INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

3CM2

ANÁLISIS DE ALGORITMOS

PRUEBAS A POSTERIORI

PRÁCTICA 01

PRESENTA

Guerrero Espinosa Ximena Mariana.
Hernández Espinoza Miguel Angel.
Jiménez Aguilar Tafnes Lorena.



Ciudad de México a 04 de noviembre de 2020

CONTENIDO

TABLAS DE CONTENIDO

GRÁFICAS

TABLAS

ECUACIÓN

1 INTRODUCCIÓN

Una computadora tiene al menos 3 elementos principales a nivel hardware:

- Procesador.
- Memoria.
- Dispositivos de entrada y salida.

Basado en esto, la computadora es capaz de procesar la información gracias al software. Sabemos que los recursos no son ilimitados, por esto en los diferentes programas que generemos, según sea su fin debemos cuidar la memoria que estos emplean, esta se puede cuidar usando menos arreglos, vectores o en su defecto reutilizándolos.

De la misma manera podemos decir que el procesamiento va directamente ligado a la cantidad de instrucciones. Para cuidar y tener en cuenta los puntos anteriores existen los análisis de eficiencia de los algoritmos (memoria y tiempo de ejecución), el cual consta de dos fases:

- Prueba A Posteriori (experimental o empírica)
  Se recogen estadísticas de tiempo y espacio consumidas por el algoritmo al ejecutarse. Consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba, haciendo medidas para: una maquina concreta, un lenguaje concreto, un compilador concreto y/o datos concretos. También depende de las condiciones de prueba.

- Análisis A Priori (o teórico)
  Entrega una función que limita el tiempo de cálculo de un algoritmo. Consiste en obtener una expresión que indique el comportamiento del algoritmo de los parámetros que influyan, Permite la predicción del costo del algoritmo. Es aplicable en la etapa de diseño de los algoritmos.

1.1 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.

1.2 PLANTEAMIENTO DEL PROBLEMA

Con base en el archivo de entrada proporcionado con 10,000,000 número diferentes; ordenarlo bajo los siguientes métodos de ordenamiento y comparar de manera experimental las complejidades de estos.

- Burbuja (Bubble Sort).
- Burbuja Optimizada.
- Inserción (Insertion Sort).
- Selección (Selection Sort).

- Shell (Shell Sort).
- Ordenamiento con árbol binario de búsqueda (Tree Sort).

2 DESARROLLO

Durante este documento se presentarán los datos obtenidos para la realización de esta práctica, todo realizado con el fin de solucionar el planteamiento que se realizó en la sección 1.2 de dicho documento por medio de la implementación de diversos algoritmos y la visualización de su ejecución.

2.1 ESPECIFICACIONES DE HARDWARE

Para ello, se hizo la utilización de un dispositivo (Huawei Matebook D 2020) que cuenta con las siguientes características:

> Procesador: AMD Ryzen 5 3500U (Núcleos CPU: 4, Hilos: 8, Núcleos GPU: 8, Velocidad base de 2.1 GHz, Reloj de impulso máximo arriba de 3.7GHz.
> Memoria: 8GB DDR4.
> Gráficos: Radeon™ Vega 8 Graphics, Radeon™ RX Vega 10 Graphics con frecuencia de 1200 MHz.

Los resultados obtenidos se presentan en las subsecciones de este documento.

2.2 SISTEMA OPERATIVO

Se desarrolló todo sobre Ubuntu 18.04 LTS en el sistema de Windows 10.

2.3 COMPILADOR

Se utilizaron dos compiladores para llevar a cabo dicha práctica:

- Mingw-w64.
- Compilador de Python3.

2.4 MÉTODOS DE ORDENAMIENTO

El ordenamiento de datos consiste en disponer o clasificar un conjunto de datos (o una estructura) en algún determinado orden con respecto a alguno de sus campos. En principio es bastante sencillo, se complica cuando en lugar de querer un ordenamiento eficaz se busca que sea eficiente, es decir, no solo se espera que clasifique los datos de manera adecuada, sino que lo haga con el menor número de comparaciones o usando menor tiempo. Para poder definir si un método es buen existen ciertos criterios de eficiencia:

- El número de pasos.

- El número de comparaciones entre llaves para ordenar n registros.
- El número de movimientos o intercambios de registros que se requieren para ordenar n registros.

2.4.1 BURBUJA (BUBBLE SORT)

Es uno de los más simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro entonces lo intercambia de posición. También se le conoce como el método del intercambio directo. Es necesario revisar varias veces toda la lista hasta que no se necesiten intercambios.

2.4.1.1 PSEUDOCÓDIGO

```
Procedimiento BurbujaSimple(A,n)
   para i=0 hasta n-2 hacer
      para j=0 hasta (n-2)-i hacer
         si (A[j]>A[j+1] entonces
            aux = A[j]
            A[j] = A[j+1]
             A[j+1] = aux
         fin si
      fin para
   fin para
fin Procedimiento
```

2.4.1.2 TIEMPO PARA N=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números, quedando de la siguiente manera:

Tabla 1: Tiempo de ordenamiento de 500,000 valores con Burbuja.

Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
361.6719706479	361.594375	0.0	99.98%

Fuente: Elaboración propia.

Se hará uso de esta información en el análisis de resultados de la sección 3 de este documento.

2.4.1.3 ANÁLISIS TEMPORAL

Se obtuvieron los valores de acuerdo con el tiempo de ejecución que le conlleva al algoritmo de Burbuja (*Bubble Sort)* se obtuvieron los siguientes valores en algunos puntos del rango de 100 – 1,000,000.

Tabla 2: Análisis temporal de n valores con Burbuja

n	Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
100	0.0000209808	0.0	0.0	0%
1,000	0.0012431145	0.0	0.0	0%
5,000	0.0260019302	0.03125	0.0	120.18%
10,000	0.1096129417	0.109375	0.0	99.78%

50,000	3.5267620087	3.53125	0.0	100.13%
100,000	14.5297629833	14.484375	0.0	99.69%
150,000	32.4854290485	32.40625	0.0	99.76%
200,000	56.2142131329	56.1875	0.0	99.95%
250,000	88.2661271095	88.25	0.0	99.98%
350,000	174.8392031193	174.75	0.0	99.61%
400,000	231.3723928928	231.265625	0.0	99.95%
450,000	289.8132240772	289.796875	0.0	99.99%
500,000	361.6719706479	361.594375	0.0	99.98%
550,000	434.2642071247	434.21875	0.0	99.99%
600,000	517.2881011963	517.109375	0.0	99.97%
650,000	607.0177869797	606.9375	0.0	99.99%
750,000	809.3916139603	809.34375	0.0	99.99%
800,000	921.6096391678	921.359375	0.0	99.97%
850,000	1039.8403170109	1,039.734375	0.0	99.99%
950,000	1300.1351728439	1,300.0625	0.0	99.99%
1,000,000	1441.5679521561	1,441.078125	0.0	99.97%

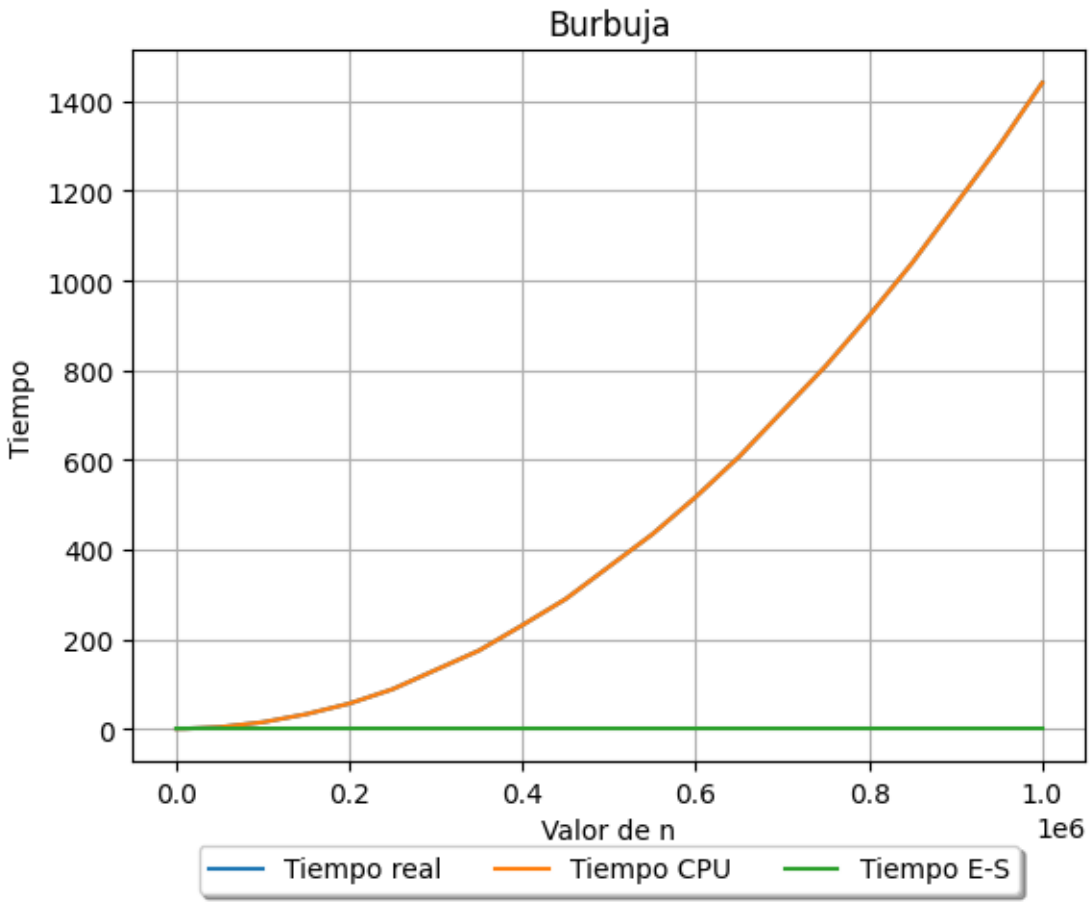
Fuente: Elaboración propia.

La información obtenida en este punto se analizará en la sección 3 de este documento.

2.4.1.3.1 GRÁFICA

Se presentan de manera gráficos el tiempo de ejecución de los n valores para la ejecución del algoritmo.

Gráfica 1: Análisis temporal de n valores con Burbuja



Fuente: Elaboración propia.

2.4.1.4 FUNCIÓN COMPLEJIDAD

Ya que la complejidad teórica de este algoritmo es cuadrática y la gráfica lo evidencia, se optó por aproximar la función complejidad con un polinomio de grado 2:

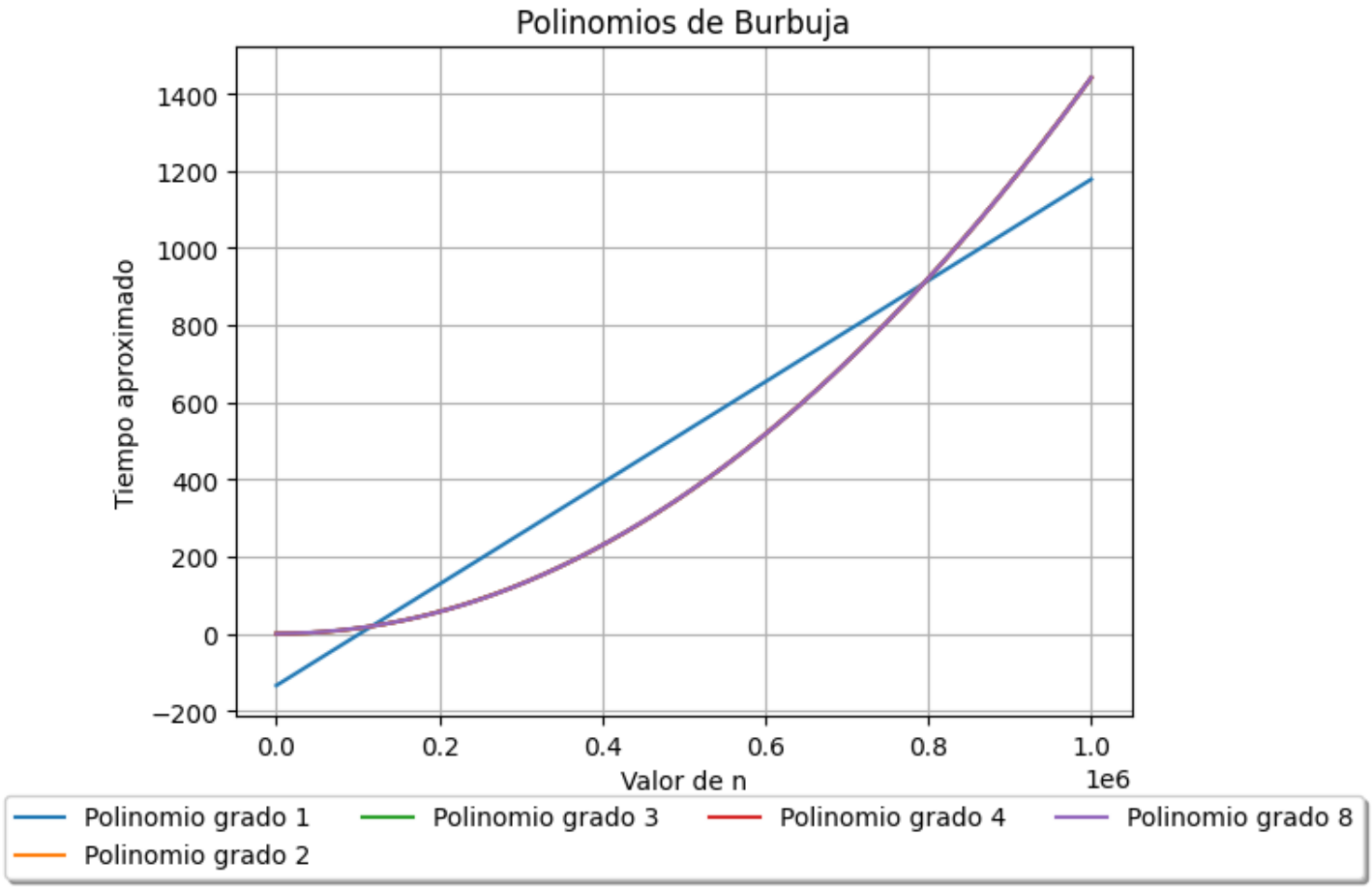
$$f(n) = 1.447 \times 10^{-9}n^2 - 6.377 \times 10^{-6}n + 0.1604$$

Ecuación 1: Polinomio de aproximación de Burbuja.

