Instituto Politécnico Nacional
Escuela Superior de Cómputo

Algoritmos de ordenamiento

Pruebas a posteriori

Análisis de Algoritmos
3CM3
Edgardo Adrián Franco Martínez

Integrantes:
Erick Quintana Martínez
Luis Armando Ramirez Espinosa
Luis Fernando Reséndiz Chávez

Índice

1. Planteamiento del problema

En esta práctica vamos a medir los tiempos de ejecución de varios algoritmos de ordenamiento conocidos, los cuales serán implementados en ANSI C. Para efectos de esta práctica, la implementación de cada algoritmo se realizó con base en pseudocódigos proporcionados por el profesor:

Algorithm 1: Ordenamiento de burbuja

```
Data: Arreglo A de tamaño n compuesto de enteros.
Result: El arreglo A es ordenado de manera no decreciente.
initialization;
for i ← 0 to n − 2 by 1 do
    for j ← 0 to (n − 2) − i by 1 do
        if A[j] > A[j + 1] then
            aux = A[j];
            A[j] = A[j + 1];
            A[j + 1] = aux;
        end
    end
end
```

Imagen 1.1: Pseudocódigo para el algoritmo “Ordenamiento de burbuja”

Algorithm 2: Ordenamiento de burbuja optimizado

```
Data: Arreglo A de tamaño n compuesto de enteros.
Result: El arreglo A es ordenado de manera no decreciente.
initialization;
cambios = “Si”;
i = 0;
while i < n-1 && cambios != “No” do
    cambios = “No”;
    for j ← 0 to (n − 2) − i by 1 do
        if A[j] > A[j + 1] then
            aux = A[j];
            A[j] = A[j + 1];
            A[j + 1] = aux;
            cambios = “Si”
        end
    end
    i = i + 1;
end
```

Imagen 1.2: Pseudocódigo para el algoritmo “Ordenamiento de burbuja optimizado”

Algorithm 3: Ordenamiento de inserción

```
Data: Arreglo A de tamaño n compuesto de enteros.
Result: El arreglo A es ordenado de manera no decreciente.
initialization;
for i ← 0 to n − 1 by 1 do
    j = i;
    temp = A[i];
    while j>0 && temp<A[j-1] do
        A[j] = A[j - 1];
        j = j - 1;
    end
    A[j] = temp;
end
```

Imagen 1.3: Pseudocódigo para el algoritmo “Ordenamiento de inserción”

Algorithm 4: Ordenamiento de selección

```
Data: Arreglo A de tamaño n compuesto de enteros.
Result: El arreglo A es ordenado de manera no decreciente.
initialization;
for k ← 0 to n − 2 by 1 do
    p = k;
    for i = k + 1 to n − 1 by 1 do
        if A[i] < A[o] then
            p = i;
        end
    end
    temp = A[p];
    A[p] = A[k];
    A[k] = temp;
end
```

Imagen 1.4: Pseudocódigo para el algoritmo “Ordenamiento de selección”

Algorithm 5: Ordenamiento Shell

```
Data: Arreglo A de tamaño n compuesto de enteros.
Result: El arreglo A es ordenado de manera no decreciente.
initialization;
k = TRUNC (n / 2);
while k ≥ 1 do
    b = 1;
    While b ≠ 0  b = 0;
    for i ← k to i ≥ n - 1 by 1 do
        if A[i - k] > A[i] then
            temp = A[i];
            A[i] = A[i - k];
            A[i - k] = temp;
            b = b + 1;
        end
    end
    k = TRUNC (k / 2);
end
```

Imagen 1.5: Pseudocódigo para el algoritmo “Ordenamiento Shell”

Algorithm 6: Ordenamiento con Árbol binario de búsqueda

```
Data: Arreglo A de tamaño n compuesto de enteros, árbol binario de
      búsqueda.
Result: El arreglo A es ordenado de manera no decreciente.
initialization;
for i ← 0 to i ≥ n do
    Insertar (ArbolBinBusqueda, A[i]);
end
GuardarRecorridoInOrden (ArbolBinBusqueda, A);
```

Imagen 1.6: Pseudocódigo para el algoritmo “Ordenamiento con Árbol Binario de Búsqueda”

En esta práctica, se realizarán mediciones sobre los siguientes valores:

- Tiempo real desde el inicio de ejecución del programa hasta su finalización.
- Tiempo que la CPU usa para computar el programa
- Tiempo que la CPU dedicó a servicios del sistema operativo (como entrada y salida)
- Porcentaje de tiempo dedicado al programa por parte del CPU

La medición de tiempos se realizará con un procedimiento (también facilitado por el profesor) que hace uso de funciones y temporizadores de Unix para obtener medidas de tiempo durante los ordenamientos.

Los valores a ordenar se encuentran en un archivo externo, y cuenta con 10 millones de números obtenidos de manera (pseudo)aleatoria. Para realizar las pruebas, cada programa solicita un argumento indicando el número de valores por ordenar.

2. Actividades y pruebas

2.1. Tabla comparativa sobre 500,000 números a ordenar

Con los algoritmos implementados, procedemos a la medición de los valores solicitados. Para lograrlo, cada programa será sometido a una prueba para ordenar 500,000 números de la lista, y se obtendrán los tiempos reales, de CPU, del sistema y el porcentaje de tiempo de CPU necesario. Los resultados se encuentran en la tabla 2.1. Se incluyen los tiempos en formato inglés para facilitar la lectura

Algoritmo	Tiempo real	Tiempo CPU	Tiempo E/S	% CPU/Wall
Burbuja	1070.736 s	1049.964 s	0.4816 s	98.105 %
Burbuja Optimizada	527.176 s	512.062 s	0.4196 s	97.212 %
Inserción	192.088 s	192.066 s	0.004 s	99.991 %
Selección	664.057 s	656.203 s	0.1406 s	98.838 %
Árbol binario de búsqueda	0.686 s	0.67 s	0.016 s	99.977 %
Shell	0.229 s	0.228 s	0.0003 s	99.996 %

Tabla 2.1: Comparación de tiempos de ejecución en tiempos decimales

Algoritmo	Tiempo real	Tiempo CPU	Tiempo E/S	% CPU/Wall
Burbuja	17’ 51’’	17’ 29’’	48’’’	98.105 %
Burbuja Optimizada	8’ 47’’	8’ 32’’	42’’’	97.212 %
Inserción	3’ 12’’	3’ 12’’	0’’’	99.991 %
Selección	11’ 04’’	10’ 56’’	14’’’	98.838 %
Árbol binario de búsqueda	69’’’	67’’’	2’’’ s	99.977 %
Shell	23’’’	23’’’	0’’’	99.996 %

Tabla 2.2: Comparación de tiempos de ejecución en formato inglés

La medición de tiempos se obtiene fijando instantes del inicio y fin de la medición de tiempo con la función uswtime (), que son a partir del momento en el que la función que realiza al ordenamiento (en su respectivo programa) es llamada, y al regresar al hilo principal, respectivamente. Las diferencias de los temporizadores entre cada invocación de uswtime () nos otorga los valores que vemos en la tabla 2.1.

2.2. Gráfica temporal sobre cada algoritmo

En esta sección vamos a observar la relación entre el número de elementos a ordenar y el tiempo real de ejecución del programa. Cada algoritmo fue ejecutado varias veces con distintos valores para el parámetro *n* indicando cuántos números se leerán y ordenarán en cada prueba.

n = [100, 150, 300, 500, 1000, 1500, 3000, 5000, 10000, 15000, 30000, 40000, 50000, 100000, 150000, 400000, 500000, 600000, 800000, 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000]

Los resultados se guardaron en archivos auxiliares, y se ingresaron en el software Matlab para generar las gráficas que se muestran a continuación, asistiéndonos de un script que proporcionó el profesor.

Nota importante: Algunos algoritmos comenzaron a mostrar tiempos de cómputo altos para valores de n elevados ($n \geq 100000$). Esto sucede con los ordenamientos Burbuja, Burbuja Optimizado, Inserción y Selección; se tomó como valor máximo de n a 3,000,000.

