INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

ANALISIS DE ALGORITMOS

PRACTICA 1:

" PRUEBAS A POSTERIORI"

EQUIPO: GALLOS DE ORO

INTEGRANTES:

- MORALES HERNANDEZ CARLOS JESUS
- PACHECO MIJANGOS MAURICIO JESUS
- PALACIOS CABRERA ALBERTO
- PÉREZ PRIEGO MARIO DANIEL

PROFESOR: FRANCO MARTINEZ EDGARDO ADRIAN

GRUPO: 3CM13

ÍNDICE

ÍNDICE DE FIGURAS

Objetivo

Realizar un análisis de algoritmos a posteriori de los algoritmos de ordenamiento más conocidos en la computación y realizar una aproximación a sus funciones de complejidad temporal.

Planteamiento del problema

Realizar un análisis temporal de 9 algoritmos de ordenamiento: burbuja simple, burbuja optimizada 1, burbuja optimizada 2, inserción, Shell, árbol binario de búsqueda, mergesort y Quicksort y a partir de los resultados obtenidos, hacer una comparativa entre ellos y realizar una aproximación a las funciones de complejidad de cada una de ellas.

Para ello, implementar cada algoritmo en lenguaje C, hacer uso de las librerías en el sistema operativo Linux y de esta forma contabilizar el tiempo de ejecución de cada algoritmo con diferentes tamaños de problema obtenidos a partir de un archivo que incluye 10 millones de números y asimismo obtener su función de complejidad utilizando herramientas para graficar en Excel y las proporcionadas por Matlab.

Comparación de algoritmos de ordenamiento

ALGORITMO	TIEMPO REAL	TIEMPO CPU	TIEMPO E/S	CPU/WALL
BURBUJA SIMPLE	1.22E+03	1.21E+03	4.51E-01	99.71%
BURBUJA OPTIMIZADA 1	8.97E+02	8.96E+02	1.78E-01	99.88%
BURBUJA OPTIMIZADA 2	8.81E+02	8.80E+02	3.27E-03	99.92%
INSERCIÓN	1.86E+02	1.86E+02	6.74E-02	99.85%
SELECCIÓN	2.91E+02	2.90E+02	1.79E-01	99.67%
SHELL	7.55E-01	7.53E-01	0.00E+00	99.72%
ORDENAMIENTO ABB	3.38E-01	3.27E-01	1.01E-02	99.74%
MERGESORT	1.11E-01	1.11E-01	0.00E+00	99.94%
QUICKSORT	7.93E-02	7.90E-02	0.00E+00	99.59%

Figura 1. Tabla comparativa de n=500000

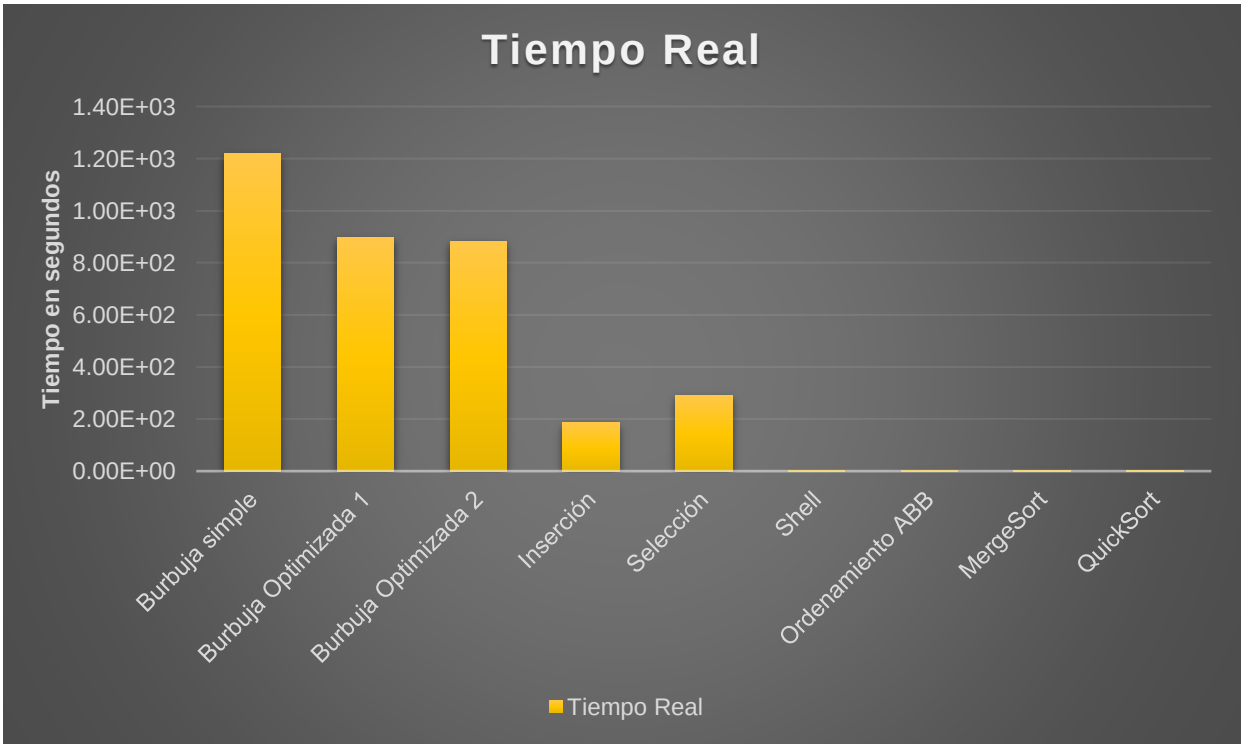

Figura 2. Tiempo real de los algoritmos

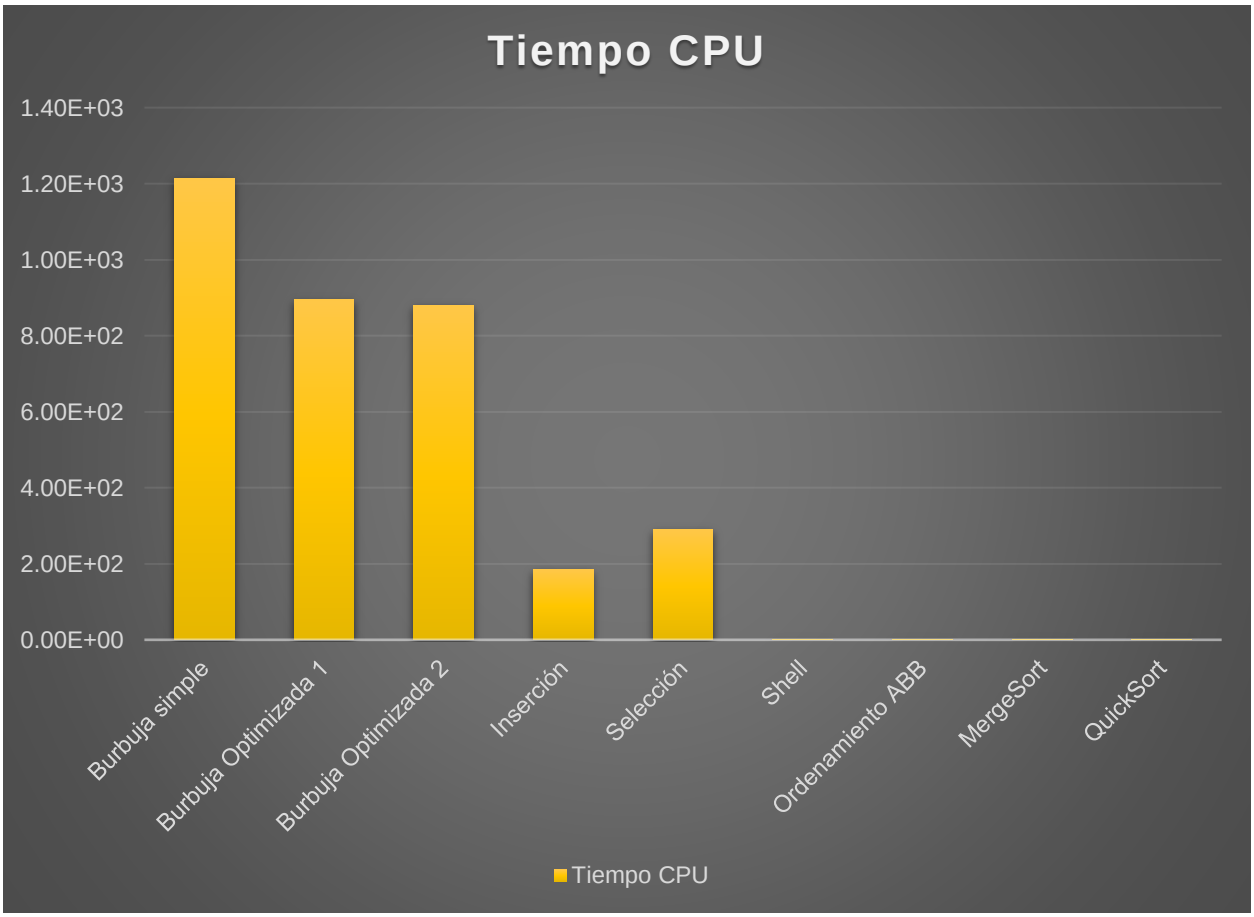

Figura 3. Tiempo de CPU de los algoritmos

Análisis temporal

Burbuja Simple

N	REAL	USUARIO	E/S	CPU/WALL
100	4.79E-05	0.00E+00	6.90E-05	143.98%
1000	4.43E-03	4.42E-03	0.00E+00	99.86%
2000	1.53E-02	1.52E-02	0.00E+00	99.52%
3000	3.85E-02	3.82E-02	0.00E+00	99.44%
5000	1.02E-01	1.00E-01	1.86E-03	99.70%
8000	2.72E-01	2.72E-01	0.00E+00	99.88%
10000	4.41E-01	4.41E-01	0.00E+00	99.89%
20000	1.93E+00	1.92E+00	0.00E+00	99.83%
30000	3.74E+00	3.74E+00	0.00E+00	99.93%
40000	7.34E+00	7.33E+00	0.00E+00	99.90%
50000	1.14E+01	1.13E+01	0.00E+00	99.91%
60000	1.76E+01	1.75E+01	0.00E+00	99.90%
70000	2.30E+01	2.30E+01	3.17E-03	99.85%
80000	3.09E+01	3.08E+01	6.57E-03	99.86%
90000	3.90E+01	3.90E+01	6.83E-03	99.82%
100000	4.81E+01	4.80E+01	3.33E-03	99.91%
200000	1.94E+02	1.93E+02	4.95E-02	99.84%
300000	4.38E+02	4.37E+02	1.82E-01	99.80%
400000	7.79E+02	7.78E+02	1.89E-01	99.84%
500000	1.22E+03	1.21E+03	4.51E-01	99.71%

Figura 4. Tabla comparativa para el algoritmo de burbuja simple

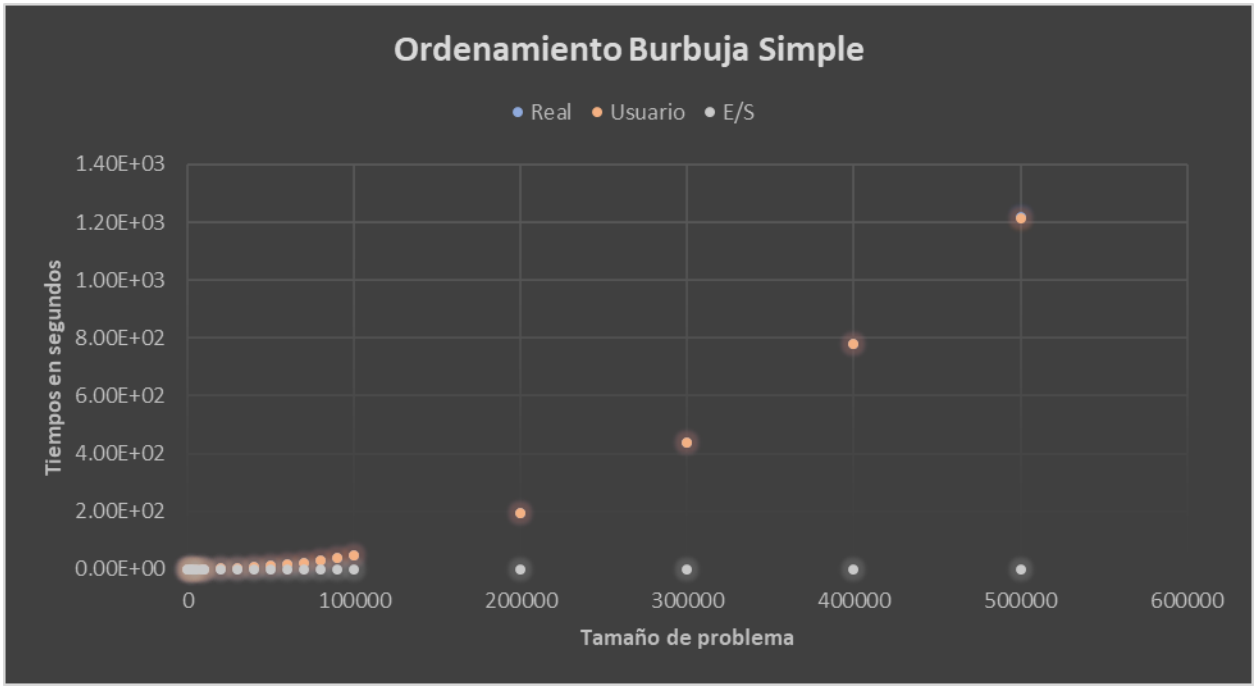

Figura 5. Gráfica comparativa para el algoritmo de burbuja simple

Burbuja Optimizada 1

N	REAL	USUARIO	E/S	CPU/WALL
100	2.48E-05	0.00E+00	3.10E-05	125.02%
1000	1.73E-03	1.74E-03	0.00E+00	100.29%
2000	1.02E-02	8.05E-03	2.02E-03	98.47%
3000	2.88E-02	2.80E-02	0.00E+00	97.27%
5000	6.22E-02	5.95E-02	0.00E+00	95.70%
8000	1.52E-01	1.51E-01	0.00E+00	99.31%
10000	2.44E-01	2.38E-01	1.22E-03	98.06%
20000	1.20E+00	1.19E+00	0.00E+00	99.17%
30000	3.09E+00	3.05E+00	3.32E-03	99.07%
40000	5.27E+00	5.22E+00	1.95E-02	99.40%
50000	8.14E+00	8.12E+00	3.32E-03	99.82%
60000	1.22E+01	1.21E+01	3.23E-03	99.44%
70000	1.72E+01	1.71E+01	1.99E-02	99.67%
80000	2.22E+01	2.22E+01	0.00E+00	99.91%
90000	2.76E+01	2.75E+01	1.31E-02	99.91%

100000	3.47E+01	3.46E+01	6.35E-03	99.90%
200000	1.50E+02	1.49E+02	1.33E-01	99.28%
300000	3.35E+02	3.33E+02	2.56E-01	99.48%
400000	5.72E+02	5.71E+02	9.90E-02	99.88%
500000	8.97E+02	8.96E+02	1.78E-01	99.88%

Figura 6. Tabla comparativa para el algoritmo de burbuja optimización 1

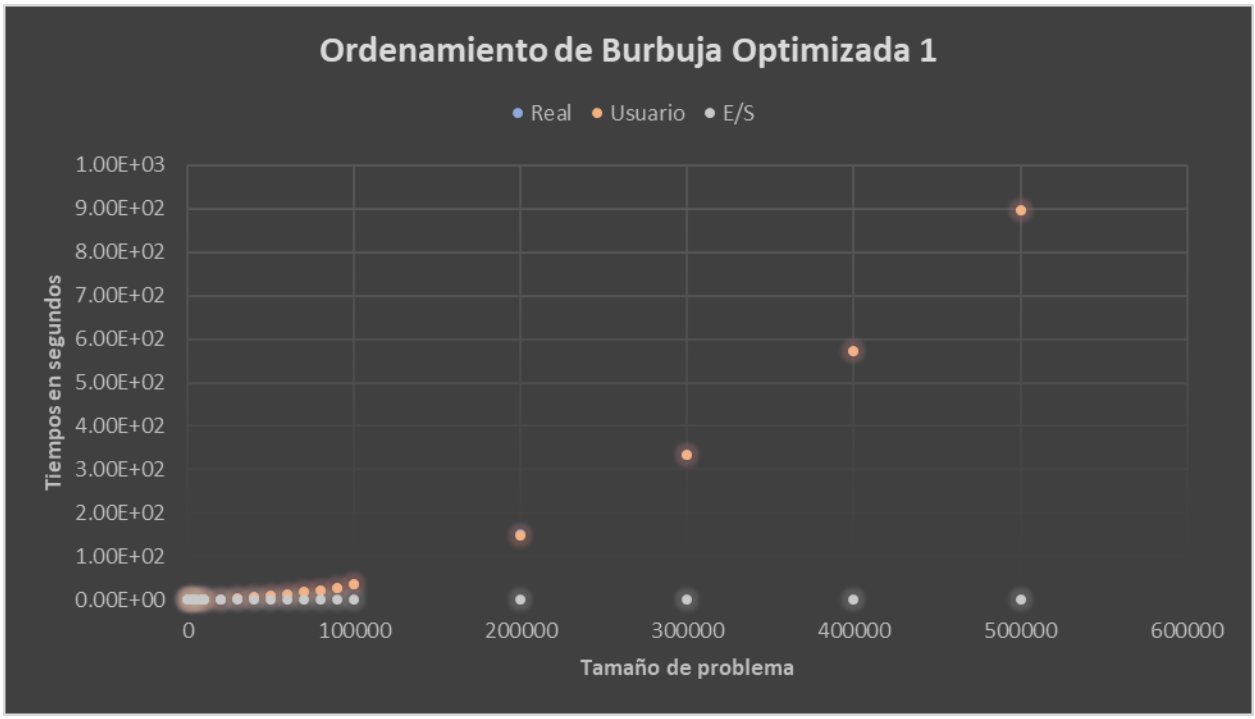

Figura 7. Gráfica comparativa para el algoritmo de burbuja optimización 1

Burbuja Optimizada 2

N	REAL	USUARIO	E/S	CPU/WALL
100	3.10E-05	3.80E-05	0.00E+00	122.60%
1000	2.23E-03	2.23E-03	0.00E+00	99.86%
2000	1.04E-02	1.02E-02	0.00E+00	97.95%
3000	3.51E-02	3.28E-02	1.94E-03	99.10%
5000	6.67E-02	6.47E-02	1.72E-03	99.64%
8000	1.86E-01	1.85E-01	0.00E+00	99.73%
10000	3.05E-01	3.03E-01	1.45E-03	99.83%

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

20000	1.26E+00	1.26E+00	0.00E+00	99.67%
30000	2.57E+00	2.56E+00	1.62E-03	99.88%
40000	5.28E+00	5.26E+00	0.00E+00	99.58%
50000	8.32E+00	8.28E+00	7.50E-03	99.60%
60000	1.24E+01	1.24E+01	3.24E-03	99.85%
70000	1.66E+01	1.66E+01	3.27E-03	99.90%
80000	2.19E+01	2.18E+01	1.36E-02	99.87%
90000	2.78E+01	2.77E+01	6.83E-03	99.92%
100000	3.47E+01	3.47E+01	0.00E+00	99.90%
200000	1.41E+02	1.40E+02	9.86E-03	99.86%
300000	3.16E+02	3.16E+02	3.28E-03	99.92%
400000	5.63E+02	5.62E+02	6.63E-03	99.92%
500000	8.81E+02	8.80E+02	3.27E-03	99.92%

Figura 8. Tabla comparativa para el algoritmo de burbuja optimización 2

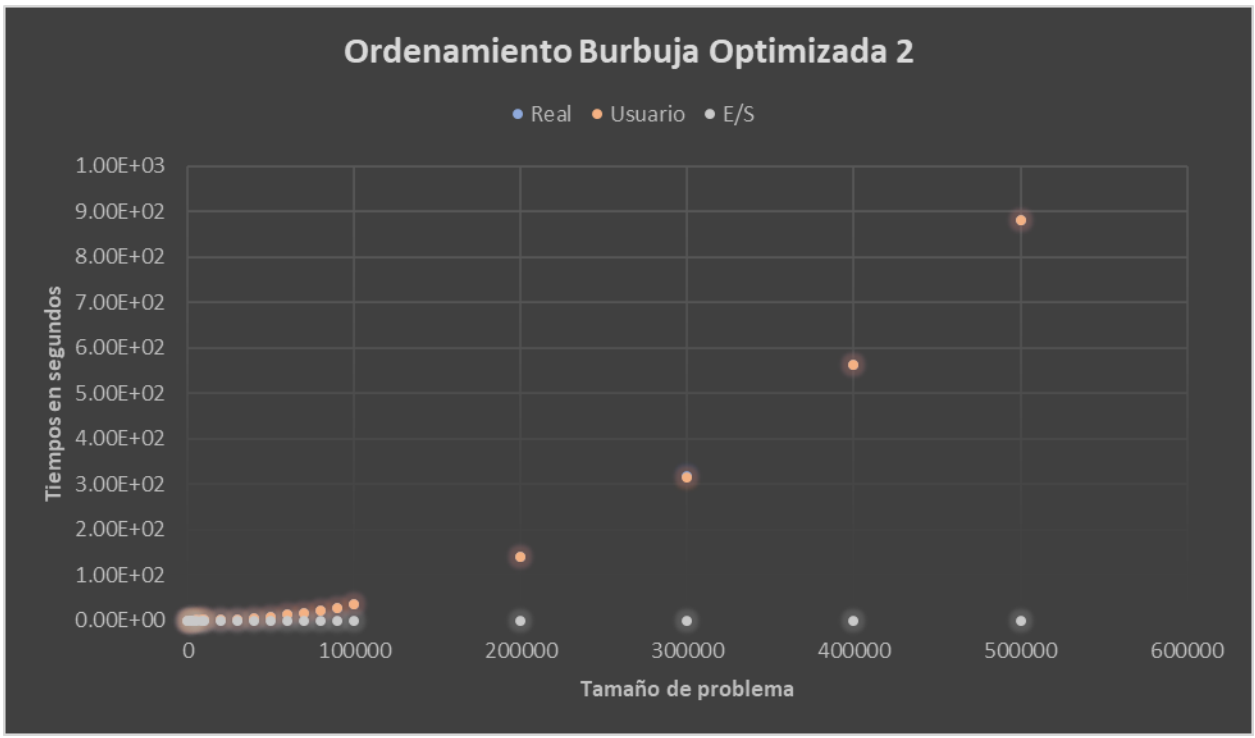

Figura 9. Gráfica comparativa para el algoritmo de burbuja optimización 2

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Inserción

N	REAL	USER	SYS(E/S)	CPU/WALL
100	9.06E-06	2.40E-05	0.00E+00	264.90%
1000	7.03E-04	0.00E+00	7.18E-04	102.12%
2000	3.25E-03	3.24E-03	0.00E+00	99.45%
3000	6.66E-03	6.33E-03	0.00E+00	95.06%
5000	2.79E-02	2.57E-02	0.00E+00	92.11%
8000	4.58E-02	4.55E-02	0.00E+00	99.37%
10000	6.44E-02	6.42E-02	0.00E+00	99.70%
20000	2.52E-01	2.52E-01	0.00E+00	99.88%
30000	5.72E-01	5.71E-01	0.00E+00	99.75%
40000	1.02E+00	1.02E+00	0.00E+00	99.75%
50000	1.66E+00	1.66E+00	0.00E+00	99.92%
60000	2.69E+00	2.69E+00	0.00E+00	99.86%
70000	3.30E+00	3.30E+00	0.00E+00	99.91%
80000	4.58E+00	4.58E+00	0.00E+00	99.90%
90000	5.86E+00	5.85E+00	0.00E+00	99.92%
100000	7.06E+00	7.05E+00	0.00E+00	99.91%
200000	2.91E+01	2.90E+01	3.23E-03	99.92%
300000	6.55E+01	6.54E+01	0.00E+00	99.92%
400000	1.19E+02	1.18E+02	3.56E-02	99.79%
500000	1.86E+02	1.86E+02	6.74E-02	99.85%

Figura 10 Tabla comparativa para el algoritmo de inserción

Figura 11 Gráfica comparativa para el algoritmo de inserción

Selección

N	REAL	USER	SYS(E/S)	CPU/WALL
100	1.69E-05	2.60E-05	0.00E+00	153.59%
1000	1.12E-03	1.12E-03	0.00E+00	100.54%
2000	5.58E-03	3.36E-03	2.21E-03	99.66%
3000	1.13E-02	8.72E-03	2.05E-03	95.62%
5000	2.82E-02	2.79E-02	0.00E+00	99.21%
8000	6.84E-02	6.79E-02	0.00E+00	99.34%
10000	1.03E-01	1.03E-01	0.00E+00	99.73%
20000	4.04E-01	4.02E-01	9.03E-04	99.73%
30000	9.24E-01	9.21E-01	0.00E+00	99.64%
40000	1.61E+00	1.61E+00	9.72E-04	99.92%
50000	2.50E+00	2.49E+00	0.00E+00	99.92%
60000	4.03E+00	4.03E+00	0.00E+00	99.92%
70000	5.38E+00	5.37E+00	0.00E+00	99.93%

