Equipo: Git Gud (Equipo Arbol)

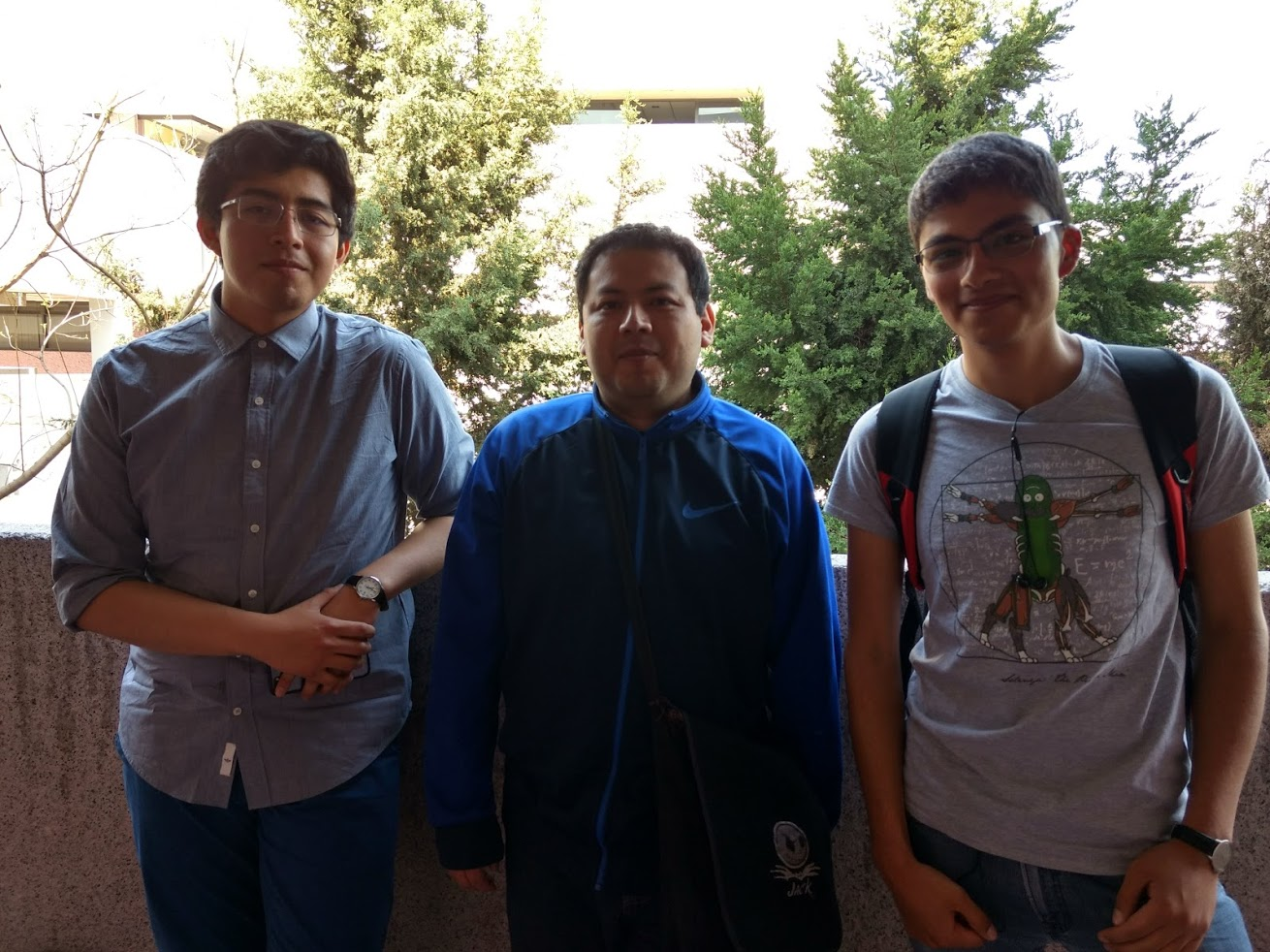
* Calva Hernández José Manuel 2017630201
* Meza Madrid Raúl Damián 2017631051
* Montaño Ayala Alan Israel 2016630260

Análisis de Algoritmos

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

PRUEBAS A POSTERIORI   
(ALGORITMOS DE ORDENAMIENTO)

Práctica 01



Instituto Politécnico Nacional

Escuela Superior de Cómputo

Índice

[Introducción 2](#_Toc508825257)

[Algoritmos 6](#_Toc508825258)

[Burbuja Simple 6](#_Toc508825259)

[Burbuja Optimizada 7](#_Toc508825260)

[Ordenamiento por Inserción 8](#_Toc508825261)

[Ordenamiento por Selección 9](#_Toc508825262)

[Ordenamiento Shell 10](#_Toc508825263)

[Ordenamiento con un Árbol Binario de Búsqueda (ABB) 12](#_Toc508825264)

[Implementación de los algoritmos 13](#_Toc508825265)

[Ordenamiento Burbuja Simple 13](#_Toc508825266)

[Ordenamiento Burbuja Optimizada 13](#_Toc508825267)

[Ordenamiento Inserción 14](#_Toc508825268)

[Ordenamiento Selección 14](#_Toc508825269)

[Ordenamiento Shell 15](#_Toc508825270)

[Ordenamiento con ABB 15](#_Toc508825271)

[Actividades y Pruebas 16](#_Toc508825272)

[Comparativa de tiempo 16](#_Toc508825273)

[Burbuja Simple 17](#_Toc508825274)

[Burbuja Optimizada 19](#_Toc508825275)

[Inserción 22](#_Toc508825276)

[Selección 24](#_Toc508825277)

[Shell 27](#_Toc508825278)

[Ordenamiento con ABB 29](#_Toc508825279)

[Comparativo tiempo real 32](#_Toc508825280)

[Errores detectados 34](#_Toc508825281)

[Posibles mejoras 34](#_Toc508825282)

[Conclusiones 35](#_Toc508825283)

[Anexo 36](#_Toc508825284)

[Referencias 51](#_Toc508825285)

# 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

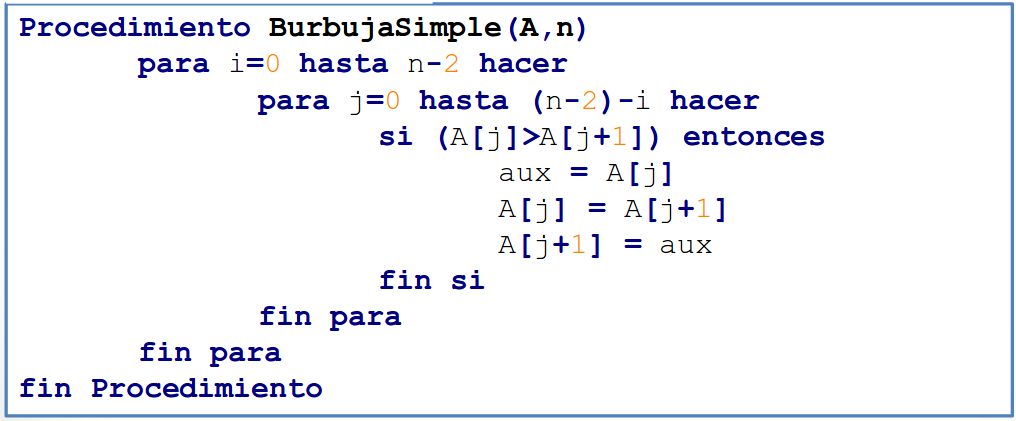
Con base en el archivo de entrada proporcionado que tiene 10,000,000 números diferentes; ordenarlo bajo los siguientes métodos de ordenamiento y comparar experimentalmente las complejidades de estos.

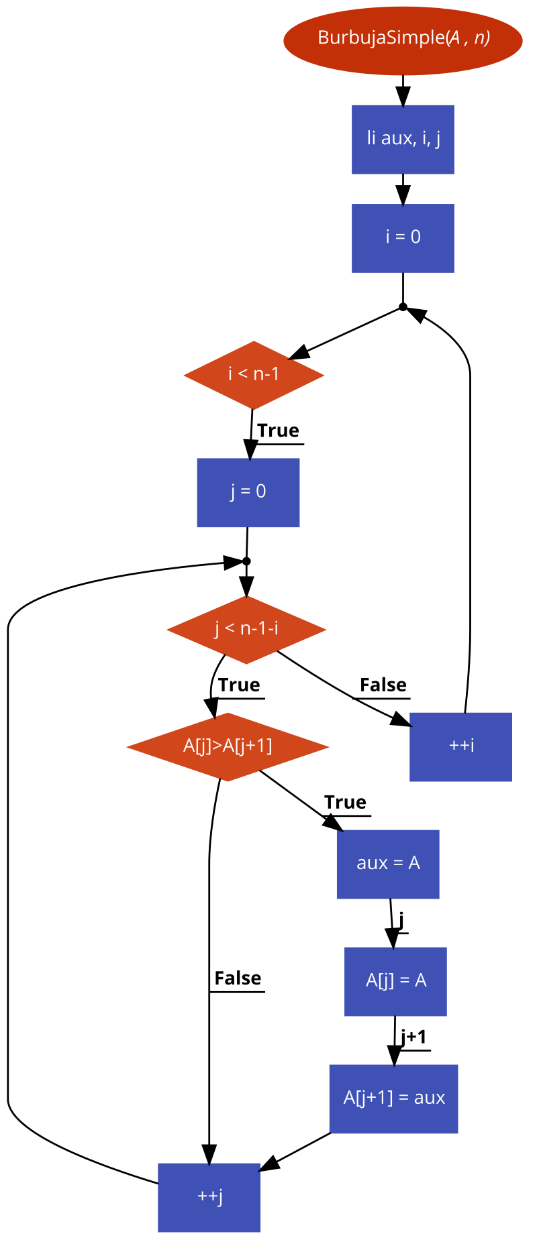
* Burbuja (Bubble Sort)
  + Burbuja Simple
  + Burbuja Optimizada
* Inserción (Insertion Sort)
* Selección (Selection Sort)
* Shell (Shell Sort)
* Ordenamiento con árbol binario de búsqueda (Tree Sort)

