Análisis de algoritmos de ordenación

Estructura de Datos

José Miguel Ramírez Sanz – José Luis Garrido Labrador

Índice

Índice de ilustraciones

Índice de tablas

Análisis de tiempo

Método interno

Números aleatorios

Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)
10000	5		135000	2		260000	4		385000	6
15000	3		140000	2		265000	4		390000	6
20000	4		145000	4		270000	15		395000	25
25000	4		150000	3		275000	5		400000	44
30000	2		155000	2		280000	5		405000	6
35000	2		160000	2		285000	5		410000	6
40000	4		165000	3		290000	5		415000	6
45000	1		170000	2		295000	4		420000	6
50000	1		175000	3		300000	5		425000	7
55000	0		180000	3		305000	5		430000	7
60000	1		185000	3		310000	5		435000	7
65000	0		190000	5		315000	5		440000	8
70000	1		195000	3		320000	5		445000	8
75000	1		200000	3		325000	6		450000	10
80000	1		205000	3		330000	6		455000	12
85000	1		210000	4		335000	24		460000	8
90000	1		215000	3		340000	5		465000	8
95000	1		220000	4		345000	5		470000	10
100000	9		225000	21		350000	7		475000	8
105000	2		230000	5		355000	7		480000	17

Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)
110000	2		235000	3		360000	5		485000	12
115000	11		240000	4		365000	5		490000	9
120000	2		245000	4		370000	5		495000	9
125000	2		250000	23		375000	30		500000	11
130000	1		255000	4		380000	8		505000	28

Utilizando la herramienta *statgraphics* hemos hecho una regresión simple utilizando en el eje X los elementos y en el eje Y el tiempo invertido. El rango ha sido entre 10000 a 505000 elementos con un total de 100 casos.

El gráfico de dispersión obtenido es el siguiente:

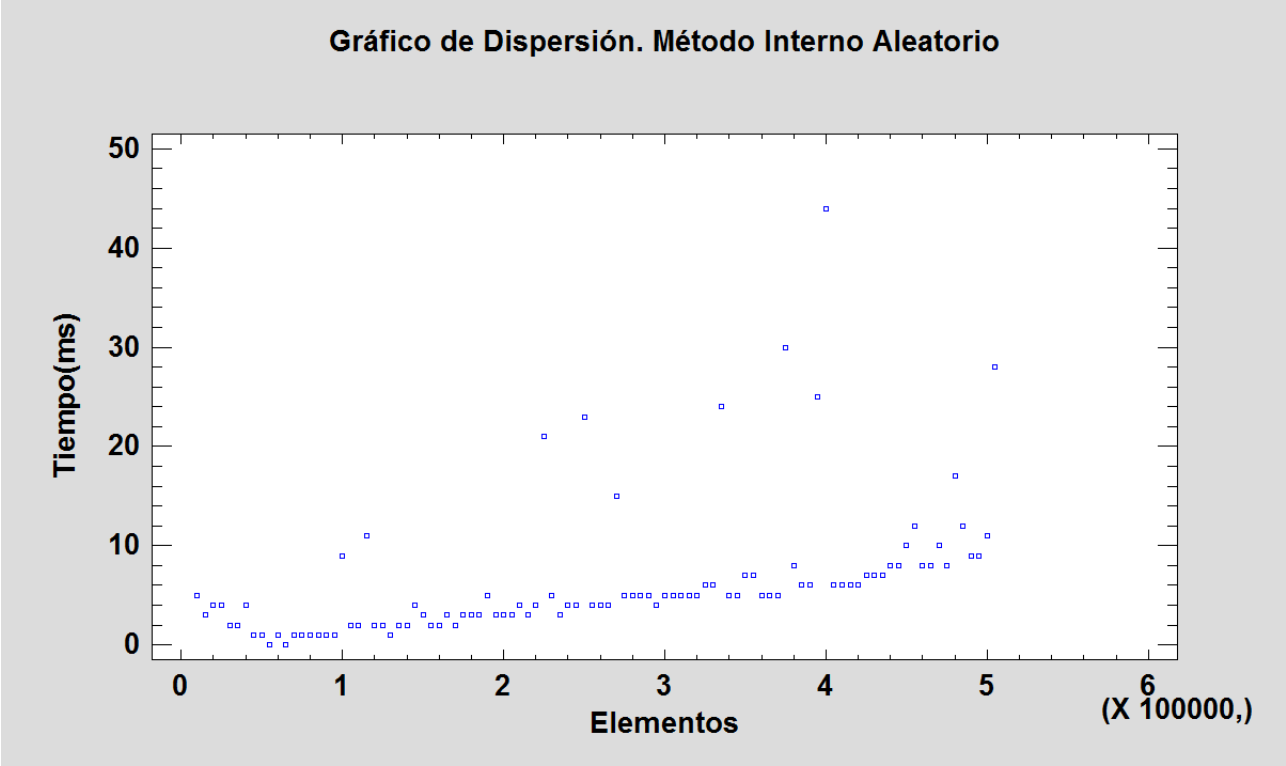

Ilustración 1: Gráfico de dispersión del método interno con datos aleatorios

Conociendo el funcionamiento de las tablas ANOVA hemos deducido que la función que más se aproxima es: $O(\sqrt{n})$ siendo el gráfico de residuos el siguiente. No hemos podido comentar la complejidad de este caso con el código ya que no disponemos de él.

