Instituto Politécnico Nacional

Escuela Superior de Cómputo

Unidad de aprendizaje: Análisis de Algoritmos

Práctica 01: Pruebas a posteriori (Algoritmos de Ordenamiento)

Equipo: Fuerza.exe

Grupo: 3CM4

Alumnos:

- Oledo Enriquez, Gilberto Irving
- Carrillo Balcazar, Eduardo Yair
- Rojas Alvarado, Luis Enrique

Profesor:
Franco Martínez, Edgardo Adrián

27 de febrero de 2019

Índice

1. Introducción

Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar esa información de manera eficiente es deseable que aquella esté ordenada. Existen varios métodos para ordenar las diferentes estructuras de datos básicas. En general los métodos de ordenamiento no son utilizados con frecuencia, en algunos casos sólo una vez. Hay métodos muy simples de implementar que son útiles en los casos en dónde el número de elementos a ordenar no es muy grande (ej, menos de 500 elementos). Por otro lado hay métodos sofisticados, más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución.

1.1. Definición

1.1.1. Ordenamiento

Es la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento. El ordenamiento se efectúa con base en el valor de algún campo en un registro. El propósito principal de un ordenamiento es el de facilitar las búsquedas de los miembros del conjunto ordenado. El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.

1.1.2. Algoritmo de ordenamiento

En computación y matemáticas un **algoritmo de ordenamiento** es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

1.2. Algoritmos de ordenamiento

1.2.1. Burbuja (Bubble Sort)

El ordenamiento de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos 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. 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. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillos de implementar.

De acuerdo a la implementación de los algoritmos se tienen dos variantes:

- Burbuja Simple
- Burbuja Mejorada

- Burbuja Optimizada

1.2.1.1. Burbuja Simple la burbuja mas simple de todas es la que compara todos con todos, generando comparaciones extras, por ejemplo, no tiene sentido que se compare con sigo mismo o que se compare con los valores anteriores a el, ya que supuestamente, ya están ordenados.

1.2.1.2. Burbuja Mejorada Una nueva versión del método de la burbuja seria limitando el numero de comparaciones, dijimos que era inútil que se compare consigo misma. Si tenemos una lista de `10.000` elementos, entonces son `10.000` comparaciones que están sobrando. Imaginemos si tenemos `1.000.000` de elementos. El método seria mucho mas óptimo con `\n"` comparaciones menos `(n = total de elementos)`.

1.2.1.3. Burbuja Optimizada Si al cambio anterior (el de la burbuja mejorada) le sumamos otro cambio, el hecho que los elementos que están detrás del que se esta comparando, ya están ordenados, las comparaciones serian aun menos y el método seria aun mas efectivo. Si tenemos una lista de `10` elementos y estamos analizando el quinto elemento, que sentido tiene que el quinto se compare con el primero, el segundo o el tercero, si supuestamente, ya están ordenados.

1.2.2. Inserción (Insertion Sort)

El bucle principal de la ordenación por inserción va examinando sucesivamente todos los elementos de la matriz desde el segundo hasta el `n-ésimo`, e inserta cada uno en el lugar adecuado entre sus predecesores dentro de la matriz. La ordenación por selección funciona seleccionando el menor elemento de la matriz y llevándolo al principio; a continuación selecciona el siguiente menor y lo pone en la segunda posición de la matriz y así sucesivamente.

1.2.3. Selección (Selection Sort)

Se basa en buscar el mínimo elemento de la lista e intercambiarlo con el primero, después busca el siguiente mínimo en el resto de la lista y lo intercambia con el segundo, y así sucesivamente

1.2.4. Shell (Shell Sort)

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 çasi 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.

1.2.5. Ordenamiento con árbol binario de búsqueda (Tree Sort)

El ordenamiento con árbol binario es un algoritmo de ordenamiento, el cual ordena sus elementos haciendo uso de un `árbol binario de búsqueda`. Se basa en ir construyendo poco a poco el árbol binario introduciendo cada uno de los elementos, los cuales quedarán ya ordenados. Después, se obtiene la lista de los elementos ordenados recorriendo el árbol en inorden.