2.4.1.4.1 GRÁFICA

En la siguiente gráfica se pueden visualizar los distintos grados de los polinomios del algoritmo de ordenamiento.

Gráfica 2: Polinomios de aproximación de Burbuja



Fuente: Elaboración propia.

2.4.1.5 APROXIMACIÓN POLINOMIAL

Con base en la aproximación seleccionada en la sección de *Función de complejidad* se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 3: Aproximación de n valores con Burbuja

n	Tiempo real (s)
15000000	325541.2953777859
20000000	578782.4690035312
500000000	361815466.573208
1000000000	1447268242.7715528
5000000000	36181833604.63663

Fuente: Elaboración propia.

2.4.2 BURBUJA OPTIMIZADA

Como lo dice el nombre es una versión optimizada del método burbuja. En este caso se toma en consideración el hecho de que al final de cada interacción el elemento mayor queda situado en su posición, por lo tanto, ya no es necesario volverlo a comparar con ningún otro número. Esto reduce el número de comparaciones por interacción.

2.4.2.1 PSEUDOCÓDIGO

```
Procedimiento BurbujaOptimizada(A,n)
   cambios = "Si"
   i=0
   Mientras i<n-1 && cambios!= "No" hacer
      cambios= "No"
      Para j=0 hasta (n-2)-i hacer
         Si (A[j] < A[j]) hacer
            aux= A[j]
            A[j] = A[j+1]
            A[j+1] = aux
            cambios = "Si"
         FinSi
      FinPara
      i=i+1
   FinMientras
fin Procedimiento
```

2.4.2.2 TIEMPO PARA N=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números, quedando de la siguiente manera:

Tabla 4: Tiempo de ordenamiento de 500,000 valores con Burbuja Optimizada.

Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
367.3791753901	367.280675	0.0	99.97%

Fuente: Elaboración propia.

Se hará uso de esta información en el análisis de resultados de la sección 3 de este documento.

2.4.2.3 ANÁLISIS TEMPORAL

Se obtuvieron los valores de acuerdo con el tiempo de ejecución que le conlleva al algoritmo de Burbuja optimizada (*Bubble Sort optimizado)* se obtuvieron los siguientes valores en algunos puntos del rango de 100 – 1,000,000.

Tabla 5: Análisis temporal de n valores con Burbuja optimizada

n	Tiempo real (s)	Tiempo CPU	Tiempo E/S	% CPU/Wall
100	0.0000209808	0.0	0.0	0%
1000	0.0014390945	0.0	0.0	0%

5000	0.0257270336	0.03125	0.0	121.47%
10000	0.1146709919	0.109375	0.0	95.38%
50000	3.3677999973	3.375	0.0	100.21%
100000	14.0249929428	14.015625	0.0	99.93%
150000	32.1988790035	32.1875	0.0	99.96%
200000	57.717689991	57.71875	0.0	100.00%
250000	90.5140211582	90.515625	0.0	100.00%
350000	178.3981897831	178.359375	0.0	99.98%
400000	234.0660550594	234.046875	0.0	99.99%
450000	296.2564468384	296.234375	0.0	99.99%
500000	367.3791753901	367.280675	0.0	99.97%
550000	443.8853068352	443.796875	0.015625	99.98%
600000	528.4593679905	528.375	0.0	99.98%
650000	621.3013300896	621.234375	0.0	99.99%
750000	831.4100210667	831.328125	0.0	99.99%
800000	942.0728759766	942.03125	0.0	100.00%
850000	1063.1419029236	1063.09375	0.015625	100.00%
950000	1330.882117033	1330.734375	0.0	99.99%
1000000	1472.7190840244	1472.59375	0.0	99.99%

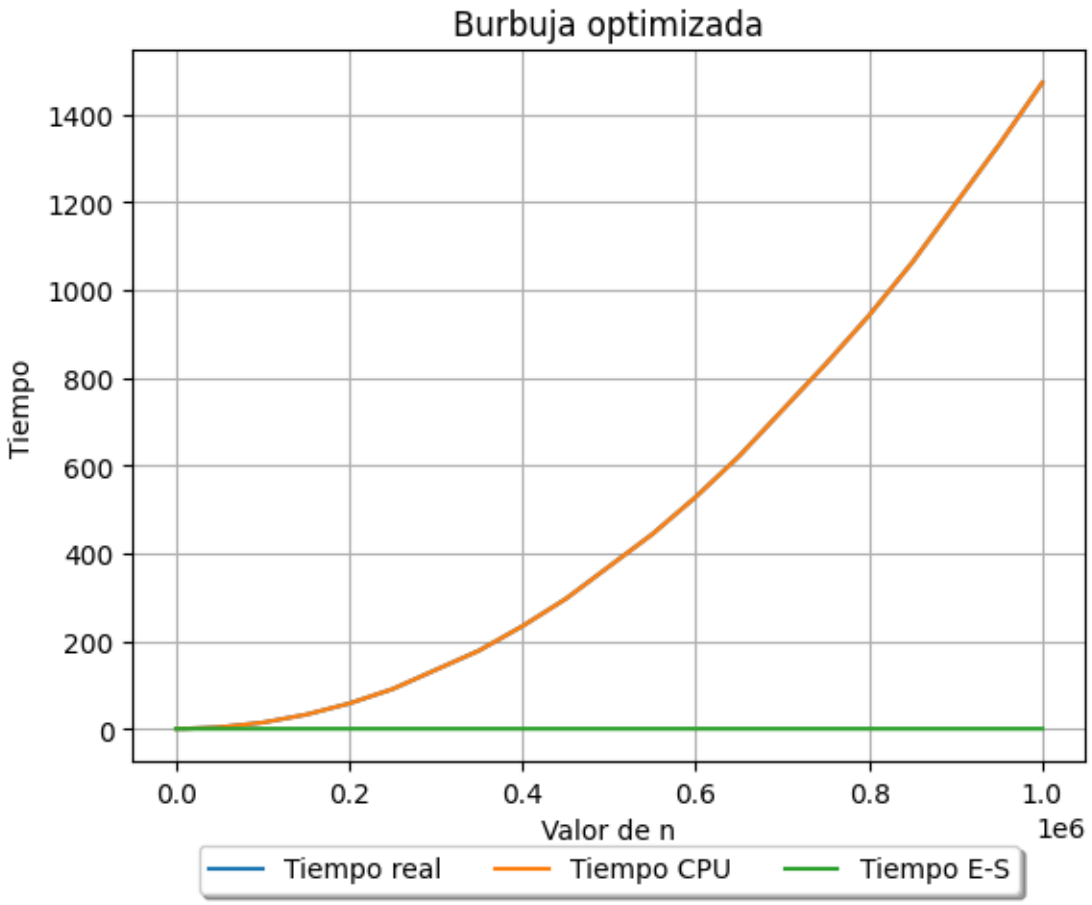
Fuente: Elaboración propia.

La información obtenida en este punto se analizará en la sección 3 de este documento.

2.4.2.3.1 GRÁFICA

Se presentan de manera gráficos el tiempo de ejecución de los n valores para la ejecución del algoritmo.

Gráfica 3: Análisis temporal de n valores con Burbuja optimizada



Fuente: Elaboración propia.

2.4.2.4 FUNCIÓN COMPLEJIDAD

Sucede lo mismo que con la versión sin optimizar, escogemos la aproximación con un polinomio de grado 2:

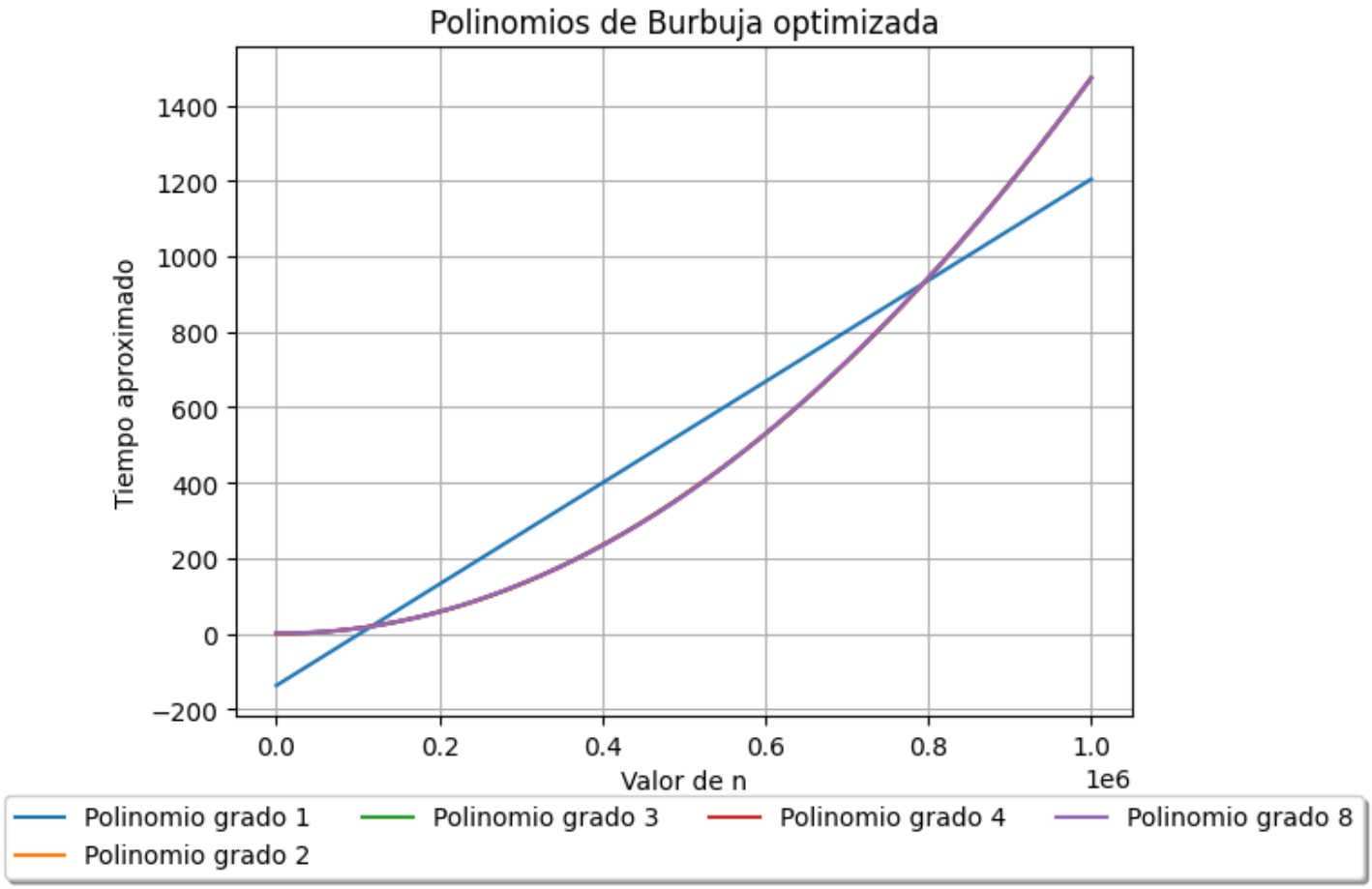
$$f(n) = 1.481 \times 10^{-9}n^2 - 6.637 \times 10^{-6}n - 0.1115$$

Ecuación 2: Polinomio de aproximación de Burbuja optimizada.

2.4.2.4.1 GRÁFICA

En la siguiente gráfica se pueden visualizar los distintos grados de los polinomios del algoritmo de ordenamiento.

Gráfica 4: Polinomios de aproximación de Burbuja optimizada



Fuente: Elaboración propia.

2.4.2.5 APROXIMACIÓN POLINOMIAL

Con base en la aproximación seleccionada en la sección de *Función de complejidad* se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 6: Aproximación de n valores con Burbuja optimizada

n	Tiempo real (s)
15000000	333090.1867745786
20000000	592204.6670932327
500000000	370207633.4753515
1000000000	1480837171.4816217
5000000000	37021062034.6298

Fuente: Elaboración propia.

