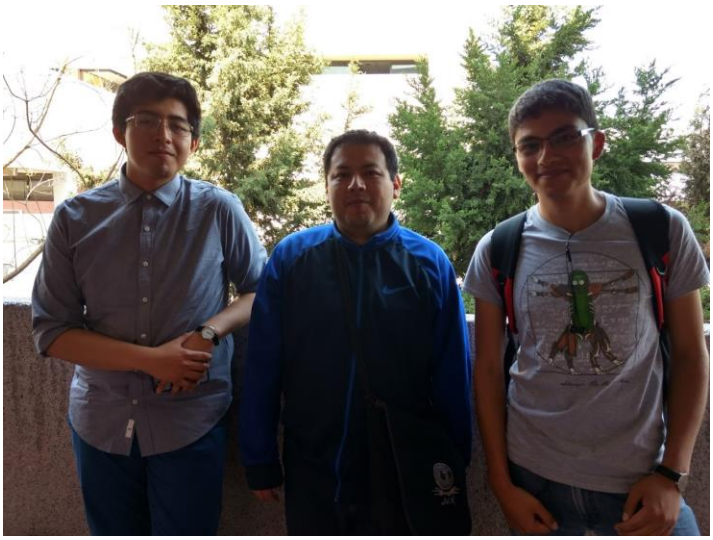
Instituto Politécnico Nacional

Escuela Superior de Cómputo

PRUEBAS A POSTERIORI (ALGORITMOS DE ORDENAMIENTO)

Práctica 01



Análisis de Algoritmos

M. en C. Edgardo Adrián Franco Martínez
Grupo: 3CM3
Fecha: 14 /Marzo/2018

Equipo: Git Gud (Equipo Arbol)

- Calva Hernández José Manuel 2017630201
- Meza Madrid Raúl Damián 2017631051
- Montaño Ayala Alan Israel 2016630260

Índice

Introducción

Analizar un algoritmo significa predecir los recursos que el algoritmo va a requerir. Ocasionalmente, recursos como memoria, ancho de banda de comunicación, o hardware de computadora se consideran primarios, pero el que más nos interesa medir es el tiempo de computación. Generalmente, al analizar varios algoritmos candidatos para un problema, podemos identificar el más eficiente. [1]

En las ciencias computacionales, el término referente a medir la economía se conoce como medidas de complejidad, de espacio de memoria (o simplemente espacial) y tiempo. La primera de ellas es medida por muchas cosas, incluyendo el número de variables, y el número y tamaño de las estructuras de datos usadas en la ejecución del algoritmo. La otra es medida por el número de acciones elementales llevadas a cabo por el procesador durante la ejecución. [2]

Ambas, tanto espacio como tiempo, requeridas por un algoritmo usualmente difieren de entrada a entrada, y, de acuerdo con el desempeño del algoritmo refleja cómo varían los recursos consumidos de acuerdo a la entrada. [2]

Analizamos los algoritmos con la intención de mejorarlos, si es posible, y de escoger uno de entre varios con los que se podría resolver un problema. Usaremos los criterios siguientes:

1. Corrección.- Implica tres pasos principales, sin embargo, antes de ello debemos de aclarar que de forma general, que sea correcto implica que si se satisfacen las condiciones previas, las condiciones posteriores se cumplirán cuando el algoritmo termine.
   Un algoritmo tiene dos aspectos: el método de solución y la sucesión de instrucciones para ponerlo en práctica, es decir, su implementación. Establecer la corrección del método y/o de las fórmulas empleadas podría ser fácil o podría requerir una larga serie de lemas y teoremas acerca de los objetos con los que el algoritmo trabaja. Una vez establecido el método, lo implementamos en un programa.
2. Cantidad de trabajo realizado.- Queremos una medida del trabajo que nos diga algo acerca de la eficiencia del método empleado por un algoritmo, con independencia no sólo de la computadora, el lenguaje de programación y el programador, sino también de los múltiples detalles de implementación, procesamiento fijo (u operaciones de “contabilidad”), como la incrementación de índices de ciclos, el cálculo de índices de arreglos y el establecimiento de apuntadores en estructuras de datos. Para analizar un algoritmo podemos aislar una operación específica que sea fundamental para el problema que se estudia (o para los tipos de algoritmos en consideración), hacer caso omiso de la inicialización, el control de ciclos y demás tareas de contabilidad, y simplemente contar las operaciones básicas escogidas que el algoritmo efectúa.
3. Cantidad de espacio usado.- El número de celdas de memoria utilizadas por un programa, al igual que el número de segundos necesarios para ejecutar un programa, depende de la implementación específica. No obstante, es posible sacar algunas conclusiones acerca del consumo de espacio con sólo examinar un algoritmo. Un programa requiere espacio de almacenamiento para sus instrucciones, las constantes y variables que usa, y los datos de entrada. También podría ocupar cierto espacio de trabajo para manipular los datos y guardar información que necesita para efectuar sus cálculos. Los datos de entrada en sí podrían representarse de varias maneras, algunas de las cuales requieren más espacio que otras. Si los datos de entrada tienen una forma natural (digamos, un arreglo de números o una matriz), entonces analizamos la cantidad de espacio extra utilizado, aparte del programa y las entradas. Si la cantidad de espacio extra es constante, sin importar el tamaño de las entradas, decimos que el algoritmo trabaja en el lugar.

4. Sencillez, claridad.- La forma más sencilla y directa de resolver un problema no es la más eficiente. No obstante, la sencillez es una característica deseable de un algoritmo, pues podría facilitar la verificación de que el algoritmo es correcto, así como la escritura, depuración y modificación de los programas.
5. Optimidad.- Por más ingeniosos que seamos, no podremos mejorar un algoritmo para un problema más allá de cierto punto. Todo problema tiene una complejidad inherente, es decir, existe una cantidad mínima de trabajo que debe efectuarse para resolverlo. Para analizar la complejidad de un problema, no la de un algoritmo específico, escogemos una clase de algoritmos (a menudo especificando los tipos de operaciones que los algoritmos podrán realizar) y una medida de la complejidad, por ejemplo, la o las operaciones básicas que se contarán. Luego, podremos preguntar cuántas operaciones se necesitan realmente para resolver el problema. Decimos que un algoritmo es óptimo (en el peor caso) si ningún otro algoritmo de la clase estudiada efectúa menos operaciones básicas (en el peor caso). [3]

Operación elemental

Una *operación elemental* es aquella cuyo tiempo de ejecución se puede acotar superiormente por una constante que solamente dependerá de la implementación particular usada: de la máquina, del lenguaje de programación, etc. De esta manera la constante *no* depende ni del tamaño ni de los parámetros del ejemplar que se esté considerando. Dado que nos preocupan los tiempos de ejecución de algoritmos definidos salvo alguna constante multiplicativa, sólo el número de operaciones elementales ejecutadas importará en el análisis, y no el tiempo exacto requerido por cada una de ellas. [4]

Enfoques de análisis

Para medir la eficiencia de los algoritmos y poder decidir cual es más conveniente según que caso, tomando en cuenta los puntos anteriores, encontramos diferentes técnicas para hacer esta revisión, las cuáles serán descritas a continuación.

El enfoque *empírico* (o a *posteriori*) para seleccionar un algoritmo consiste en programar técnicas competidoras e ir probándolas en distintos casos con ayuda de una computadora. El enfoque *teórico* (o a *priori*) que es el más usado, sobre todo en la parte teórica de la materia, consiste en determinar matemáticamente la cantidad de recursos necesarios para cada uno de los algoritmos *como función del tamaño de los casos considerados*. Este análisis se hace usualmente para calcular el *tiempo de ejecución*, sólo ocasionalmente se hablará de la eficiencia en almacenamiento, o bien, su necesidad de otros recursos. [4]

Función complejidad

Este análisis nos dará como resultado una función que será llamada *función de complejidad*, la cual nos permitirá calcular el costo de ejecutar el algoritmo según la entrada que le proporcionamos nosotros. Sin embargo, cabe recalcar que esto es simplemente una aproximación teórica que busca simular el comportamiento del algoritmo, mas no es una verdad absoluta ni mucho menos exacta, ya que ésta se ve sujeta por muchas otras variantes que quizá no se tomaron en cuenta, o bien, son a nivel de hardware y ello reduce el desempeño.

Casos de análisis

El tiempo que requiere un algoritmo, o el espacio de almacenamiento que consume, pueden variar considerablemente entre dos ejemplares distintos del mismo tamaño. Esto nos da lugar a distintos tipos de análisis.

El análisis en <<el caso peor>> es adecuado para algoritmos cuyos tiempos de respuesta sean críticos. Si, por otra parte, es preciso utilizar muchas veces un algoritmo en muchos casos distintos, quizá sea más importante conocer el tiempo medio de ejecución para ejemplares de tamaño n.

Suele ser más difícil analizar el comportamiento medio de un algoritmo que analizar su comportamiento en el caso peor. Además, semejante análisis de comportamiento medio podría ser confuso si de hecho los casos que hubiera que resolver no se seleccionan aleatoriamente cuando el algoritmo se utiliza en la práctica.

Un análisis útil del comportamiento medio del algoritmo requiere por tanto un conocimiento *a priori* acerca de la distribución de los casos que hay que resolver. Este suele ser un requisito poco realista.

Algoritmos de Ordenamiento

Una de los procedimientos más usados en la computación es el ordenamiento de arreglos. Desde los inicios de la computación, las computadoras no eran capaces de ordenar arreglos grandes debido a su complejidad y poco poder de procesamiento, es por eso que fue necesaria la invención de métodos de ordenamiento más eficaces. Hoy en día, incluso con los grandes avances en tecnología de hardware, resulta necesario mantener esa eficacia, pues que ahora las computadoras deben de realizar dicho proceso repetidas veces con cantidades aún más grandes de información.

El problema de ordenar un conjunto de objetos fue uno de los primeros problemas que se estudiaron intensamente en las ciencias de la computación. Muchas de las aplicaciones más conocidas del paradigma de diseño de algoritmos Divide y Vencerás son algoritmos de ordenamiento. Durante los años sesenta, cuando el procesamiento comercial de datos se automatizó en gran escala, el programa de ordenamiento era el que se ejecutaba con mayor frecuencia en muchas instalaciones de cómputo. Una compañía de software se mantuvo operando durante años gracias a que tenía un programa de ordenamiento mejor. Con el hardware actual, los aspectos de desempeño del ordenamiento han cambiado un poco. En los años sesenta, la transferencia de datos entre almacenamiento lento (cinta o disco) y la memoria principal era un importante cuello de botella del desempeño. La memoria principal era del orden de 100,000 bytes y los archivos a procesar eran varios órdenes de magnitud mayores. La atención se concentraba en los algoritmos para efectuar este tipo de ordenamiento. Hoy, las memorias principales 1,000 veces mayores (o sea, de 100 megabytes) son cosa común, y las hay 10,000 veces mayores (de unos cuantos gigabytes), de modo que la mayor parte de los archivos cabe en la memoria principal.

Hay varias razones de peso para estudiar los algoritmos de ordenamiento. La primera es que tienen utilidad práctica porque el ordenamiento es una actividad frecuente. Así como tener las entradas de un directorio telefónico o de un diccionario en orden alfabético facilita su uso, el trabajo con conjuntos grandes de datos en las computadoras se facilita si los datos están ordenados. La segunda es que se ha ideado una buena cantidad de algoritmos para ordenar (más de los que cubriremos aquí), y si el lector estudia varios de ellos se convencerá del hecho de que es posible enfocar un problema dado desde muchos puntos de vista distintos. El tratamiento de los algoritmos en este capítulo deberá dar al lector algunas ideas acerca de cómo se puede mejorar un algoritmo dado y cómo escoger entre varios algoritmos. La tercera es que el ordenamiento es uno de los pocos problemas para los que es fácil deducir cotas inferiores firmes del comportamiento en el peor caso y en el caso promedio. Las cotas son firmes en el sentido de que existen algoritmos que efectúan aproximadamente la cantidad mínima de trabajo especificada. Por ello, tenemos algoritmos de ordenamiento prácticamente óptimos. [3]

Planteamiento del problema

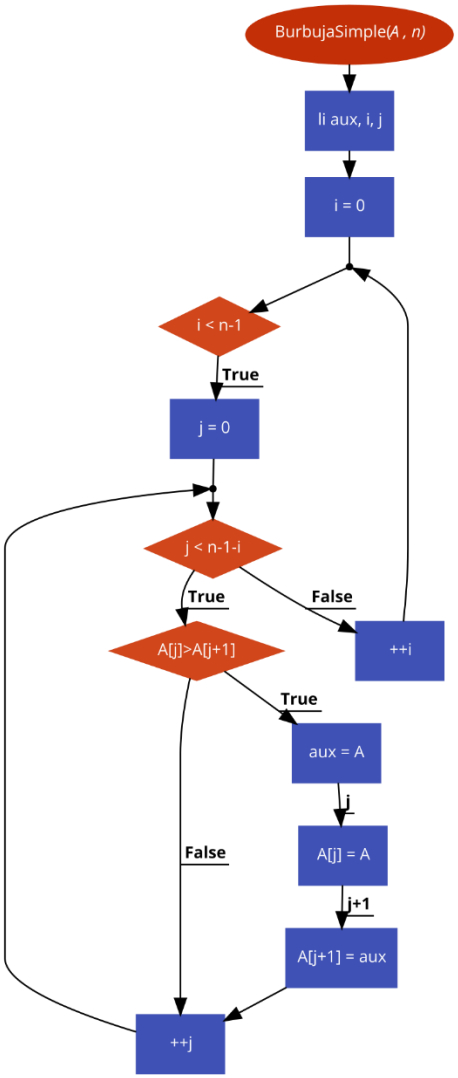
Con base en el archivo de entrada proporcionado que tiene 10,000,000 números diferentes; ordenarlo bajo los siguientes métodos de ordenamiento y comparar experimentalmente las complejidades de estos.