# 

# Algoritmos

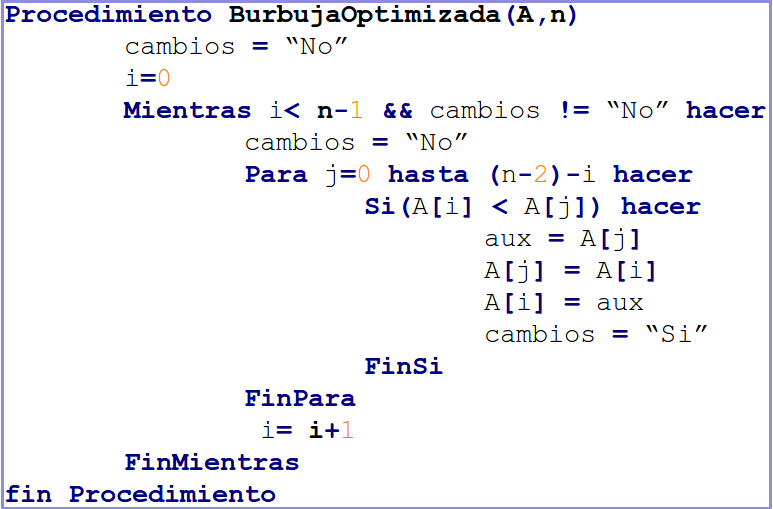
## Burbuja Simple

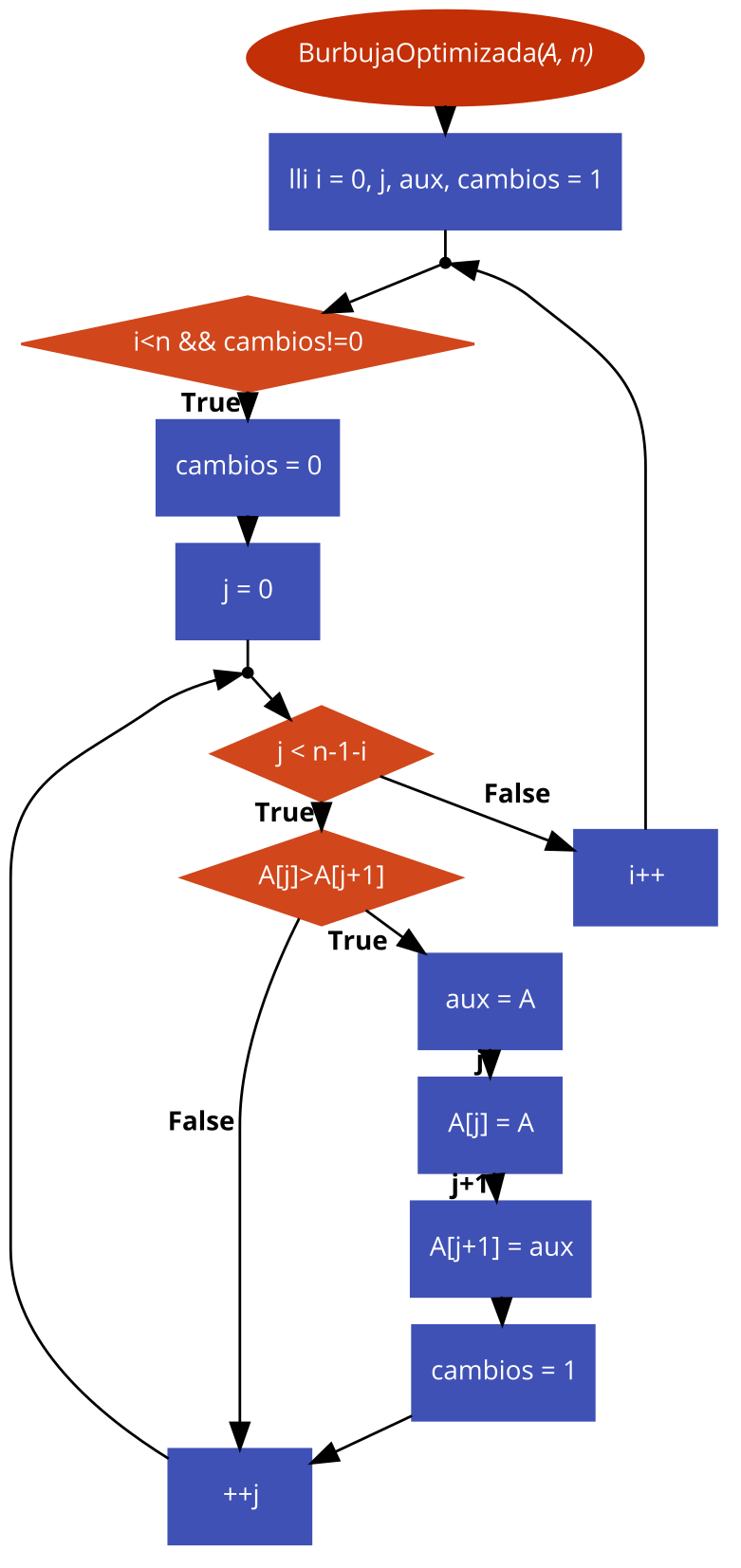
* Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiandolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.
  + El método 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.



## Burbuja Optimizada

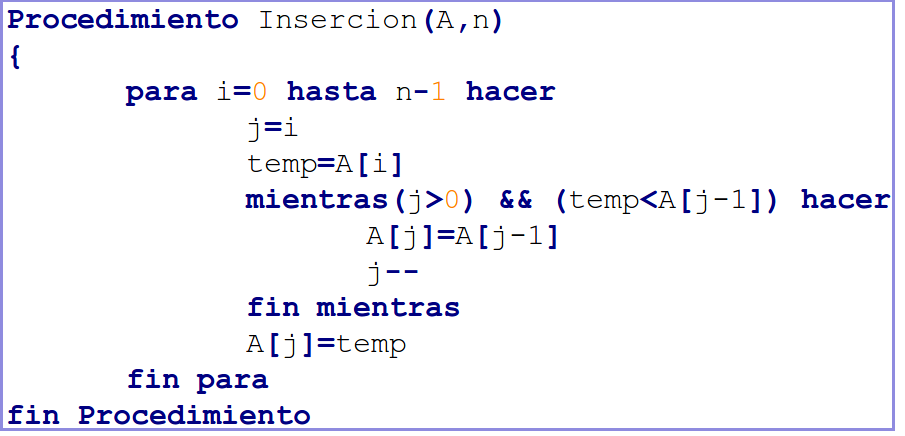
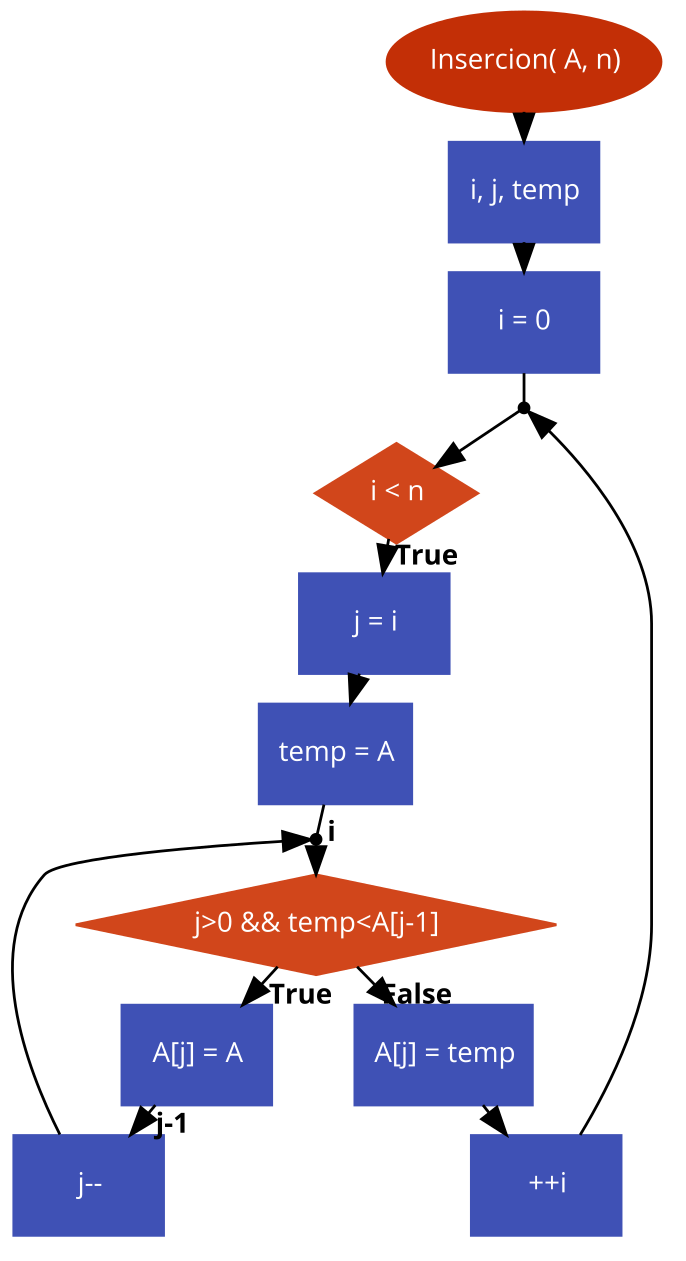
* Como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo así el número de comparaciones por iteración, además pude existir la posibilidad que realizar iteraciones de más si el arreglo ya fue ordenado totalmente.





