PROYECTO FINAL NIVELATORIO DE ALGORITMIA

YENSY HELENA GÓMEZ VILLEGAS
JOHN HAIBER OSORIO RÍOS

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
2014

PROYECTO FINAL NIVELATORIO ALGORITMIA

YENSY HELENA GÓMEZ VILLEGAS
JOHN HAIBER OSORIO RÍOS

Trabajo Presentado a:
Mg. HUGO HUMBERTO MORALES

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
2014

Index of Tables

Illustration Index

INTRODUCCIÓN

En el siguiente trabajo se realizará el proceso de implementación de los metodos de ordenamiento **INSERTION SORT, HEAP INSERT SORT, MERGE SORT, HEAP SORT, QUICK SORT, COUNTING SORT** y **MERGE SORT OPTIMIZADO;** el lenguaje de programación utilizado para este proceso es C.

Se decide utilizar este lenguaje de programación porque es el que mejor se adecúa a los requerimientos solicitados en clase y además tiende a tener mejor desempeño que otros lenguajes de programación como Java o Python. Se utilizará como plataforma de pruebas, un equipo con las siguientes características:

Procesador	Intel Core i7 3770K, 3.9GHz
Memoria RAM	16 GB a 1600MHz
Disco Duro	SSD 256 GB
Sistema Operativo	Ubuntu 12.04 LTS

IMPLEMENTACIÓN

Se construyeron por cada uno de los algoritmos un programa diferente, estos programas reciben por línea de comandos un archivo de texto que contiene el vector de entrada desordenado, el nombre del archivo de salida que contendrá el vector ordenado y la cantidad de datos a ordenar. Es importante aclarar que también se construyó un programa que construye los vectores desordenados, los cuales son escritos en un archivo de texto, para este fin se generaron conjuntos de números utilizando una distribución uniforme. Este programa recibe por linea de comandos el nombre del archivo de salida que contendrá el vector desordenado y la cantidad de datos a generar.

La totalidad de los algoritmos de ordenamiento implementados se encuentran en un repositorio alojado en github cuya url es **https://github.com/kala855/sortingAlgorithms.** Allí puede consultarse el código fuente de la implementación de los métodos de ordenamiento y del programa que genera los números aleatorios.

ANÁLISIS

En esta sección del trabajo se consignarán los datos resultado del análisis de los tiempos de ejecución y del consumo de memoria de cada uno de los algoritmos implementados, es importante aclarar que la **Entrada (n)** que aparece en todas las tablas hace referencia a la cantidad de datos a ordenar que posee el vector de entrada.

Eficiencia en Tiempo de Ejecución Algoritmo Insertion Sort

Para este algoritmo se obtuvieron los siguientes resultados:

Insertion Sort			
Entrada (n)	**Tiempo Real (segundos)**	**Complejidad (n^2)**	**Factor Constante**
100000	6.21	10000000000	6.21E-010
200000	24.77	40000000000	6.1925E-010
300000	56.49	90000000000	6.276666667E-010
400000	100.79	160000000000	6.299375E-010
500000	155.48	250000000000	6.2192E-010
600000	221.7	360000000000	6.158333333E-010
700000	300.69	490000000000	6.136530612E-010
800000	392.66	640000000000	6.1353125E-010
900000	499.82	810000000000	6.170617284E-010
1000000	618.27	1000000000000	6.1827E-010
Promedio Factor Constante			**6.198123540E-010**

Tabla 1: Tiempos Ejecución Insertion Sort

En la Tabla 1 se pueden ver los tiempos obtenidos en la ejecución del algoritmo Insertion Sort, cabe aclarar que este algoritmo tiene una complejidad n^2. También en la Figura 1 se puede ver claramente como la gráfica de la complejidad temporal pareciera ser una parábola, este comportamiento tiene mucho sentido ya que este algoritmo es n^2 en tiempo de ejecución.

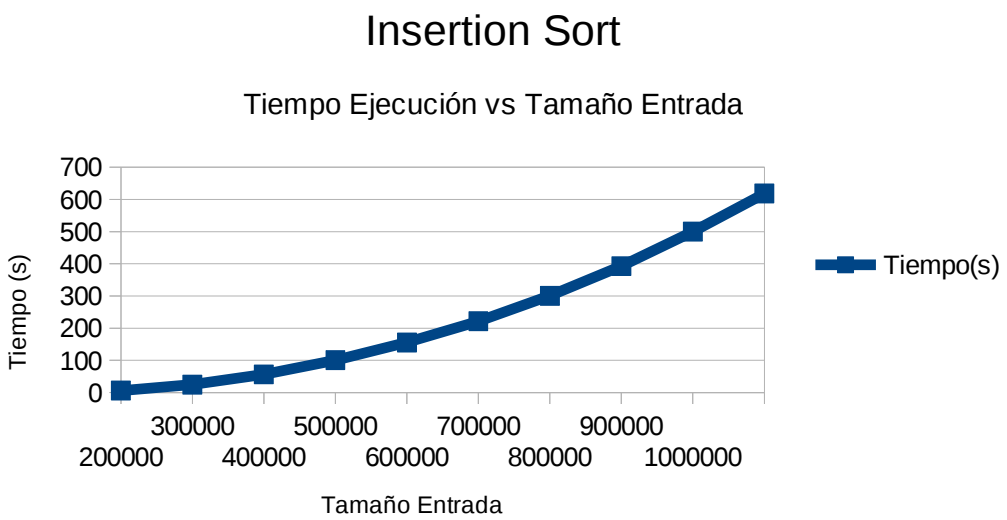

Figura 1: Complejidad en Tiempo de Ejecución Insertion Sort

Con el cálculo del factor constante se puede realizar un proceso de extrapolación a través del cual se puede estimar el tiempo que tardaría el algoritmo para ejecutarse con otros valores de entrada como se puede ver en la Tabla 2 . Es importante aclarar que esto es válido para el equipo en el cuál se realizaron estas pruebas.

Entrada (n)	Tiempo Estimado (s)	Complejidad (n^2)
200000000	24792494.1584782	40000000000000000
210000000	27333724.8097222	44100000000000000
220000000	29998917.9317586	48400000000000000
230000000	32788073.5245874	52900000000000000
240000000	35701191.5882086	57600000000000000
250000000	38738272.1226222	62500000000000000
260000000	41899315.1278282	67600000000000000
270000000	45184320.6038265	72900000000000000
280000000	48593288.5506173	78400000000000000
290000000	52126218.9682004	84100000000000000

Tabla 2: Tiempo de Ejecución Estimado de acuerdo al Factor Constante Calculado

Eficiencia en Tiempo de Ejecución Algoritmo Heap Insert Sort

Este es un algoritmo que combina el método de ordenamiento **Heap Sor**t e **Insertion Sort**, con el fin de obtener tiempos de ejecución bajos en comparación con **Insertion Sort,** la idea detrás de este algoritmo radica en la construcción de un Montón Máximo con el vector a ordenar y posteriormente llamar a la función **Insertion Sort** para que ordene el vector que cumple con las características de montón máximo. En la Tabla 3 se pueden ver los tiempos de ejecución para este algoritmo.