80000	6.96E+00	6.95E+00	0.00E+00	99.93%
90000	8.86E+00	8.85E+00	0.00E+00	99.94%
100000	1.09E+01	1.09E+01	0.00E+00	99.93%
200000	4.46E+01	4.46E+01	0.00E+00	99.93%
300000	1.02E+02	1.02E+02	3.44E-02	99.83%
400000	1.83E+02	1.83E+02	2.57E-02	99.76%
500000	2.91E+02	2.90E+02	1.79E-01	99.67%

Figura 12 Tabla comparativa para el algoritmo de selección

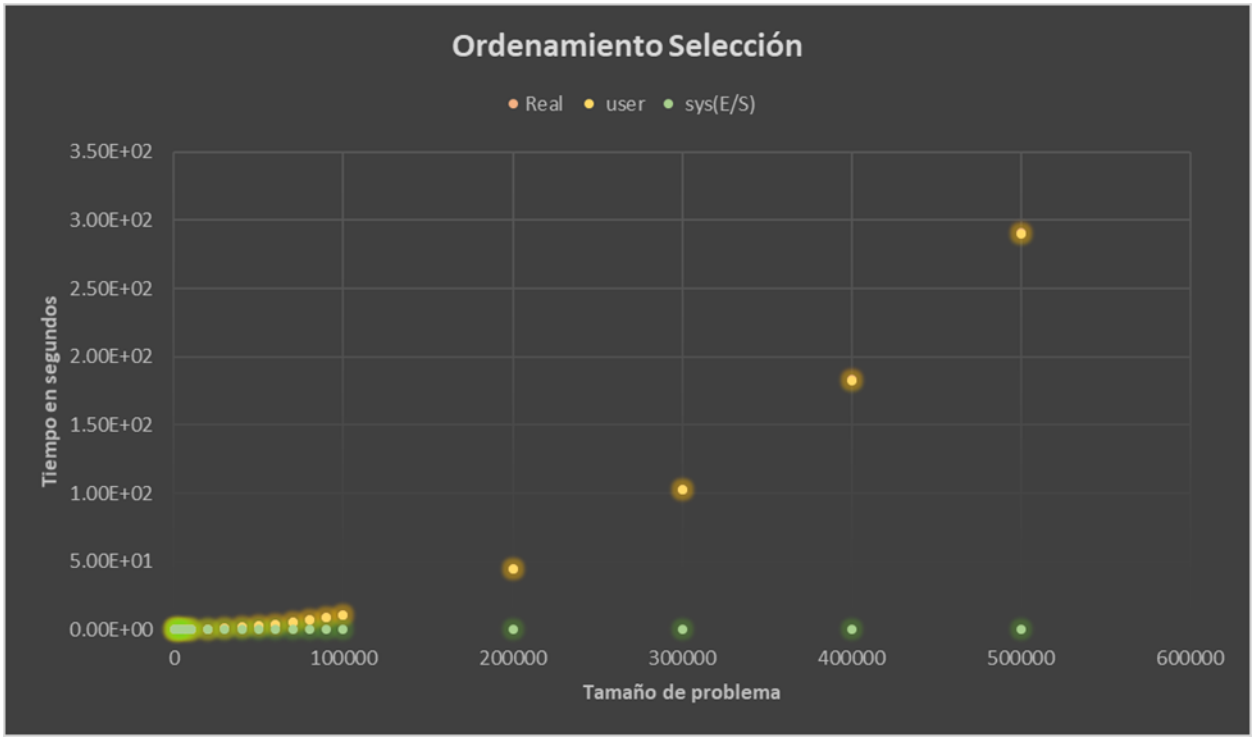

Figura 13 Gráfica comparativa para el algoritmo de selección

Shell

N	REAL	USUARIO	E/S	CPU/WALL
100	1.62E-05	4.20E-05	0.00E+00	259.0599529
1000	2.95E-04	0.00E+00	3.12E-04	105.7045919
5000	2.07E-03	1.99E-03	0.00E+00	96.13758305
10000	5.19E-03	4.73E-03	0.00E+00	91.01229663
50000	3.64E-02	3.60E-02	0.00E+00	98.7067812

100000	9.78E-02	9.73E-02	0.00E+00	99.51200667
200000	2.63E-01	2.62E-01	0.00E+00	99.8869671
400000	6.45E-01	6.44E-01	0.00E+00	99.85866152
600000	9.20E-01	9.19E-01	1.76E-04	99.88054979
800000	1.18E+00	1.18E+00	0.00E+00	99.84692051
1000000	1.36E+00	1.36E+00	1.10E-05	99.83409673
2000000	4.20E+00	4.20E+00	1.60E-05	99.9029125
3000000	6.00E+00	5.99E+00	0.00E+00	99.90794223
4000000	9.68E+00	9.66E+00	3.43E-03	99.85998148
5000000	1.30E+01	1.30E+01	3.38E-03	99.7456689
6000000	1.59E+01	1.58E+01	2.93E-02	99.3216543
7000000	1.87E+01	1.87E+01	6.47E-03	99.85619749
8000000	2.52E+01	2.51E+01	2.20E-05	99.89024154
9000000	2.62E+01	2.61E+01	3.00E-06	99.90831452
10000000	3.27E+01	3.24E+01	2.92E-02	99.38243883

Figura 14 Tabla comparativa para el algoritmo Shell

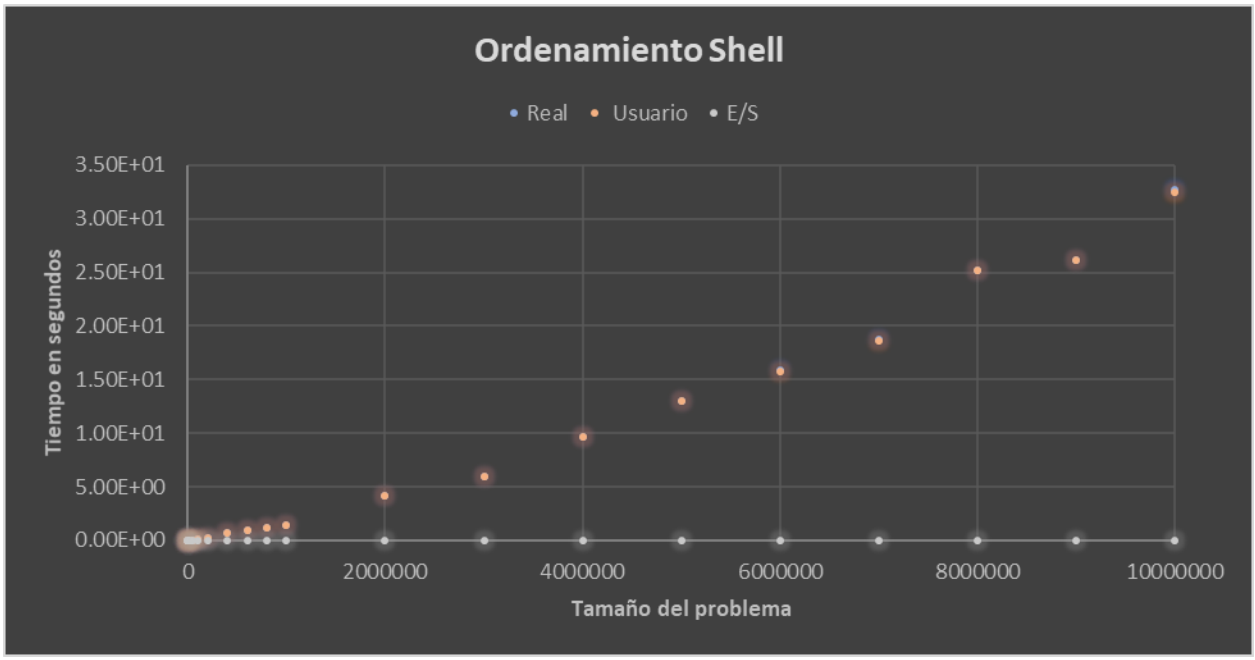

Figura 15. Gráfica comparativa del algoritmo Shell

Ordenamiento basado en ABB

N	REAL	USUARIO	E/S	CPU/WALL
100	1.29E-05	2.00E-05	0.00E+00	155.3445926
1000	1.56E-04	1.64E-04	0.00E+00	105.1782654
5000	1.42E-03	1.42E-03	0.00E+00	99.84763923
10000	3.01E-03	3.02E-03	0.00E+00	100.1629788
50000	2.18E-02	2.14E-02	0.00E+00	98.23392871
100000	5.73E-02	5.23E-02	0.00E+00	91.32620783
200000	1.05E-01	1.01E-01	3.02E-03	99.46130335
400000	2.32E-01	2.31E-01	0.00E+00	99.83756167
600000	4.15E-01	4.14E-01	1.40E-05	99.72415333
800000	5.93E-01	5.80E-01	1.00E-02	99.48978325
1000000	7.68E-01	7.46E-01	2.00E-02	99.75637159
2000000	1.90E+00	1.87E+00	2.32E-02	99.74379265
3000000	3.26E+00	3.20E+00	4.97E-02	99.84990975
4000000	4.65E+00	4.57E+00	6.95E-02	99.69111119
5000000	6.00E+00	5.92E+00	7.31E-02	99.78904562
6000000	7.70E+00	7.59E+00	8.57E-02	99.67113669
7000000	9.14E+00	9.01E+00	1.06E-01	99.74149593
8000000	1.08E+01	1.06E+01	9.94E-02	99.75450985
9000000	1.26E+01	1.24E+01	1.32E-01	99.71240619
10000000	1.48E+01	1.46E+01	1.76E-01	99.74650696

Figura 16. Tabla comparativa para el algoritmo de ABB

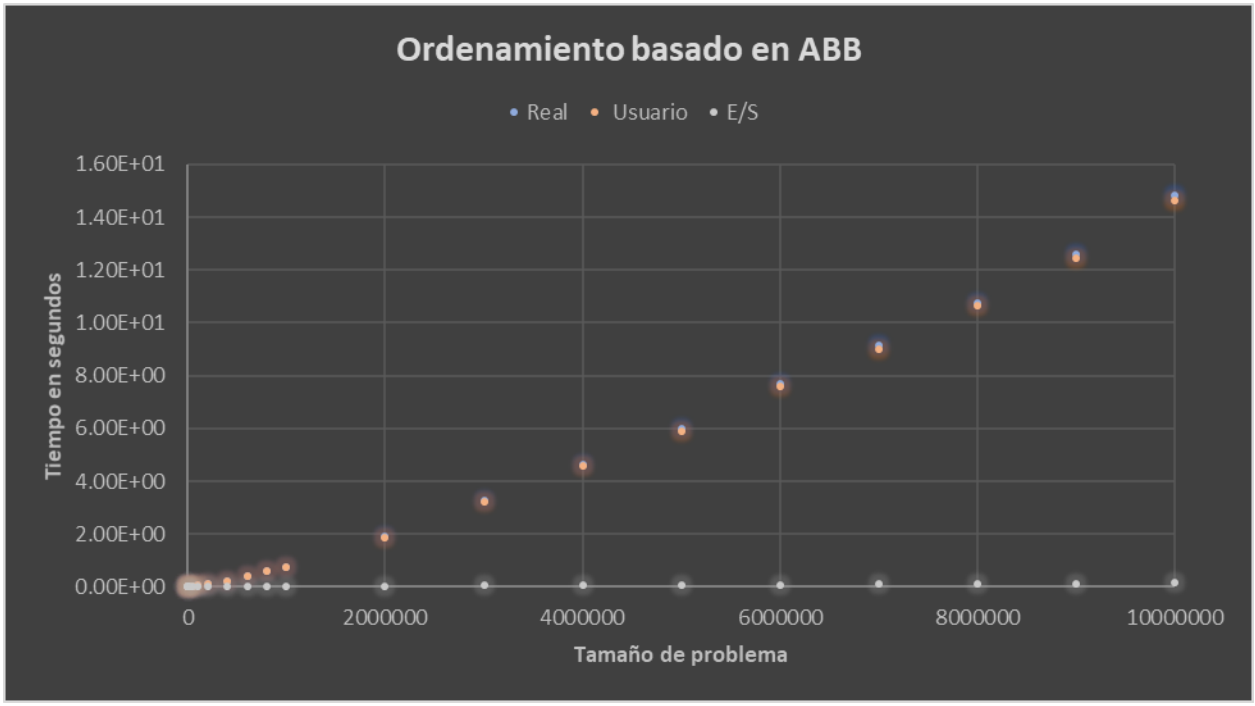

Figura 17. Gráfica comparativa del algoritmo ABB

MergeSort

N	REAL	USUARIO	E/S	CPU/WALL
100	1.62E-05	0.00E+00	3.30E-05	203.5471059
1000	1.39E-04	1.52E-04	0.00E+00	109.3540666
5000	7.85E-04	8.01E-04	0.00E+00	102.0236108
10000	1.63E-03	1.64E-03	0.00E+00	100.5479634
50000	1.03E-02	1.03E-02	0.00E+00	99.34285263
100000	2.07E-02	2.06E-02	0.00E+00	99.65744034
200000	4.30E-02	4.27E-02	1.99E-04	99.76749297
400000	8.86E-02	8.84E-02	0.00E+00	99.78554702
600000	1.36E-01	1.36E-01	0.00E+00	99.87633312
800000	1.89E-01	1.88E-01	0.00E+00	99.63587628
1000000	2.33E-01	2.29E-01	3.33E-03	99.92659895
2000000	4.95E-01	4.87E-01	6.69E-03	99.86621953
3000000	7.48E-01	7.34E-01	1.34E-02	99.91807054
4000000	1.02E+00	1.01E+00	9.89E-03	99.92092889
5000000	1.63E+00	1.61E+00	1.66E-02	99.83516561

6000000	1.57E+00	1.54E+00	2.00E-02	99.91872939
7000000	2.33E+00	2.29E+00	3.66E-02	99.91706832
8000000	2.13E+00	2.10E+00	2.00E-02	99.86030685
9000000	2.70E+00	2.66E+00	3.00E-02	99.85054363
10000000	3.38E+00	3.35E+00	2.33E-02	99.71510787

Figura 18. Tabla comparativa para el algoritmo Mergesort

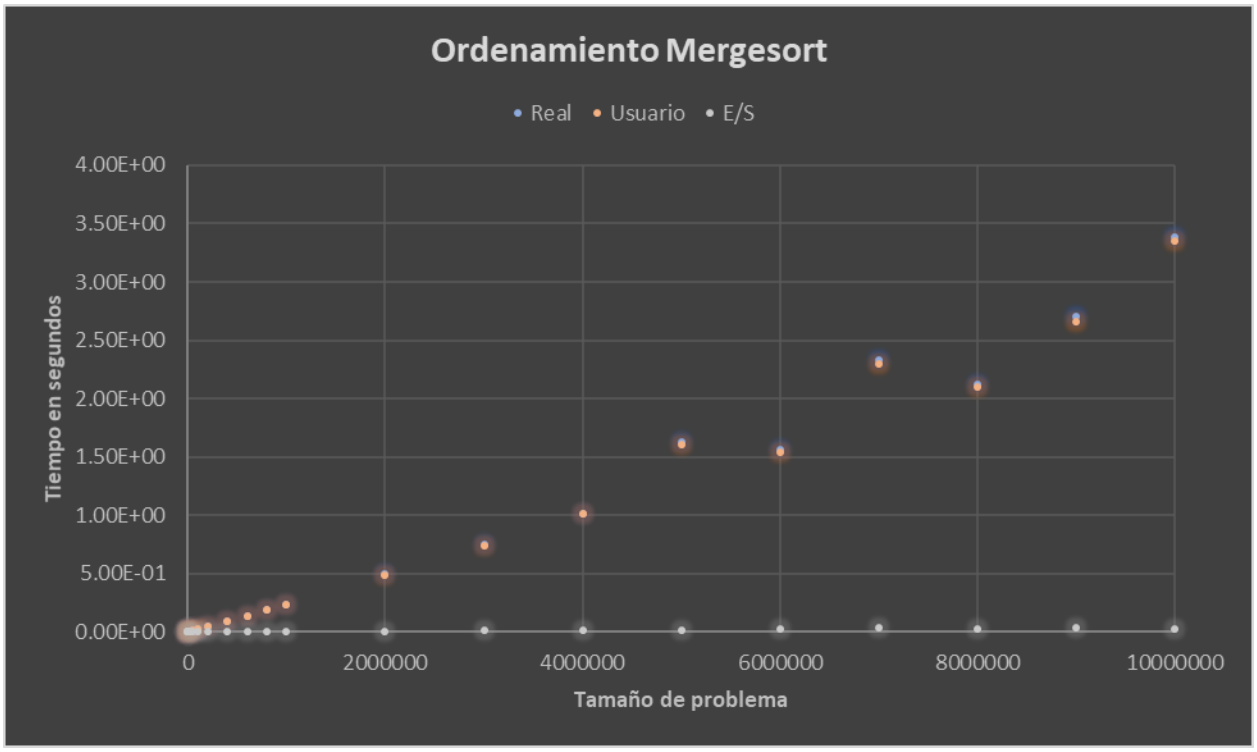

Figura 19. Gráfica comparativa del algoritmo Mergesort

QuickSort

N	REAL	USUARIO	E/S	CPU/WALL
100	1.10E-05	3.00E-05	0.00E+00	273.5415652
1000	9.49E-05	1.07E-04	0.00E+00	112.7614392
5000	5.39E-04	0.00E+00	5.47E-04	101.4721047
10000	1.15E-03	1.16E-03	0.00E+00	100.5282193
50000	7.77E-03	7.08E-03	0.00E+00	91.15926102
100000	1.57E-02	1.57E-02	0.00E+00	99.58692106
200000	3.57E-02	3.42E-02	0.00E+00	95.88117411

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

400000	6.29E-02	6.27E-02	0.00E+00	99.70744818
600000	1.07E-01	1.06E-01	2.17E-04	99.89767817
800000	1.73E-01	1.73E-01	0.00E+00	99.87277937
1000000	2.18E-01	2.17E-01	2.10E-05	99.50191122
2000000	4.59E-01	4.53E-01	0.00E+00	98.71640149
3000000	5.85E-01	5.81E-01	3.28E-03	99.77721882
4000000	7.24E-01	7.23E-01	6.90E-05	99.90828166
5000000	1.15E+00	1.15E+00	1.50E-05	99.89110622
6000000	1.42E+00	1.42E+00	5.10E-05	99.90473136
7000000	1.30E+00	1.30E+00	0.00E+00	99.92772692
8000000	1.94E+00	1.93E+00	6.60E-03	99.84896966
9000000	1.80E+00	1.80E+00	4.60E-05	99.89660978
10000000	2.23E+00	2.22E+00	6.36E-03	99.73023373

Figura 20 Tabla comparativa del algoritmo Quicksort

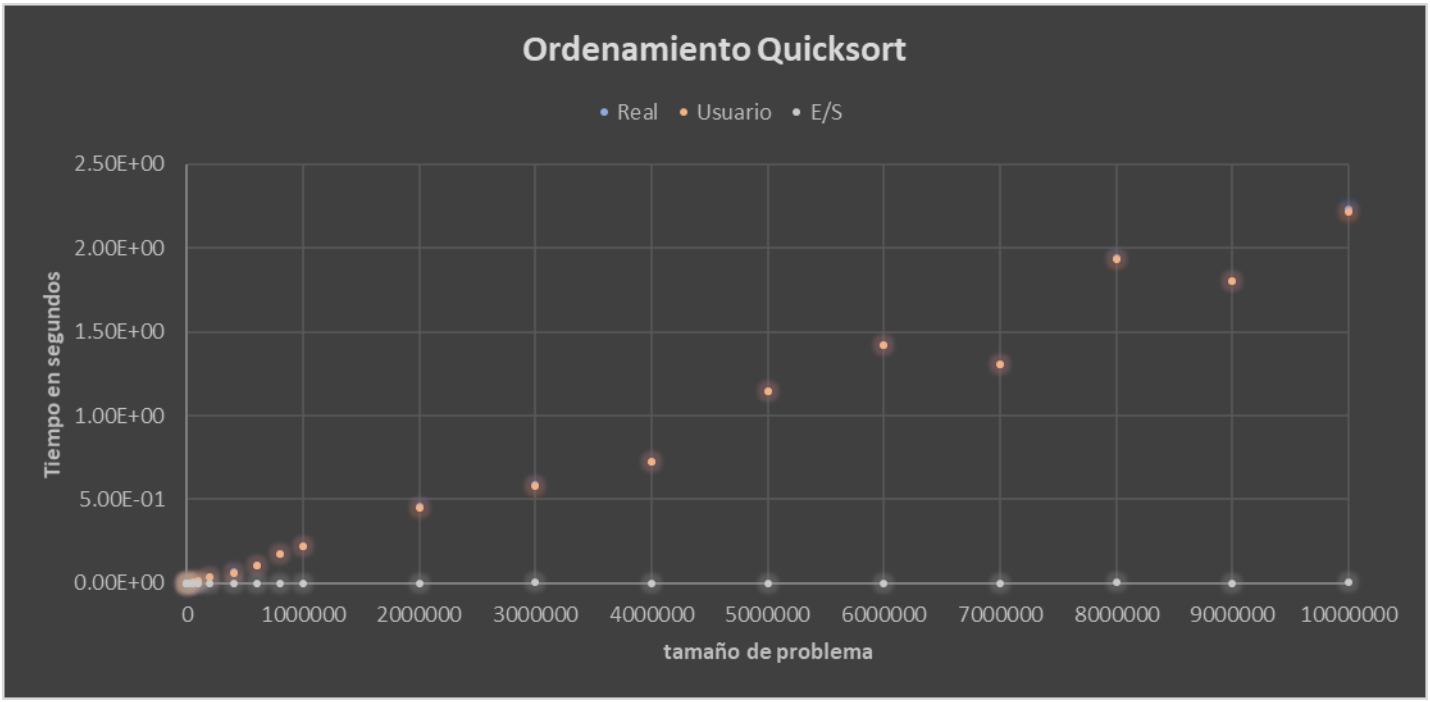

Figura 21 Gráfica comparativa del algoritmo Quicksort

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Comparativa (Tiempo Real)

Debido a que los algoritmos de burbuja y de inserción son de complejidad cuadrática, tardarían mucho tiempo en terminar en ejecutarse en ciertos números n grandes, por ello, se dividieron en dos comparativas, por un lado, tenemos los algoritmos de Mergesort, Árbol binario de búsqueda, Shell y quicksort; por otro lado, tenemos los algoritmos de inserción y las tres versiones de burbuja.