1.2.5.1. Árbol binario de búsqueda Un árbol binario de búsqueda también llamado BST (acrónimo del inglés Binary Search Tree) es un tipo particular de árbol binario que presenta una estructura de datos en forma de árbol usada en informática. Para una fácil comprensión queda resumido en que es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores.

2. 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.

1. Burbuja (Bubble Sort)
    - *a*) Burbuja Simple
    - *b*) Burbuja Optimizada
2. Inserción (Insertion Sort)
3. Selección (Selection Sort)
4. Shell (Shell Sort)
5. Ordenamiento con árbol binario de búsqueda (Tree Sort)

3. Diseño de la solución

3.1. Burbuja Simple

Algorithm 1 SimpleBubbleSort

```
1: procedure SimpleBubbleSort(A, n)
2:     for i = 0 to n − 2 do
3:         for j = 0 to (n − 2) − i do
4:             if A[j] > A[j + 1] then
5:                 aux = A[j]
6:                 A[j] = A[j + 1]
7:                 A[j + 1] = aux
8:             end if
9:         end for
10:    end for
11: end procedure
```

3.2. Burbuja optimizada

Algorithm 2 OptimizedBubbleSort

```
1: procedure OptimizedBubbleSort(A, n)
2:     Cambios = 1
3:     i = 0
4:     while (i < n − 1) && (cambios ≠ 0) do
5:         Cambios = 0
6:         for j = 0 to (n − 2) − i do
7:             if A[i] < A[j] then
8:                 aux = A[j]
9:                 A[j] = A[i]
10:                A[i] = aux
11:            end if
12:        end for
13:        i = i + 1
14:    end while
15: end procedure
```

3.3. Inserción

Algorithm 3 InsertionSort

```
1: procedure InsertionSort(A, n)
2:     for i = 0 to n − 1 do
3:         j = i
4:         temp = A[i]
5:         while (j > 0) && (temp < A[j − 1]) do
6:             A[j] = A[j − 1]
7:             j − −
8:         end while
9:         A[j] = temp
10:    end for
11: end procedure
```

3.4. Selección

Algorithm 4 SelectionSort

```
1: procedure SelectionSort(A, n)
2:     for k = 0 to n − 2 do
3:         p = k
4:         for i = k + 1 to n − 1 do
5:             if A[i] < A[p] then
6:                 p = i
7:             end if
8:         end for
9:         temp = A[p]
10:        A[p] = A[k]
11:        A[k] = temp
12:    end for
13: end procedure
```

3.5. Shell

Algorithm 5 ShellSort

```
1:  procedure ShellSort(A, n)
2:      k = TRUNC(n/2)
3:      while k ≥ 1 do
4:          b = 1
5:          while b ≠ 0 do
6:              b = 0
7:              for i = k to i ≥ n − 1 do
8:                  if A[i − k]¿A[i] then
9:                      temp = A[i]
10:                     A[i] = A[i − k]
11:                     A[i − k] = temp
12:                     b = b + 1
13:                 end if
14:             end for
15:         end while
16:         k = TRUNC(k/2)
17:     end while
18: end procedure
```

3.6. Ordenamiento con árbol binario de búsqueda

Algorithm 6 SortWithBST

```
1: procedure SortWithBST(A, n)
2:     for i = 0 to i ≥ n do
3:         Insertar(ArbolBinBusqueda, A[i]);
4:     end for
5:     GuardarRecorridoInOrden(ArbolBinBusqueda, A[i]);
6: end procedure
```