2.2.1. Ordenamiento Burbuja

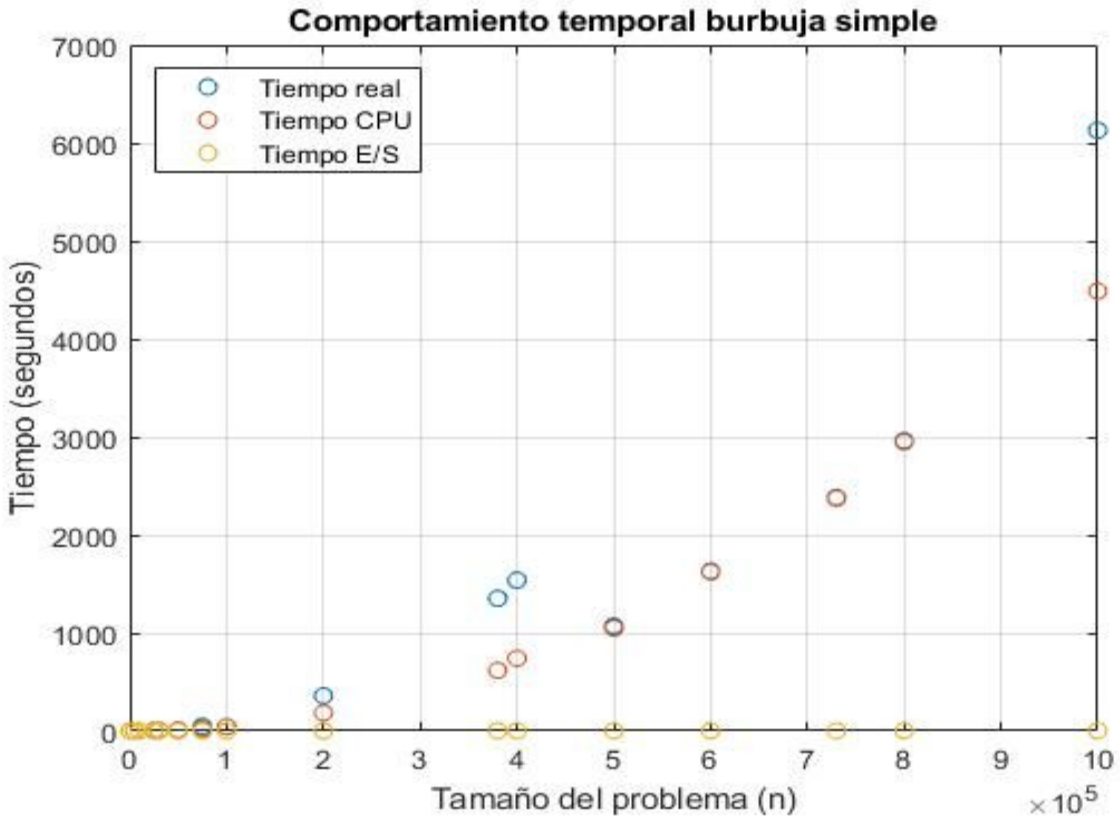


Imagen 2.1. Ordenamiento Burbuja

2.2.2. Ordenamiento Burbuja Optimizada

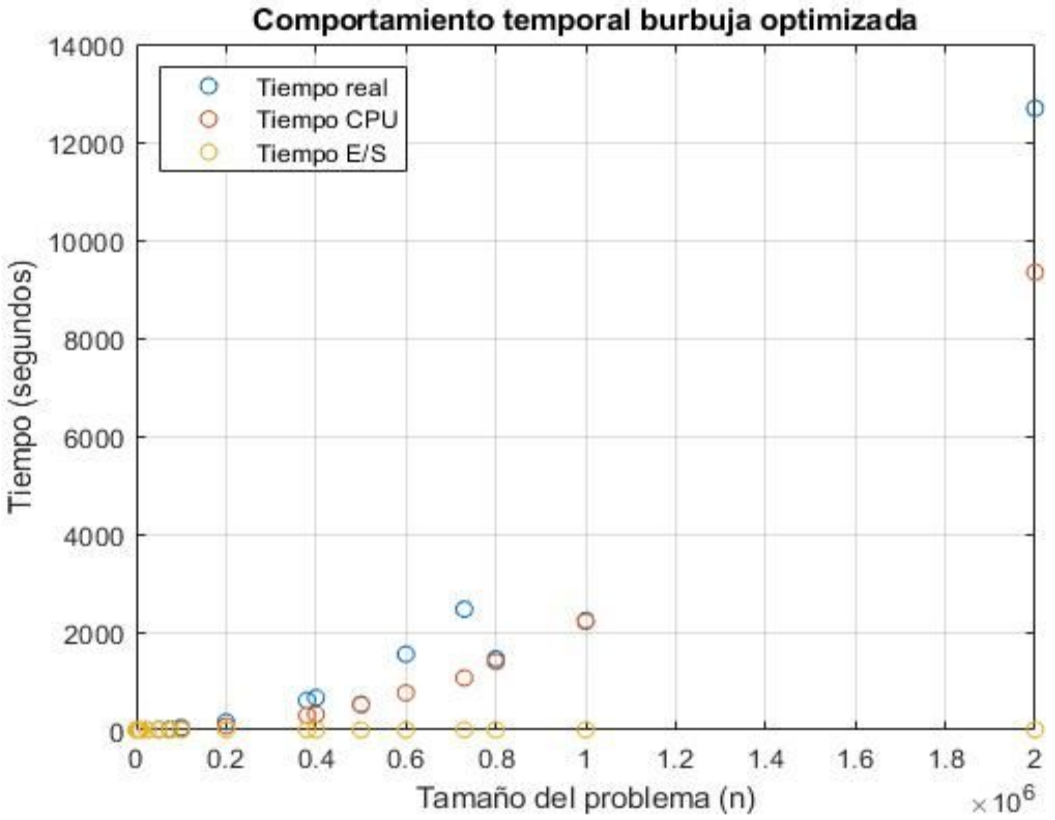


Imagen 2.2. Ordenamiento Burbuja optimizado

2.2.3. Ordenamiento de Inserción

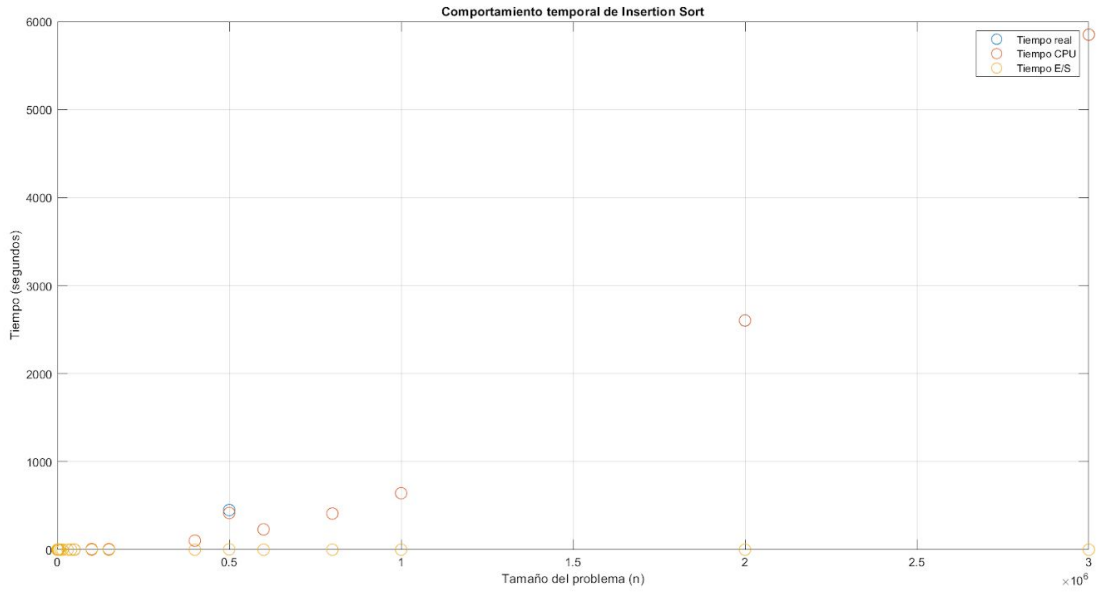


Imagen 2.3. Ordenamiento de Inserción

2.2.4. Ordenamiento de Selección

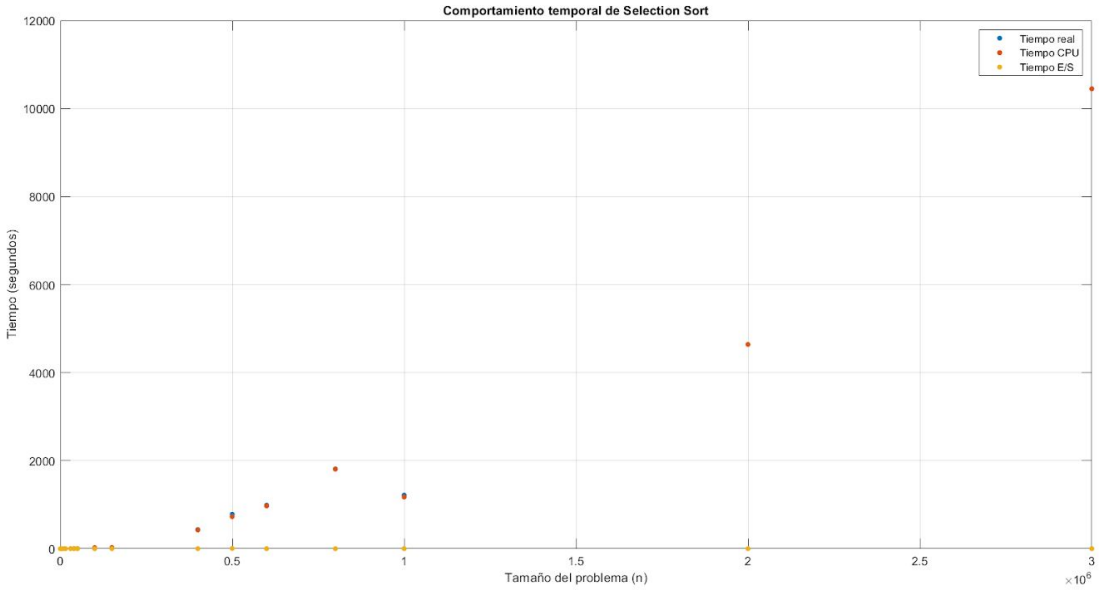


Imagen 2.4. Ordenamiento de Selección

2.2.5. Ordenamiento con Árbol Binario de Búsqueda

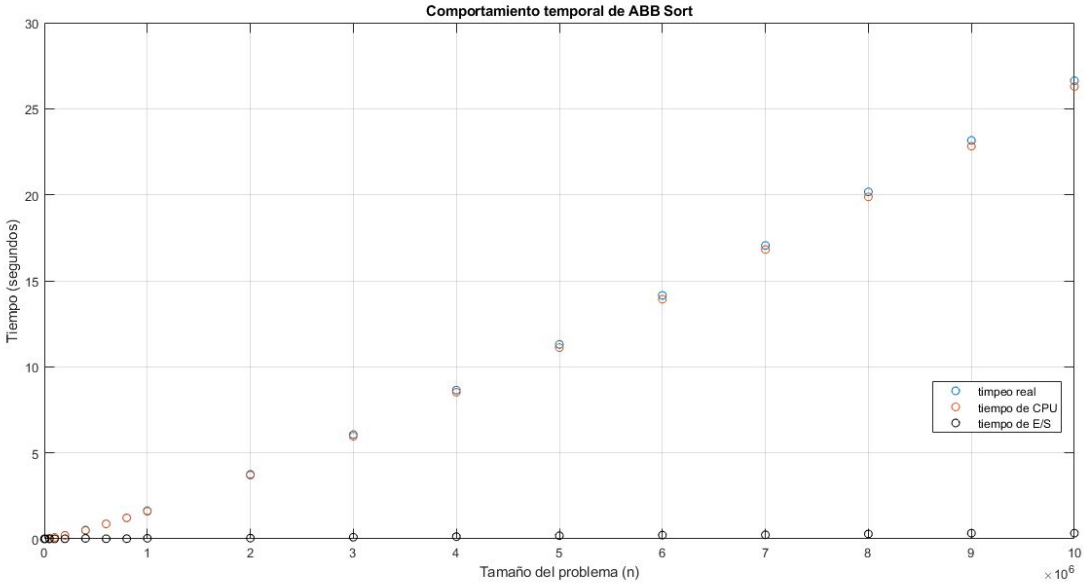


Imagen 2.5. Ordenamiento con Árbol Binario de Búsqueda

2.2.6. Ordenamiento Shell

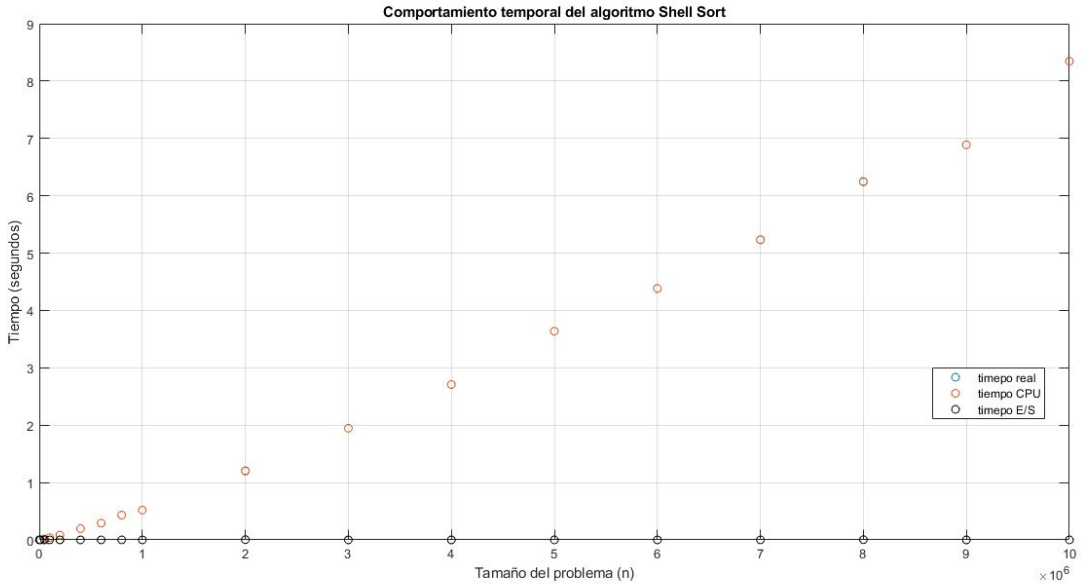


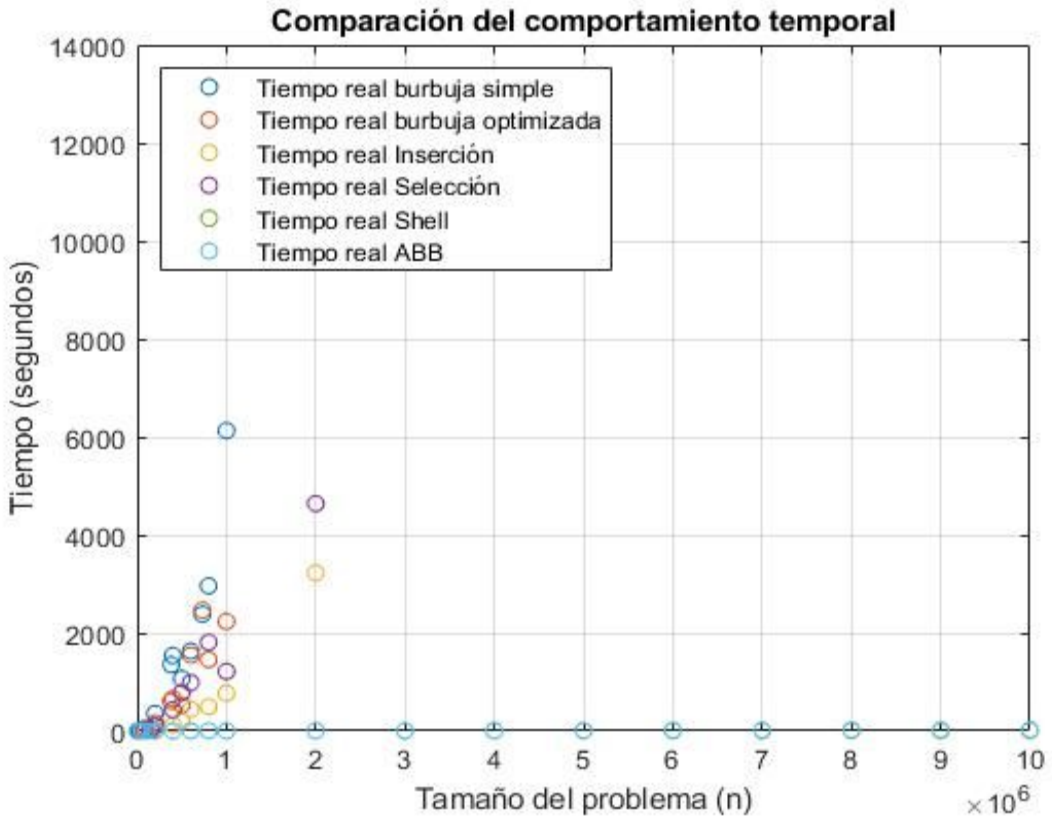
Imagen 2.6. Ordenamiento Shell

Podemos observar en las imágenes que los puntos para los tiempos reales y de CPU prácticamente se traslapan, y que los tiempos del sistema se mantienen muy cerca del eje Tamaño del problema (información que concuerda con lo visto en la sección 2.1). Esto lleva a la conclusión natural que el principal consumidor de tiempo es el CPU intentando computar la información, y no otras operaciones como entrada y salida de datos.

Más importante aún es que podemos ver el crecimiento del tiempo de ejecución conforme aumenta el número de valores a ordenar porque hay más instrucciones por realizar; efectivamente, la dificultad del problema aumenta. Este efecto sucede en cada uno de los algoritmos de la práctica.

2.3. Comparativa de tiempo real de los algoritmos

la gráfica del tiempo real de todos los algoritmos de ordenamiento implementados en esta práctica es la que se muestra a continuación.



Se puede observar que el mejor algoritmo es el ABB porque se puede probar para todas los 10 millones de números ya que en los de burbuja, inserción y selección no se pudo lograr el ordenamiento de todos esos números. Además el tiempo es muy bajo a comparación de los otros algoritmos.

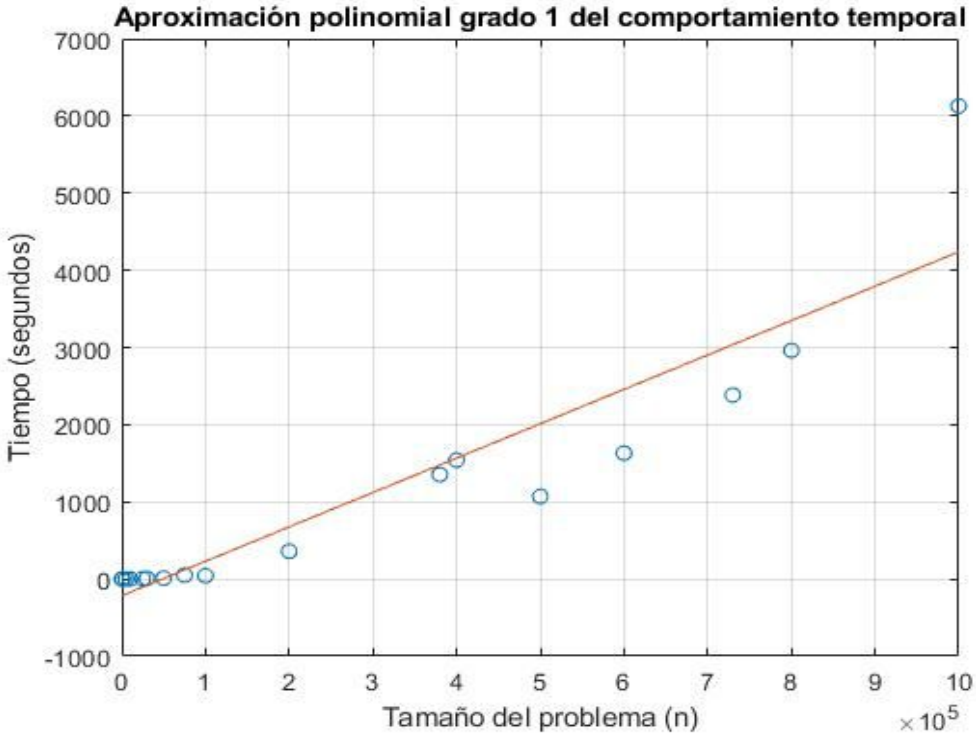
2.4. Aproximación polinomial del comportamiento temporal real de los algoritmos

Ya que tenemos un conjunto de puntos Tamaño del problema-Tiempo de ejecución, podemos realizar una aproximación polinomial que describa el comportamiento del tiempo en función de n. Tener un polinomio será útil para la sección 2.7 en donde realizaremos diferentes interpolaciones de puntos.

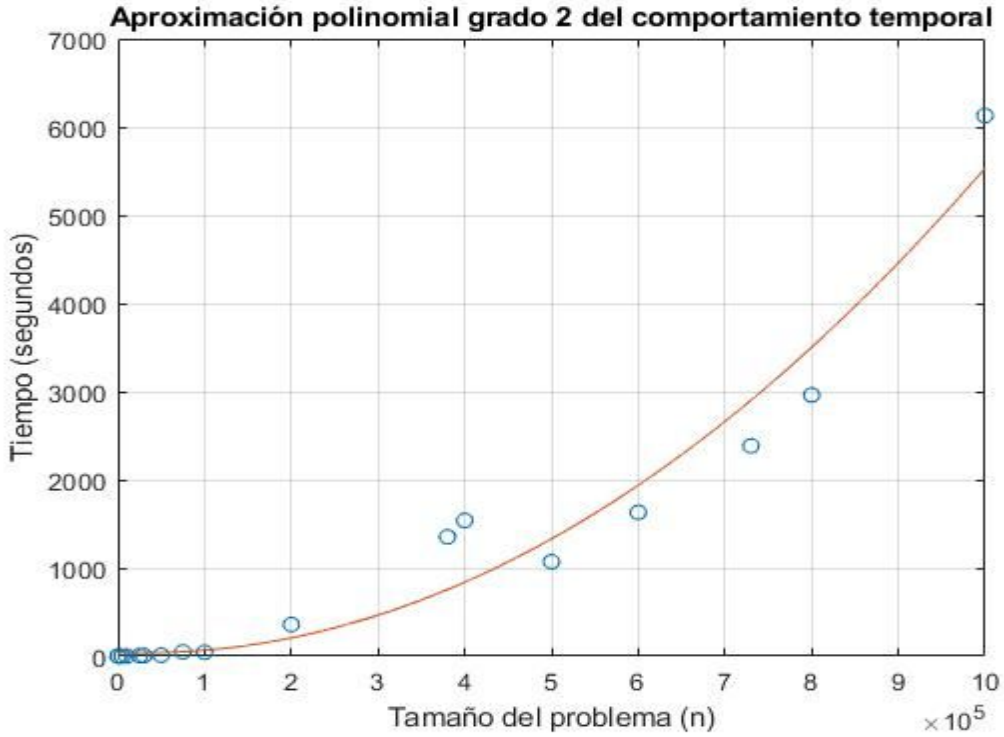
Vamos a obtener polinomios lineales, cuadráticos, y de grados 3, 4 y 8. Esta sección también fue resuelta con el script de Matlab mencionado en la sección anterior.