N	MERGE	ABB	SHELL	QUICK
100	1.62E-05	1.29E-05	1.62E-05	1.10E-05
1000	1.39E-04	1.56E-04	2.95E-04	9.49E-05
5000	7.85E-04	1.42E-03	2.07E-03	5.39E-04
10000	1.63E-03	3.01E-03	5.19E-03	1.15E-03
50000	1.03E-02	2.18E-02	3.64E-02	7.77E-03
100000	2.07E-02	5.73E-02	9.78E-02	1.57E-02
200000	4.30E-02	1.05E-01	2.63E-01	3.57E-02
400000	8.86E-02	2.32E-01	6.45E-01	6.29E-02
600000	1.36E-01	4.15E-01	9.20E-01	1.07E-01
800000	1.89E-01	5.93E-01	1.18E+00	1.73E-01
1000000	2.33E-01	7.68E-01	1.36E+00	2.18E-01
2000000	4.95E-01	1.90E+00	4.20E+00	4.59E-01
3000000	7.48E-01	3.26E+00	6.00E+00	5.85E-01
4000000	1.02E+00	4.65E+00	9.68E+00	7.24E-01
5000000	1.63E+00	6.00E+00	1.30E+01	1.15E+00
6000000	1.57E+00	7.70E+00	1.59E+01	1.42E+00
7000000	2.33E+00	9.14E+00	1.87E+01	1.30E+00
8000000	2.13E+00	1.08E+01	2.52E+01	1.94E+00
9000000	2.70E+00	1.26E+01	2.62E+01	1.80E+00
10000000	3.38E+00	1.48E+01	3.27E+01	2.23E+00

Figura 22 Primera tabla comparativa de tiempo real para merge, ABB, shell y quicsort

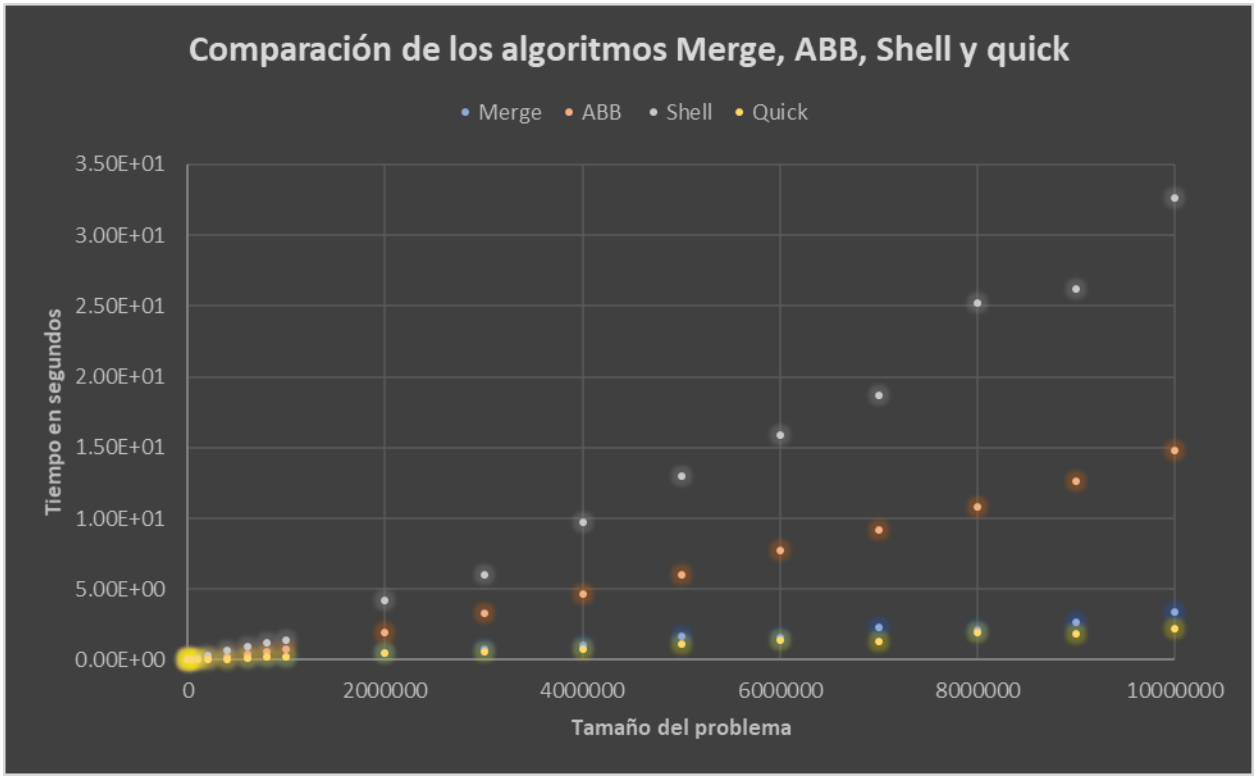

Figura 23 Gráfica comparativa de tiempo real de los algoritmos merge, abb, shell y quicksort

N	INSERCIÓN	BURBUJA SIMPLE	BURBUJA OPTIMIZADA	BURBUJA OPTIMIZADA 2
100	9.06E-06	4.79E-05	2.48E-05	3.10E-05
1000	7.03E-04	4.43E-03	1.73E-03	2.23E-03
2000	3.25E-03	1.53E-02	1.02E-02	1.04E-02
3000	6.66E-03	3.85E-02	2.88E-02	3.51E-02
5000	2.79E-02	1.02E-01	6.22E-02	6.67E-02
8000	4.58E-02	2.72E-01	1.52E-01	1.86E-01
10000	6.44E-02	4.41E-01	2.44E-01	3.05E-01
20000	2.52E-01	1.93E+00	1.20E+00	1.26E+00
30000	5.72E-01	3.74E+00	3.09E+00	2.57E+00
40000	1.02E+00	7.34E+00	5.27E+00	5.28E+00
50000	1.66E+00	1.14E+01	8.14E+00	8.32E+00
60000	2.69E+00	1.76E+01	1.22E+01	1.24E+01
70000	3.30E+00	2.30E+01	1.72E+01	1.66E+01
80000	4.58E+00	3.09E+01	2.22E+01	2.19E+01

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

90000	5.86E+00	3.90E+01	2.76E+01	2.78E+01
100000	7.06E+00	4.81E+01	3.47E+01	3.47E+01
200000	2.91E+01	1.94E+02	1.50E+02	1.41E+02
300000	6.55E+01	4.38E+02	3.35E+02	3.16E+02
400000	1.19E+02	7.79E+02	5.72E+02	5.63E+02
500000	1.86E+02	1.22E+03	8.97E+02	8.81E+02

Figura 24 Segunda tabla comparativa de tiempo real para los algoritmos de inserción, burbuja simple, optimizada 1 y optimizada 2

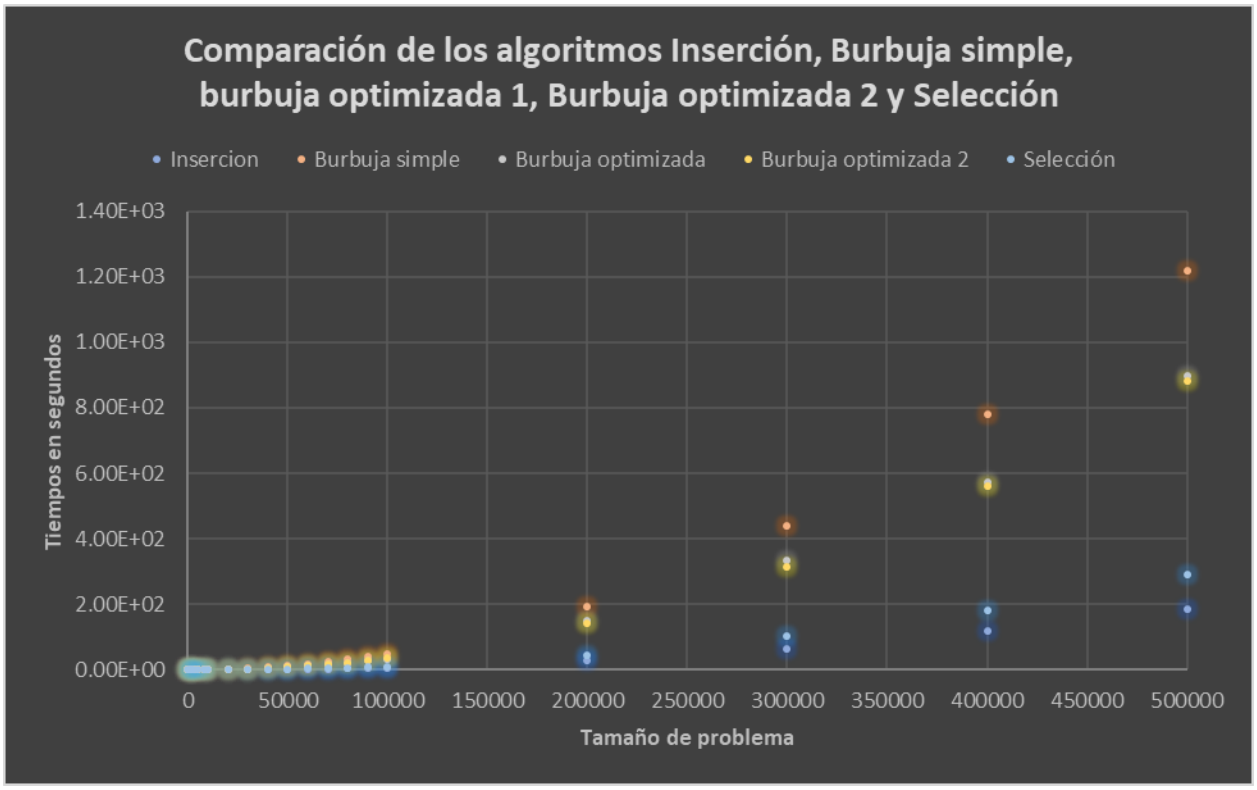

Figura 25 Gráfica comparativa de tiempo real de los algoritmos inserción, burbuja simple, optimizada 1 y optimizada 2

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Aproximación a la función del comportamiento temporal (tiempo real)

En base a los resultados obtenidos mediante las simulaciones obtenidas anteriormente, se pudo realizar una aproximación a la función del comportamiento temporal en tiempo real gracias al software MATLAB, por ello, se presentan las funciones y la gráfica correspondiente a cada uno de los algoritmos.

Burbuja Simple

Grado 1

$$y = 0.002155x - 71.44$$

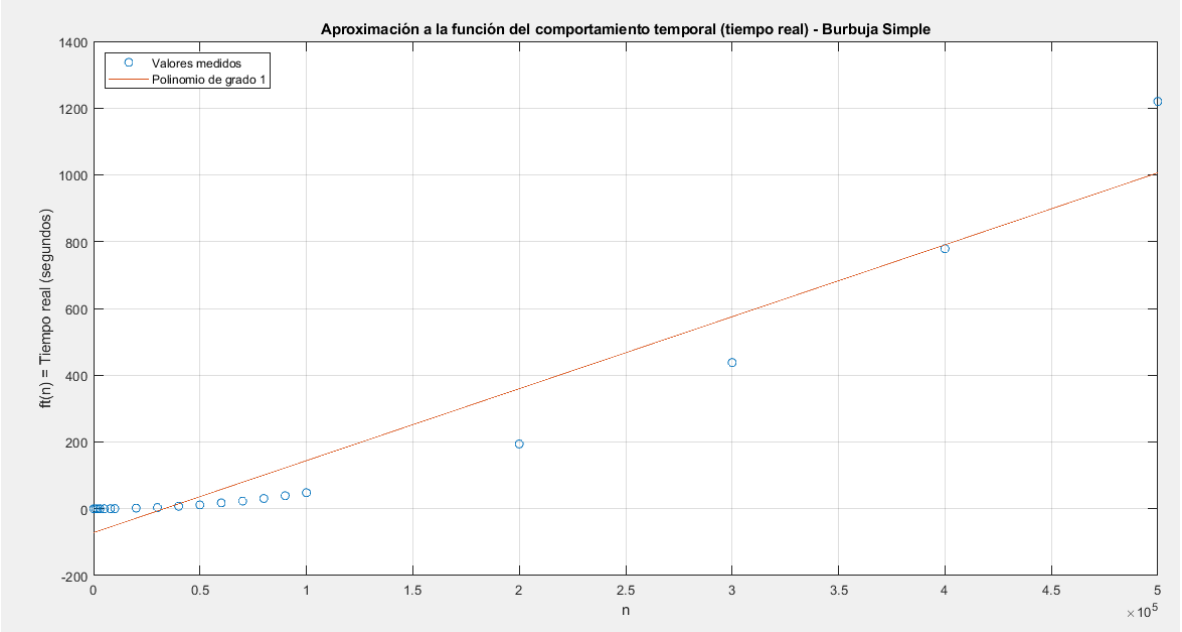

Figura 26 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo burbuja simple

Grado 2

$$y = 4.899\text{x}10^{-9}x^2 - 1.041\text{x}10^{-5}x + 0.03615$$

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

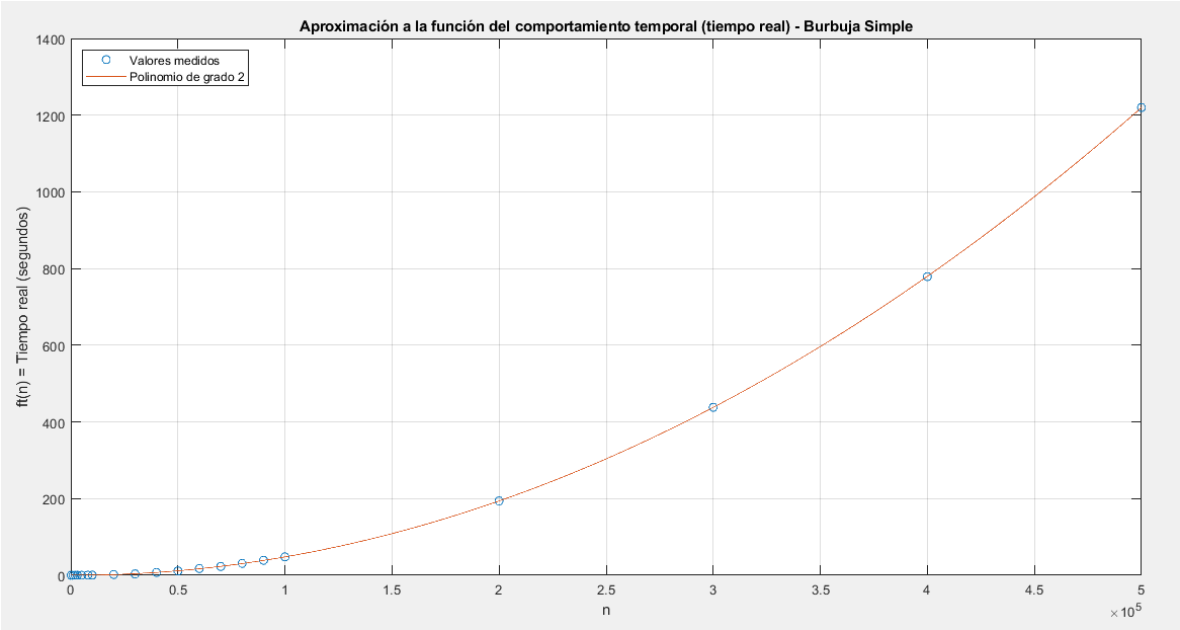

Figura 27 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo burbuja simple

Grado 3

$$y = 4.818\text{x}10^{-17}x^3 + 4.866\text{x}10^{-9}x^2 - 5.224\text{x}10^{-6}x - 0.06155$$

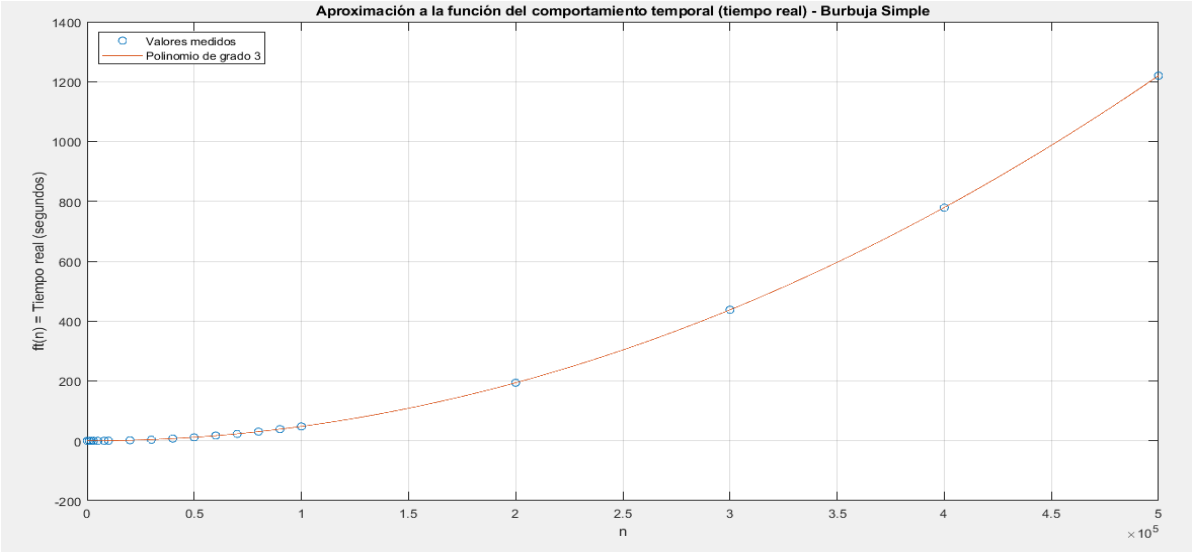

Figura 28 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo burbuja simple

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 6

$$y = 2.997\text{x}10^{-32}x^6 - 4.017\text{x}10^{-26}x^5 + 2.002\text{x}10^{-20}x^4 - 4.589\text{x}10^{-15}x^3 + 5.371\text{x}10^{-9}x^2 - 2.646\text{x}10^{-5}x + 0.07757$$

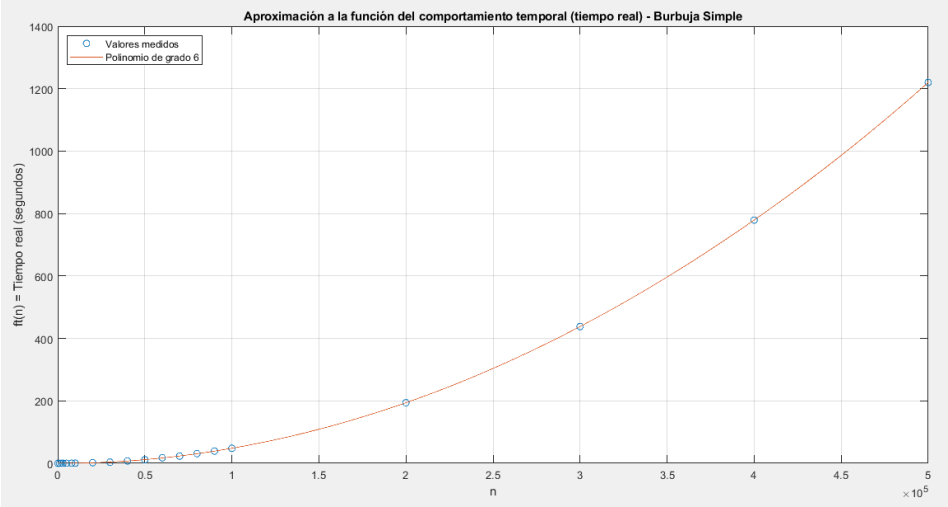

Figura 29 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo burbuja simple

Burbuja Optimizada 1

Grado 1

$$y = 0.001593x - 52.52$$

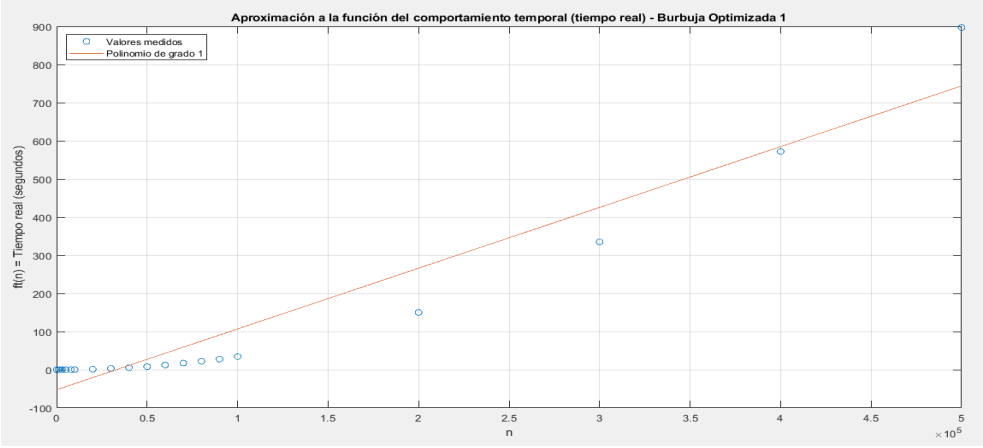

Figura 30 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo burbuja optimizada 1

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 3.509\text{x}10^{-9}x^2 + 4.161\text{x}10^{-5}x - 1.32$$

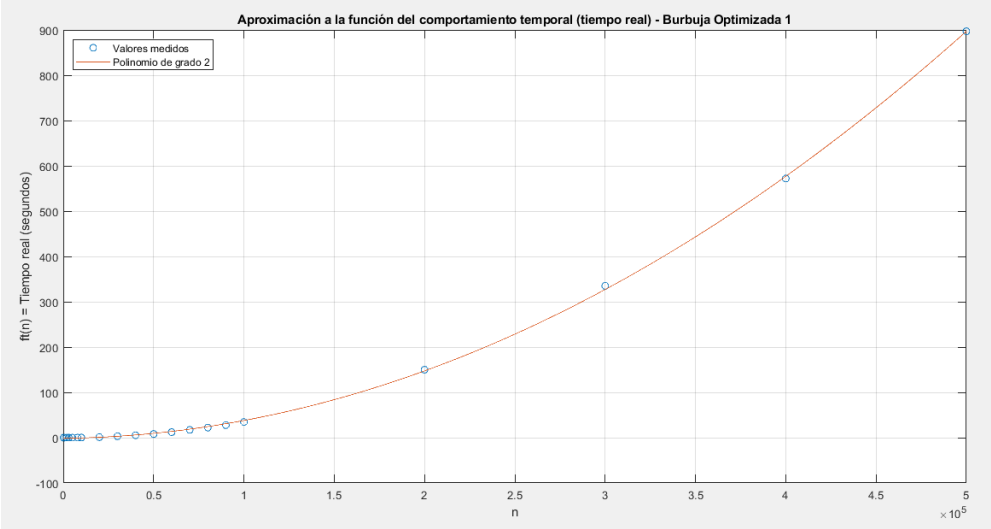

Figura 31 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo burbuja optimizada 1

Grado 3

$$y = -4.708\text{x}10^{-16}x^3 + 3.834\text{x}10^{-9}x^2 - 9.093\text{x}10^{-6}x - 0.3651$$

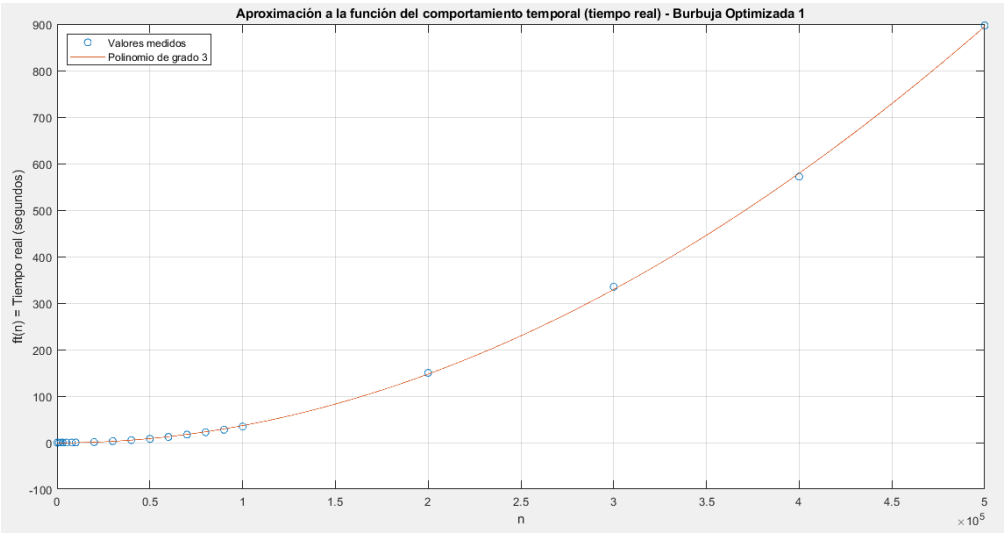

Figura 32 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo burbuja optimizada

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 6

$$y = 4.03\text{x}10^{-32}x^6 - 7.696\text{x}10^{-27}x^5 - 2.213\text{x}10^{-20}x^4 + 9.965\text{x}10^{-15}x^3 + 2.542\text{x}10^{-9}x^2 + 1.934\text{x}10^{-5}x - 0.09011$$

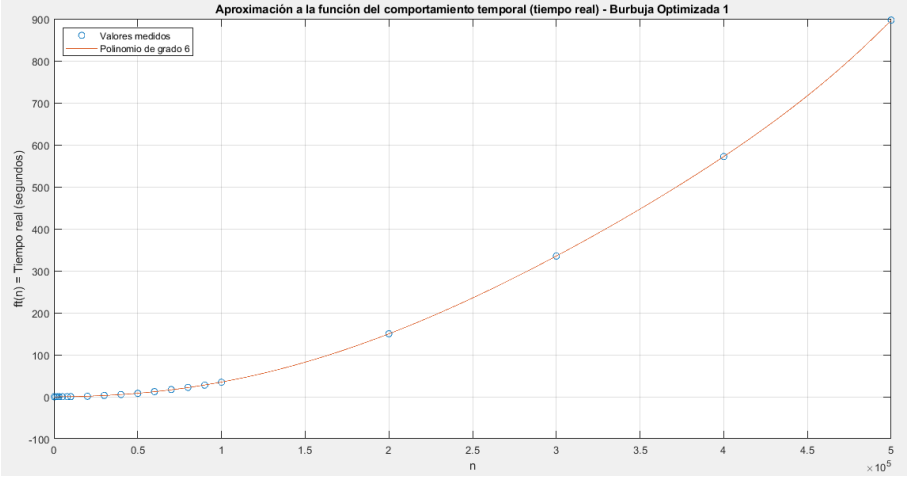

Figura 33 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo burbuja optimizada 1

Burbuja Optimizada 2

Grado 1

$$y = 0.001557x - 51.65$$

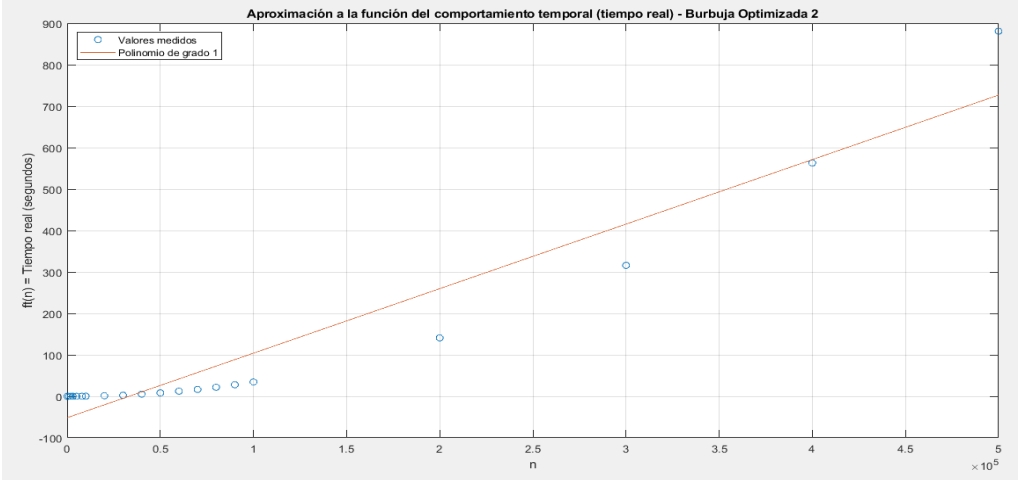

Figura 34 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo burbuja optimizada 2