Heap Insert Sort			
Entrada (n)	**Tiempo (s)**	**Complejidad (n^2)**	**Factor Constante**
100000	2.72	10000000000	2.72E-010
200000	10.88	40000000000	2.72E-010
300000	24.4	90000000000	2.71111111111111E-010
400000	43.41	160000000000	2.713125E-010
500000	67.99	250000000000	2.7196E-010
600000	97.87	360000000000	2.71861111111111E-010
700000	133.65	490000000000	2.72755102040816E-010
800000	175.21	640000000000	2.73765625E-010
900000	219.99	810000000000	2.71592592592593E-010
1000000	271.41	1000000000000	2.7141E-010
Promedio Factor			**2.71976804185563E-010**

Tabla 3: Tiempos de Ejecución Heap Insert Sort

La Figura 2 muestra la gráfica de complejidad temporal. Nótese como se dibuja de nuevo una parábola dado que la complejidad de **Heap Insert Sort** es n^2.

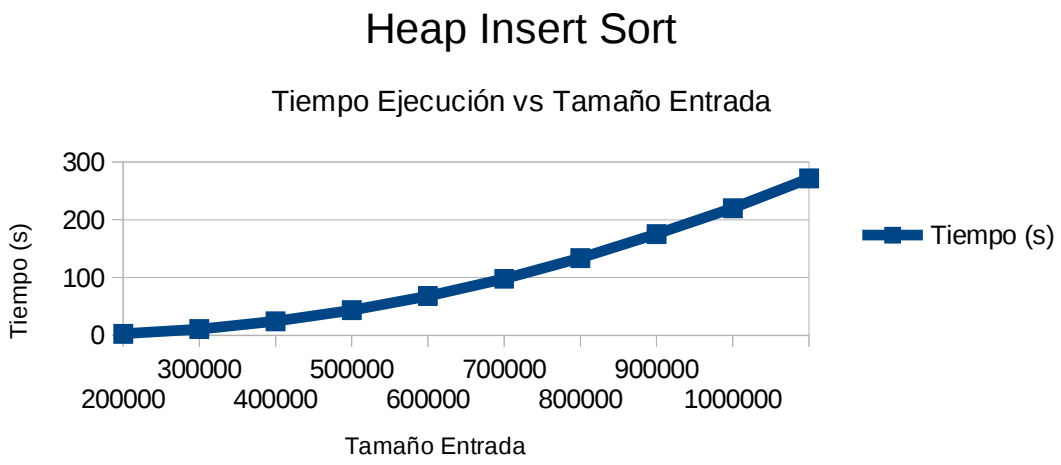

Figura 2: Complejidad Temporal

También se pueden obtener tiempos estimados de la ejecución de este algoritmo para tamaños de entrada mayores, esta información puede verse en la Tabla 4.

Entrada (n)	Tiempo Estimado (s)	Complejidad (n^2)
200000000	10879072.1674225	4.00000000E+016
210000000	11994177.0645833	4.41000000E+016
220000000	13163677.3225813	4.84000000E+016
230000000	14387572.9414163	5.29000000E+016
240000000	15665863.9210884	5.76000000E+016
250000000	16998550.2615977	6.25000000E+016
260000000	18385631.9629441	6.76000000E+016
270000000	19827109.0251276	7.29000000E+016
280000000	21322981.4481481	7.84000000E+016
290000000	22873249.2320059	8.41000000E+016

Tabla 4: Tiempos Estimados por el Factor Constante Heap Insert Sort

Eficiencia en tiempo de ejecución Algoritmo Merge Sort

Se obtienen los datos contenidos en la Tabla 5, este algoritmo tiene una complejidad nlogn, lo cual queda evidenciado en la toma de los datos.

Merge Sort			
Entrada (n)	**Tiempo (s)**	**Complejidad (nlogn)**	**Factor Constante**
100000	0.02	500000	0.00000004
200000	0.04	1060205.9991328	3.77285169417248E-008
300000	0.07	1643136.3764159	4.26014547573269E-008
400000	0.09	2240823.99653118	4.01637969511755E-008
500000	0.11	2849485.00216801	3.86034669129009E-008
600000	0.14	3466890.75023019	4.03820051124209E-008
700000	0.17	4091568.62800998	4.15488570413355E-008
800000	0.19	4722471.98959355	4.02331661084882E-008
900000	0.21	5358818.25849539	3.91877443626838E-008
1000000	0.25	6000000	4.16666666666667E-008
2000000	0.49	12602059.991328	3.8882531930271E-008
3000000	0.78	19431363.764159	4.0141289590734E-008
4000000	1.04	26408239.9653118	3.93816475981011E-008
5000000	1.31	33494850.0216801	3.9110490094808E-008
6000000	1.6	40668907.5023019	3.93420944467082E-008
7000000	1.87	47915686.2800998	0.000000039
8000000	2.16	55224719.8959355	3.91129190708484E-008
9000000	2.45	62588182.5849539	3.91447698081742E-008
10000000	2.74	70000000	3.91428571428571E-008
11000000	3.02	77455319.5367405	0.000000039
12000000	3.31	84950174.9525715	0.000000039
13000000	3.58	92481263.5799889	3.87105437514225E-008
14000000	3.86	100045792.499495	3.85823321857286E-008
15000000	4.14	107641368.885835	3.84610493423852E-008
16000000	4.45	115265919.722495	3.86063808861585E-008
17000000	4.72	122917631.663431	3.83996985308354E-008
18000000	5.05	130594905.09186	3.86691961409051E-008
19000000	5.31	138296318.418104	3.83958160328358E-008
20000000	5.61	146020599.91328	3.8419236760647E-008
Promedio Factor			3.93655051919197E-008

Tabla 5: Tiempos de Ejecución Merge Sort

La Figura 3, muesta la gráfica de los tiempos de ejecución de Merge Sort, se puede observar que efectivamente la gráfica es una función del tipo nlogn.

La Tabla 6, muestra el estimado de los tiempos para la ejecución del algoritmo Merge Sort utilizando el factor constante.