- Burbuja (Bubble Sort)
    - Burbuja Simple
    - Burbuja Optimizada
- Inserción (Insertion Sort)
- Selección (Selection Sort)
- Shell (Shell Sort)
- Ordenamiento con árbol binario de búsqueda (Tree Sort)

Algoritmos

Burbuja Simple

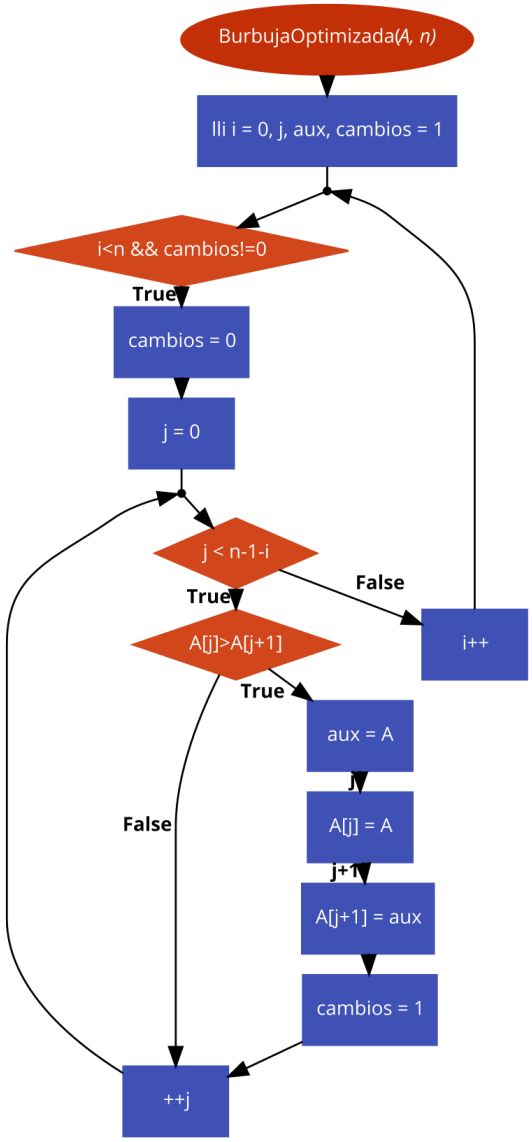
- Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiandolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.
    - El método de la burbuja es uno de los mas 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 los intercambia de posición.
    - Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas" . También es conocido como el método del intercambio directo.




```
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
```

Burbuja Optimizada

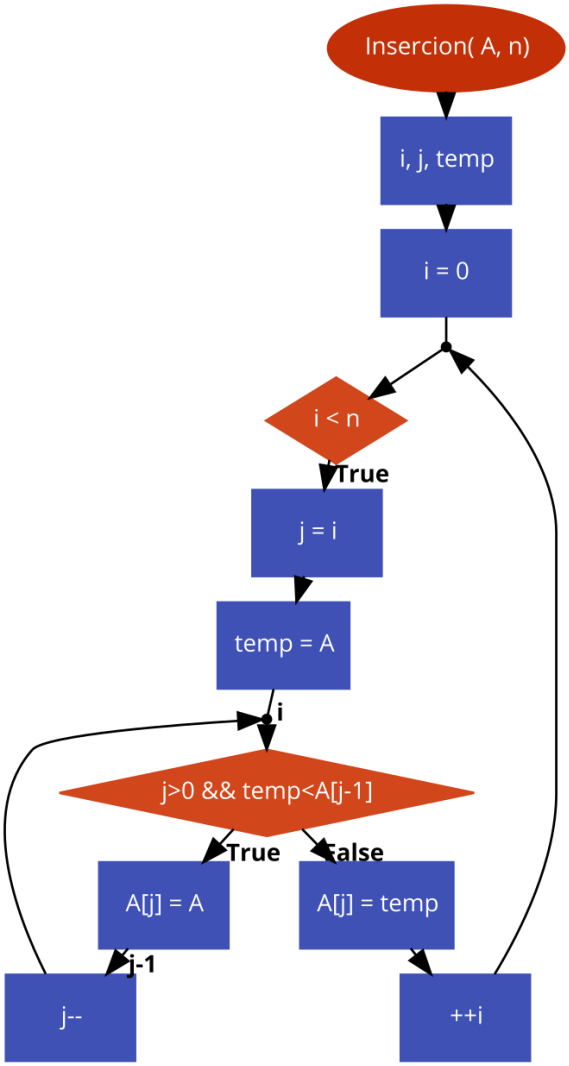
- Como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo así el número de comparaciones por iteración, además pude existir la posibilidad que realizar iteraciones de más si el arreglo ya fue ordenado totalmente.




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

Ordenamiento por Inserción

- Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.
- Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y éste es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.



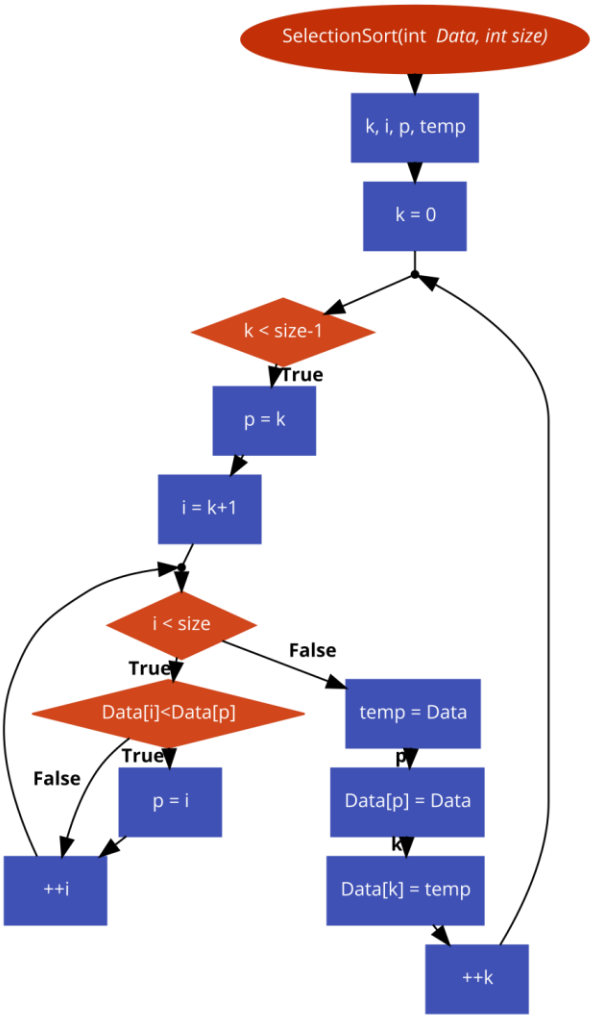

```
Procedimiento Insercion(A,n)
{
       para i=0 hasta n-1 hacer
              j=i
              temp=A[i]
              mientras(j>0) && (temp<A[j-1]) hacer
                     A[j]=A[j-1]
                     j--
              fin mientras
              A[j]=temp
       fin para
fin Procedimiento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

Ordenamiento por Selección

El algoritmo de ordenamiento por selección ordena un arreglo buscando repetidamente el elemento mínimo (considerando orden ascendente) de una parte no ordenada y poniéndolo en el inicio. El algoritmo mantiene 2 sub arreglos del arreglo original

1. El sub arreglo que ya está ordenado
2. El sub arreglo que resta por ordenar

En cada iteración del ordenamiento de selección, el elemento mínimo(considerando un orden ascendente)del subarreglo no ordenado es tomado y llevado al subarreglo ordenado




```
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
                        fin si
                fin para
                temp = A[p]
                A[p] = A[k]
                A[k] = temp
        fin para
fin Procedimiento
```

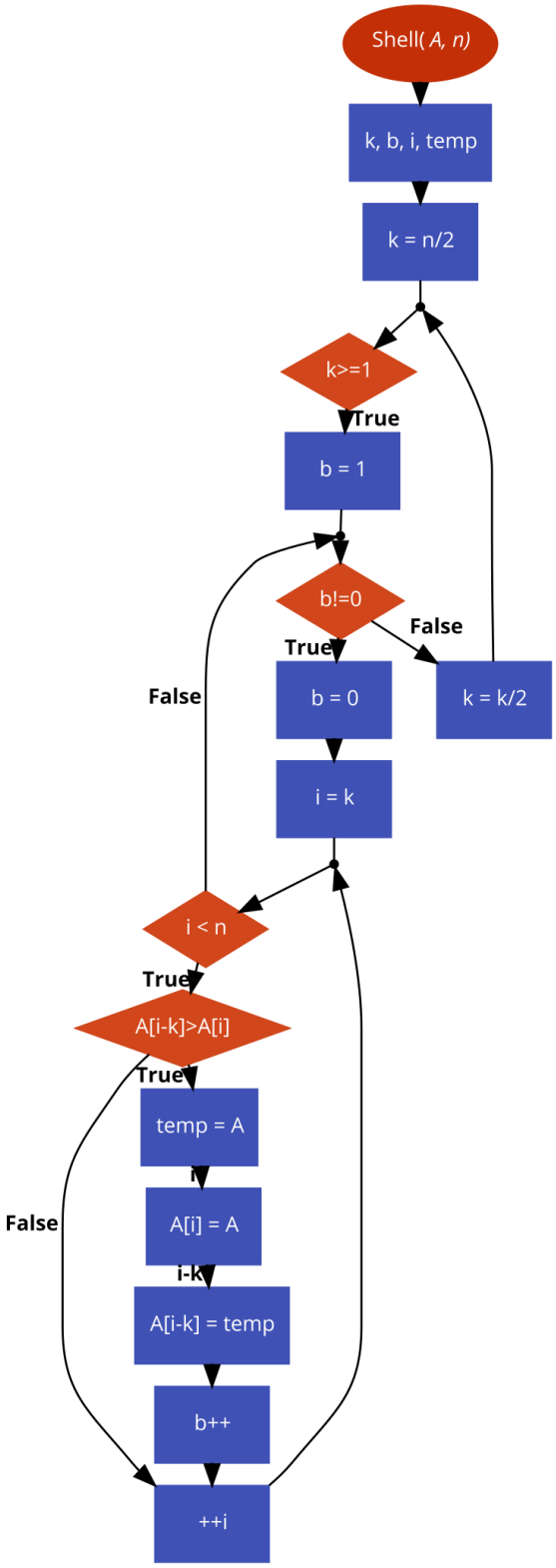
Ordenamiento Shell

- El Shell es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

- El algoritmo Shell mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del ordenamiento Shell es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.
- Shell 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 esté separado k elementos del anterior (a esta separación a menudo se le llama salto o gap)
- Se debe empezar con k=n/2, siendo n el número de elementos del arreglo, y utilizando siempre la división entera (TRUNC)
- Después iremos variando k haciéndolo más pequeño mediante sucesivas divisiones por 2, hasta llegar a k=1.

```
Procedimiento Shell(A,n)
        k = TRUNC(n/2)
        mientras k >= 1 hacer
                b= 1
                mientras b!=0 hacer
                        b=0
                        para i=k hasta i>=n-1 hacer
                                si A[i-k]>A[i]
                                        temp=A[i]
                                        A[i]=A[i-k]
                                        A[i-k]=temp
                                        b=b+1
                                fin si
                        fin para
                fin mientras
                k=TRUNC(k/2)
        fin mientras
fin Procedimiento
```


Shell(A, n)
k, b, i, temp
k = n/2
k>=1
True
b = 1
b!=0
False
True
False
b = 0
k = k/2
i = k
i < n
True
A[i-k]>A[i]
True
temp = A
False
A[i] = A
i-k
A[i-k] = temp
b++
++i

Ordenamiento con un Árbol Binario de Búsqueda (ABB)

- El ordenamiento con la ayuda de un árbol binario de búsqueda es muy simple debido a que solo requiere de dos pasos simples.

1. Insertar cada uno de los números del vector a ordenar en el árbol binario de búsqueda.
2. Reemplazar el vector en desorden por el vector resultante de un recorrido InOrden del Árbol Binario, el cual entregará los números ordenados.

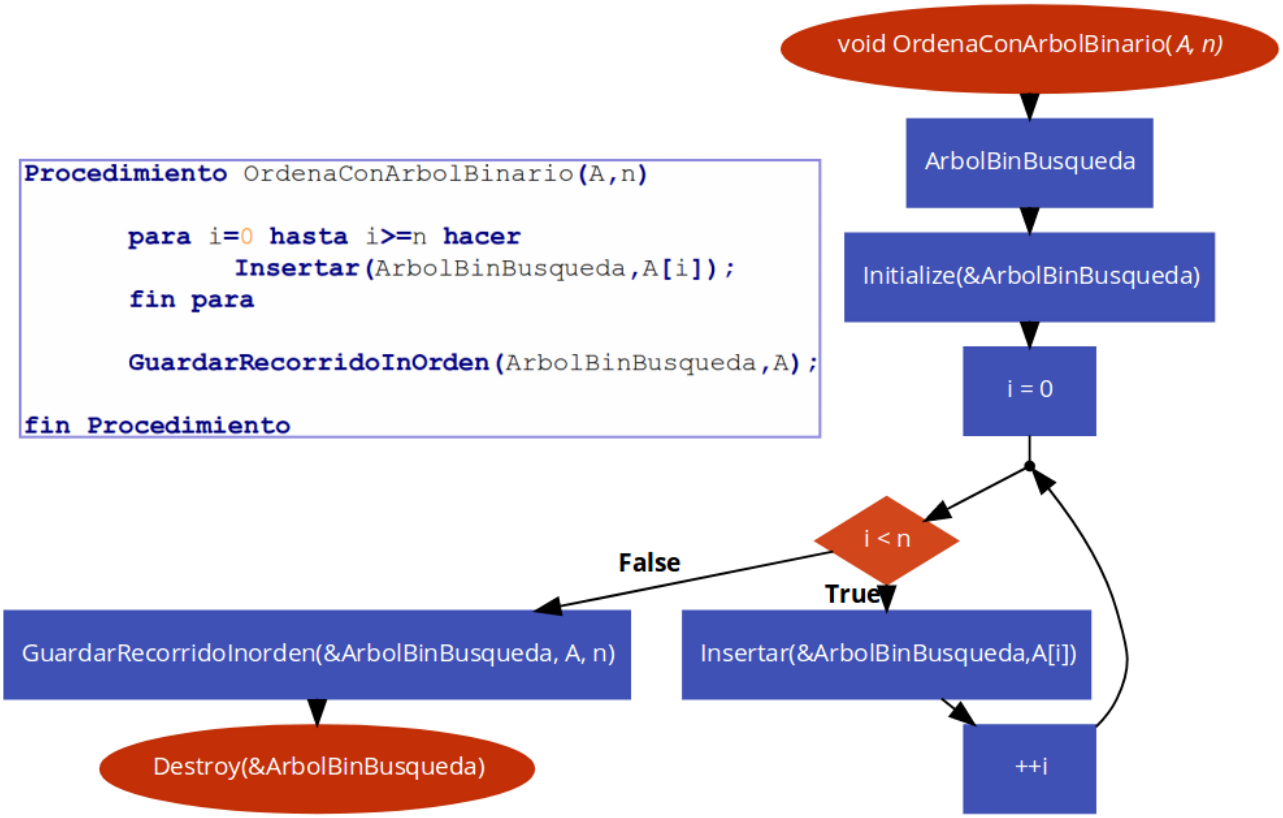
- La eficiencia de este algoritmo está dada según la eficiencia en la implementación del árbol binario de búsqueda, lo que puede resultar mejor que otros algoritmos de ordenamiento.

```
Procedimiento OrdenaConArbolBinario(A,n)

        para i=0 hasta i>=n hacer
                Insertar(ArbolBinBusqueda,A[i]);
        fin para

        GuardarRecorridoInOrden(ArbolBinBusqueda,A);

fin Procedimiento
```



Implementación de los algoritmos

Ordenamiento Burbuja Simple

```
1.  /*Descripción: Función encargada de ordenar los números por medio del algoritmo de Burbuja Simple
2.  Recibe: int * Data (arreglo de números a ordenar), int size (cantidad de números)
3.  Devuelve:
4.  Observaciones: */
5.  void SimpleBubbleSort(int * Data, int size) {
6.      for (int i = 0; i < size - 1; i++) { // Se hará una iteración por todo el arreglo de números
7.          for (int j = 0; j < size - 1 - i; j++) {
8.  // A partir del inicio, iremos realizando una segunda iteración que recorrerá los elementos -i
9.              if (Data[j] > Data[j + 1]) { // Si existe un número que sea mayor que su sudecesor, se in
    tercambiarán
10.                 int temp = Data[j + 1];
11.                 Data[j + 1] = Data[j];
12.                 Data[j] = temp;
13.             }
14.         }
15.     }
16. }
```

Ordenamiento Burbuja Optimizada

```
1.  /*Descripción: Función encargada de ordenar los números por medio del algoritmo de Burbuja Optimizada
2.  Recibe: int * Data (arreglo de números a ordenar), int size (cantidad de números)
3.  Devuelve:
4.  Observaciones: */
5.  void OptimizedBubbleSort(int * Data, int size) {
6.      int changes = 1; // Declaramos una bandera para los cambios realizados
7.      int i = 0;
8.      while ((i < size - 1) && (changes != 0)) { // Realizaremos una iteración hasta llegar al final de
    l arreglo de números, o bien, hasta que ya no hayamos realizado cambios, lo que indicará que ya trerm
    inamos de ordenar
9.          changes = 0; // Iniciamos el ciclo indicando que no hemos realizado cambios
10.         for (int j = 0; j < size - 1 - i; j++) {
11. // Nuevamente llevaremos la iteración desde el primer elemento hasta los n elementos - i
12.             if (Data[j] > Data[j + 1]) {
13. // Si existe un elemento mayor que su sucesor, se intercambian
14.                 changes = 1; // Indicamos que realizamos un cambio en la iteración de i
15.                 int temp = Data[j + 1];
16.                 Data[j + 1] = Data[j];
17.                 Data[j] = temp;
18.             }
19.         }
20.         i++;
21.     }
22. }
```

Ordenamiento Inserción