4. Implementación de la Solución

4.0.1. BubbleSort

Código (`BubbleSort.c`):

```
/*****************************************
 *    Instituto Politecnico Nacional
 *    Escuela Superior de Computo
 *    Analisis de Algoritmos
 *    Fecha: 21/02/2018
 *    Autores:
 *            Oledo Enriquez Gilberto Irving
 *            Carrillo Balcazar Eduardo Yair
 *            Rojas Alvarado Luis Enrique
 *     Version: 1.0
 *     Equipo: Fuerza.exe
*****************************************/

/*=====================================================================
====================      FUNCIONES     ============================
=====================================================================*/

/****************************************
 * Nombre de la funcion: BubbleSortv1
 * Descripcion:
 *             Este es el algoritmo (bubble sort) más trivial, sin optimización nada, el algoritmo en bruto.
 *             Para cada elemento en el Array, compruebe si hay 2 elementos contiguos que están en el orden incorrecto,
 *             si es valido, intercambia.
 * Parametros:
 *         Parametro Data         Un puntero a la matriz de int para ordenar.
 *         Parametro DataSize  El tamaño de la matriz de datos.
 * Return:
         Nada.
 ****************************************/

void BubbleSortv1(int Data[], int DataSize) {                  //=== BubbleSortv1 ======
    for (int i = 0; i < DataSize - 1; ++i) {                   //Hacer esto (Size - 1) veces
        for (int j = 0; j < DataSize - 1; ++j) {               //Hacer esto (Size - 1) veces
            if (Data[j] > Data[j + 1])                         //Si necesitamos intercambiar
                Swap(&Data[j], &Data[j + 1]);                  //Intercambiamos!
        }
    }
}



/****************************************
 * Nombre de la funcion: BubbleSortv2
 * Descripcion:
```

```
45  *             Esta es la optimización más importante, reducir el segundo bucle porque los
    ↪  últimos
46  *             elementos i ya están en su lugar.
47  * Parametros:
48  *         Parametro Data         Un puntero a la matriz de int para ordenar.
49  *         Parametro DataSize  El tamaño de la matriz de datos.
50  * Return:
51          Nada.
52  ****************************************/
53
54  void BubbleSortv2(int Data[], int DataSize) {                    //=== BubbleSortv2 ======
55      for (int i = 0; i < DataSize - 1; i++) {                     //Hacer esto (Size - 1)
        ↪  veces
56          for (int j = 0; j < DataSize - i - 1; j++) {             //Los últimos (i) están
            ↪  ordenados
57              if (Data[j] > Data[j + 1])                           //Si necesitamos
                ↪  intercambiar
58                  Swap(&Data[j], &Data[j + 1]);                    //Intercambiamos!
59          }
60      }
61  }
62
63
64  /****************************************
65   * Nombre de la funcion: BubbleSortv3
66   * Descripcion:
67   *             Esta es la siguiente optimización más importante, romper la búsqueda si ya
    ↪  está ordenada.
68   * Parametros:
69   *         Parametro Data         Un puntero a la matriz de int para ordenar.
70   *         Parametro DataSize  El tamaño de la matriz de datos.
71   * Return:
72          Nada.
73   ****************************************/
74  void BubbleSortv3(int Data[], int DataSize) {                     //=== BubbleSortv3
    ↪  ======
75
76      for (int i = 0; i < DataSize - 1; i++) {                     //Hacer esto (Size - 1)
        ↪  veces
77          bool Swapped = false;                                    //Supongamos que no hay
            ↪  intercambio
78
79          for (int j = 0; j < DataSize - i - 1; j++) {             //Los últimos (i) están
            ↪  ordenados
80              if (Data[j] > Data[j + 1]) {                         //¿Necesitamos
                ↪  intercambiar?
81                  Swap(&Data[j], &Data[j + 1]);                    //Intercambiamos!
82                  Swapped = true;                                  //Intercambio es TRUE
83              }
84          }
85
86          if (!Swapped) break;                                     //No los cambies, estan
            ↪  ordenados!
87      }
88  }
```