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 3.536\text{x}10^{-9}x^2 - 6.266\text{x}10^{-6}x - 0.06548$$

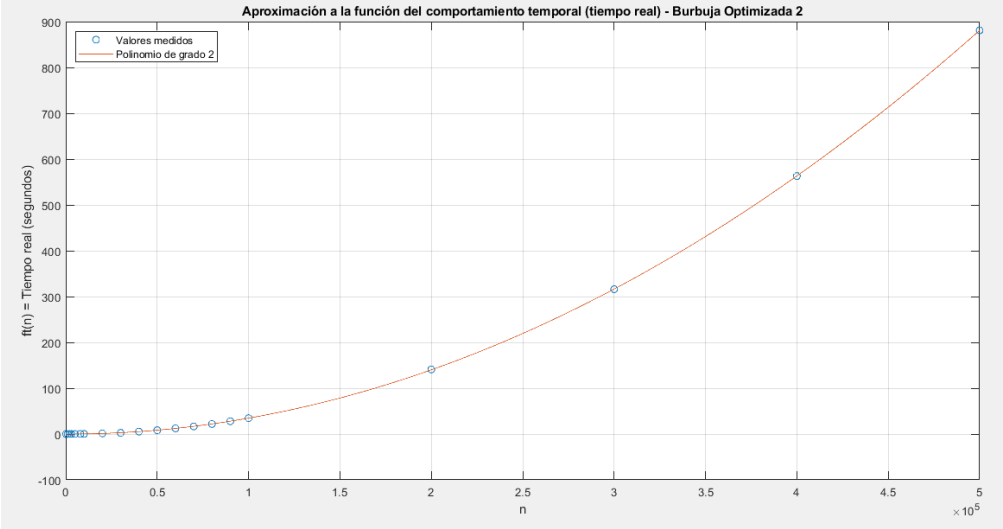

Figura 35 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo burbuja optimizada 2

Grado 3

$$y = 4.67\text{x}10^{-18}x^3 + 3.533\text{x}10^{-9}x^2 - 5.763\text{x}10^{-6}x - 0.07494$$

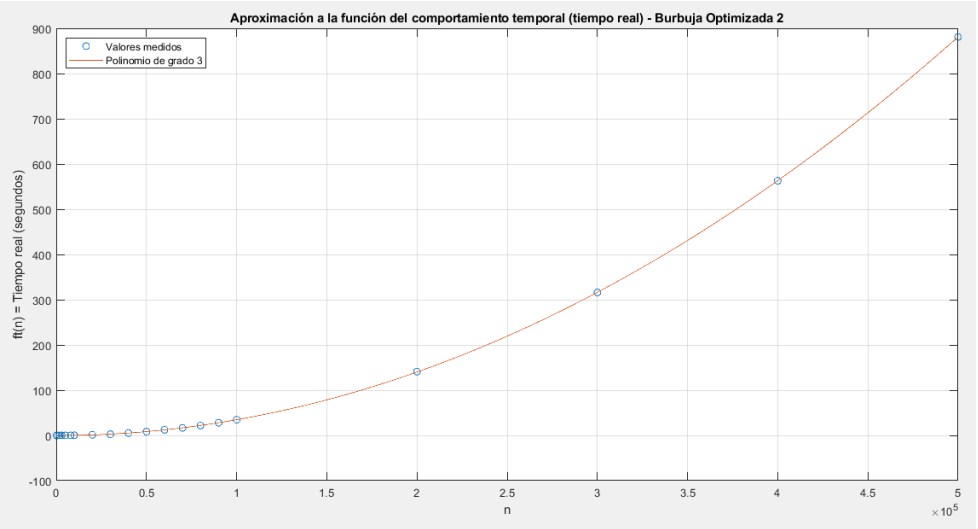

Figura 36 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo burbuja optimizada 2

Grado 6

$$y = -4.235\text{x}10^{-32}x^6 + 5.486\text{x}10^{-26}x^5 - 2.475\text{x}10^{-20}x^4 + 4.459\text{x}10^{-15}x^3 + 3.286\text{x}10^{-9}x^2 - 6.872\text{x}10^{-6}x + 0.01026$$

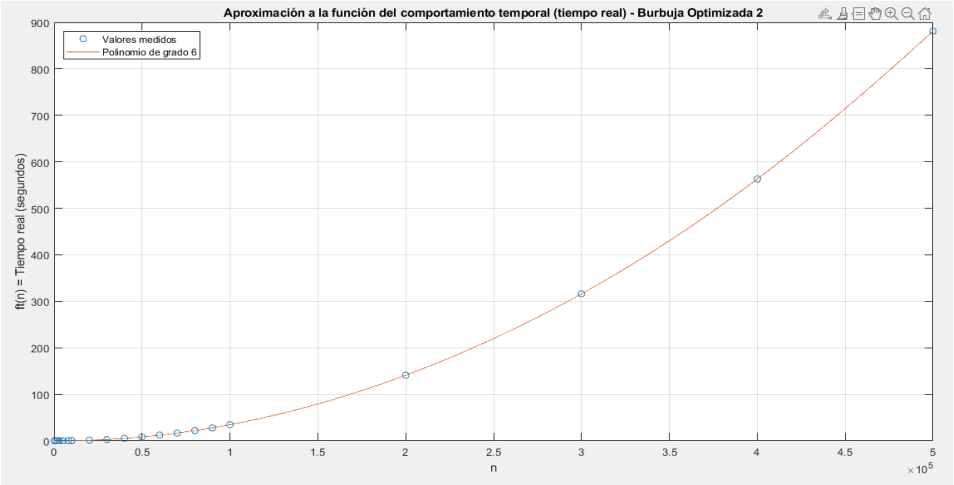

Figura 37 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo burbuja optimizada 2

Inserción

Grado 1

$$y = 0.000328x - 10.96$$

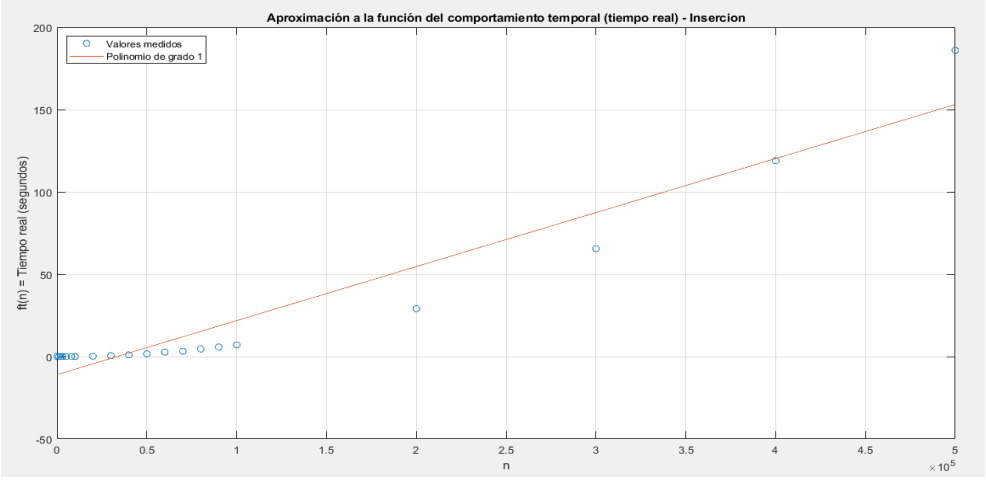

Figura 38 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo inserción

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 7.572\text{x}10^{-10}x^2 - 6.671\text{x}10^{-6}x + 0.08937$$

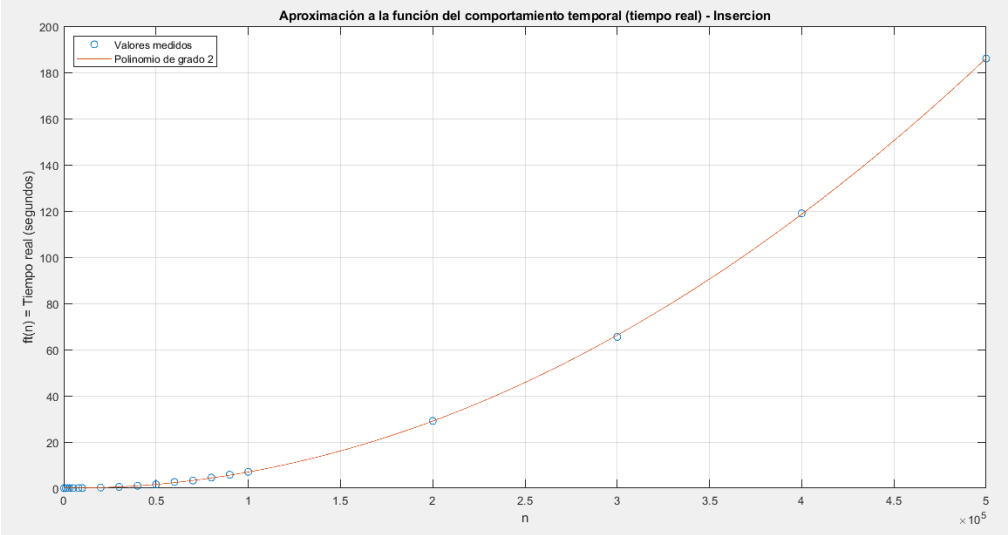

Figura 39 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo inserción

Grado 3

$$y = 2.824\text{x}10^{-17}x^3 + 7.377\text{x}10^{-10}x^2 - 3.63\text{x}10^{-6}x + 0.03212$$

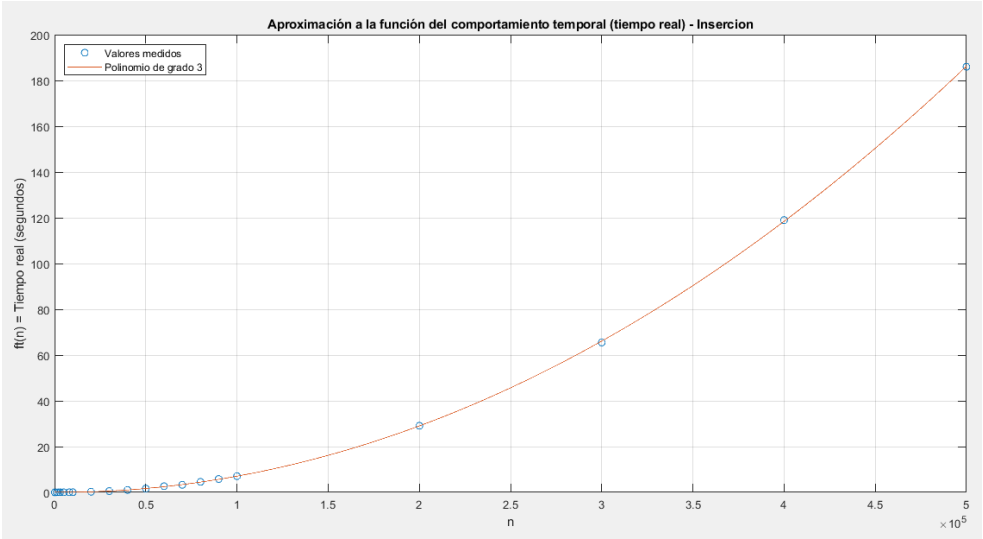

Figura 40 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo inserción

Grado 6

$$y = -2.272\text{x}10^{-32}x^6 + 2.659\text{x}10^{-26}x^5 - 1.064\text{x}10^{-20}x^4 + 1.739\text{x}10^{-15}x^3 + 6.321\text{x}10^{-10}x^2 - 7.11\text{x}10^{-7}x + 0.002911$$

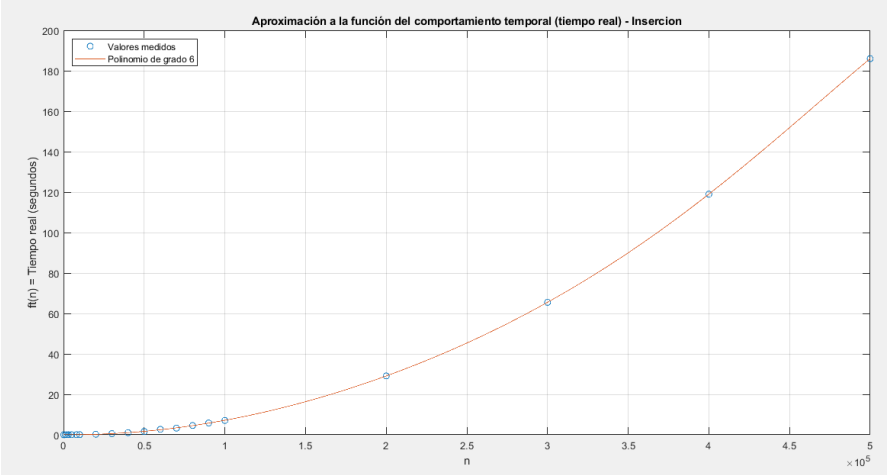

Figura 41 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo inserción

Selección

Grado 1

$$y = 0.0005102x - 17.11$$

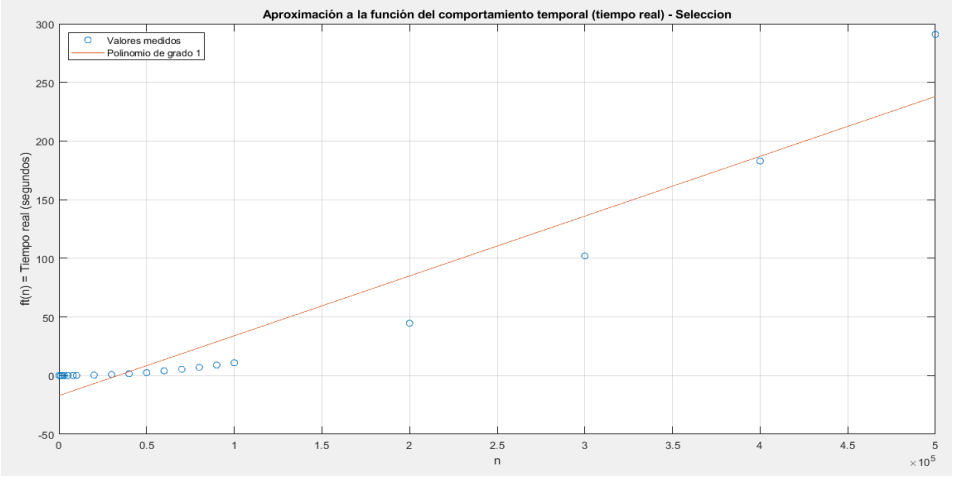

Figura 42 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo selección

Grado 2

$$y = 1.195\text{x}10^{-9}x^2 - 1.787\text{x}10^{-5}x + 0.3159$$

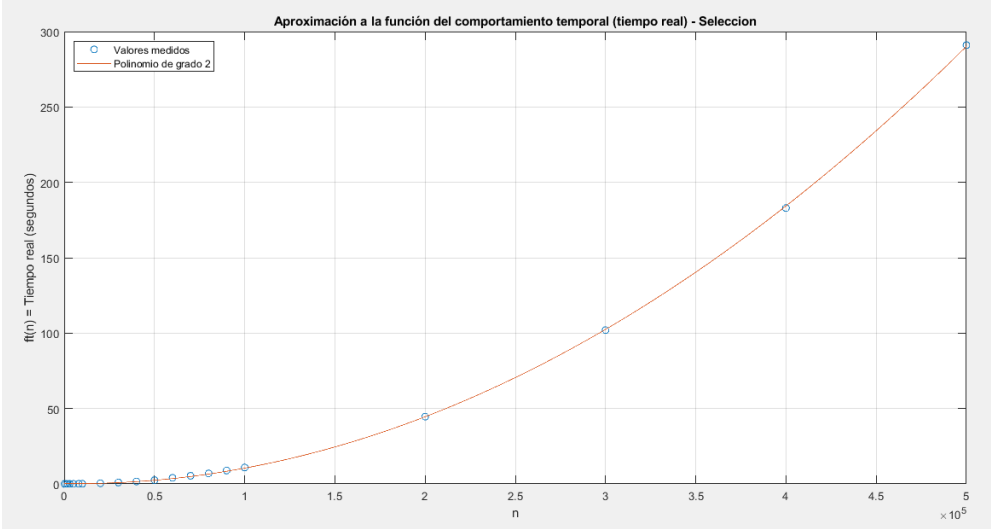

Figura 43 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo selección

Grado 3

$$y = 1.758\text{x}10^{-16}x^3 + 1.074\text{x}10^{-9}x^2 + 1.064\text{x}10^{-6}x - 0.04058$$

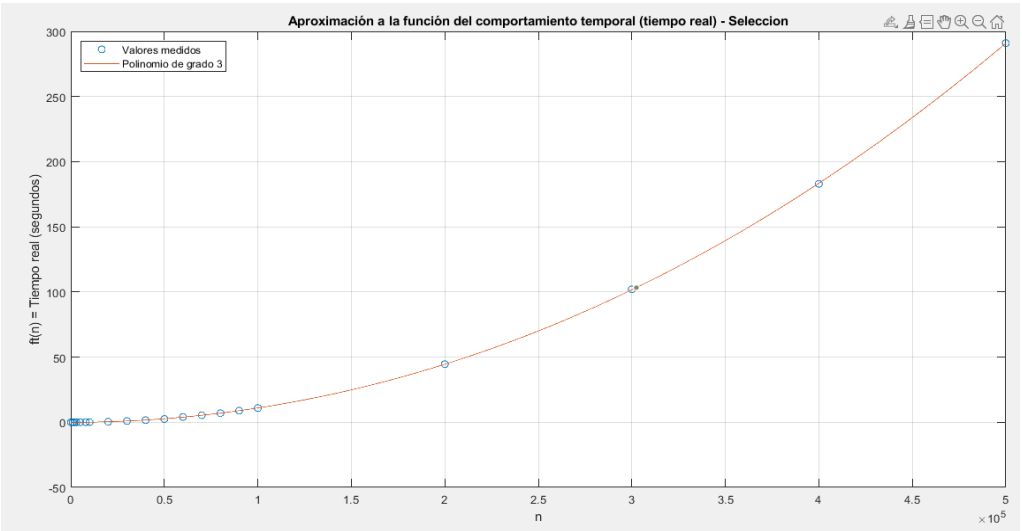

Figura 44 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo selecció

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 6

$$y = 1.555\text{x}10^{-32}x^6 - 1.931\text{x}10^{-26}x^5 + 8.693\text{x}10^{-21}x^4 - 1.612\text{x}10^{-15}x^3 + 1.252\text{x}10^{-9}x^2 - 6.676\text{x}10^{-6}x + 0.0206$$

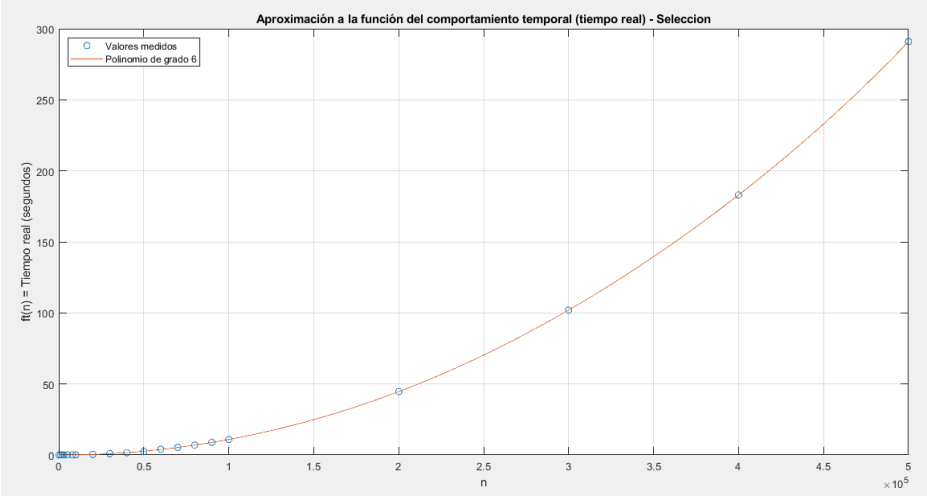

Figura 45 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo selección

Shell

Grado 1

$$y = 3.042\text{x}10^{-6}x - 0.891$$

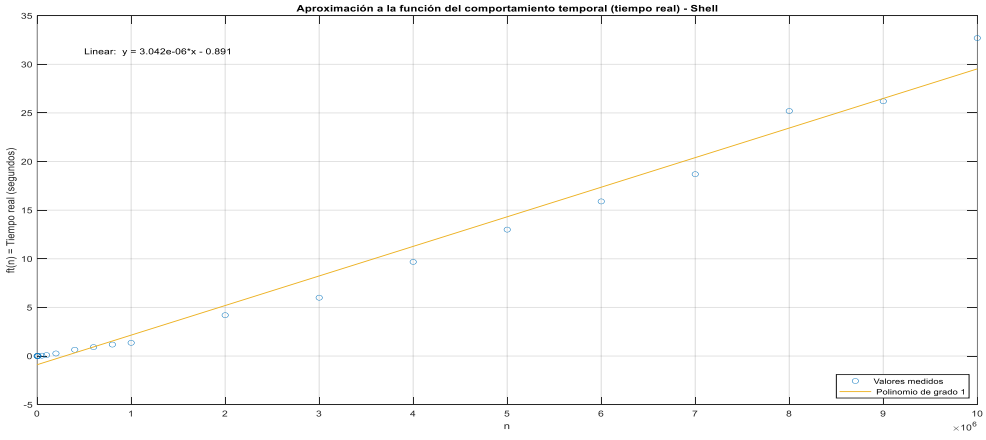

Figura 46 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo shell

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 1.433\text{x}10^{-13}x^2 + 1.807\text{x}10^{-6}x - 0.1256$$

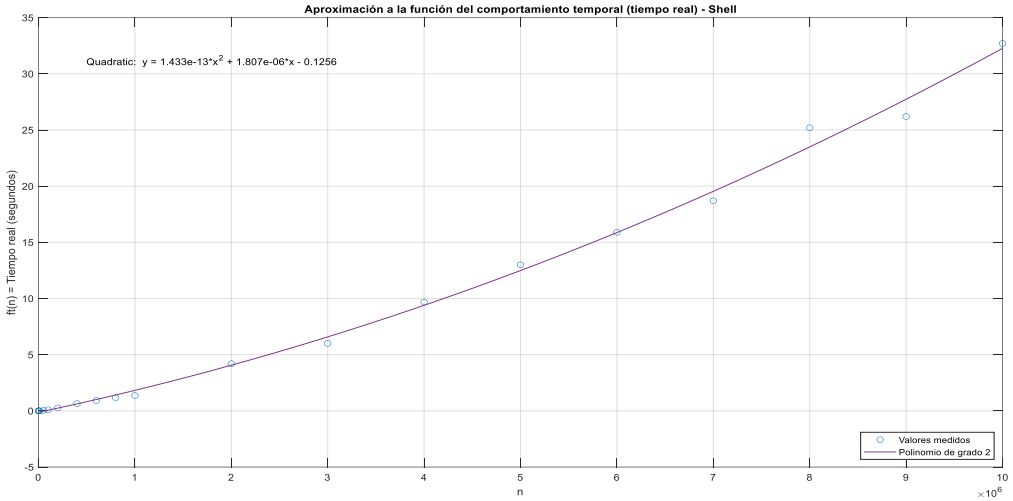

Figura 47 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo shell

Grado 3

$$y = -3.396\text{x}10^{-21}x^3 + 1.916\text{x}10^{-13}x^2 + 1.644\text{x}10^{-6}x - 0.08034$$