```
1.  /*Descripción: Función encargada de ordenar los números por medio del algoritmo de Inserción
2.  Recibe: int * Data (arreglo de números a ordenar), int size (cantidad de números)
3.  Devuelve:
4.  Observaciones: */
5.  void insertion(int * Data, int size) {
6.      int i, j; // Variables usadas para las iteraciones
7.      int aux; // Variable usada como auxiliar para llevar control del número a insetar en el arreglo
8.      for (i = 0; i < size; i++) { // Iteración a través de todo el arreglo
9.          aux = Data[i]; // Posicionamos el auxiliar en el primer dato a revisar
10.         for (j = i; j > 0 && aux < Data[j - 1]; j--) {
11. // Iteración que buscará la posición donde debemos colocar el número entre sus predecesores
12.             Data[j] = Data[j - 1];
13. // Hasta encontrar su lugar, iremos recorriendo a sus predecesores
14.         }
15.         Data[j] = aux; // Al llegar a su posición, colocaremos el número
16.     }
17. }
```

Ordenamiento Selección

```
1.  /*Descripción: Función encargada de ordenar los números por medio del algoritmo de Selección
2.  Recibe: int * Data (arreglo de números a ordenar), int size (cantidad de números)
3.  Devuelve:
4.  Observaciones: */
5.  void SelectionSort(int * Data, int size) {
6.      for (int k = 0; k < size - 1; k++) {
7.  // Iteración que recorrerá todo el arreglo de números, hasta la penúltima posición
8.          int p = k; // Colocaremos un índice auxiliar en la posición k
9.          for (int i = k + 1; i < size; i++) {
10. // Iteración que recorrerá el arreglo desde una posición delante de su predecesora hasta el final
11.             if (Data[i] < Data[p]) {
12. // Si el número es menor que el que se encuentra en nuestro índice auxiliar,
13.                 p = i;
14. // Cambiaremos el índice auxiliar para llevar control del menor número disponible para esa posición
15.             }
16.         }
17.         int aux = Data[p]; // Una vez recorrido el resto de números, intercambiaremos los números par
    a colocar el menor disponible en la posición k
18.         Data[p] = Data[k];
19.         Data[k] = aux;
20.     }
21. }
```

Ordenamiento Shell

```
1.  /*Descripción: Función encargada de ordenar los números por medio del algoritmo de Shell
2.  Recibe: int * Data (arreglo de números a ordenar), int size (cantidad de números)
3.  Devuelve:
4.  Observaciones: */
5.  void ShellSort(int * Data, int size) {
6.      int k = trunc(size / 2); // Declararemos k, colocándola en la mitad del arreglo
7.      while (k >= 1) { // Hasta llegar al inicio del arreglo, realizaremos una iteración
8.          int b = 1;
9.  // Declararemos b a modo de bandera para saber si podemos seguir realizando saltos
10.         while (b != 0) { // Mnatendremos b hasta llegar al primer elemento del arreglo
11.             b = 0; // Iniciaremos suponiendo que el salto no es posible
12.             for (int i = k; i <= size - 1; i++) {
13. // Itereación que irá dando saltos cada vez más chicos a partir de k
14.                 if (Data[i - k] > Data[i]) { // Si el número en la posición del salto es mayor que el
     indicado, los intercambiaremos
15.                     int temp = Data[i];
16.                     Data[i] = Data[i - k];
17.                     Data[i - k] = temp;
18.                     b++; // En caso de haber cambios, la bandera se colocará en 1 y seguiremos con lo
    s salto de este tamaño
19.                 }
20.             }
21.         }
22.         k = trunc(k / 2); // Reduciremos el tamaño de los saltos a la mitad
23.     }
24. }
```

Ordenamiento con ABB