## 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.

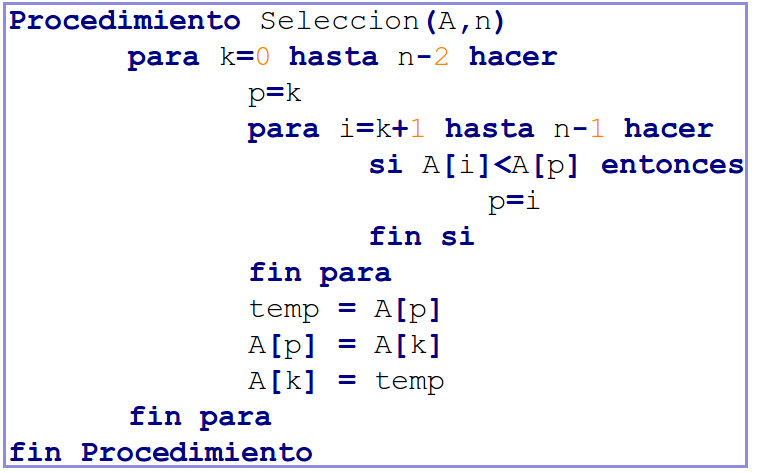


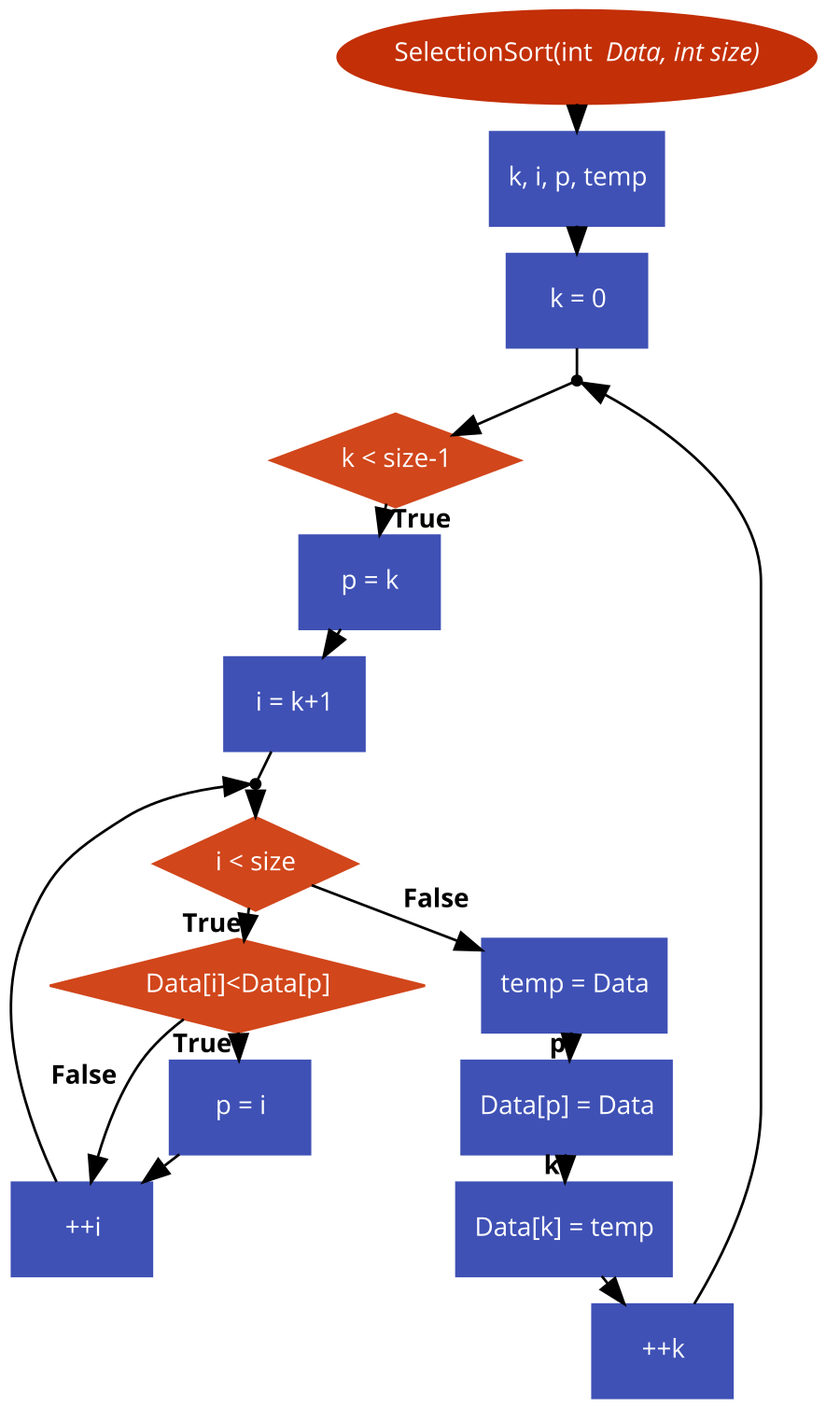
## Ordenamiento por Selección

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



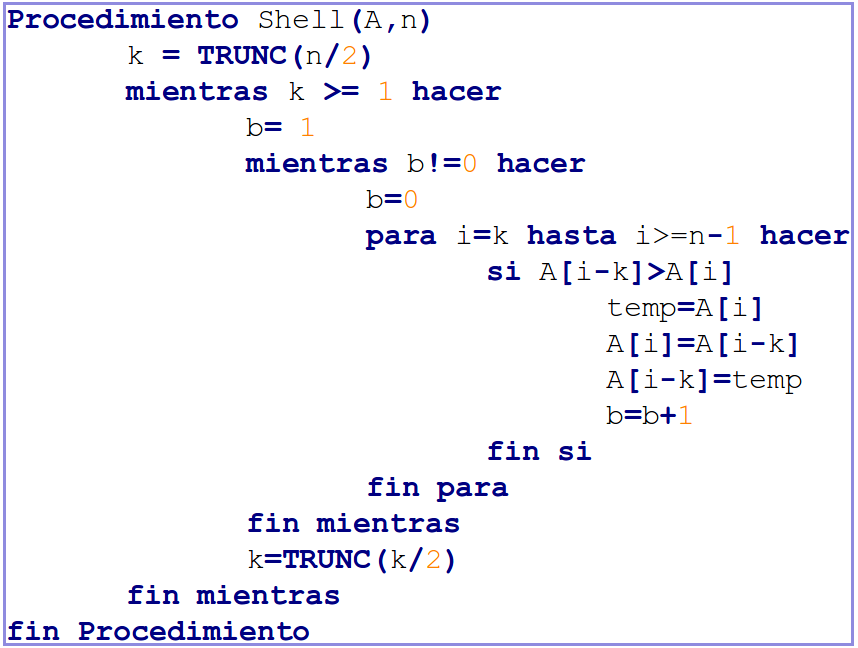


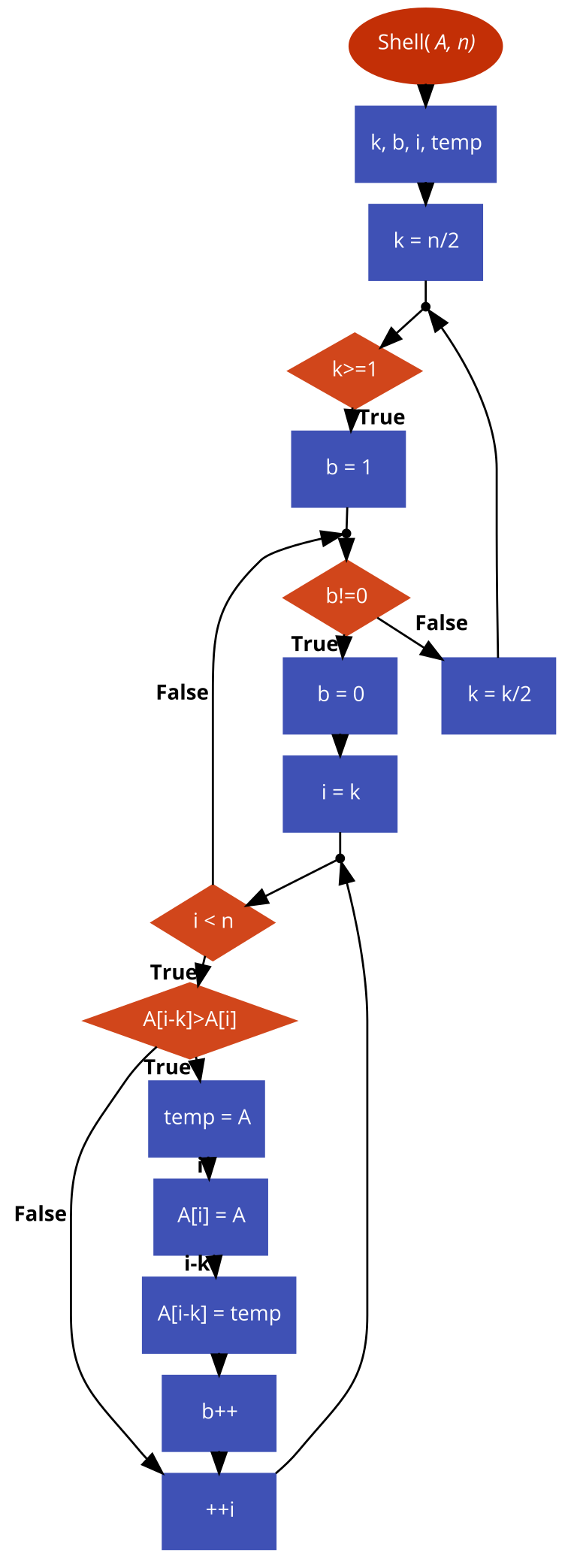
## Ordenamiento Shell

* 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.



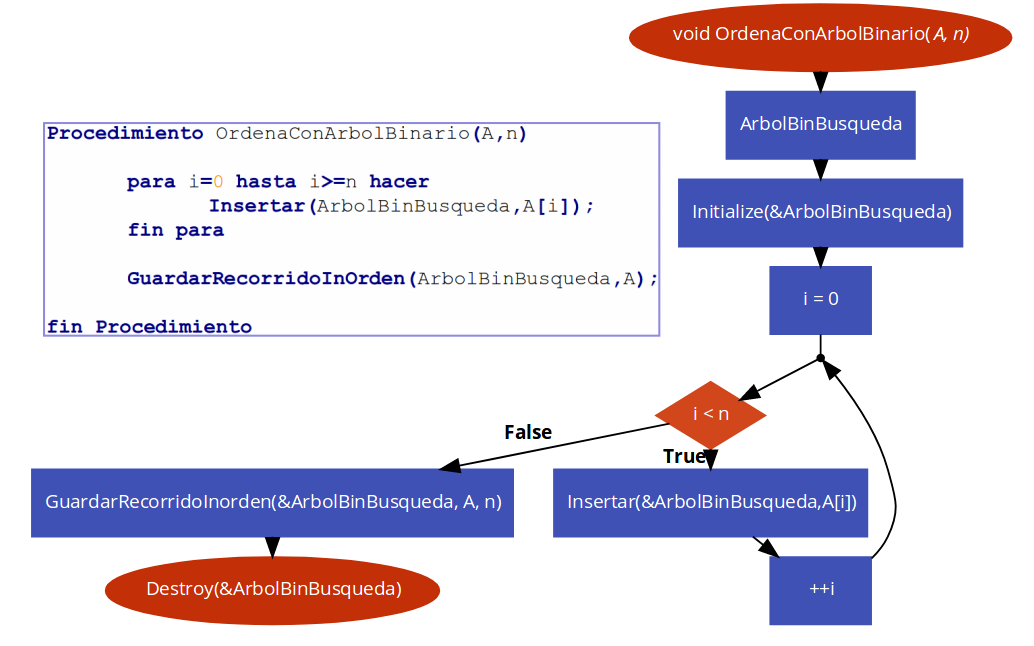


## Ordenamiento con un Árbol Binario de Búsqueda (ABB)

* 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.