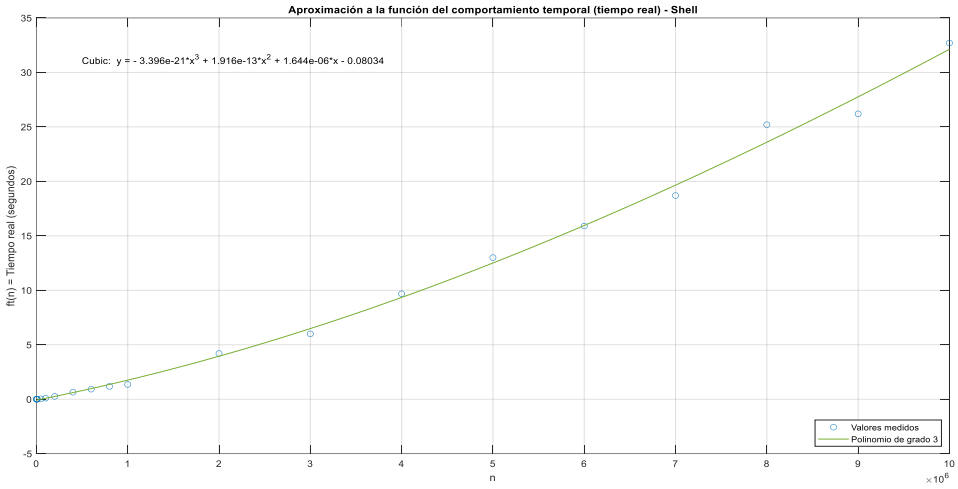

Figura 48 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo shell

Grado 6

$$y = 4.665\text{x}10^{-40}x^6 - 1.306\text{x}10^{-32}x^5 + 1.366\text{x}10^{-25}x^4 - 6.628\text{x}10^{-19}x^3 + 1.652\text{x}10^{-12}x^2 + 4.411\text{x}10^{-7}x + 0.04266$$

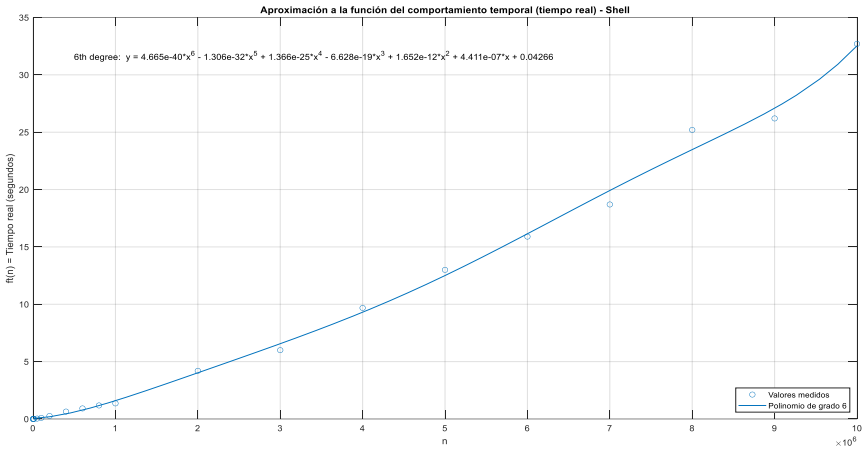

Figura 49 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo Shell

ABB

Grado 1

$$y = 1.405\text{x}10^{-6}x - 0.3627$$

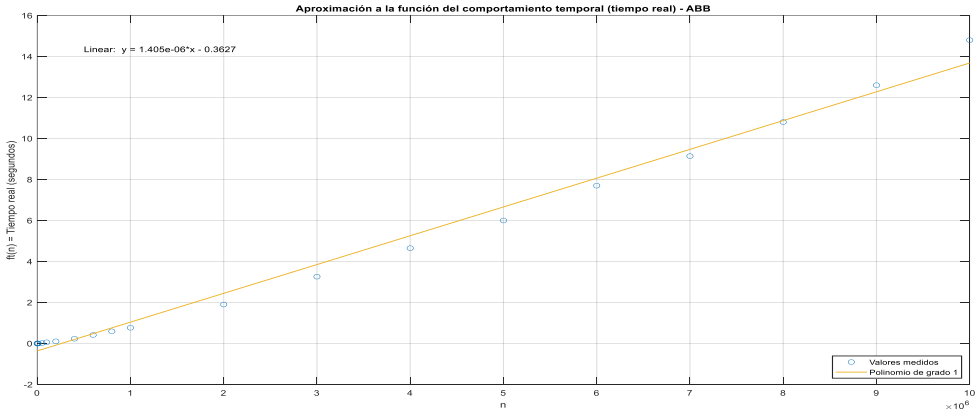

Figura 50 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo ABB

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 5.34\text{x}10^{-14}x^2 + 9.441\text{x}10^{-7}x - 0.0773$$

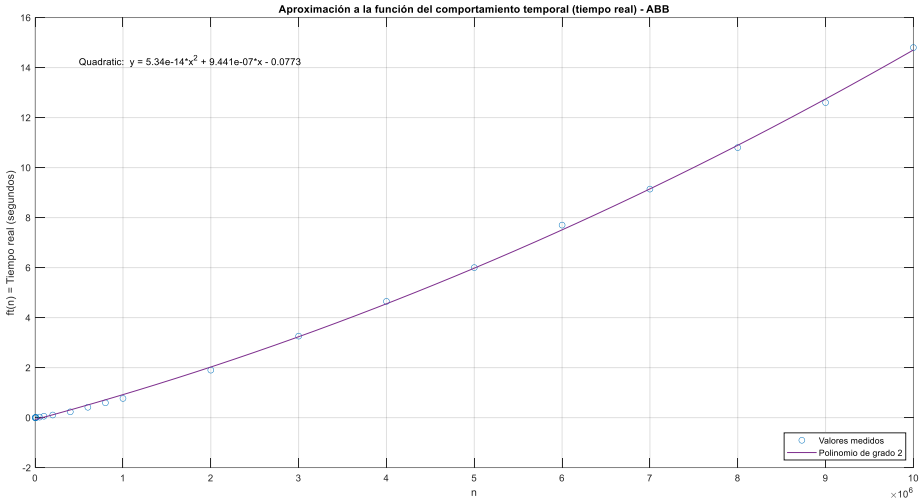

Figura 51 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo ABB

Grado 3

$$y = -1.188\text{x}10^{-21}x^3 + 7.031\text{x}10^{-14}x^2 + 8.872\text{x}10^{-7}x - 0.06148$$

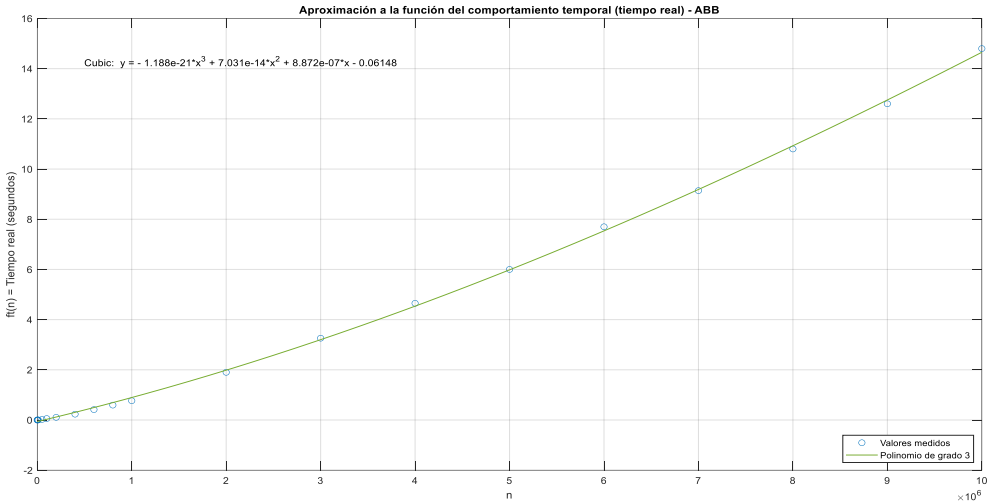

Figura 52 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo ABB

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 6

$$y = 4.812\text{x}10^{-41}x^6 - 1.391\text{x}10^{-33}x^5 + 1.645\text{x}10^{-26}x^4 - 1.034\text{x}10^{-19}x^3 + 4.005\text{x}10^{-13}x^2 + 4.64\text{x}10^{-7}x - 0.0001643$$

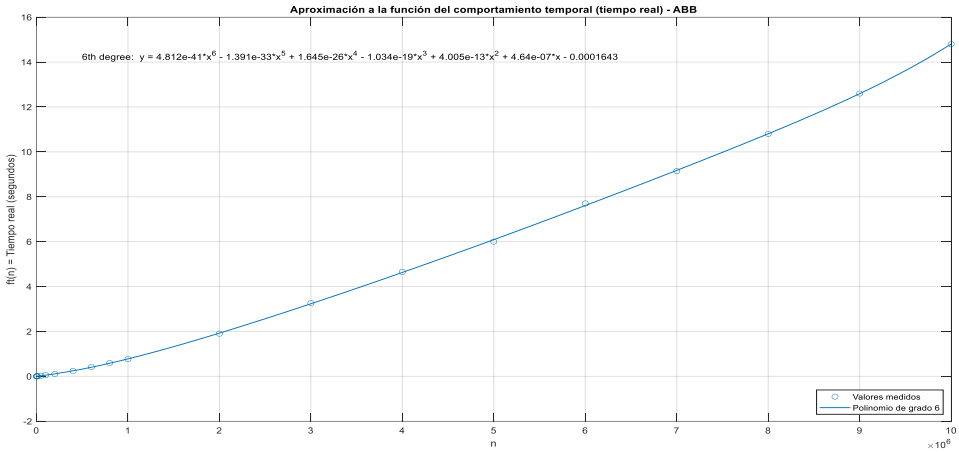

Figura 53 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo ABB

MergeSort

Grado 1

$$y = 3.098\text{x}10^{-7}x - 0.04932$$

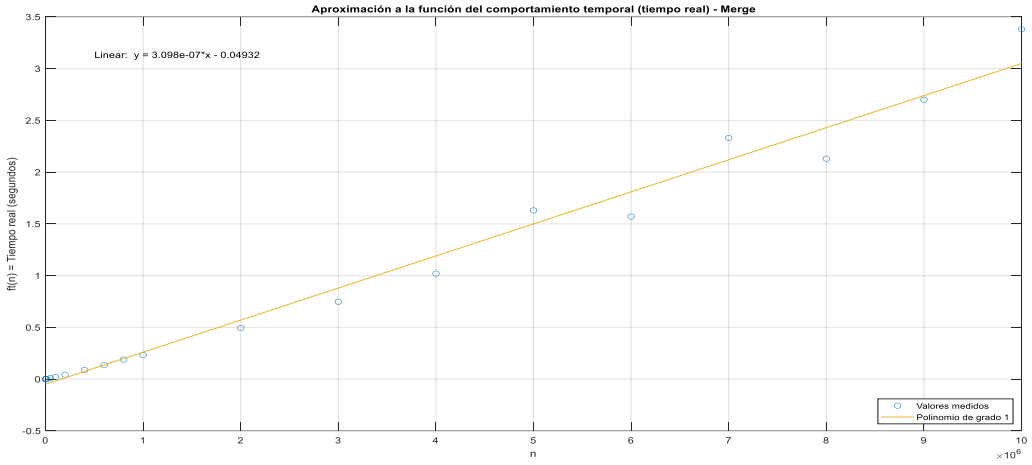

Figura 54 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo merge

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 8.804\text{x}10^{-15}x^2 + 2.339\text{x}10^{-7}x - 0.002275$$

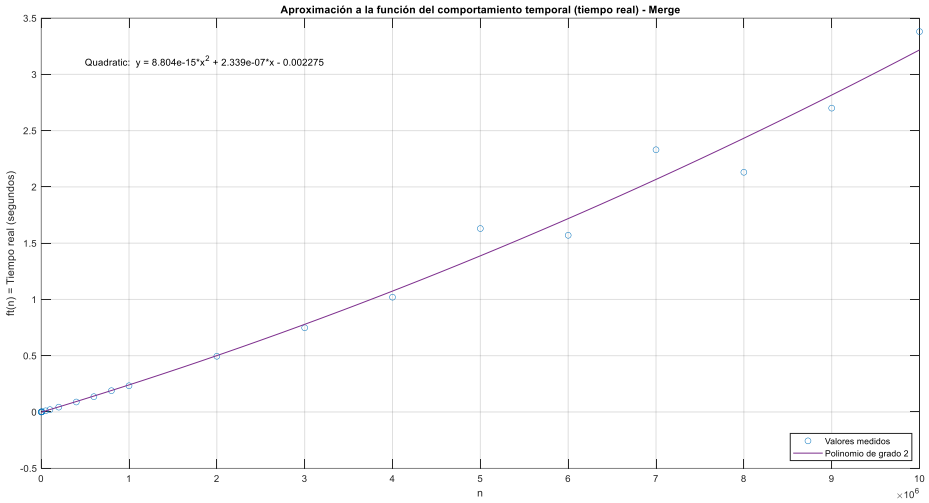

Figura 55 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo merge

Grado 3

$$y = 9.497\text{x}10^{-22}x^3 - 4.713\text{x}10^{-15}x^2 + 2.794\text{x}10^{-7}x - 0.01492$$

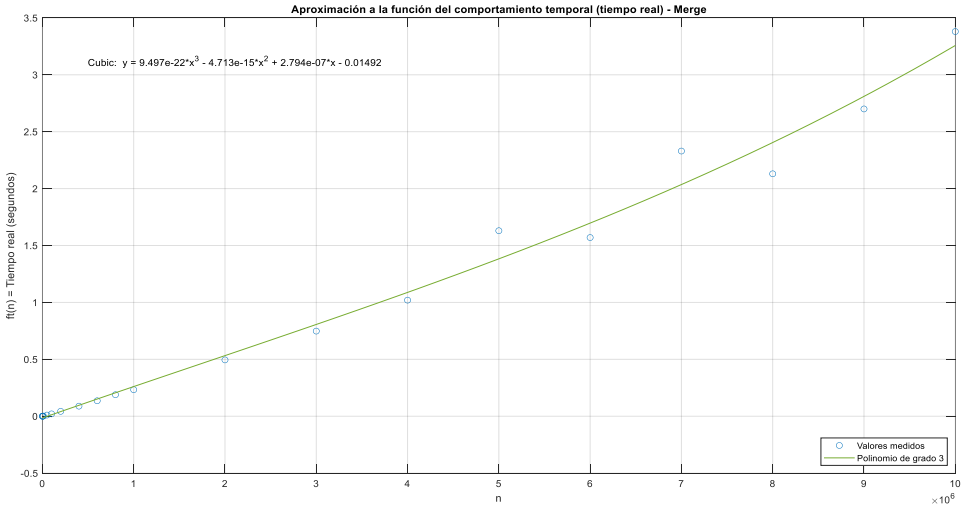

Figura 56 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo merge

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 6

$$y = 1.659\text{x}10^{-41}x^6 - 1.592\text{x}10^{-34}x^5 - 1.679\text{x}10^{-27}x^4 + 2.245\text{x}10^{-20}x^3 - 5.684\text{x}10^{-14}x^2 + 2.743\text{x}10^{-7}x - 0.004147$$

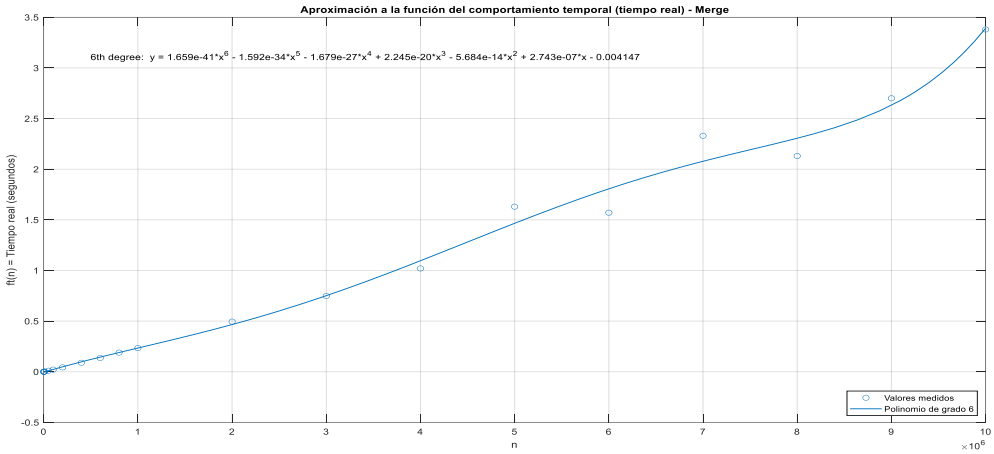

Figura 57 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo merge

QuickSort

Grado 1

$$y = 2.175\text{x}10^{-7}x - 0.01012$$

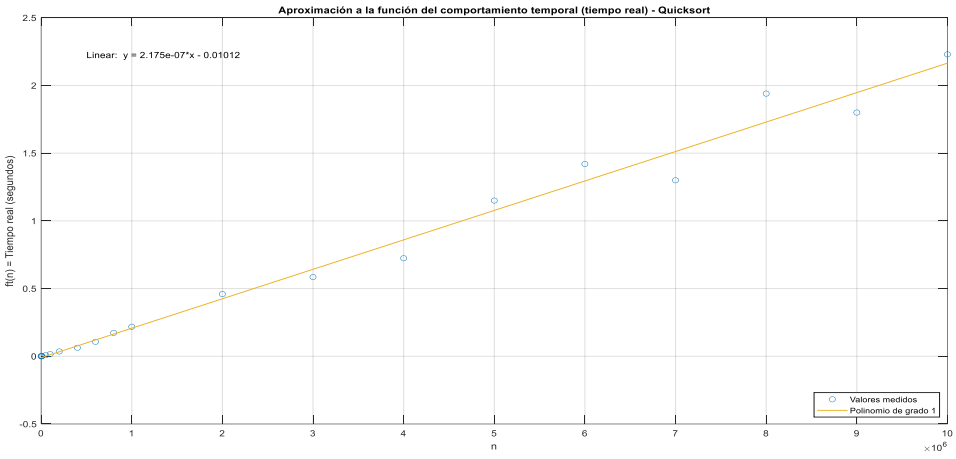

Figura 58 Gráfica de aproximación a la función de comportamiento temporal de grado 1 para el algoritmo quicksort

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 2

$$y = 1.124\text{x}10^{-15}x^2 + 2.078\text{x}10^{-7}x - 0.004116$$

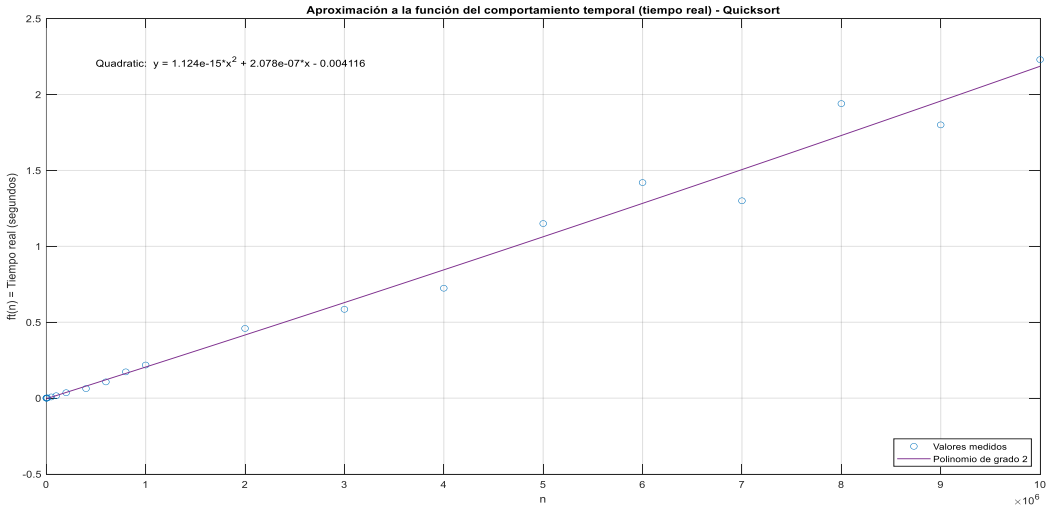

Figura 59 Gráfica de aproximación a la función de comportamiento temporal de grado 2 para el algoritmo quicksort

Grado 3

$$y = -2.004\text{x}10^{-22}x^3 + 3.977\text{x}10^{-15}x^2 + 1.982\text{x}10^{-7}x - 0.001448$$

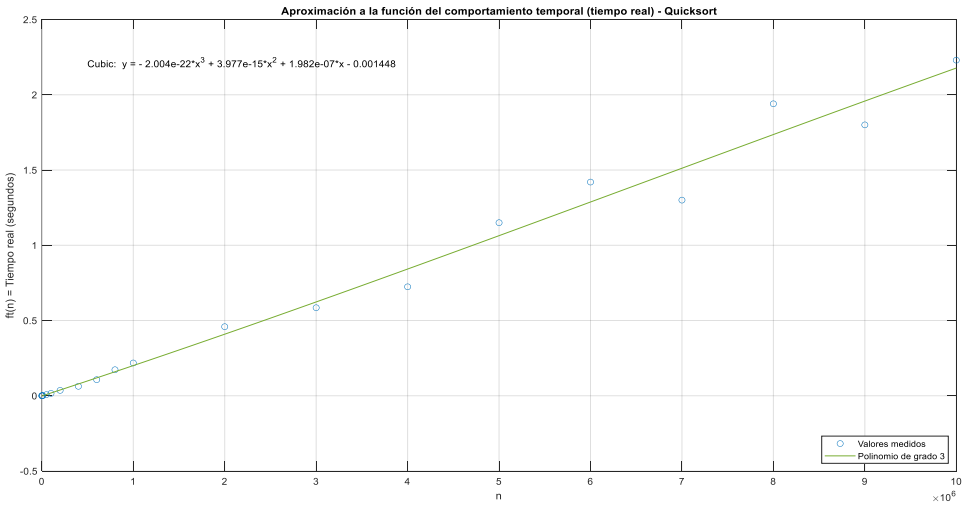

Figura 60 Gráfica de aproximación a la función de comportamiento temporal de grado 3 para el algoritmo quicksort

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Grado 6

$$y = 3.713\text{x}10^{-41}x^6 - 9.248\text{x}10^{-34}x^5 + 7.969\text{x}10^{-27}x^4 - 2.725\text{x}10^{-20}x^3 + 3.069\text{x}10^{-14}x^2 + 2.064\text{x}10^{-7}x - 0.00439$$

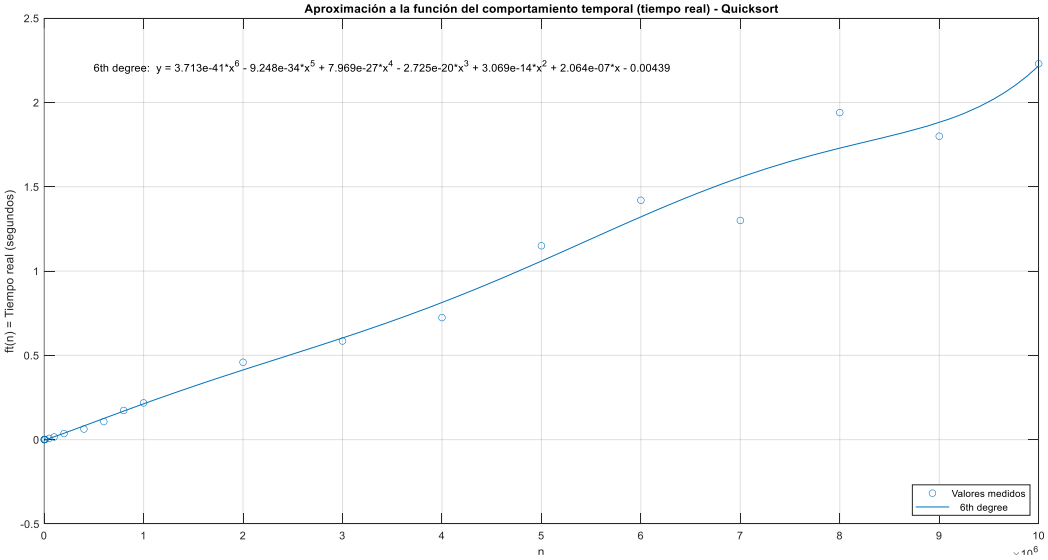

Figura 61 Gráfica de aproximación a la función de comportamiento temporal de grado 6 para el algoritmo quicksort

Comparativa de las aproximaciones de la función de complejidad temporal

Burbuja Simple

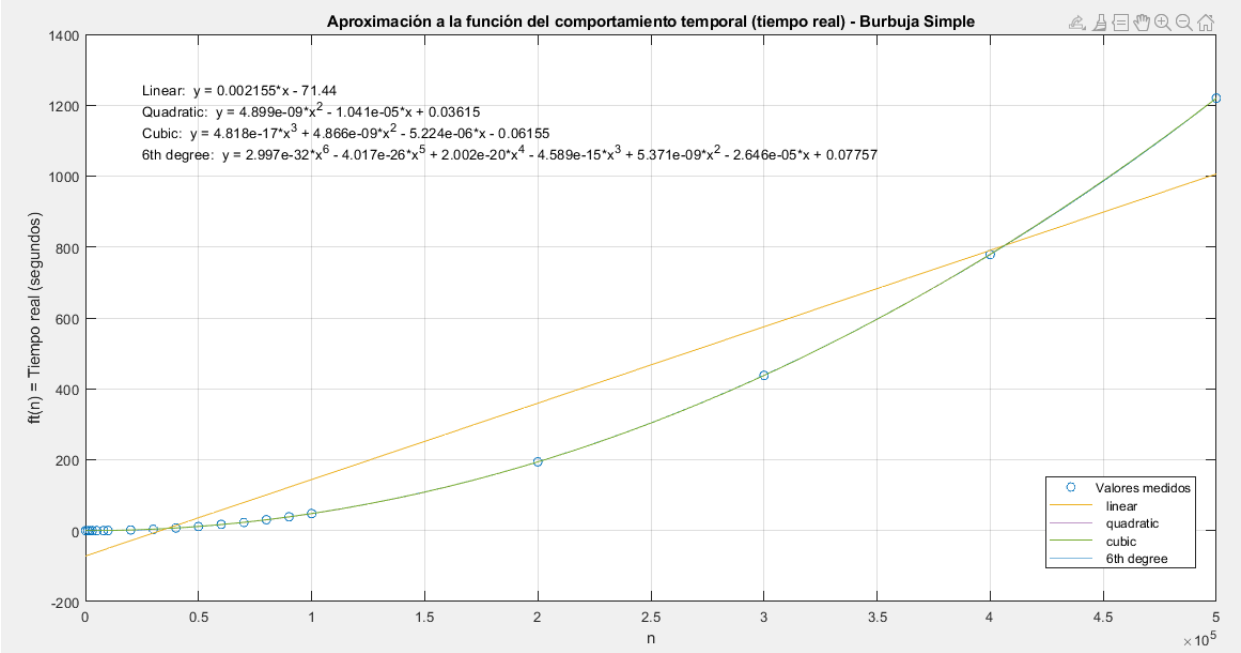

Figura 62 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo burbuja simple

Para el ordenamiento de burbuja simple, podemos notar que su aproximación se vuelve bastante cercana a los puntos de muestreo desde el polinomio de grado 2, sin embargo, al usar un polinomio de grado 6 podemos lograr la intersección de la curva con los puntos de manera más precisa.