2.4.3 Inserción (Insertion Sort)

Inicialmente se tiene un solo elemento, el cual representa un conjunto ya ordenado. Posteriormente tenemos $\boldsymbol{k}$ elementos ordenados de menor a mayor, se toma el elemento $\boldsymbol{k+1}$ y se compara con los elementos que ya están ordenados, deteniéndose únicamente cuando se encuentra un elemento menor, para este punto todos los elementos mayores ya han sido desplazados una posición a la derecha o cuando ya no se encuentran elementos, es decir, cuando todos los elementos fueron desplazados y el elemento actual es el más pequeño. En este punto se inserta el elemento $\boldsymbol{k+1}$, desplazando a los demás elementos.

2.4.3.1 Pseudocódigo

```
Procedimiento Insercion(A,n)
   para i=0 hasta n-1 hacer
      j=1
      temp=A[i]
      Mientras(j>0) && (temp<A[j-1] hacer
         A[j]=A[j-1]
         j--
      FinMientras
      A[j]=temp
   FinPara
fin Procedimiento
```

2.4.3.2 Tiempo para n=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números, quedando de la siguiente manera:

Tabla 7: Tiempo de ordenamiento de 500,000 valores con Inserción.

Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
34.5096738515	34.328125	0.0	99.47%

Fuente: Elaboración propia.

Se hará uso de esta información en el análisis de resultados de la sección 3 de este documento.

2.4.3.3 Análisis temporal

Se obtuvieron los valores de acuerdo con el tiempo de ejecución que le conlleva al algoritmo de Inserción (*Insertion Sort)* se obtuvieron los siguientes valores en algunos puntos del rango de 100 – 2,000,000.

Tabla 8: Análisis temporal de n valores con Inserción

n	Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
100	0.0000059605	0.0	0.0	0%
1000	0.0001609325	0.0	0.0	0%
5000	0.0043139458	0.0	0.0	0%
10000	0.0151209831	0.015625	0.0	103.33%
50000	0.3797299862	0.375	0.0	98.75%

100000	1.4296069145	1.4375	0.0	100.55%
150000	3.2089948654	3.21875	0.0	100.30%
200000	5.696518898	5.6875	0.0	99.84%
250000	8.8949859142	8.890625	0.0	99.95%
350000	17.4639129639	17.46875	0.0	100.03%
400000	22.7541179657	22.75	0.0	99.98%
450000	28.7965009212	28.796875	0.0	100.00%
500000	34.5096738515	34.328125	0.0	99.47%
550000	43.3938369751	43.390625	0.0	99.99%
600000	51.6088781357	51.609375	0.0	100.00%
650000	60.2633161545	60.265625	0.0	100.00%
750000	80.4808869362	80.421875	0.0	99.93%
800000	91.6957380772	91.578125	0.0	99.87%
850000	103.3698170185	103.359375	0.0	99.99%
950000	128.6030809879	128.609375	0.0	100.00%
1000000	142.8965969086	142.890625	0.0	100.00%
2000000	602.5115787983	602.421875	0.0	99.99%

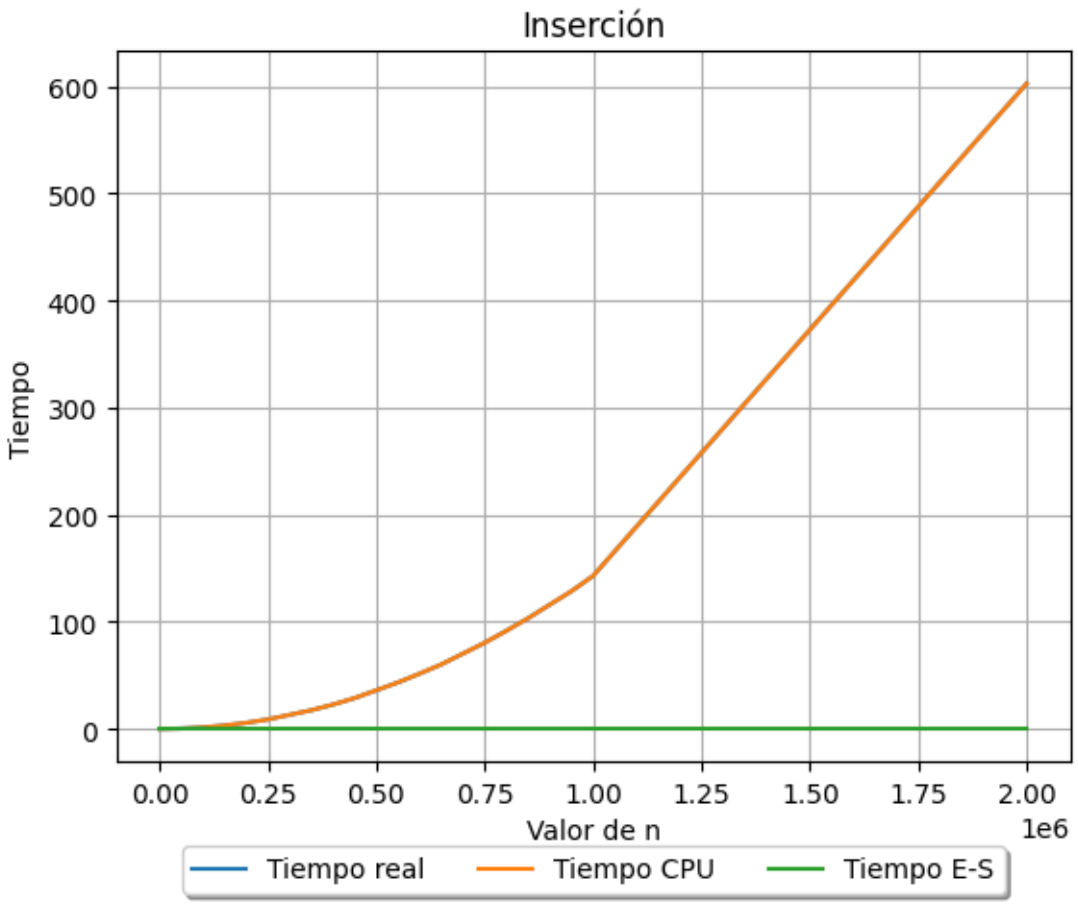
Fuente: Elaboración propia.

La información obtenida en este punto se analizará en la sección 3 de este documento.

2.4.3.3.1 GRÁFICA

Se presentan de manera gráficos el tiempo de ejecución de los n valores para la ejecución del algoritmo.

Gráfica 5: Análisis temporal de n valores con Inserción



Fuente: Elaboración propia.

2.4.3.4 FUNCIÓN COMPLEJIDAD

La complejidad teórica de este algoritmo también es cuadrática, por lo tanto, escogemos un polinomio de grado 2 para aproximarla:

$$f(n) = 1.556 \times 10^{-10} n^2 - 1.097 \times 10^{-5} n + 0.9685$$

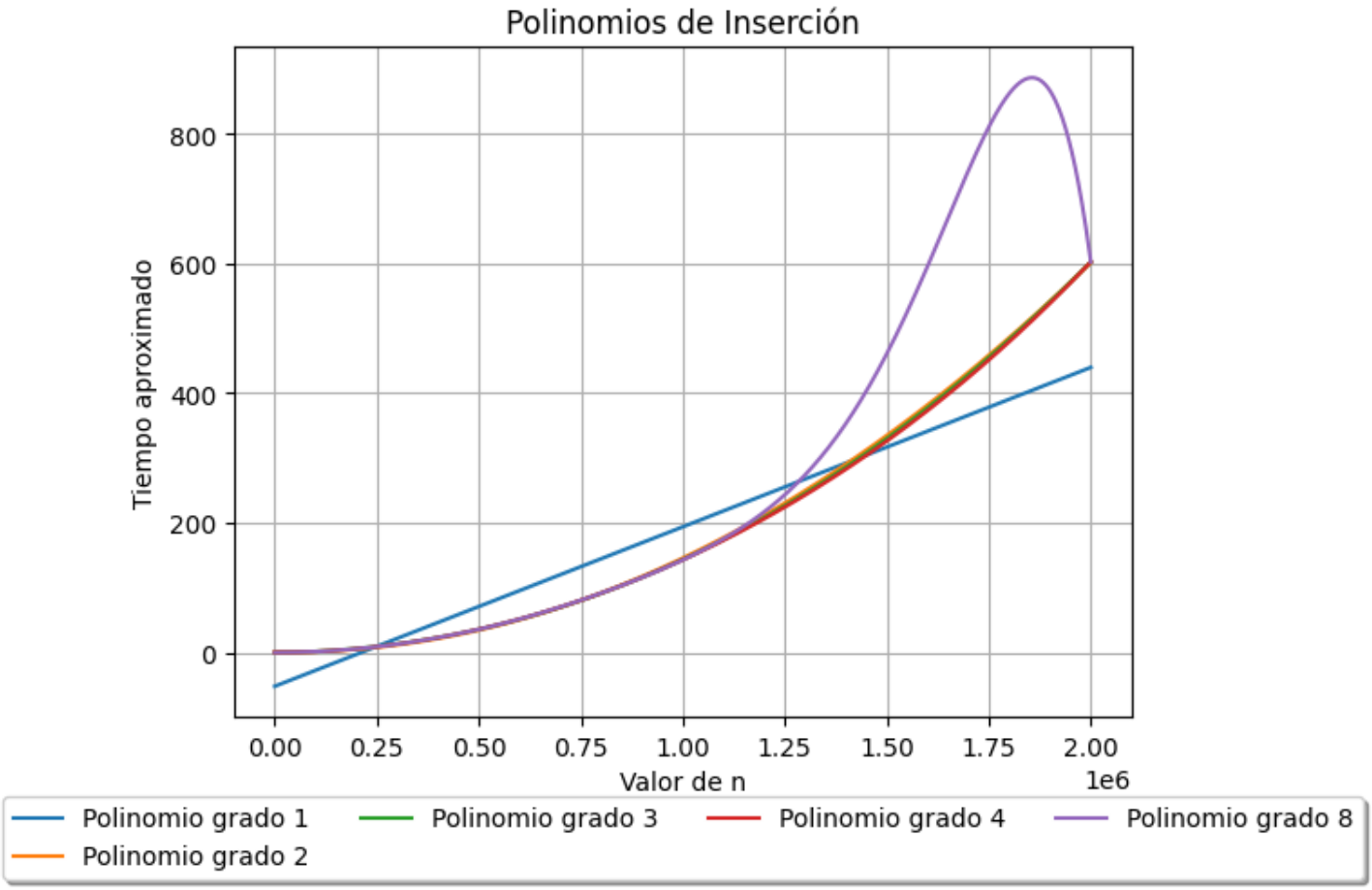
Ecuación 3: Polinomio de aproximación de Inserción.

En la gráfica pareciera que de 1M a 2M la función de complejidad se vuelve lineal, pero eso no es cierto, pues no hay valores intermedios entre esos dos casos. Si los hubiera, la gráfica se hubiera visto más "suave", tal y como lo observamos en la siguiente gráfica de las aproximaciones.

2.4.3.4.1 GRÁFICA

En la siguiente gráfica se pueden visualizar los distintos grados de los polinomios del algoritmo de ordenamiento.

Gráfica 6: Polinomios de aproximación de Inserción



Fuente: Elaboración propia.

2.4.3.5 APROXIMACIÓN POLINOMIAL

Con base en la aproximación seleccionada en la sección de *Función de complejidad* se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 9: Aproximación de n valores con Inserción

n	Tiempo real (s)
15000000	34857.07722862971
20000000	62040.52808777987
500000000	38906385.06771858
1000000000	155636508.9803695
5000000000	3891132133.5665545

Fuente: Elaboración propia.

2.4.4 Selección (Selection Sort)

Se basa en la simple acción de buscar el mínimo elemento de la lista e intercambiamos con el primero, después busca el siguiente mínimo en el resto de la lista, lo intercambia con el segundo y así sucesivamente hasta conseguir que todos estén ordenados.

2.4.4.1 Pseudocódigo

```
Procedimiento Seleccion(A,n)
   para k=0 hasta n-2 hacer
      p=k
      para i=k+1 hasta n-1 hacer
         si A[i]>A[p] entonces
            p=i
         FinSi
      FinPara
      temp= A[p]
      A[p] = A[k]
      A[k] = temp
   FinPara
fin Procedimiento
```

2.4.4.2 Tiempo para n=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números, quedando de la siguiente manera:

Tabla 10: Tiempo de ordenamiento de 500,000 valores con Selección.

Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
65.0992487240	65.09125	0.0	99.99%

Fuente: Elaboración propia.

Se hará uso de esta información en el análisis de resultados de la sección 3 de este documento.

2.4.4.2.1 Análisis temporal

Se obtuvieron los valores de acuerdo con el tiempo de ejecución que le conlleva al algoritmo de Selección (*Selection Sort)* se obtuvieron los siguientes valores en algunos puntos del rango de 100 – 2,000,000.