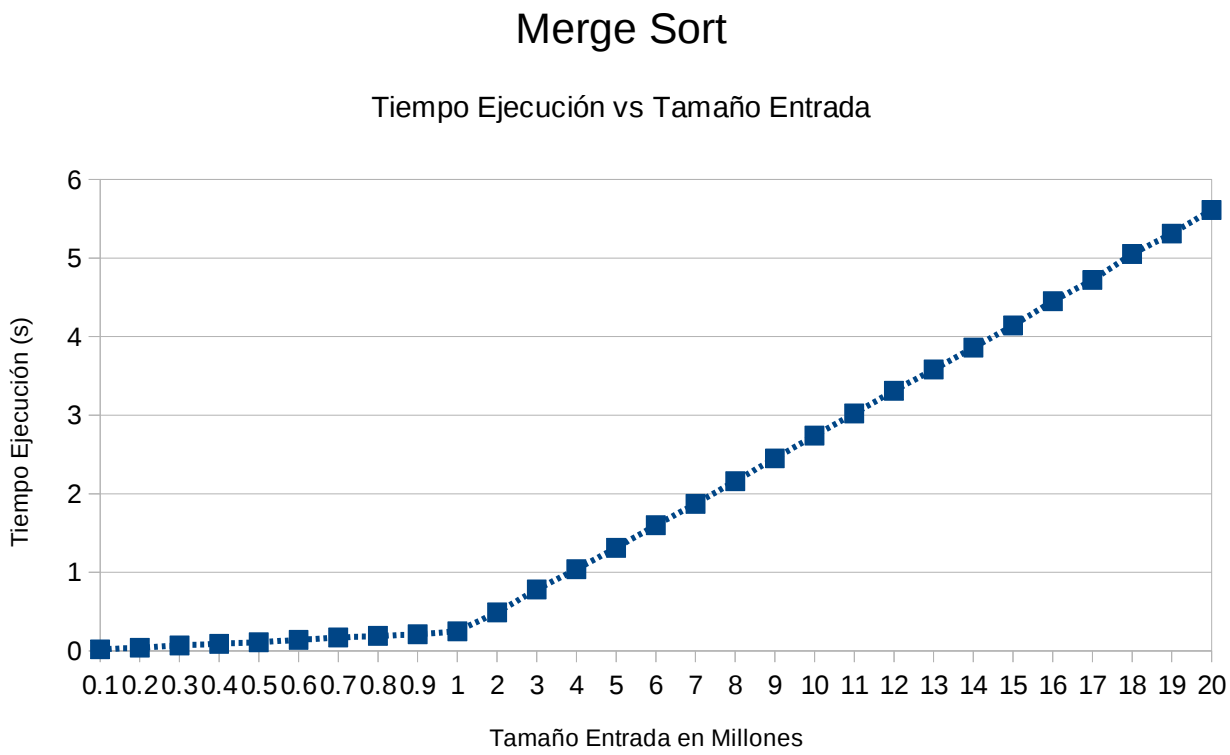

Figura 3: Merge Sort

Entrada (n)	Tiempo Estimado (s)	Complejidad (nlogn)
200000000	65.3548478785	1660205999.1328
210000000	68.7977570396	1747666051.89412
220000000	72.2488103381	1835332989.78089
230000000	75.7076373247	1923197402.28405
240000000	79.1738997948	2011250698.01079
250000000	82.6472877532	2099485002.16801
260000000	86.127516025	2187893070.47241
270000000	89.6143213889	2276468216.32293
280000000	93.1074601351	2365204248.77582
290000000	96.6067059735	2454095419.3907

Tabla 6: Tiempos Estimados Merge Sort

Eficiencia en Tiempo de Ejecución Heap Sort

Para este algoritmo se encontraron los datos consignados en la Tabla 7

Heap Sort			
Entrada (n)	**Tiempo(s)**	**Complejidad (nlogn)**	**Factor Constante**
100000	0.02	500000	0.00000004
200000	0.06	1060205.9991328	5.65927754125872E-008
300000	0.09	1643136.3764159	5.4773298973706E-008
400000	0.13	2240823.99653118	0.000000058
500000	0.16	2849485.00216801	5.61504973278558E-008
600000	0.2	3466890.75023019	5.76885787320298E-008
700000	0.23	4091568.62800998	5.62131595265128E-008
800000	0.27	4722471.98959355	5.717344657522E-008
900000	0.31	5358818.25849539	5.78485750115809E-008
1000000	0.35	6000000	5.83333333333334E-008
2000000	0.75	12602059.991328	5.95140794851087E-008
3000000	1.19	19431363.764159	6.12411982217608E-008
4000000	1.7	26408239.9653118	6.43738470353575E-008
5000000	2.19	33494850.0216801	6.5383185731015E-008
6000000	2.74	40668907.5023019	6.73733367399878E-008
7000000	3.28	47915686.2800998	6.84535744897019E-008
8000000	3.85	55224719.8959355	6.97151566772067E-008
9000000	4.38	62588182.5849539	0.00000007
10000000	5	70000000	7.14285714285714E-008
11000000	5.57	77455319.5367405	7.19124268457495E-008
12000000	6.19	84950174.9525715	7.28662419289417E-008
13000000	6.79	92481263.5799889	7.34202771151282E-008
14000000	7.41	100045792.499495	7.40660832891836E-008
15000000	8.04	107641368.885835	7.46924726359366E-008
16000000	8.66	115265919.722495	7.51306198818275E-008
17000000	9.31	122917631.663431	7.57417782462028E-008
18000000	9.92	130594905.09186	0.000000076
19000000	10.59	138296318.418104	7.65747065513619E-008
20000000	11.26	146020599.91328	7.71124074732416E-008
Promedio Factor			**6.54389430443571E-008**

Tabla 7: Tiempos de Ejecución Heap Sort

Como se puede observar este algoritmo tiene unos tiempos de ejecución un poco mas elevados si se compara con el algoritmo Merge Sort, sin embargo el algoritmo de ordenamiento por Montones tiene una complejidad espacial menor, esto se podrá verificar en el siguiente capítulo.

En la Figura 4, se puede ver la gráfica de complejidad temporal para este algoritmo, y la tabla Tabla 8 muestra los tiempos estimados para el algoritmo Heap Sort.

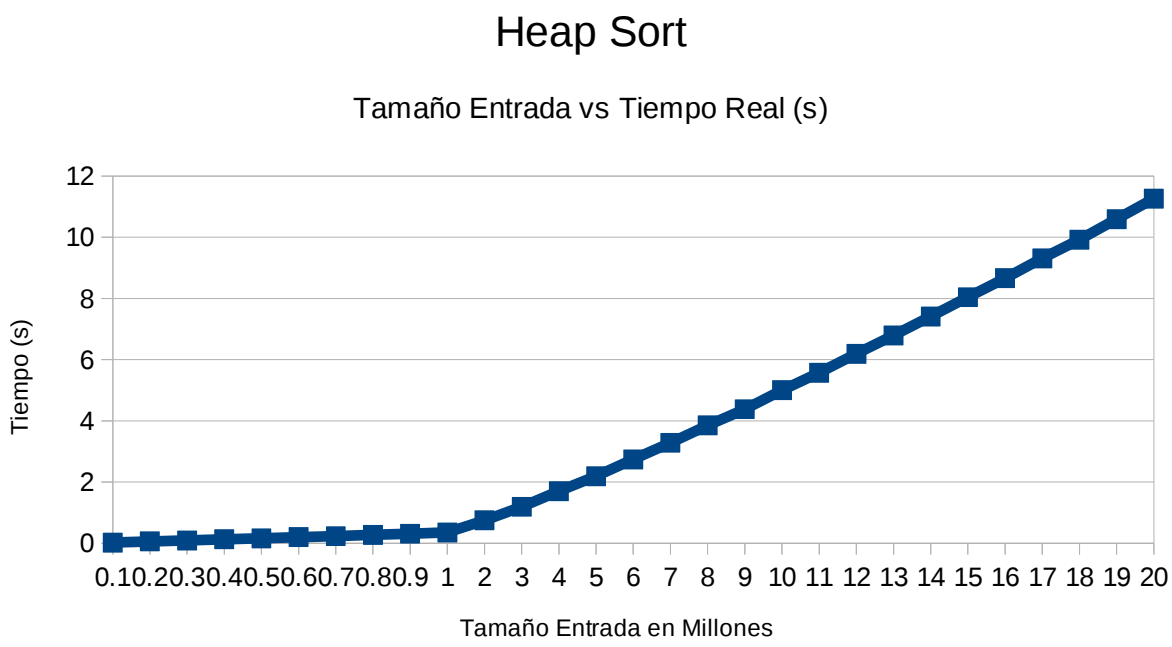

Figura 4: Tiempo Ejecución Heap Sort

Entrada (n)	Tiempo Estimado (s)	Complejidad (nlogn)
200000000	108.6421258192	1660205999.1328
210000000	114.3654192305	1747666051.89412
220000000	120.1022509857	1835332989.78089
230000000	125.8520052711	1923197402.28405
240000000	131.6141198751	2011250698.01079
250000000	137.3880794794	2099485002.16801
260000000	143.1734100258	2187893070.47241
270000000	148.9696739502	2276468216.32293
280000000	154.7764661239	2365204248.77582
290000000	160.5934103749	2454095419.3907