Burbuja Optimizada 1

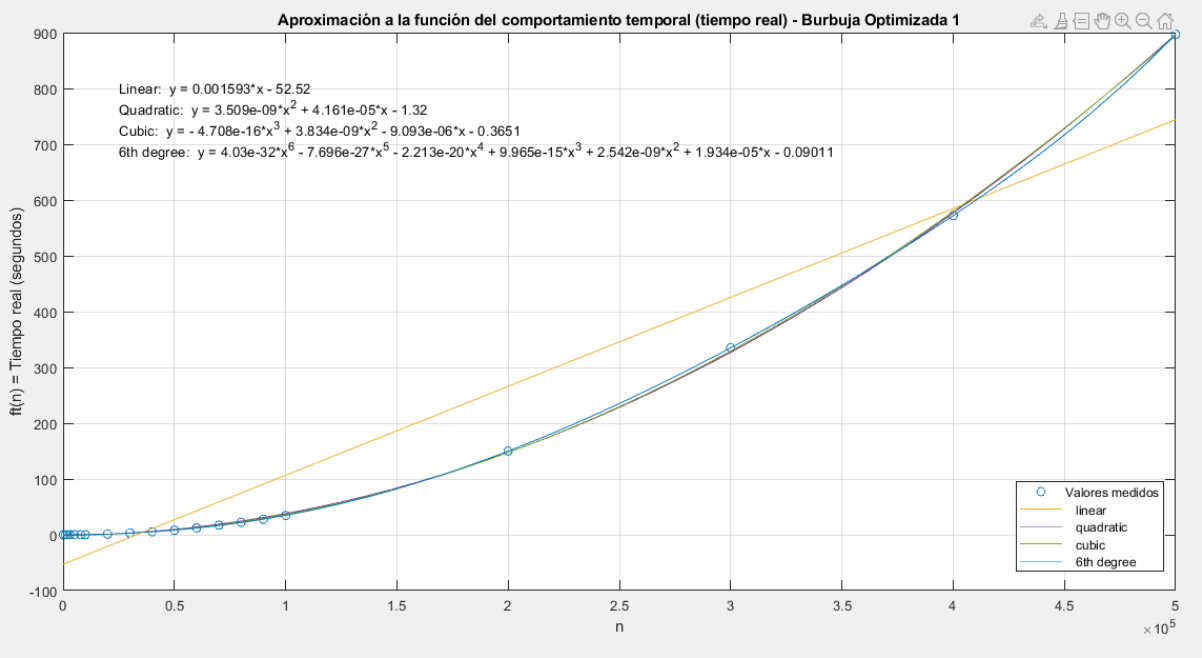

Figura 63 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo burbuja optimizada 1

El ordenamiento de burbuja optimizado 1, tiene un comportamiento similar al ordenamiento de burbuja básico, ya que con un polinomio de grado 2 se aproxima bastante a los puntos de muestreo. Sin embargo, al usar un polinomio de grado 6 podemos lograr la intersección de la curva con los puntos de manera más precisa.

Burbuja Optimizada 2

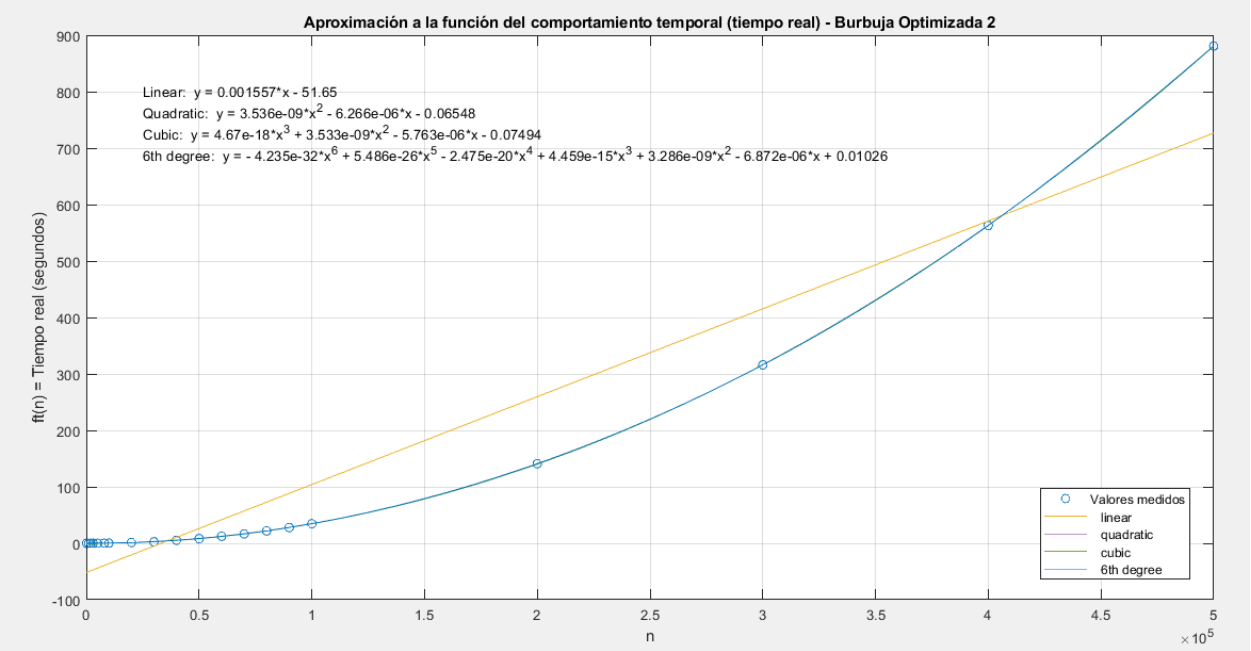

Figura 64 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo burbuja optimizada 2

El ordenamiento de burbuja optimizado 2, también logra una aproximación bastante cercana al usar un polinomio de grado 2. Sin embargo, al usar un polinomio de grado 6 podemos lograr la intersección de la curva con los puntos de manera más precisa.

Inserción

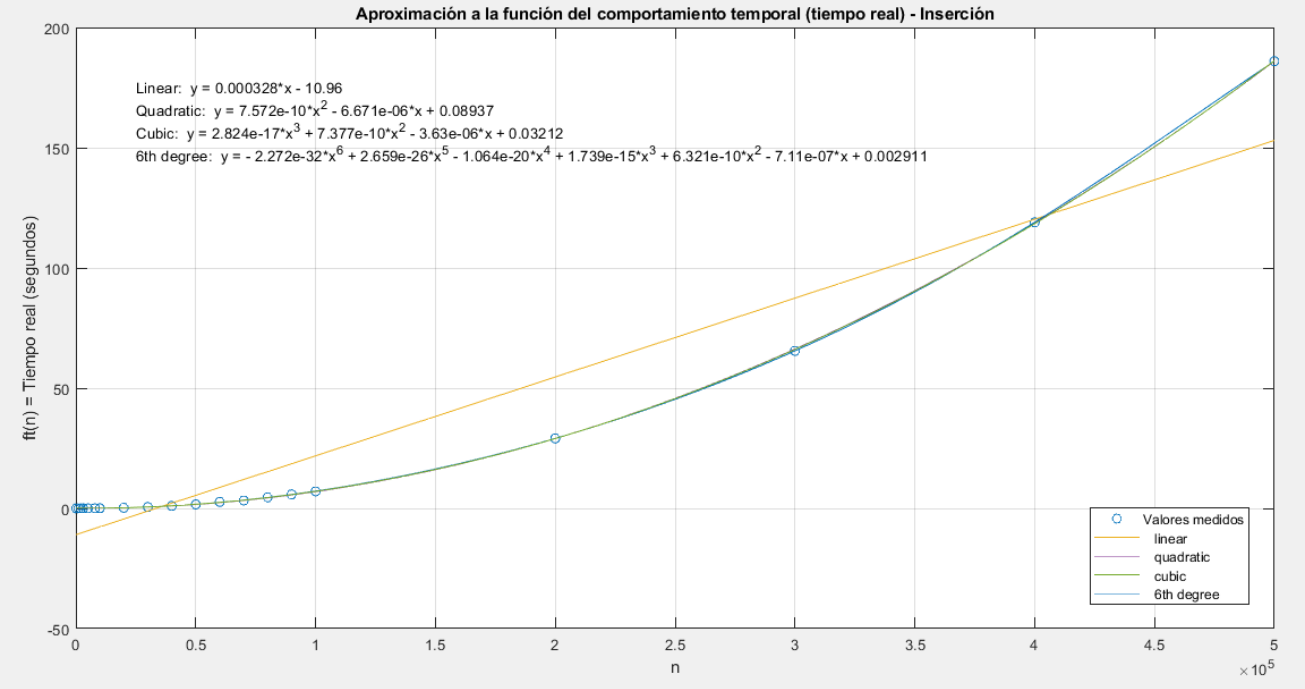

Figura 65 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo inserción

El ordenamiento por inserción logra aproximarse bastante por medio de polinomios de grado 2 y 3, sin embargo, el uso de un polinomio de grado 6, permite la intersección más precisa entre los puntos de muestreo y su curva generada.

Selección

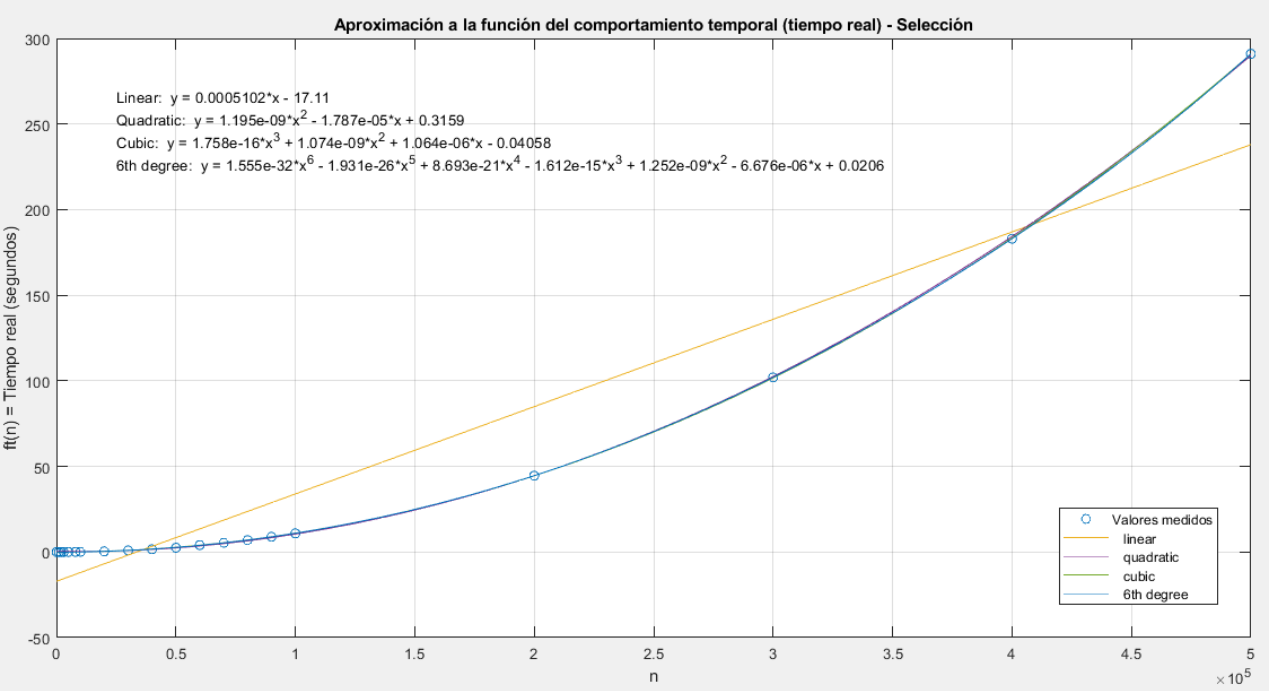

Figura 66 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo selección

El ordenamiento por selección logra aproximarse bastante por medio de polinomios de grado 2 y 3, sin embargo, el uso de un polinomio de grado 6, permite la intersección más precisa entre los puntos de muestreo y su curva generada.

Shell

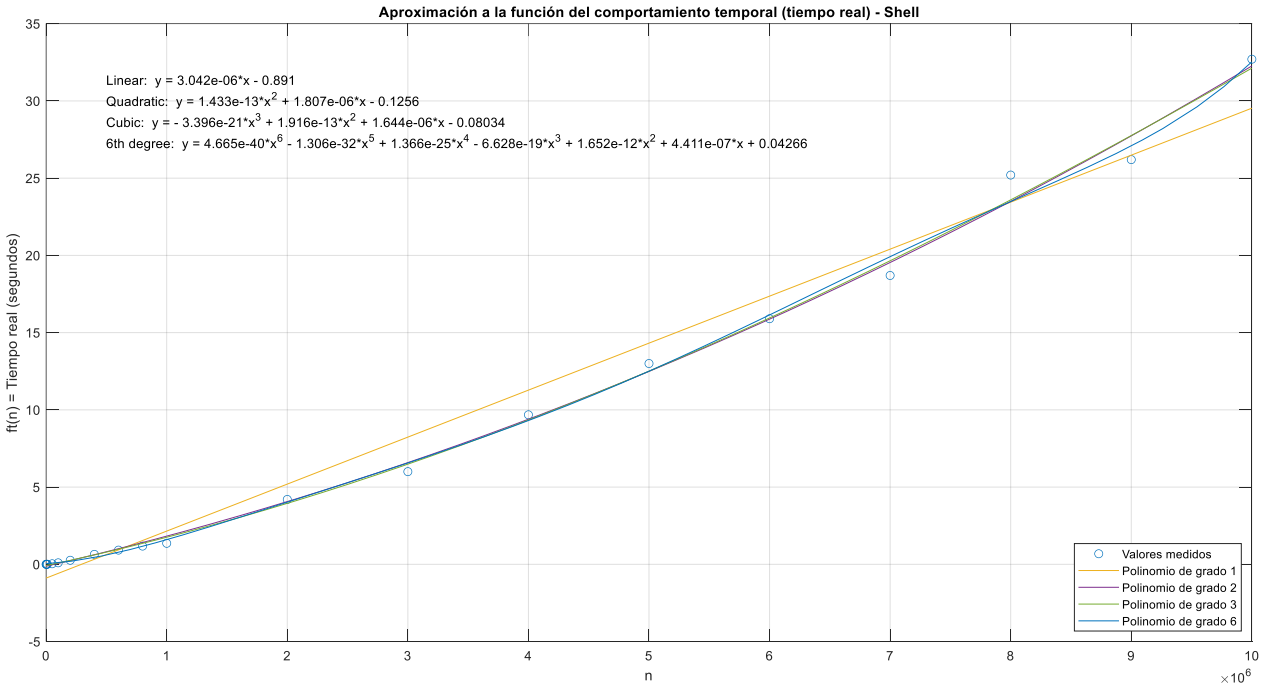

Figura 67 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo shell

El ordenamiento por el algoritmo Shell no logra la intersección total de todos sus puntos por medio de ninguna aproximación polinomial. Sin embargo, el modelo del polinomio de grado 6 logra acercarse bastante a los puntos de muestreo, siendo este el modelo más cercano a los datos del muestreo.

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

ABB

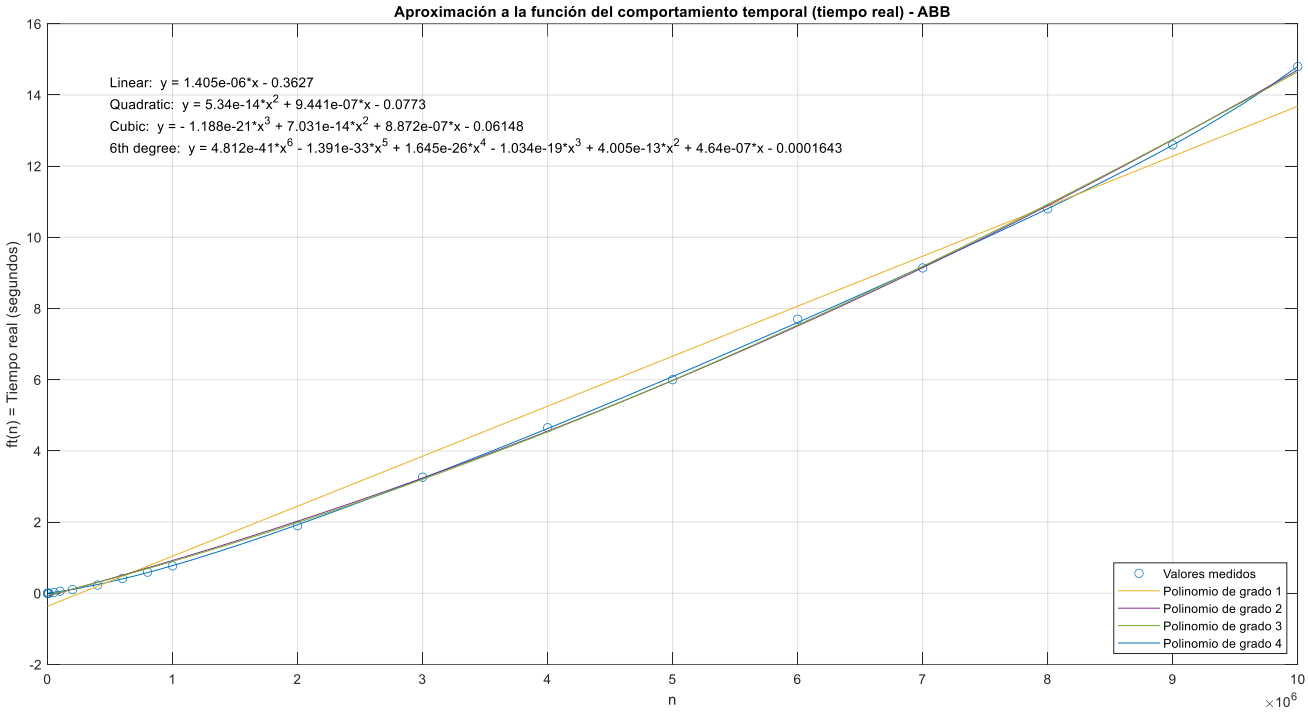

Figura 68 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo ABB

El ordenamiento basado en ABB se aproxima bastante desde el uso de polinomios de grado 2 y 3, pero logra una mayor precisión en las intersecciones por medio del polinomio de grado 6.

MergeSort

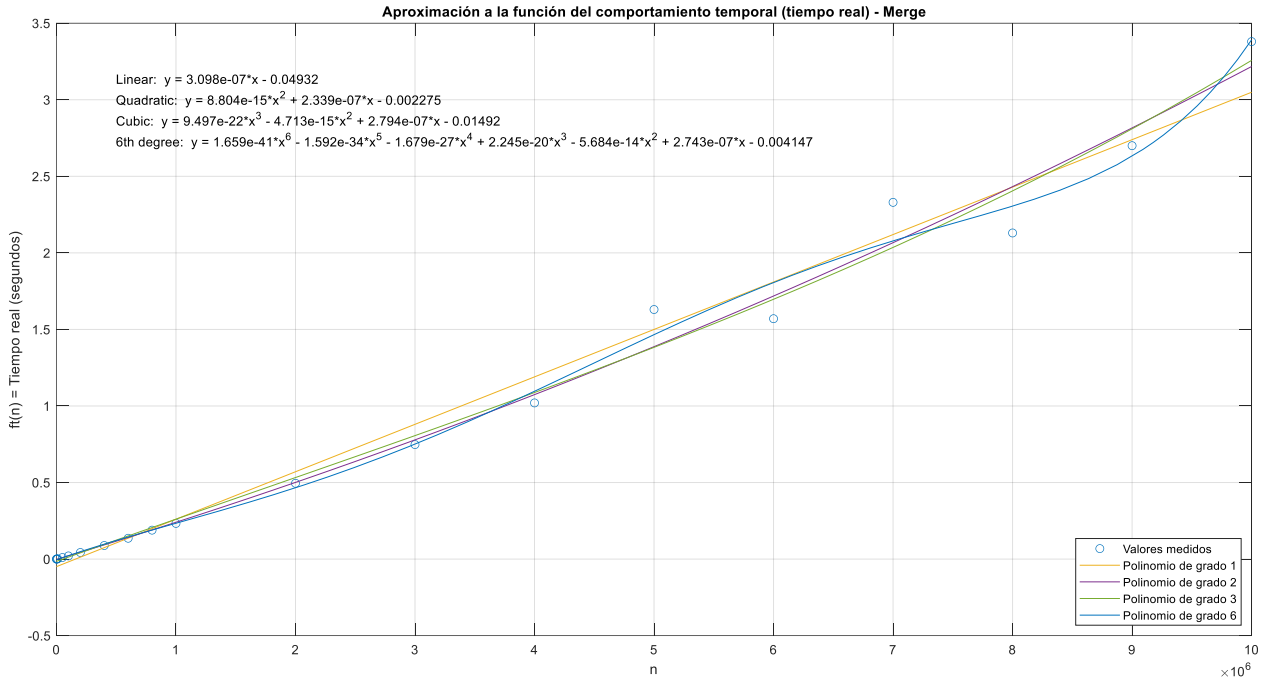

Figura 69 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo merge

En el caso de ordenamiento por Merge, se puede notar que los modelos por aproximación polinomial no logran la intersección total con los puntos de muestreo. La curva del polinomio de grado 6 logra una mejor aproximación ya que la distancia de separación entre los puntos de muestreo y la curva tienen una menor magnitud de separación, con respecto a los otros polinomios.

Quicksort

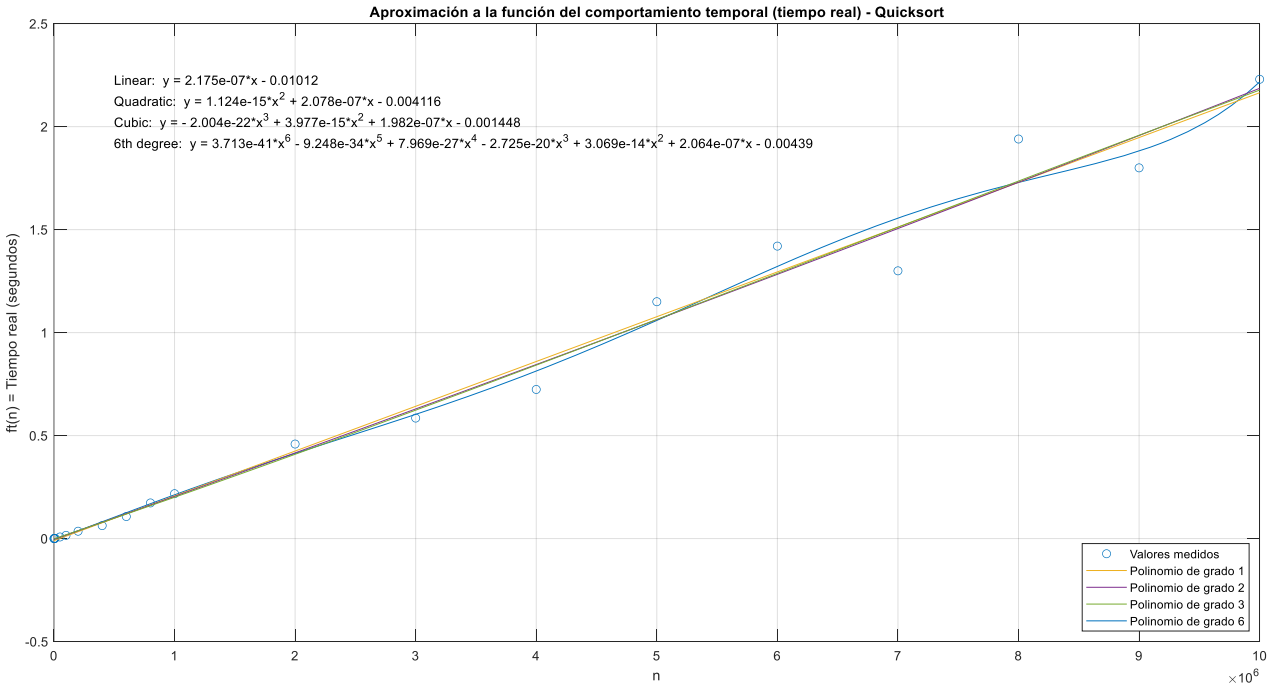

Figura 70 Gráfica comparativa de aproximación a la función del comportamiento temporal para el algoritmo quicksort

El ordenamiento por Quicksort presenta el mismo caso que el ordenamiento por Merge, ya que la curva generada en la aproximación polinomial no logra cruzar con todos los puntos de muestreo. A causa de esto, se considera que el modelo generado por el polinomio de grado 6 es el que logra el mejor ajuste, permitiendo que la magnitud de separación entre curva y puntos sea la menor con respecto a los otros modelos.