4.0.2. InsertionSort

Código (`InsertionSort.c`):

```
1  /*****************************************
2   *    Instituto Politecnico Nacional
3   *    Escuela Superior de Computo
4   *    Analisis de Algoritmos
5   *    Fecha: 21/02/2018
6   *    Autores:
7   *            Oledo Enriquez Gilberto Irving
8   *            Carrillo Balcazar Eduardo Yair
9   *            Rojas Alvarado Luis Enrique
10  *     Version: 1.0
11  *     Equipo: Fuerza.exe
12  *****************************************/
13
14
15  /*=====================================================================
16  ====================      FUNCIONES     ============================
17  =====================================================================*/
18
19  /**************************************
20   * Nombre de la funcion: InsertionSort
21   * Descripcion:
22   *             Este es un algoritmo de clasificación realmente intuitivo, para ello decimos
     ↪  que crearemos un nuevo
23   *             subarray. Entonces el subarray [0, 1] ya está ordenado, ahora:
24   *             Llevamos el siguiente elemento contiguos al subarreglo, lo ponemos donde
     ↪  pertenece y movemos todos los
25   *             otros 1 lugar.
26   *             Ahora nuestro nuevo subarreglo es de [0, 2], repetimos (n-1) y listo.
27   * Parametros:
28   *         Parametro Data         Un puntero a la matriz de int para ordenar.
29   *         Parametro DataSize  El tamaño de la matriz de datos.
30   * Return:
31           Nada.
32   **************************************/
33
34  void InsertionSort(int Data[], int DataSize){                    //= InsertionSort ===
35
36      for (int i = 1; i < DataSize; ++i) {                         //Recorre cada elemento
37
38          int j = i;                                               //A[0..i-1] ya esta
             ↪  ordenado
39          int Temp = Data[i];                                      //Buscamos en siguiente
             ↪  elemento
40
41          while (j > 0 && Temp < Data[j - 1]) {                    //Mover los elementos
             ↪  del subarreglo
42              Data[j] = Data[j - 1];                               //un elemento adelante
43              j--;                                                 //hasta que encontremos
                 ↪  el lugar correcto
44          }
45
```

```
46          Data[j] = Temp;                                    //Ponemos ahi Temp
47      }
48  }
```

4.0.3. SelectionSort

Código (`SelectionSort.c`):

```
/*****************************************
 *    Instituto Politecnico Nacional
 *    Escuela Superior de Computo
 *    Analisis de Algoritmos
 *    Fecha: 21/02/2018
 *    Autores:
 *            Oledo Enriquez Gilberto Irving
 *            Carrillo Balcazar Eduardo Yair
 *            Rojas Alvarado Luis Enrique
 *     Version: 1.0
 *     Equipo: Fuerza.exe
*****************************************/


/*=====================================================================
====================      FUNCIONES     ============================
=====================================================================*/

/****************************************
 * Nombre de la funcion: InsertionSort
 * Descripcion:
 *             Seleccionamos el valor más bajo para cada índice
 *            Así que, al final, encontramos los datos más pequeños,
 *            luego los ponemos al principio, al lado solo tenemos
 *            que ordenar la matriz a partir de la siguiente posición,
 *            por lo que hacemos lo mismo otra vez.
 * Parametros:
 *         Parametro Data         Un puntero a la matriz de int para ordenar.
 *         Parametro DataSize  El tamaño de la matriz de datos.
 * Return:
         Nada.
 ****************************************/


void SelectionSort(int Data[], int DataSize){                    //== SelectionSort ==

    for (int i = 0; i < DataSize - 1; ++i) {                     //Para cada indice
        int TiniestIndex = i;                                    //Guardar el indice
        ↪  actual
        for (int j = i + 1; j < DataSize; ++j)                   //Comprobar si es mas
        ↪  pequeño
            if (Data[j] < Data[TiniestIndex])                    //Si existe más pequeño
                TiniestIndex = j;                                //cambiamos el indice

        Swap(&Data[TiniestIndex], &Data[i]);                     //Intercambiar los datos
        ↪  más pequeños
    }
}
```