# 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 intercambiarán
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 del arreglo de números, o bien, hasta que ya no hayamos realizado cambios, lo que indicará que ya trerminamos 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]; // Ṕosicionamos 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 para 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 los 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

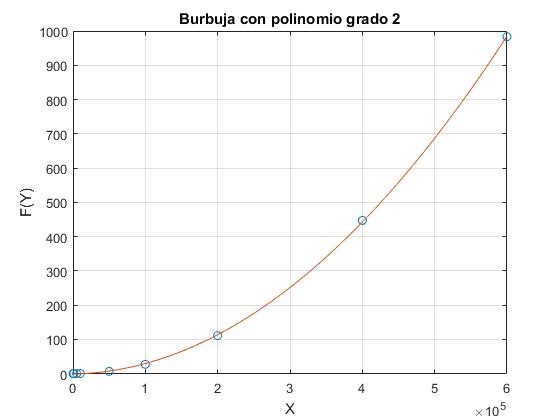
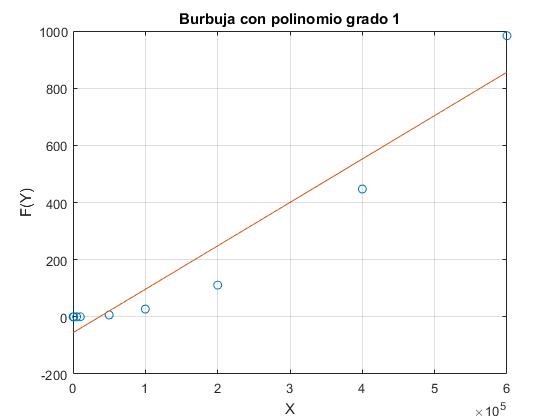
N=2

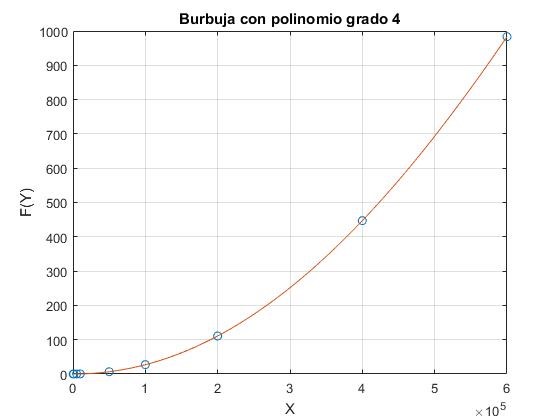
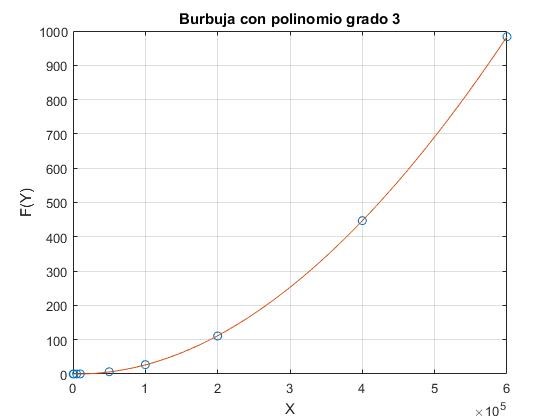
N=3

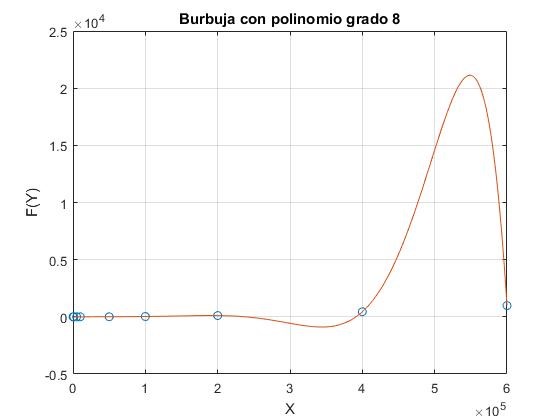
N=4

N=8

**Gráficas polinomios**





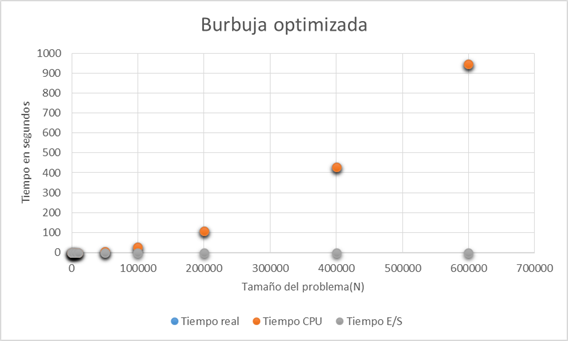


El polinomio escogido como mejor aproximación es:

|  |  |
| --- | --- |
| 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

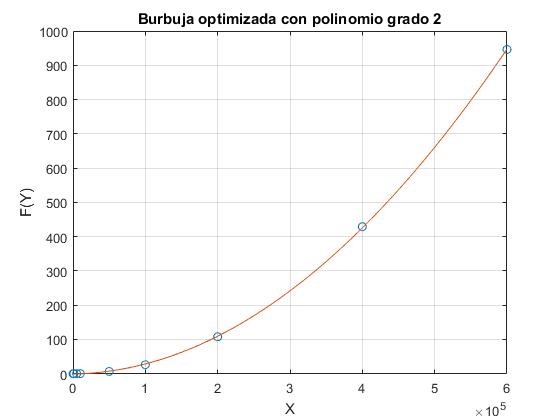
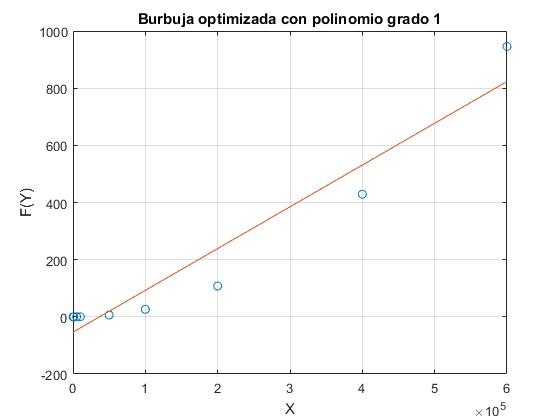
N=2

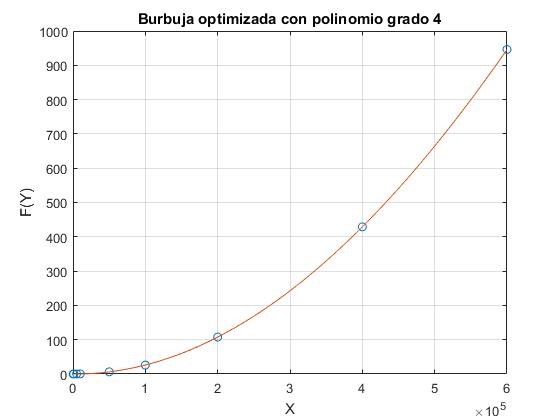
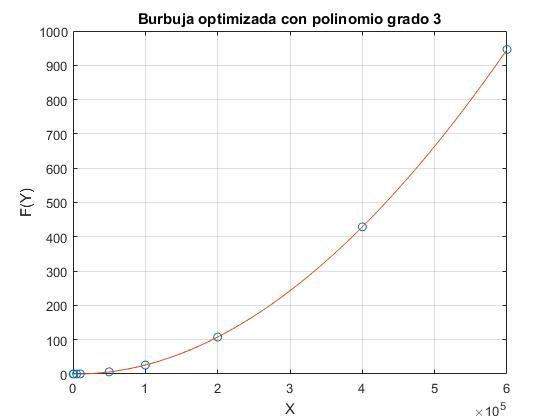
N=3

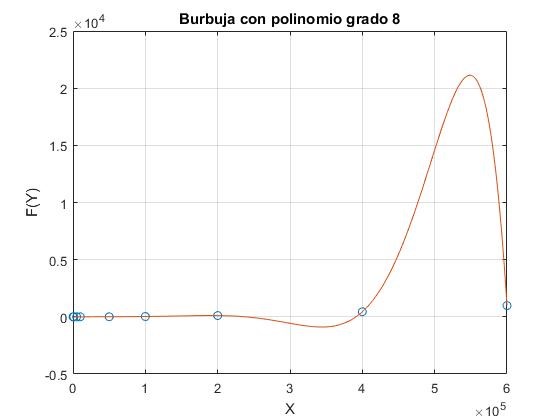
N=4

N=8

**Gráficas polinomios**





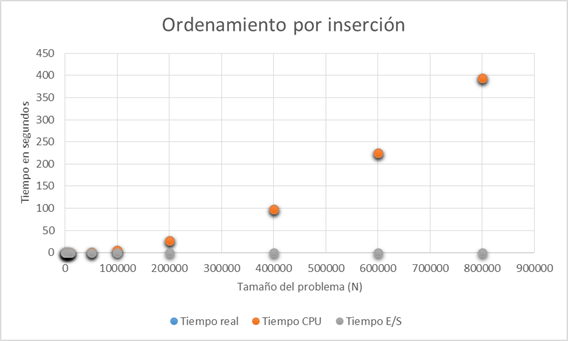


El polinomio escogido como mejor aproximación es:

|  |  |
| --- | --- |
| 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

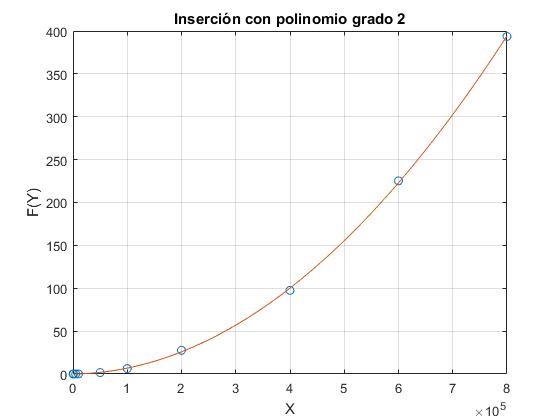
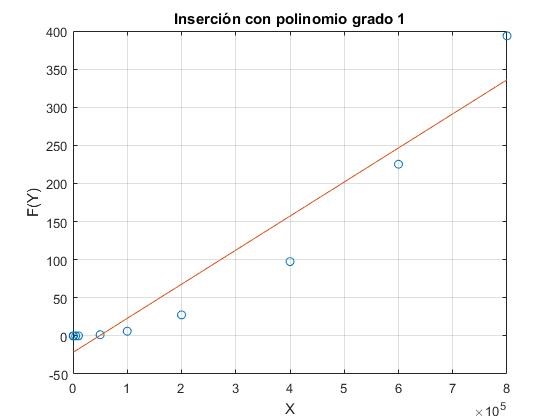
N=2