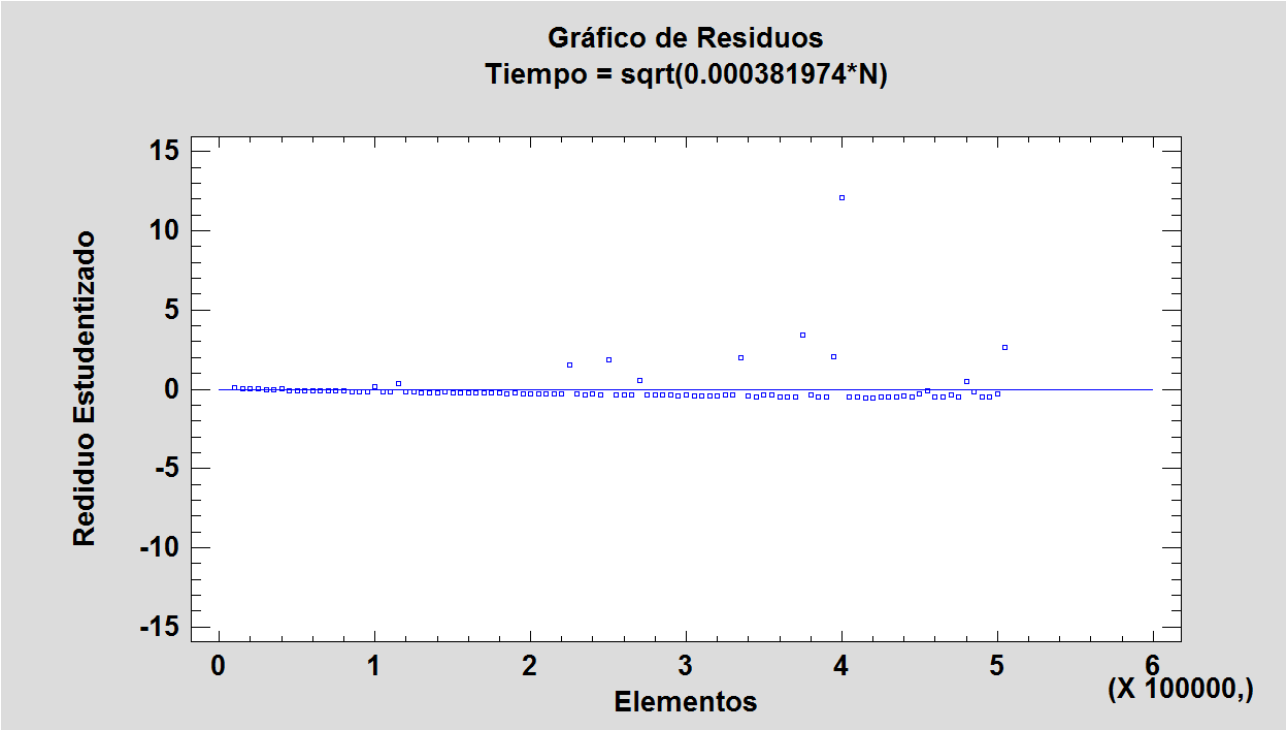

Ilustración 2: Gráfico de residuos del método interno

Números ordenados

En el estudio de esta situación no hubo ninguna mejora en la ejecución del algoritmo, los ordena igualmente, invirtiendo poco tiempo pero el mismo aproximadamente que con números aleatorios. Mostramos algunos datos y su gráfico de dispersión:

460000	38
465000	19
470000	12
475000	15
480000	29
485000	21
490000	11
495000	18

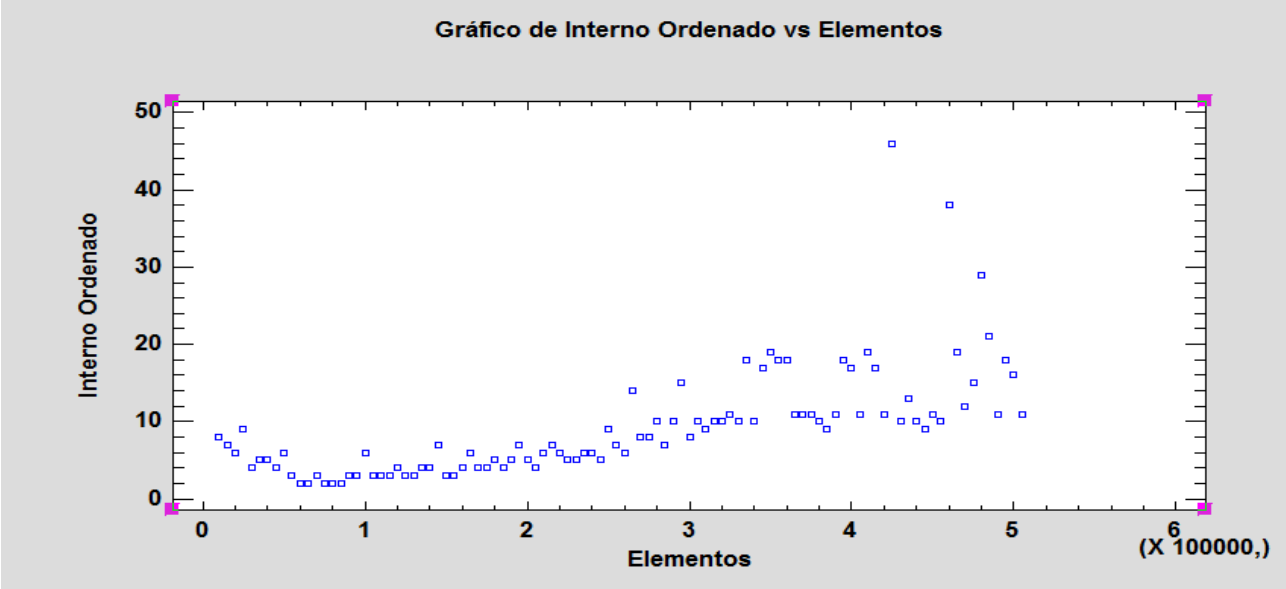

Ilustración 3: Gráfico de dispersión para una lista ordenada

Como se puede observar es bastante similar, ya que casi nunca supera los 10 ms al igual que el caso anterior con una lista totalmente desordenada.

Números inversamente ordenados

En este caso ocurre lo mismo que en los otros dos. Para comprobarlo se puede ver en este gráfico de dispersión:

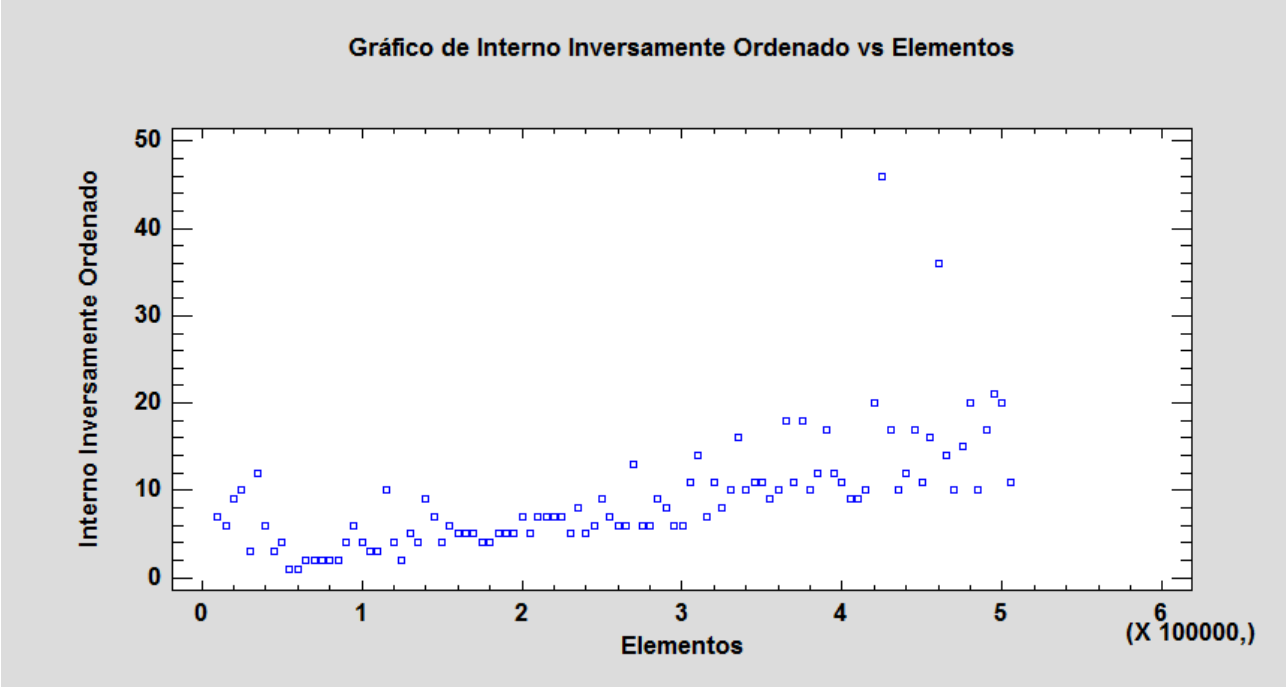

Ilustración 4: Gráfico de dispersión del método interno con lista inversamente ordenado

En esta situación lo único a destacar es que tarda un poco más sin embargo parece algo más bien cosa del azar.