Tabla 8: Tiempos Estimados Algoritmo Heap Sort

Eficiencia en Tiempo de Ejecución Quick Sort

Quick Sort			
Entrada (n)	**Tiempo Real (segundos)**	**COmplejidad (nlogn)**	**Factor Constante**
0.1	0.01	500000	0.00000002
0.2	0.02	1060205.9991328	1.88642584708624E-008
0.3	0.04	1643136.3764159	2.43436884327582E-008
0.4	0.06	2240823.99653118	2.6775864634117E-008
0.5	0.07	2849485.00216801	2.45658425809369E-008
0.6	0.08	3466890.75023019	2.30754314928119E-008
0.7	0.1	4091568.62800998	2.44405041419621E-008
0.8	0.11	4722471.98959355	2.32928856417563E-008
0.9	0.12	5358818.25849539	2.23929967786765E-008
1	0.15	6000000	0.000000025
2	0.3	12602059.991328	2.38056317940435E-008
3	0.45	19431363.764159	2.31584363023465E-008
4	0.61	26408239.9653118	2.309885099504E-008
5	0.78	33494850.0216801	2.32871620411834E-008
6	0.95	40668907.5023019	2.3359368577733E-008
7	1.11	47915686.2800998	2.31656913669418E-008
8	1.29	55224719.8959355	2.335910444509E-008
9	1.46	62588182.5849539	2.33270873142589E-008
10	1.64	70000000	2.34285714285714E-008
11	1.81	77455319.5367405	2.33683110575954E-008
12	1.99	84950174.9525715	2.34254961936339E-008
13	2.2	92481263.5799889	2.37886023053435E-008
14	2.33	100045792.499495	2.32893352312818E-008
15	2.54	107641368.885835	2.35968756834924E-008
16	2.72	115265919.722495	2.35976080922137E-008
17	2.96	122917631.663431	2.40811668752696E-008
18	3.08	130594905.09186	2.35843810126708E-008
19	3.31	138296318.418104	2.39341150788487E-008
20	3.43	146020599.91328	2.34898363795043E-008
Factor Constante			**2.34102449775498E-008**

Tabla 9: Tiempos Ejecución Quick Sort (Entrada en Millones de Datos)

Este algoritmo generalmente es el más utilizado porque tiene una complejidad promedio nlogn y adicionalmente realiza el proceso de ordenamiento en sitio, lo que lo hace bastante interesante de utilizar en sistemas con pocos recursos en memoria.

En la Figura 5 se puede observar la complejidad temporal para este algoritmo.

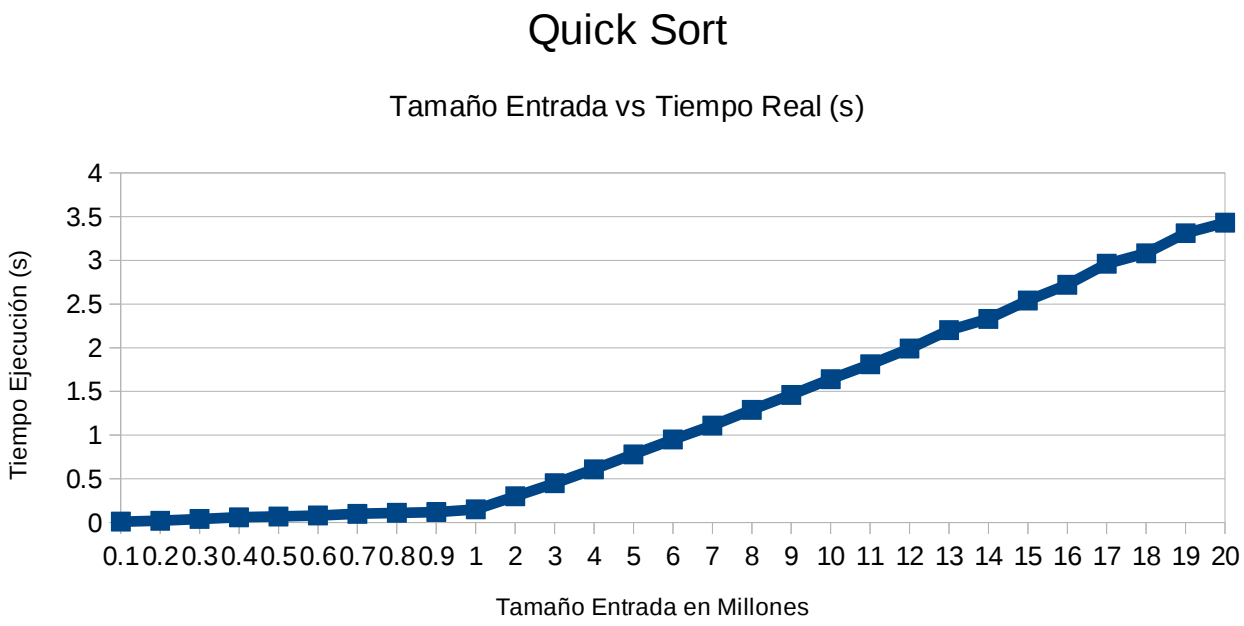

Figura 5: Quick Sort

La siguiente tabla muestra los valores estimados para un conjunto de vectores de tamaño n. Para este cálculo se usó el valor del factor constante obtenido en la Tabla 9.

Entrada (n)	Tiempo Estimado (s	Complejidad (n+k)
200000000	70.7881282694	1660205999.1328
210000000	74.5172639529	1747666051.89412
220000000	78.2552208374	1835332989.78089
230000000	82.0015976761	1923197402.28405
240000000	85.7560281478	2011250698.01079
250000000	89.5181764858	2099485002.16801
260000000	93.2877338073	2187893070.47241
270000000	97.0644150078	2276468216.32293
280000000	100.847956117	2365204248.77582
290000000	104.6381120319	2454095419.3907

Tabla 10: Tiempo Estimado Quick Sort

Eficiencia en Tiempo de Ejecución Couting Sort.

Para este algoritmo se obtuvieron los tiempos consignados en la Tabla 11.

El algoritmo Counting Sort tiene una complejidad n, esta linealidad lo hace un algoritmo bastante rápido para ordenar valores en donde el tamaño del vector a ordenar tienda a ser igual al máximo valor que se guarda en el vector.

En la Figura 6 la se puede obervar la complejidad temporal para este algoritmo, observándose la tendencia lineal.