Tabla 11: Análisis temporal de n valores con Selección

Algoritmo	Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
100	0.0000100136	0.0	0.0	0%
1000	0.0005028248	0.0	0.0	0%
5000	0.0082349777	0.015625	0.0	189.74%
10000	0.0313720703	0.03125	0.0	99.61%
50000	0.7255239487	0.71875	0.0	99.07%
100000	2.8370079994	2.84375	0.0	100.24%

150000	6.1971230507	6.1875	0.0	99.84%
200000	11.0459430218	11.046875	0.0	100.01%
250000	17.1855490208	17.171875	0.0	99.92%
350000	33.7272980213	33.734375	0.0	100.02%
400000	44.0444178581	44.03125	0.0	99.97%
450000	55.8070011139	55.8125	0.0	100.01%
500000	65.0992487240	65.09125	0.0	99.99%
550000	82.7965221405	82.78125	0.0	99.98%
600000	99.1163930893	99.109375	0.0	99.99%
650000	115.9292440414	115.9375	0.0	100.01%
750000	154.5529050827	154.546875	0.0	100.00%
800000	175.4107708931	175.40625	0.0	100.00%
850000	198.290088892	198.265625	0.0	99.99%
950000	248.526859045	248.515625	0.0	100.00%
1000000	275.4536950588	275.4375	0.0	99.99%
2000000	1204.4664349556	1204.328125	0.0	99.99%

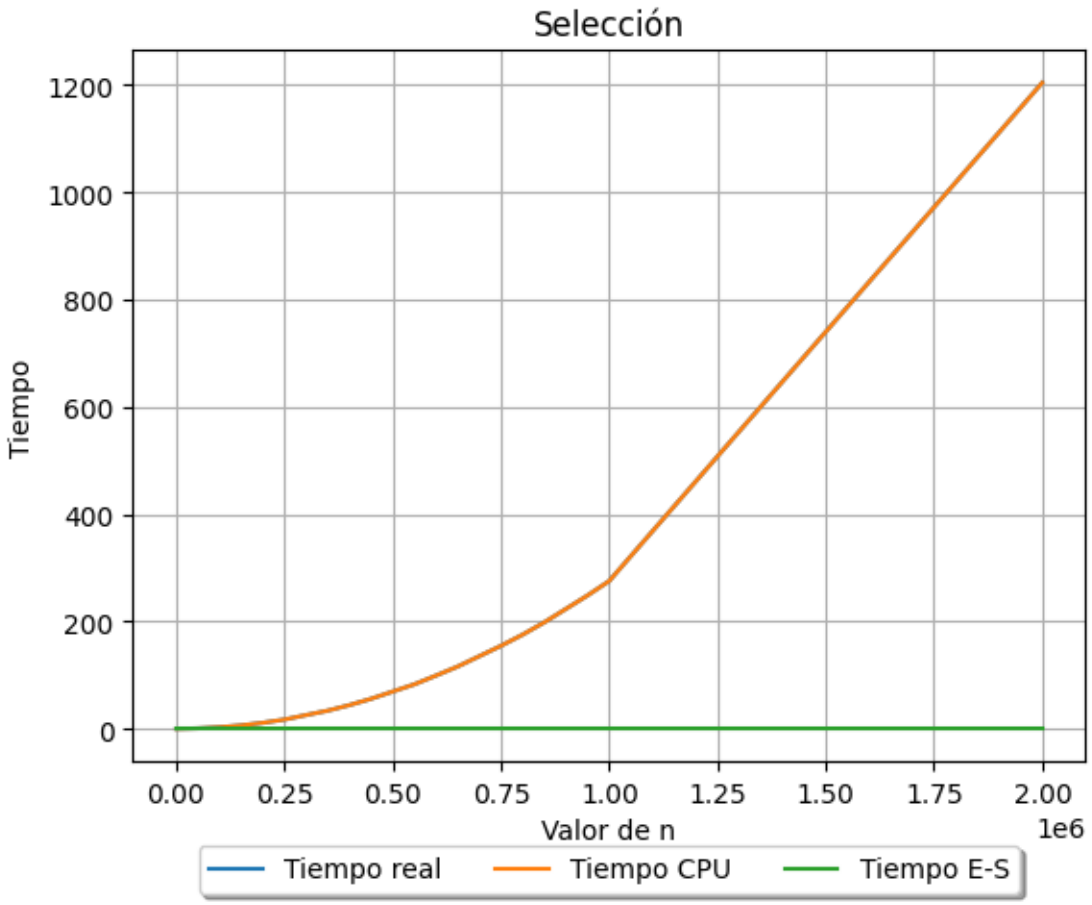
Fuente: Elaboración propia.

La información obtenida en este punto se analizará en la sección 3 de este documento.

2.4.4.2.2 GRÁFICA

Se presentan de manera gráficos el tiempo de ejecución de los n valores para la ejecución del algoritmo.

Gráfica 7: Análisis temporal de n valores con Selección



Fuente: Elaboración propia.

2.4.4.3 FUNCIÓN COMPLEJIDAD

Para este algoritmo ocurre lo mismo, su complejidad es cuadrática, por lo que nos quedamos con la aproximación de grado 2. Además, notemos que aproximaciones de grados mayores tienden a ser inestables con valores más grandes.

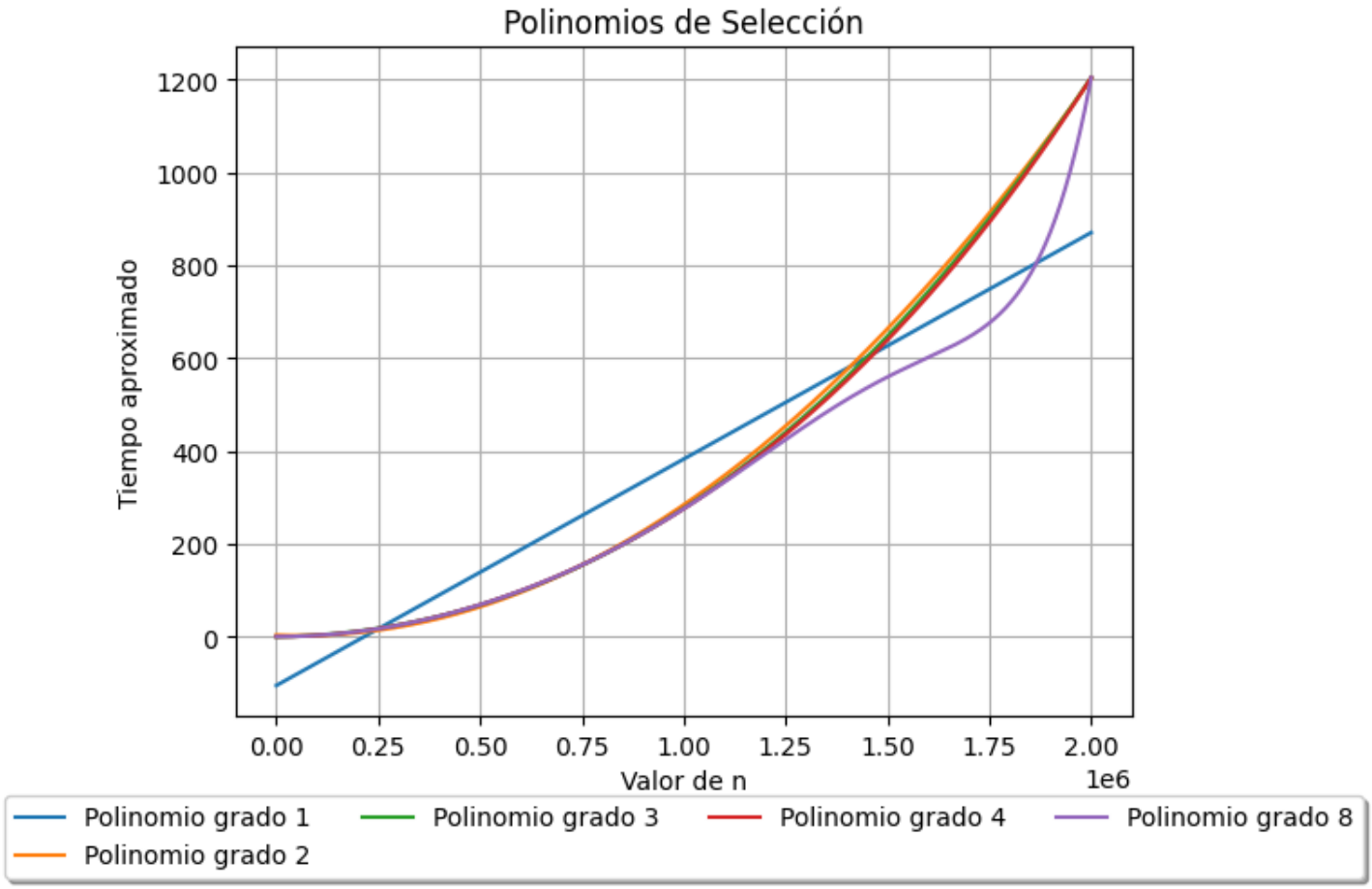
$$f(n) = 3.182 \times 10^{-10} n^2 - 3.738 \times 10^{-5} n + 3.324$$

Ecuación 4: Polinomio de aproximación de Selección.

2.4.4.3.1 GRÁFICA

En la siguiente gráfica se pueden visualizar los distintos grados de los polinomios del algoritmo de ordenamiento.

Gráfica 8: Polinomios de aproximación de Selección



Fuente: Elaboración propia.

2.4.4.4 APROXIMACIÓN POLINOMIAL

Con base en la aproximación seleccionada en la sección de *Función de complejidad* se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 12: Aproximación de n valores con Selección

n	Tiempo real (s)
15000000	71047.19341739401
20000000	126552.73560375615
500000000	79541944.52665336
1000000000	318205148.05971766
5000000000	7955876220.228267

Fuente: Elaboración propia.

2.4.5 SHELL (SHELL SORT)

Este método mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga movimientos más grandes hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños cada vez más pequeños. El último paso del ordenamiento Shell es un simple ordenamiento por inserción, sabemos que este último es eficiente si la entrada esta casi ordenada, por esto está garantizado que para este punto ser eficiente.
Propone que se haga sobre el arreglo una serie de ordenaciones basadas en la inserción directa, pero dividiendo el arreglo original en varios sub-arreglos tales que cada elemento este separado k elementos del anterior. Se debe empezar con $\boldsymbol{k = n/2}$, se hará más pequeño sucesivamente hasta llegar a $\boldsymbol{k = 1}$.

2.4.5.1 PSEUDOCÓDIGO

```
1  Procedimiento Shell(A,n)
2     k=TRUNC(n/2)
3        Mientras k>=1 hacer
4           b=1
5              Mientras b!=0 hacer
6                 b=0
7                 Para i=k hasta i>=n-1 hacer
8                    Si A[i-k]>A[i]
9                       temp = A[i]
10                      A[i]=A[i-k]
11                      A[i-k]=temp
12                      b=b+1
13                   FinSi
14                FinPara
15             FinMientras
16             k=TRUNC(k/2)
17       FinMientras
18 fin Procedimiento
```

2.4.5.2 TIEMPO PARA N=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números, quedando de la siguiente manera:

Tabla 13: Tiempo de ordenamiento de 500,000 valores con Shell.

Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
0.251294343	0.250625	0.0	99.73%

Fuente: Elaboración propia.

Se hará uso de esta información en el análisis de resultados de la sección 3 de este documento.

2.4.5.3 ANÁLISIS TEMPORAL

Se obtuvieron los valores de acuerdo con el tiempo de ejecución que le conlleva al algoritmo de Shell (*Shell Sort)* se obtuvieron los siguientes valores en algunos puntos del rango de 100 – 10,000,000.

Tabla 14: Análisis temporal de n valores con Shell

Algoritmo	Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
100	0.0000071526	0	0.0	0%
1000	0.000175953	0	0.0	0%
5000	0.001497984	0.015625	0.0	1043.07%
10000	0.002776146	0.015625	0.0	562.83%
50000	0.016304016	0.015625	0.0	95.84%
100000	0.040283918	0.046875	0.0	116.36%
150000	0.054943085	0.0625	0.0	113.75%
200000	0.094957113	0.09375	0.0	98.73%
250000	0.112835169	0.109375	0.0	96.93%
350000	0.164132118	0.15625	0.0	95.20%
400000	0.208369017	0.21875	0.0	104.98%
450000	0.210871935	0.21875	0.0	103.74%
500000	0.251294343	0.250625	0.0	99.73%
550000	0.277137041	0.265625	0.0	95.85%
600000	0.304375887	0.296875	0.0	97.54%
650000	0.330961943	0.328125	0.0	99.14%
750000	0.37919116	0.375	0.0	98.89%
800000	0.498213053	0.484375	0.0	97.22%
850000	0.486995935	0.484375	0.0	99.46%
950000	0.5029881	0.515625	0.0	102.51%
1000000	0.580879927	0.578125	0.0	99.53%
2000000	1.499490023	1.5	0.0	100.03%
3000000	2.239770889	2.234375	0.0	99.76%
4000000	3.49049592	3.484375	0.0	99.82%
5000000	4.627238989	4.625	0.0	99.95%
6000000	5.4028759	5.390625	0.0	99.77%
7000000	6.452430964	6.453125	0.0	100.01%
8000000	8.829813004	8.828125	0.0	99.98%
9000000	8.883088112	8.875	0.0	99.91%
10000000	10.7675209	10.765625	0.0	99.98%