4.0.4. ShellSort

Código (`ShellSort.c`):

```
/*****************************************
 *    Instituto Politecnico Nacional
 *    Escuela Superior de Computo
 *    Analisis de Algoritmos
 *    Fecha: 21/02/2018
 *    Autores:
 *            Oledo Enriquez Gilberto Irving
 *            Carrillo Balcazar Eduardo Yair
 *            Rojas Alvarado Luis Enrique
 *     Version: 1.0
 *     Equipo: Fuerza.exe
*****************************************/


/*=====================================================================
====================      FUNCIONES     ============================
=====================================================================*/

/***************************************
 * Nombre de la funcion: ShellSort
 * Descripcion:
 *             ShellSort es principalmente una variación de Insertion Sort.
 *            En la ordenación por inserción, movemos los elementos solo una
 *            posición adelante. Cuando un elemento tiene que moverse mucho más
 *            adelante, hay muchos movimientos involucrados. La idea de shellSort
 *            es permitir el intercambio de elementos lejanos.
 *            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.
 *            Esto a traves de k=n/2, siendo n el número de elementos del arreglo.
 *            Después iremos variando k haciéndolo más pequeño
 *            mediante sucesivas divisiones por 2, hasta llegar a k=1.
 * Parametros:
 *         Parametro Data         Un puntero a la matriz de int para ordenar.
 *         Parametro DataSize  El tamaño de la matriz de datos.
 * Return:
         Nada.
 ***************************************/

void ShellSort(int Data[], int DataSize) {                          //=== ShellSort ======
    int Gap = DataSize >> 1;                                        //Consegimos la division
     ↪  entera n/2

    while (Gap > 0) {                                               //Hasta que la separacon
     ↪  sea 1

        for(int i = Gap; i < DataSize; i++) {                       //Para cada subarray

            int j = i;                                              //Sea j = i para
             ↪  modificar j
```

```
49              int Temporal = Data[i];                                 //Temp. sera el
                ↪ siguiente
50
51              while (j >= Gap && Temporal < Data[j - Gap]) {          //Cambiar antes
                ↪ gap-sort
52                  Data[j] = Data[j - Gap];                            //hasta el lugar
                    ↪ correcto
53                  j -= Gap;                                           //se encuentra para
                    ↪ Data[i]
54              }
55
56              Data[j] = Temporal;                                     //Temp. Encuentra su
                ↪ lugar
57          }
58
59          Gap >>= 1;                                                  //Reducir gap a la
            ↪ mitad
60      }
61  }
```

4.0.5. SortWithBST

Código (`SortWithBST.c`):

```
/*****************************************
 *    Instituto Politecnico Nacional
 *    Escuela Superior de Computo
 *    Analisis de Algoritmos
 *    Fecha: 21/02/2018
 *    Autores:
 *            Oledo Enriquez Gilberto Irving
 *            Carrillo Balcazar Eduardo Yair
 *            Rojas Alvarado Luis Enrique
 *     Version: 1.0
 *     Equipo: Fuerza.exe
*****************************************/


#include "TreeAuxFunction.c"

/*=========================================================================
====================      FUNCIONES     ============================
=========================================================================*/

/****************************************
 * Nombre de la funcion: ShellSort
 * Descripcion:
 *             Los datos se colocan en un arbol binario de busqueda.
 * Parametros:
 *         Parametro Data         Un puntero a la matriz de int para ordenar.
 *         Parametro DataSize  El tamaño de la matriz de datos.
 * Return:
         Nada.
 ****************************************/

void SortWithBST(int Data[], int DataSize) {                    //=== SortWithBST
 ↪  ======
    Node* Tree = NULL;                                          //Empezar con un árbol
     ↪  vacío

    for(int i = 0; i < DataSize; i++)                           //Para cada elemento
        IterativeInsertBST(&Tree, Data[i]);                     //insertarlo en el
         ↪  arbol

    IterativeCreateInOrder(&Tree, Data, DataSize);              //Recorrido inOrden
}
```

5. Tabla de Datos Experimentales

Especificaciones del equipo de computo donde se realizaron las pruebas:

- **Marca y modelo:** Acer Aspire 5
- **Procesador:** Intel core i7 8550U 1.8GHz
- **RAM:** 12 GB RAM
- **Disco Duro:** 1000 GB
- **Sistema Operativo:** Ubuntu

5.1. BubbleSort

5.1.1. SimpleBubbleSort - BubbleSortv1

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	0.0002360344
1000	0.0149390697
5000	0.1021978855
10000	0.4030380249
50000	10.9339311123
100000	43.779255867
200000	174.3581941128
400000	697.1250939369
600000	1575.6919999123
800000	2811.151638031
1000000	4384.4562010765

5.1.2. SimpleBubbleSort - BubbleSortv2

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	3.00407e-05
1000	0.0023970604
5000	0.1145870686
10000	0.3060879707
50000	8.041435957
100000	32.8288900852
200000	131.2398660183
400000	525.5603160858
600000	1190.169369936
800000	2119.7290890217
1000000	3313.0644769669

5.1.3. OptimizedBubbleSort - BubbleSortv3

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	3.19481e-05
1000	0.004240036
5000	0.0674700737
10000	0.314250946
50000	8.2729401588
100000	33.6116600037
200000	134.5853319168
400000	536.9529631138
600000	1203.8222651482
800000	2148.5643658638
1000000	3371.828168869