Counting Sort			
Entrada (n)	**Tiempo Real (s)**	**Complejidad (n+k)**	**Factor Constante**
100000	0.02	3100000	6.45161290322581E-009
200000	0.03	3200000	9.375E-009
300000	0.04	3300000	1.21212121212121E-008
400000	0.04	3400000	1.17647058823529E-008
500000	0.05	3500000	1.42857142857143E-008
600000	0.05	3600000	1.38888888888889E-008
700000	0.06	3700000	1.62162162162162E-008
800000	0.07	3800000	1.8421052631579E-008
900000	0.08	3900000	2.05128205128205E-008
1000000	0.08	4000000	0.00000002
2000000	0.17	5000000	0.000000034
3000000	0.24	6000000	0.00000004
4000000	0.32	7000000	4.57142857142857E-008
5000000	0.39	8000000	4.875E-008
6000000	0.47	9000000	5.22222222222222E-008
7000000	0.54	10000000	0.000000054
8000000	0.62	11000000	5.63636363636364E-008
9000000	0.7	12000000	5.83333333333333E-008
10000000	0.78	13000000	0.00000006
11000000	0.85	14000000	6.07142857142857E-008
12000000	0.92	15000000	6.13333333333333E-008
13000000	0.98	16000000	6.125E-008
14000000	1.08	17000000	6.35294117647059E-008
15000000	1.15	18000000	6.38888888888889E-008
16000000	1.23	19000000	6.47368421052632E-008
17000000	1.32	20000000	0.000000066
18000000	1.39	21000000	6.61904761904762E-008
19000000	1.5	22000000	6.81818181818182E-008
20000000	1.57	23000000	6.82608695652174E-008
Factor Constante			**4.26381595454992E-008**

Tabla 11: Tiempos Ejecución Counting Sort

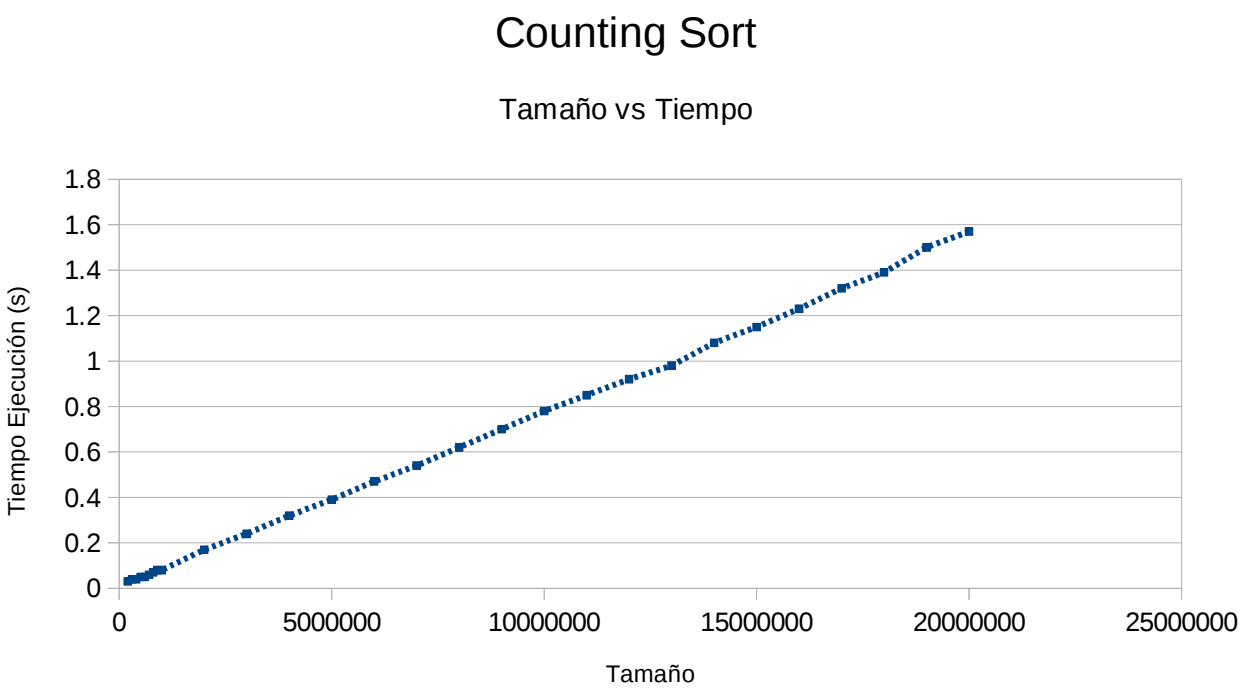

Figura 6: Complejidad Counting Sort

La siguiente tabla, muestra los tiempos de ejecución estimados para otro conjunto de entradas mayores.

Entrada (n)	Tiempo Estimado (s)	Complejidad (n+k)
200000000	70.7881282694	1660205999.1328
210000000	74.5172639529	1747666051.89412
220000000	78.2552208374	1835332989.78089
230000000	82.0015976761	1923197402.28405
240000000	85.7560281478	2011250698.01079
250000000	89.5181764858	2099485002.16801
260000000	93.2877338073	2187893070.47241
270000000	97.0644150078	2276468216.32293
280000000	100.847956117	2365204248.77582
290000000	104.6381120319	2454095419.3907

Tabla 12: Tiempos Estimados Algoritmo Counting Sort

Eficiencia en Tiempo de Merge Sort Optimizado

El algoritmo Merge Sort Optimizado, busca disminuir la complejidad espacial del algoritmo Merge Sort Tradicional. En la Tabla 13, se muestran los datos obtenidos.

Merge Sort Optimizado			
Entrada (n)	**Tiempo(s)**	**Complejidad (nlogn)**	**Factor Constante**
0.1	0.02	500000	0.00000004
0.2	0.03	1060205.9991328	2.82963877062936E-008
0.3	0.05	1643136.3764159	3.04296105409478E-008
0.4	0.07	2240823.99653118	3.12385087398031E-008
0.5	0.1	2849485.00216801	3.50940608299099E-008
0.6	0.11	3466890.75023019	3.17287183026164E-008
0.7	0.13	4091568.62800998	3.17726553845507E-008
0.8	0.15	4722471.98959355	3.17630258751222E-008
0.9	0.16	5358818.25849539	2.98573290382353E-008
1	0.2	6000000	3.33333333333333E-008
2	0.4	12602059.991328	3.1740842392058E-008
3	0.61	19431363.764159	3.13925469876253E-008
4	0.84	26408239.9653118	3.18082538292355E-008
5	1.06	33494850.0216801	3.16466561072492E-008
6	1.3	40668907.5023019	0.000000032
7	1.52	47915686.2800998	3.17223881781546E-008
8	1.73	55224719.8959355	3.13265509224851E-008
9	1.97	62588182.5849539	3.1475590417185E-008
10	2.2	70000000	3.14285714285714E-008
11	2.45	77455319.5367405	3.16311392768557E-008
12	2.68	84950174.9525715	3.15479044215773E-008
13	2.92	92481263.5799889	3.15739630598195E-008
14	3.15	100045792.499495	3.14855819650376E-008
15	3.39	107641368.885835	3.14934679397792E-008
16	3.62	115265919.722495	3.14056401815491E-008
17	3.84	122917631.663431	3.12404327030525E-008
18	4.11	130594905.09186	3.1471365572103E-008
19	4.34	138296318.418104	3.13818910701521E-008
20	4.57	146020599.91328	3.12969540100101E-008
Factor Promedio			**3.1812028343147E-008**

Tabla 13: Tiempos Ejecución Merge Sort Optimizado

Se puede observar que los tiempos de ejecución son muy similares al compararlos con Merge Sort Tradicional. A nivel temporal son algoritmos muy similares. En la Figura 7, se puede observar la gráfica de la complejidad de este algoritmo. Por otro lado la tabla tal, muestra los tiempos estimados para un conjunto de entradas con una cantidad de datos mayor.