```
1.  /*Descripción: Función encargada de ordenar los números por medio de un ABB
2.  Recibe: int * Data (arreglo de números a ordenar), int size (cantidad de números)
3.  Devuelve:
4.  Observaciones: */
5.  void BinarySearchTreeSort(int * Data, int size) {
6.      arbol_bin BinaryTree; // Declaramos un ABB
7.      Initialize( & BinaryTree); // Inicializamos el ABB declarado anteriormente
8.      int i; // Variable de la iteración
9.      for (i = 0; i < size; ++i) // Recorreremos el arreglo de números iterativamente
10.     {
11.         Insert( & BinaryTree, Data[i]); // Cada número del arreglo, será añadido al ABB
12.     }
13.     InorderTraversal( & BinaryTree, Data, size);
14. // Realizaremos el recorrido inorden por el ABB para conseguir el orden de los números
15.     Destroy( & BinaryTree); // Destruimos el árbol una vez utilizado
16.     return;
17. }
```

Actividades y Pruebas

Comparativa de tiempo

N = 600,000

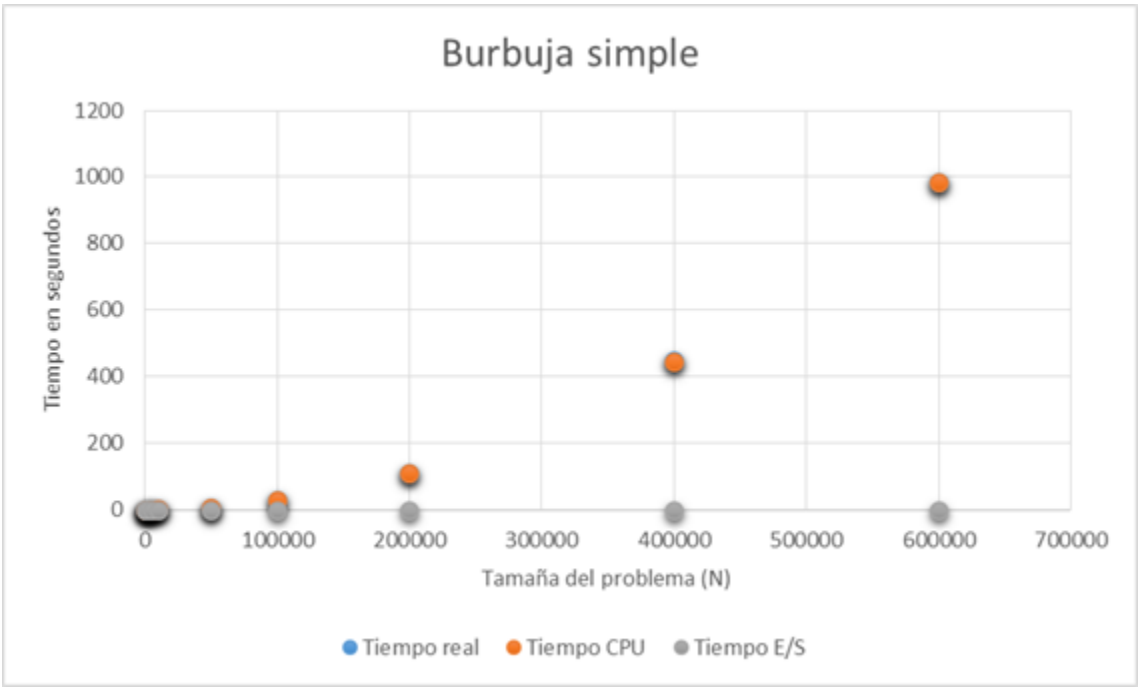
Algoritmo	Tiempo Real	Tiempo CPU	Tiempo E/S	% CPU/Wall
Burbuja Simple	983.5239081383	983.3510270000	0.1039900000	99.9929954790
Burbuja Optimizada	946.0755770206	946.0204970000	0.0121350000	99.9954607199
Inserción	225.2116699219	225.1990020000	0.0000000000	99.9943751042
Selección	394.3842461109	394.3951350000	0.0000000000	100.0027609848
Shell	13.3105909824	13.3095520000	0.0000000000	99.9921943178
Con ABB	7.4325480461	7.3594330000	0.0720520000	99.9856974202

N = 10,000,000

Algoritmo	Tiempo Real	Tiempo CPU	Tiempo E/S	% CPU/Wall
Burbuja Simple	-	-	-	-
Burbuja Optimizada	-	-	-	-
Inserción	-	-	-	-
Selección	-	-	-	-
Shell	24.8975100517	24.8981950000	0.0000000000	100.0027510714
Con ABB	11.5536689758	11.5055490000	0.0480340000	99.9992558569

Burbuja Simple

Gráfica de tiempo



Polinomios

N=1

$$1.519527452015423 \times 10^{-3}x - 55.5794$$

N=2

$$2.674812512303883 \times 10^{-9}x^2 + 3.878102482045647 \times 10^{-5}\,x - 1.0038$$

N=3

$$-4.556540957464123 \times 10^{-16}\,x^3 + 3.068260930536768 \times 10^{-9}\,x^2 - 3.804021234261248 \times 10^{-5}\,x + 0.2616$$

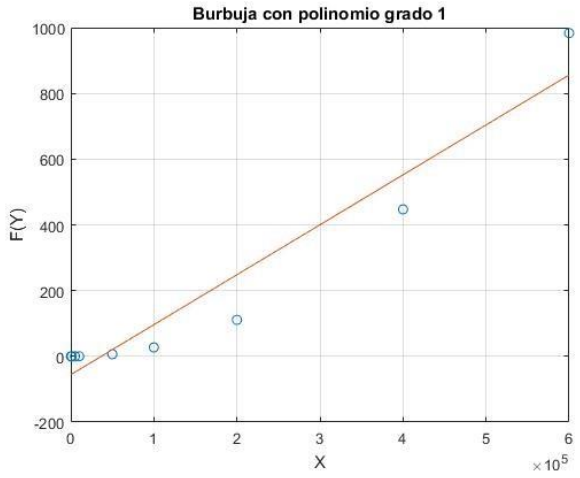
N=4

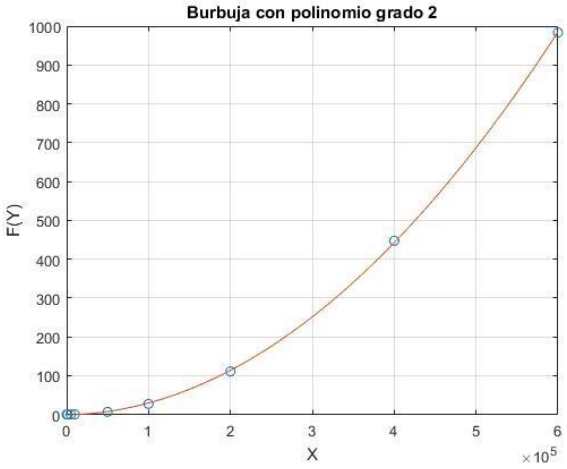
$$-9.789651168578290 \times 10^{-22}\,x^4 + 6.502980944869945 \times 10^{-16}\,x^3 + 2.700680047239788 \times 10^{-9}\,x^2 - 3.847341670980356 \times 10^{-6}\,x - 0.0027$$

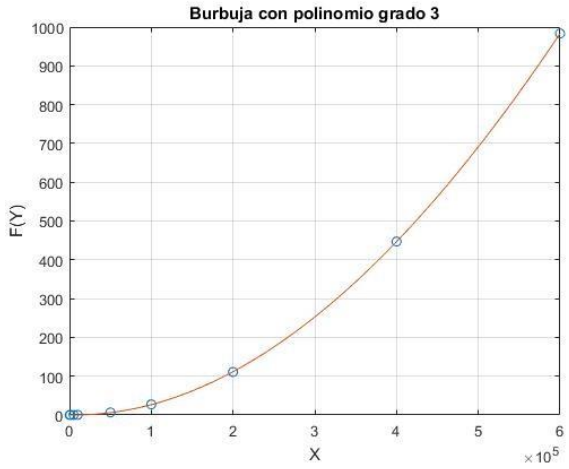
N=8

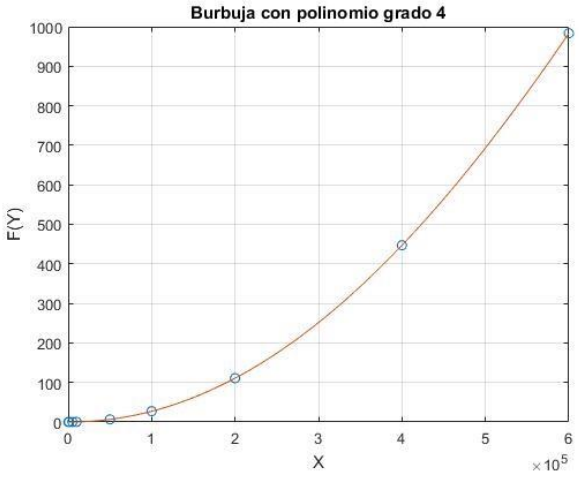
$$-2.12766813740185 \times 10^{-40}\,x^8 + 2.90802435291546 \times 10^{-34}\,x^7 - 1.37842222072597 \times 10^{-28}\,x^6 + 2.78234703523473 \times 10^{-23}\,x^5 - 2.45628634646540 \times 10^{-18}\,x^4 + 8.99719529233187 \times 10^{-14}\,x^3 + 1.54361647048513 \times 10^{-9}\,x^3 + 1.07143550787699 \times 10^{-6}\,x - 9.85876467786142 \times 10^{-5}$$

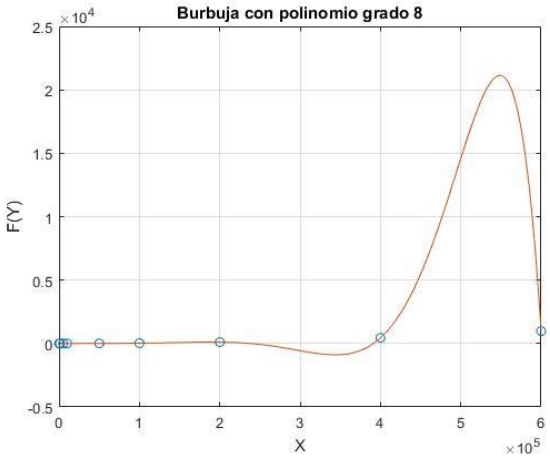
Gráficas polinomios











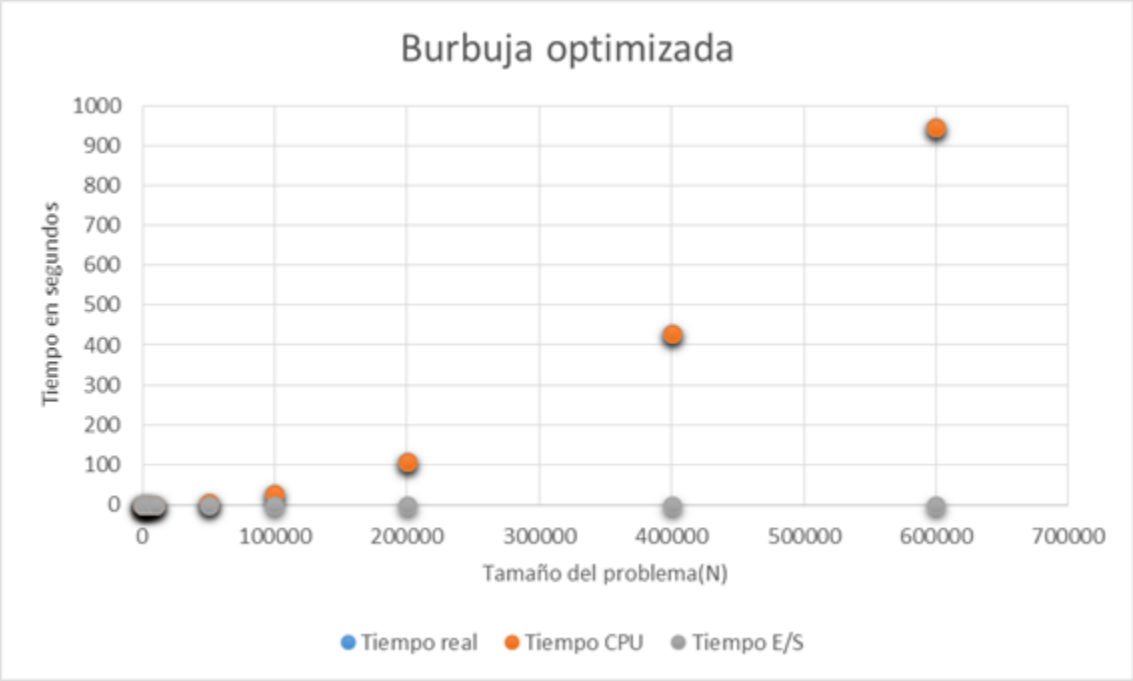
El polinomio escogido como mejor aproximación es:

$$2.674812512303883 \times 10^{-9}x^2 + 3.878102482045647 \times 10^{-5}\, x - 1.0038$$

n	Tiempo con base al polinomio
50000000	6,688,969.328 s
100000000	26,752,002.22 s
500000000	668,722,517.6 s
1000000000	2,674,851,292 s
5000000000	66,870,506,710 s

Burbuja Optimizada

Gráfica de tiempo



Polinomios

N=1

$$0.00146107454749725\,x\ -\ 53.3890$$

N=2

$$2.57238614542577 \times 10^{-9}x^2 + 3.70302166576954 \times 10^{-5}x - 0.9033$$

N=3

$$-3.65584879519962 \times 10^{-16}x^3 + 2.88806155341560 \times 10^{-9}x^2 - 2.46057545647931 \times 10^{-5}x + 0.1119$$

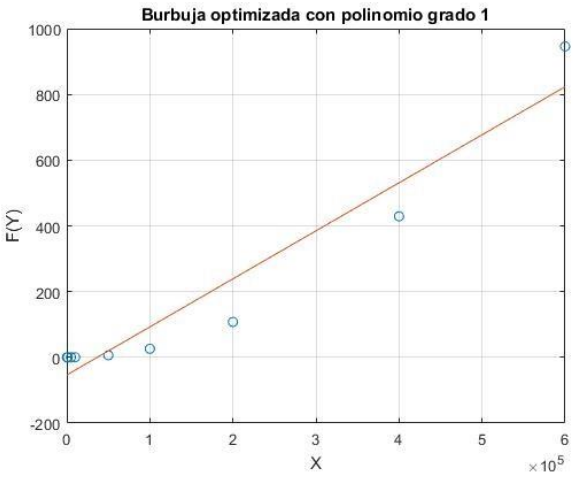
N=4

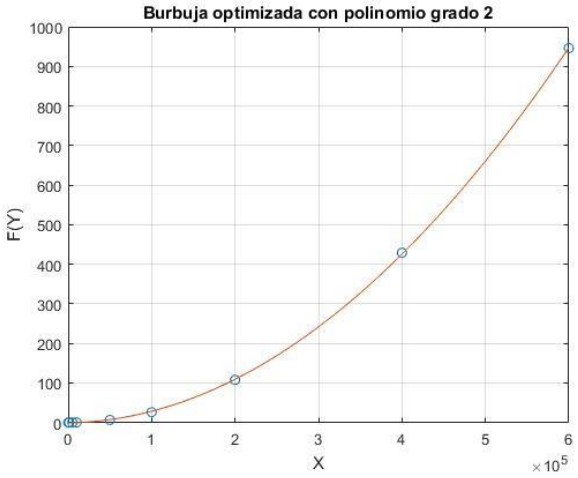
$$-1.46590636433398 \times 10^{-22}x^4 - 1.99979146854441 \times 10^{-16}x^3 + 2.83301984182101 \times 10^{-9}x^2 - 1.94857001451546 \times 10^{-5}x + 0.0723656904176590$$

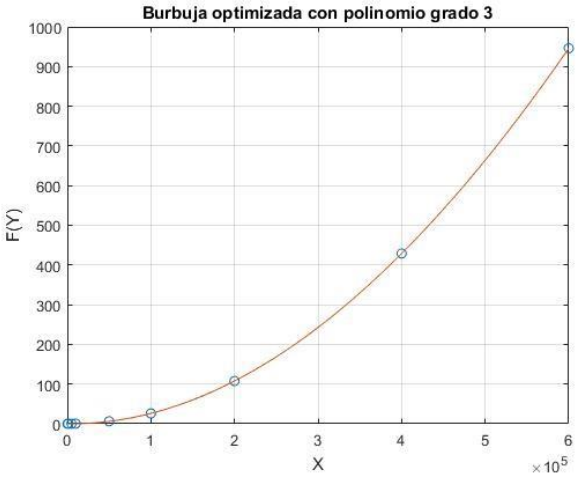
N=8

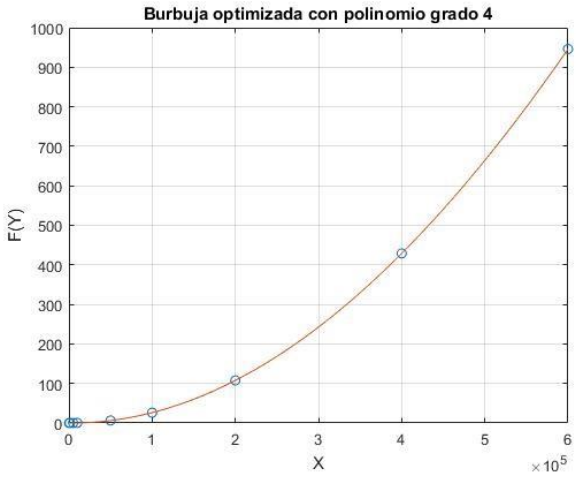
$$-1.42039654437986 \times 10^{-40}x^8 + 1.93394267315673 \times 10^{-34}x^7 - 9.09852541878559 \times 10^{-29}x^6 + 1.80694523858545 \times 10^{-23}x^5 - 1.53617410922537 \times 10^{-18}x^4 + 5.15883706335045 \times 10^{-14}x^3 + 2.04939414024803 \times 10^{-9}x^2 - 3.78720484839896 \times 10^{-7}x + 3.92621592414839 \times 10^{-5}$$

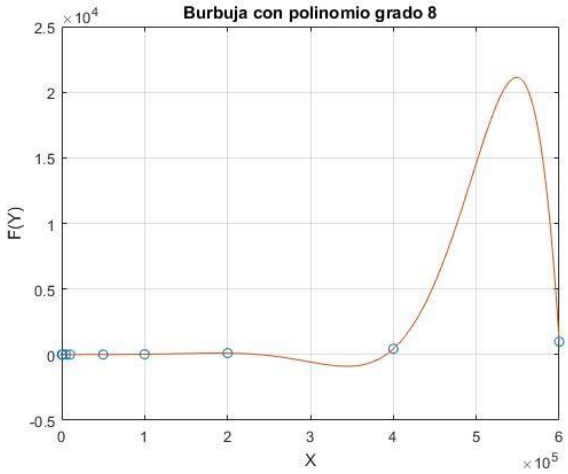
Gráficas polinomios











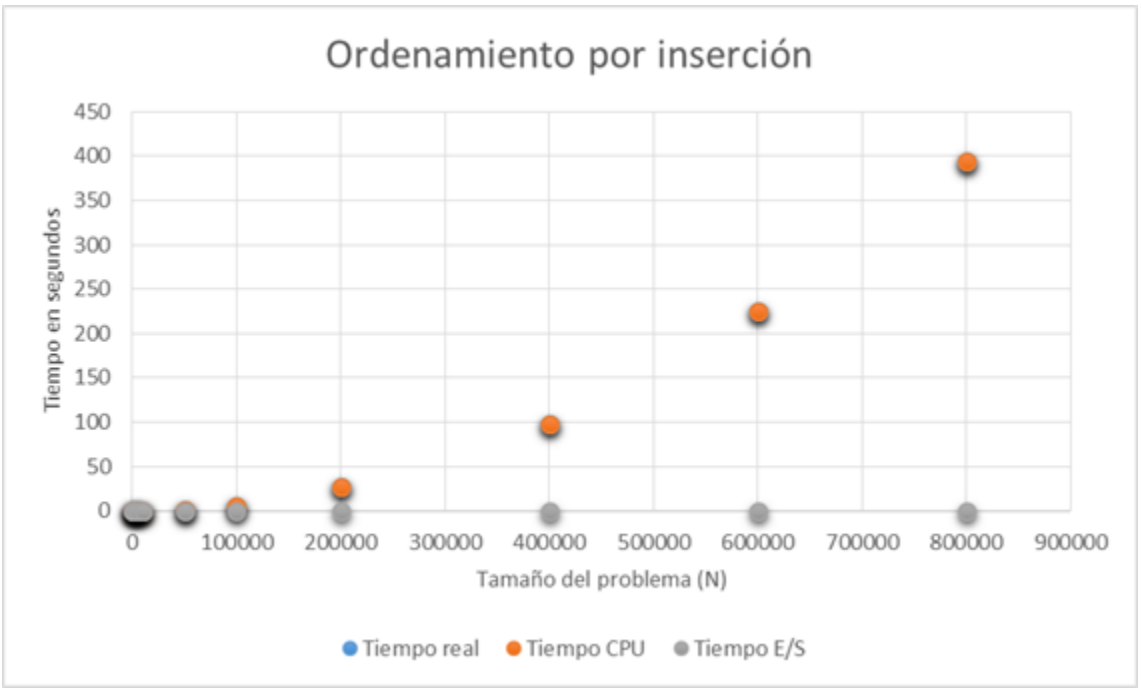
El polinomio escogido como mejor aproximación es:

$$2.57238614542577 \times 10^{-9}x^2 + 3.70302166576954 \times 10^{-5}x - 0.9033$$

n	Tiempo con base al polinomio
50,000,000	6,432,815.971 s
100,000,000	25,727,563.57 s
500,000,000	643,115,050.6 s
1,000,000,000	2,572,423,175 s
5,000,000,000	64,309,838,790 s

Inserción

Gráfica de tiempo



Polinomios

N=1

$$0.000447131277926480\ x\ -\ 21.6850$$

N=2

$$6.06112107755850 \times 10^{-9}x^2\ +\ 8.12838295127691 \times 10^{-6}x\ -\ 0.0913$$

N=3

$$-2.17174305938586 \times 10^{-17}x^3\ +\ 6.31351564517502 \times 10^{-10}x^2\ +\ 1.32741326008042 \times 10^{-6}x\ +\ 0.0611$$

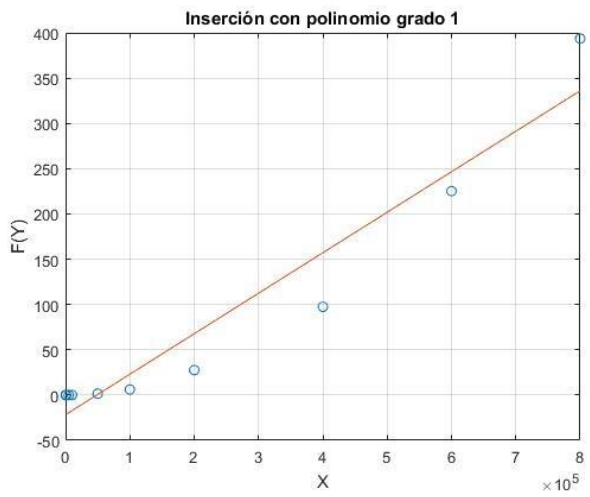
N=4

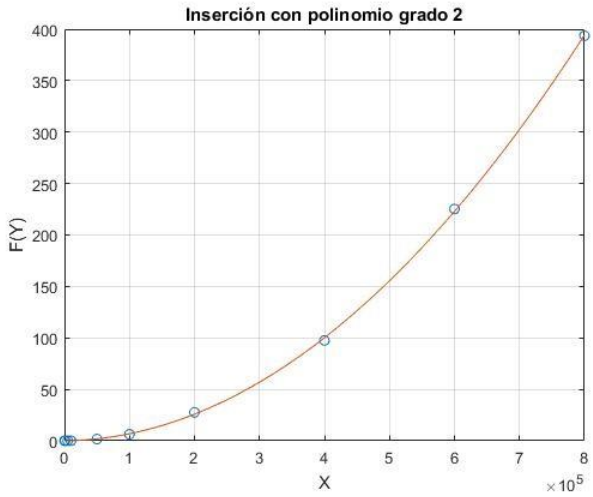
$$-3.99436916387043 \times 10^{-22}x^4\ +\ 5.91262163603944 \times 10^{-16}x^3\ +\ 3.50569313728177 \times 10^{-10}x^2\ +\ 3.82633106859991 \times 10^{-5}x\ -\ 0.3702$$

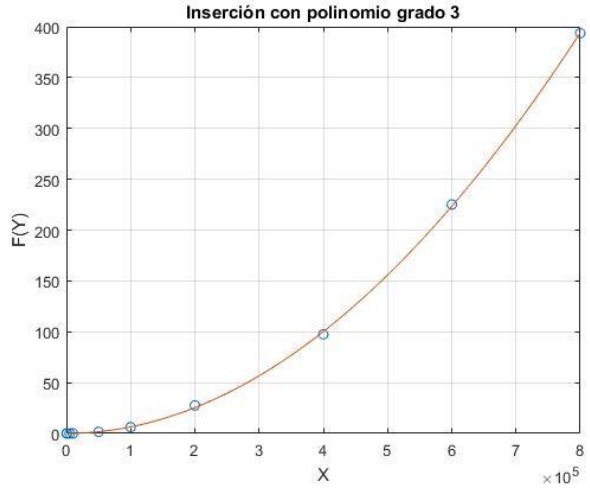
N=8

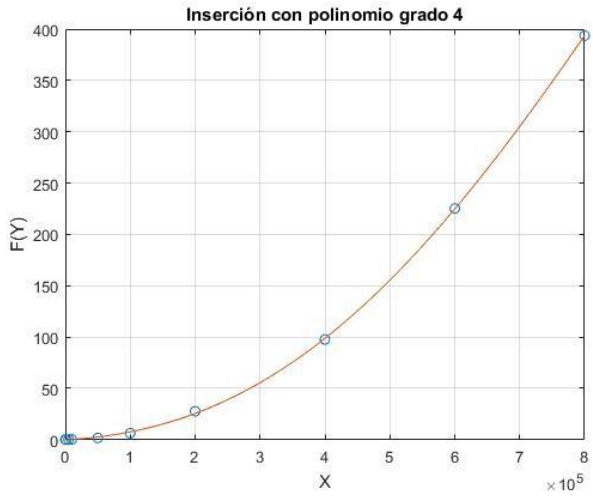
$$-4.05272935997384 \times 10^{-43}x^8\ +\ 7.61591715336753 \times 10^{-37}x^7\ -\ 4.78097812168175 \times 10^{-31}x^6\ +\ 1.08284590585810 \times 10^{-25}x^5\ -\ 3.82518584728370 \times 10^{-21}x^4\ -\ 6.48850712561805 \times 10^{-16}x^3\ +\ 6.50582134728221 \times 10^{-10}x^2\ +\ 2.58421024246413 \times 10^{-7}x\ -\ 0.1167$$

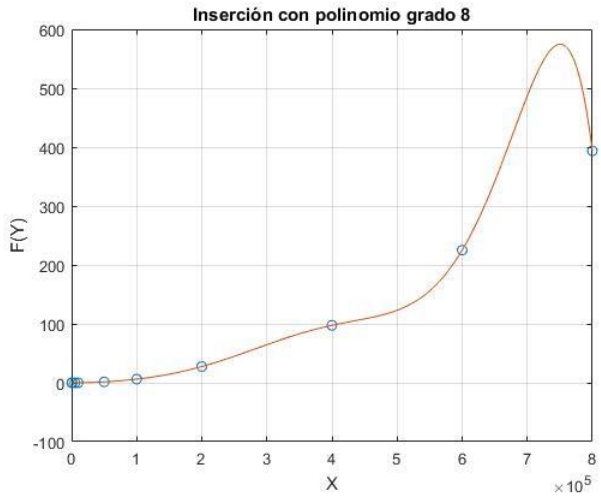
Gráficas polinomios











El polinomio escogido como mejor aproximación es:

$$6.06112107755850 \times 10^{-9}x^2 + 8.12838295127691 \times 10^{-6}x - 0.0913$$

n	Tiempo con base al polinomio
50,000,000	15,153,209.02