Método merge-sort

Números aleatorios

Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)
10000	3		135000	17		260000	30		385000	38
15000	3		140000	21		265000	30		390000	37
20000	4		145000	19		270000	28		395000	38
25000	5		150000	41		275000	30		400000	55
30000	6		155000	18		280000	32		405000	42
35000	5		160000	12		285000	61		410000	48
40000	4		165000	18		290000	26		415000	45
45000	8		170000	21		295000	22		420000	44
50000	5		175000	28		300000	26		425000	76
55000	10		180000	16		305000	24		430000	47
60000	12		185000	18		310000	24		435000	47
65000	20		190000	23		315000	31		440000	42
70000	8		195000	21		320000	42		445000	76
75000	8		200000	30		325000	35		450000	50
80000	12		205000	32		330000	33		455000	51
85000	8		210000	31		335000	32		460000	51
90000	9		215000	47		340000	33		465000	71
95000	14		220000	19		345000	32		470000	49
100000	31		225000	24		350000	51		475000	53
105000	9		230000	25		355000	34		480000	47

Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)
110000	15		235000	27		360000	35		485000	61
115000	10		240000	27		365000	43		490000	44
120000	22		245000	28		370000	36		495000	41
125000	12		250000	38		375000	55		500000	48
130000	10		255000	26		380000	37		505000	68

Volviendo a utilizar la misma herramienta conseguimos el siguiente gráfico de dispersión:

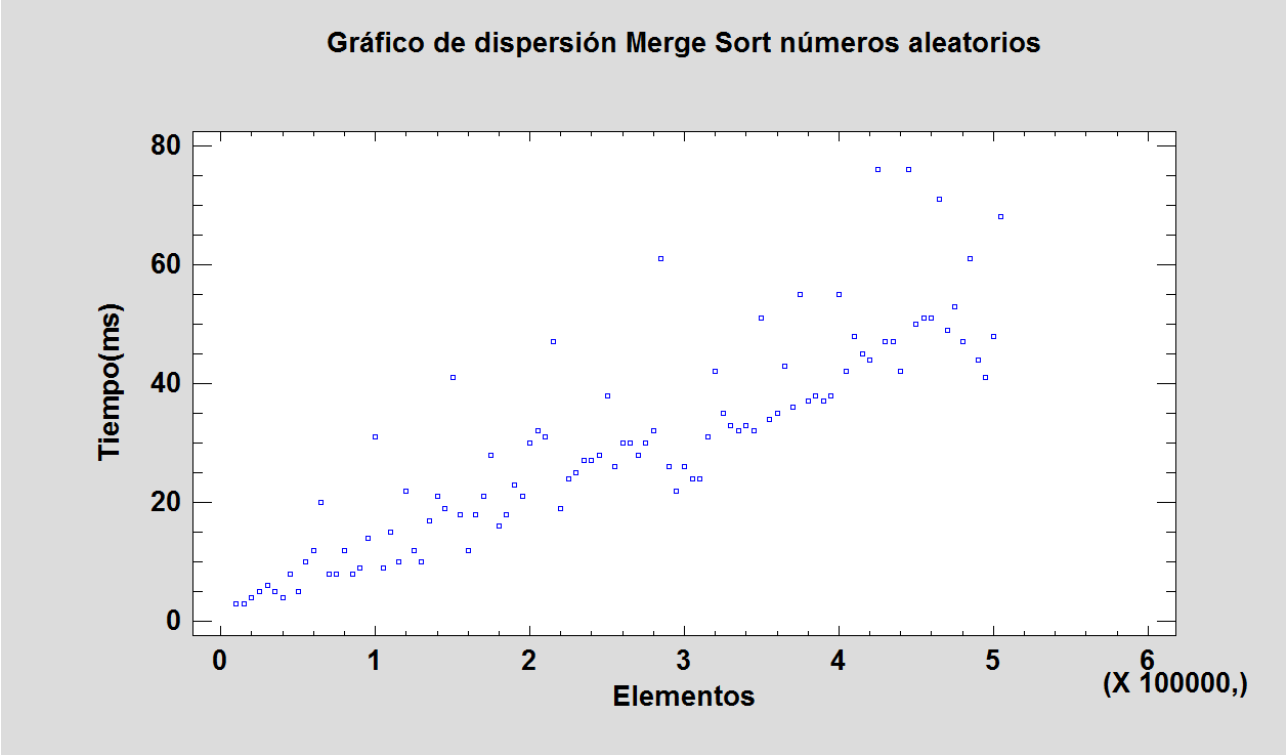

Ilustración 5: Gráfico de dispersión para números aleatorios con merge-sort

Sin embargo no existe en Statgraphics una función predefinida que defina con exactitud este modelo por lo que basándonos en el siguiente gráfico:

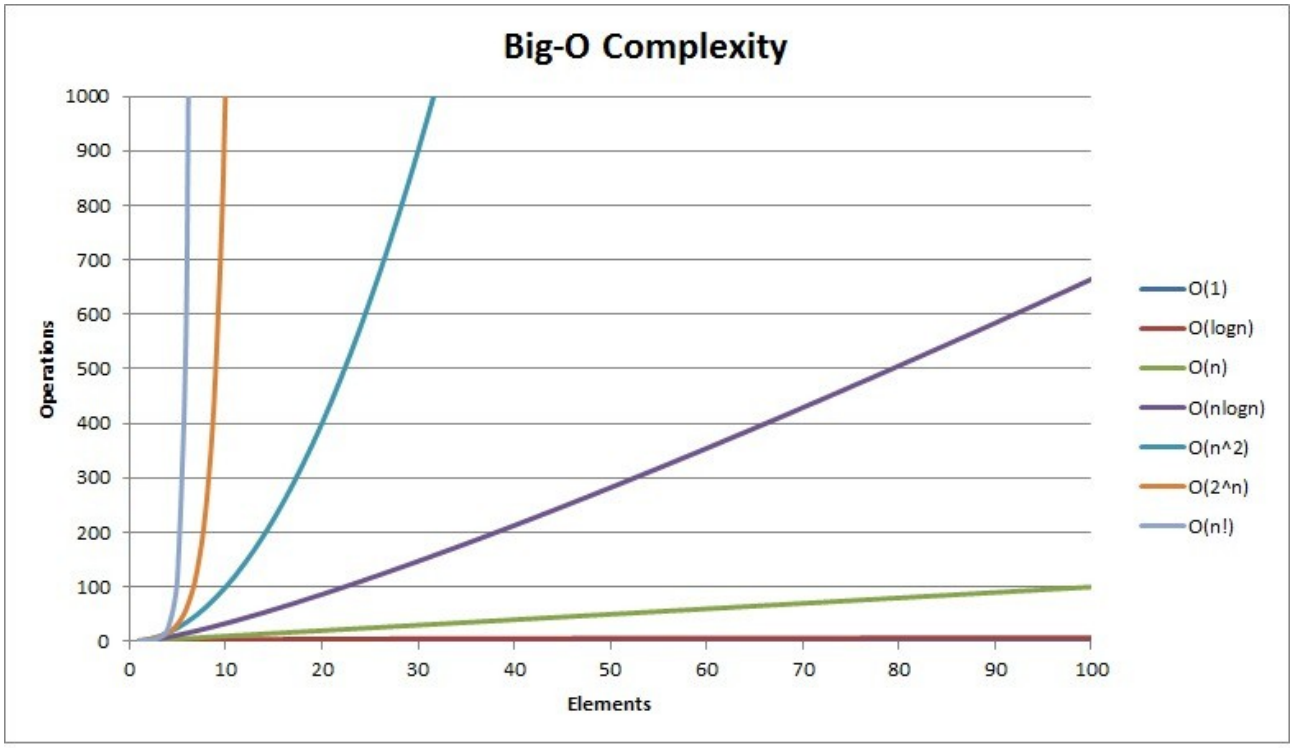

Ilustración 6: Gráficas de O grandes

Y analizando el código deducimos que se trata de la complejidad $O(n\log(n))$.

Tenemos el factor n por el uso de un while que recorre todos los elementos. Luego se ejecutan bucles for según cual sea mayor y lo hace hasta que encuentra un hueco en el array nuevo donde se mezclan los dos arrays ordenados. Este bucle es cada vez más grande pero nunca recorre todo el array si no que recorre como máximo la mitad del total (La primera división) por tanto junto con los datos obtenidos deducimos que el segundo factor es log(n) porque los bucles se recorren como máximo la mitad de los arrays divididos. La mayoría del código extra tiene una complejidad de $O(1)$ ya que son ordenes if, mover datos o sumas. Usamos la gráfica mostrada para justificar nuestra interpretación.

Números Ordenados e inversamente ordenados

Debido que la implementación no comprueba nunca el estado del *array* para saber si está ordenado o inversamente ordenado no hay ninguna diferencia con los datos anteriores. Aquí dejamos los gráficos de dispersión de ambos casos:

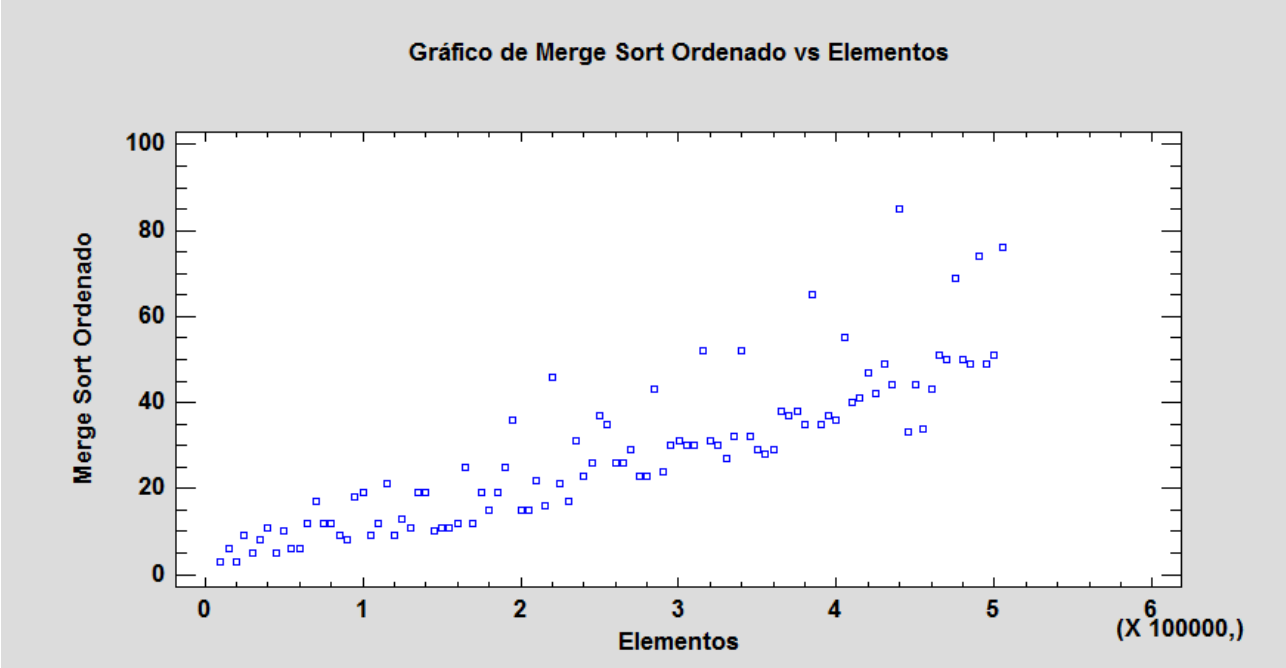

Ilustración 7: Gráfica elementos/tiempo para una lista ordenada sobre el algoritmo MergeSort