2.4.1. Ordenamiento Burbuja

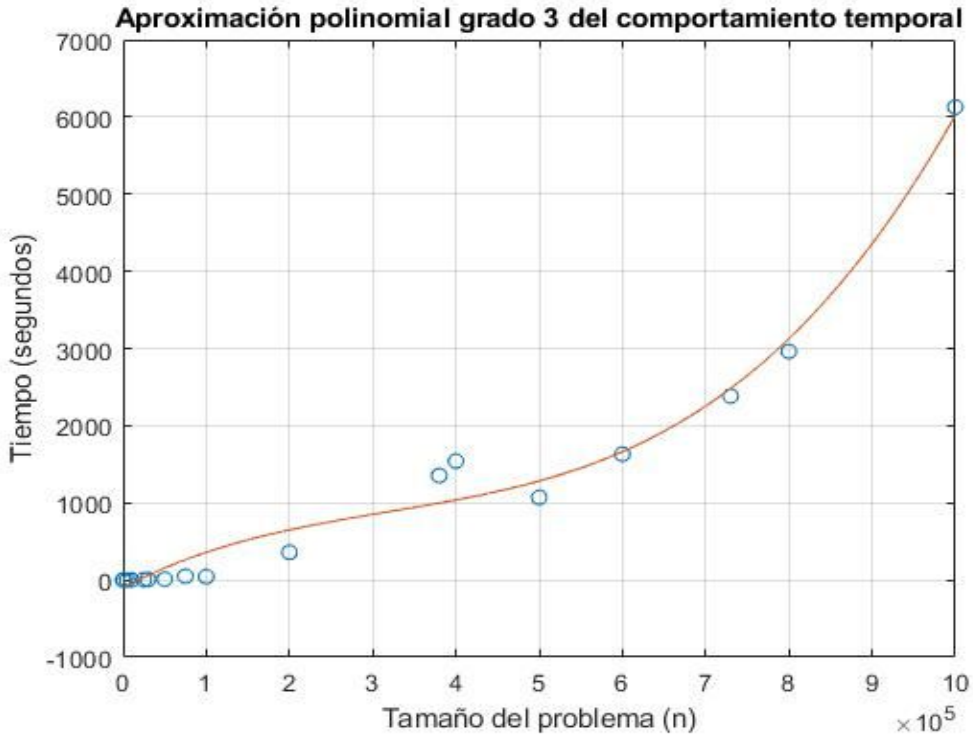
Aproximación polinomial de grado 1



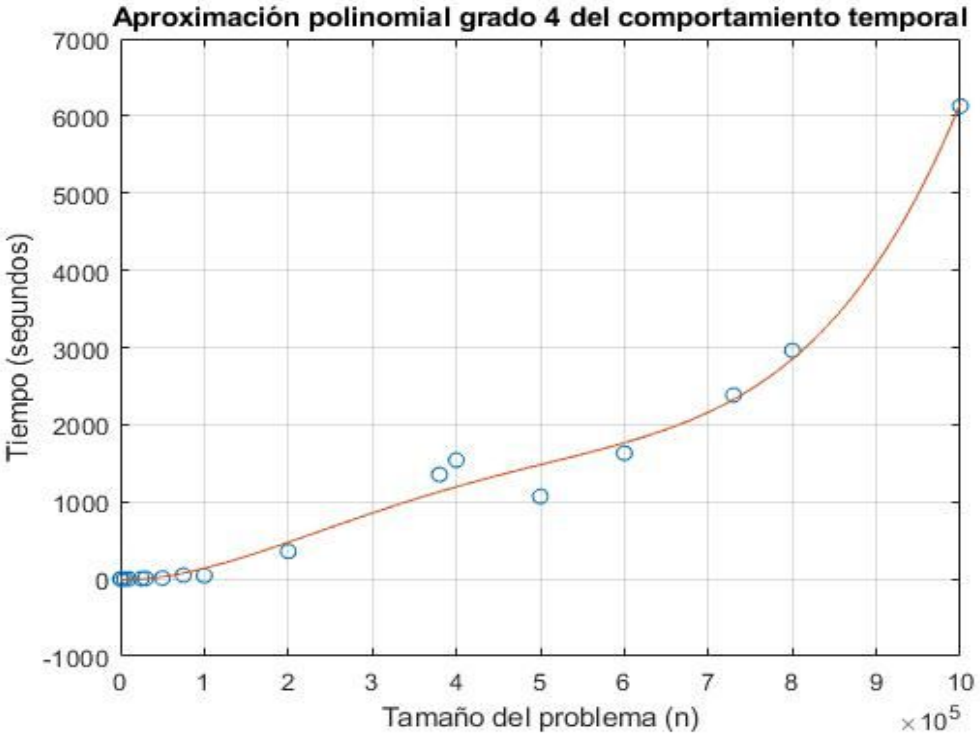
Aproximación polinomial de grado 2



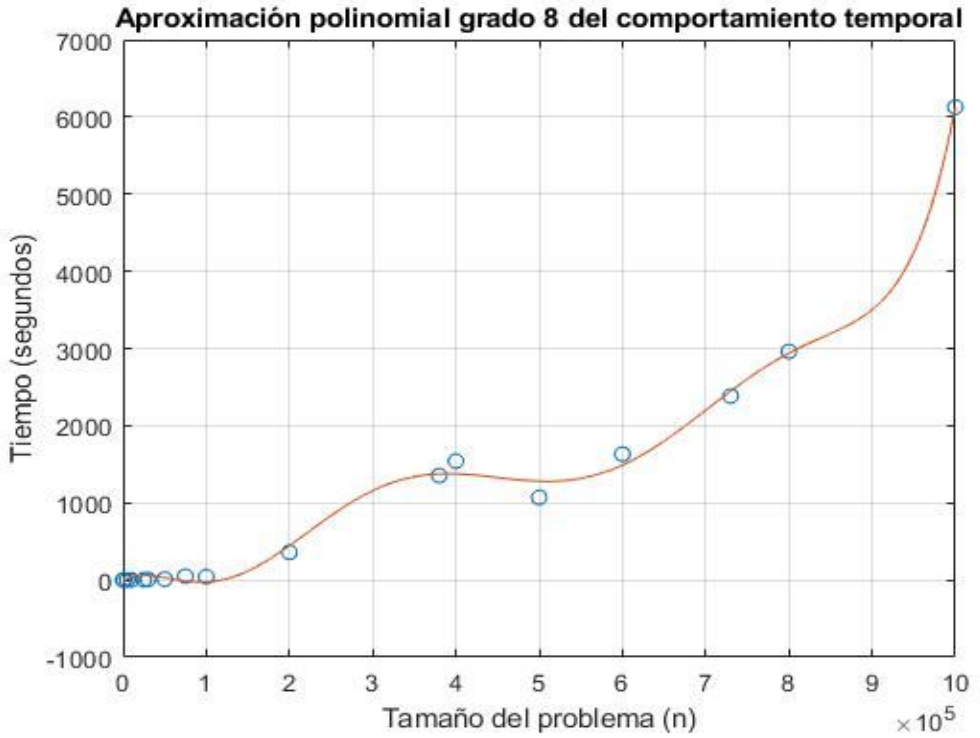
Aproximación polinomial de grado 3



Aproximación polinomial de grado 4

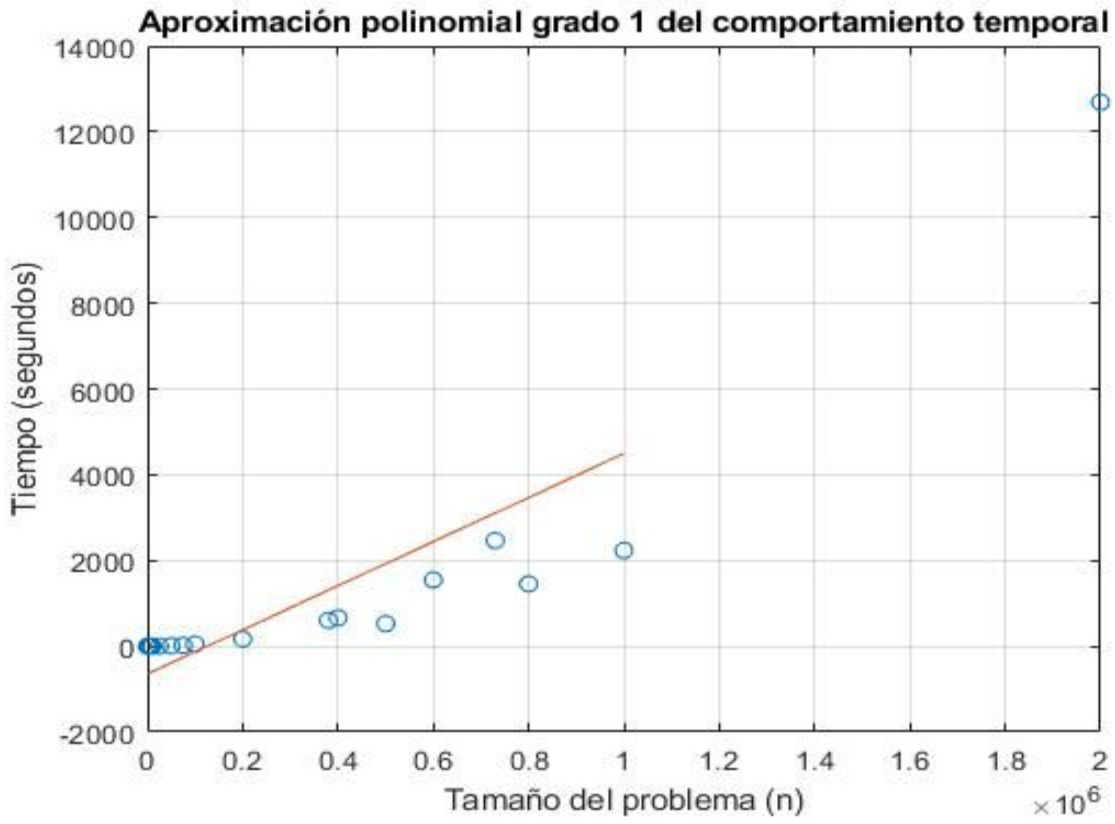


Aproximación polinomial de grado 8

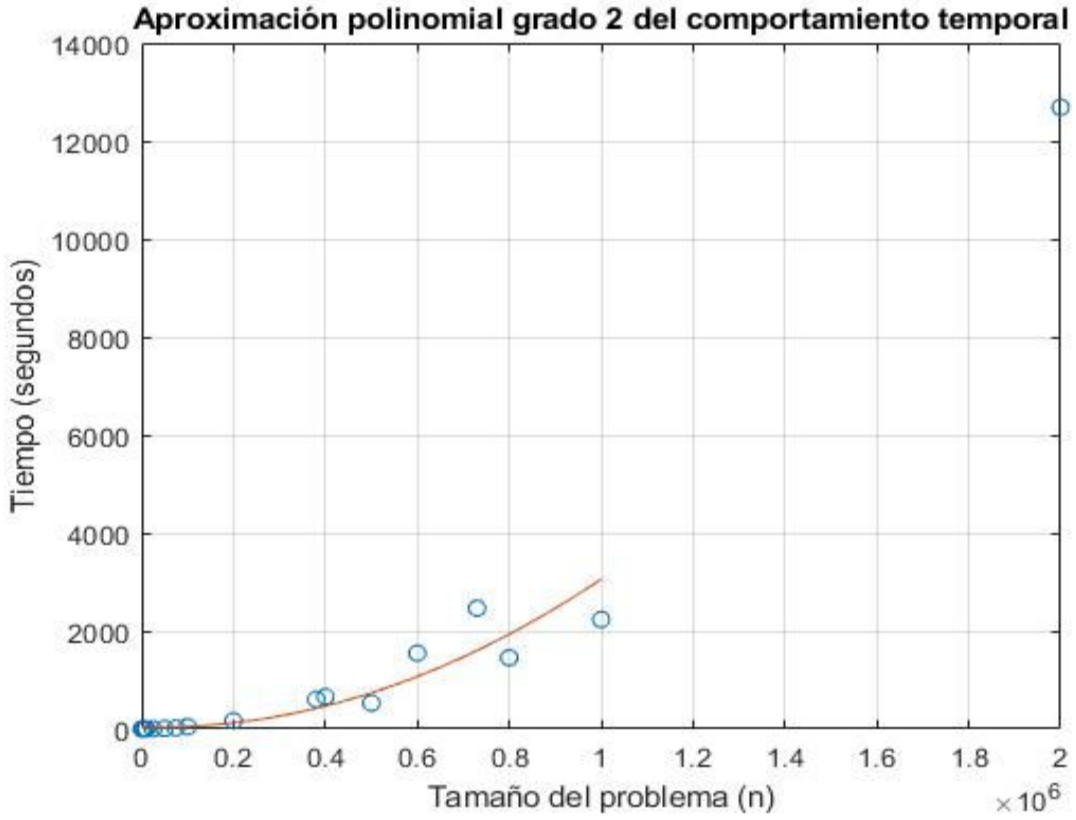


2.4.2. Ordenamiento Burbuja Optimizado

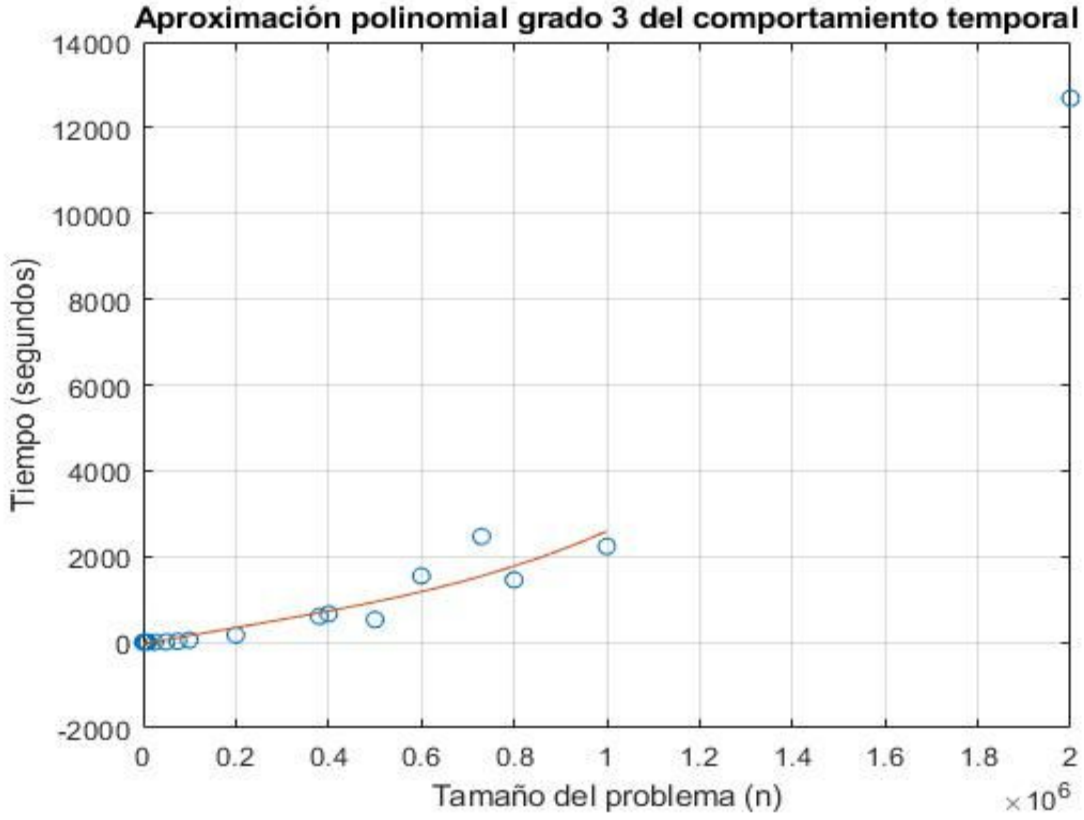
Aproximación polinomial de grado 1



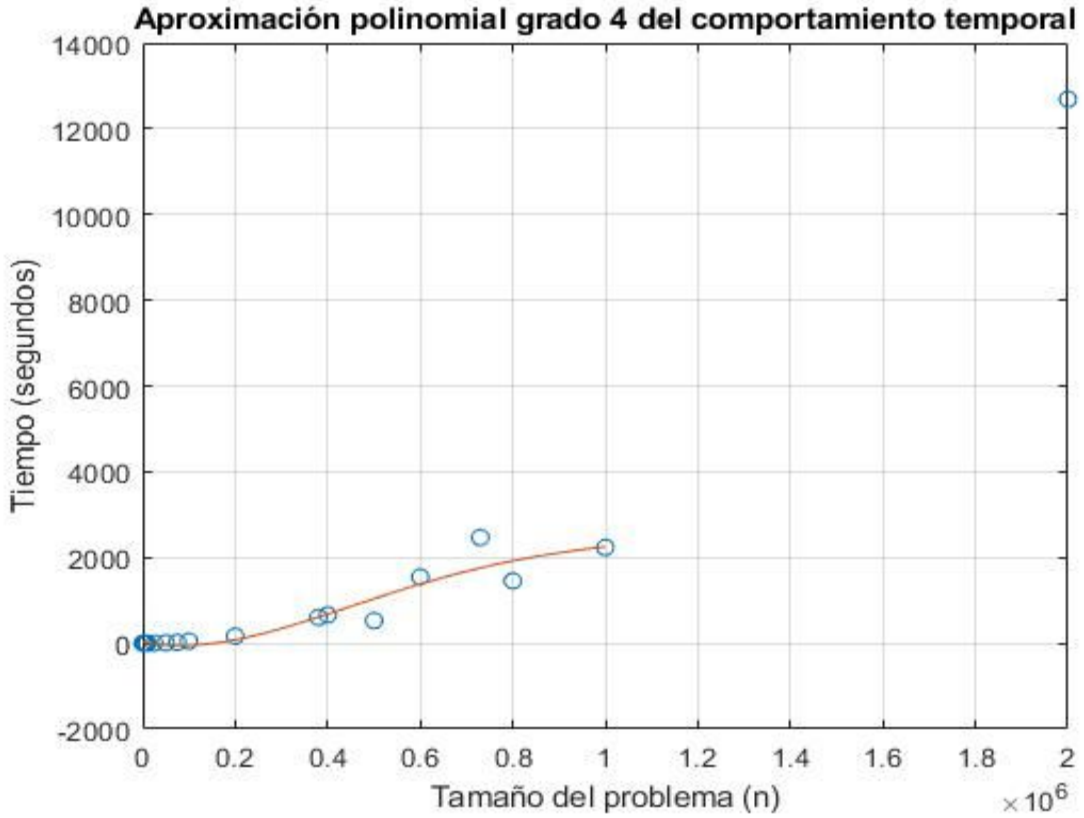
Aproximación polinomial de grado 2



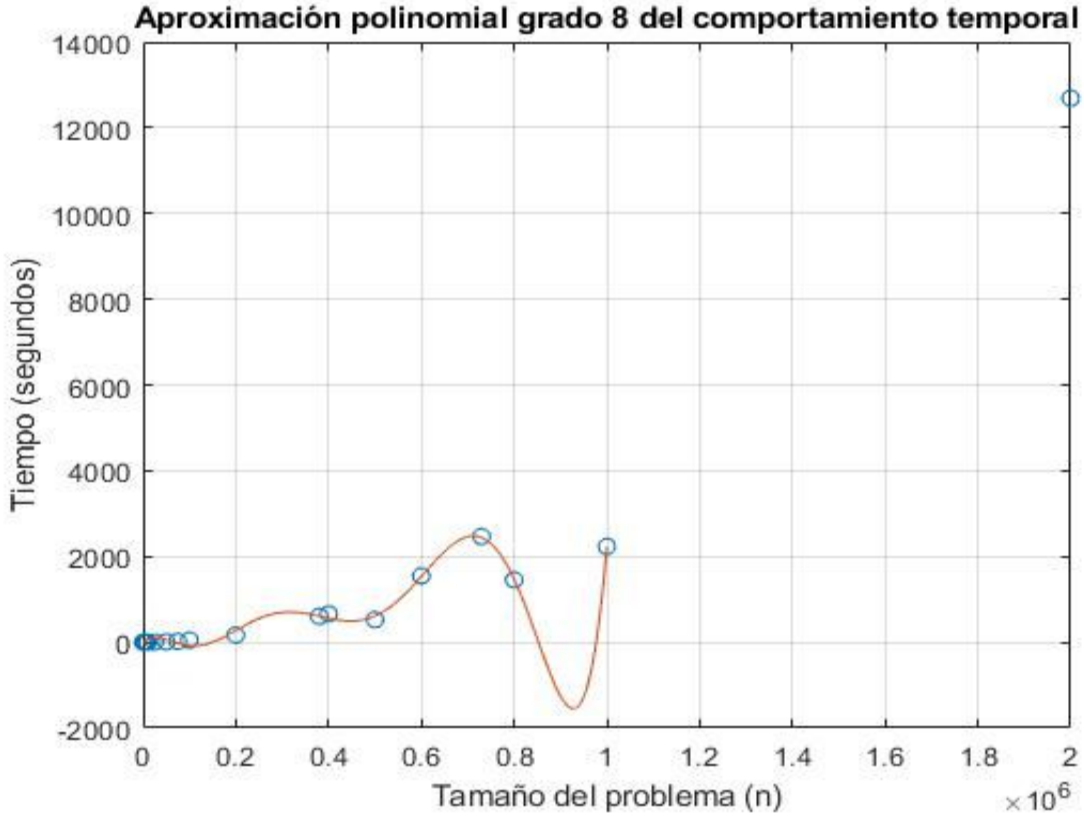
Aproximación polinomial de grado 3



Aproximación polinomial de grado 4

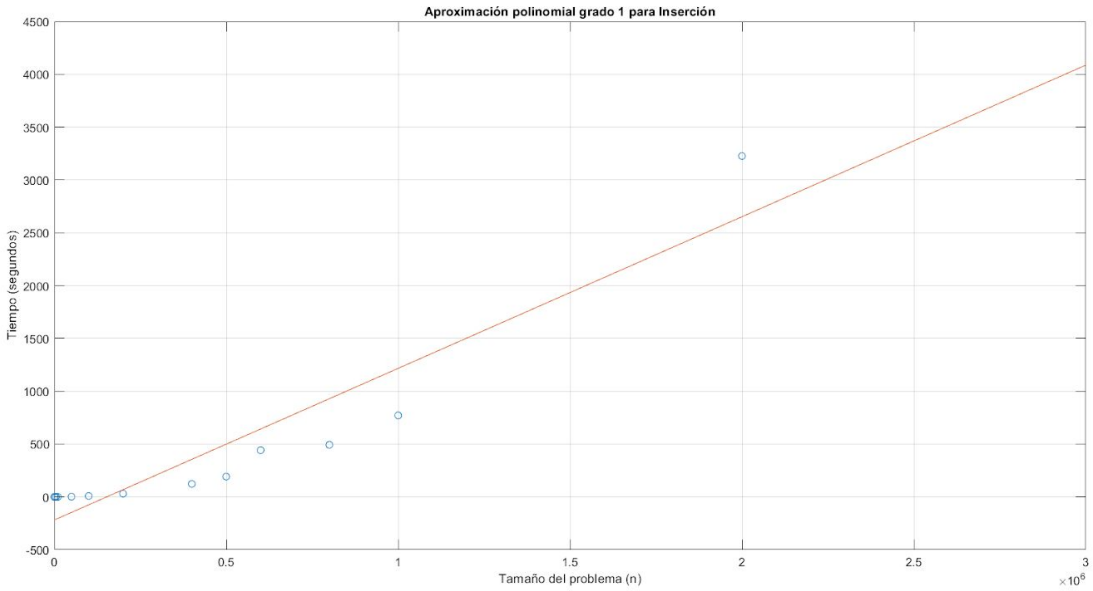


Aproximación polinomial de grado 8

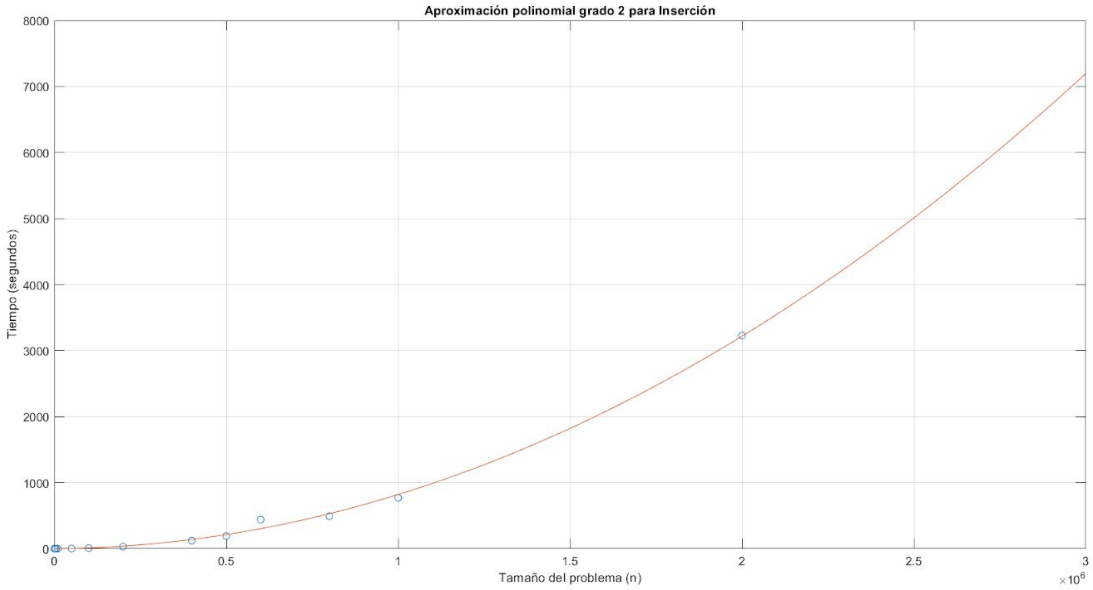


2.4.3. Ordenamiento de Inserción