100,000,000	60,612,023.52
500,000,000	1,515,284,333
1,000,000,000	6,061,129,206
5,000,000,000	151,528,067,600

Selección

Gráfica de tiempo

Polinomios

N=1

$$0.000603609685093194\ x\ -\ 22.3614042325015$$

N=2

$$1.11382753018410 \times 10^{-9}x^2 -\ 1.29928349880812 \times 10^{-5}x +\ 0.364605572560146$$

N=3

$$2.15548807887764 \times 10^{-16}x^3 +\ 9.27705341648736 \times 10^{-10}x^2\ +\ 2.33477273936483 \times 10^{-5}x \\ -\ 0.233975301320910$$

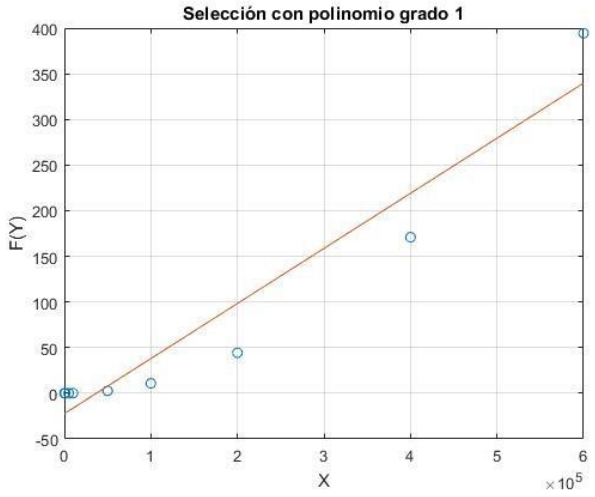
N=4

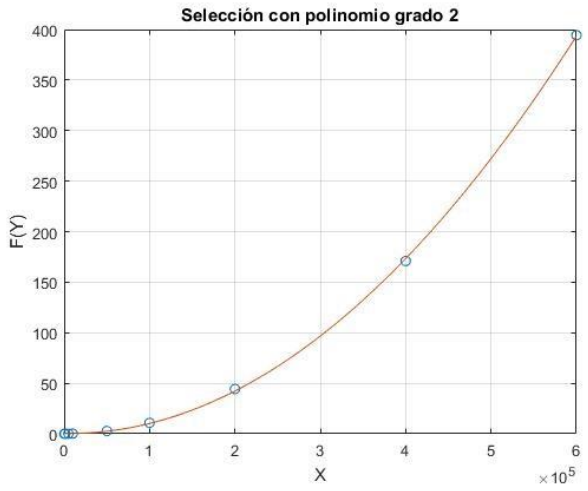
$$1.12767578716850 \times 10^{-21}x^4\ -\ 1.05840415041085 \times 10^{-15}x^3\ +\ 1.35112396482753 \times 10^{-9}x^2 \\ -\ 1.60392453249942 \times 10^{-5}x\ +\ 0.0704379495316086$$

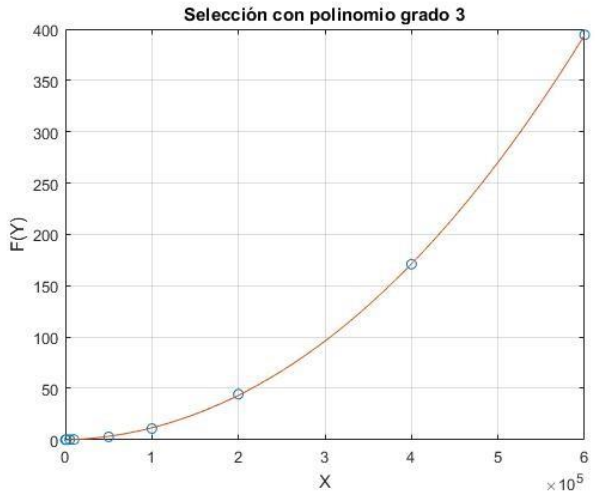
N=8

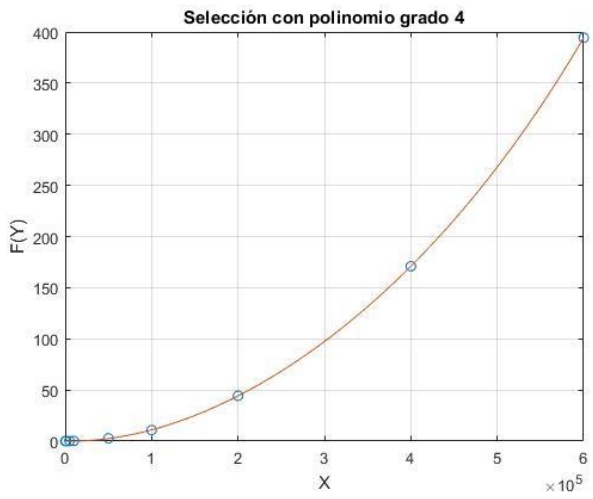
$$9.19694168356404 \times 10^{-41}x^8\ -\ 1.25313300358198 \times 10^{-34}x^7\ +\ 5.90668670478977 \times 10^{-29}x^6 \\ -\ 1.17887906574298 \times 10^{-23}x^5\ +\ 1.01361759945001 \times 10^{-18}x^4 \\ -\ 3.40943347691645 \times 10^{-14}x^3\ +\ 1.39982064244229 \times 10^{-9}x^2 \\ -\ 6.43467051199520 \times 10^{-8}x\ +\ 8.44381638972885 \times 10^{-6}$$

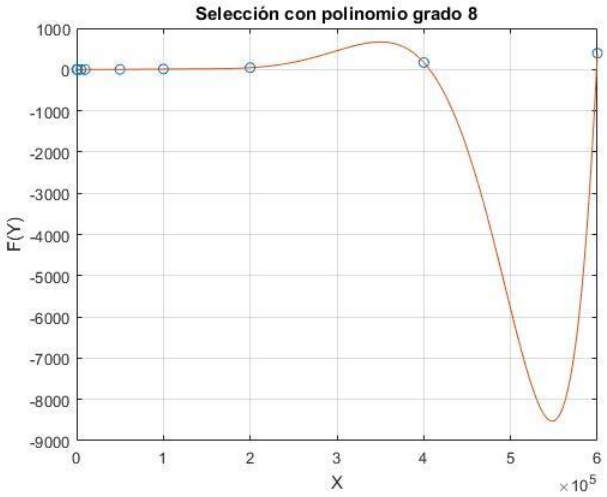
Gráficas polinomios











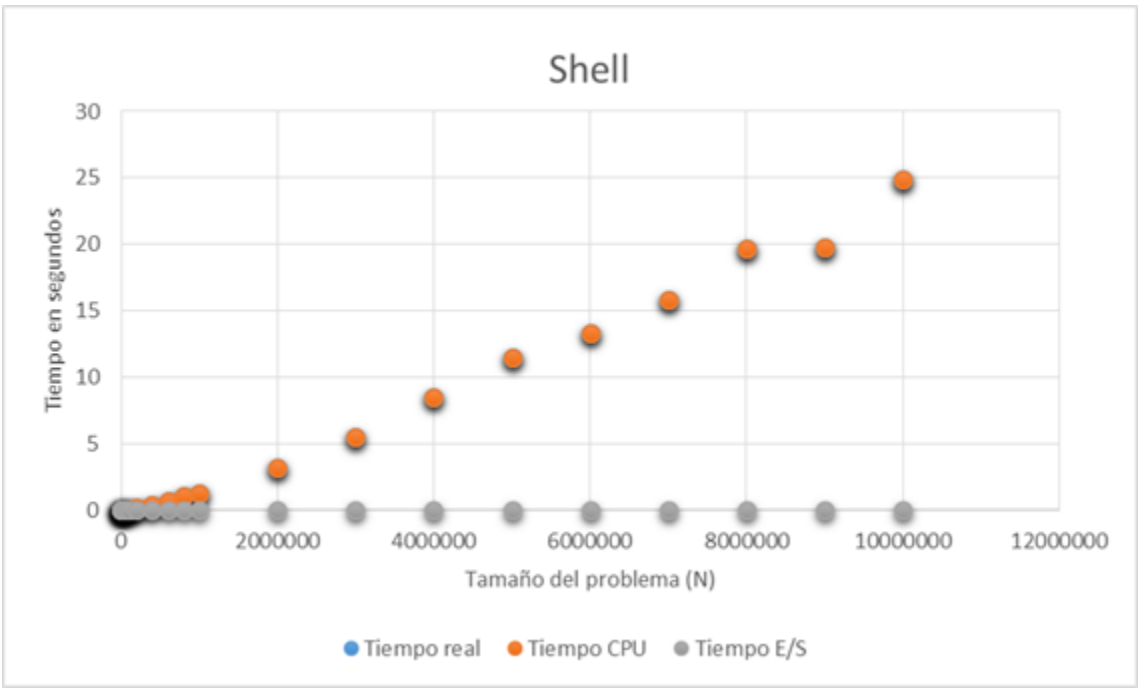
El polinomio escogido como mejor aproximación es:

$$1.11382753018410 \times 10^{-9}x^2 - 1.29928349880812 \times 10^{-5}x + 0.364605572560146$$

n	Tiempo con base al polinomio
50,000,000	2,783,919.548
100,000,000	11,136,976.38
500,000,000	278,450,386.4
1,000,000,000	1,113,814,538
5,000,000,000	27,845,623,290

Shell

Gráfica de tiempo



Polinomios

N=1

$$2.38905155086007 \times 10^{-6}x - 0.5161$$

N=2

$$5.42484102211795 \times 10^{-14}x^2 + 1.92113155283617 \times 10^{-6}x - 0.2263$$

N=3

$$-8.03893837339493 \times 10^{-21}x^3 + 1.68671544042433 \times 10^{-13}x^2 + 1.53614986995103 \times 10^{-6}x - 0.1192$$

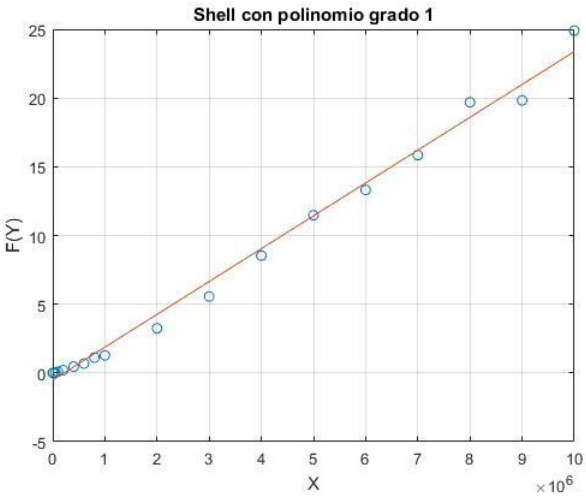
N=4

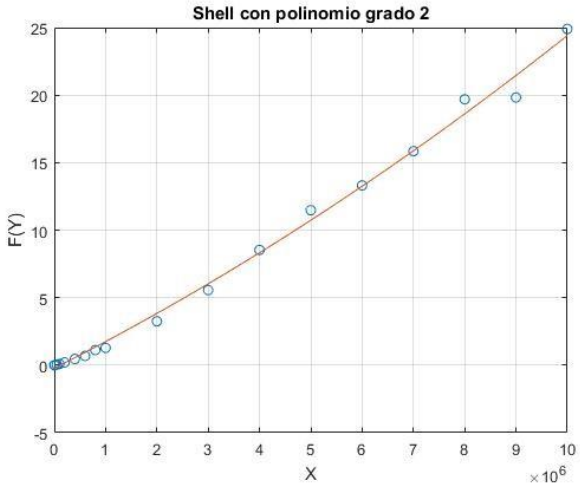
$$5.04740333657904 \times 10^{-27}x^4 - 1.05703802947671 \times 10^{-19}x^3 + 7.46076345063883 \times 10^{-13}x^2 + 5.07108208356875 \times 10^{-7}x + 0.0543$$

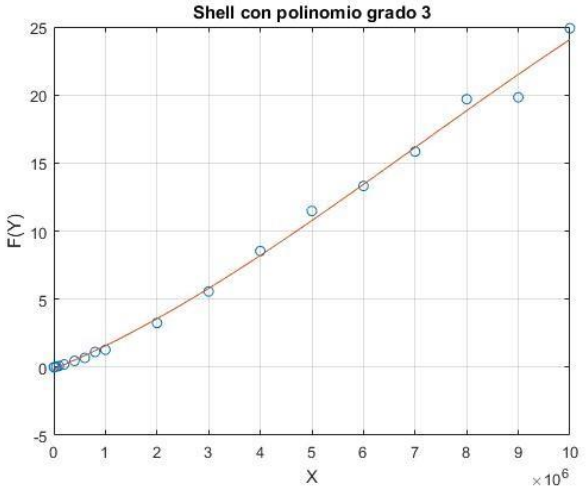
N=8

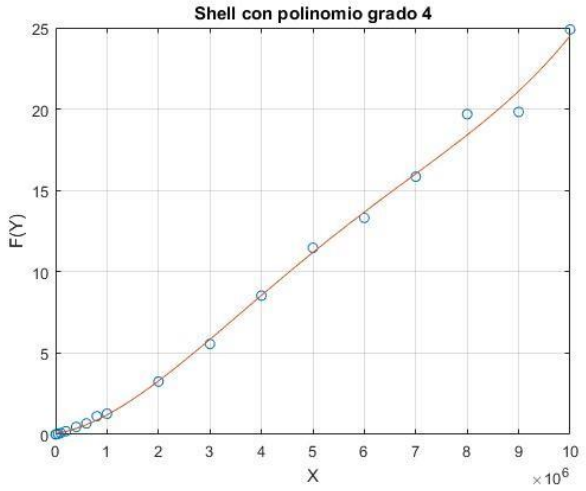
$$1.68630608660661 \times 10^{-52}x^8 - 6.16880127140382 \times 10^{-45}x^7 + 9.03631009235164 \times 10^{-38}x^6 - 6.76258818199406 \times 10^{-31}x^5 + 2.73233154629809 \times 10^{-24}x^4 - 5.80866099227392 \times 10^{-18}x^3 + 6.07450711326964 \times 10^{-12}x^2 - 9.53572024510682 \times 10^{-7}x + 0.0535$$

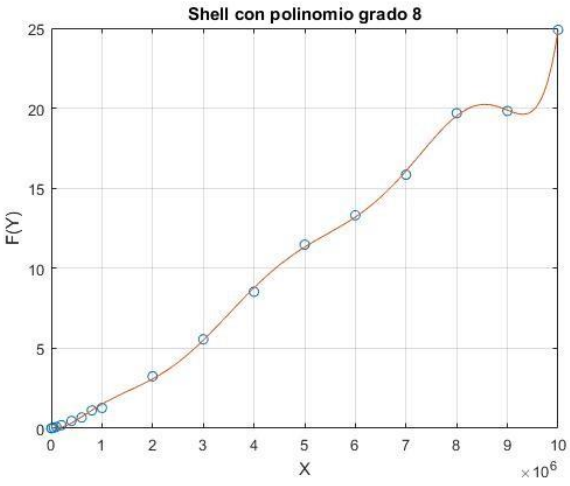
Gráficas polinomios











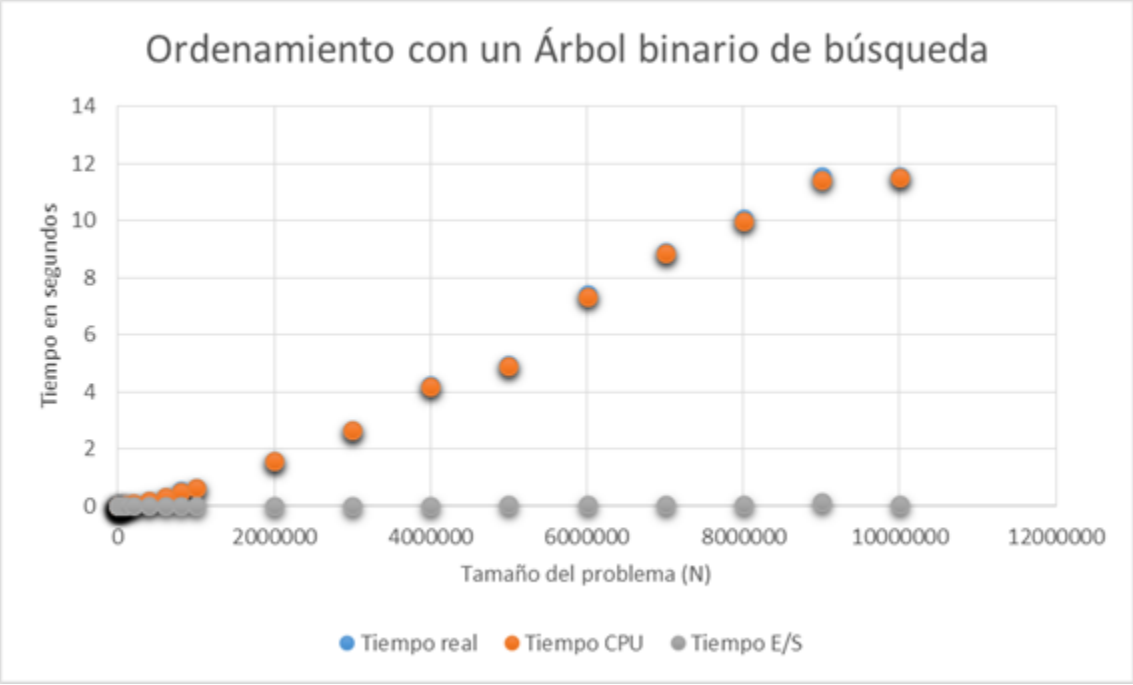
El polinomio escogido como mejor aproximación es:

$$2.38905155086007 \times 10^{-6}x \ - \ 0.5161$$

n	Tiempo con base al polinomio
50,000,000	118.9364775
100,000,000	238.3890551
500,000,000	1,194.009675
1,000,000,000	2,388.535451
5,000,000,000	11,944.74165

Ordenamiento con ABB

Gráfica de tiempo



Polinomios

N=1

$$1.23868836989332 \times 10^{-6}x \; - \; 0.2949$$

N=2

$$2.28323014645839 \times 10^{-14}x^2 \; + \; 1.04174821233608 \times 10^{-6}x \; - \; 0.1728$$

N=3

$$-1.68406177636744 \times 10^{-20}x^3 \; + \; 2.62535129189052 \times 10^{-13}x^2 \; + \; 2.35257471103194 \times 10^{-7}x \; + \; 0.0514$$

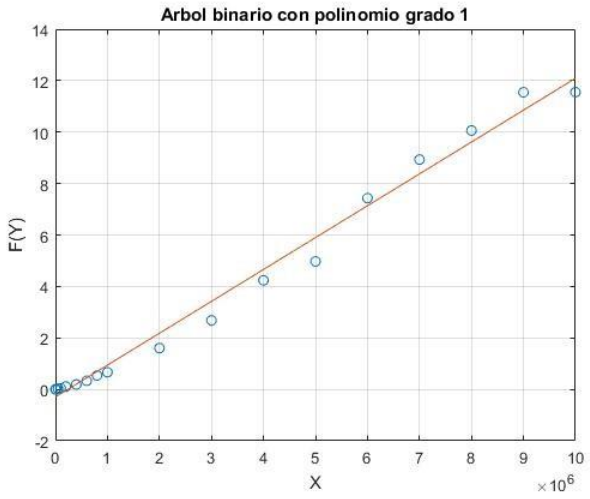
N=4

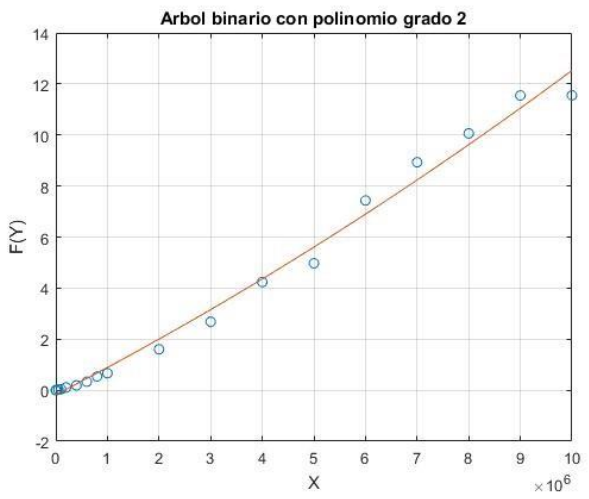
$$-2.37552277394066 \times 10^{-27}x^4 \; + \; 2.91246230817835 \times 10^{-20}x^3 \; - \; 9.21613837237452 \times 10^{-15}x^2 \; + \; 7.19568262106368 \times 10^{-7}x - \; 0.0303$$

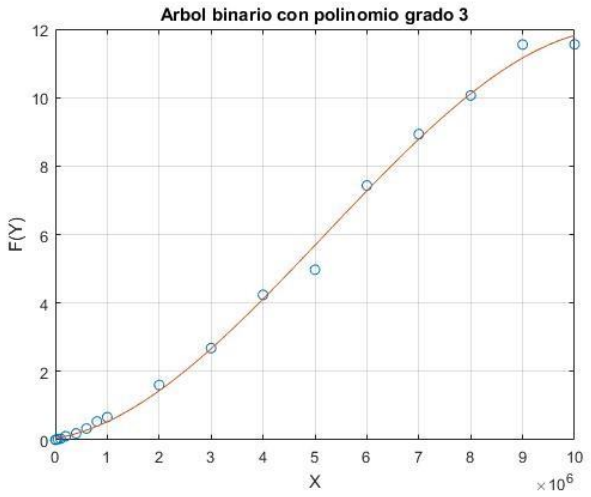
N=8

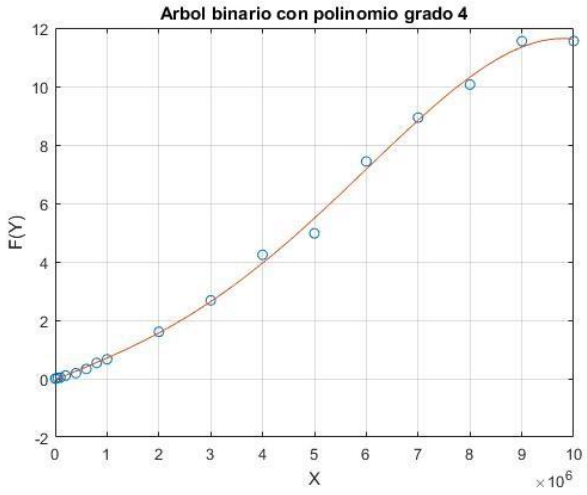
$$-5.58635741189029 \times 10^{-53}x^8 \; + \; 2.10681403254416 \times 10^{-45}x^7 \; - \; 3.19009255543104 \times 10^{-38}x^6 \; + \; 2.47263444820234 \times 10^{-31}x^5 \; - \; 1.03560646854061 \times 10^{-24}x^4 \; + \; 2.26241046069984 \times 10^{-18}x^3 \; - \; 2.15123112344179 \times 10^{-12}x^2 \; + \; 1.31625470894367 \times 10^{-6}x \; - \; 0.0306$$

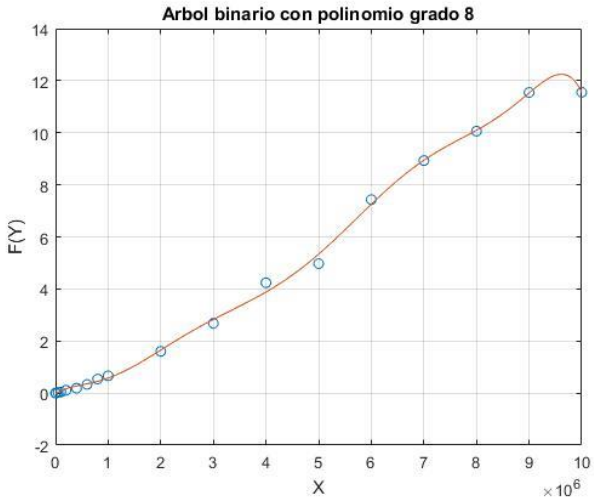
Gráficas polinomios











El polinomio escogido como mejor aproximación es:

$$1.23868836989332 \times 10^{-6}x \; - \; 0.2949$$

n	Tiempo con base al polinomio
50,000,000	61.63951849
100,000,000	123.573937
500,000,000	619.0492849
1,000,000,000	1,238.39347
5,000,000,000	6,193.146949

Comparativo tiempo real

Cuestionario

Responda a las siguientes preguntas:

i. ¿Cuál de los 5 algoritmos es más fácil de implementar?

El ordenamiento por burbuja simple.

ii. ¿Cuál de los 5 algoritmos es el más difícil de implementar?

Tomando en cuenta la implementación de las funciones también, el algoritmo de ordenamiento por ABB, ya que pasar las funciones recursivas a iterativas requirió de una mayor inversión de tiempo para conseguir el objetivo deseado.

iii. ¿Cuál algoritmo tiene menor complejidad temporal?

El ordenamiento con ABB.

iv. ¿Cuál algoritmo tiene mayor complejidad temporal?

El ordenamiento por burbuja simple.

v. ¿Cuál algoritmo tiene menor complejidad espacial? ¿Por qué?

Tanto el ordenamiento por burbuja simple y el algoritmo de inserción tienen la menor complejidad espacial, ya que son los que ocupan menos variables adicionales al usar únicamente dos para los ciclos y una auxiliar para guardar el número a intercambiar.

vi. ¿Cuál algoritmo tiene mayor complejidad espacial? ¿Por qué?

El ordenamiento con ABB debido a que va construyendo una estructura de datos adicional para poder realizar posteriormente el recorrido sobre la misma y mostrar ordenado el arreglo. Esta estructura adicional ocupa su propio espacio de memoria, sumado al que ya tenemos de base y una variable para el ciclo inicial.

vii. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?