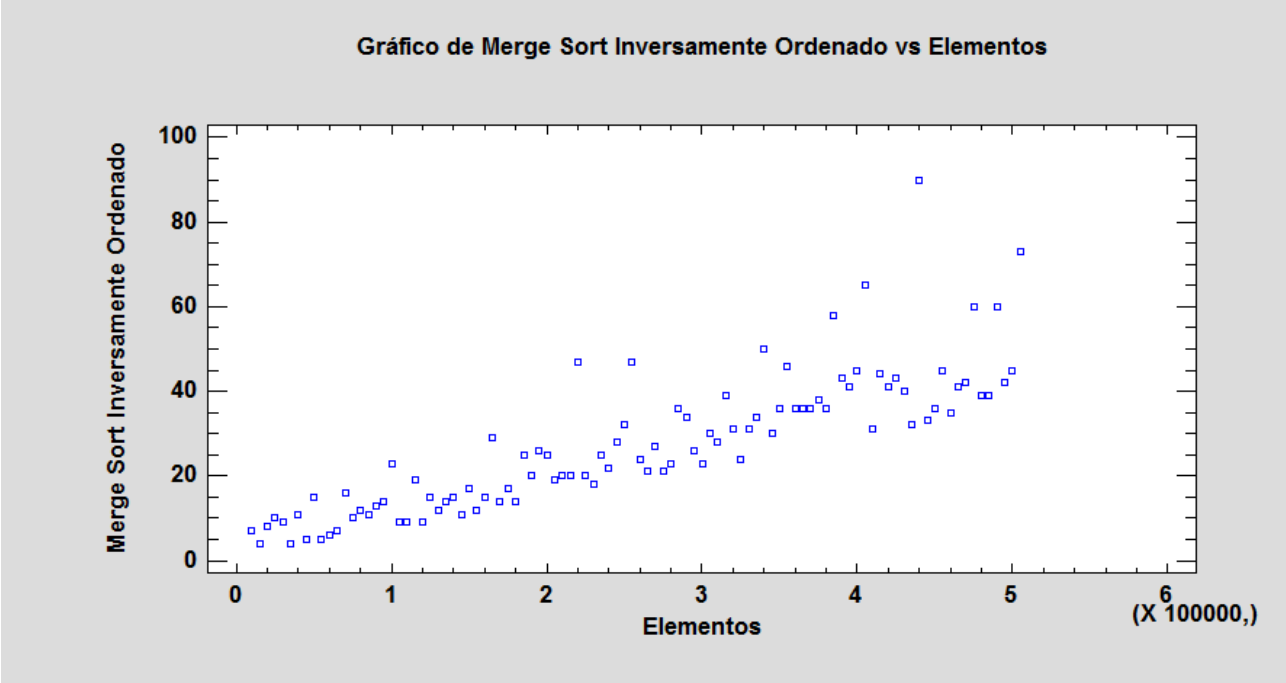

Ilustración 8: Gráfico para la relación elementos/tiempo con una array inversamente ordenado sobre MergeSort

Método rangos

El método rangos lo conocimos en la asignatura de Arquitectura de computadores, consiste en que cada elemento se compare con todos los demás y por cada número menor o igual pero en un índice superior se añade un valor a un entero y luego este entero será el índice en el array ya ordenado. Este sistema es poco eficaz sobre arquitecturas secuenciales por lo que como veremos su complejidad será de $O(n^2)$ pero es muy eficaz en sistemas paralelos cuya complejidad será de O(n) ya que cada número funcionaría en un hilo distinto. Ya sin más dilación vamos a comentar los resultados y como llegamos a las conclusiones aquí expuestas.

Números aleatorios

Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)
10000	130		135000	19764		260000	126686		385000	229227
15000	255		140000	21079		265000	81438		390000	199641
20000	442		145000	22811		270000	73842		395000	207914
25000	707		150000	24824		275000	76625		400000	228088
30000	995		155000	25910		280000	127547		405000	313592
35000	1341		160000	28118		285000	92310		410000	269648
40000	1729		165000	30754		290000	94859		415000	216607
45000	2190		170000	31150		295000	100043		420000	213697
50000	2701		175000	33025		300000	105817		425000	224386
55000	3243		180000	35206		305000	108237		430000	229436
60000	3844		185000	37710		310000	119285		435000	233704
65000	4565		190000	41159		315000	122311		440000	246810
70000	5470		195000	41319		320000	122717		445000	249647
75000	6063		200000	44253		325000	124950		450000	257060
80000	6867		205000	46838		330000	126912		455000	295881
85000	7774		210000	48378		335000	128092		460000	379010
90000	8967		215000	55125		340000	137903		465000	386409

Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)		Elementos	Tiempo(ms)
95000	9646		220000	57112		345000	191059		470000	349442
100000	10995		225000	98429		350000	158612		475000	436551
105000	12009		230000	62450		355000	167041		480000	351428
110000	13341		235000	57790		360000	174984		485000	390691
115000	15301		240000	63952		365000	183267		490000	386607
120000	15610		245000	71141		370000	166569		495000	381135
125000	16819		250000	19764		375000	175717		500000	375136
130000	18065		255000	21079		380000	180906		505000	365734

Utilizando el statgraphics conseguimos el siguiente gráfico de dispersión:

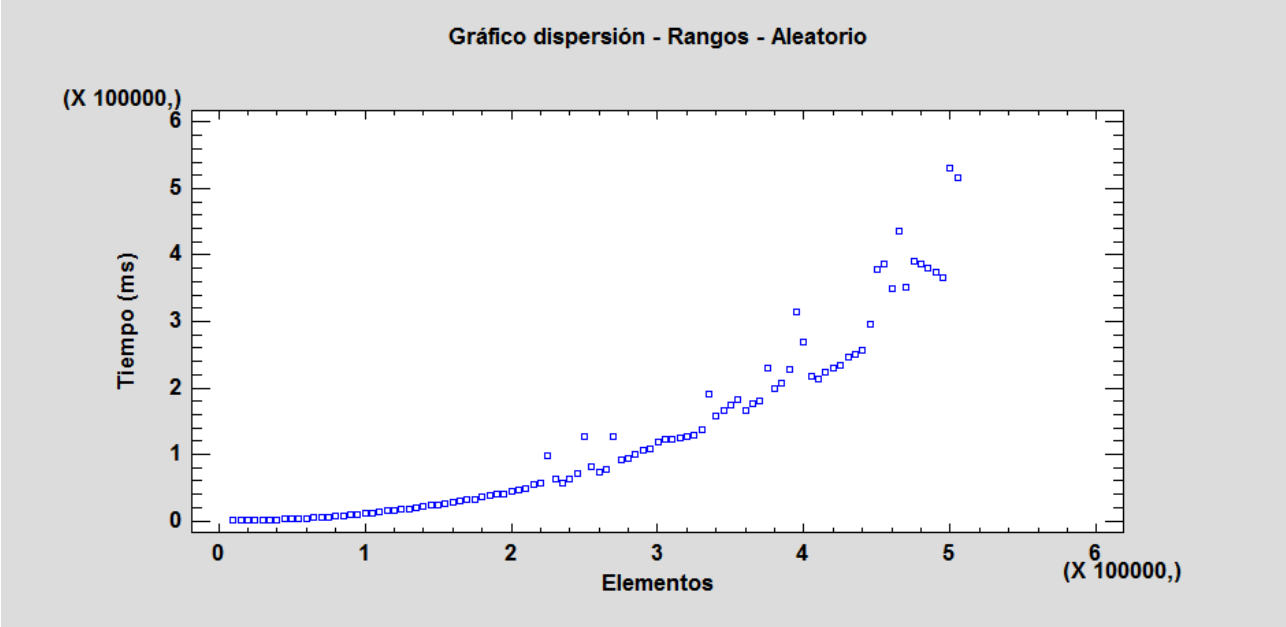

Ilustración 9: Gráfico dispersión array aleatorio método rangos

Ya a simple vista podemos ver que es exponencial pero vamos a continuar con el análisis de statgraphics con el gráfico de residuos:

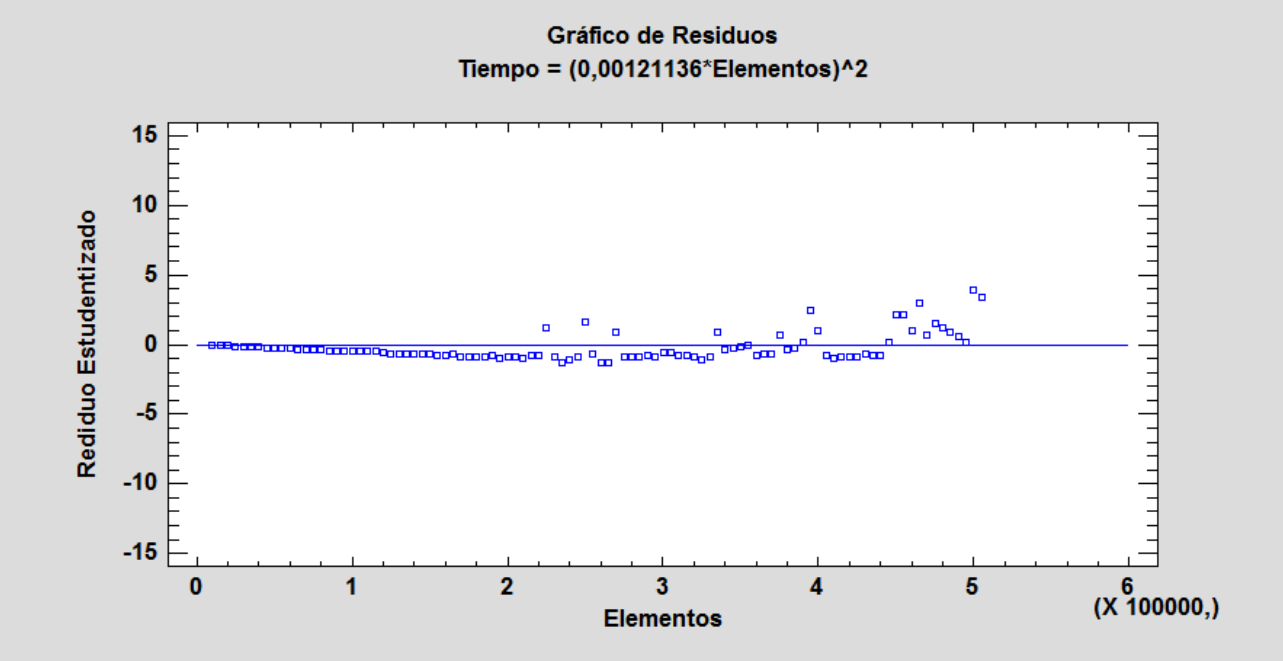

Ilustración 10: Gráfico de residuos para el rangos con array aleatorio

A partir de los gráficos de Statgraphics podemos ver que se trata de una complejidad de $O(n^2)$ ya su gráfico de dispersión es muy parecido a una función n^2 y su gráfico de residuos en comparación con está función demuestra que los valores obtenidos se acercan a la gráfica de n^2.

Además si analizamos el código vemos que hay un bucle for que recorre todo el array (O(n)) y otro bucle for anidado que recorre nuevamente todo el array (O(n)) haciendo así que el método total tenga una complejidad de $O(n^2)$.
El resto de elementos son solo asignaciones y comparaciones por lo que es O(1).

Números Ordenados e inversamente ordenados

El algoritmo no varía nunca entre arrays ya ordenado o desordenado porque el código no tiene nunca eso en cuenta.
Aquí ponemos los gráficos de dispersión que justifican esto:

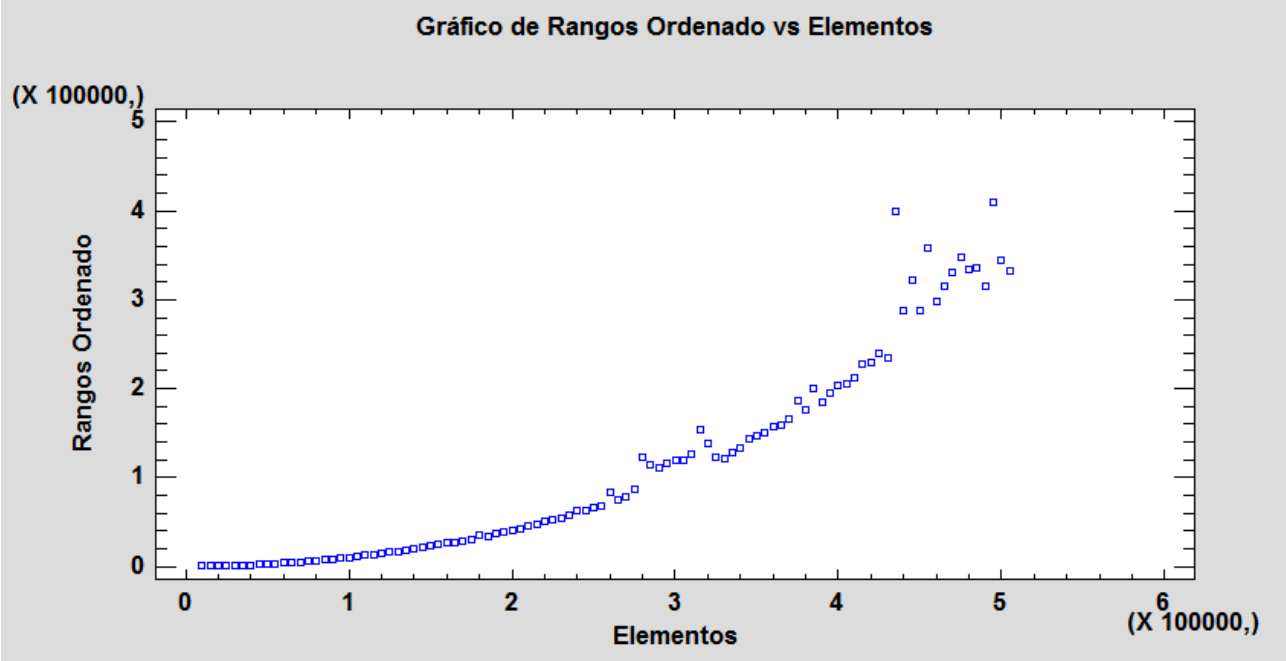

Ilustración 11: Gráfico dispersión algoritmo de rangos con array ordenado

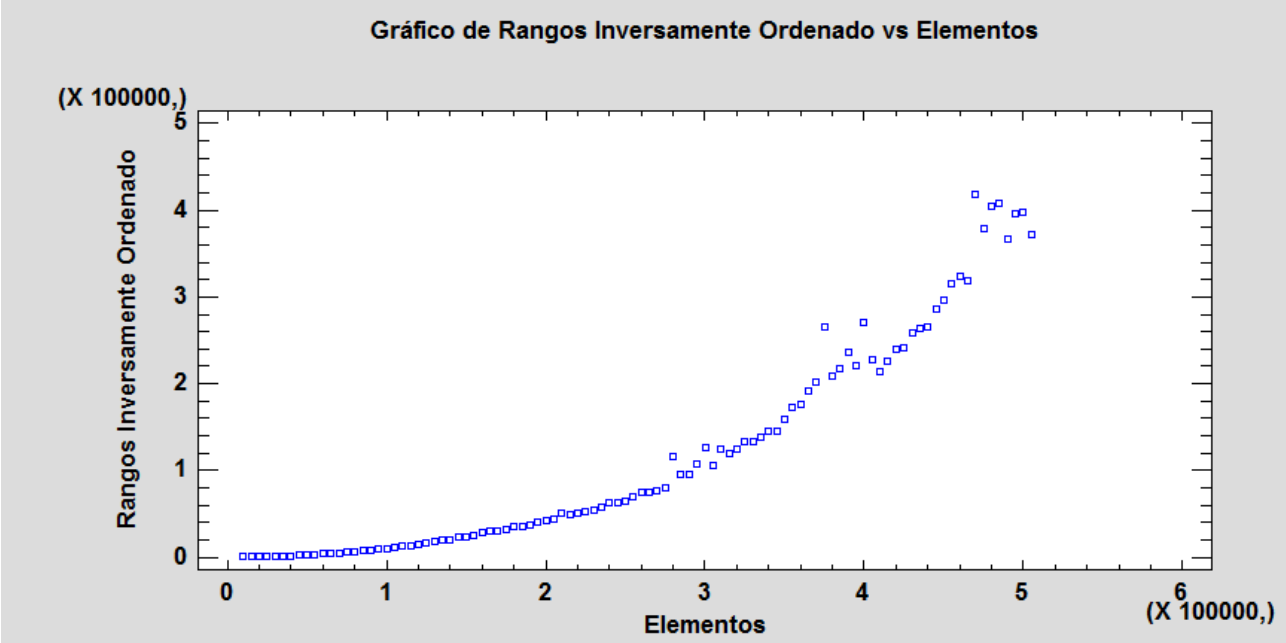

Ilustración 12: Gráfico de dispersión algortimo de rangos array inversamente ordenado

Análisis de memoria

Para el análisis de la memoria hemos usado el programa Java Visual VM. No hemos podido ver el consumo de memoria para cada caso pero si hemos observado la evolución del sistema.

Método Interno

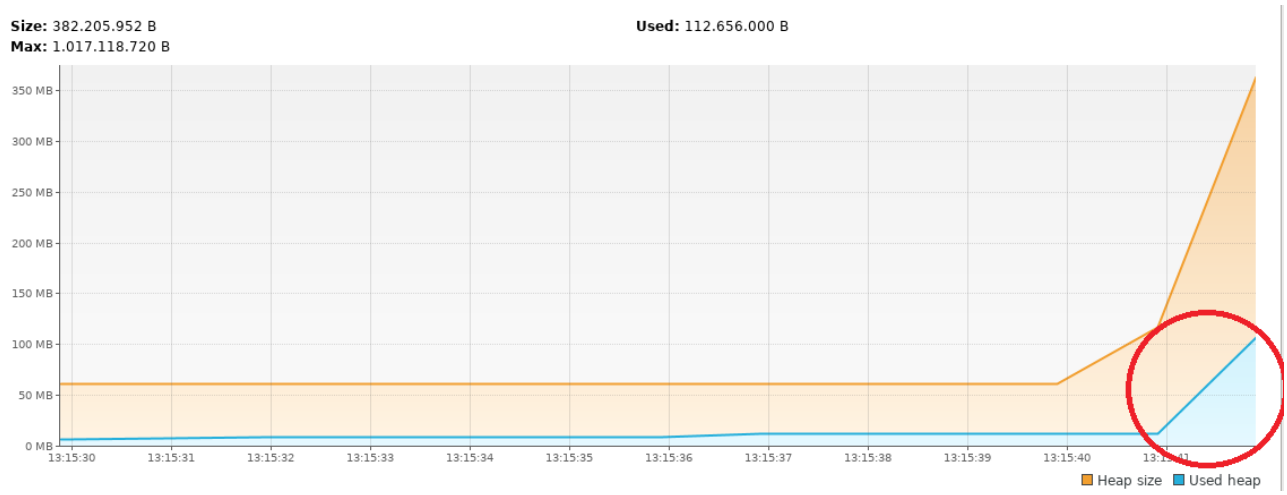

Ilustración 13: Consumo de memoria método interno

La memoria consumida es la gráfica azul y la gráfica naranja es la memoria reservada para el proceso. Comparando con la Ilustración Gráficas de O grandes deducimos que se trata de una complejidad de memoria O(nlog(n)).

Método MergeSort

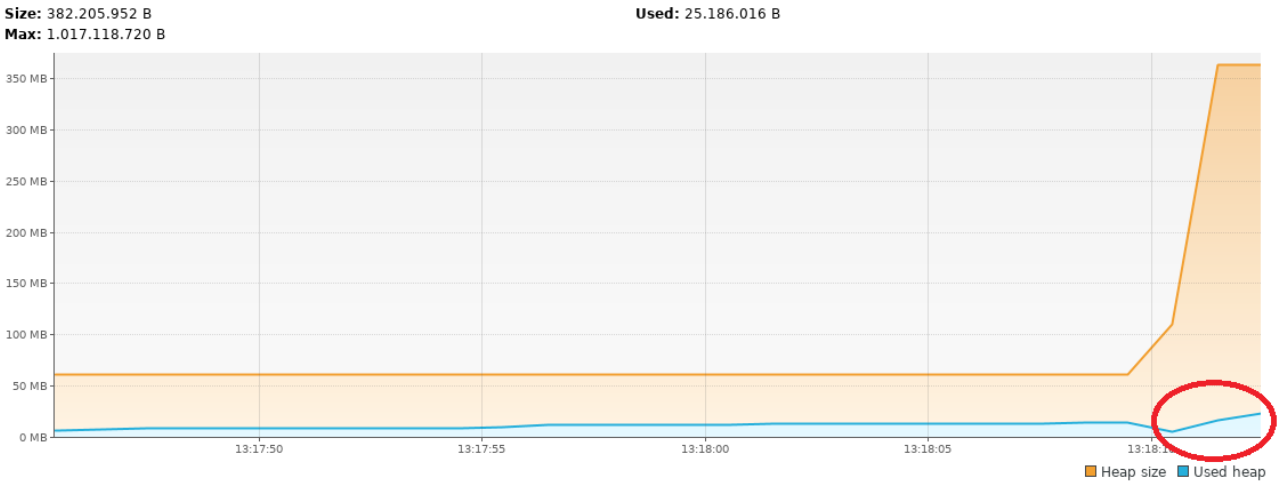

Ilustración 14: Consumo de memoria método merge sort

En este caso la memoria consumida es muy inferior al método anterior y utilizando la misma comparación de gráficas sacamos en conclusión que se trata de una complejidad de memoria de O(n).

Método Rangos

Debido a la larga duración de la ejecución de este método no hemos podido realizar una gráfica que abarque toda la ejecución por lo que traemos dos capturas de distintos momentos, pero nuestro análisis se realiza a partir del código.

Como el método funciona exclusivamente con un nuevo array del mismo tamaño del array original y una única variable entera la memoria consumida total es 2n + 1 por lo que la complejidad total sería O(n).