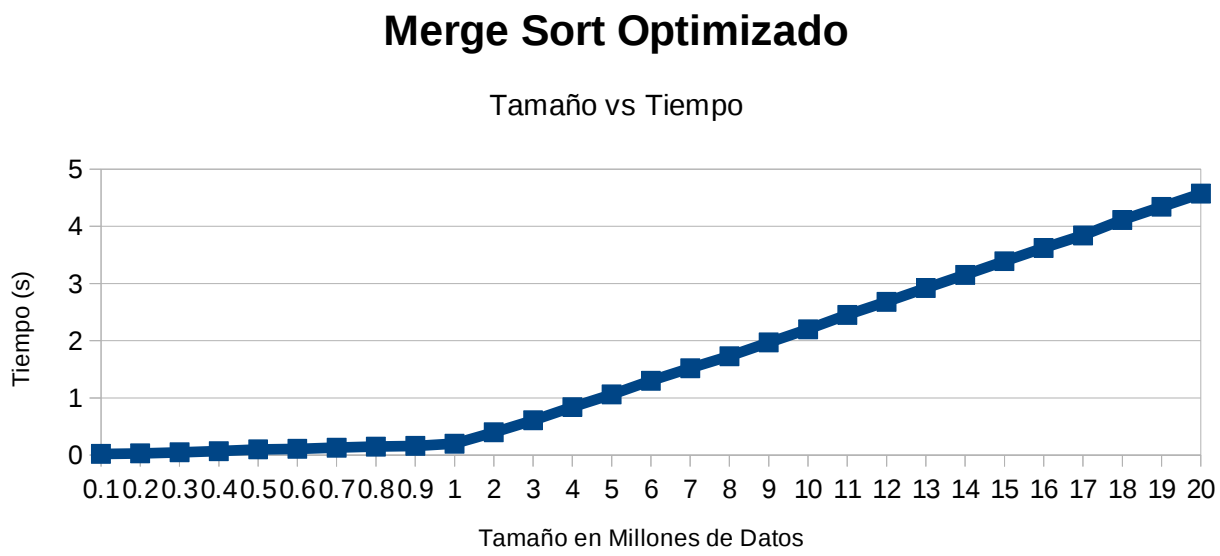

Figura 7: Complejidad Temporal Merge Sort Optimizado

Entrada (n)	Tiempo Estimado (s)	Complejidad (nlogn)
200000000	52.8145202999	1660205999.1328
210000000	55.5968019772	1747666051.89412
220000000	58.38566509	1835332989.78089
230000000	61.1808102709	1923197402.28405
240000000	63.9819642103	2011250698.01079
250000000	66.788876395	2099485002.16801
260000000	69.6013163696	2187893070.47241
270000000	72.4190714199	2276468216.32293
280000000	75.2419445994	2365204248.77582
290000000	78.0697530384	2454095419.3907

Tabla 14: Tiempo Estimado Algoritmo Merge Sort Optimizado

Eficiencia Espacial Algoritmo Insertion Sort

Este algoritmo tiene una complejidad espacial lineal, no necesita de vectores auxiliares para realizar el ordenamiento. La Figura 8 muestra este comportamiento.

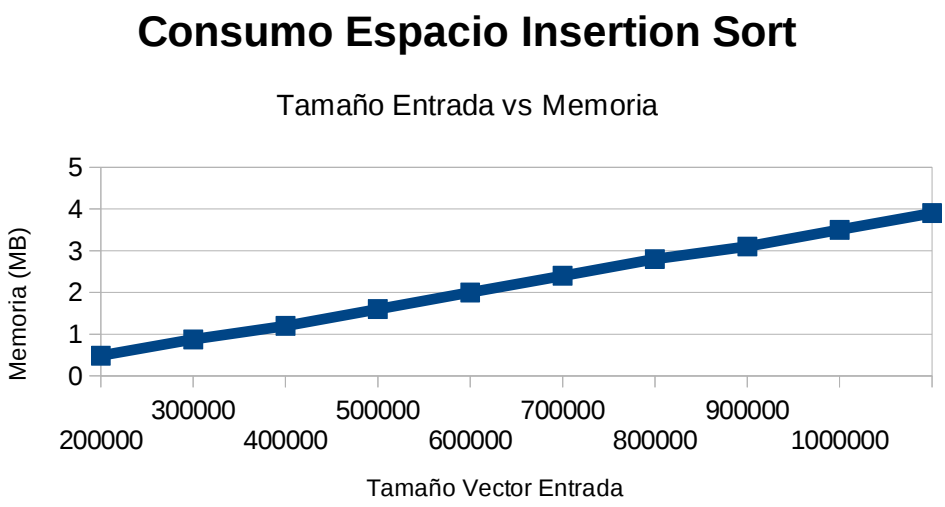

Figura 8: Complejidad Espacial Insertion Sort

Eficiencia Espacial Algoritmo Heap Insert Sort

Este algoritmo tiene un consumo de memoria lineal, los vectores de entrada tienen un tamaño que oscila entre 100000 y 1000000 de datos, esto se traduce en que la memoria consumida debe ser máximo unos 4 MB, ya que se estan almacenando vectores de enteros de 32 bits, esto se puede confirmar en la Figura 9.

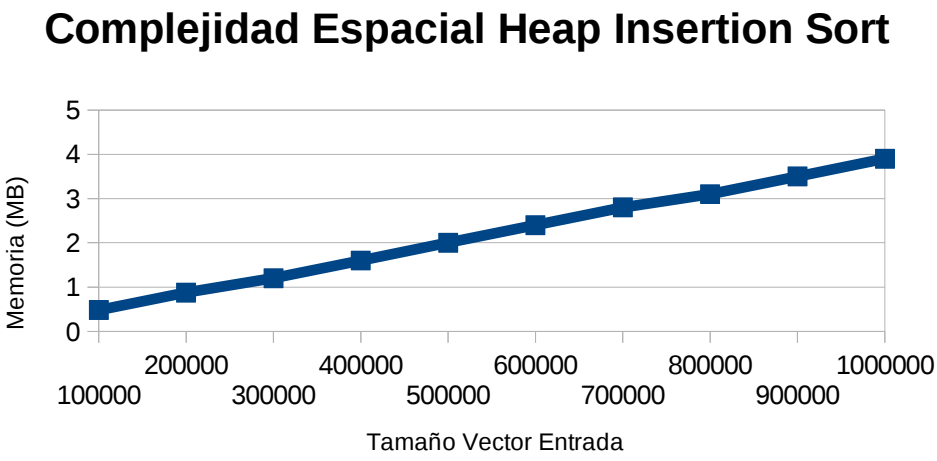

Figura 9: Consumo de Memoria Heap Insertion Sort

Eficiencia Espacial Algoritmo Merge Sort

Para los algoritmos con complejidad nlogn y n, se utilizaron vectores de entrada con un tamaño que oscila entre 100000 y 20000000 de datos, en las figuras se ve que los datos de entrada oscilan entre 10 millones y 20 millones esto es porque se utilizó como herramienta de medición de memoria el programa **top** de entornos UNIX, esta aplicación necesita que el tiempo de ejecución del algoritmo esté por encima de un segundo, ya que en caso contrario no se alcanzaría a visualizar el consumo de memoria.