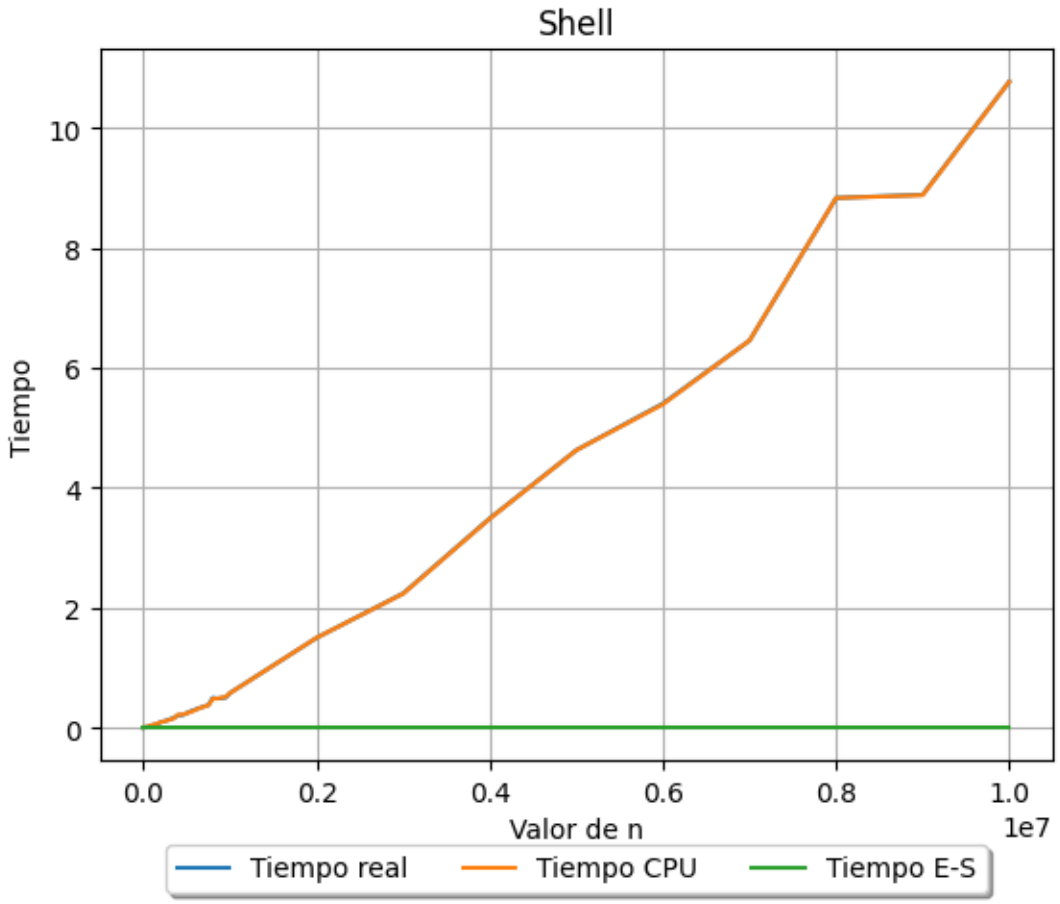
Fuente: Elaboración propia.

La información obtenida en este punto se analizará en la sección 3 de este documento.

2.4.5.3.1 GRÁFICA

Se presentan de manera gráficos el tiempo de ejecución de los n valores para la ejecución del algoritmo.

Gráfica 9: Análisis temporal de n valores con Shell



Fuente: Elaboración propia.

2.4.5.4 FUNCIÓN COMPLEJIDAD

La complejidad teórica de este algoritmo también es cuadrática, pero se optó por usar una aproximación lineal únicamente, pues para estos valores específicos del arreglo, la gráfica tiende a ser una línea.

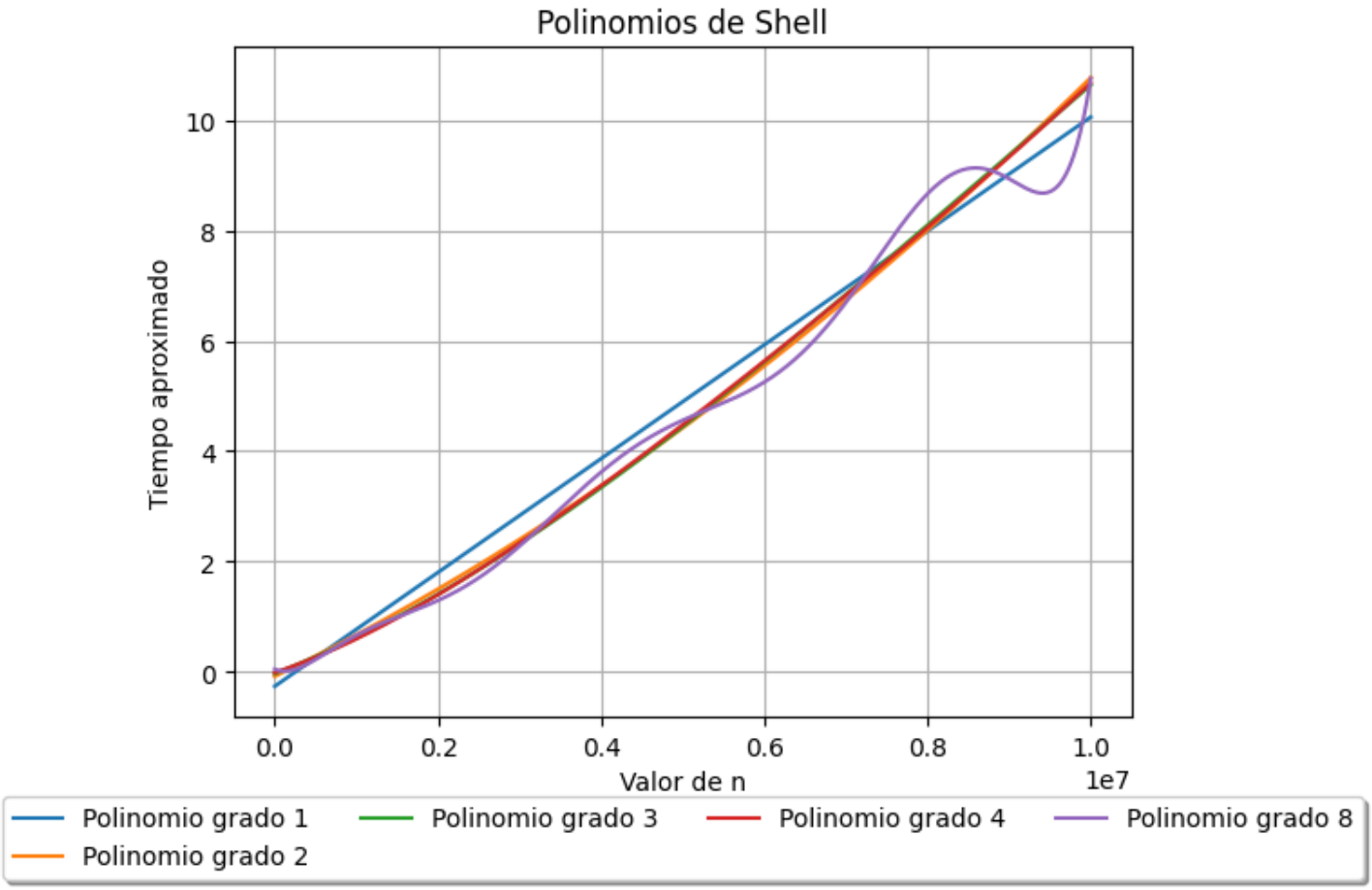
$$f(n) = 1.032 \times 10^{-6}n - 0.2639$$

Ecuación 5: Polinomio de aproximación de Shell.

2.4.5.4.1 GRÁFICA

En la siguiente gráfica se pueden visualizar los distintos grados de los polinomios del algoritmo de ordenamiento.

Gráfica 10: Polinomios de aproximación de Shell



Fuente: Elaboración propia.

2.4.5.5 APROXIMACIÓN POLINOMIAL

Con base en la aproximación seleccionada en la sección de *Función de complejidad* se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 15: Aproximación de n valores con Shell

n	Tiempo real (s)
15000000	15.2181978
20000000	20.37888072
500000000	515.8044412
1000000000	1031.872733
5000000000	5160.419071

Fuente: Elaboración propia.

2.4.6 Ordenamiento con árbol binario de búsqueda (Tree Sort)

Este resulta muy simple debido a que solo requiere de dos pasos simples:

- Insertar cada uno de los numero del vector a ordenar en el árbol binario de búsqueda.
- Reemplazar el vector en desorden por el vector resultante de un recorrido InOrden del ABB el cual entregara los números ordenados.

La eficiencia de este algoritmo está dada según la eficiencia en la implementación del árbol binario de búsqueda, lo que puede resultar mejor que otros algoritmos de ordenamiento.

2.4.6.1 Pseudocódigo

```
1 Procedimiento OrdenaConArbolBinarioBusqueda (A,n)
2    Para i=0 hasta i>=n hacer
3       Insertar(ArbolBinBusqueda,A[i]);
4    FinPara
5    GuardarRecorridoInOrden(ArbolBinBusqueda,A);
6 fin Procedimiento
```

2.4.6.2 Tiempo para n=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números, quedando de la siguiente manera:

Tabla 16: Tiempo de ordenamiento de 500,000 valores con ABB.

Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
0.276817095	0.27675	0	99.98%

Fuente: Elaboración propia.

Se hará uso de esta información en el análisis de resultados de la sección 3 de este documento.

2.4.6.3 Análisis temporal

Se obtuvieron los valores de acuerdo con el tiempo de ejecución que le conlleva al algoritmo de Árbol binario de búsqueda (*Tree Sort)* se obtuvieron los siguientes valores en algunos puntos del rango de 100 – 10,000,000.

Tabla 17: Análisis temporal de n valores con Árbol

Algoritmo	Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
100	0.0000181198	0	0	0%
1000	0.000124931	0	0	0%
5000	0.00075388	0	0	0%
10000	0.001408815	0	0	0%
50000	0.008979082	0	0	0%
100000	0.021374941	0.015625	0	73.10%
150000	0.037196875	0.03125	0	84.01%

200000	0.073771954	0.078125	0	105.90%
250000	0.085552931	0.0625	0.015625	91.32%
350000	0.148402929	0.140625	0	94.76%
400000	0.181099892	0.15625	0.03125	103.53%
450000	0.218028069	0.21875	0	100.33%
500000	0.276817095	0.27675	0	99.98%
550000	0.300177813	0.25	0.0625	104.10%
600000	0.338112116	0.3125	0.03125	101.67%
650000	0.384364843	0.359375	0.015625	97.56%
750000	0.474510193	0.46875	0.03125	105.37%
800000	0.521678925	0.5	0.015625	98.84%
850000	0.570526123	0.578125	0	101.33%
950000	0.675677061	0.65625	0.03125	101.75%
1000000	0.720030069	0.671875	0.046875	99.82%
2000000	1.902303934	1.84375	0.0625	100.21
3000000	3.237220049	3.140625	0.109375	100.39%
4000000	4.705645084	4.59375	0.09375	99.61%
5000000	6.241189957	6.015625	0.21875	99.89%
6000000	7.878377914	7.609375	0.28125	100.16%
7000000	9.544706821	9.21875	0.3125	99.86%
8000000	11.2976079	11.015625	0.28125	99.99%
9000000	13.16247678	12.859375	0.3125	100.07%
10000000	14.9785769	14.546875	0.421875	99.93%

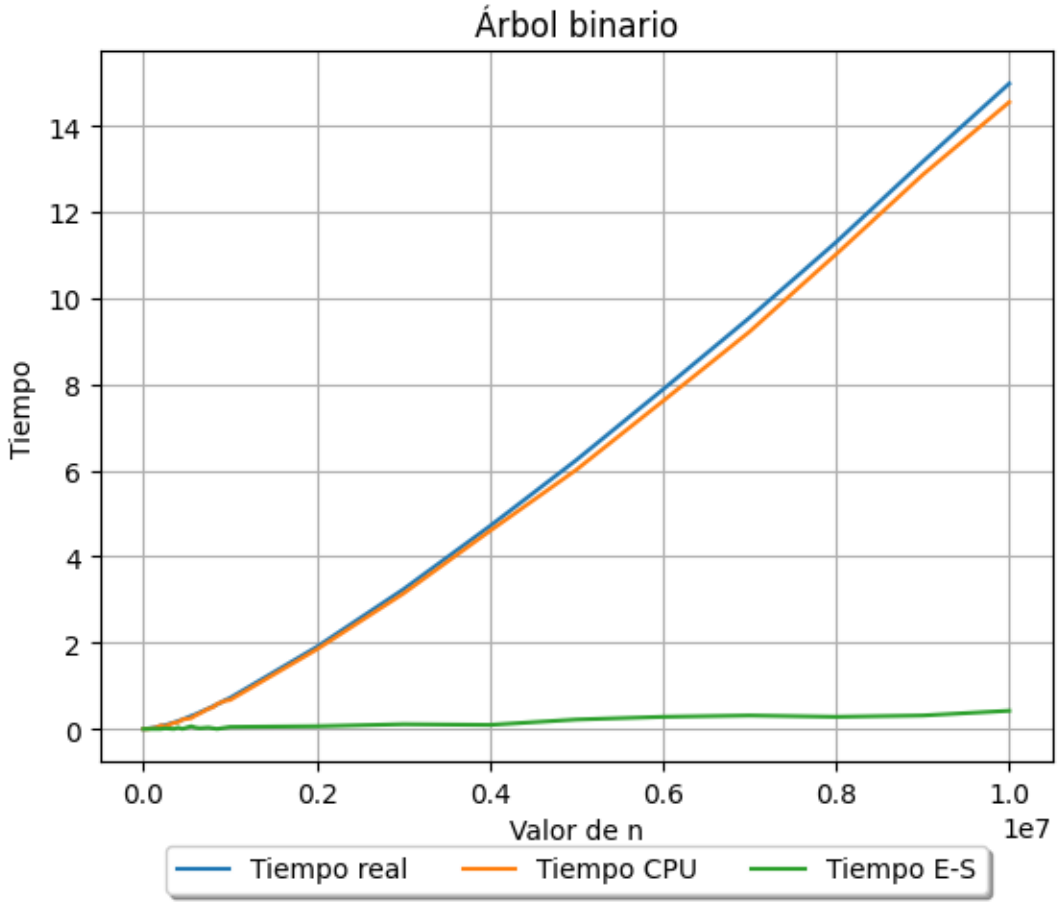
Fuente: Elaboración propia.