5.2. InsertionSort

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	9.0599e-06
1000	0.0007219315
5000	0.0313210487
10000	0.0653419495
50000	1.7327408791
100000	6.6620988846
200000	26.729391098
400000	106.2742159367
600000	239.5793950558
800000	428.7088711262
1000000	668.0265488625

5.3. SelectionSort

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	1.81198e-05
1000	0.0012779236
5000	0.0402719975
10000	0.1424100399
50000	3.0003089905
100000	11.9663519859
200000	47.6513719559
400000	189.7624928951
600000	427.0184149742
800000	761.3676571846
1000000	1186.4890911579

5.4. ShellSort

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	9.0599e-06
1000	0.0001261234
5000	0.0008060932
10000	0.0018730164
50000	0.0114870071
100000	0.0303308964
200000	0.0571689606
400000	0.1289229393
600000	0.1959331036
800000	0.2847340107
1000000	0.3559679985
2000000	0.8037760258
3000000	1.3025050163
4000000	1.824203968
5000000	2.4454939365
6000000	2.9070160389
7000000	3.4747941494
8000000	4.155189991
9000000	4.6561021805
10000000	5.5090708733

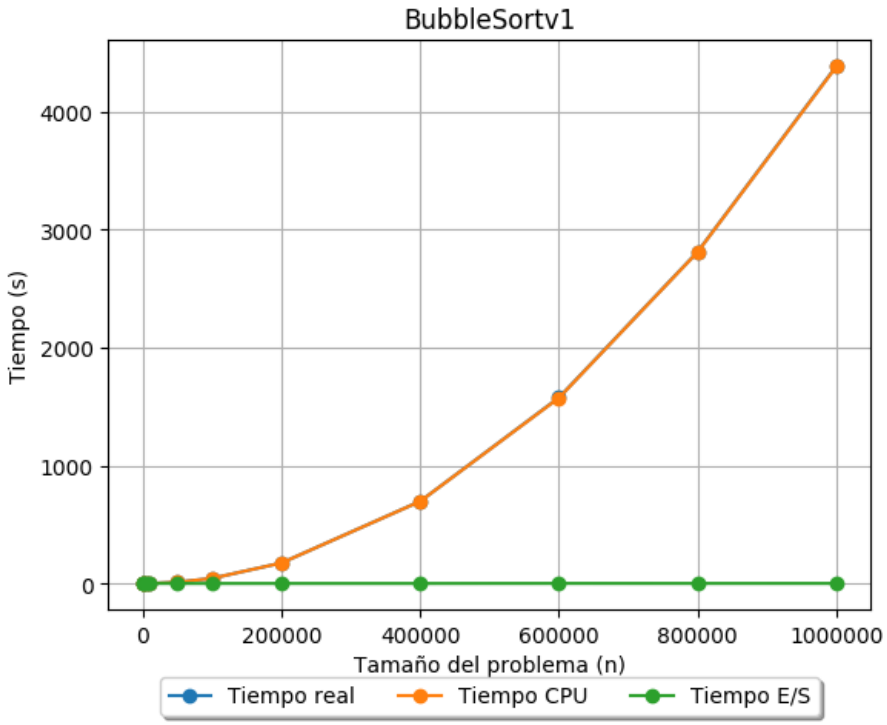
5.5. SortWithBST

Tamaño del arreglo a ordenar (n)	Tiempo (s)
100	2.31266e-05
1000	0.0001819134
5000	0.0009860992
10000	0.0020160675
50000	0.0211930275
100000	0.031537056
200000	0.0760622025
400000	0.2109770775
600000	0.3711650372
800000	0.5501980782
1000000	0.7447781563
2000000	2.0143699646
3000000	3.2221980095
4000000	4.6125769615
5000000	5.9545509815
6000000	7.6806359291
7000000	8.9136009216
8000000	10.4616150856
9000000	12.1246578693
10000000	14.0399329662