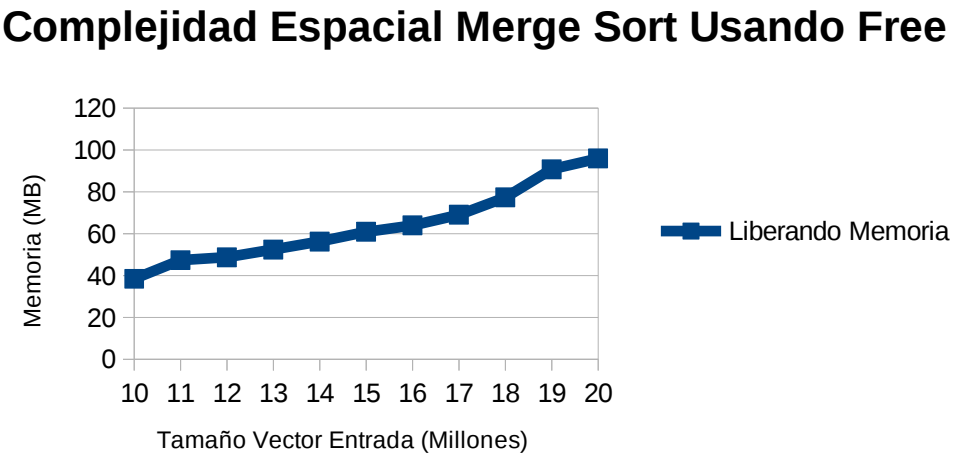

Figura 10: Complejidad Espacial Usando Free Merge Sort

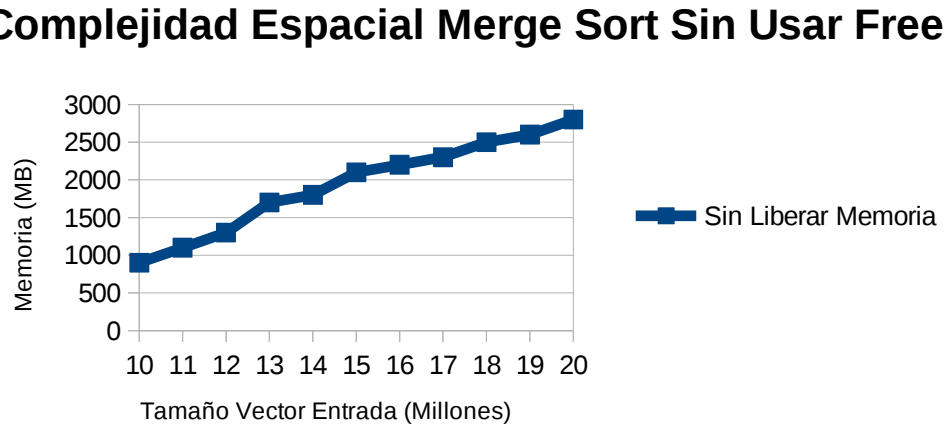

Figura 11: Complejidad Espacial Sin Liberar Memoria Merge Sort

La Figura 10 muestra la complejidad espacial del algoritmo **Merge Sort** usando llamados a la función **free** en **C** antes de retornar del metodo **merge** implementado, en este caso y debido al uso de la aplicación **top** no se visualiza un consumo elevado de memoria, ya que **top** no alcanza a sensar el consumo por completo en el método e incluso pareciera inicialmente que el algoritmo **Merge Sort Optimizado** consumiera más.

La Figura 11 muestra un consumo de memoria que tiene mucho más sentido, en este caso no se realizan llamados a **free** dentro del metodo **merge,** esto traduce en que se podrá ver de una manera un poco más acertada el consumo total de cada uno de los llamados, es por esto que se ve un consumo de memoria de casi 3 GB.

Eficiencia Espacial Heap Sort

En la Figura 12 se ve que la complejidad espacial para este método es lineal, de nuevo para la medición se utiliza **top.**

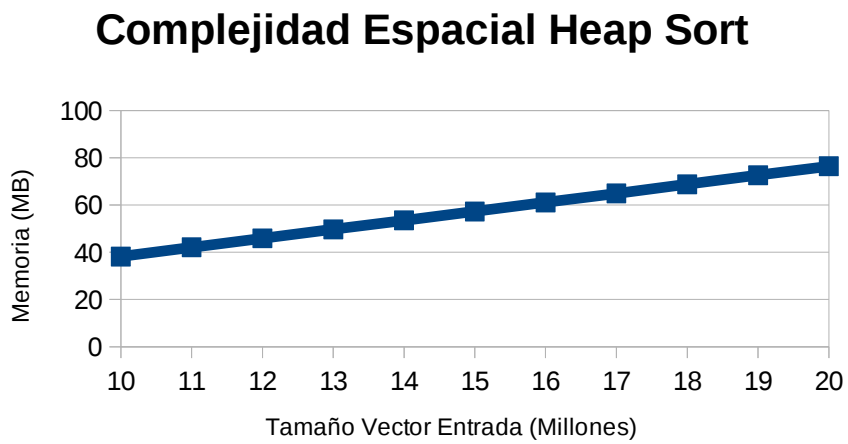

Figura 12: Consumo Espacial Heap Sort

Eficiencia Espacial Quick Sort

En la Figura 13 se puede ver de nuevo que este método tiene una complejidad espacial n, este método de ordenamiento vale la pena repetir que es el más utilizado debido a su complejidad temporal, espacial y a que la constante oculta es mas pequeña que en los demás casos.

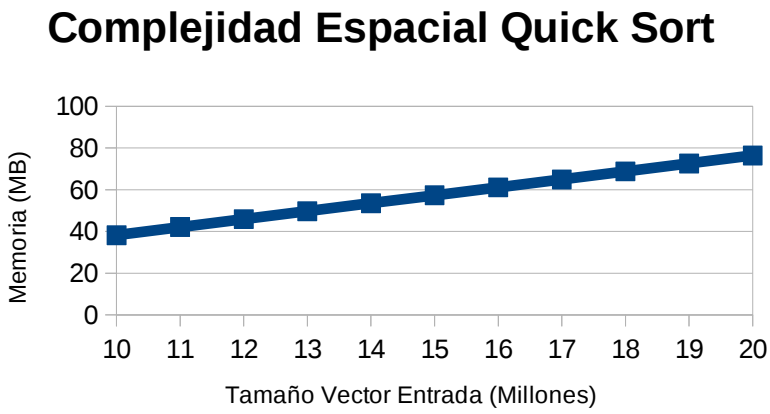

Figura 13: Complejidad Espacial Quick Sort

Eficiencia Espacial Counting Sort

La Figura 14 muestra el consumo de memoria del algoritmo **Counting Sort,** se ve claramente que el consumo de memoria es del doble comparado con los algoritmos **Heap Sort** y **Quick Sort**.

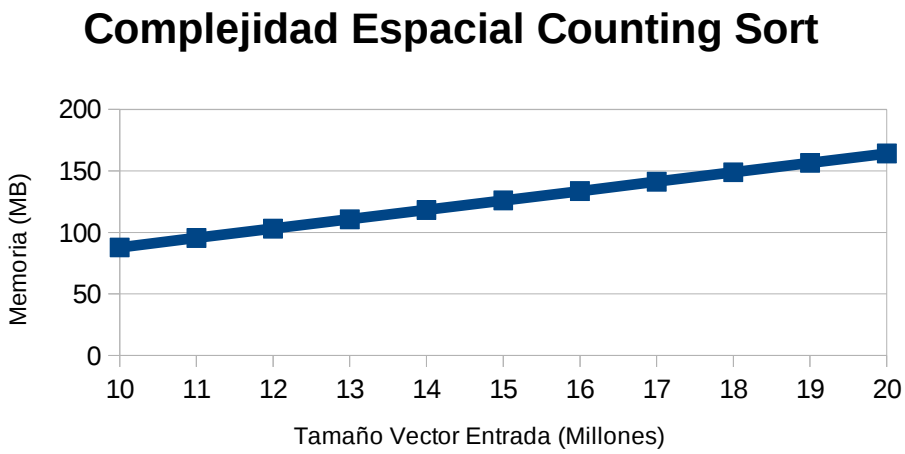

Figura 14: Consumo de Memoria Counting Sort

Eficiencia Espacial Merge Sort Optimizado

En la Figura 15 se observa la complejidad del algoritmo **Merge Sort Optimizado,** al compararse con la Figura 11 se ve entonces que el consumo de Merge Sort Optimizado es mas bajo.

