a mitad ,correspondiente a la prueba 6 |
| **introsort** | Su mejor caso en las pruebas realizadas fue con el arreglo ordenado | El algoritmo no fue eficiente con los arreglos aleatorios y el arreglo de ceros y unos |
| **Dual pivot QS** | Su mejor caso en las pruebas realizadas fue con el arreglo ordenado | Este algoritmo tuvo un comportamiento parecido al Introsort presentando ineficiencia en los mismos tipos de entradas |

| | | |
|---|---|---|
| **QS 3 way** | Su mejor tiempo de ejecución fue con la entrada de ceros y unos | Presento el peor caso para arreglos ordenados |

En general los algoritmos que fueron mas fieles a la teoría fueron el mergesort, el heapsort , el quicksort randomizado y el QS 3 way, aunque se supone que todos los tipos de Quicksort son O(n log n) en el peor caso, corrían muy a menudo en tiempo cuadrático. Según las pruebas realizadas (sin los algoritmos cuadráticos) el algoritmo mas efectivo en la mayoría de los casos fue **Quicksort with 3-way partitioning** que seria la mejor opción para ordenar arreglos grandes siempre y cuando los mismos no estén ordenados de forma ascendiente. En general, fuera de los casos cuadráticos los resultados obtenidos corresponden a la teoría estudiada respecto a los algoritmos, como era de esperar eso varia de acuerdo a las entradas recibidas y al tamaño de las entradas.