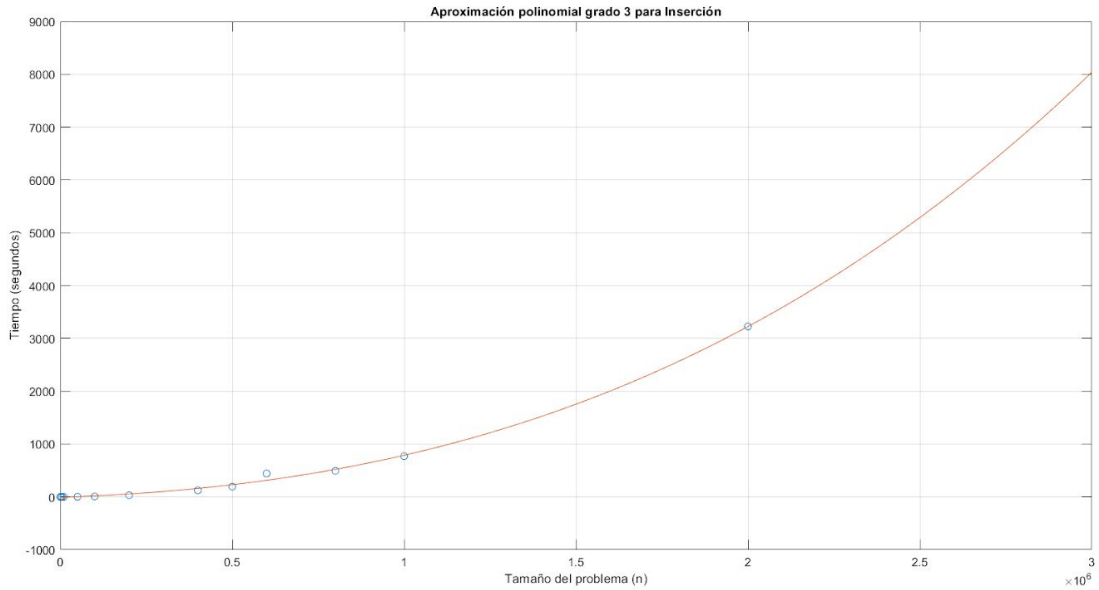
Aproximación polinomial de grado 1



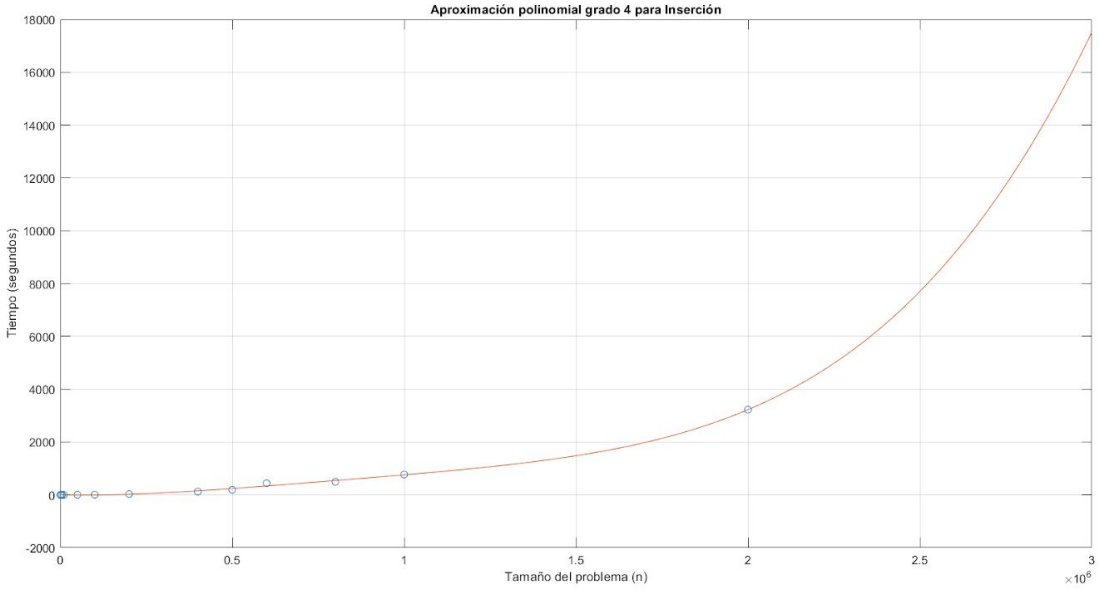
Aproximación polinomial de grado 2



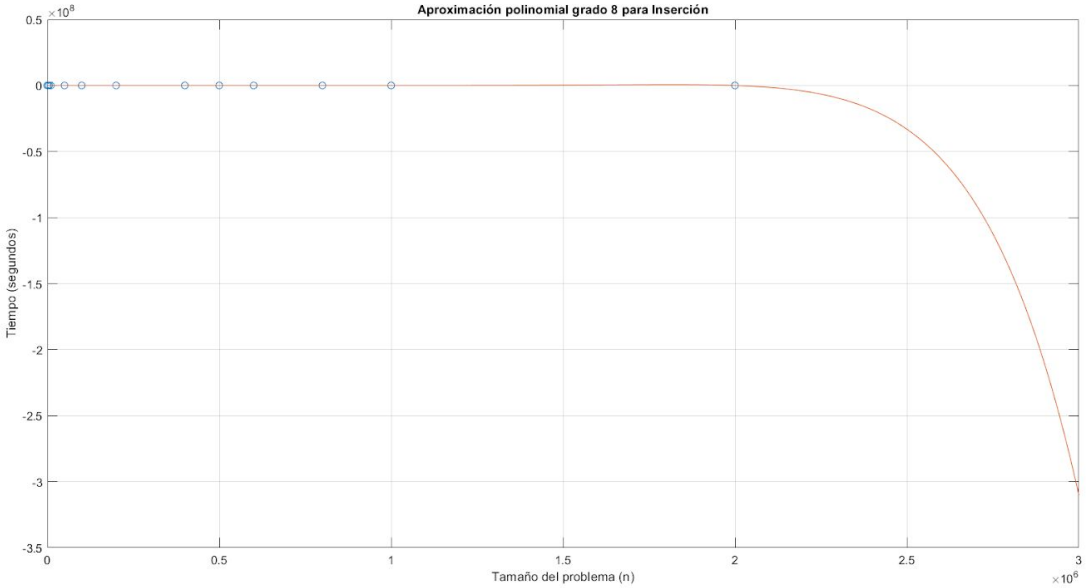
Aproximación polinomial de grado 3



Aproximación polinomial de grado 4

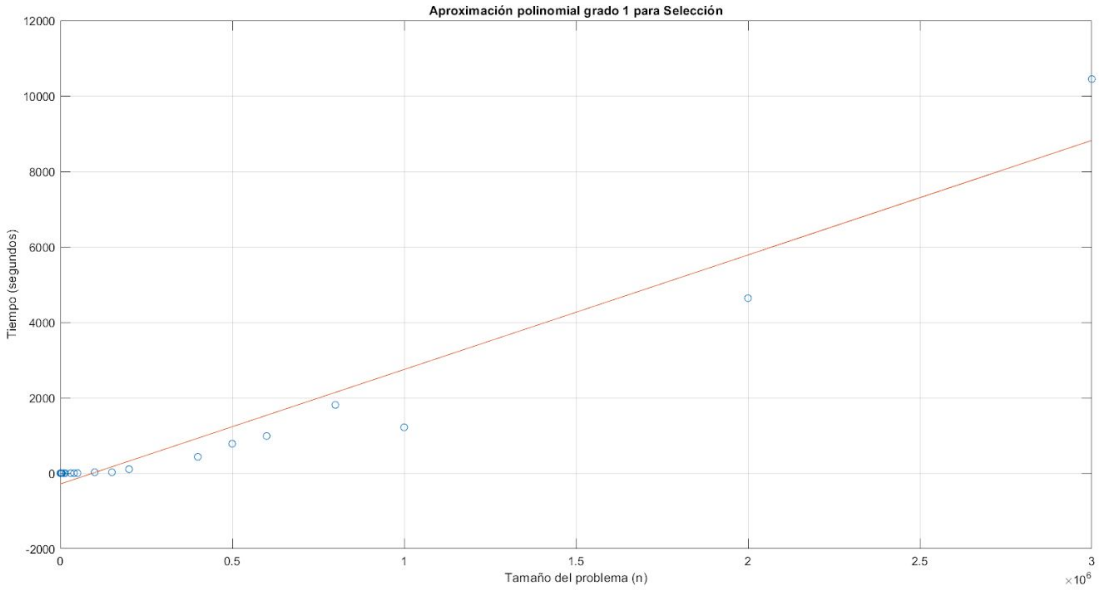


Aproximación polinomial de grado 8

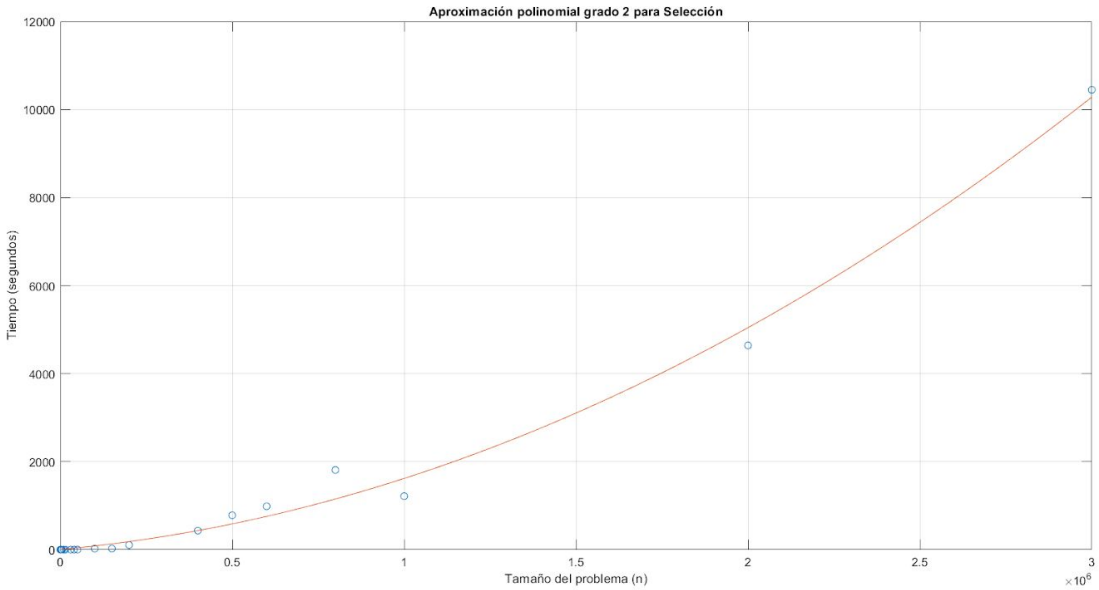


2.4.4. Ordenamiento de Selección

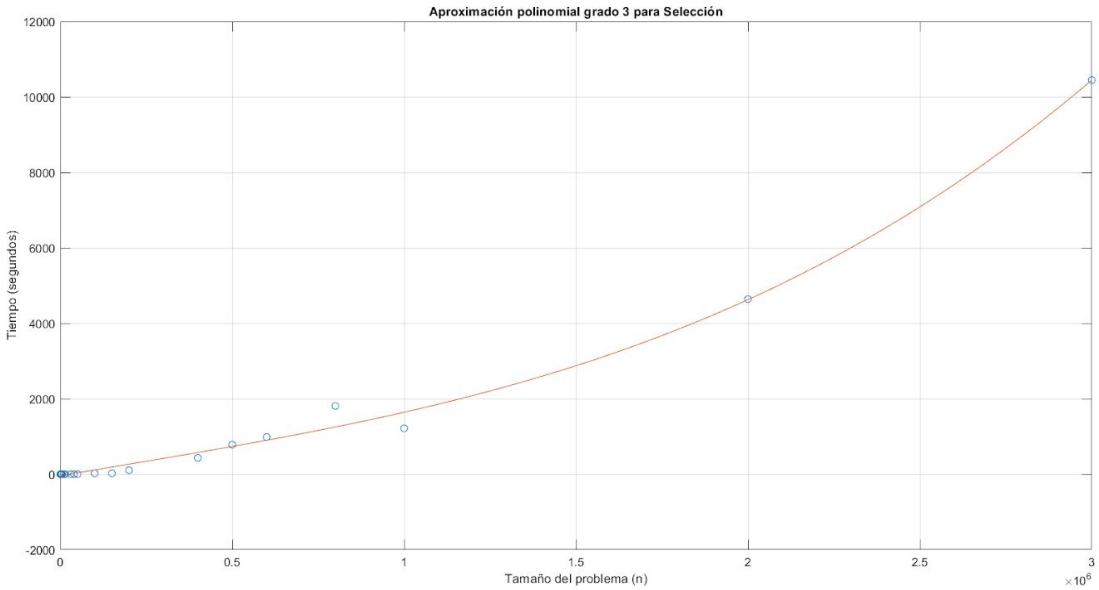
Aproximación polinomial de grado 1



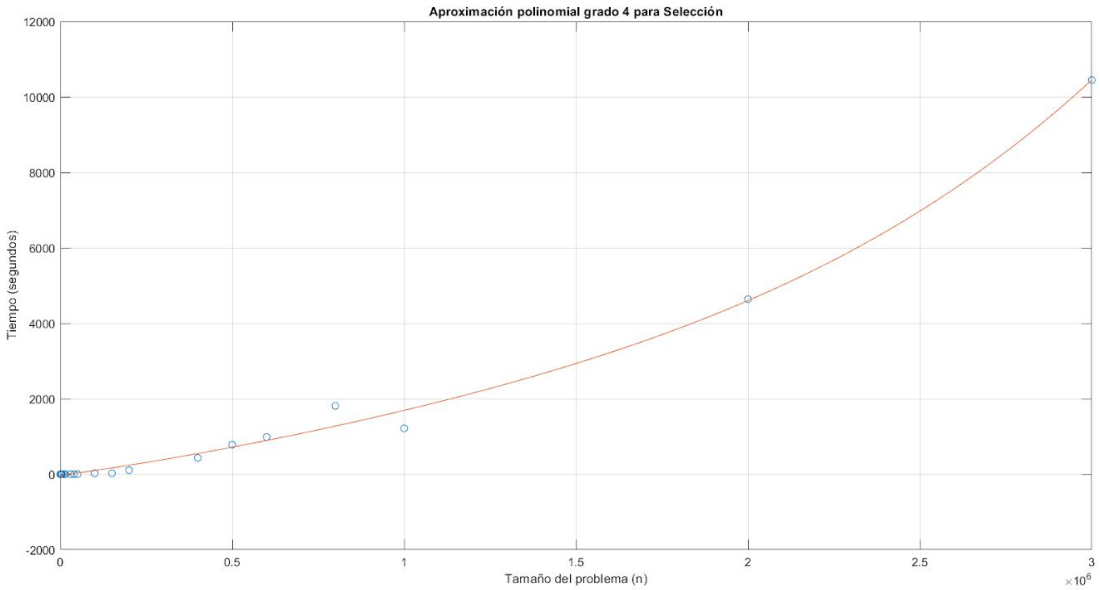
Aproximación polinomial de grado 2



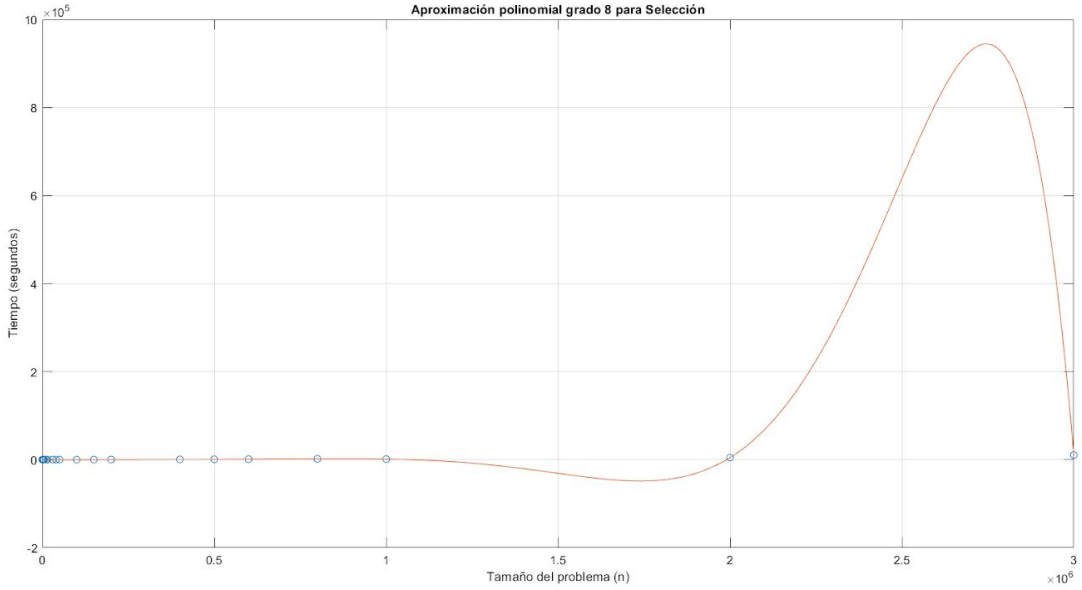
Aproximación polinomial de grado 3



Aproximación polinomial de grado 4

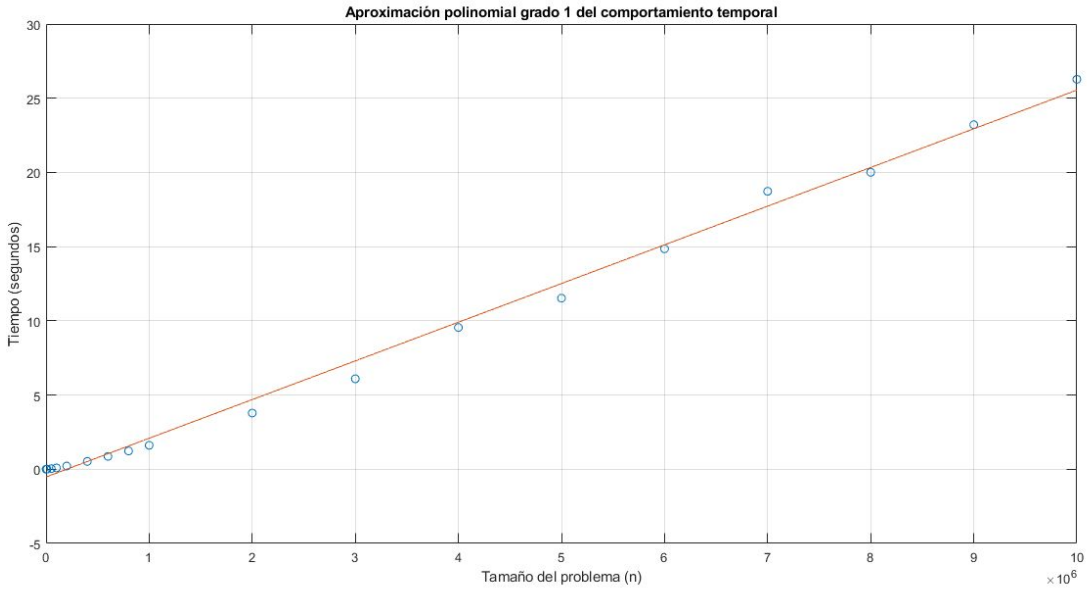


Aproximación polinomial de grado 8

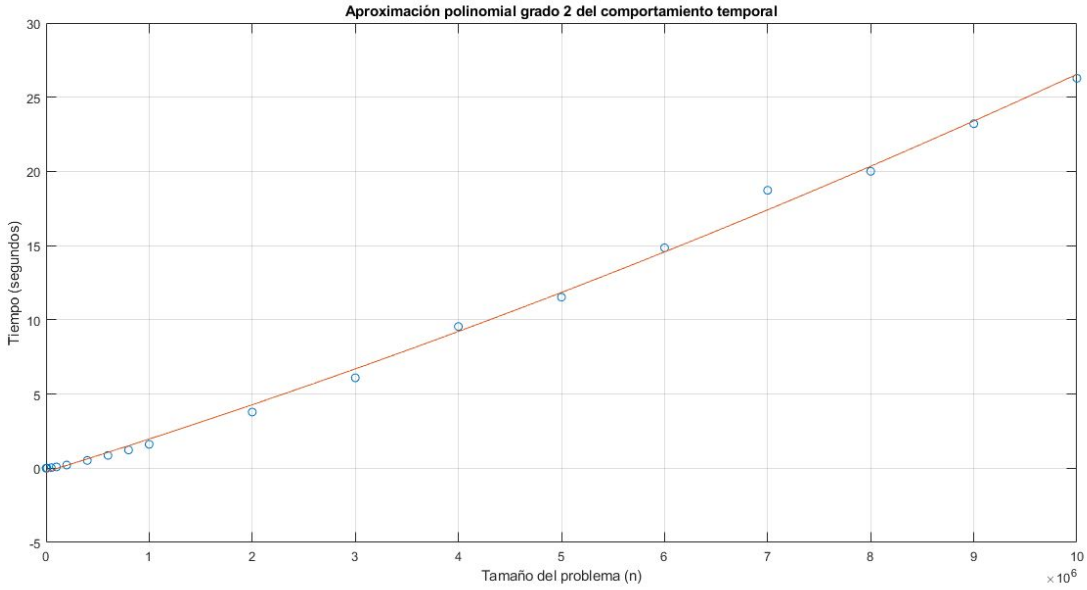


2.4.5. Ordenamiento con Árbol Binario de Búsqueda

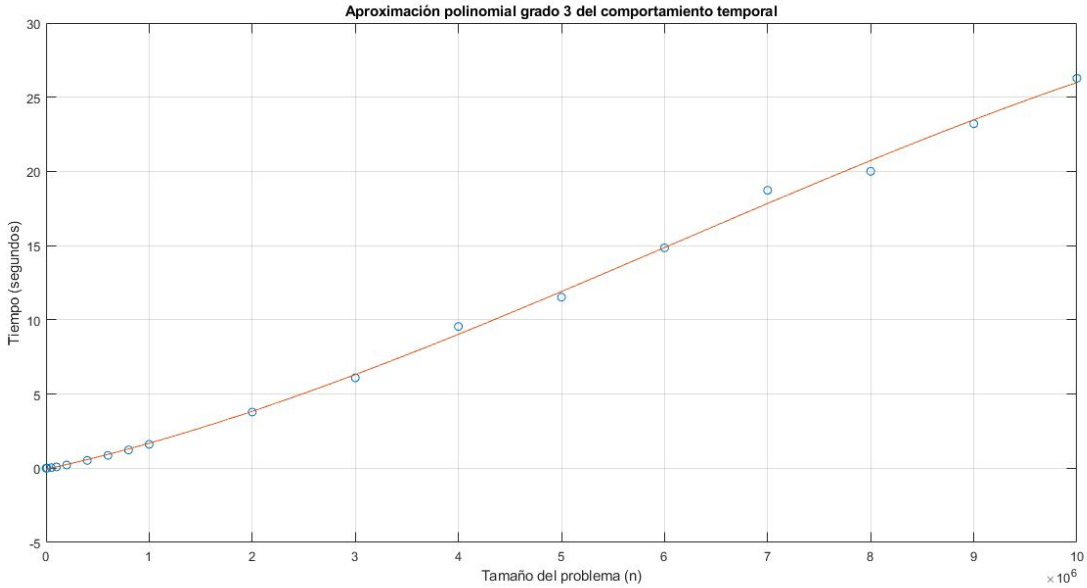