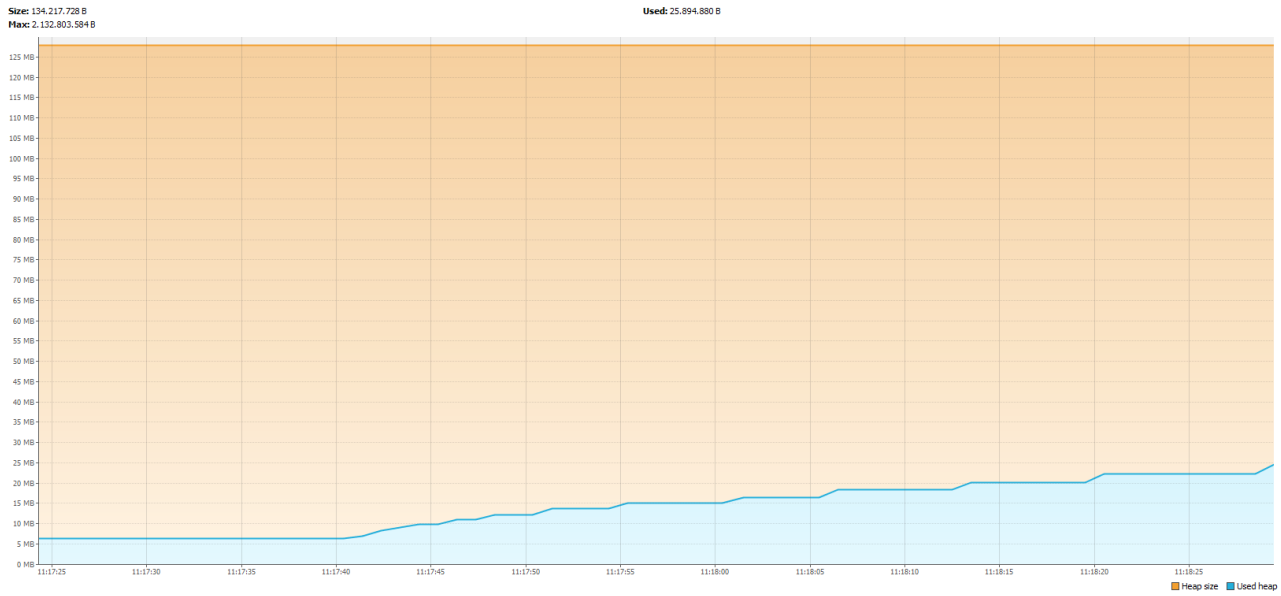

Ilustración 15: Gráfica parcial de la memoria del método rangos

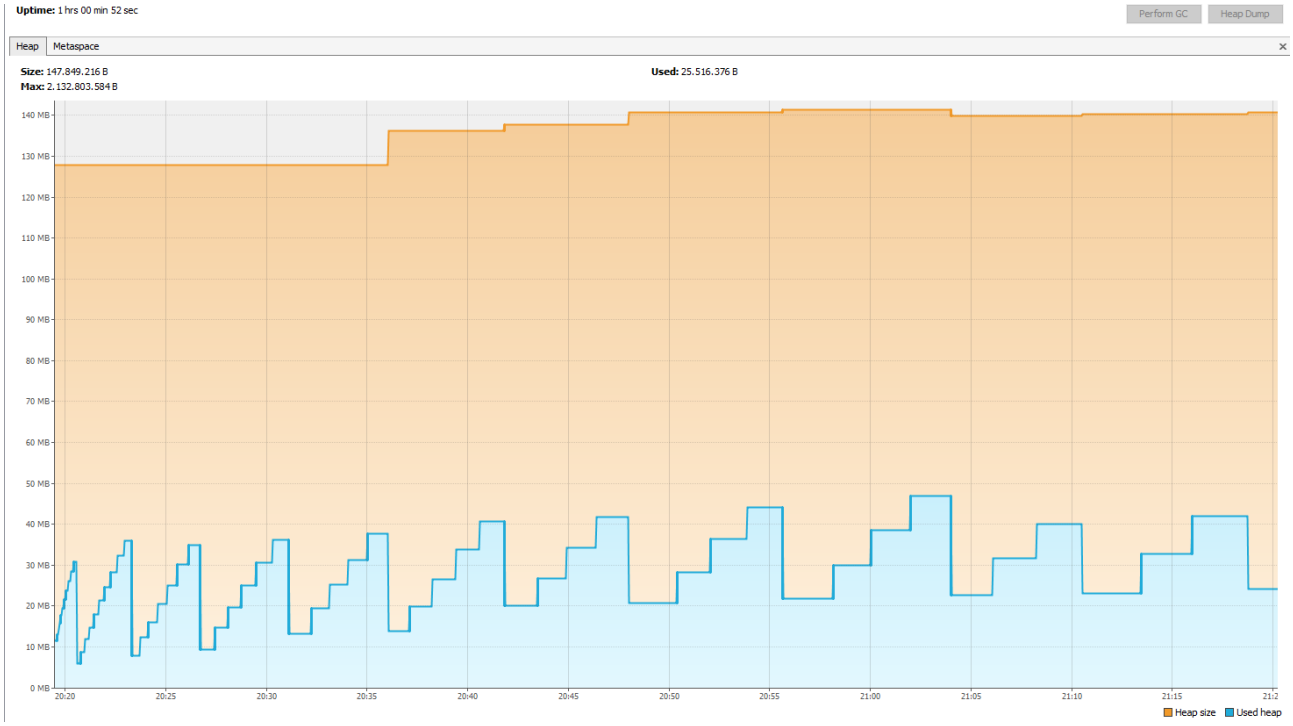

Ilustración 16: Gráfico a una hora de la ejecución del método rangos

En esta gráfica aunque no podemos sacar grandes conclusiones de que complejidad tiene en memoria si podemos observar al recolector de basura y como cada vez hay más memoria utilizándose y como tarda más en actuar el recolector ya que por cada caso es mayor el tiempo que necesita.

Conclusiones

Como conclusiones sacamos que el sistema de rangos es el peor a la hora de ordenar datos (En cuestión de tiempo) a excepción de sistemas paralelos en los que cada número tiene un hilo [1]. Por otro lado el merge sort aunque es rápido no lo es tanto como el método interno de las últimas versiones de java. También hay que destacar que nuestro método se probó solo sobre variables enteras ya que en cadenas (y en otros objetos Comparables) la comparación lleva más tiempo.

Además hemos podido observar que, aunque el método interno es mucho más rápido a la hora de ordenar, este consume más memoria (aunque no una destacablemente alta). Por otro lado el consumo de merge sort y rangos no es muy diferente y están en la misma O.

1 Enlace con el código en paralelo mega

Bibliografía y herramientas

IDE de programación - Eclipse - https://www.eclipse.org/

Lenguaje de programación – Java 8 (openJDK (Linux) y Java SE JDK (Windows))

Análisis estadístico – Statgraphics XVII - http://www.statgraphics.net/

Análisis de memoria – Java Visual VM - https://visualvm.github.io/

Rodriguez Díez, Juan José. Apuntes de Estructuras de Datos

Marticorena Sánchez, Raúl. Apuntes de Metodología de la programación

Represa López, César. Apuntes de Arquitectura de computadores

Lorente Marín, Ana. Apuntes de Estadística

Wikipedia. https://es.wikipedia.org/wiki/Ordenamiento_por_mezcla