Comparación entre algoritmos

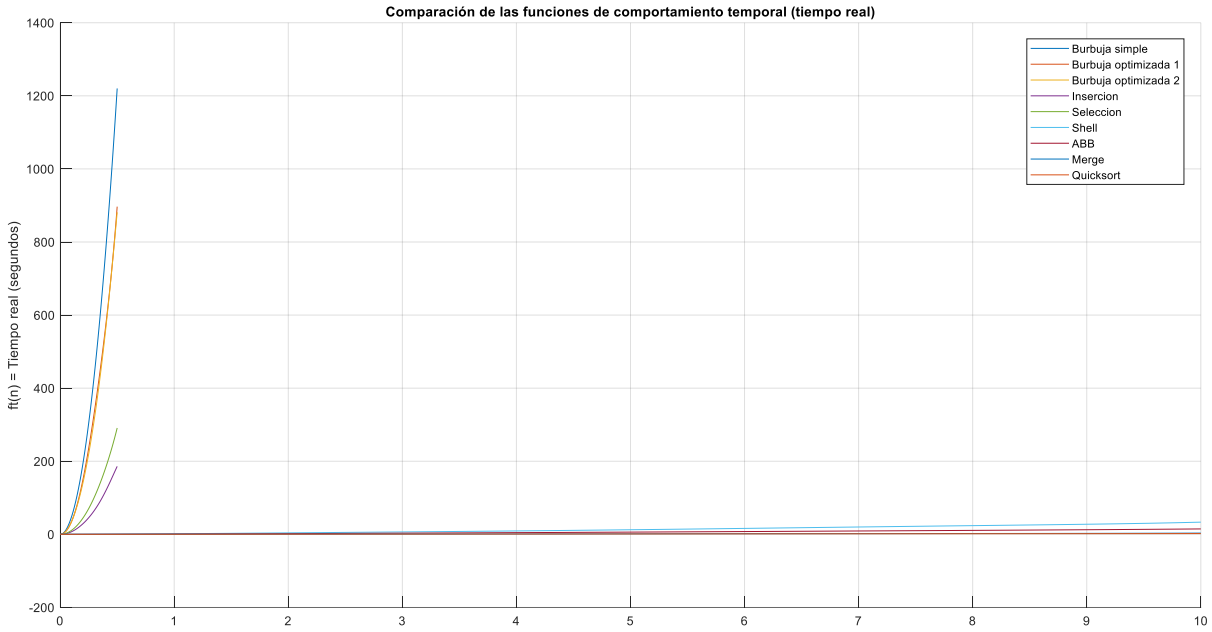

Figura 71 Gráfica comparativa de los 9 algoritmos de ordenamiento

Debido a que los algoritmos de burbuja, inserción y selección fueron probados con valores de N que iban de 0 a 500000, las escalas de los algoritmos no permiten realizar una comparación clara entre ellos. Para dar solución a esto, se presentan dos nuevas gráficas que permiten visualizar de un lado los algoritmos con una N máxima de 500000 y del otro lado los de N máxima de 10000000, de esta forma las escalas nos permiten visualizar claramente el comportamiento de cada algoritmo.

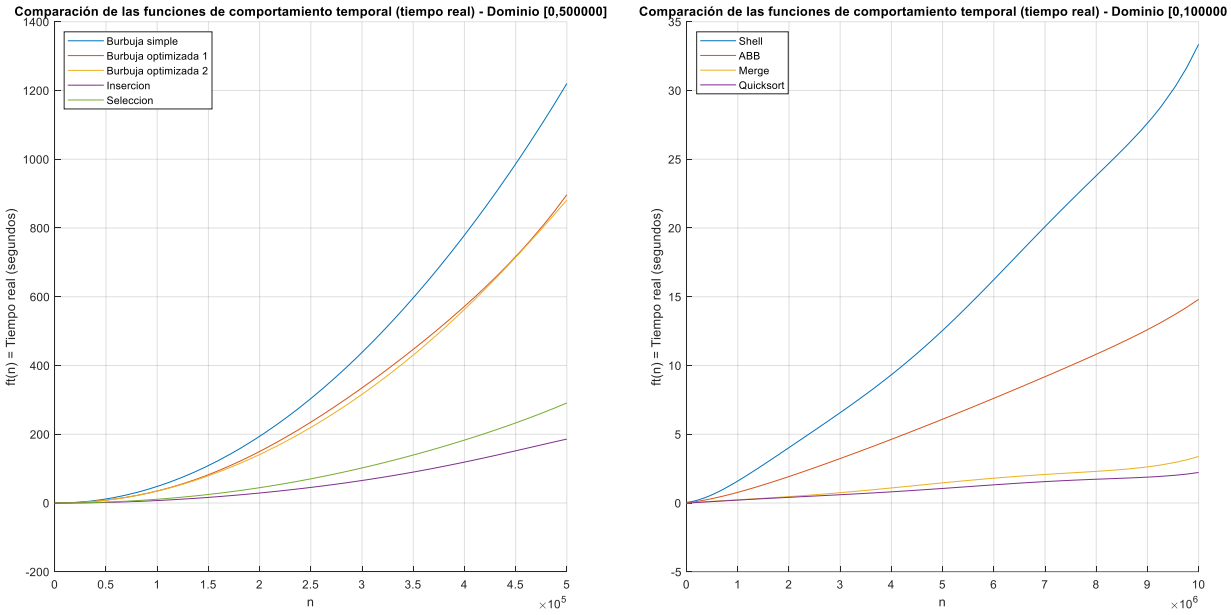

Figura 72 Gráfica comparativa de los 9 algoritmos de ordenamiento, clasificadas según su dominio (N)

Aproximación en tiempo real para ordenar datos

De acuerdo con el punto 8, los tiempos (en segundos) para ordenar 15,000,000, 20,000,000, 500,000,000, 1,000,000,000, 5,000,000,000 para cada algoritmo está dada por la siguiente tabla.

VALOR	Burbuja Simple	Burbuja optimización 1	Burbuja optimización 2	Inserción	Selección	SHELL	ABB	MERGE	QUICK
15,000,000	3.11872E+11	4.52112E+11	-4.41971E+11	-2.39136E+11	1.62896E+11	453.073223	72.7048357	50.1686968	42.1226413
20,000,000	1.7927E+12	2.55111E+12	-2.53877E+12	-1.37068E+12	9.34786E+11	5287.26466	602.759836	446.025853	490.39961
500,000,000	4.67027E+20	6.29446E+20	-6.60006E+20	-3.5417E+20	2.42366E+20	6.8894E+12	7.0942E+11	2.5414E+11	5.5175E+11
1,000,000,000	2.99299E+22	4.02923E+22	-4.22952E+22	-2.26934E+22	1.55307E+22	4.5358E+14	4.6745E+13	1.6429E+13	3.6213E+13
5,000,000,000	4.68156E+26	6.29663E+26	-6.61547E+26	-3.54917E+26	2.42908E+26	7.2483E+18	7.4754E+17	2.5872E+17	5.7727E+17

Figura 73 Tabla de aproximación en tiempo real para ordenar 15,000,000, 20,000,000, 500,000,000, 1,000,000,000, 5,000,000,000

Cabe aclarar que los algoritmos de burbuja optimización 2 e inserción no se tomaron en cuenta los tiempos, ya que, al momento de realizar los cálculos con la función obtenida, los resultados salen de forma negativa, además de que, para estos algoritmos al tratar de ordenar más de 500,000 datos, son imposibles de ordenar.

Cuestionario

1. ¿Cuál de los 8 algoritmos es el más fácil de implementar?
    a. Burbuja simple
2. ¿Cuál de los 8 algoritmos es el más difícil de implementar?
    a. El ordenamiendo basado en ABB
3. ¿Cuál algoritmo tiene menor complejidad temporal?
    a. Observando las gráficas y tablas comparativas, Quicksort, aunque se acerca demasiado el algoritmo de mergesort.
4. ¿Cuál algoritmo tiene mayor complejidad temporal?
    a. Observando las gráficas y tablas comparativas, el algoritmo de burbuja simple.
5. ¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?
    a. Los de burbuja, selección, inserción y shell, ya que la memoria que usa el algoritmo como tal es constante.
6. ¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?
    a. El algoritmo de mergesort y el ABB, ya que hacen uso de elementos de memoria que dependen del tamaño del problema, para ordenar hacen uso de un arreglo y un árbol respectivamente.
7. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?
    a. Los resultados de la experimentación fueron los esperados debido a que previamente se había estudiado el comportamiento teórico de los algoritmos, por ello se tenía una noción de como serían sus tiempos de ejecución en comparación.
8. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?
    a. Para la realización de las pruebas experimentales se trató de crear las mismas condiciones durante cada ejecución del programa, manteniendo únicamente el programa en ejecución además de los procesos existentes en el sistema. También, se utilizó un mismo equipo para todas las pruebas realizadas, evitando variaciones por los recursos físicos de la computadora. Los recursos del equipo son: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz - 2.90 GHz (RAM 8GB).
9. ¿Facilitó las pruebas mediante scripts u otras automatizaciones? ¿Cómo lo hizo?
    a. Si, ya que en un solo archivo .sh se puede escribir varias compilaciones de los distintos algoritmos y sus respectivas ejecuciones, entonces disminuye el tiempo en el que se le emplea para la compilación y ejecución
10. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?
    a. Tener un conocimiento previo de los algoritmos de ordenación, ya que se nos resultó más sencillo ver que algoritmo iba a realizar el menor tiempo. Además de

un conocimiento básico de MATLAB ya que al momento de realizar la aproximación a la función del comportamiento temporal se nos resulto algo sencillo y rápido el poder obtener las funciones y graficarlas.

Anexos

Códigos

Burbuja.c

```c
/*
* Titulo: Practica 1, algoritmo "ordenamiento de burbuja"
* Descripcion: Implementacion del algoritmo de ordenamiento Burbuja
* Fecha: 13-sep-2021
* Version: 1.0
*/

#include<stdio.h>
#include<stdlib.h>

void burbujaSimple(int A[], int n);/*Prototipo de la funcion*/
/*
*Recibe: Un areglo y el tamanio
*Regresa: void
*Errores: Ninguno
*/

/*Funcion principal*/
int main(int argc, char* argv[]){
    int i, n=atoi(argv[1]); /*Tomar el segundo argumento como tamaño del algoritmo*/
    int *A = (int*)malloc(sizeof(int)*n); /*Reserva memoria para el arreglo*/

    for (i = 0; i < n; i++){/*Llena el arreglo*/
        scanf("%d",&A[i]);
    }

    burbujaSimple(A,n);/*Llamada a la funcion de ordenamiento*/

    for (i = 0; i < n; i++){/*Impresion de los elementos del arreglo*/
        printf("%d\n",A[i]);
    }

    free(A);/*Libera la memoria*/

    return 0;
}

/*Definicion de funciones*/
```


```
void burbujaSimple(int A[], int n){/*Recibe un arreglo y el tamanio del
arreglo*/
    int i, j, aux;/*declaracion de variables*/

    for(i=0; i<=n-2; i++){
        for(j=0; j<=n-2; j++){
            if(A[j]>A[j+1]){/*Si el elemento actual es mayor que el
siguiente los intercambia*/
                aux = A[j];
                A[j] = A[j+1];
                A[j+1] = aux;
            }
        }
    }
}
```

Figura 74. Código del algoritmo burbuja

Burbuja1.c

```
/*
* Titulo: Practica 1, algoritmo "Ordenamiento de Burbuja Osptimiazion 1"
* Descripcion: Implementacion del algoritmo Burbuja con una primera
optimizacion
* Fecha: 13-sep-2021
* Version: 1.0
*/
#include<stdio.h>
#include<stdlib.h>

void burbujaOpt1(int A[], int n);/*Protipo de la funcion*/
/*
*Recibe: Un areglo y el tamanio
*Regresa: void
*Errores: Ninguno
*/

/*Funcion principal*/
int main(int argc, char* argv[]){
    int i, n=atoi(argv[1]); /*Tomar el segundo argumento como tamaño del
algoritmo*/
    int *A = (int*)malloc(sizeof(int)*n); /*Reserva memoria para el
arreglo*/

    for (i = 0; i < n; i++){/*Llena el arreglo*/
        scanf("%d",&A[i]);
    }

    burbujaOpt1(A,n);/*Llamada a la funcion de ordenamiento*/

    for (i = 0; i < n; i++){/*Impresion de los elementos del arreglo*/
        printf("%d\n",A[i]);
    }

    free(A);/*Libera la memoria*/

    return 0;
}

/*Definicion de funciones*/
void burbujaOpt1(int A[], int n){/*recibe un arreglo y el tamanio de este*/
    int i, j, aux;/*declaracion de variables*/

    for(i=0; i<=n-2; i++){
```


```
        for(j=0; j<=n-2-i; j++){/*Optimizacion: Despues de cada iteracion el
mayor queda ordenado, ya no se compara.*/
            if(A[j]>A[j+1]){/*Si el elemento actual es mayor que el
siguiente los intercambia*/
                aux = A[j];
                A[j] = A[j+1];
                A[j+1] = aux;
            }
        }
    }
}
```

Figura 75. Código del algoritmo burbuja optimización 1

Burbuja2.c

```
/*
* Titulo: Practica 1, algoritmo "Ordenamiento de Burbuja Optimizacion 2"
* Descripcion: Implementacion del algoritmo Burbuja con una segunda
optimizacion
* Fecha: 13-sep-2021
* Version: 1.0
*/
#include<stdio.h>
#include<stdlib.h>

void burbujaOpt2(int A[], int n);/*Prototipo de la funcion*/
/*
*Recibe: Un areglo y el tamanio
*Regresa: void
*Errores: Ninguno
*/

/*Funcion principal*/
int main(int argc, char* argv[]){
    int i, n=atoi(argv[1]); /*Tomar el segundo argumento como tamaño del
algoritmo*/
    int *A = (int*)malloc(sizeof(int)*n); /*Reservacion de memoria para el
arreglo*/

    for (i = 0; i < n; i++){/*Llena el arreglo*/
        scanf("%d",&A[i]);
    }

    burbujaOpt2(A,n);/*Llamada a la funcion de ordenamiento*/

    for (i = 0; i < n; i++){/*Impresion de los elementos del arreglo*/
        printf("%d\n",A[i]);
    }

    free(A);/*Libera la memoria*/

    return 0;
}

/*Definicion de funciones*/
void burbujaOpt2(int A[], int n){/*recibe un arreglo y el tamanio de este*/
    int i, j, aux, cambios;/*declaracion de variables*/

    cambios = 1;/*bandera*/
```


```
    i = 0;
    while(i <= n-1 && cambios != 0){/*Revisa si hubo cambios en el arreglo*/
        cambios = 0;/*Se coloca la bandera en sin cambios*/
        for(j=0; j<=n-2-i; j++){/*Optimizacion: Despues de cada iteracion el
mayor queda ordenado, ya no se compara.*/
            if(A[j]>A[j+1]){/*Si el elemento actual es mayor que el
siguiente los intercambia*/
                aux = A[j];
                A[j] = A[j+1];
                A[j+1] = aux;
                cambios = 1;/*Mientras entre a la condicion hay cambios*/
            }
        }
        i = i+1;
    }
}
```

Figura 76. Código del algoritmo burbuja optimización 2

Gallos de Oro
Análisis de algoritmos
3CM13
Practica 01: “Pruebas a posteori”

Insercion.c

```
/*
TItulo: Practica 1, algoritmo "Ordenamiento por Insercion"
Descripcion: Implementacion del algoritmo de insercion
Fecha: 13/09/2021
Version 1.0
*/

#include<stdio.h>
#include<stdlib.h>

#define TIPO int //tipo de dato que se ingresaran
#define FORMATO "%d" //formato de datos que se ingresaran

//Se declara el esqueleto de la funcion insercion
void Insercion(int *, int);

int main(int argc, char* argv[])
{
	int n;
	//recibiendo el tamanio
	if (argc != 2)
	{
		printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n", argv[0]);
		exit(1);
	}
	//Tomar el segundo argumento como tamaÃ±o del algoritmo
	else
	{
		n = atoi(argv[1]);
	}
	/*Se declara un apuntador de tipo enetero del tamaÃ±o del numero de
elemntos que este
	va a tener*/
	TIPO * A = (TIPO *)malloc(sizeof(TIPO) * n);
	TIPO i;

	//Se introducen los elemntos por medio de la terminal mediante el
archivo txt
	for(i = 0; i<n; i++)
		scanf(FORMATO, &A[i]);

	//Se manda a llamar a la funcion Seleccion
	Insercion(A,n);
```


```
}

/*
Version 1.0
La siguiente funcion realiza un ordenamiento por el metodo de insercion
Recibe como parametros un arrego y el tamaÃ±o del arreglo
La funcion es de tipo void, por lo tanto no devuelve ningun valor
*/
void Insercion(TIPO * A, TIPO n)
{
        /*Variables a utilizar*/
        TIPO i; //Servira para iterar el for y acceder al valor en una
posicion del arreglo
        TIPO temp;//Almacenara el valor que se encuentre en la poscion del
arreglo
        TIPO j;// Servira para acceder

        /*Desarrollo del algoritmo*/

        for(i = 0; i<=n-1; i++) //Realiza un ciclo for desde i = 0 hasta el
numero de elementos en el arreglo menos 1
        {
                j = i; //Se iguala el valor de j con el de i
                temp = A[i]; //Se le asigna el valor del arreglo en la
posicion que valla la vaiable i a la varible temp

                while(j>0 && (temp<A[j-1]))//Realiza un ciclo while mientras j
sea mayor a 0 y que el valor en temporal sea menor al valor del arreglo en la
posicion j-1
                {
                        A[j] = A[j-1]; //Toma el valor que esta en la posicion
j-1 y lo asigna al valor en la posicon j
                        j--;//Se decremnta el valor de j
                }
                A[j] = temp;//Asigna el valor tomado
        }

}
```

Figura 77. Código del algoritmo inserción

Seleccion.c

```
/*
TItulo: Practica 1, algoritmo "Ordenamiento por Seleccion"
Descripcion: Implementacion del algoritmo de seleccion
Fecha: 13/09/2021
Version 1.0
*/

#include<stdio.h>
#include<stdlib.h>
#define TIPO int //tipo de dato que se ingresaran
#define FORMATO "%d" //formato de datos que se ingresaran

//se hace el esqueleto de la funcion seleccion
void Seleccion(TIPO *, TIPO);

int main(int argc, char* argv[])
{
        int n;
        //recibiendo el tamanio
        if (argc != 2)
        {
                printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n", argv[0]);
                exit(1);
        }
        //Tomar el segundo argumento como tamaÃ±o del algoritmo
        else
        {
                n = atoi(argv[1]);
        }

        /*Se declara un apuntador de tipo enetero del tamaÃ±o de numero de
elemntos que este
        va a tener*/
        TIPO * A = (TIPO *)malloc(sizeof(TIPO) * n);
        TIPO i;

        //Se introducen los elemntos por medio de la terminal mediante el
archivo txt
        for(i = 0; i<n; i++)
                scanf(FORMATO, &A[i]);

        //Se manda a llamar a la funcion Seleccion
        Seleccion(A,n);
```


```
}
/*
Version 1.0
La siguiente funcion realiza un ordenamiento por el metodo de seleccion
Busca l minimo elemento entre una posicion k y el final del areglo e
intercambia el minimo con el elemento en la
posicion k
Recibe como parametros un arrego y el tamaÃ±o del arreglo
La funcion es de tipo void, por lo tanto no devuelve ningun valor
*/
void Seleccion(TIPO * A, TIPO n){
        /*Variables a utilizar*/
        TIPO k; //Servira para iterar el for y acceder al valor en una
posicion del arreglo
        TIPO temp;//Almacenara el valor que se encuentre en la poscion del
arreglo
        TIPO p;// Servira para almacenar la posicion del valor mas pequeÃ±o
del arreglo por cada iteracion
        TIPO i;//Servira para recorrer el arreglo y realizar comparaciones con

        /*Desarrollo del algoritmo*/

        for(k = 0; k<=n-2; k++) //Empieza el ciclo desde la primera posicion
del arreglo hasta n-2
        {
                p = k; //Le asigna el valor de k a la variable p

                /*Se realiza un ciclo desde la k+1 hasta n-1, esto para
encontrar el valor mas pequeÃ±o
                del arreglo*/
                for(i = k+1; i<=n-1; i++)
                        if(A[i]<A[p])//Si el valor que esta en la posicion i es
menor que el valor en la posicion p
                                p = i;//Se asigna la posicion del elemento mas
pequeÃ±o

                temp = A[p];//Guarda el valor del elemento mas pequeÃ±o en la
variable temp
                //Se hace un intercambio entre el valor pequeÃ±o con el valor
que esta en la posicion K
                A[p] = A[k];
                A[k] = temp;
        }
}
```

Figura 78. Código del algoritmo selección

Shell.c

```
#include<stdio.h>
#include<stdlib.h>

/*
 * Titulo: Practica 1, algoritmo "Ordenamiento por Shell"
 * Descripcion: implementacion del algoritmo Shell
 * Fecha: 13-sep-2021
 * Version: 1
*/

#define TIPO int     /*tipo de datos que se ingresaran*/
#define FORMATO "%d" /*formato de datos que se ingresaran*/
#define POSICION int /*posicion en arreglo*/
#define BANDERA int  /*bandera de estado de algoritmo*/

/*
 * Funcion: Se realiza el ordenamiento de un arreglo por el metodo de
algoritmo Shell
 * Descripcion: Se realiza el ordenamiento del arreglo de longitud "n" con el
algoritmo Shell, el cual consiste en la creacion de sub-arreglos de tamaÃ±o k
= n/2. Se realiza la comparacion de los elementos de cada sub-arreglo, si el
sub arreglo con posicion mas cercano al valor n es menor que el elemento mas
cercano al 0, se realiza un intercambio de valores. Una vez ordenado el sub-
arreglo, se divide k entre 2 nuevamente. El proceso se realiza hasta que k <=
1.
 * Recibe:
 *  - Arreglo de tipo entero
 *  - Valor entero del tamano del arreglo
 * Regresa:
 *  - void
 * Errores: ninguno
*/
void Shell(TIPO *numeros, TIPO n);

int main(int argc, char* argv[])
{
        TIPO n;

    /*Verifica que se ingrese la longitud n de numeros*/
        if (argc!=2)
        {
                printf("\n Se debe de indicar el tamano de algoritmo");
                exit(1);
        }
```


```
        else
        {
        /*convierte la cadena a numero*/
                n=atoi(argv[1]);
        }

        /*Solicita memoria al sistema con la funcion malloc*/
        TIPO *numeros;
        numeros = malloc(n * sizeof(TIPO));

        /*Lee y almacena los n numeros en el arreglo creado*/
        for(int i = 0; i < n; i++)
         scanf(FORMATO,&numeros[i]);

    /*Se realiza llama a la funcion que realiza el algoritmo Shell*/
        Shell(numeros,n);

    /*Se imprime el arreglo ordenado*/
        printf("Arreglo en orden: \n");
    for(int i = 0; i < n; i++)
                printf("%d\n",numeros[i]);

    /*Se libera la memoria*/
    free(numeros);

        return 0;
}

void Shell(TIPO *numeros, TIPO n)
{
    /*Variable para iteraciones*/
        POSICION i;
    /*Variable para dimension de sub-arreglos*/
    POSICION k;
    /*Variable temporal para el intercambio de valores*/
    TIPO temporal;
    /*Variable bandera que indica si el subarreglo realizo un intercambio*/
    BANDERA b;

    /*Se realiza la division del arreglo en sub arreglos de n/2 de longitud*/
        k = (n/2);

    /*Se realiza un ciclo que su ultimo caso es el ordenamiento de arreglos
     de longitud 1*/
        while(k >= 1)
```


```
        {
        /*Se asigna el valor 1 a la bandera para forzar la entrada al while*/
                b = 1;
        /*Se ejecuta el while mientras que b sea diferente de 0
         indicando que el arreglo realizo un intercambio de valores,
         por lo cual requiere revisar si se mantiene ordenado*/
                while(b != 0)
                {
            /*Se asigna el valor 0 a b indicando que no se ha realizado un
intercambio*/
                        b = 0;
            /*Se hace un recorrido desde el valor k al valor n-1 con
incrementos de una unidad*/
                        for(i = k; i <= n - 1; i++)
                        {
                 /*Si compara si el valor del arreglo en la posicion i-k es
mayor que
                  el valor en la posicion i, esto se debe a que los valores en
la posicion i deben ser mayores a los que estan mÃ¡s cerca del 0*/
                                if(numeros[i - k] > numeros[i])
                                {
                    /*Si la condicion se cumple se debe realizar un
intercambio de valores*/
                                        temporal = numeros[i];
                                        numeros[i] = numeros[i - k];
                                        numeros[i - k] = temporal;
                    /*Se incrementa en uno la bandera, indicando que re ha
realizado un intercambio*/
                                        b = b + 1;
                                }
                        }
                }
                /*Al finalizar el analisis de los sub arreglos, se divide
nuevamente entre 2, generando sub-arreglos de menor dimension en cada ciclo*/
                k = (k/2);
        }
}
```