Aproximación polinomial de grado 1



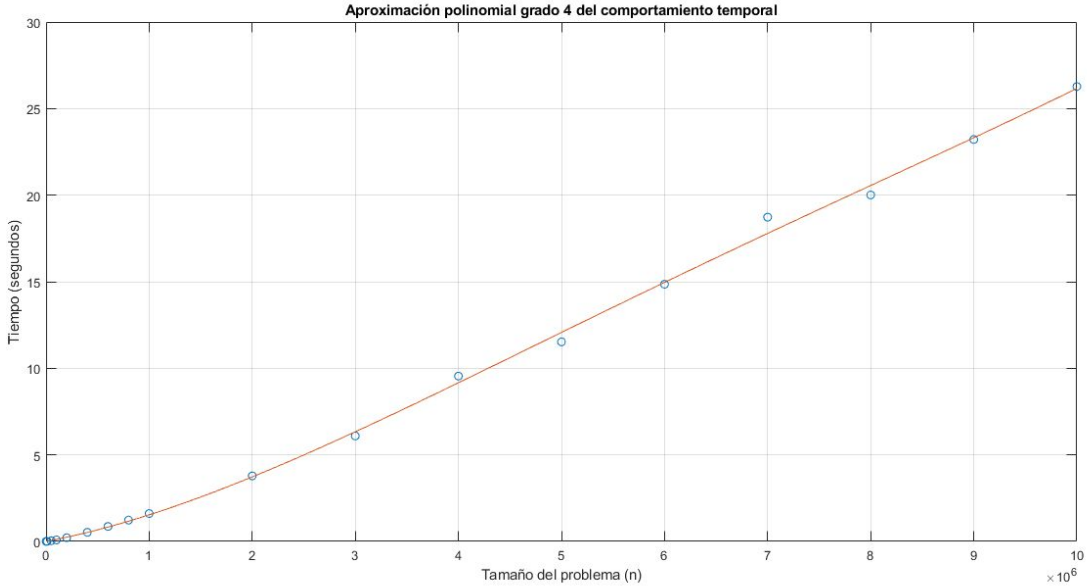
Aproximación polinomial de grado 2



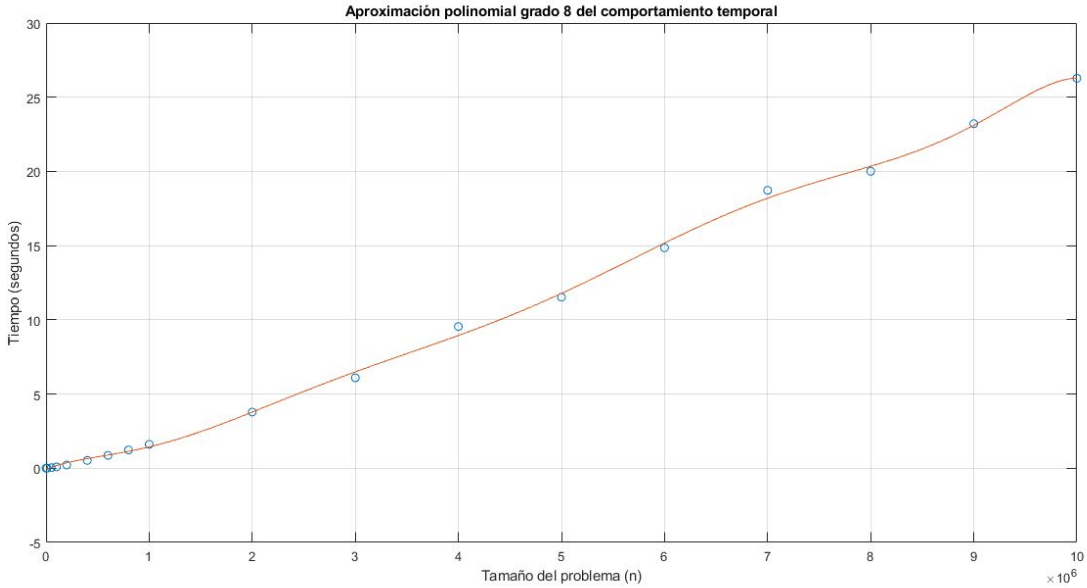
Aproximación polinomial de grado 3



Aproximación polinomial de grado 4

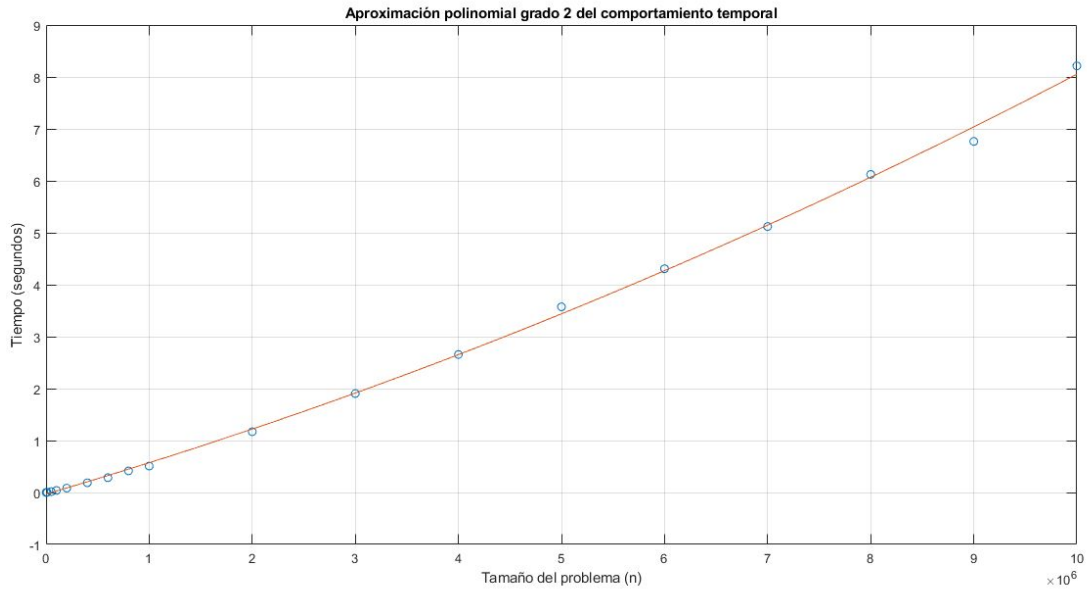


Aproximación polinomial de grado 8

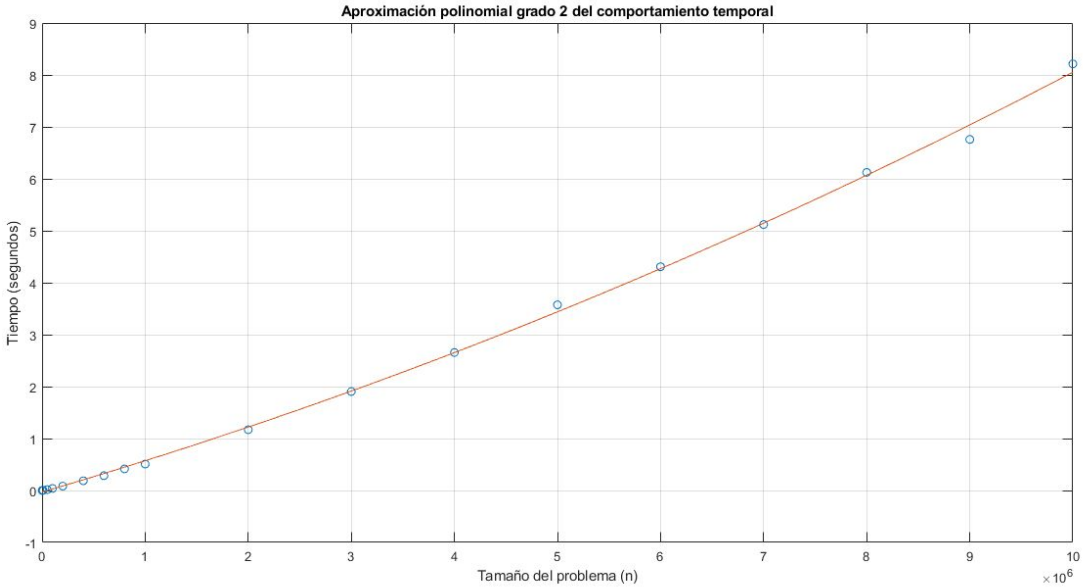


2.4.6. Ordenamiento Shell

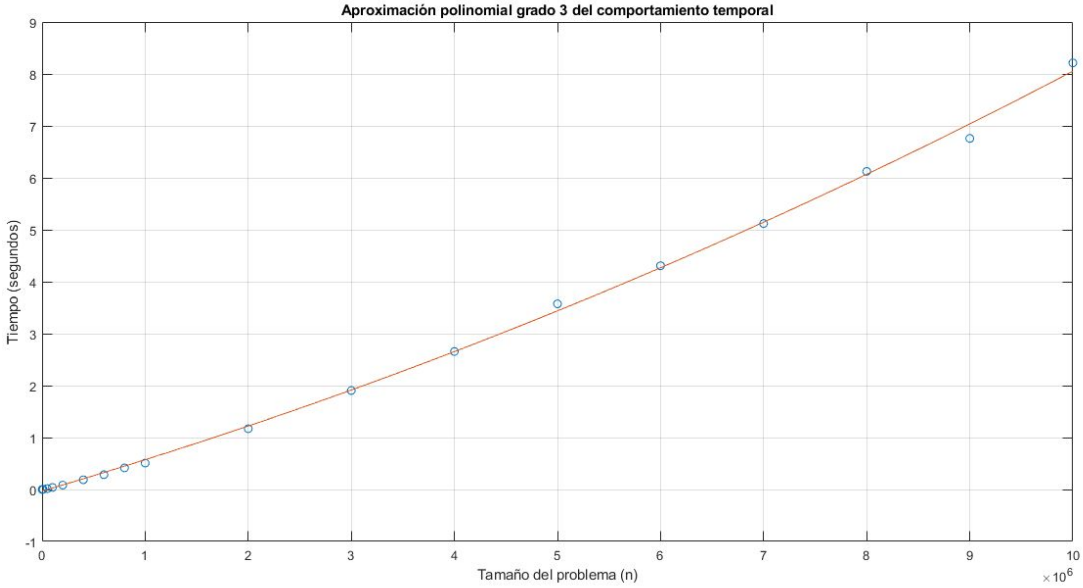
Aproximación polinomial de grado 1



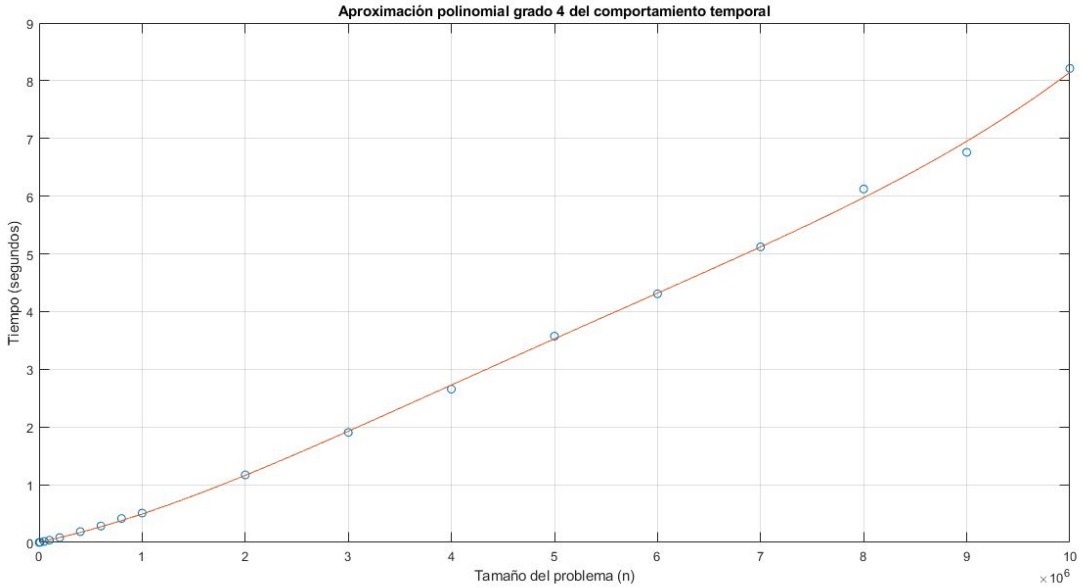
Aproximación polinomial de grado 2



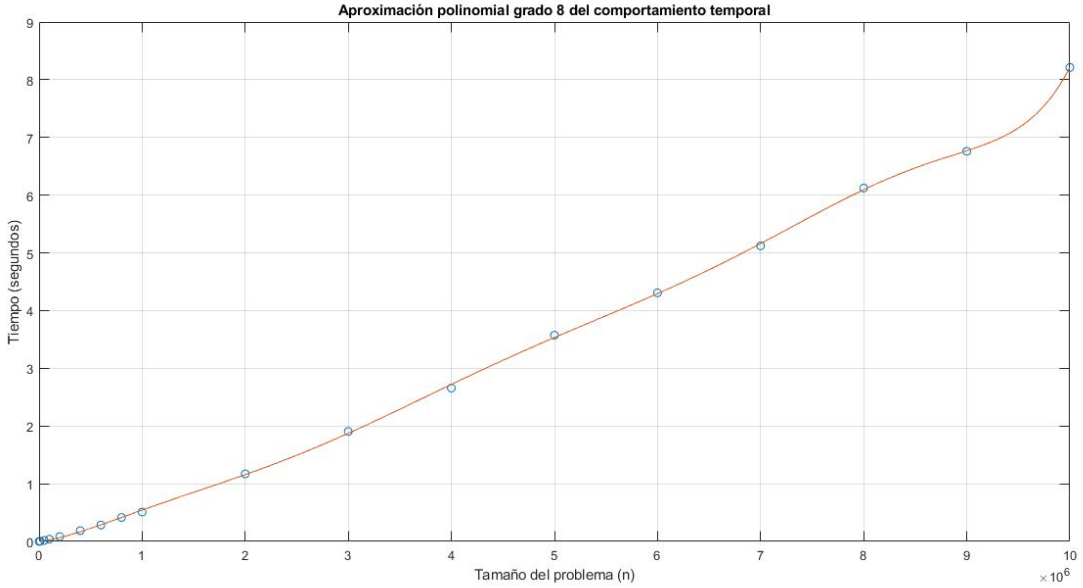
Aproximación polinomial de grado 3



Aproximación polinomial de grado 4



Aproximación polinomial de grado 8



2.5. Comparativa de las aproximaciones de la función complejidad encontrada para cada algoritmo

En esta sección compararemos los polinomios de cada algoritmo para determinar cuál es el que mejor describe a la función de tiempo.