La información obtenida en este punto se analizará en la sección 3 de este documento.

2.4.6.3.1 GRÁFICA

Se presentan de manera gráficos el tiempo de ejecución de los n valores para la ejecución del algoritmo.

Gráfica 11: Análisis temporal de n valores con Árbol ABB



Fuente: Elaboración propia.

2.4.6.4 FUNCIÓN COMPLEJIDAD

La complejidad teórica de este algoritmo es cuadrática en el peor caso cuando el arreglo ya está ordenado, pero en el caso promedio es $O(n \log n)$. La gráfica experimental evidencia eso, pues se asemeja a una línea, pero con un ligero aumento de pendiente conforme crece la entrada. Por tal motivo, escogimos una aproximación de grado 1:

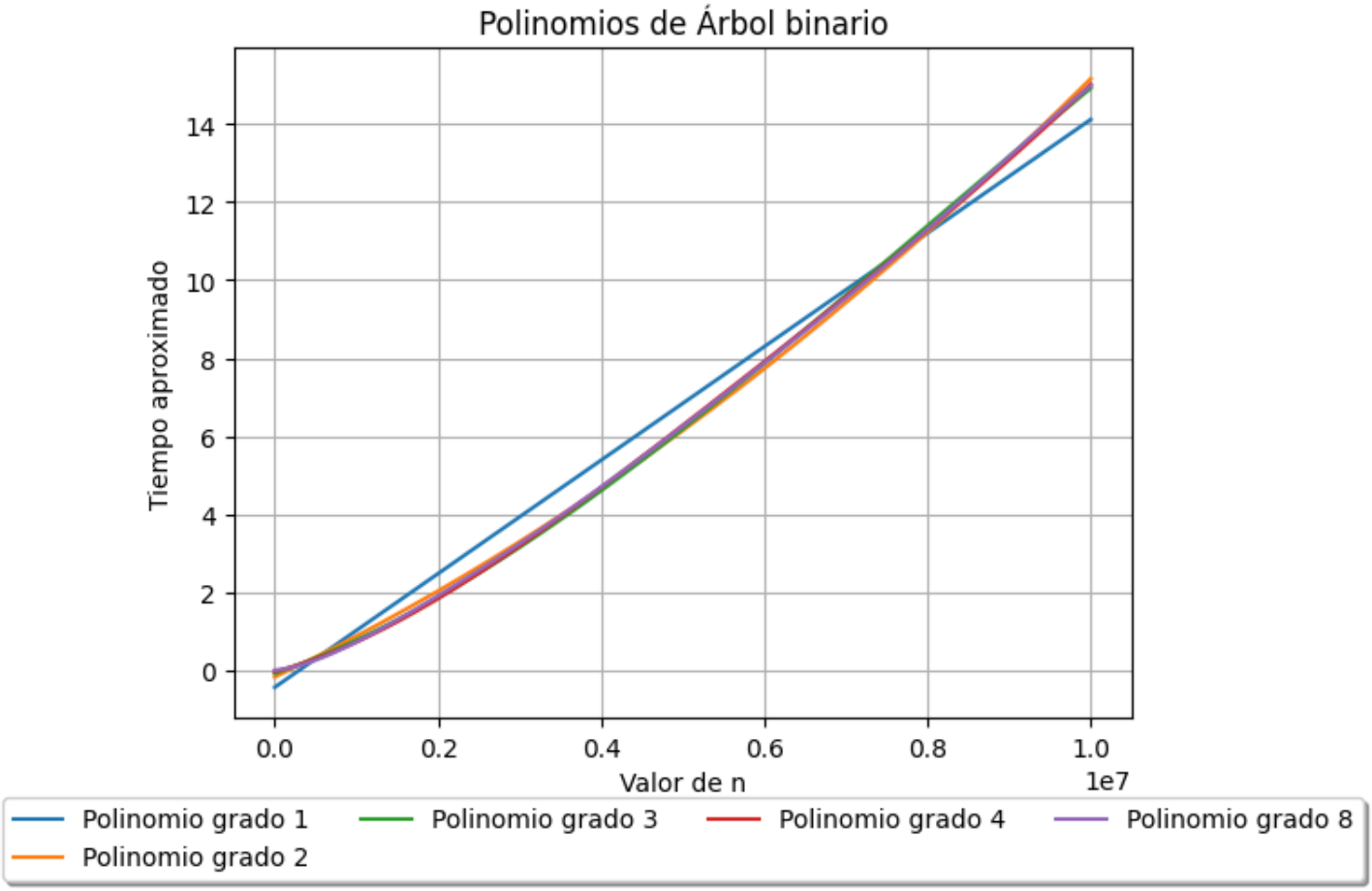
$$f(n) = 1.455 \times 10^{-6} n - 0.4379$$

Ecuación 6: Polinomio de aproximación de Árbol ABB.

2.4.6.4.1 GRÁFICA

En la siguiente gráfica se pueden visualizar los distintos grados de los polinomios del algoritmo de ordenamiento.

Gráfica 12: Polinomios de aproximación de Árbol ABB



Fuente: Elaboración propia.

2.4.6.5 APROXIMACIÓN POLINOMIAL

Con base en la aproximación seleccionada en la sección de *Función de complejidad* se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 18: Aproximación de n valores con Árbol

n	Tiempo real (s)
15000000	21.39436724
20000000	28.67178999
500000000	727.3043737
1000000000	1455.046648
5000000000	7276.984846

Fuente: Elaboración propia.

3 RESULTADOS

Se presentan los resultados obtenidos de los datos obtenidos a lo largo del desarrollo de la práctica a través de diversas gráficas y tablas. Todo esto se presenta a continuación.

3.1 TABLA COMPARATIVA PARA N=500000

Se midió el tiempo que tardó el algoritmo en ordenar 500,000 números con los diversos algoritmos, quedando de la siguiente manera:

Tabla 19: Tiempo de ordenamiento de 500,000 valores.

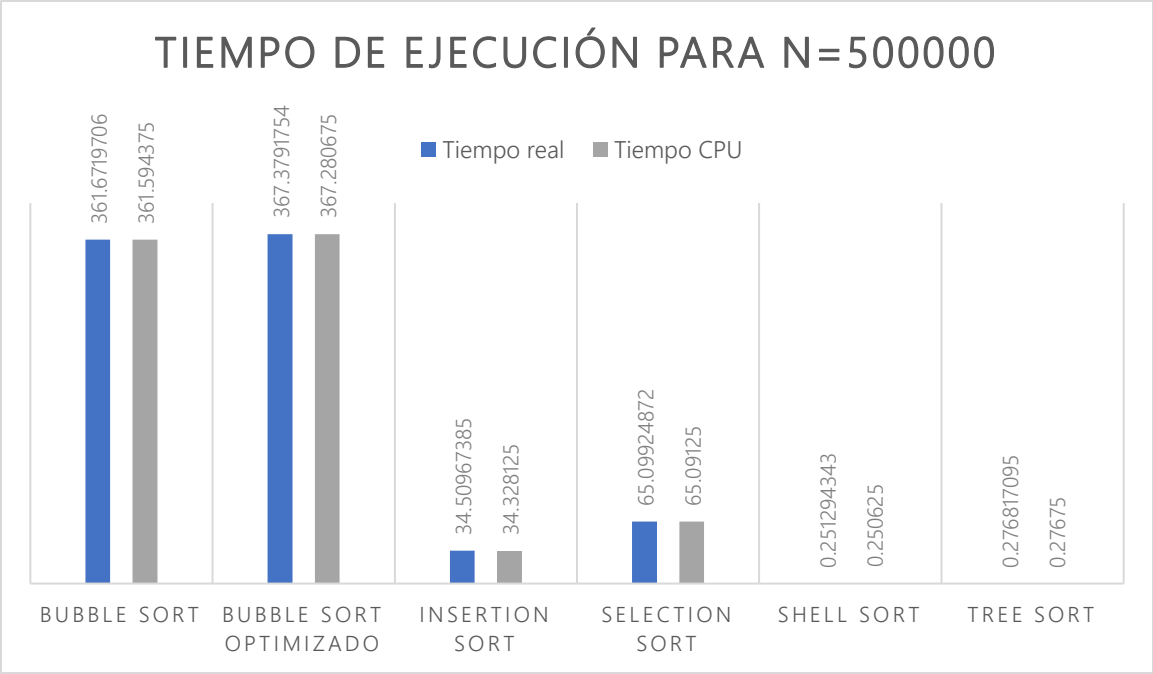
Algoritmo	Tiempo real (s)	Tiempo CPU (s)	Tiempo E/S (s)	% CPU/Wall
Burbuja (Bubble Sort)	361.6719706479	361.594375	0	99.98%
Burbuja Optimizada	367.3791753901	367.280675	0	99.97%
Inserción (Insertion Sort)	34.5096738515	34.328125	0	99.47%
Selección (Selection Sort)	65.0992487240	65.09125	0	99.99%
Shell (Shell Sort)	0.251294343	0.250625	0	99.73%
Árbol (Tree Sort)	0.276817095	0.27675	0	99.98%

Fuente: Elaboración propia.

3.2 GRÁFICA COMPARATIVA PARA N=500000

Con los datos presentados en la sección anterior, se realizó la siguiente gráfica con el fin de visualizar la información de mejor manera y observar el tiempo en el que se ejecutan los diversos algoritmos, viendo cuáles son más eficaces y eficientes.

Gráfica 13: Tiempo de ejecución para 500,000 valores



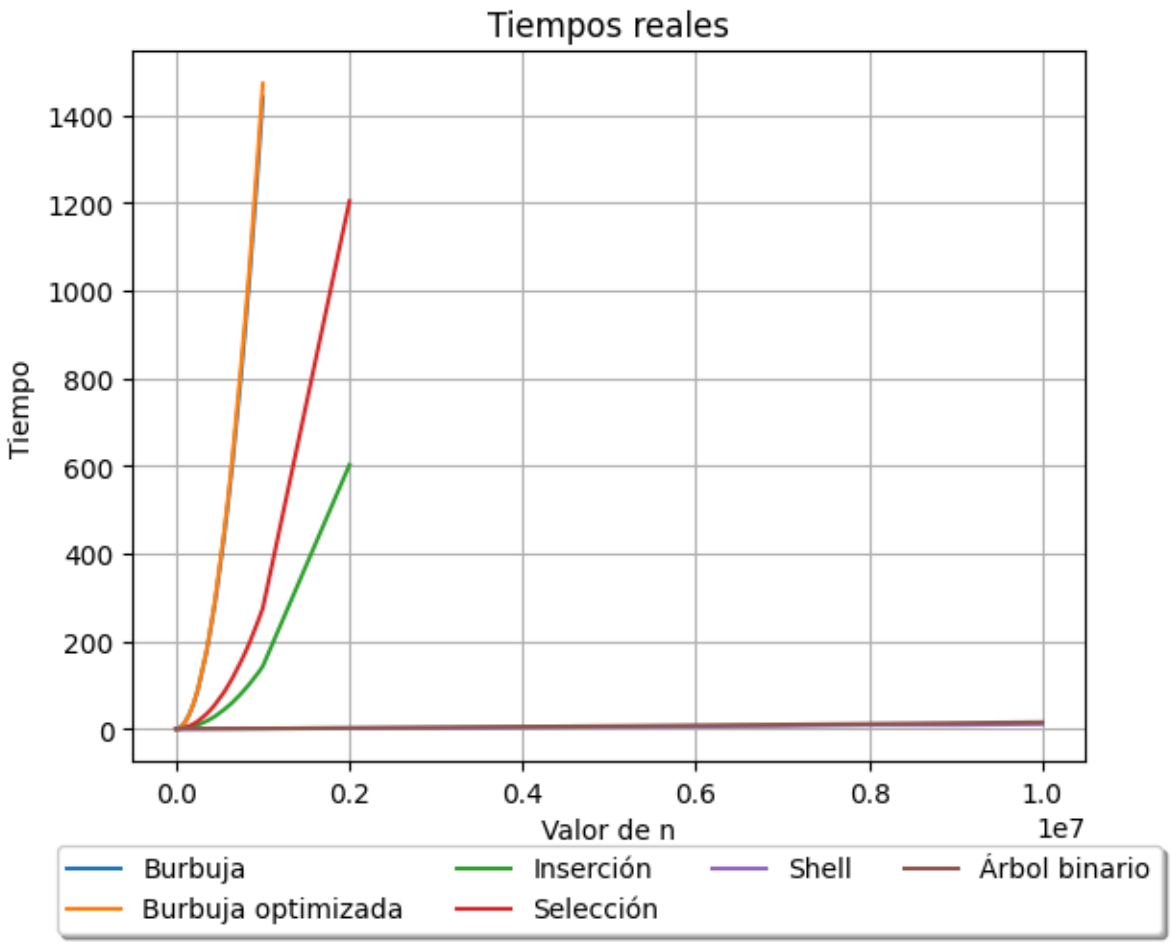
Fuente: Elaboración propia.