En general, sí, dado que al analizar los algoritmos es posible apreciar aproximadamente cómo se van a desarrollar durante la ejecución, sin embargo, algunos cambios en los tiempos fueron impresionantes, sobre todo el ordenamiento por shell, ya que se aproximaba bastante al realizado por ABB y es bastante de recalcar.

viii. ¿Sus resultados experimentales difieren mucho de los del resto de los equipos? ¿A qué se debe?

Comparándolo con los resultados del equipo Compilando Conocimiento, se puede observar que los resultados son distintos. Esto se debe principalmente a la diferencia que se percibe en el hardware, además, el entorno controlado que se manejó fue distinto.

Todo ello da lugar a que los resultados varíen, aunque tampoco es que haya una diferencia notable y al final es casi imperceptible la diferencia.

ix. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?
- Sistema operativo: Ubuntu 17.10
- Compilador: GCC
- Procesador: Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8
- RAM: DDR4 8GB
- GPU: Nvidia GeForce 1050, 4GB GDDR5 VRAM
- Entorno controlado: Todos los programas fueron ejecutados uno por uno de manera exclusiva; es decir, eran las únicas tareas que el procesador estaba realizando.

x. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?

La implementación en sí no es tan compleja, ya que al final lo más complicado es convertir la función recursiva en iterativa; así que es mejor enfocarse en lo que sería preparar el reporte en sí, pero sobre todo, aprender a programar scripts ya que esto puede facilitar mucho la práctica al final.

Errores detectados

Los únicos errores que se encontraron fueron resueltos durante el desarrollo de la práctica, por mencionar algunos:

- Desbordamiento de la memoria.
- Errores menores en la lógica de las condiciones en el pseudocódigo proporcionado.
- Fallos al imprimir los archivos.

Quedaremos atentos ante otros posibles errores que sean encontrados en el futuro, ya que ningún software es perfecto y el número limitado de personas que ocupamos éste, es una muestra muy pequeña como para dar un veredicto sustentado.

Posibles mejoras

Los códigos main pudieron haberse compactado en una solo, para ello hubiera bastado que el script indicara cuál es el que debía ejecutarse, ya que esto ahorraría bastante código y sería más sencillo de entender.

Por otra parte, la impresión de las gráficas pudo haberse hecho de manera inclusiva en el script, sin embargo, esto es más complicado y por falta de tiempo no se consiguió realizar.

Pudo haberse implementado una opción dentro del mismo programa que defina cuando debe detener la ejecución de cierto algoritmo de ordenamiento, es decir, que en algoritmos como el de burbuja simple, automáticamente detecte que no va a llegar al máximo número de elementos y según parámetros del usuario, el propio programa decida hasta qué número podrá ejecutar.

Conclusiones

Calva Hernández José Manuel:

Esta práctica me parece una buena forma de introducir la materia, ya que nos permite observar a grandes rasgos cómo funciona el análisis de algoritmos. Muchas veces hacemos los análisis espaciales y temporales por medio de las cotas ajustadas, sin embargo, no siempre está claro porqué se hace de esta maneta y no de cualquier otra. Al llevar esto al campo práctico, podemos observar que, de manera muy general, los tiempos de algoritmos de orden similar son bastante parecidos.

Esto no quiere decir que sean iguales, es más bien que siguen la misma tendencia y por ello los consideramos similares de manera teórica. Aunado a ello, podemos observar que pequeñas variaciones en los algoritmos, por ejemplo, poner una bandera para optimizar el algoritmo de burbuja, nos permiten reducir de manera considerable el tiempo para n's muy grandes.

Por último, al comparar los resultados obtenidos con otro equipo, se perciben cierta diferencia que nos demuestran cómo afecta el hardware en la ejecución de los programas, ya que llegan a reducir de manera significativa los tiempos. Lo que, es más, la diferencia entre tener un programa abierto a ejecutar únicamente el algoritmo, nos da otra muestra de la importancia de controlar el entorno de ejecución.

Meza Madrid Raúl Damián:

Hemos observado el comportamiento de una de las operaciones más recurrentes de la computación; el ordenamiento de números. Esta vez, gracias a la práctica, logramos apreciar la manera en la que estos algoritmos, gracias a su eficiencia, han mejorado de manera notable este proceso. Aun cuando muchas aplicaciones no requieren de procesar grandes cantidades de números, existen otras aplicaciones más complejas que requieren de ordenar cantidades grandes, la cuales tomarían mucho tiempo de no ser por estos algoritmos, los cuales, aunque han sido implementados hace mucho tiempo, resultan relevantes hoy en día.

Montaño Ayala Alan Israel:

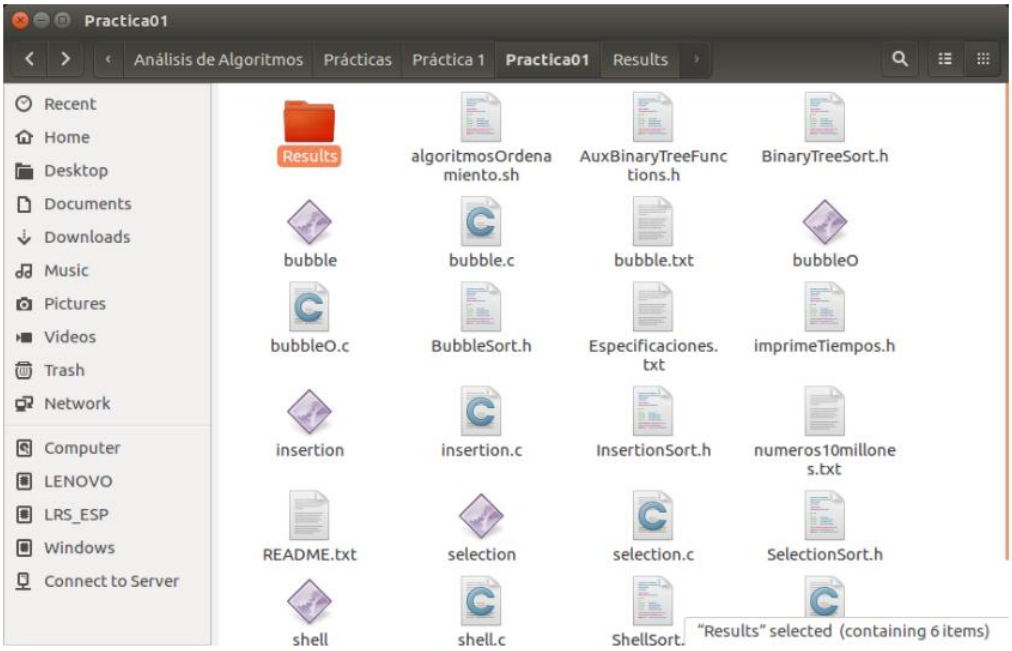
En la práctica se pudo observar el tiempo que tardan diferentes algoritmos al ordenar diferentes cantidades de números, siendo que para algunos de ellos al llegar a cierta cantidad el tiempo que toma es demasiado grande, pero pudiendo apreciar que para cantidades pequeñas el tiempo para todos es relativamente pequeño. Con lo mencionado anteriormente podemos decir que para problemas donde el tamaño del problema sea pequeño (en esta ocasión la cantidad de números), es posible usar cualquiera de estos algoritmos dado que no hay gran diferencia, pero cuando el tamaño del problema crece es muy importante usar un algoritmo adecuado.

Anexo

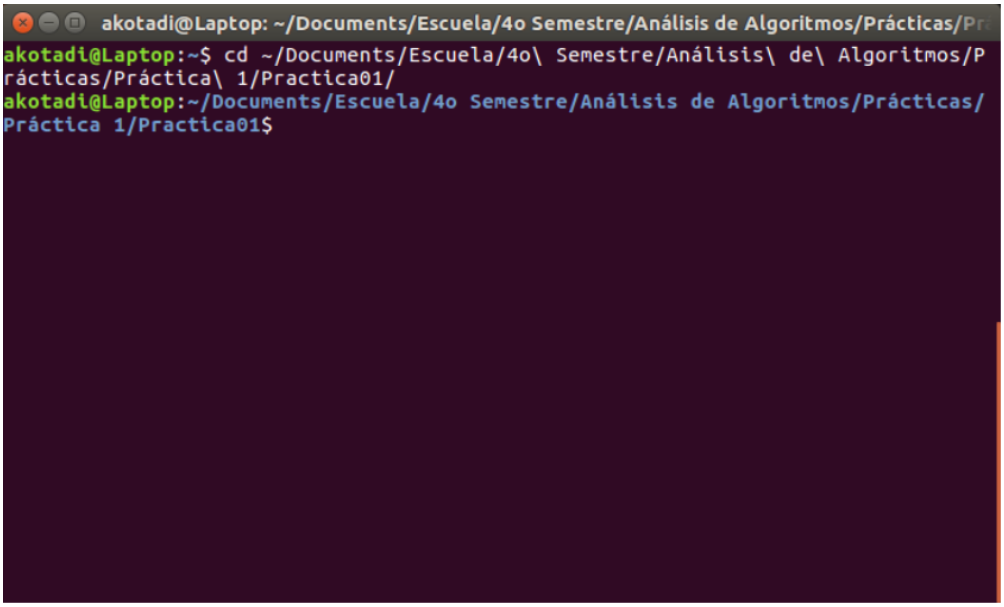
Compilación

En Linux, ejecutar las siguientes instrucciones:

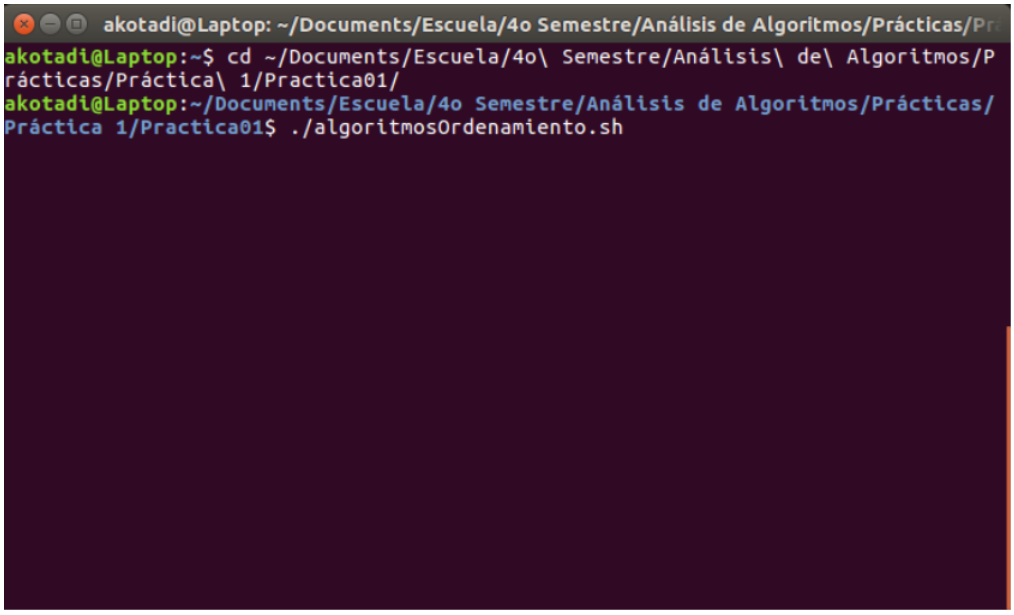
1. Colocar los archivos en una misma carpeta:



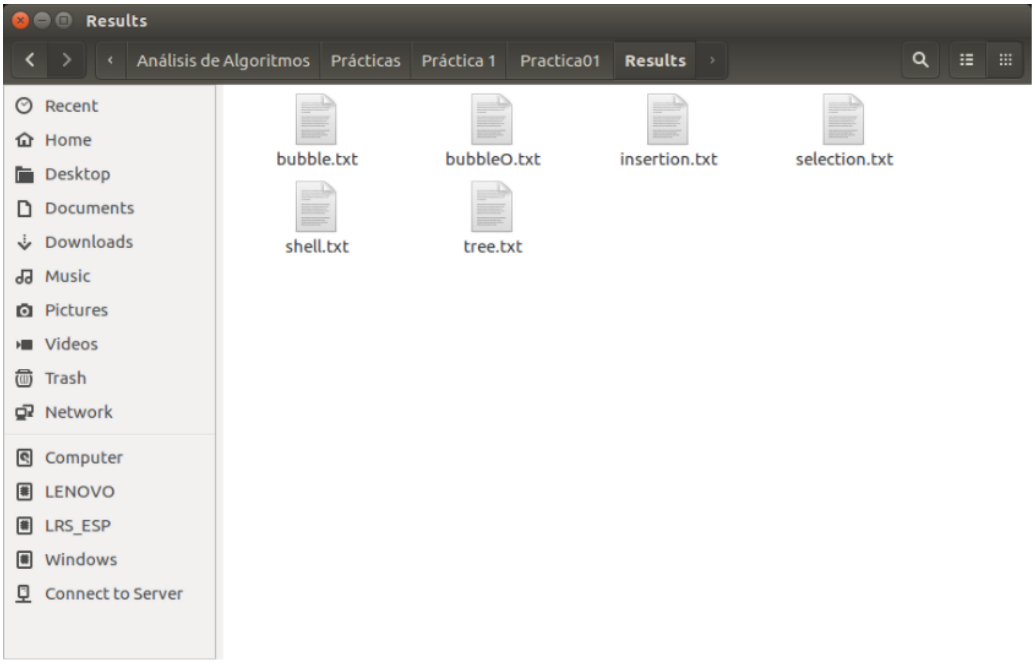
2. Acceder por medio de un terminal hasta la ruta donde se encuentren los archivos:



3. Escribir el siguiente comando para ejecutar el script:



4. Verificar los resultados del código que son impresos en varios txt:



Script