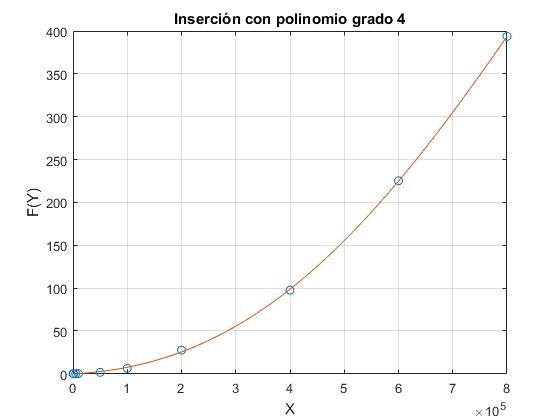
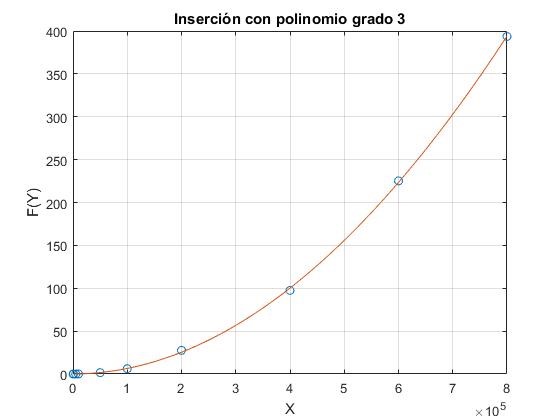
N=3

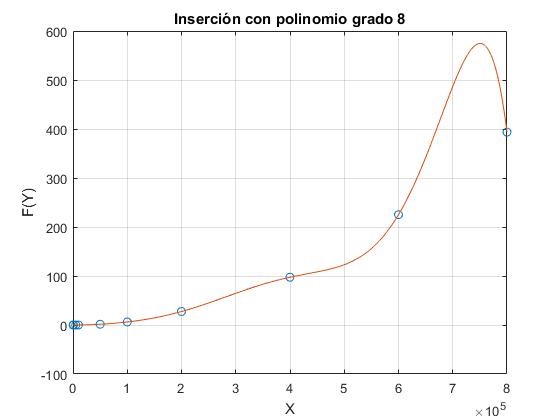
N=4

N=8

**Gráficas polinomios**





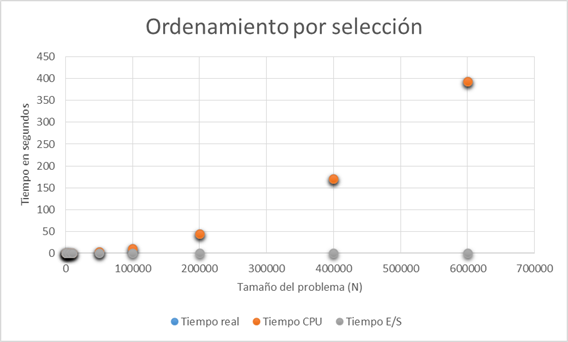


El polinomio escogido como mejor aproximación es:

|  |  |
| --- | --- |
| 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

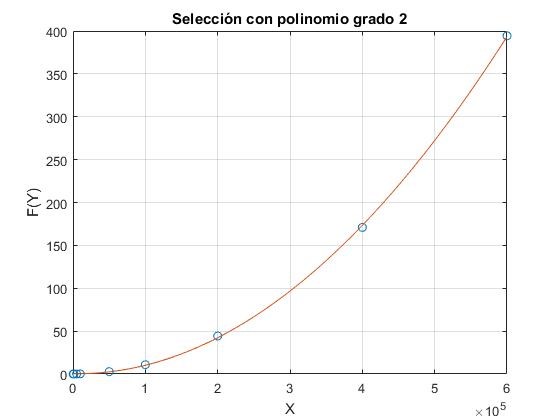
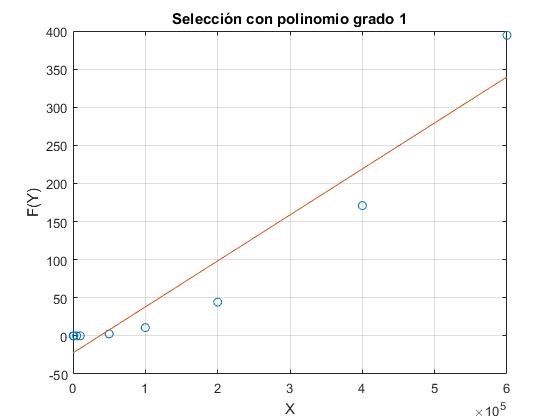
N=2

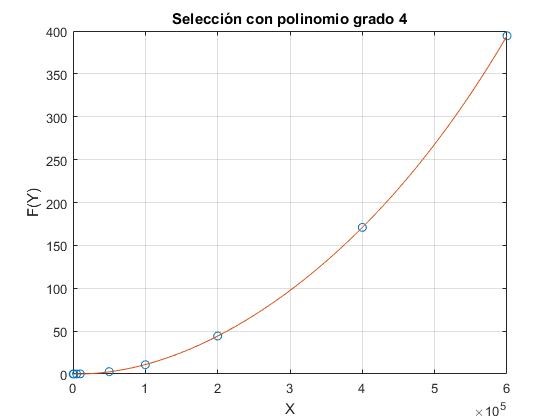
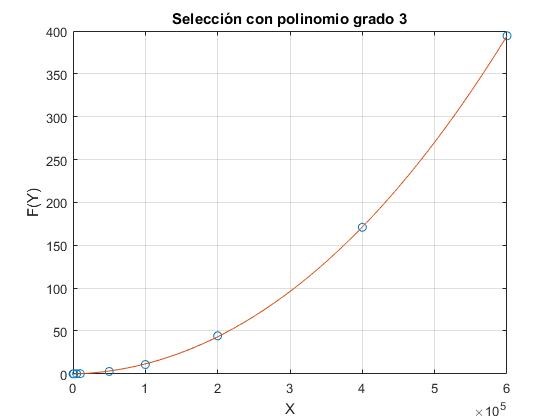
N=3

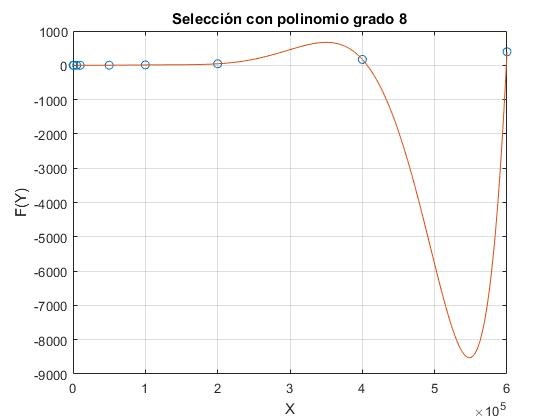
N=4

N=8

**Gráficas polinomios**





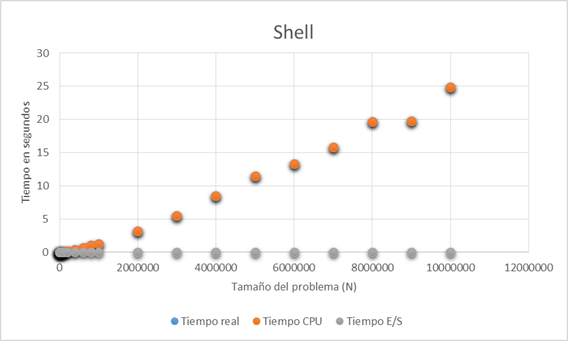


El polinomio escogido como mejor aproximación es:

|  |  |
| --- | --- |
| 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

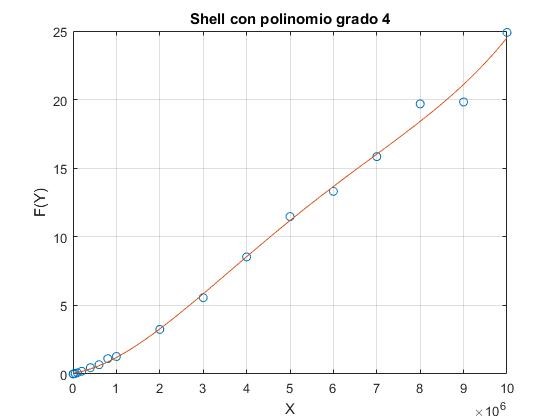
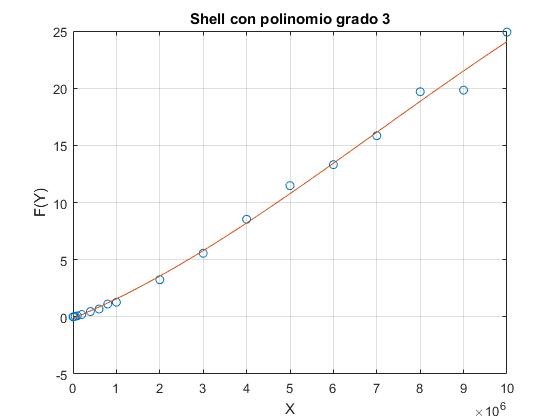
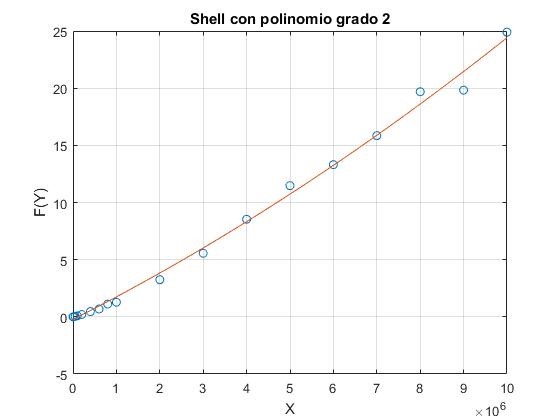
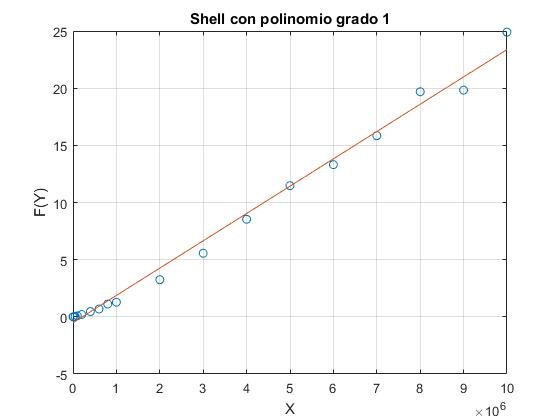
N=2