3.3 GRÁFICA COMPARATIVA GLOBAL

En esta sección se representa por medio de una gráfica el comportamiento de los diversos algoritmos, acorde con los valores n que se tomaron para cada algoritmo, teniendo la gráfica en el rango que se especifica:

- Burbuja:
  100 – 1,000,000.
- Burbuja optimizada:
  100 - 1,000,000.
- Inserción:
  100 - 2,000,000.
- Selección:
  100 - 2,000,000.
- Shell:
  100 - 10,000,000.
- Árbol binario:
  100 - 10,000,000.

Gráfica 14: Comparativa de tiempo real de ejecución

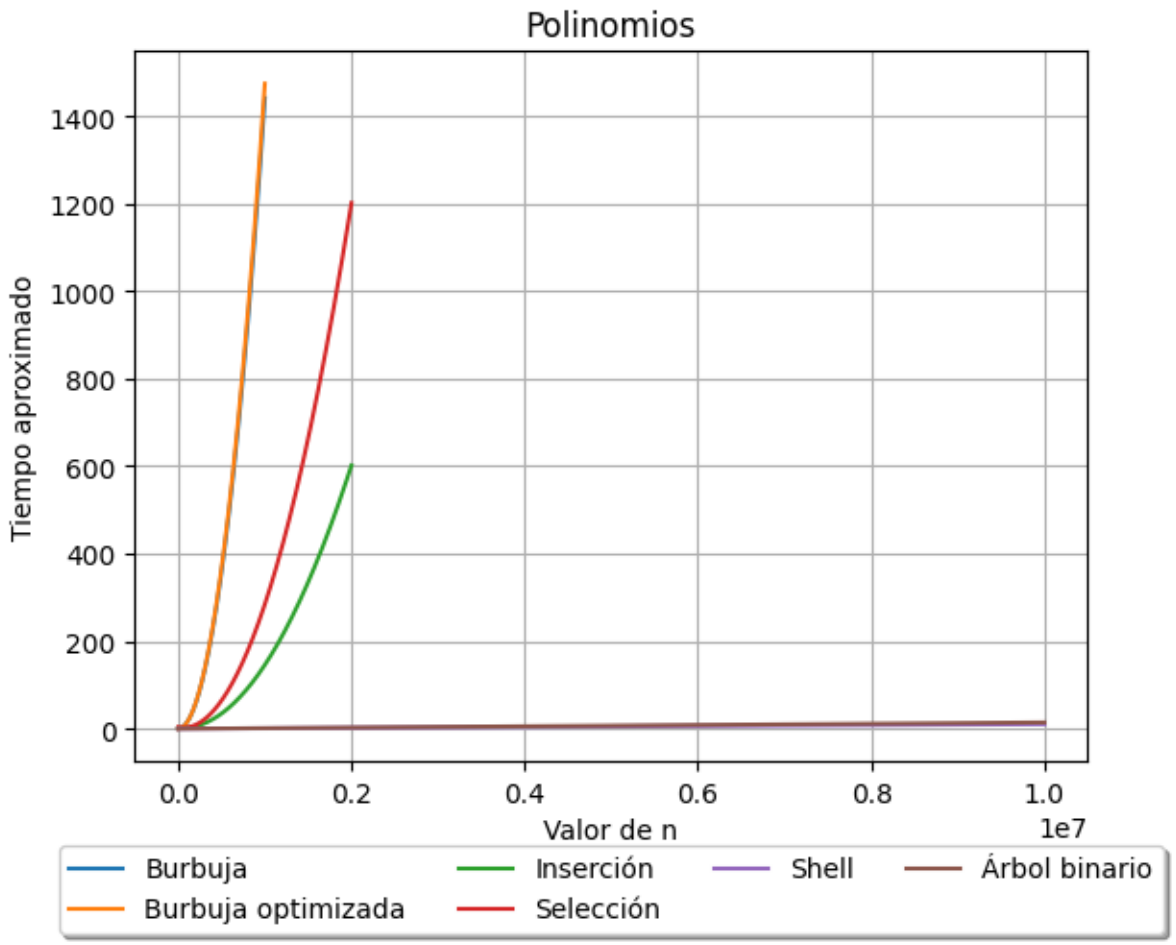


Fuente: Elaboración propia.

3.4 Comparativa global de mejor polinomio

En la sección se eligió el polinomio que fuera acorde con el algoritmo específico, aquí se conjuntan todos a través de una gráfica, con el fin de poder visualizar de mejor manera la información.

Gráfica 15: Aproximación polinomial global



Fuente: Elaboración propia.

3.5 APROXIMACIÓN DE VALORES N

En la sección *Función de complejidad* respectiva a cada algoritmo se seleccionó una aproximación polinomial, la cual fuera más acorde con el comportamiento de cada algoritmo (ver detalles en cada sección), con base en la aproximación seleccionada se realizó el cálculo del tiempo real de cada algoritmo para ordenar 15 000 000, 20 000 000, 500 000 000, 1 000 000 000 y 5 000 000 000. Los resultados se presentan a continuación:

Tabla 20: Aproximación para n valores

Algoritmo	15000000	20000000	500000000	1000000000	5000000000
Burbuja (Bubble Sort)	325541.2953	578782.4690	361815466.5732	1447268242.7715	36181833604.63663

Burbuja Optimizada	333090.1867	592204.6670	370207633.4753	1480837171.4816	37021062034.6298
Inserción (Insertion Sort)	34857.0772	62040.5280	38906385.0677	155636508.9803	3891132133.5665
Selección (Selection Sort)	71047.1934	126552.7356	79541944.5266	318205148.0597	7955876220.2282
Shell (Shell Sort)	15.2181	20.3788	515.8044	1031.8727	5160.4190
Árbol (Tree Sort)	21.3943	28.6717	727.3043	1455.0466	7276.9848

Fuente: Elaboración propia.

4 CUESTIONARIO

1. ¿Cuál de los 5 algoritmos fue más fácil de implementar?
   El algoritmo de burbuja es uno de los más simples, su implementación es muy gráfica entendiendo su funcionamiento
2. ¿Cuál de los 5 algoritmos fue más difícil de implementar?
   El *tree sort,* ya que conlleva más líneas de código y su implementación requiere de diversas funciones para poder realizar el ordenamiento.
3. ¿Cuál algoritmo tiene menor complejidad temporal?
   Teóricamente el *tree sort*, pues los números en el arreglo desordenado están distribuidos de forma uniforme y el árbol resultante estará más o menos balanceado, logrando la inserción de cada elemento en $\boldsymbol{O}(\log n)$ y la inserción de todos en $\boldsymbol{O}(n \log n)$.
   Sin embargo, de forma experimental el más rápido es *Shell Sort*, pues a pesar de que es cuadrático, los saltos usados sirven para reducir el número de inserciones y tener una constante muy baja.
4. ¿Cuál algoritmo tiene mayor complejidad temporal?
   Bubble Sort sin optimizaciones, pues siempre realiza todas las posibles comparaciones entre pares de elementos, sin importar cómo estén distribuidos los elementos en el arreglo.
5. ¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?
   En este caso podríamos decir que pueden ser diversos, tales como burbuja, burbuja optimizada, inserción y selección. Esto se debe a que son algoritmos de ordenamiento tipo in place, por ende, su complejidad temporal es de $\boldsymbol{O}(1)$.
6. ¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?
   El *tree sort*, ya que necesitamos construir un árbol binario con n nodos, requiriendo memoria adicional además del arreglo de entrada.
7. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?

En algunos sí y en otros no, pero eso lo pudimos ir definiendo a través de las aproximaciones que obtuvimos y visualizar si el comportamiento era el esperado, aunque alguno sí nos sorprendió.

8. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?
   Resulta difícil tener un entorno controlado. Sin embargo, para realizar un control posible se ejecutaron los algoritmos durante la noche, dejando pocos procesos corriendo y se realizaron en un solo dispositivo para encontrar datos que pudieran compararse de buena manera.
9. ¿Facilitó las pruebas mediante scripts u otras automatizaciones? ¿Cómo lo hizo?
   Para poder optimizar la realización de la práctica se recurrió al uso de un script que permitiera poder automatizar la realización de gráficas y la obtención de los datos de salida de cada uno de los algoritmos. Este se realizó por medio del lenguaje de programación Python, el código fuente se adjunta en anexos (*sección 6*).
10. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?
    Que es importante entender el funcionamiento de los algoritmos para poder llevarlos a su implementación. Además, es importante tener presente el dispositivo que tenga mejores características para poder disminuir el tiempo en el que van a ejecutarse los algoritmos y poder optimizar el tiempo. También es importante definir desde el primer momento en qué dispositivo se van a ejecutar para no tener que realizar el procedimiento más veces de las necesarias y también obtener los datos necesarios.

5 CONCLUSIONES

Guerrero Espinosa Ximena Mariana

Tras las pruebas realizadas me parece que resulta evidente que para el ordenamiento de números muy grandes el total perdedor resulta ser el ordenamiento burbuja pues este requiere de muchas operaciones para cumplir su objetivo. Notamos igualmente que los algoritmos tanto de inserción como selección resultan un poco más eficientes dado que se hacen menos comparaciones, pero no dejan de ser algoritmos poco competentes.

De la misma forma, al haber corrido todos los algoritmos en diferentes computadoras con especificaciones diferentes notamos un cambio grande con respecto a los tiempos de ejecución, haciendo resaltar el hecho que el tiempo de procesamiento no solo va ligado a que tan bien este implementado el algoritmo.

Los puntos anteriores son un claro ejemplo de los retos que podemos llegar a enfrentar al desarrollar cualquier código, sabemos que existen una gran cantidad de lenguajes de programación, herramientas y soluciones diversas a un mismo problema, pero cada una tiene un campo de acción enfocado a condiciones específicas y es por esto por lo que el desarrollador debe elegir su mejor opción teniendo siempre presente lo antes mencionado.

Hernández Espinoza Miguel Angel

En esta práctica vimos diferentes tipos de algoritmos de ordenamiento en la cual muchas de ellas aun teniendo la misma función el tiempo de ejecución cambia demasiado, en muchos casos esto también se debe a que dependiendo el estado inicial el tiempo de ejecución va a terminar ser de otra manera, por lo tanto concluyo que se debe de investigar y conocer varios algoritmos que cumplan la misma función para después aplicar el conveniente para un algoritmo mayor ya que la mayoría de las veces el algoritmo de menos líneas no siempre es el más conveniente, aun teniendo uno de los mejores casos posibles de ese algoritmo este se puede tardar incluso más que uno optimizado.

Jiménez Aguilar Tafnes Lorena

Esta práctica fue de ayuda para poder entender el comportamiento de algunos ordenamientos. Adquirí conocimientos que antes no tenía, ya que no solía darme cuenta de los tiempos de ejecución son una parte importante. Se hizo la implementación a código de lenguaje C a partir del pseudocódigo, pese a que esto solemos hacerlo durante toda la trayectoria escolar en su mayoría, pero me aportó para ser más consciente de los grandes cambios que pueden hacer los pequeños detalles que pueden hacer. Y en específico de cada algoritmo pude concluir que:

- *Bubble Sort optimizado* tardó incluso unos segundos más que la versión sin optimizar, eso no lo esperábamos. Tal vez se deba a cómo están distribuidos los números. Lo que sí fue un comportamiento esperado es el hecho de que estos algoritmos fueron los más tardados, debido a que hacer *swaps* entre elementos adyacentes es un poco más costoso.
- *Selection Sort* fue el siguiente en tardar más, lo cual es esperado, pero su ventaja con el anterior es que no realiza tantos *swaps*.
- *Insertion Sort* tardó aproximadamente la mitad del anterior, y es esperado, pues no realiza *swaps*, solo desplazamientos. Es por eso por lo que este algoritmo es uno de los más eficientes para ordenar arreglos pequeños y medianos.
- *Shell Sort* realmente nos sorprendió, ya que a pesar de que es cuadrático, fue el más rápido de todos, algo que iba totalmente en contra de lo que esperábamos. Los saltos escogidos para este arreglo en particular resultaron ser muy eficientes.
- *Tree Sort* fue el segundo más rápido y su comportamiento fue el esperado, porque los números estaban distribuidos aleatoriamente, entonces los árboles se crearon más o menos balanceados. Aunque como tal, puede que el tiempo de ordenamiento en sí sea menor, y que el resto del tiempo se deba a que hay que copiar el recorrido inorden del árbol de vuelta en el arreglo.

Otra cosa que se puso en práctica durante la realización de la ejecución de los algoritmos fue la automatización que nos permitió hacer un script, nunca había solido usarlos y en esta ocasión fue de gran ayuda para poder automatizar la obtención de diversas tareas al mismo tiempo de la ejecución.

Definitivamente esta práctica me permite ser más consciente de realizar una buena investigación y conocer diversos algoritmos para poder elegir de mejor manera la implementación de algoritmos, en este caso de ordenamiento de acuerdo si necesitamos algún método *in Place* o bien uno *out Place.*

6 ANEXO

En esta sección se incluyen los códigos fuente correspondiente a cada uno de los algoritmos de ordenamiento que se implementaron y sobre los cuales se basa el desarrollo y los resultados de la práctica presentados en este documento.

6.1 MAIN

Contiene en su estructura a los diversos algoritmos a través de la importación de estos como bibliotecas para poder hacer el llamado a las funciones que los conforman. Esto funciona por medio de un case que permite elegir el algoritmo a ejecutar.