```
#Implementación Práctica 01: Pruebas a posteriori(Algoritmos de Ordenamiento)
# Por: Git Gud(Equipo Arbol)
# Versión: 1.0
# Descripción: Script utilizado para ejecutar la Práctica 01
# Observaciones:

#!/bin/bash
#Arreglo de las cantidades de numeros a ordenar
NUMEROS = (100 1000 5000 10000 50000 100000 200000 400000 600000 800000 1000000 2000000 3000000 40000
00 5000000 6000000 7000000 8000000 9000000 10000000)

# Arreglo de métodos
METODOS = (bubble bubbleO insertion selection shell tree)

# Ciclo para la compilación de los programas
for i in {
    0..5
}
do
    if [$i == 4] then
gcc ${METODOS[i]}.c tiempo.c - o ${METODOS[i]} - lm
else
gcc ${METODOS[i]}.c tiempo.c - o ${METODOS[i]}
fi
done

# Ordenamiento con el método burbuja
for i in {
    0..8
}
do . / ${METODOS[0]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[0]}.txt
done

# Ordenamiento con el método burbuja optimizado
for i in {
    0..8
}
do . / ${METODOS[1]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[1]}.txt
done

# Ordenamiento con el método de inserción
for i in {
    0..9
}
do . / ${METODOS[2]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[2]}.txt
done

# Ordenamiento con el método de selección
for i in {
    0..8
}
do . / ${METODOS[3]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[3]}.txt
done

# Ordenamiento con el método shell
for i in {
    0..19
```

```
}
do . / ${METODOS[4]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[4]}.txt
done

# Ordenamiento con el método de árbol
for i in {
    0..19
}
do . / ${METODOS[5]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[5]}.txt
done

```

Funciones Auxiliares para el ordenamiento con ABB

```
/*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
Por: Git Gud (Equipo Arbol)
Versión: 1.0
Descripción: Funciones Auxiliares del Algoritmo de Ordenamiento por medio de un Árbol Binario de Búsqueda (ABB)
Observaciones: */

//LIBRERÍAS
#include "stdlib.h"

//DEFINICIONES DE ESTRUCTURAS
typedef struct node // Nuestro nodo del árbol contendrá hijo izquierdo, derecho y un número entero
{
    struct node * left, * right;
    int number;
}
node;

//DEFINICIONES DE SINÓNIMOS
typedef node * position; // La posición será la dirección hacia un nodo específico
typedef position arbol_bin; // El árbol binario será simplemente una posición de un nodo, usualmente la raíz del mismo


//DEFINICIÓN DE FUNCIONES
/*Descripción: Función encargada de inicializar la estructura del ABB
Recibe: arbol_bin *BinaryTree (apuntador al ABB declarado por el usuario)
Devuelve:
Observaciones: */
void Initialize(arbol_bin * BinaryTree) {
* BinaryTree = NULL;
// El apuntador enviado por el usuario se coloca en un valor NULL
        return;
    }

/*Descripción: Función encargada de insertar un nuevo elemento en el ABB
Recibe: arbol_bin * BinaryTree (apuntador al ABB utilizado por el usuario),
int newNumber (nuevo elemento que se va a incluir en el ABB)
Devuelve:
Observaciones: */
void Insert(arbol_bin * BinaryTree, int newNumber) {
```

```
40.         arbol_bin * aux = BinaryTree; // Declaramos un apuntador para recorrer el árbol
41.         while ( * aux != NULL) { // Recorremos el árbol hasta encontrar el espacio libre donde irá el
     nuevo elemento
42.             if (newNumber > (( * aux) - > number))
43. // En caso de que el valor sea mayor, iremos a la parte derecha del árbol
44.             {
45.                 aux = & (( * aux) - > right);
46.             } else { // Caso contrario, viajaremos a la parte izquierda del árbol
47.                 aux = & (( * aux) - > left);
48.             }
49.         } * aux = (node * ) malloc(sizeof(node)); // Una vez ubicados en su lugar, le haremos espacio
     en memoria al nuevo nodo
50.         ( * aux) - > number = newNumber;
51. // En el nodo colocaremos el número que desea introducir el usuario al árbol
52.         ( * aux) - > left = NULL;
53. // Nos aseguramos de que ambos hijos estén apuntando a un valor NULL para evitar errores
54.         ( * aux) - > right = NULL;
55.         return;
56.     }
57.
58. /*Descripción: Función encargada de hacer el recorrido inorden por el ABB
59. Recibe: arbol_bin *BinaryTree (apuntador al ABB utilizado por el usuario), int* Data (arreglo que con
    tiene los números), int size (cantidad de números)
60. Devuelve:
61. Observaciones: */
62. void InorderTraversal(arbol_bin * BinaryTree, int * Data, int size) {
63.         position auxPos = * BinaryTree; // Declaramos un apuntador auxiliar para viajar por el árbol

64.         node * * Stack = (node * * ) malloc(size * sizeof(arbol_bin)); // Inicializamos una pila de n
    odos para guardar valores de nuestro viaje
65.         int stackTop = -1, i = 0;
66. // Iniciamos el apuntador del tope de la pila y el índice donde colocaremos los números en el arreglo

67.         do { // Iniciaremos el viaje por el árbol hasta que ya no haya apuntadores por los cuales via
    jar
68.             while (auxPos != NULL) { // Haremos un recorrido hasta llegar a la parte más izquierda a
    partir de cierto nodo
69.                 Stack[++stackTop] = auxPos; // Iremos colocando en la pila los nodos de la izquierda
    que vayamos encontrando
70.                 auxPos = auxPos - > left;
71.             }
72.             if (stackTop >= 0) {
73. // Una vez llegado a la parte más izquierda, verificamos si quedan nodos en la pila
74.                 auxPos = Stack[stackTop--];
75. // Sacaremos el último nodo de la pila que será la "raíz" de ese subárbol
76.                 Data[i++] = auxPos - > number;
77. // Ese nodo será ingresado al arreglo de números, posteriormente moveremos el índice un lugar más
78.                 auxPos = auxPos - > right;
79. // Ya que quitamos la "raíz", pasaremos a recorrer el lado derecho del subárbol
80.             }
81.         } while (auxPos != NULL || stackTop >= 0);
82. // Cuando llegamos a un apuntador nulo y no tenemos más nodos que recorrer en la pila, terminamos
83.         free(Stack); // Liberamos la memoria reservada a la pila
84.         return;
85.     }
86.
87. /*Descripción: Función recursiva encargada de liberar el espacio ocupado por el ABB
88. Recibe: arbol_bin *BinaryTree (apuntador al ABB utilizado por el usuario)
89. Devuelve:
90. Observaciones: */
91. void Destroy(arbol_bin * BinaryTree) {
```

```
92.     if ( * BinaryTree == NULL) // Veirficamos no estar apuntando a un valor nulo en el árbol enviado

93.         return;
94.     else {
95.         if (( * BinaryTree) - > left != NULL)
96. // Verificamos si el árbol izquierdo existe, para eliminarlo primero
97.             Destroy( & (( * BinaryTree) - > left));
98. // Llamamos recursivamente la función por el lado izquierdo
99.         if (( * BinaryTree) - > right != NULL)
100.       // Posteriormente eliminamos el lado derecho del árbol verificando que existe
101.                   Destroy( & (( * BinaryTree) - > right));
102.       // Llamamos recursivamente la función por el lado derecho
103.               free( * BinaryTree); // Liberamos el nodo una vez que ya no tiene hijos
104.               return;
105.           }
106.       }
```

ImprimeTiempos.h