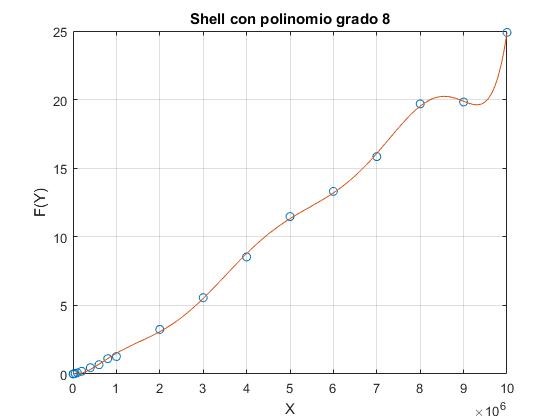
N=3

N=4

N=8

**Gráficas polinomios**



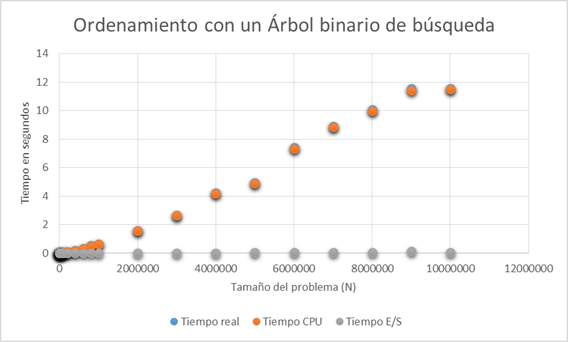


El polinomio escogido como mejor aproximación es:

|  |  |
| --- | --- |
| 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

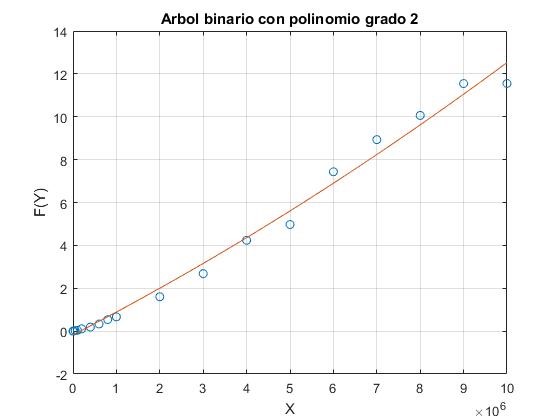
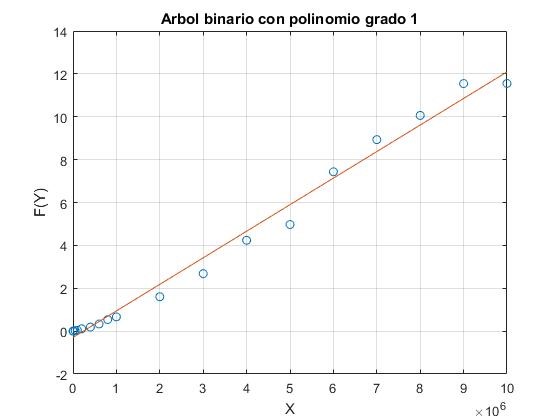
N=2

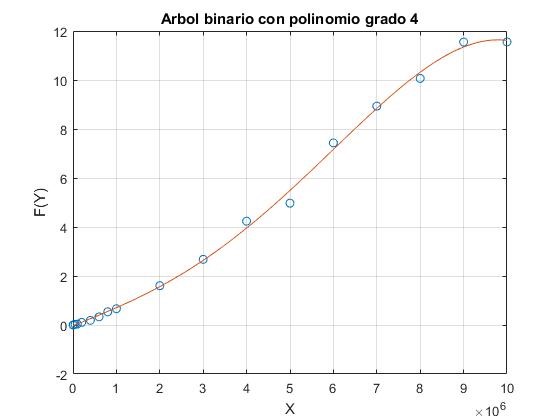
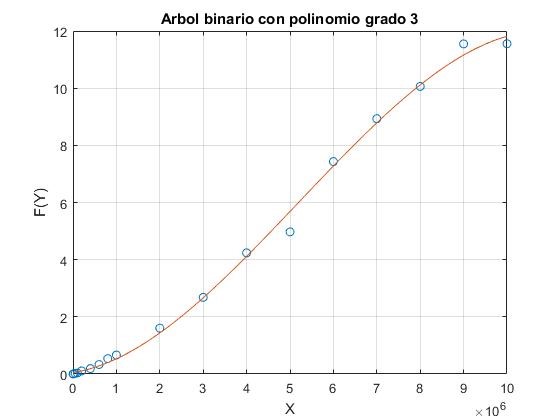
N=3

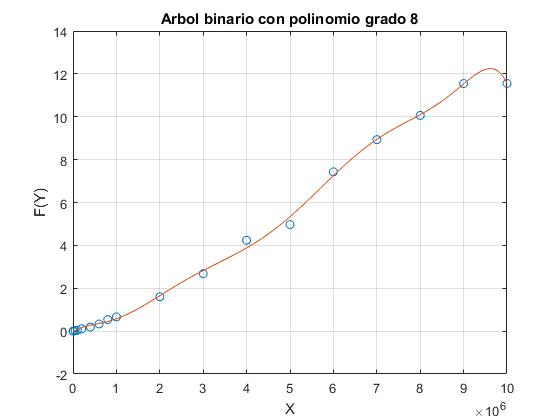
N=4

N=8

**Gráficas polinomios**



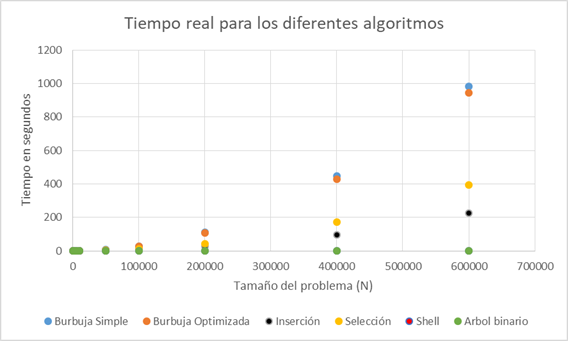




El polinomio escogido como mejor aproximación es:

|  |  |
| --- | --- |
| 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:

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

El ordenamiento por burbuja simple.

1. ¿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.

1. ¿Cuál algoritmo tiene menor complejidad temporal?

El ordenamiento con ABB.

1. ¿Cuál algoritmo tiene mayor complejidad temporal?

El ordenamiento por burbuja simple.

1. ¿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.

1. ¿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.

1. ¿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.

1. ¿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.

1. ¿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.

1. ¿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.

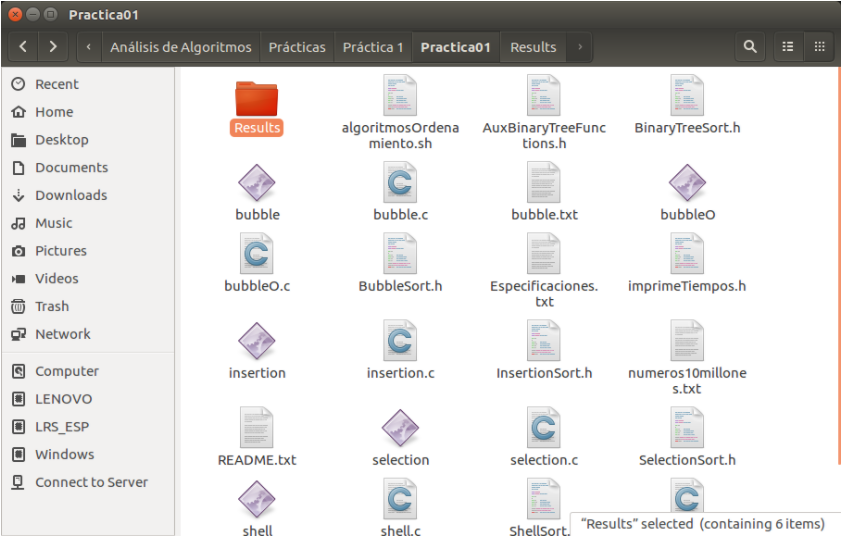
# 

# Anexo

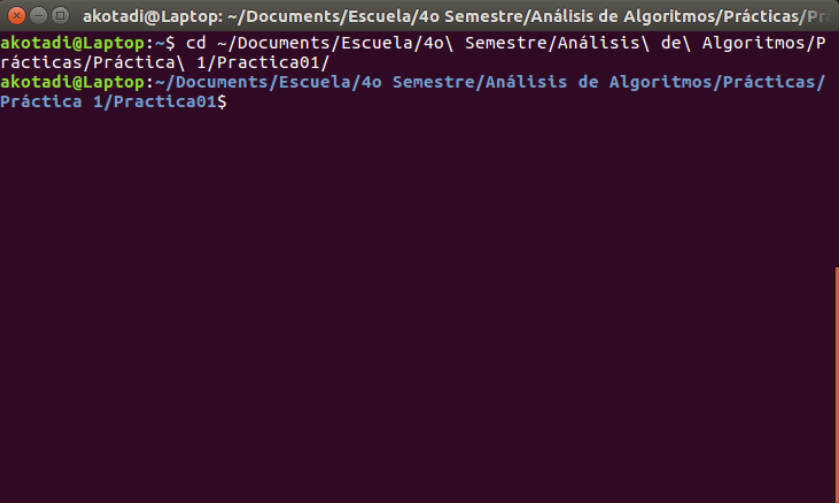
Compilación

En Linux, ejecutar las siguientes instrucciones:

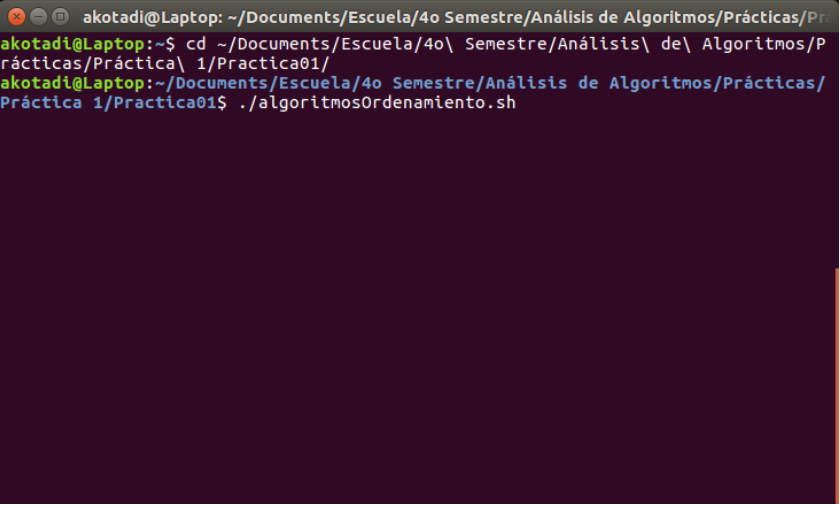
1. Colocar los archivos en una misma carpeta:



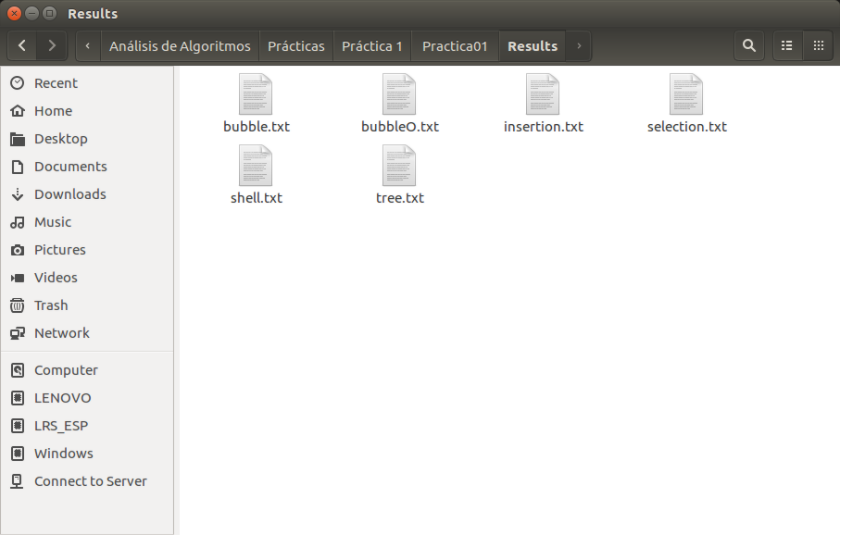
1. Acceder por medio de un terminal hasta la ruta donde se encuentren los archivos:



1. Escribir el siguiente comando para ejecutar el script:



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



Script

1. #Implementación Práctica 01: Pruebas a posteriori(Algoritmos de Ordenamiento)
2. # Por: Git Gud(Equipo Arbol)
3. # Versión: 1.0
4. # Descripción: Script utilizado para ejecutar la Práctica 01
5. # Observaciones:
6. #!/bin/bash
7. #Arreglo de las cantidades de numeros a ordenar
8. NUMEROS = (100 1000 5000 10000 50000 100000 200000 400000 600000 800000 1000000 2000000 3000000 4000000 5000000 6000000 7000000 8000000 9000000 10000000)
9. # Arreglo de métodos
10. METODOS = (bubble bubbleO insertion selection shell tree)
11. # Ciclo para la compilación de los programas
12. for i in {
13. 0..5
14. }
15. do
16. if [$i == 4] then
17. gcc ${METODOS[i]}.c tiempo.c - o ${METODOS[i]} - lm
18. else
19. gcc ${METODOS[i]}.c tiempo.c - o ${METODOS[i]}
20. fi
21. done
22. # Ordenamiento con el método burbuja
23. for i in {
24. 0..8
25. }
26. do . / ${METODOS[0]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[0]}.txt
27. done
28. # Ordenamiento con el método burbuja optimizado
29. for i in {
30. 0..8
31. }
32. do . / ${METODOS[1]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[1]}.txt
33. done
34. # Ordenamiento con el método de inserción
35. for i in {
36. 0..9
37. }
38. do . / ${METODOS[2]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[2]}.txt
39. done
40. # Ordenamiento con el método de selección
41. for i in {
42. 0..8
43. }
44. do . / ${METODOS[3]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[3]}.txt
45. done
46. # Ordenamiento con el método shell
47. for i in {
48. 0..19
49. }
50. do . / ${METODOS[4]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[4]}.txt
51. done
52. # Ordenamiento con el método de árbol
53. for i in {
54. 0..19
55. }
56. do . / ${METODOS[5]} ${NUMEROS[i]} <numeros10millones.txt >> ${METODOS[5]}.txt
57. done

Funciones Auxiliares para el ordenamiento con ABB

1. /\*Implementación Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)
2. Por: Git Gud (Equipo Arbol)
3. Versión: 1.0
4. Descripción: Funciones Auxiliares del Algoritmo de Ordenamiento por medio de un Árbol Binario de Búsqueda (ABB)
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include "stdlib.h"
8. //DEFINICIONES DE ESTRUCTURAS
9. typedef struct node // Nuestro nodo del árbol contendrá hijo izquierdo, derecho y un número entero
10. {
11. struct node \* left, \* right;
12. int number;
13. }
14. node;
15. //DEFINICIONES DE SINÓNIMOS
16. typedef node \* position; // La posición será la dirección hacia un nodo específico
17. typedef position arbol\_bin; // El árbol binario será simplemente una posición de un nodo, usualmente la raíz del mismo
18. //DEFINICIÓN DE FUNCIONES
19. /\*Descripción: Función encargada de inicializar la estructura del ABB
20. Recibe: arbol\_bin \*BinaryTree (apuntador al ABB declarado por el usuario)
21. Devuelve:
22. Observaciones: \*/
23. void Initialize(arbol\_bin \* BinaryTree) {
24. \* BinaryTree = NULL;
25. // El apuntador enviado por el usuario se coloca en un valor NULL
26. return;
27. }
28. /\*Descripción: Función encargada de insertar un nuevo elemento en el ABB
29. Recibe: arbol\_bin \* BinaryTree (apuntador al ABB utilizado por el usuario), int newNumber (nuevo elemento que se va a incluir en el ABB)
30. Devuelve:
31. Observaciones: \*/
32. void Insert(arbol\_bin \* BinaryTree, int newNumber) {
33. arbol\_bin \* aux = BinaryTree; // Declaramos un apuntador para recorrer el árbol
34. while ( \* aux != NULL) { // Recorremos el árbol hasta encontrar el espacio libre donde irá el nuevo elemento
35. if (newNumber > (( \* aux) - > number))
36. // En caso de que el valor sea mayor, iremos a la parte derecha del árbol
37. {
38. aux = & (( \* aux) - > right);
39. } else { // Caso contrario, viajaremos a la parte izquierda del árbol
40. aux = & (( \* aux) - > left);
41. }
42. } \* aux = (node \* ) malloc(sizeof(node)); // Una vez ubicados en su lugar, le haremos espacio en memoria al nuevo nodo
43. ( \* aux) - > number = newNumber;
44. // En el nodo colocaremos el número que desea introducir el usuario al árbol
45. ( \* aux) - > left = NULL;
46. // Nos aseguramos de que ambos hijos estén apuntando a un valor NULL para evitar errores
47. ( \* aux) - > right = NULL;
48. return;
49. }
51. /\*Descripción: Función encargada de hacer el recorrido inorden por el ABB
52. Recibe: arbol\_bin \*BinaryTree (apuntador al ABB utilizado por el usuario), int\* Data (arreglo que contiene los números), int size (cantidad de números)
53. Devuelve:
54. Observaciones: \*/
55. void InorderTraversal(arbol\_bin \* BinaryTree, int \* Data, int size) {
56. position auxPos = \* BinaryTree; // Declaramos un apuntador auxiliar para viajar por el árbol
57. node \* \* Stack = (node \* \* ) malloc(size \* sizeof(arbol\_bin)); // Inicializamos una pila de nodos para guardar valores de nuestro viaje
58. int stackTop = -1, i = 0;
59. // Iniciamos el apuntador del tope de la pila y el índice donde colocaremos los números en el arreglo
60. do { // Iniciaremos el viaje por el árbol hasta que ya no haya apuntadores por los cuales viajar
61. while (auxPos != NULL) { // Haremos un recorrido hasta llegar a la parte más izquierda a partir de cierto nodo
62. Stack[++stackTop] = auxPos; // Iremos colocando en la pila los nodos de la izquierda que vayamos encontrando
63. auxPos = auxPos - > left;
64. }
65. if (stackTop >= 0) {
66. // Una vez llegado a la parte más izquierda, verificamos si quedan nodos en la pila
67. auxPos = Stack[stackTop--];
68. // Sacaremos el último nodo de la pila que será la "raíz" de ese subárbol
69. Data[i++] = auxPos - > number;
70. // Ese nodo será ingresado al arreglo de números, posteriormente moveremos el índice un lugar más
71. auxPos = auxPos - > right;
72. // Ya que quitamos la "raíz", pasaremos a recorrer el lado derecho del subárbol
73. }
74. } while (auxPos != NULL || stackTop >= 0);
75. // Cuando llegamos a un apuntador nulo y no tenemos más nodos que recorrer en la pila, terminamos
76. free(Stack); // Liberamos la memoria reservada a la pila
77. return;
78. }
80. /\*Descripción: Función recursiva encargada de liberar el espacio ocupado por el ABB
81. Recibe: arbol\_bin \*BinaryTree (apuntador al ABB utilizado por el usuario)
82. Devuelve:
83. Observaciones: \*/
84. void Destroy(arbol\_bin \* BinaryTree) {
85. if ( \* BinaryTree == NULL) // Veirficamos no estar apuntando a un valor nulo en el árbol enviado
86. return;
87. else {
88. if (( \* BinaryTree) - > left != NULL)
89. // Verificamos si el árbol izquierdo existe, para eliminarlo primero
90. Destroy( & (( \* BinaryTree) - > left));
91. // Llamamos recursivamente la función por el lado izquierdo
92. if (( \* BinaryTree) - > right != NULL)
93. // Posteriormente eliminamos el lado derecho del árbol verificando que existe
94. Destroy( & (( \* BinaryTree) - > right));
95. // Llamamos recursivamente la función por el lado derecho
96. free( \* BinaryTree); // Liberamos el nodo una vez que ya no tiene hijos
97. return;
98. }
99. }