6.1.1 CÓDIGO FUENTE

```
// Bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"
// Algoritmos
#include "bubbleSort.c"
#include "bubbleSortOp.c"
#include "insertionSort.c"
#include "selectionSort.c"
#include "shellSort.c"
#include "treeSort.c"

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

   /* Variables para time */

   // Variables para medición de tiempos
   double utime0, stime0, wtime0, utime1, stime1, wtime1;
   int n;        // n determina el tamaño del algoritmo dado por argumento al ejecutar
   int i; // Variables para loops

   // Variables
   int algoritmo; // Variable para elegir el algoritmo.

   if(argc!=3){
     printf("\nArgumento no v%clido",160);
     exit(1);
   }
   // Tomar el segundo argumento como tamaño del algoritmo
   else{
```

```
32      n=atoi(argv[1]);
33      algoritmo=atoi(argv[2]);
34    }
35
36    // Lectura
37    int *arr=malloc(sizeof(int)*n);
38    for(i=0; i<n; i++){
39      scanf("%d", &arr[i]);
40    }
41
42    /*-----------------------------------------------------------------
43      Iniciar el conteo del tiempo para las evaluaciones de rendimiento
44     -----------------------------------------------------------------*/
45     uswtime(&utime0, &stime0, &wtime0);
46
47     // Case para seleccionar el algoritmo a ejecutar
48     switch(algoritmo){
49       case 1: // Bubble Sort
50               bubbleSort(arr, n);
51               break;
52       case 2: // Bubble Sort Optimizado
53               bubbleSortOp(arr, n);
54               break;
55       case 3: // Insertion Sort
56               insertionSort(arr, n);
57               break;
58       case 4: // Selection Sort
59               selectionSort(arr, n);
60               break;
61       case 5: // Shell Sort
62               shellSort(arr, n);
63               break;
64       case 6: // Tree Sort
65               treeSort(arr, n);
66               break;
67       default:
68               printf("\n Opci%cn no v%clida", 162, 160);
69               break;
70         }
71
72     uswtime(&utime1, &stime1, &wtime1);
73
74     // Verificar que las soluciones son correctas
75     for(i=0; i<n-1; i++){
76       if(arr[i]>arr[i+1]){
77         printf("No ordenado.\n");
78         exit(1);
79       }
80     }
81
82     /*-----------------------------------------------------------
83       Impresión del cálculo del tiempo de ejecución del programa
84      -----------------------------------------------------------*/
```

```
85      // Tiempo real
86      printf("%.10f ",wtime1-wtime0);
87      // Tiempo de CPU
88      printf("%.10f ",utime1-utime0);
89      //Tiempo E/S
90      printf(" %.10f",stime1-stime0);
91      // CPU/Wall %
92      printf(" %.10f \n",100.0*(utime1-utime0+stime1-stime0)/(wtime1-wtime0));
93
94      // Terminar programa normalmente
95      exit (0);
96  }
```

6.2 Burbuja (Bubble Sort)

Función de ordenamiento por medio del método de burbuja (Bubble Sort).

6.2.1 Código fuente

```
1  #include <stdio.h>
2
3  void bubbleSort(int arr[], int n){
4    int i, j, temp; // variables para loops
5    // Iteración por los n valores del array
6    for(i=0; i<n-1; i++){
7      for(j=0; j<(n-1)-i; j++){
8        if(arr[j]>arr[j+1]) {
9          // Swap si arr[j] es mayor
10         temp=arr[j];
11         arr[j]=arr[j+1];
12         arr[j+1]=temp;
13       }
14     }
15   }
16 }
```

6.3 Burbuja Optimizada

Función de ordenamiento por medio del método de burbuja (Bubble Sort optimizado).

6.3.1 Código fuente

```
1  #include <stdio.h>
2
3  void bubbleSortOp(int arr[], int n){
4    // Variable bandera y auxiliar
5    int cambios=1, aux;
6    int i, j; // Variables para loops
7    while(i<n-1 && cambios!=0){
8      cambios=0;
9      // Iteración con decremento para optimizar
10     for(j=0; j<(n-1)-i; j++){
```

```
11        if(arr[j]>arr[j+1]){
12          // Swap si arr[j] es mayor
13          aux=arr[j];
14          arr[j]=arr[j+1];
15          arr[j+1]=aux;
16          cambios=1; // Cambio de bandera
17        }
18      }
19      // Incremento a la iteración del for
20      i=i+1;
21    }
22  }
```

6.4 Inserción (Insertion Sort)

Función de ordenamiento por medio del método de inserción (Insertion Sort).

6.4.1 Código fuente

```
1  #include<stdio.h>
2
3  void insertionSort(int arr[], int n){
4    int i, j; // Variables para loops
5    int temp; // Variable auxiliar
6    // Iteración por los n valores del array
7    for(i=0; i<n; i++){
8      j=i;
9      temp=arr[i];
10     while(j>0 && temp<arr[j-1]){
11       arr[j]=arr[j-1];
12       j--;
13     }
14     // Swap
15     arr[j]=temp;
16   }
17 }
```

6.5 Selección (Selection Sort)

Función de ordenamiento por medio del método de selección (Selection Sort).

6.5.1 Código fuente

```
1  #include<stdio.h>
2
3  void selectionSort(int arr[], int n){
4    int i=0, k=0, p=0; // Variables para Loops
5    int temp=0; // Variable auxiliar
6    // Iteración por los n valores del array
7    for(k=0; k<n-1; k++){
8      p=k;
9      // Búsqueda del mínimo elemento
```

```
10      for(i=k+1; i<n; i++){
11        if(arr[i]<arr[p]){
12          p=i;
13        }
14      }
15      // Swaps
16      temp=arr[p];
17      arr[p]=arr[k];
18      arr[k]=temp;
19    }
20  }
```

6.6 SHELL (SHELL SORT)

Función de ordenamiento por medio del método de Shell (Shell Sort).

6.6.1 CÓDIGO FUENTE

```
1  #include <stdio.h>
2
3  void shellSort(int arr[], int n){
4    int distancia=n/2; // Variable para partición
5    int i, j; // Variables para Loops
6    int temp; // Variable auxiliar
7    while(distancia>0){
8      i=1;
9      while(i!=0){
10       i=0;
11       // Iteración por los n elementos del array
12       for(j=distancia; j<=n-1; j++){
13         if(arr[j-distancia]>arr[j]){
14           // Swaps
15           temp=arr[j];
16           arr[j]=arr[j-distancia];
17           arr[j-distancia]=temp;
18           i++;
19         }
20       }
21     }
22     // Particiones a la mitad
23     distancia/=2;
24   }
25 }
```

6.7 ORDENAMIENTO CON ÁRBOL BINARIO DE BÚSQUEDA (TREE SORT)

Función de ordenamiento por medio del método de un árbol binario de búsqueda (Tree Sort).

6.7.1 CÓDIGO FUENTE

```
1  #include <stdio.h>
2
```

```c
// Declaración de struct
typedef struct nodo {
  struct nodo* izq, *der;
  int num;
}nodo;

// Función para crear nodos
nodo* crear_nodo(int numb){
  nodo* new_nodo=malloc(sizeof(nodo));
  new_nodo->num=numb;
  new_nodo->izq=NULL; // Hijo izquierdo
  new_nodo->der=NULL; // Hijo derecho
  return new_nodo;
}

// Función para insertar en el árbol
void insertar(int num_nodo, nodo**raiz){
  while(*raiz!=NULL){
    if(num_nodo<(*raiz)->num){
      raiz=&(*raiz)->izq; // Hijos menores
    }else{
      raiz=&(*raiz)->der; // Hijos mayores
    }
  }
  // Insertar valor en el nodo
  *raiz=crear_nodo(num_nodo);
}

int x=0; // Variable para iterar

// Función para In Order
void inOrder(nodo*raiz, int*arr){
  if(raiz==NULL){
    return;
  }else{
    // Llamadas recursivas a la función
    inOrder(raiz->izq, arr);
    arr[x]=raiz->num;
    x++;
    inOrder(raiz->der, arr);
  }
}

// Aplicación del algortimo
void treeSort(int*arr, int n){
  int i=0;
  nodo*tree=NULL;
  for(i=0; i<n; i++){
    insertar(arr[i], &tree);
  }
  inOrder(tree, arr);
}
```

6.8 SCRIPT

Script para la automatización de compilación y obtención de las salidas, las compilaciones se hicieron con gcc main.c tiempo.c -O2, la bandera -O2 nos ayudará a una disminución de tiempo para la ejecución.

6.8.1 CÓDIGO FUENTE

```
import os
import subprocess
import numpy as np
import matplotlib.pyplot as plt

# Función graficar
def graficar(data_x, data_y, legends, label_x, label_y, title, file, marker):
  for i in range(0, len(data_x)):
      plt.plot(data_x[i], data_y[i], label=legends[i], marker=marker)
  plt.grid(True)
  plt.xlabel(label_x)
  plt.ylabel(label_y)
  plt.title(title)
  plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.1), shadow=True, ncol=4)
  plt.savefig(file, bbox_inches='tight')
  plt.clf()

# Compilación
estado=os.system("gcc main.c tiempo.c -O2")
if estado!=0:
  print("No compiló.")
  exit(0)
valores=[100,1000,5000,10000,50000,100000,150000,200000,250000,350000,400000,450000,500000,550000,600000,650000,750000,800000,850000,950000,1000000,2000000,3000000,4000000,5000000,6000000,7000000,8000000,9000000,10000000]
n_grandotas=[15000000,20000000,500000000,1000000000,5000000000]
etiquetas=['Burbuja','Burbuja optimizada','Inserción','Selección','Shell','Árbol binario']
grados=[1,2,3,4,8]
eje_global_x=[]
eje_global_y=[]
poly_eje_x=[]
poly_eje_y=[]

# Gráficas indivudales de tiempo
for x in range(1,7):
  eje_x=[]
  tReal=[]
  tCPU=[]
  tES=[]
  tPor=[]
  for n in valores:
    if x<=2 and n>1000000:
      continue
    if x>=3 and x<=4 and n>2000000:
      continue
    print(F"Algoritmo: {etiquetas[x-1]} en {n}")
    salida=subprocess.check_output(F"./a.out {n} {x} < numeros10millones.txt", shell = True, universal_newlines = True)
```

```
    datos=[float(i) for i in salida.split()]
    tiempoReal=datos[0]
    tiempoCPU=datos[1]
    tiempoE_S=datos[2]
    cpuPor=datos[3]
    tReal.append(tiempoReal)
    tCPU.append(tiempoCPU)
    tES.append(tiempoE_S)
    tPor.append(cpuPor)
    eje_x.append(n)

    print(F"{tiempoReal} {tiempoCPU} {tiempoE_S} {cpuPor}")

  # Guardar en un archivo las salidas
  archivo=open(F"{etiquetas[x-1]}.txt", "w")
  archivo.write("Tiempos reales\n")
  for i in range(0,len(eje_x)):
    archivo.write(F"{eje_x[i]} {tReal[i]} {tCPU[i]} {tES[i]} {tPor[i]}% \n")

  # Polinomio
  polinomios_x=[]
  polinomios_y=[]
  mejor_polinomio=None;
  for grado in grados:
    polinomio = np.poly1d(np.polyfit(eje_x, tReal, grado))
    eje_x_poly = np.linspace(0, eje_x[-1], 1000)
    polinomios_x.append(eje_x_poly)
    evaluaciones = polinomio(eje_x_poly)
    polinomios_y.append(evaluaciones)

    # Mejor polinomio
    if x<5 and grado==2:
      poly_eje_x.append(eje_x_poly)
      poly_eje_y.append(evaluaciones)
      mejor_polinomio=polinomio
    elif x>=5 and grado==1:
      poly_eje_x.append(eje_x_poly)
      poly_eje_y.append(evaluaciones)
      mejor_polinomio=polinomio

  # Estimación de las n gradotas
  archivo.write(F"Mejor polinomio:\n {mejor_polinomio}\n")

  # Aproximaciones con los n valores
  archivo.write("Polinomio \n")
  for n_grandota in n_grandotas:
    archivo.write(F"{n_grandota} {mejor_polinomio(n_grandota)}\n")
  archivo.close()

  # Gráfica de los polimonios
  graficar(polinomios_x, polinomios_y, [F"Polinomio grado {grados[i]}" for i in range(0,
len(grados))],"Valor de n", "Tiempo aproximado", F"Polinomios de {etiquetas[x-1]}",
F"pol{etiquetas[x-1]}.png", "")

  eje_global_x.append(eje_x)
```

```
 97   eje_global_y.append(tReal)
 98
 99   # Gráfica de los tiempos reales
100   graficar([eje_x, eje_x, eje_x],[tReal, tCPU, tES], ["Tiempo real","Tiempo CPU", "Tiempo E-
     S"], "Valor de n", "Tiempo", etiquetas[x-1], F"{etiquetas[x-1]}.png", "")
101
102 # Gráfica global de los tiempos reales
103 graficar(eje_global_x, eje_global_y, etiquetas, "Valor de n", "Tiempo", "Tiempos reales",
    "general.png", "")
104
105 # Gráfica global de los polinomios
106 graficar(poly_eje_x, poly_eje_y, etiquetas, "Valor de n", "Tiempo aproximado", "Polinomios",
    "global_polinomios.png", "")
```

7 BIBLIOGRAFÍA

[1] J. C. O. María del Carmen Gómez Fuentes, Introducción al Análisis y al Diseño de Algoritmos, Ciudad de México: Universidad Autónoma Metropolitana, 2014.

[2] A. V. G. Sara Baase, Algoritmos computacionales: Introducción al análisis y diseño, Ciudad de México: Pearson Educación, 2002.