Figura 79. Código del algoritmo Shell

ABB.c

```
#include<stdio.h>
#include<stdlib.h>

/*
 * Titulo: Practica 1, algoritmo "Ordenamiento por ABB"
 * Descripcion: implementacion del algoritmo ABB
 * Fecha: 13-sep-2021
 * Version: 1
*/

#define TIPO int     /*tipo de datos que se ingresaran*/
#define FORMATO "%d" /*formato de datos que se ingresaran*/
#define POSICION int /*posicion en arreglo*/

/*
* Se define la estructura Nodo para la construccion del ABB
* La estructura tiene como elementos:
* - Variable "dato" de tipo int que almacena el valor numerico del nodo
* - Apuntador a estructura Nodo "izq" que indica si el nodo tiene
ramificacion a la izquierda
* - Apuntador a estructura Nodo "der" que indica si el nodo tiene
ramificacion a la derecha
*/

typedef struct Nodo
{
    int dato;
    struct Nodo *izq, *der;
}nodo;

typedef nodo *apnodo; /*Apuntador a estructura nodo*/

/*
 * Funcion: Realiza la insercion de un valor entero a la estructura del Arbol
Binario de Busqueda de manera recursiva
 * Descripcion: Se realiza la insercion de un nodo al ABB, si la raiz actual
es igual a NULL se agrega un nuevo nodo con el valor recibido por la funcion.
Si la raiz no es NULL se compara el valor del nodo raiz con el valor recibido
por la funcion, si el valor recibido es menor o igual que el valor del nodo
raiz, se invoca a la misma misma funcion pasando como parametro un apuntador
al nodo izquierdo de la raiz; en caso contrario se llama a la misma funcion
pasando como parametro un apuntador al nodo derecho de la raiz. El algoritmo
se ejecuta de manera recursiva hasta lograr la insercion.
 * Recibe:
```


```
 *  - Apuntador a apnodo
 *  - Valor entero a insertar
 * Regresa:
 *  - void
 * Errores: ninguno
*/

void Insertar(apnodo *arbol, TIPO dato);

/*
 * Funcion: Realiza un recorrido InOrden del ABB para almacenar los datos
ordenados en un arreglo
 * Descripcion: Realiza de manera recursiva para cada nodo en el ABB los
siguientes pasos:
 *  - Atraviesa el sub-arbol izquierdo
 *  - Almacena el valor del nodo actual y aumenta la variable global k
 *  - Atraviesa el sub-arbol derecho
 * Recibe:
 *  - Apuntador a apnodo
 *  - Arreglo de tipo entero
 * Regresa:
 *  - void
 * Errores: ninguno
*/

void InOrden(apnodo arbol, TIPO *numeros);

/*Contador utilizado para la insercion de valores al arreglo en la funcion
InOrden*/
POSICION k = 0;

int main(int argc, char* argv[])
{
        TIPO n;

        /*Verifica que se ingrese la longitud n de numeros*/
        if (argc!=2)
        {
                printf("\n Se debe de indicar el tamano de n");
                exit(1);
        }
        else
        {
                /*convierte la cadena a numero*/
                n=atoi(argv[1]);
```


```
	}

	/*Solicita memoria al sistema con la funcion malloc*/
	TIPO *numeros;
	numeros = malloc(n * sizeof(TIPO));

	/*Lee y almacena los n numeros en el arreglo creado*/
	for(int i = 0; i < n; i++)
		scanf(FORMATO,&numeros[i]);

	/*Se crea un apuntador de tipo nodo para ser la raiz inicial del ABB*/
	apnodo arbol = NULL;

	/*Se insertan los n numeros en el ABB*/
	for(int i = 0; i < n; i++)
		Insertar(&arbol,numeros[i]);

	/*Se realiza un recorrido In Orden para almacenar los datos en el
arreglo*/
	InOrden(arbol,numeros);

	/*Se imprime el arreglo ordenado*/
	printf("Arreglo en orden: \n");
	for(int i = 0; i < n; i++)
		printf("%d\n",numeros[i]);

	/*Se libera la memoria*/
	free(numeros);
	return 0;
}


void Insertar(apnodo *arbol, TIPO dato)
{
	/*Se verifica si el apuntador a nodo es igual a NULL*/
	if(*arbol == NULL)
	{
		/*Al ser NULL se reserva memoria para la creacion de un nuevo
nodo*/
		*arbol = malloc(sizeof(apnodo));
		/*Se asigna el valor recibido en la funcion como valor del
nodo*/
		(*arbol)->dato = dato;
		/*El apuntador izquierdo y derecho del nodo apuntan a NULL*/
		(*arbol)->izq = NULL;
```


```
                (*arbol)->der = NULL;
        }
        else
        {
                /*Al ser un nodo no NULL se compara si el valor recibido en la
funcion
                es menor o mayor que el valor del nodo actual*/
                if (dato <= (*arbol)->dato)
                {
                /*Si el valor recibido es menor o igual que el valor del nodo
actual
                se invoca a la misma funcion enviando como parametro el
apuntador izquierdo del nodo actual*/
                Insertar(&(*arbol)->izq, dato);
                }
                else
                {
                /*Si el valor recibido es mayor que el valor del nodo actual
                se invoca a la misma funcion enviando como parametro el
apuntador derecho del nodo actual*/
                Insertar(&(*arbol)->der, dato);
                }
        }
}

void InOrden(apnodo arbol, TIPO *numeros)
{
        /*Se verifica si el apuntador a nodo es diferente de NULL*/
        if (arbol != NULL)
        {
                /*Se invoca a la misma funcion pasando como parametro el nodo
izquierdo del nodo actual*/
                InOrden(arbol->izq, numeros);
                /*Se guarda el valor del nodo actual en la posicion k del
arreglo recibido*/
                numeros[k] = arbol->dato;
                /*Se incrementa la variable global k para avanzar la posicion
en el arreglo*/
                 k++;
                /*Se invoca a la misma funcion pasando como parametro el nodo
derecho del nodo actual*/
                InOrden(arbol->der, numeros);
        }
}
```

Figura 80. Código del algoritmo ABB

Mergesort.c

```
#include <stdio.h>
#include <stdlib.h>
/**
 * Titulo: Practica 1, algoritmo "Ordenamiento Mergesort"
 * Descripción: implementacion del algoritmo mergesort
 * Fecha: 13-sep-2021
 * Versión: 1
 * nota: revisar pagina 106 Algorithmics, the spirit of computing
*/
#define TIPO int     /*tipo de datos que se ingresaran*/
#define FORMATO "%d" /*formato de datos que se ingresaran*/
#define POSICION int /*Significado del los nombres primero, medio y final*/
/**
 * Función:realiza el ordenamiento MergeSort
 * Descripcion: toma el arreglo y crea dos particiones de manera recursiva
obteniendo la posicion de en medio. Las particiones son: del inicio a la
posicion de en medio, de la posición de en medio hasta la posicion final.
 * Recibe:
 *  - arreglo de numeros
 *  - la primera posicion
 *  - ultima posicion
 * Regresa:
 *  - void
 * Errores: ninguno
*/
void MergeSort(TIPO numeros[], POSICION primero, POSICION final);
/**
 * Función:realiza la fusion o mezcla de manera ordenada (comparando los
numeros de dos particiones de inicio al medio, medio al final)
 * Descripcion: va comparando el arreglo, tomando en cuenta las particiones,
compara los valores de lado izquierdo con cada valor del lado derecho, hasta
encontrar si es mayor o menor, se va guardando en una variable temporal en
cada iteracion, para despues sobreescribir el arreglo original.
```


```
 * Recibe:
 *  - arreglo de numeros
 *  - la primera posicion
 *  - posicion de en medio
 *  - ultima posicion
 * Regresa:
 *  - void
 * Errores: ninguno
*/
void Merge(TIPO numeros[], POSICION primero, POSICION medio, POSICION final);

/**
 * Función:principal, recibe una longitud del arreglo y diferentes numeros
para crear  un arreglo, para despues llamar a la funcion Mergesort para
ordenar dicho arreglo
 * Descripción: Obtiene el valor de n y los numeros ingresados desde un
archivo de texto, convierte cada cadena en numeros, solicita memoria para la
creacion del arreglo e ir guardando en el los datos ingresados, despues
mandar a llamar a la función Mergesort, al final, con el arreglo ordenado,
imprimir en un archivo.
 * Recibe:
 *  - numero de argumentos
 *  - cadena de caracteres
 * Regresa:
 *  - en caso de error, el nombre del error o constante EXIT_FAILURE
 *  - en caso de exito, regresa la constante EXIT_SUCCESS
 * Errores: ninguno
*/
int main(int argc, char const *argv[])
{
    TIPO n, *numeros;
    int i, j;
    /*Verifica que se ingrese la longitud n de numeros*/
    if (argc != 2)
    {
```


```
        printf("Ingrese n");
        exit(EXIT_FAILURE);
    }
    /*convierte la cadena a numero*/
    n = atoi(argv[1]);

    /*Solicita memoria con la funcion malloc, en caso de fracaso, regresará
error*/
    if ((numeros = malloc(sizeof(TIPO) * n)) == NULL)
        perror("No se pudo solicitar memoria para el arreglo");
    /*lee numero por numero de un archivo*/
    for (i = 0; i < n; i++)
        scanf(FORMATO, &numeros[i]);
    /*se llama a la función*/
    MergeSort(numeros, 0, n - 1);
    /*se imprime el arreglo ordenado*/
    for (j = 0; j < n; j++)
        printf("%d\n", numeros[j]);
    /*Se libera la memoria*/
    free(numeros);
    return EXIT_SUCCESS;
}
void MergeSort(TIPO numeros[], POSICION inicio, POSICION final)
{
    POSICION medio;
    /*compara si la posicion inicial es menor a la final, si no es asi
significa que es el caso base, es decir, un solo numero*/
    if (inicio < final)
    {
        /*se obtiene la posicion media del arreglo*/
        medio = (inicio + final) / 2;
        /*se llama de manera recursiva la funcion Mergesort del punto inicial
al punto medio*/
        MergeSort(numeros, inicio, medio);
```


```
        /*se llama de manera recursiva la funcion Mergesort del punto medio
al punto final*/
        MergeSort(numeros, medio + 1, final);
        /*se fusiona o mezcla ambas particiones, ordenando los numeros*/
        Merge(numeros, inicio, medio, final);
    }
}
void Merge(TIPO numeros[], POSICION inicio, POSICION medio, POSICION final)
{
    /*variables para las iteraciones*/
    POSICION l, i, j, k, f;

    /*rango del arreglo*/
    l = final - inicio + 1;
    i = inicio;
    j = medio + 1;
    /*se solicita memoria para el arreglo temporal con el que se hará las
distintas operaciones, siendo de longitud l ya que es el minimo que se
requiere*/
    TIPO *temporal = malloc(sizeof(TIPO) * l);
    /*se recorre el arreglo tomando en cuenta su rango*/
    for (k = 0; k < l; k++)
    {
        /*Se verifica que esté dentro del rango*/
        if (i <= medio && j <= final)
        {
            /*si el numero de la posicion i es menor al numero de la posicion
que se encuentra de la parte media se guarda en el arreglo temporal y avaza a
la siguiente posicion a partir de la posicion de inicio*/
            if (numeros[i] < numeros[j])
            {
                temporal[k] = numeros[i];
                i++;
            }
```


```
            /*en caso contrario, se guarda el numero de la posicion "a la
derecha" del numero de en medio y avanza a apartir de esa posicion*/
            else
            {
                temporal[k] = numeros[j];
                j++;
            }
        }
        else
        {
            /*en caso que solo se compare con una sola partición, se verifica
que si la particion es de lado izquierdo o derecho*/
            if (i <= medio)
            {
                temporal[k] = numeros[i];
                i++;
            }
            else
            {
                temporal[k] = numeros[j];
                j++;
            }
        }
    }
    /*Copia los elementos del arreglo temporal al arreglo de numeros*/
    for (f = inicio, k = 0; f <= final; f++, k++)
    {
        numeros[f] = temporal[k];
    }
    /*Se libera la memoria*/
    free(temporal);
}
```

Figura 81 Código del algoritmo mergesort

Quicksort.c

```
#include <stdio.h>
#include <stdlib.h>
/**
 * Titulo: Practica 1, algoritmo "Ordenamiento QuickSort"
 * Descripción: implementacion del algoritmo QuickSort
 * Fecha: 13-sep-2021
 * Versión: 1
*/
#define TIPO int     /*tipo de datos que se ingresaran*/
#define FORMATO "%d" /*formato de datos que se ingresaran*/
#define POSICION int /*Significado del los nombres primero, medio y final*/

/**
 * Función:realiza la función de quick sort
 * Descripcion: obteniendo el pivote, y a partir de este, dividir el arreglo
en dos particiones de manera recursiva hasta el caso base (1)
 * Recibe:
 *  - arreglo de numeros
 *  - la primera posicion
 *  - ultima posicion
 * Regresa:
 *  - void
 * Errores: ninguno
*/
void QuickSort(TIPO numeros[], POSICION inicio, POSICION final);
/**
 * Función:realiza la función de obtener el pivote
 * Descripcion: en un ciclo infinito, toma el pivote como la posicion
inicial, a partir de aqui va comparando de izquierda a derecha, y de derecha
a izquierda los numeros mas grandes o pequeños que el numero pivote. Al
recorrer la parte del arreglo, y encuentra que un numero no concuerda con la
posicion que sea mayor o menor, lo intercambia con uno que si sea, es decir,
reacomodar los elementos a ordenar de modo que todas las
```


```
 claves “pequeñas” precedan a las claves “grandes”
 * Recibe:
 *  - arreglo de numeros
 *  - la primera posicion
 *  - ultima posicion
 * Regresa:
 *  - la posición nueva del pivote
 * Errores: ninguno
*/
POSICION Pivot(TIPO numeros[], POSICION inicio, POSICION final);
/**
 * Función:realiza la función intercambiar posiciones
 * Descripcion: algoritmo swap, teniendo variable A y B, A lo guarda en una
variable temporal, B se guarda en la posición donde esta A y despues lo que
se guardó en la variable temporal (A) se guarda donde estaba la variable B
 * Recibe:
 *  - arreglo de numeros
 *  - la primera variable
 *  - la segunda variable
 * Regresa:
 *  - void
 * Errores: ninguno
*/
void Intercambiar(TIPO numeros[], POSICION incio, POSICION final);

int main(int argc, char const *argv[])
{
    TIPO n, *numeros;
    int i, j;
    /*Verifica que se ingrese la longitud n de numeros*/
    if (argc != 2)
    {
        printf("Ingrese n");
        exit(EXIT_FAILURE);
    }
```


```
    /*convierte la cadena a numero*/
    n = atoi(argv[1]);

    /*Solicita memoria con la funcion malloc, en caso de fracaso, regresará
error*/
    if ((numeros = malloc(sizeof(int) * n)) == NULL)
        perror("No se pudo solicitar memoria para el arreglo");
    /*lee numero por numero de un archivo*/
    for (i = 0; i < n; i++)
        scanf(FORMATO, &numeros[i]);
/*se llama a la función*/
    QuickSort(numeros, 0, n-1);
    /*se imprime el arreglo ordenado*/
    for (i = 0; i < n; i++)
        printf("%d\n", numeros[i]);
    /*Se libera la memoria*/
    free(numeros);
    return EXIT_SUCCESS;
}

void QuickSort(TIPO numeros[], POSICION inicio, POSICION final)
{
    POSICION pivote;
    /*compara que el numero este en el rango*/
    if (inicio < final)
    {
        /*obtiene el pivote*/
        pivote = Pivot(numeros, inicio, final);
        /*realiza quicksort del inicio al pivote, sin usar el pivote*/
        QuickSort(numeros, inicio, pivote - 1);
        /*realiza quicksort del pivote a la posicion final, sin usar el
pivote*/
        QuickSort(numeros, pivote + 1, final);
    }
}
```


```
POSICION Pivot(TIPO numeros[], POSICION inicio, POSICION final)
{
    POSICION pivote, i, j,k;
    /*toma el pivote como el primer elemento*/
    pivote = numeros[inicio];
    /*guarda la siguiente posición*/
    i = inicio + 1;
    /*guarda la ultima posición*/
    j = final;

    do{
        /*busca que numero no es menor que el pivote para poder ser
intercambiado*/
        while (numeros[i] <= pivote && i < final)
            i++;
        /*busca que numero no es mayor que el pivote para poder ser
intercambiado*/
        while (numeros[j] > pivote)
            j--;

        /*si la posicipon de la izquierda es menor a la derecha, se
intercambian sus valores*/
        if (i < j)
        {
            Intercambiar(numeros, i, j);
        }
        /*en caso contrario, se intercambia el inicio con el valor de la
derecha y se retorna la posición del nuevo pivote*/
        else
        {
            Intercambiar(numeros, inicio, j);
            return j;
        }
    }while(1);
```


```
}

void Intercambiar(TIPO numeros[], POSICION inicio, POSICION final)
{
    /*se guarda en una variable temporal*/
    TIPO temporal;
    temporal = numeros[final];
    /*realiza el cambio con la otra variable*/
    numeros[final] = numeros[inicio];
    /*la variable guardada en temporal, se coloca en la posicion antigua de
la otra variable*/
    numeros[inicio] = temporal;
}
```

Figura 82 Código del algoritmo Quicksort

Código de medición de tiempos

Para cada uno de los algoritmos se utilizó como plantilla el siguiente código, insertando en los diferentes apartados, las instrucciones o funciones correspondientes

```
//*************************************************************
//M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
//Curso: Análisis de algoritmos
//(C) Enero 2013
//ESCOM-IPN
//Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
//Compilación: "gcc main.c tiempo.x  -o main(teimpo.c si se tiene la
implementación de la libreria o tiempo.o si solo se tiene el codigo objeto)"
//Ejecución: "./main n" (Linux y MAC OS)
//*************************************************************
/**
 * Titulo: Practica 1, algoritmo " "
 * Descripción: implementacion del algoritmo ""
 * Fecha: 13-sep-2021
 * Versión: n
*/
//*************************************************************
//LIBRERIAS INCLUIDAS
//*************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"
//*************************************************************
//DEFINICION DE CONSTANTES DEL PROGRAMA
//*************************************************************
//*************************************************************************
*****
//DECLARACION DE ESTRUCTURAS
//*************************************************************************
*****
```


```
//*************************************************************
//DECLARACIÓN DE FUNCIONES
//*************************************************************
//*************************************************************
//VARIABLES GLOBALES
//*************************************************************
//*************************************************************
//PROGRAMA PRINCIPAL
//*************************************************************
int main(int argc, char *argv[])
{
        //*********************************************************************
        //Variables del main
        //*********************************************************************
        double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables
para medición de tiempos
        TIPO n, *numeros;
        int i, j;

        //*********************************************************************
        //Recepción y decodificación de argumentos
        //*********************************************************************

        //Si no se introducen exactamente 2 argumentos (Cadena de ejecución y
cadena=n)
        if (argc != 2)
        {
                printf("\nIndique el tamanio del algoritmo - Ejemplo:
[user@equipo]$ %s 100\n", argv[0]);
                exit(1);
        }
        //Tomar el segundo argumento como tamaño del algoritmo
        else
        {
                n = atoi(argv[1]);
```


```
        }
        /*Solicita memoria con la funcion malloc, en caso de fracaso,
regresará error*/
        if ((numeros = malloc(sizeof(int) * n)) == NULL)
                perror("No se pudo solicitar memoria para el arreglo");
        /*lee numero por numero de un archivo*/
        for (i = 0; i < n; i++)
                scanf(FORMATO, &numeros[i]);
        //*****************************************************************
        //Iniciar el conteo del tiempo para las evaluaciones de rendimiento
        //*****************************************************************
        uswtime(&utime0, &stime0, &wtime0);
        //*****************************************************************

        //*****************************************************************
        //Algoritmo
        //*****************************************************************
        /*se llama a la función*/
        //*****************************************************************
        //*****************************************************************
        //Evaluar los tiempos de ejecución
        //*****************************************************************
        uswtime(&utime1, &stime1, &wtime1);

        /*Se libera la memoria*/
        free(numeros);

        //Cálculo del tiempo de ejecución del programa
        /*Se imprime el caso de pruba*/
        if(n==100)
                printf("N\tReal\t user\t sys(E/S)\t CPU/wall\n");
    printf("%d\t", n);
        printf("%.10e\t",  wtime1 - wtime0);
        printf("%.10e\t",  utime1 - utime0);
        printf("%.10e\t",  stime1 - stime0);
```


```
        printf("%.10f %%\t\n",100.0 * (utime1 - utime0 + stime1 - stime0) /
(wtime1 - wtime0));
        //*********************************************************************
        //Terminar programa normalmente
        return EXIT_SUCCESS;
}
//*************************************************************************
//DEFINICIÓN DE FUNCIONES
//*************************************************************************
```

Figura 83 Código (plantilla) de medición de tiempos para cada algoritmo

Scripts

Burbuja.sh

```
#!/bin/bash
gcc Burbuja.c -o burbuja
./burbuja 10000000 > resultado.txt < numeros10millones.txt
```

Figura 84. Script para la ejecución del algoritmo burbuja

Burbuja1.sh

```
#!/bin/bash
gcc Burbuja1.c -o burbuja1
./burbuja1 10000000 > resultado.txt < numeros10millones.txt
```

Figura 85. Script para la ejecución del algoritmo burbuja optimización 1

Burbuja2.sh

```
#!/bin/bash
gcc Burbuja.c -o burbuja
./burbuja2 10000000 > resultado.txt < numeros10millones.txt
```

Figura 86. Script para la ejecución del algoritmo burbuja optimización 2

Insercion.sh

```
#!/bin/bash
gcc Insercion.c -o insercion
./insercion 10000000 > resultado.txt < numeros10millones.txt
```

Figura 87. Script para la ejecución del algoritmo insercion

Seleccion.sh

```
#!/bin/bash
gcc Seleccion.c -o seleccion
./seleccion 10000000 > resultado.txt < numeros10millones.txt
```

Figura 88. Script para la ejecución del algoritmo seleccion

Mergesort.sh

```
#!/bin/bash
gcc mergesort.c -o mergesort
./mergesort 10000000 > resultado.txt < numeros10millones.txt
```

Figura 89 Script para la ejecución del algoritmo mergesort

Quicksort.sh

```
#!/bin/bash
gcc quicksort.c -o quicksort
./quicksort 10000 > resultado.txt < numeros10millones.txt
```

Figura 90 Script para la ejecución del algoritmo Quicksort

Compilación y ejecución

Para cada algoritmo

Para compilar y ejecutar cada uno de los algoritmos, escribir en la consola lo siguiente (para el caso de so de Linux)

gcc algoritmo.c -o algoritmo

./algoritmo n > archivo.txt < numeros10millones.txt

Donde “algoritmo” se escribe el algoritmo que se quiera compilar y ejecutar; n el tamaño de problema y “archivo” donde queremos que el resultado se imprima en formato .txt.

O de una manera mas fácil, utilizar scripts que realicen lo explicado anteriormente

./algoritmo.sh

Donde “algoritmo” se escribe el nombre al algoritmo a ejecutar.

Para cada algoritmo (Tiempos)

Para compilar y ejecutar cada uno de los algoritmos, escribir en la consola lo siguiente (para el caso de so de Linux)

gcc algoritmo.c tiempo.c -o algoritmo

./algoritmo n > archivo.txt < numeros10millones.txt

Donde “algoritmo” se escribe el algoritmo que se quiera compilar y ejecutar (en algunos casos se tiene agregado una letra T y en otros Tie); n el tamaño de problema y “archivo” donde queremos que el resultado se imprima en formato .txt.

O de una manera más fácil, utilizar scripts que realicen lo explicado anteriormente

./algoritmo.sh

Donde “algoritmo” se escribe el nombre al algoritmo a ejecutar (en algunos casos se tiene agregado Tie y en otros la palabra Tiempo).