```
1.  /*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
2.  Por: Git Gud (Equipo Arbol)
3.  Versión: 1.0
4.  Descripción: Programa secundario que se encargará de imprimir los tiempos de ejecución de los distint
    os programas
5.  Observaciones: */
6.
7.  //LIBRERÍAS
8.  #include "stdio.h"
9.
10.
11.
12. //DEFINICIÓN DE FUNCIONES
13.
14. /*Descripción: Función encargada de imprimir los tiempos de ejecución de los programas
15. Recibe: int size (número de datos que se ordenaron), double utime0 (tiempo de inicio del usuario),
     double stime0 (tiempo de inicio del sistema), double wtime0 (tiempo de inicio real),    double utime
    1 (tiempo de finalización del usuario), double stime1 (tiempo de finalización del sistema), double wt
    ime1 (tiempo de finalización real)
16. Devuelve:
17. Observaciones: El tiempo se consigue por medio de la diferencia entre el tiempo de inicio y el final,
     el porcentaje de     tiempo dedicado a la ejecución vendría dado por la relación CPU/Wall*/
18. void imprimeTiempos(int size, double utime0, double stime0, double wtime0, double utime1, double stim
    e1, double wtime1) {
19.     printf("%15d", size); // Imprimimos el número con el que estamos trabajando
20.     printf("%25.10f", wtime1 - wtime0);
21. // Los respectivos tiempos estarán dados por la diferencia del inicio y el final
22.     printf("%25.10f", utime1 - utime0);
23.     printf("%25.10f", stime1 - stime0);
24.     printf("%25.10f\n", 100.0 * (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
25. }
```

Main Burbuja Simple

```
1.  /*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
2.  Por: Git Gud (Equipo Arbol)
3.  Versión: 1.0
4.  Descripción: Programa que ordenará por medio del algoritmo de Burbuja Simple
5.  Observaciones: */
6.
7.  //LIBRERÍAS
8.  #include < stdio.h >
9.  #include < stdlib.h >
10. #include "BubbleSort.h"
11. #include "tiempo.h"
12. #include "imprimeTiempos.h"
13.
14.
15.
16. // FUNCIÓN PRINCIPAL
17.
18. /*Variables usadas en el programa:
19. -int size: variable que tomará el tamaño de la línea de comando
20. -int* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
21. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
22. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
23. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
24. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
25. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
26. -double wtime1: variable que medirá el tiempo de finalización de ejecución real*/
27. int main(int argc, char * * argv) {
28.     int i;
29.     int size = atoi(argv[1]);
30.     int * Data;
31.     double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
32.     Data = calloc(size, sizeof(int));
33.     for (i = 0; i < size; i++) {
34.         scanf("%d", Data + i);
35.     }
36.     uswtime( & utime0, & stime0, & wtime0);
37.     SimpleBubbleSort(Data, size);
38.     uswtime( & utime1, & stime1, & wtime1);
39.     imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
40.     free(Data);
41. }
```

Main Burbuja Optimizada

```c
/*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
Por: Git Gud (Equipo Arbol)
Versión: 1.0
Descripción: Programa que ordenará por medio del algoritmo de Burbuja Optimizada
Observaciones: */

//LIBRERÍAS
#include < stdio.h >
#include < stdlib.h >
#include "BubbleSort.h"
#include "tiempo.h"
#include "imprimeTiempos.h"



// FUNCIÓN PRINCIPAL

/*Variables usadas en el programa:
-int size: variable que tomará el tamaño de la línea de comando
-int* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
-double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
-double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
-double wtime0: variable que medirá el tiempo de inicio de ejecución real
-double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
-double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
-double wtime1: variable que medirá el tiempo de finalización de ejecución real*/
int main(int argc, char * * argv) {
    int i;
    int size = atoi(argv[1]);
    int * Data;
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
    Data = calloc(size, sizeof(int));
    for (i = 0; i < size; i++) {
        scanf("%d", Data + i);
    }
    uswtime( & utime0, & stime0, & wtime0);
    OptimizedBubbleSort(Data, size);
    uswtime( & utime1, & stime1, & wtime1);
    imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
    free(Data);
}
```

Main Inserción

```
1.
2.  /*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
3.  Por: Git Gud (Equipo Arbol)
4.  Versión: 1.0
5.  Descripción: Programa que ordenará por medio del algoritmo de Inserción
6.  Observaciones: */
7.
8.  //LIBRERÍAS
9.  #include < stdio.h >
10. #include < stdlib.h >
11. #include "BubbleSort.h"
12. #include "tiempo.h"
13. #include "imprimeTiempos.h"
14.
15.
16.
17. // FUNCIÓN PRINCIPAL
18.
19. /*Variables usadas en el programa:
20. -int size: variable que tomará el tamaño de la línea de comando
21. -int* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
22. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
23. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
24. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
25. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
26. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
27. -double wtime1: variable que medirá el tiempo de finalización de ejecución real*/
28. int main(int argc, char * * argv) {
29.     int i;
30.     int size = atoi(argv[1]);
31.     int * Data;
32.     double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
33.     Data = calloc(size, sizeof(int));
34.     for (i = 0; i < size; i++) {
35.         scanf("%d", Data + i);
36.     }
37.     uswtime( & utime0, & stime0, & wtime0);
38.     InsertionSort(Data, size);
39.     uswtime( & utime1, & stime1, & wtime1);
40.     imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
41.     free(Data);
42. }
43.
```

Main Selección

```
1.
/*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
Por: Git Gud (Equipo Arbol)
Versión: 1.0
Descripción: Programa que ordenará por medio del algoritmo de Selección
Observaciones: */

//LIBRERÍAS
#include < stdio.h >
#include < stdlib.h >
#include "BubbleSort.h"
#include "tiempo.h"
#include "imprimeTiempos.h"



// FUNCIÓN PRINCIPAL

/*Variables usadas en el programa:
-int size: variable que tomará el tamaño de la línea de comando
-int* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
-double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
-double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
-double wtime0: variable que medirá el tiempo de inicio de ejecución real
-double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
-double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
-double wtime1: variable que medirá el tiempo de finalización de ejecución real*/
int main(int argc, char * * argv) {
    int i;
    int size = atoi(argv[1]);
    int * Data;
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
    Data = calloc(size, sizeof(int));
    for (i = 0; i < size; i++) {
        scanf("%d", Data + i);
    }
    uswtime( & utime0, & stime0, & wtime0);
    SelectionSort(Data, size);
    uswtime( & utime1, & stime1, & wtime1);
    imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
    free(Data);
}
```

Main Shell

```
1.
/*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
Por: Git Gud (Equipo Arbol)
Versión: 1.0
Descripción: Programa que ordenará por medio del algoritmo de Shell
Observaciones: */

//LIBRERÍAS
#include < stdio.h >
#include < stdlib.h >
#include "BubbleSort.h"
#include "tiempo.h"
#include "imprimeTiempos.h"



// FUNCIÓN PRINCIPAL

/*Variables usadas en el programa:
-int size: variable que tomará el tamaño de la línea de comando
-int* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
-double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
-double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
-double wtime0: variable que medirá el tiempo de inicio de ejecución real
-double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
-double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
-double wtime1: variable que medirá el tiempo de finalización de ejecución real*/
int main(int argc, char * * argv) {
    int i;
    int size = atoi(argv[1]);
    int * Data;
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
    Data = calloc(size, sizeof(int));
    for (i = 0; i < size; i++) {
        scanf("%d", Data + i);
    }
    uswtime( & utime0, & stime0, & wtime0);
    ShellSort(Data, size);
    uswtime( & utime1, & stime1, & wtime1);
    imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
    free(Data);
}
```

Main ABB

```
1.
/*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
Por: Git Gud (Equipo Arbol)
Versión: 1.0
Descripción: Programa que ordenará por medio del algoritmo de Ordenamiento con ABB
Observaciones: */

//LIBRERÍAS
#include < stdio.h >
#include < stdlib.h >
#include "BubbleSort.h"
#include "tiempo.h"
#include "imprimeTiempos.h"



// FUNCIÓN PRINCIPAL

/*Variables usadas en el programa:
-int size: variable que tomará el tamaño de la línea de comando
-int* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
-double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
-double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
-double wtime0: variable que medirá el tiempo de inicio de ejecución real
-double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
-double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
-double wtime1: variable que medirá el tiempo de finalización de ejecución real*/
int main(int argc, char * * argv) {
    int i;
    int size = atoi(argv[1]);
    int * Data;
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
    Data = calloc(size, sizeof(int));
    for (i = 0; i < size; i++) {
        scanf("%d", Data + i);
    }
    uswtime( & utime0, & stime0, & wtime0);
    BinarySearchTreeSort(Data, size);
    uswtime( & utime1, & stime1, & wtime1);
    imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
    free(Data);
}
```

Tiempo.c

```
1.  //*************************************************************
2.  //TIEMPO.C
3.  //*************************************************************
4.  //*************************************************************
5.  //M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
6.  //Curso: Análisis de algoritmos
7.  //(C) Enero 2013
8.  //ESCOM-IPN
9.  //Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
10. //Compilación de la libreria: "gcc -c tiempo.c " (Generación del código objeto)
11. //*************************************************************
12.
13. //*************************************************************
14. //Librerias incluidas
15. //*************************************************************
16. #include < sys / resource.h >
17. #include < sys / time.h >
18. #include "tiempo.h"
19. //*************************************************************
20. //uswtime (Definición)
21. //*************************************************************
22. //Descripción: Función que almacena en las variables referenciadas
23. //el tiempo de CPU, de E/S y Total actual del proceso actual.
24. //
25. //Recibe: Variables de tipo doble para almacenar los tiempos actuales
26. //Devuelve:
27. //*************************************************************#include <stdio.h>
28. void uswtime(double * usertime, double * systime, double * walltime) {
29.         double mega = 1.0e-6;
30.         struct rusage buffer;
31.         struct timeval tp;
32.         struct timezone tzp;
33.         getrusage(RUSAGE_SELF, & buffer);
34.         gettimeofday( & tp, & tzp); * usertime = (double) buffer.ru_utime.tv_sec + 1.0e-
    6 * buffer.ru_utime.tv_usec; * systime = (double) buffer.ru_stime.tv_sec + 1.0e-
    6 * buffer.ru_stime.tv_usec; * walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
35.     }
36. /*En Unix, se dispone de temporizadores ejecutables (en concreto time) que nos proporcionan medidas d
    e los tiemposde ejecución de programas. Estos temporizadores nos proporcionan tres medidas de tiempo:
        * real:  Tiempo real que se ha tardado desde que se lanzó el programa a ejecutarse hasta que el p
    rograma finalizó y proporcionó los resultados.     * user: Tiempo que la CPU se ha dedicado exclusivam
    ente a la computación del programa.     * sys:   Tiempo que la CPU se ha dedicado a dar servicio al si
    stema operativo por necesidades del programa (por ejemplo para llamadas al sistema para efectuar I/O)
    .El tiempo real también suele recibir el nombre de elapsed time o wall time. Algunos temporizadores t
    ambién proporcionan el porcentaje de tiempo que la CPU se ha dedicado al programa. Este porcentaje vi
    ene dado por la relación entre el tiempo de CPU (user + sys)y el tiempo real, y da una idea de lo car
    gado que se hallaba el sistema en el momento de la ejecución del programa.El grave inconveniente de l
    os temporizadores ejecutables es que no son capaces de proporcionar medidas de tiempo de ejecución de
     segmentos de código. Para ello, hemos de invocar en nuestros propios programas  a un conjunto de tem
    porizadores disponibles en la mayor parte de las librerías de C de Unix, que serán los que nos propor
    cionen medidas sobre los tiempos de ejecución de trozos discretos de código.En nuestras prácticas vam
    os a emplear una función que actúe de temporizador y que nos proporcione los tiempos de CPU (user, sy
    s)y el tiempo real. En concreto, vamos a emplear el procedimiento uswtime listado a continuación. Est
    e procedimiento en realidad invoca a dos funciones de Unix: getrusage y gettimeofday. La primera de e
    llas nos proporciona el tiempo de CPU, tanto de usuario como de sistema, mientras que la segunda nos
    proporciona el tiempo real (wall time). Estas dos funciones son las que disponen de mayor resolución
```

```
   de todos los temporizadores disponibles en Unix. Modo de Empleo: La función uswtime se puede emplear
   para medir los tiempos de ejecución de determinados segmentos de código en nuestros programas. De for
   ma esquemática, el empleo de esta función constaría de los siguientes pasos:    1.- Invocar a uswtime
    para fijar el instante a partir del cual se va a medir el tiempo.            uswtime(&utime0, &stime
   0, &wtime0);    2.- Ejecutar el código cuyo tiempo de ejecución se desea medir.    3.- Invocar a uswt
   ime para establecer el instante en el cual finaliza la medición        del tiempo de ejecución.
         uswtime(&utime1, &stime1, &wtime1);    4.- Calcular los tiempos de ejecución como la diferencia
    entre la primera y segunda        invocación a uswtime:            real:   wtime1 - wtime0
     user:   utime1 - utime0           sys :   stime1 - stime0           El porcentaje de tiempo dedic
   ado a la ejecución de ese segmento de código            vendría dado por la relación CPU/Wall:  CPU/W
   all = (user + sys) / real x 100 %*/
37.
```

Tiempo.h

```
1.  //*****************************************************************
2.  //TIEMPO.H
3.  //*****************************************************************
4.  //*****************************************************************
5.  //M. EN C. EDGARDO ADRIÁN FRANCO MARTÍNEZ
6.  //Curso: Análisis de algoritmos
7.  //(C) Enero 2013
8.  //ESCOM-IPN
9.  //Ejemplo de medición de tiempo en C y recepción de parametros en C bajo UNIX
10. //Compilación de la libreria: "gcc -c tiempo.c " (Generación del código objeto)
11. //*****************************************************************
12. //*****************************************************************
13. //uswtime (Declaración)
14. //*****************************************************************
15. //Descripción: Función que almacena en las variables referenciadas
16. //el tiempo de CPU, de E/S y Total actual del proceso actual.
17. //
18. //Recibe: Variables de tipo doble para almacenar los tiempos actuales
19. //Devuelve:
20. //*****************************************************************
21. void uswtime(double * usertime, double * systime, double * walltime);
22. /* Modo de Empleo:La función uswtime se puede emplear para medir los tiempos de ejecución de determin
    ados segmentos de código en nuestros programas. De forma esquemática, el empleo de esta función const
    aría de los siguientes pasos:    1.- Invocar a uswtime para fijar el instante a partir del cual se va
     a medir el tiempo.            uswtime(&utime0, &stime0, &wtime0);    2.- Ejecutar el código cuyo tie
    mpo de ejecución se desea medir.    3.- Invocar a uswtime para establecer el instante en el cual fina
    liza la medición        del tiempo de ejecución.            uswtime(&utime1, &stime1, &wtime1);    4.
    - Calcular los tiempos de ejecución como la diferencia entre la primera y segunda        invocación a
     uswtime:            real:   wtime1 - wtime0           user:   utime1 - utime0           sys :   st
    ime1 - stime0           El porcentaje de tiempo dedicado a la ejecución de ese segmento de código
             vendría dado por la relación CPU/Wall:    CPU/Wall = (user + sys) / real x 100 %*/
```

Ajuste Mínimos Cuadrados (Scilab)

```
x = [0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2]';
y = [4.8 6.2 6.8 7.2 7.8 9.2 8.8 9.2 8.8 9.2 7.8]';
m = size(x, 1);
t = (x(1): 0.01: x(m))';
n = 1;
A = zeros(m, n + 1);
for i = 0: n A(: , i + 1) = x. ^ i;
end cf = A\ y;
p = poly(cf, 'x', 'c');
ft = horner(p, t);
clf() plot2d(t, ft) p
```

Referencias

[1]T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to algorithms*, 3rd ed. Cambridge, Massachusetts, USA: The MIT Press, 2009.

[2]D. Harel and Y. Feldman, *Algorithmics: The spirit of computing*, 3rd ed. Great Britain: Addison-Wesley, 2004.

[3]S. Baase and A. Van Gelder, *Algoritmos computacionales. Introducción al análisis y diseño*, 3rd ed. México: Pearson Educación, 2002.

[4]G. Brassard and P. Bratley, *Fundamentos de algoritmia*. México: Pearson Educación, 1997.

Bibliografía

S. Skiena, *The algorithm design manual*, 2nd ed. London: Springer, 2008.

T. Cormen, *Algorithms unlocked*. Massachusetts: The MIT Press, 2013.

J. Kleinberg and E. Tardos, *Algorithm design*. Harlow, Essex: Pearson Education, 2006.