2.5.1. Ordenamiento Burbuja

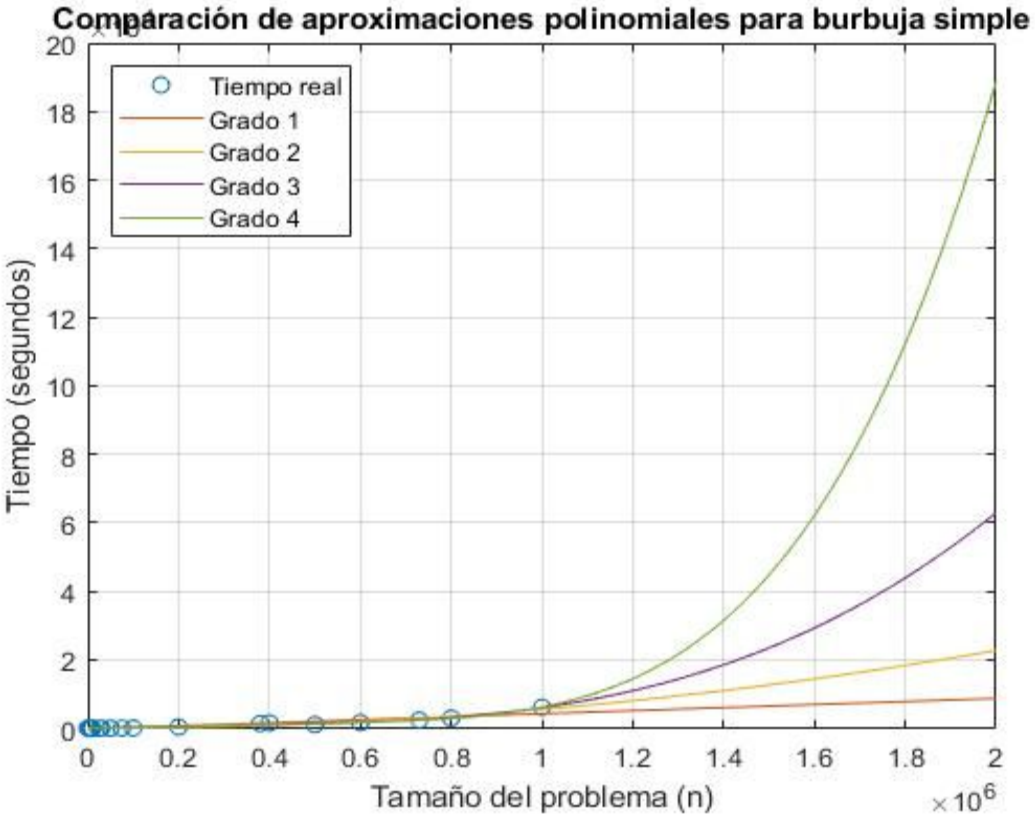


Imagen 2.7: Comparación de los polinomios del ordenamiento Burbuja

En la gráfica anterior se observa que los puntos de tiempo real son muy cercanos a cada una de los polinomios, pero el grado dos podría describir de mejor forma a la complejidad temporal del ordenamiento de burbuja porque conserva la propiedad de monótona creciente.

2.5.2. Ordenamiento Burbuja Optimizado

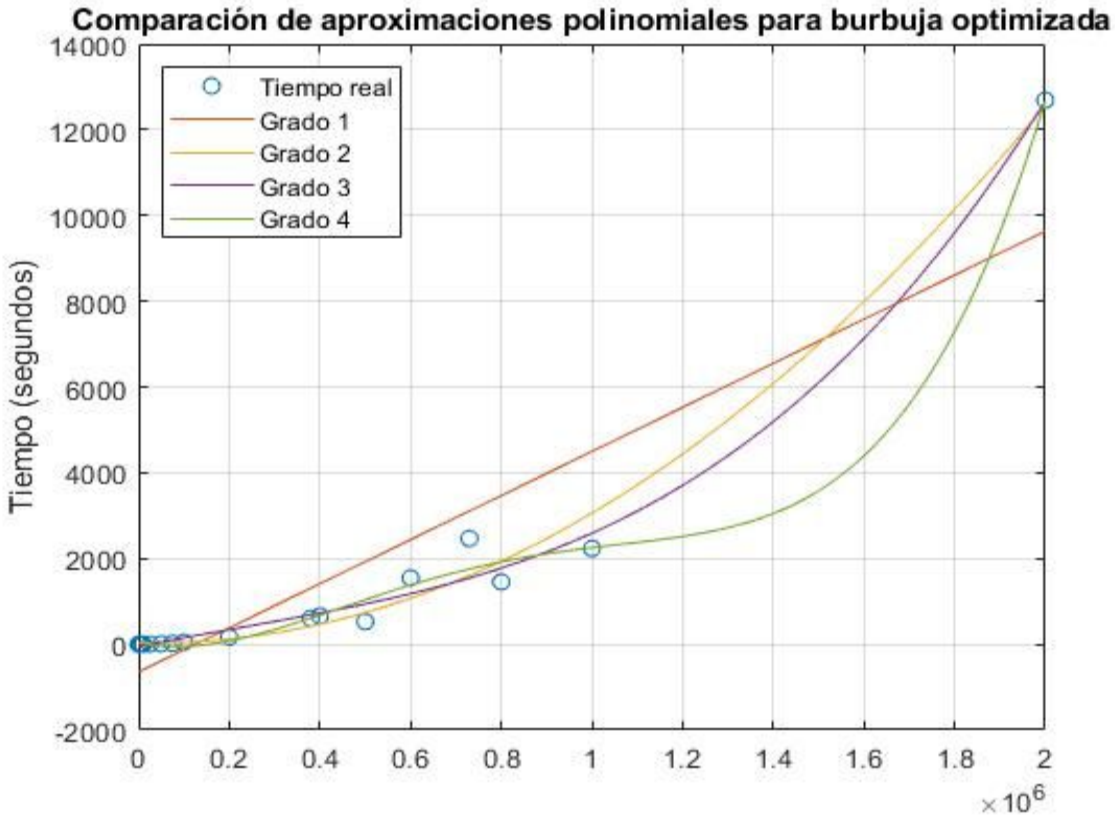


Imagen 2.8: Comparación de los polinomios del ordenamiento Burbuja optimizado

La gráfica de los polinomios de burbuja optimizada se puede visualizar que también el mejor polinomio es el de grado dos puesto que no puede ser la de grado uno porque no es la mejor representación del tiempo real, tampoco se pueden considerar como candidatas los polinomios de grado 2, 3, 4, 8 ya que estas presentan cambios de decrecimiento.

2.5.3. Ordenamiento de Inserción

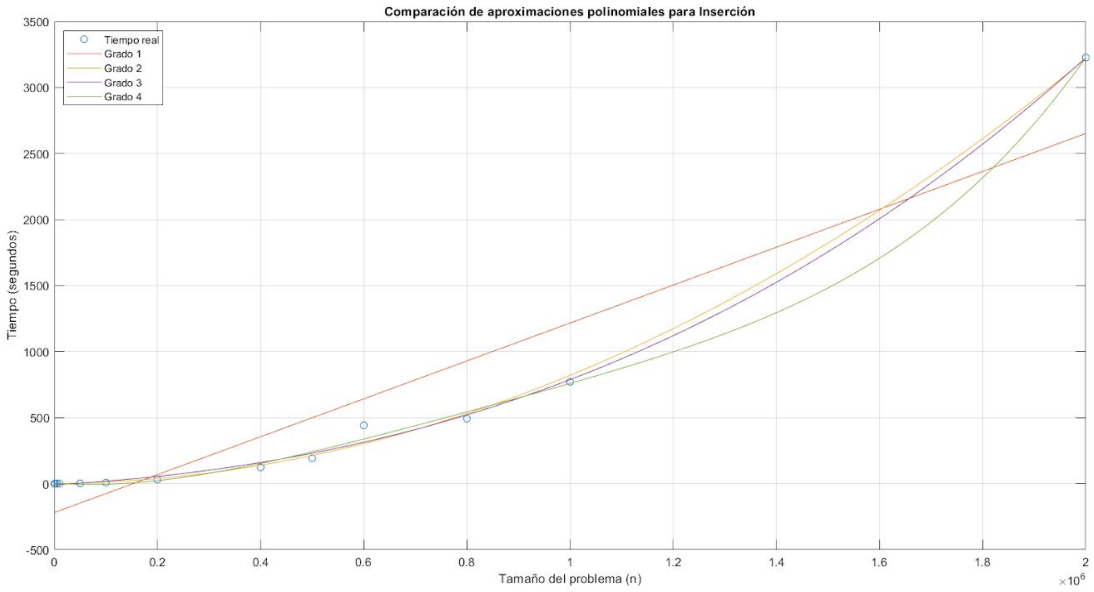


Imagen 2.9: Comparación de los polinomios del ordenamiento de Inserción

Podemos observar que un polinomio de grado 2 se mantiene ligeramente encima de todos los valores de $f(n)$, con la única excepción siendo para $n = 6 * 10^5$. Otras funciones son muy inestables (la pendiente de las rectas tangentes varía mucho) que si bien darían aproximación más exacta, están descritas por polinomios complicados; mientras que otras, como la recta, se alejan demasiado de otros puntos

2.5.4. Ordenamiento de Selección

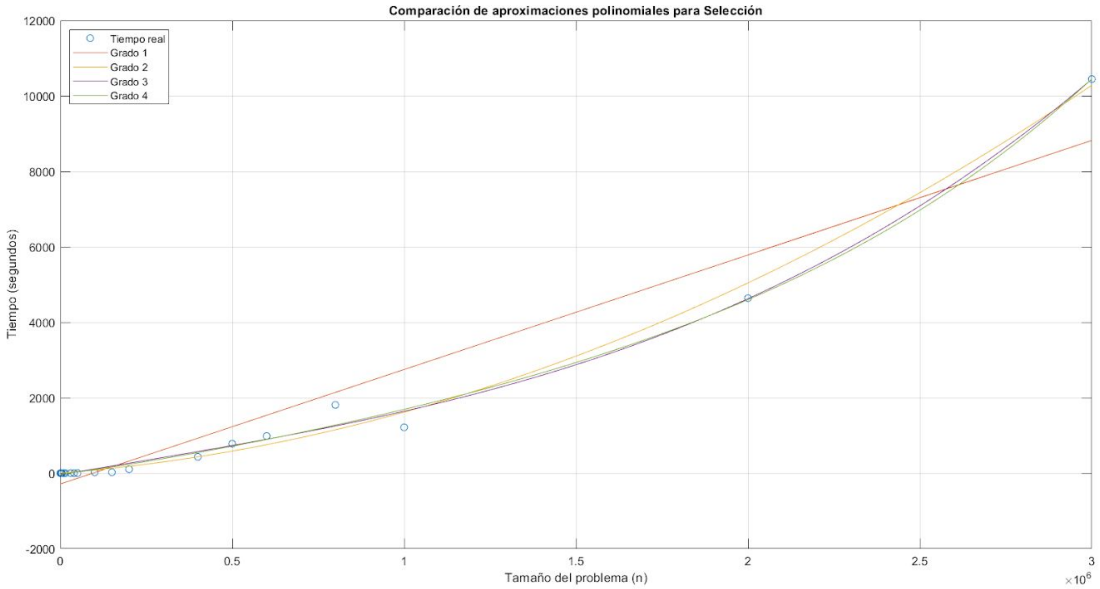


Imagen 2.10: Comparación de los polinomios del ordenamiento de Selección

En esta gráfica podemos observar que los polinomios para Selección, a diferencia de Inserción, no están ni sobre o debajo de los valores de Tiempo Real, en cambio, alternan. Sin embargo, podemos escoger nuevamente al polinomio de grado dos porque las funciones de mayor grado se alejan de los primeros valores, y la función lineal no es una buena representación. El polinomio cuadrático se mantiene constante a distancias moderadas de los valores del Tiempo contra Tamaño de Problema.

2.5.5. Ordenamiento con Árbol Binario de Búsqueda

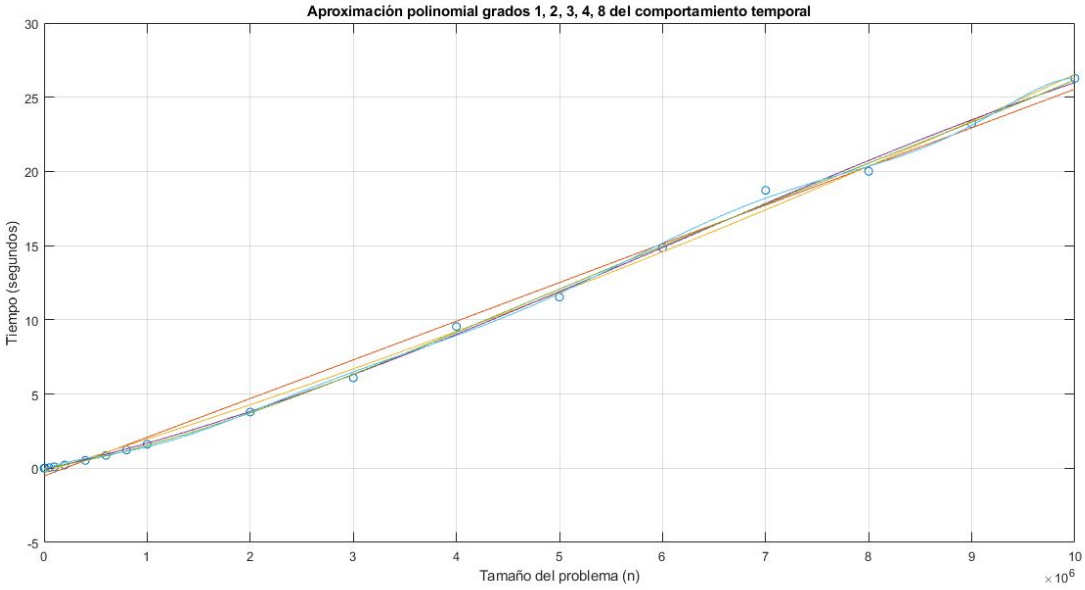


Imagen 2.11: Comparación de los polinomios del ordenamiento ABB

Como se ve en la gráfica, todas las aproximaciones polinomiales con números muy grandes, resulta que son bastante similares, sin embargo, como un algoritmo debe ser monótonamente creciente respecto al tamaño del problema, se puede utilizar un polinomio de grado 2, en otras palabras, una parábola, objeto que tampoco cambia drásticamente su trayectoria.

2.5.6. Ordenamiento Shell

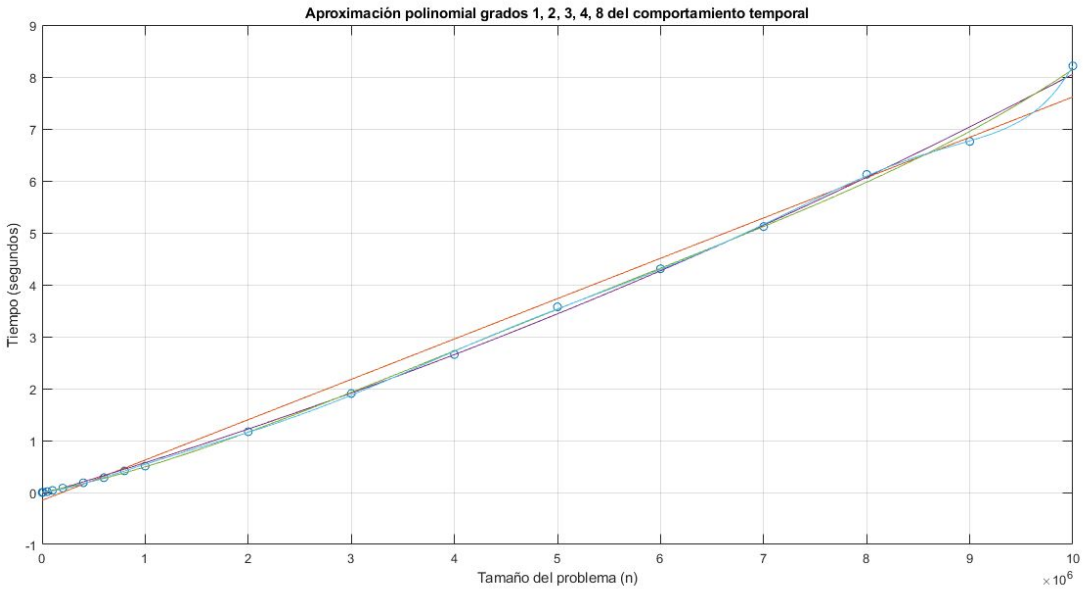


Imagen 2.12: Comparación de los polinomios del ordenamiento Shell

En ésta gráfica, las aproximaciones son muy similares, sobre todo cuando las probamos con tamaños de problema muy grandes, sin embargo, la complejidad temporal de un algoritmo de ordenamiento debe ser monótonamente creciente, por lo tanto un polinomio grado dos es más que suficiente para aproximar los resultados estimados.

2.6. Gráfica comparativa de los modelos de complejidad de los seis algoritmos

Con las aproximaciones polinomiales observamos qué polinomio representa mejor a la función de tiempo real $f(n)$, dichas ecuaciones se muestran en esta sección.

- Polinomio de ordenamiento burbuja:

$$P(x) = 43.65x^2 - 3.45 * 10^{-4}x + 5.84 * 10^{-9}$$

- Polinomio de ordenamiento burbuja optimizada:

$$P(x) = 40.87x^2 - 2 * 10^{-4}\, x$$

- Polinomio de ordenamiento por inserción:

$$P(x) = 7.9 * 10^{-10}x^2 + 2.93 * 10^{-5}x + 1.4$$

- Polinomio de ordenamiento por selección

$$P(x) = 9.07 * 10^{-10}x^2 + 7.06 * 10^{-4}x + 6.61$$

- Polinomio de ordenamiento con Árbol Binario de Búsqueda:

$$P(x) = \ -0.24x^2 \ + \ 2.15 * 10^{-6}x \ + \ 5.2 * 10^{-13}$$

- Polinomio de ordenamiento Shell:

$$P(x) = \ -- 2.9 * 10^{-2}x^2 + 5.8 * 10^{-7}x + 2.3 * 10^{-13}$$

El graficador Geogebra nos muestra una comparación entre el comportamiento de los polinomios. Aún después de ajustar la escala, no es posible observar las parábolas que forman los algoritmos Burbuja, Burbuja optimizado, Inserción y Selección. Las imágenes 2.14 y 2.15 muestran acercamientos para una mejor visualización

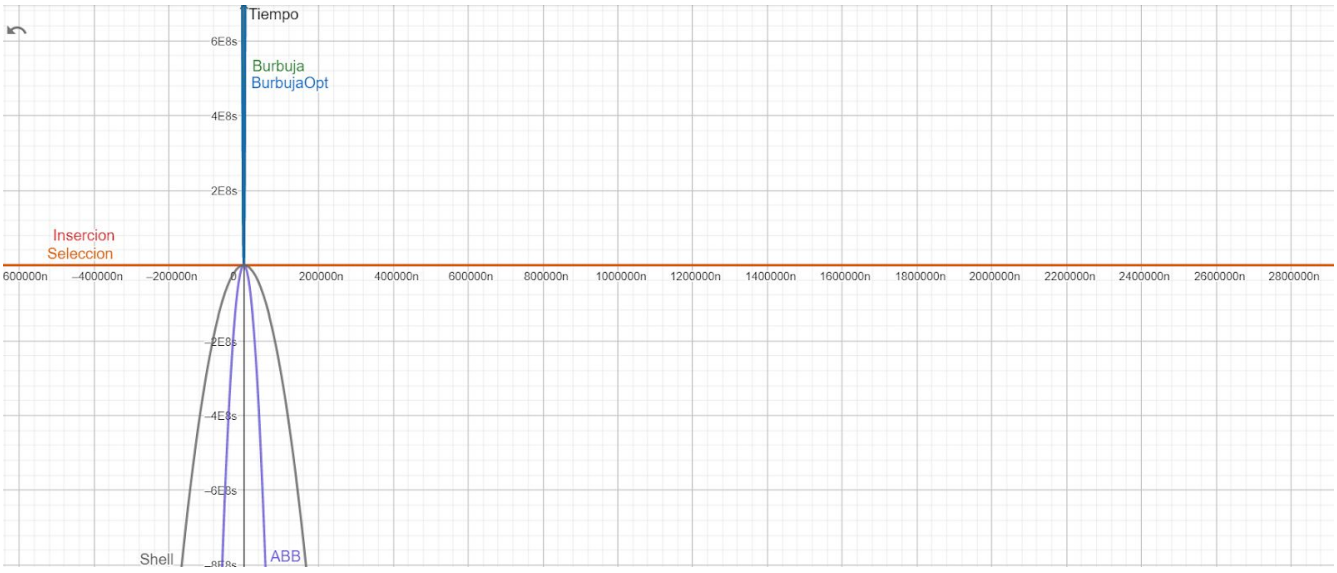


Imagen 2.13. Comparación de polinomios que describen a los algoritmos de ordenamiento