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 distintos programas
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include "stdio.h"
8. //DEFINICIÓN DE FUNCIONES
10. /\*Descripción: Función encargada de imprimir los tiempos de ejecución de los programas
11. 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 utime1 (tiempo de finalización del usuario), double stime1 (tiempo de finalización del sistema), double wtime1 (tiempo de finalización real)
12. Devuelve:
13. 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\*/
14. void imprimeTiempos(int size, double utime0, double stime0, double wtime0, double utime1, double stime1, double wtime1) {
15. printf("%15d", size); // Imprimimos el número con el que estamos trabajando
16. printf("%25.10f", wtime1 - wtime0);
17. // Los respectivos tiempos estarán dados por la diferencia del inicio y el final
18. printf("%25.10f", utime1 - utime0);
19. printf("%25.10f", stime1 - stime0);
20. printf("%25.10f\n", 100.0 \* (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
21. }

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. //LIBRERÍAS
7. #include < stdio.h >
8. #include < stdlib.h >
9. #include "BubbleSort.h"
10. #include "tiempo.h"
11. #include "imprimeTiempos.h"
12. // FUNCIÓN PRINCIPAL
14. /\*Variables usadas en el programa:
15. -int size: variable que tomará el tamaño de la línea de comando
16. -int\* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
17. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
18. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
19. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
20. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
21. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
22. -double wtime1: variable que medirá el tiempo de finalización de ejecución real\*/
23. int main(int argc, char \* \* argv) {
24. int i;
25. int size = atoi(argv[1]);
26. int \* Data;
27. double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
28. Data = calloc(size, sizeof(int));
29. for (i = 0; i < size; i++) {
30. scanf("%d", Data + i);
31. }
32. uswtime( & utime0, & stime0, & wtime0);
33. SimpleBubbleSort(Data, size);
34. uswtime( & utime1, & stime1, & wtime1);
35. imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
36. free(Data);
37. }

Main Burbuja Optimizada

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 Optimizada
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include < stdio.h >
8. #include < stdlib.h >
9. #include "BubbleSort.h"
10. #include "tiempo.h"
11. #include "imprimeTiempos.h"
12. // FUNCIÓN PRINCIPAL
14. /\*Variables usadas en el programa:
15. -int size: variable que tomará el tamaño de la línea de comando
16. -int\* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
17. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
18. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
19. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
20. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
21. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
22. -double wtime1: variable que medirá el tiempo de finalización de ejecución real\*/
23. int main(int argc, char \* \* argv) {
24. int i;
25. int size = atoi(argv[1]);
26. int \* Data;
27. double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
28. Data = calloc(size, sizeof(int));
29. for (i = 0; i < size; i++) {
30. scanf("%d", Data + i);
31. }
32. uswtime( & utime0, & stime0, & wtime0);
33. OptimizedBubbleSort(Data, size);
34. uswtime( & utime1, & stime1, & wtime1);
35. imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
36. free(Data);
37. }

Main Inserción

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 Inserción
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include < stdio.h >
8. #include < stdlib.h >
9. #include "BubbleSort.h"
10. #include "tiempo.h"
11. #include "imprimeTiempos.h"
12. // FUNCIÓN PRINCIPAL
14. /\*Variables usadas en el programa:
15. -int size: variable que tomará el tamaño de la línea de comando
16. -int\* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
17. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
18. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
19. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
20. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
21. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
22. -double wtime1: variable que medirá el tiempo de finalización de ejecución real\*/
23. int main(int argc, char \* \* argv) {
24. int i;
25. int size = atoi(argv[1]);
26. int \* Data;
27. double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
28. Data = calloc(size, sizeof(int));
29. for (i = 0; i < size; i++) {
30. scanf("%d", Data + i);
31. }
32. uswtime( & utime0, & stime0, & wtime0);
33. InsertionSort(Data, size);
34. uswtime( & utime1, & stime1, & wtime1);
35. imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
36. free(Data);
37. }

Main Selección

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 Selección
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include < stdio.h >
8. #include < stdlib.h >
9. #include "BubbleSort.h"
10. #include "tiempo.h"
11. #include "imprimeTiempos.h"
12. // FUNCIÓN PRINCIPAL
14. /\*Variables usadas en el programa:
15. -int size: variable que tomará el tamaño de la línea de comando
16. -int\* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
17. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
18. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
19. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
20. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
21. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
22. -double wtime1: variable que medirá el tiempo de finalización de ejecución real\*/
23. int main(int argc, char \* \* argv) {
24. int i;
25. int size = atoi(argv[1]);
26. int \* Data;
27. double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
28. Data = calloc(size, sizeof(int));
29. for (i = 0; i < size; i++) {
30. scanf("%d", Data + i);
31. }
32. uswtime( & utime0, & stime0, & wtime0);
33. SelectionSort(Data, size);
34. uswtime( & utime1, & stime1, & wtime1);
35. imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
36. free(Data);
37. }

Main Shell

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 Shell
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include < stdio.h >
8. #include < stdlib.h >
9. #include "BubbleSort.h"
10. #include "tiempo.h"
11. #include "imprimeTiempos.h"
12. // FUNCIÓN PRINCIPAL
14. /\*Variables usadas en el programa:
15. -int size: variable que tomará el tamaño de la línea de comando
16. -int\* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
17. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
18. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
19. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
20. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
21. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
22. -double wtime1: variable que medirá el tiempo de finalización de ejecución real\*/
23. int main(int argc, char \* \* argv) {
24. int i;
25. int size = atoi(argv[1]);
26. int \* Data;
27. double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
28. Data = calloc(size, sizeof(int));
29. for (i = 0; i < size; i++) {
30. scanf("%d", Data + i);
31. }
32. uswtime( & utime0, & stime0, & wtime0);
33. ShellSort(Data, size);
34. uswtime( & utime1, & stime1, & wtime1);
35. imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
36. free(Data);
37. }

Main ABB

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 Ordenamiento con ABB
5. Observaciones: \*/
6. //LIBRERÍAS
7. #include < stdio.h >
8. #include < stdlib.h >
9. #include "BubbleSort.h"
10. #include "tiempo.h"
11. #include "imprimeTiempos.h"
12. // FUNCIÓN PRINCIPAL
14. /\*Variables usadas en el programa:
15. -int size: variable que tomará el tamaño de la línea de comando
16. -int\* Data: apuntador de entero que será inicializado como arreglo para los datos a ordenar
17. -double utime0: variable que medirá el tiempo de inicio de ejecución del usuario
18. -double stime0: variable que medirá el tiempo de inicio de ejecución del sistema
19. -double wtime0: variable que medirá el tiempo de inicio de ejecución real
20. -double utime1: variable que medirá el tiempo de finalización de ejecución del usuario
21. -double stime1: variable que medirá el tiempo de finalización de ejecución del sistema
22. -double wtime1: variable que medirá el tiempo de finalización de ejecución real\*/
23. int main(int argc, char \* \* argv) {
24. int i;
25. int size = atoi(argv[1]);
26. int \* Data;
27. double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición de tiempos
28. Data = calloc(size, sizeof(int));
29. for (i = 0; i < size; i++) {
30. scanf("%d", Data + i);
31. }
32. uswtime( & utime0, & stime0, & wtime0);
33. BinarySearchTreeSort(Data, size);
34. uswtime( & utime1, & stime1, & wtime1);
35. imprimeTiempos(size, utime0, stime0, wtime0, utime1, stime1, wtime1);
36. free(Data);
37. }

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. //Librerias incluidas
14. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
15. #include < sys / resource.h >
16. #include < sys / time.h >
17. #include "tiempo.h"
18. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
19. //uswtime (Definición)
20. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
21. //Descripción: Función que almacena en las variables referenciadas
22. //el tiempo de CPU, de E/S y Total actual del proceso actual.
23. //
24. //Recibe: Variables de tipo doble para almacenar los tiempos actuales
25. //Devuelve:
26. //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*#include <stdio.h>
27. void uswtime(double \* usertime, double \* systime, double \* walltime) {
28. double mega = 1.0e-6;
29. struct rusage buffer;
30. struct timeval tp;
31. struct timezone tzp;
32. getrusage(RUSAGE\_SELF, & buffer);
33. 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;
34. }
35. /\*En Unix, se dispone de temporizadores ejecutables (en concreto time) que nos proporcionan medidas de 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 programa finalizó y proporcionó los resultados.    \* user: Tiempo que la CPU se ha dedicado exclusivamente a la computación del programa.    \* sys:   Tiempo que la CPU se ha dedicado a dar servicio al sistema 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 también proporcionan el porcentaje de tiempo que la CPU se ha dedicado al programa. Este porcentaje viene dado por la relación entre el tiempo de CPU (user + sys)y el tiempo real, y da una idea de lo cargado que se hallaba el sistema en el momento de la ejecución del programa.El grave inconveniente de los 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 temporizadores disponibles en la mayor parte de las librerías de C de Unix, que serán los que nos proporcionen medidas sobre los tiempos de ejecución de trozos discretos de código.En nuestras prácticas vamos a emplear una función que actúe de temporizador y que nos proporcione los tiempos de CPU (user, sys)y el tiempo real. En concreto, vamos a emplear el procedimiento uswtime listado a continuación. Este procedimiento en realidad invoca a dos funciones de Unix: getrusage y gettimeofday. La primera de ellas 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 forma 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, &stime0, &wtime0);    2.- Ejecutar el código cuyo tiempo de ejecución se desea medir.    3.- Invocar a uswtime 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 dedicado a la ejecución de ese segmento de código            vendría dado por la relación CPU/Wall:  CPU/Wall = (user + sys) / real x 100 %\*/

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 determinados segmentos de código en nuestros programas. De forma 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, &stime0, &wtime0);    2.- Ejecutar el código cuyo tiempo de ejecución se desea medir.    3.- Invocar a uswtime 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 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)

1. x = [0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2]';
2. y = [4.8 6.2 6.8 7.2 7.8 9.2 8.8 9.2 8.8 9.2 7.8]';
3. m = size(x, 1);
4. t = (x(1): 0.01: x(m))';
5. n = 1;
6. A = zeros(m, n + 1);
7. for i = 0: n A(: , i + 1) = x. ^ i;
8. end cf = A\ y;
9. p = poly(cf, 'x', 'c');
10. ft = horner(p, t);
11. 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.