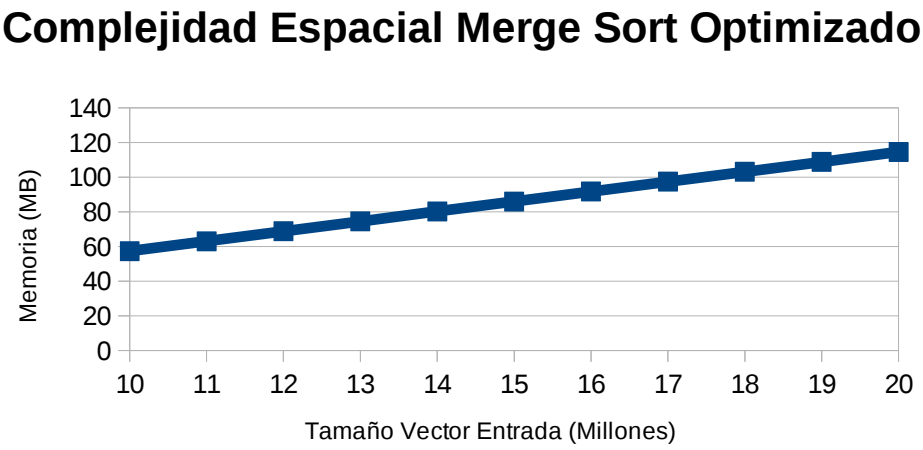

Figura 15: Consumo de Memoria Merge Sort Optimizado

CONCLUSIONES

Se implementaron siete algoritmos de ordenamiento: **Insertion Sort, Heap Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Counting Sort y Merge Sort Optimizado**. Se pudo verificar a nivel práctico la complejidad temporal y espacial de cada uno de los métodos.

La Figura 16, muestra los tiempos de los algoritmos con complejidad **n^2**. Se puede concluir que el **factor constante** del algoritmo **Insertion Sort** es mayor que el de **Heap Insert Sort,** esto se puede verificar en la Tabla 1 y la Tabla 3.

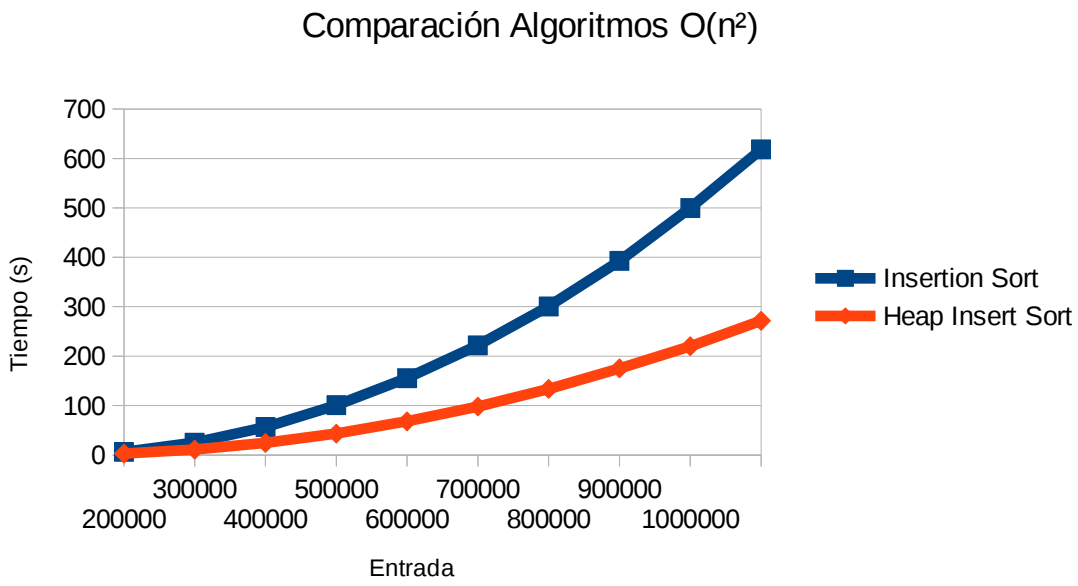

Figura 16: Tiempos de Ejecución Algoritmos Complejidad n^2

La Figura 17, muestra los tiempos de los algoritmos con complejidad **n+k** y **nlogn**. Se puede decir entonces que de todos los algoritmos evaluados el que tiene mejor eficiencia temporal es **Counting Sort**, sin embargo su consumo de memoria lo hace menos utilizado comparado con **Quick Sort**, el cual tiene una complejidad temporal promedio **nlogn** y espacial **n,** adicional a esto tiene un **factor constante** menor que **Counting Sort**.

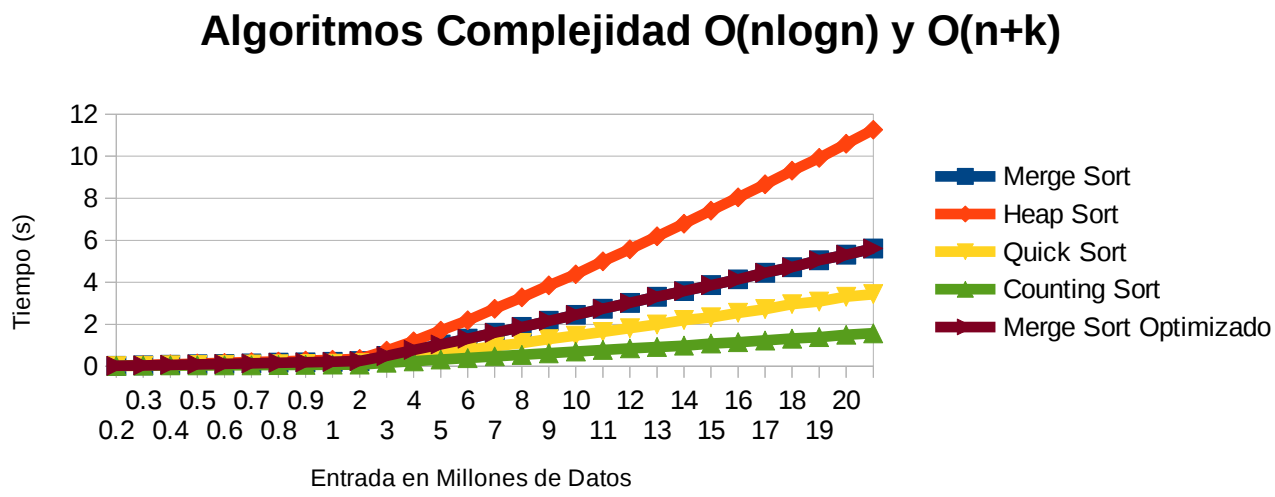

Figura 17: Complejidad Algoritmos O(nlogn) y O(n+k)