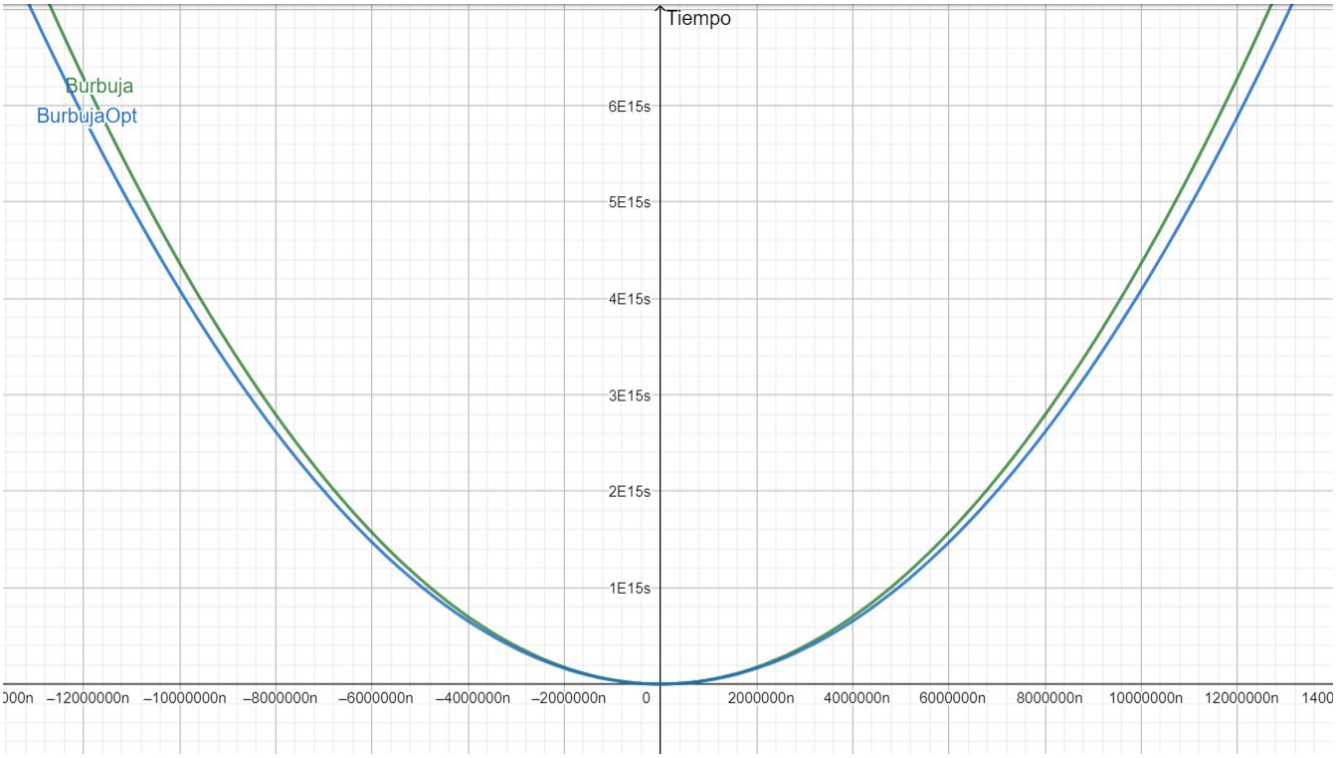


Imagen 2.14. Acercamiento de la gráfica de los polinomios para los algoritmos de Burbuja

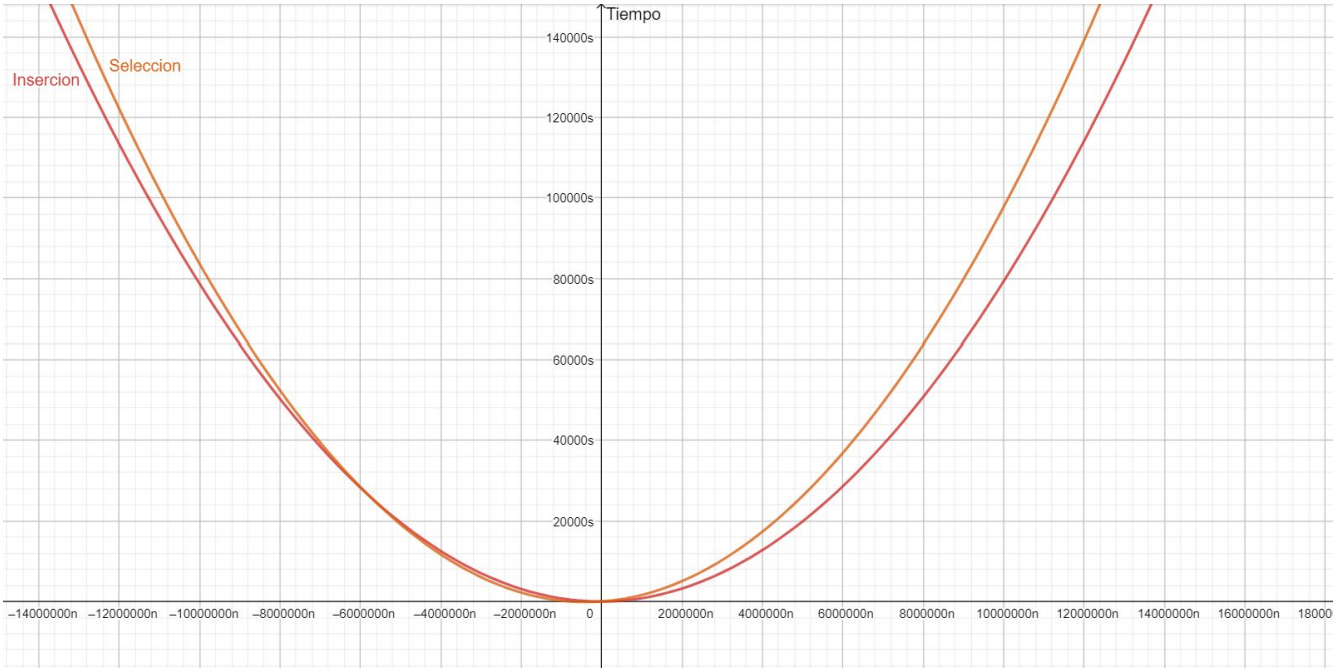


Imagen 2.15. Acercamiento de la gráfica de los polinomios para Inserción y Selección

Sin embargo, la imagen 2.13 aún nos permite observar el comportamiento de las funciones mencionadas. Vemos que el ordenamiento Burbuja y Burbuja optimizada crecen muy rápido, mucho más que Inserción y Selección.

Dos casos notables son los de ABB y Shell, que muestran gráficas reflejadas en el eje n. Esto no significa que los tiempos sean negativos, sino que el crecimiento de $f(n)$ es menor, y aún para valores de n muy grandes, el tiempo será mucho menor que con cualquier otro algoritmo de ordenamiento.

2.7. Cálculos a priori de tiempo real de cada algoritmo para determinadas n

Gracias a los polinomios, tenemos una ecuación para encontrar valores de Tiempo real para cualquier valor de n **sin necesidad de ejecutar el programa**. Probaremos cada algoritmo con cinco valores de n mayores a 10^7

2.7.1. Ordenamiento Burbuja

Tamaño de problema (n)	Tiempo real	Conversiones
15000000	9.82×10^{15} s	3,114,218.67 siglos
20000000	1.75×10^{16} s	5,536,212.58 siglos
500000000	1.09×10^{19} s	3,460,172,501.27 siglos
1000000000	4.36×10^{19} s	13,840,690,005.07 siglos
5000000000	1.09×10^{21} s	346,017,250,126.84 siglos

2.7.2. Ordenamiento Burbuja Optimizada

Tamaño de problema (n)	Tiempo real	Conversiones
15000000	9.2×10^{15} s	2,915,924.66 siglos
20000000	1.63×10^{16} s	5,183,866.06 siglos
500000000	1.02×10^{17} s	3,239,916,286.15 siglos
1000000000	4.09×10^{19} s	12,959,665,144.6 siglos
5000000000	1.02×10^{21} s	323,991,628,614.92 siglos

2.7.3. Ordenamiento por Inserción

Tamaño de problema (n)	Tiempo real	Conversiones
15000000	1.64×10^5 s	2 días, 1h 29’ 50’’
20000000	3.17×10^5 s	3 días, 15h 56’ 27’’
500000000	1.97×10^8 s	6.26 años
1000000000	7.9×10^8 s	25.05 años
5000000000	1.97×10^{10} s	6.26 siglos

2.7.4. Ordenamiento por Selección

Tamaño de problema (n)	Tiempo real	Conversiones
15000000	2.15×10^5 s	2 días, 11h 37’ 51’’
20000000	3.77×10^5 s	4 días, 8h 42’ 6’’
500000000	2.27×10^8 s	7.2 años
1000000000	9.08×10^8 s	28.78 años
5000000000	2.27×10^{10} s	7.19 siglos

2.7.5. Ordenamiento con Árbol Binario de Búsqueda

Tamaño de problema (n)	Tiempo real	Conversiones
15000000	43.85 s	44’’
20000000	63.66 s	1’ 3’’
500000000	1.41×10^4 s	3h 54’ 37’’
1000000000	5.41×10^4 s	15h 2’ 35’’

5000000000	1.31 x 10^6 s	15 días, 14h 6’ 15’’

2.7.6. Ordenamiento Shell

Tamaño de problema (n)	Tiempo real	Conversiones
15000000	13.83 s	14’’
20000000	20.76 s	21’’
500000000	6.04 x10^3 s	1h 40’ 40’’
1000000000	2.36 x 10^4 s	6h 32’ 59’’
5000000000	5.78 x 10^5 s	6 días, 16h 31’ 36’’

Es claro que en esta sección se verifica la importancia del análisis de algoritmos a priori. Si las actividades hubieran requerido los tiempos a posteriori del ordenamiento de 15 millones de números, simplemente no hubiera sido posible sin algún tipo de computadora especial. Pero realizar buenas pruebas de algoritmos para generar modelos matemáticos es una alternativa muy viable para obtener resultados aceptables.

2.8. Cuestionario

- ¿Cuál de los algoritmos es más fácil de implementar?

El algoritmo más fácil de implementar es el ordenamiento burbuja, porque es un procedimiento fácil de memorizar.

- ¿Cuál de los cinco algoritmos es más difícil de implementar?

El algoritmo con mayor dificultad de implementación es el ordenamiento con un árbol binario de búsqueda, ya que es el único que necesita una estructura de datos.

- ¿Cuál algoritmo tiene menor complejidad temporal?

Todos los algoritmos de ordenamiento en el peor de los casos tienen complejidad temporal $O(n^2)$, sin embargo, el ordenamiento Shell fue el algoritmo que ordenó los elementos en menor tiempo, ya que su mejor complejidad temporal y su complejidad temporal media, es O(n log n).

- ¿Cuál algoritmo tiene mayor complejidad temporal?

Todos los algoritmos de ordenamiento en el peor de los casos tienen complejidad temporal $O(n^2)$, algunos pueden ordenar números en O(n log n), sin embargo el ordenamiento burbuja ordena siempre en complejidad temporal cuadrática, por lo tanto, es el más lento.

- ¿Cuál algoritmo tiene menor complejidad espacial? ¿Porqué?

A excepción del ordenamiento ABB, todos los algoritmos tienen complejidad espacial F(n), porque se utiliza un arreglo de enteros de tamaño n (y algunas variables adicionales de tamaño 1), y ese mismo arreglo se modifica sin necesidad de utilizar más memoria.

- ¿Cuál algoritmo tiene mayor complejidad espacial? ¿Porqué?

Como se mencionó, el ordenamiento ABB es el único que necesita memoria adicional; en este caso, requiere n espacios de memoria adicionales al tamaño del arreglo a ordenar, haciendo su complejidad F(2n)

- ¿El comportamiento experimental de los algoritmos era el esperado? ¿Porqué?

Sí, se sospechaba que los valores de tiempo de aquellos algoritmos que involucran ciclos anidados serían elevados, y que habría un punto en el que no sería posible continuar computando para valores de n grandes.

- ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fué?

Aunque, en un comienzo, todas las pruebas se habían realizado en diferentes equipos, el equipo realizó las pruebas por segunda vez en un solo equipo de cómputo para obtener resultados consistentes. La máquina tiene las siguientes especificaciones:

- ❏ Procesador i5 6200u a 2.8Ghz
- ❏ 8gb de memoria RAM DDR3 a 1333 mhz
- ❏ 240gb SSD
- ❏ GPU Intel integrada
- ❏ Sistema Operativo Zorin OS 15.1 (Ubuntu 16.04)

- ¿Facilitó las pruebas mediante scripts u otras automatizaciones? ¿Cómo lo hizo?

Utilizamos scripts bash para realizar pruebas en cada uno de los algoritmos con cada una de las n dadas por el profesor. Los scripts inicializan el archivo ejecutable, se envía n como argumento de línea de comando, y mediante redirección de salida, los resultados se guardan en un archivo .txt diferente para cada test. Los nombres de dichos archivos tienen ligeras variaciones para cada algoritmo, pero todos seguían un formato similar al siguiente:

Nombre de archivo: <Algoritmo de ordenamiento><n>.txt

- ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?
- ❏ Dado que la duración de cada algoritmo de ordenamiento varía de unas cuantas milésimas de segundo a varias horas, se sugiere comenzar con las pruebas lo más pronto posible para evitar rezagos. Para el caso de los algoritmos Burbuja, Inserción y Selección, esto también permitiría continuar tests con valores de n más grandes y conseguir más información.
- ❏ Es importante que desde el inicio de la práctica se establezca a una persona que pueda realizar las pruebas en su computadora para poder trabajar en un ambiente controlado como se menciona previamente, y no en máquinas con especificaciones distintas que perjudican a la práctica.

3. Anexo

3.1. Ordenamiento Burbuja

Instrucciones de compilación

1. Compilación

El código tiene comandos únicamente funcionales en Linux, por lo tanto, para poder compilar y ejecutar el código, es necesario estar en un entorno Linux. En una terminal, ejecutar el siguiente comando: `gcc main.c tiempo.c -o`

2. Ejecución

Para poder ejecutar el código en Linux, se utilizará el siguiente comando `./bin <n> < input.txt>`, donde <n> se reemplaza por el número de números a ordenar, asumiendo que el archivo .txt que contiene los 10 millones de números de prueba se llama `input.txt` y se encuentra en el mismo directorio de los códigos.

Código

```
//*****************************************************************
//Quintana Martinez Erick
//Curso: Análisis de algoritmos
//(C) Noviembre 2020
//ESCOM-IPN
//Medición de tiempo en C y recepción de parámetros en C bajo UNIX del
algoritmo de burbuja simple
//versión 1.3
//*****************************************************************

//*****************************************************************
//LIBRERÍAS INCLUIDAS
//*****************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"

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

      //Variables del main
      //*************************************************************
```

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

	//*****************************************************************

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


	//Si no se introducen exactamente 2 argumentos (Cadena de ejecución
y cadena=n)
	if (argc!=2)
	{
		printf("\nIndique el tamaño 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]);
	}

	//*****************************************************************

	//Iniciar el conteo del tiempo para las evaluaciones de rendimiento
	//*****************************************************************

	uswtime(&utime0, &stime0, &wtime0);
	//*****************************************************************
	//Reservar memoria del arreglo
	int *A = malloc(sizeof(int)*n);

	//leer números
	for(i=0;i<n;i++)
		scanf("%d",&A[i]);
	//*****************************************************************

	//Algoritmo Burbuja Simple

	//Realiza el ordenamiento a pares, es decir revisa cada elemento
	//del arreglo y lo compara con el siguiente, si es mayor el elemento
	//siguiente entonces los intercambia de posición
```

```
	//*************************************************************************

	int j, aux;


	for(i = 1; i < n; i++) {
		for(j = 0; j < n - 1; j++) {
			if(A[j] > A[j + 1]) {
				aux = A[j];
				A[j] = A[j + 1];
				A[j + 1] = aux;
			}
		}
	}
	//*************************************************************************
	//Imprimir números
	for(i=0;i<n;i++)
		printf("\n%d",A[i]);
	//*************************************************************************

	//Evaluar los tiempos de ejecución
	//*************************************************************************
	uswtime(&utime1, &stime1, &wtime1);

	//Cálculo del tiempo de ejecución del programa
	printf("\n");
	printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
	printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
	printf("sys (Tiempo en acciÃ³nes de E/S)  %.10f s\n",  stime1 -
stime0);
	printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
	printf("\n");

	//Mostrar los tiempos en formato exponencial
	printf("\n");
	printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
	printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 -
utime0);
	printf("sys (Tiempo en acciÃ³nes de E/S)  %.10e s\n",  stime1 -
stime0);
	printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
	printf("\n");
	//*************************************************************************
```

```
	//Terminar programa normalmente
	exit (0);
}
```

3.2. Ordenamiento Burbuja Optimizado

Instrucciones de compilación

1. Compilación

El código tiene comandos únicamente funcionales en Linux, por lo tanto, para poder compilar y ejecutar el código, es necesario estar en un entorno Linux. En una terminal, ejecutar el siguiente comando : `gcc main.c tiempo.c -o`

2. Ejecución

Para poder ejecutar el código en Linux, se utilizará el siguiente comando `./bin <n> < input.txt>`, donde <n> se reemplaza por el número de números a ordenar, asumiendo que el archivo .txt que contiene los 10 millones de números de prueba se llama `input.txt` y se encuentra en el mismo directorio de los códigos.

Código

```
//*****************************************************************
//Quintana Martinez Erick
//Curso: Análisis de algoritmos
//(C) Noviembre 2020
//ESCOM-IPN
//Medición de tiempo en C y recepción de parámetros en C bajo UNIX del
algoritmo de burbuja optimizada
//versión 1.3
//*****************************************************************

//*****************************************************************
//LIBRERÍAS INCLUIDAS
//*****************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"

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

	//Variables del main
```

```
	//*****************************************************************

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

	//*****************************************************************

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


	//Si no se introducen exactamente 2 argumentos (Cadena de ejecución
y cadena=n)
	if (argc!=2)
	{
		printf("\nIndique el tamaño 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]);
	}

	//*****************************************************************

	//Iniciar el conteo del tiempo para las evaluaciones de rendimiento
	//*****************************************************************

	uswtime(&utime0, &stime0, &wtime0);
	//*****************************************************************
	//Reservar memoria del arreglo
	int *A = malloc(sizeof(int)*n);

	//leer números
	for(i=0;i<n;i++)
		scanf("%d",&A[i]);

	//*****************************************************************

	//Algoritmo Burbuja Optimizada

	//Se comparan los elementos del arreglo entre pares pero
```

```
      //como al final de cada iteración el elemento mayor queda
      //situado en su posición ya no será necesario comparar con
      //otro número
      //*********************************************************************

      int j, aux;

      for(i = 0;i < n; i++){
            for(j = 0; j < i; j++){
                  if(A[i] < A[j]){
                        aux = A[j];
                        A[j] = A[i];
                        A[i] = aux;
                  }
            }
      }
      //*********************************************************************
      //Imprimir números
      for(i=0;i<n;i++)
            printf("\n%d",A[i]);
      //*********************************************************************

      //Evaluar los tiempos de ejecución
      //*********************************************************************
      uswtime(&utime1, &stime1, &wtime1);

      //Cálculo del tiempo de ejecución del programa
      printf("\n");
      printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
      printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
      printf("sys (Tiempo en acciÃ³nes de E/S)  %.10f s\n",  stime1 -
stime0);
      printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
      printf("\n");

      //Mostrar los tiempos en formato exponencial
      printf("\n");
      printf("real (Tiempo total)  %.10e s\n",  wtime1 - wtime0);
      printf("user (Tiempo de procesamiento en CPU) %.10e s\n",  utime1 -
utime0);
      printf("sys (Tiempo en acciÃ³nes de E/S)  %.10e s\n",  stime1 -
stime0);
      printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));
      printf("\n");
```

```
	//*************************************************************

	//Terminar programa normalmente
	exit (0);
}
```

3.3. Ordenamiento por Inserción

Instrucciones de compilación

1. Compilación

El código tiene comandos únicamente funcionales en Linux, por lo tanto, para poder compilar y ejecutar el código, es necesario estar en un entorno Linux.
En una terminal, ejecutar el siguiente comando, donde InsertionSort es el archivo que contiene la función [ordenamiento] Inserción () y el hilo principal:

```
gcc InsertionSort.c tiempo.c -o is
```

2. Ejecución

Para poder ejecutar el código en Linux, se utilizará el siguiente comando, donde <n> se reemplaza por el número de números a ordenar, asumiendo que el archivo .txt que contiene los 10 millones de números de prueba se llama `input.txt` y se encuentra en el mismo directorio de los códigos.

```
./is <n> < input.txt
```

Código

```
/*
    Titulo: Ordenamiento de inserción
    Descripción: Implementación del algoritmo de ordenamiento selección
para ordenar n números.
    Fecha: 27/10/2020
    Version: 1.2
    Autor: Luis Armando Ramírez Espinosa
*/

/*******************************************************/
/*        BIBLIOTECAS UTILIZADAS                       */
/*******************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"

/*******************************************************/
/*        PROTOTIPOS DE FUNCIONES                      */
/*******************************************************/

void Insercion (int *, int);
void printArr (int *, int);
```

```
/*********************************************************/
/*        VARIABLES GLOBALES                             */
/*********************************************************/

//Se optó por inicializar un arreglo del
//máximo tamaño del problema establecido
int A[10000007];

/*********************************************************/
/*    FUNCIÓN PRINCIPAL DEL PROGRAMA                     */
/*********************************************************/

int
main (int argc, char **argv)
{
  //Variables para la medición del tiempo
  double utime0, stime0, wtime0, utime1, stime1, wtime1;

  //Si no se proporcionan dos argumentos en la línea de comandos
  //(el nombre del ejecutable y el argumento n), el programa termina
  //la ejecución
  if (argc != 2)
    return 0;

  //Inicializamos n con el segundo elemento
  //del argumento argv de la función main
  int n = atoi (argv[1]);

  //Lectura de los números
  for (int i = 0; i < n; i++)
    scanf ("%d", &A[i]);

    //Comienza la medición de los tiempos haciendo uso
  //de la función proporcionada por "tiempo.h"
  uswtime (&utime0, &stime0, &wtime0);

  //Llamada a la función Insercion () para ordenar el arreglo
  Insercion (A, n);

  //En este punto se detiene la medición de tiempo del algoritmo
  uswtime (&utime1, &stime1, &wtime1);

  //Cálculo de los tiempos de ejecución del programa
  printf ("Tiempo real: %.10f s\n", wtime1 - wtime0);
  printf ("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
  printf ("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
  printf ("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));

  return 0;
}
```

```
/*********************************************************/
/*    IMPLEMENTACIÓN DE FUNCIONES                        */
/*********************************************************/

//Insercion (int *, int) -> void
//Recibe el arreglo de enteros por ordenar,
//y un entero que representa su tamaño.
//Ordena el algoritmo moviendo cada elemento en desorden hacia
//el inicio del arreglo hasta encontrar el lugar correcto.
void
Insercion (int *A, int n)
{
  int j = 0;
  int temp = 0;

  for (int i = 0; i < n; i++)
    {
      j = i;
      temp = A[i];

      while (j > 0 && temp < A[j - 1])
        {
       A[j] = A[j-1];
       j--;
      }

      A[j] = temp;
    }

  return;
}

//printArr (int *, int) -> void
//Recibe el arreglo de enteros por ordenar,
//y un entero que representa su tamaño.
//Ordena el algoritmo moviendo cada elemento en desorden hacia
//el inicio del arreglo hasta encontrar el lugar correcto.
void
printArr (int *A, int n)
{
  for (int i = 0; i < n; i++)
    printf ("%d\n", A[i]);

  return;
}
```

3.4. Ordenamiento por Selección

Instrucciones de compilación

1. Compilación

El código tiene comandos únicamente funcionales en Linux, por lo tanto, para poder compilar y ejecutar el código, es necesario estar en un entorno Linux.
En una terminal, ejecutar el siguiente comando, donde SelectionSort es el archivo que contiene la función [ordenamiento] Selección () y el hilo principal:

```
gcc SelectionSort.c tiempo.c -o ss
```

2. Ejecución

Para poder ejecutar el código en Linux, se utilizará el siguiente comando, donde <n> se reemplaza por el número de números a ordenar, asumiendo que el archivo .txt que contiene los 10 millones de números de prueba se llama `input.txt` y se encuentra en el mismo directorio de los códigos.

```
./ss <n> < input.txt
```

Código

```
/*
     Titulo: Ordenamiento de selección
     Descripción: Implementación del algoritmo de ordenamiento selección
para ordenar n números.
     Fecha: 27/10/2020
     Version: 1.2
     Autor: Luis Armando Ramírez Espinosa
*/

/*********************************************************/
/*    BIBLIOTECAS UTILIZADAS                             */
/*********************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "tiempo.h"

/*********************************************************/
/*    PROTOTIPOS DE FUNCIONES                            */
/*********************************************************/

void Seleccion (int *, int);
void printArr (int *, int);

/*********************************************************/
/*    VARIABLES GLOBALES                          */
/*********************************************************/

//Se optó por inicializar un arreglo del
//máximo tamaño del problema establecido
int A[10000007];

/*********************************************************/
/*    FUNCIÓN PRINCIPAL DEL PROGRAMA                     */
```

```
/*********************************************************/

int
main (int argc, char **argv)
{
  //Variables para la medición del tiempo
  double utime0, stime0, wtime0, utime1, stime1, wtime1;

  //Si no se proporcionan dos argumentos en la línea de comandos
  //(el nombre del ejecutable y el argumento n), el programa termina
  //la ejecución
  if (argc != 2)
    return 0;

  //Inicializamos n con el segundo elemento
  //del argumento argv de la función main
  int n = atoi (argv[1]);

  //Lectura de los números
  for (int i = 0; i < n; i++)
    scanf ("%i", &A[i]);

  //Comienza la medición de los tiempos haciendo uso
  //de la función proporcionada por "tiempo.h"
  uswtime (&utime0, &stime0, &wtime0);

  //Lamada a la función Seleccion () para ordenar el arreglo
  Seleccion (A, n);

  //En este punto se detiene la medición de tiempo del algoritmo
  uswtime (&utime1, &stime1, &wtime1);

  //Cálculo de los tiempos de ejecución del programa
  printf ("Tiempo real: %.10f s\n", wtime1 - wtime0);
  printf ("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
  printf ("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
  printf ("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));

  return 0;
}
/*********************************************************/
/*    IMPLEMENTACIÓN DE FUNCIONES                        */
/*********************************************************/
//Seleccion (int *, int) -> void
```

```c
//Recibe el arreglo de enteros por ordenar,
//y un entero que representa su tamaño.
//Ordena el algoritmo intercambiando las posiciones del i-ésimo
//número de la lista con el i-ésimo número más pequeño en el arreglo,
//i indexado en 1.
void
Seleccion (int *A, int n)
{
  int k = 0;
  int p = 0;
  int temp = 0;

  for (k = 0; k < n - 1; k++)
    {
      p = k;
      for (int i = k + 1; i < n; i++)
        {
       if (A[i] < A[p])
         p = i;
      }

      temp = A[p];
      A[p] = A[k];
      A[k] = temp;
    }

  return;
}

//printArr (int *, int) -> void
//Recibe un arreglo de enteros a imprimir,
//y un entero que representa su tamaño
//Itera por todo el contenido del arreglo e imprime cada elemento
void
printArr (int *A, int n)
{
  for (int i = 0; i < n; i++)
    printf ("%d\n", A[i]);

  return;
}
```

3.5. Ordenamiento por Árbol binario de búsqueda con rotaciones

Instrucciones de compilación

1. Compilación

El código tiene comandos únicamente funcionales en Linux, por lo tanto, para poder compilar y ejecutar el código, es necesario estar en un entorno Linux.
Abrir la terminal y ejecutar el siguiente comando (con nombre del archivo del código `abbSort.c`):
`gcc abbSort.c tiempo.c -o bin`

2. Ejecución

Para poder ejecutar el código en Linux, se utilizará el siguiente comando, donde <n> se reemplaza por el número de números a ordenar, asumiendo que el archivo .txt que contiene los 10 millones de números de prueba se llama `input.txt` y se encuentra en el mismo directorio de los códigos.
`./bin <n> < input.txt`

Código

```
/*
    Titulo: Ordenamiento con Árbol binario de búsqueda
    Descripción: Implemntación de un árbol binario de búsqueda con
rotaciones, su
    recorrido inOrder para devolver los elementos ordenados.
    Fecha: 26/10/2020
    Version: 1.6.1
    Autor: Luis Fernando Reséndiz Chávez
*/

/*********************************************************
        BIBLIOTECAS UTILIZADAS
*********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"

/*********************************************************
        ESTRUCTURA DEL ÁRBOL BINARIO DE BÚSQUEDA
*********************************************************/

typedef struct node {
    int value; // Valor del nodo.
    struct node* left; // Apuntador a un hijo izquierdo.
    struct node* right; // Apuntador a un hijo derecho.
```

```
    int height; // Altura del nodo respecto al árbol.
} node;

/*********************************************************
        PROTOTIPOS DE FUNCIONES
*********************************************************/

int max(int a, int b);
int height(node *root);
int getBalance(node* root);
node* createNode(int value);
node* rightRotation(node* root);
node* leftRotation(node* root);
node* insert(node* root, int value);
void inOrder(node* root);

/*********************************************************
        VARIABLES GLOBALES
*********************************************************/

int sorted = 0; // Posición desde donde empezar a guardar los números
ordenados.
int* a; // Apuntador de tipo entero para recibir los números.

/*********************************************************
        FUNCIÓN PRINCIPAL DEL PROGRAMA
*********************************************************/

int main(int argc, char *argv[]) {
    // Si no se da un tamaño del problema al ejecutar, finaliza.
    if(argc != 2)
        return 0;

    // Variables para medir el tiempo del algoritmo.
    double utime0, stime0, wtime0,utime1, stime1, wtime1;

    // Se recibe la n (tamaño del problema).
    int n = atoi(argv[1]);
    // Se le asigna memoria al apuntador "a", para n números.
    a = malloc(sizeof(int) * n);

    // Variable iterativa.
    int i;

    // Inicialización del árbol.
    node* ABBSort = NULL;
```

```
    for(i = 0; i < n; ++i) {
        scanf("%d", &a[i]);
    }

    // Incio del algoritmo, comenzamos a contar el tiempo.
    uswtime(&utime0, &stime0, &wtime0);


    for(i = 0; i < n; ++i) {
        ABBSort = insert(ABBSort, a[i]);
    }

    inOrder(ABBSort);

    // Fin del algoritmo, dejamos de medir el tiempo
    uswtime(&utime1, &stime1, &wtime1);

    //Cálculo del tiempo de ejecución del programa
    printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
     printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
     printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
     printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));


    return 0;
}

/*********************************************************
        IMPLEMENTACIÓN DE FUNCIONES
*********************************************************/

// max(int, int) -> int
// Recibe dos numeros enteros y devuelve el mayor de ellos.
int max(int a, int b) {
    return (a > b) ? a : b;
}

// height(node*) -> int
// Recibe una apuntador a una estructura nodo y devuelve su altura
respecto al árbol.
int height(node *root) {
    return (root == NULL) ? 0 : root -> height;
}
```

```
// createNode(int) -> node*
// Recive un número entero y devuleve un apuntador a una estrcutura nodo
con dicho valor.
node* createNode(int value) {
    node* root =(node*) malloc(sizeof(node));

    root -> value = value;
    root -> left = NULL;
    root -> right = NULL;
    root -> height = 1;

    return root;
}

// rightRotation(node*) -> node*
// Recibe un apuntador a una estructura nodo, rota su subarbol a la
derecha para balancearlo y retorna la nueva raiz
node* rightRotation(node* root) {
    node* next = root -> left;
    node* temp = next -> right;

    next -> right = root;
    root -> left = temp;

    root -> height = max(height(root -> left), height(root -> right)) +
1;
    next -> height = max(height(next -> left), height(next -> right)) +
1;

    return next;
}

// leftRotation(node*) -> node*
// Recibe un apuntador a una estructura nodo, rota su subarbol a la
izquierda para balancearlo y retorna la nueva raiz
node* leftRotation(node* root) {
    node* next = root -> right;
    node* temp = next -> left;

    next -> left = root;
    root -> right = temp;

    root -> height = max(height(root -> left), height(root -> right)) +
1;
    next -> height = max(height(next -> left), height(next -> right)) +
1;
```

```
    return next;
}

// getbalance(node*) -> int
// Recibe un apuntador a una estructura nodo y devuelve du factor de
balance, sirve para
// ver hacia que lado queda balanceado el árbol
int getBalance(node* root) {
    return (root == NULL) ? 0 : height(root -> left) - height(root ->
right);
}

// insert(node*, int) -> node*
// Recibe un apuntador a una estructura nodo que representa la raiz del
arbol, tambien
// recibe un número entero y lo inserta en el árbol creando un nuevo
nodo, devuelve el árbol
// con el nuevo numero insertado.
node* insert(node* root, int value) {
    if(root == NULL) {
        return createNode(value);
    }

    if(value < root -> value) {
        root -> left = insert(root -> left, value);
    } else if(value > root -> value) {
        root -> right = insert(root -> right, value);
    } else {
        return root;
    }

    root -> height = max(height(root -> left), height(root -> right)) +
1;

    int balance = getBalance(root);

    if(balance > 1 && value < root -> left -> value) {
        return rightRotation(root);
    }

    if(balance < -1 && value > root -> right -> value) {
        return leftRotation(root);
    }

    if(balance > 1 && value > root -> left -> value) {
        root -> left = leftRotation(root -> left);
        return rightRotation(root);
```

```
    }

    if(balance < -1 && value < root -> right -> value) {
        root -> right = rightRotation(root -> right);
        return leftRotation(root);
    }

    return root;
}

// inOrder(root*) -> void
// Realiza el recorrido inOrder (hijo izquierdo, raiz, hijo derecho) de
un árbol binario.
// Almacena los valores ya ordenados de cada nodo en el arreglo incial
(variable global).
void inOrder(node* root) {
    if(root != NULL) {
        inOrder(root -> left);
        a[sorted++] = root -> value;
        inOrder(root -> right);
    }
}
```

3.6. Ordenamiento Shell

Instrucciones de compilación

Compilación

El código tiene comandos únicamente funcionales en Linux, por lo tanto, para poder compilar y ejecutar el código, es necesario estar en un entorno Linux.

Abrir la terminal y ejecutar el siguiente comando (con nombre del archivo del código `shellSort.c`):

```
gcc shellSort.c tiempo.c -o bin
```

Ejecución

Para poder ejecutar el código en Linux, se utilizará el siguiente comando, donde <n> se reemplaza por el número de números a ordenar, asumiendo que el archivo .txt que contiene los 10 millones de números de prueba se llama `input.txt` y se encuentra en el mismo directorio de los códigos.

```
./bin <n> < input.txt
```

Código

```
/*
    Titulo: Ordenamiento Shell
    Descripción: Implemntación del algoritmo de ordenamiento Shell
optimizado
    Fecha: 26/10/2020
    Version: 1.3
```

```
    Autor: Luis Fernando Reséndiz Chávez
    Observaciones: La implementación la tomé de un libro, ya que la del
profesor no
    funcionaba.
*/

/**********************************************************
        BIBLIOTECAS UTILIZADAS
**********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "tiempo.h"

/**********************************************************
        PROTOTIPOS DE FUNCIONES
**********************************************************/

void ShellSort(int* a, int n);

/**********************************************************
        FUNCIÓN PRINCIPAL DEL PROGRAMA
**********************************************************/

int main(int argc, char *argv[]) {
    // Si no se da un número n iniciando el progra, este finaliza.
    if(argc != 2)
        return 0;

    // Variables para calcular el tiempo de ejecución del algoritmo.
    double utime0, stime0, wtime0,utime1, stime1, wtime1;

    // Obteniendo la n (tamaño del problema) y se crea una variable
iterativa.
    int n = atoi(argv[1]), i;

    // Se asigna memoria a un apuntador tipo entero para almacenar los
números.
    int* a = malloc(sizeof(int) * n);

    // Inicia el conteo de tiempo del algoritmo.
    uswtime(&utime0, &stime0, &wtime0);

    // Se reciben los número a ordenar.
    for(i = 0; i < n; ++i) {
        scanf("%d", &a[i]);
```

```
    }

    // Se ejecuta el algoritmo.
    ShellSort(a, n);

    // Finaliza el conteo de tiempo para el algoritmo.
    uswtime(&utime1, &stime1, &wtime1);

    //Cálculo del tiempo de ejecución del programa.
     printf("real (Tiempo total)  %.10f s\n",  wtime1 - wtime0);
     printf("user (Tiempo de procesamiento en CPU) %.10f s\n",  utime1 -
utime0);
     printf("sys (Tiempo en acciónes de E/S)  %.10f s\n",  stime1 -
stime0);
     printf("CPU/Wall   %.10f %% \n",100.0 * (utime1 - utime0 + stime1 -
stime0) / (wtime1 - wtime0));


    return 0;
}

/*********************************************************
        IMPLEMENTACIÓN DE FUNCIONES
*********************************************************/

// ShellSort(int*, int) -> void
// Recibe un apuntador al arreglo original a ordenar, y un entero que
representa su tamaño.
// Ordena los numeros del arreglo mediante el algoritmo de ShellSort.
void ShellSort(int* a, int n) {
    int i, j, k, tmp;

    for (i = n / 2; i > 0; i = i / 2){
        for (j = i; j < n; j++){
            for(k = j - i; k >= 0; k = k - i){
                if (a[k+i] >= a[k])
                    break;
                else {
                    tmp = a[k];
                    a[k] = a[k+i];
                    a[k+i] = tmp;
                }
            }
        }
    }
}
```

3.7. Repositorio de Github

Para permitir una forma de trabajo asíncrona y evitar sus problemas relacionados, el equipo creó un repositorio en GitHub, que además de contener los archivos fuentes utilizados para la compilación y ejecución del programa, también almacena los Shell scripts auxiliares, los archivos de Matlab utilizados e imágenes de las gráficas, y los archivos que guardan la información de las varias pruebas realizadas durante la práctica.

El repositorio se puede acceder mediante el siguiente enlace:

https://github.com/LuisJackRamirez/AnalisisAlgoritmosOrdenamiento

4. Bibliografía

T. H. Cormen, Introduction to Algorithms, Massachusetts: MIT Press, 2009.

R. Sedgewick, An Introduction to the Analysis of Algorithms, San Francisco: Addison-Wesley, 2013.