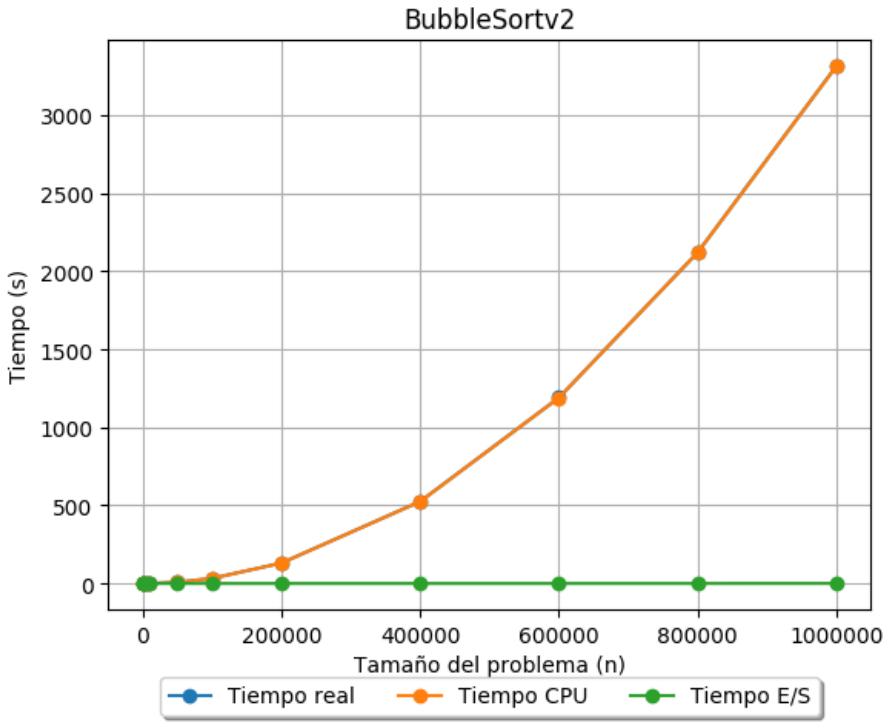
6. Actividades y Pruebas

6.1. Comportamiento temporal de cada algoritmo (Real, CPU y E/S)

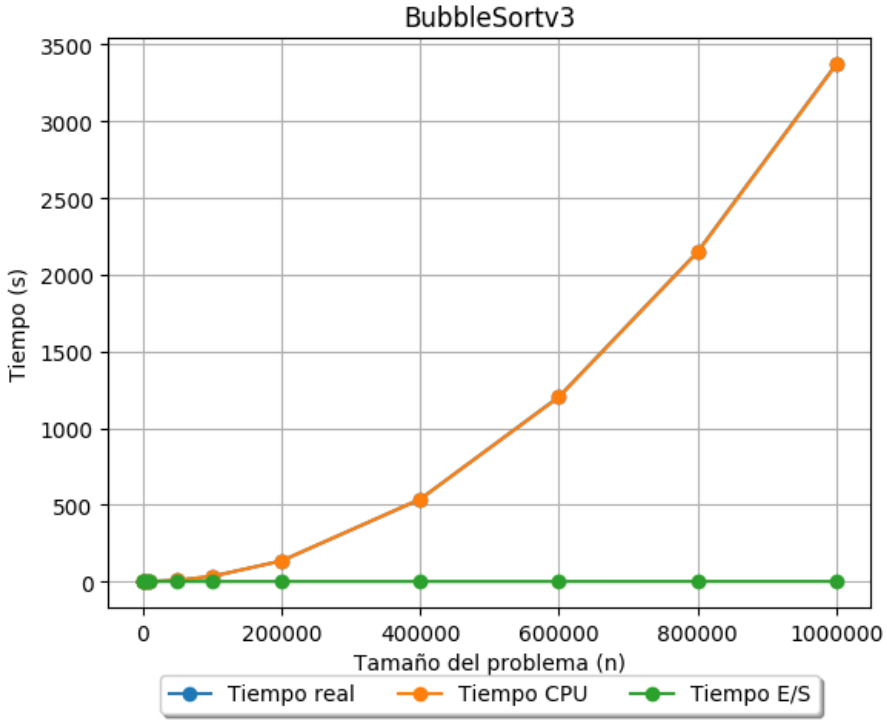
6.1.1. SimpleBubbleSort - BubbleSortv1



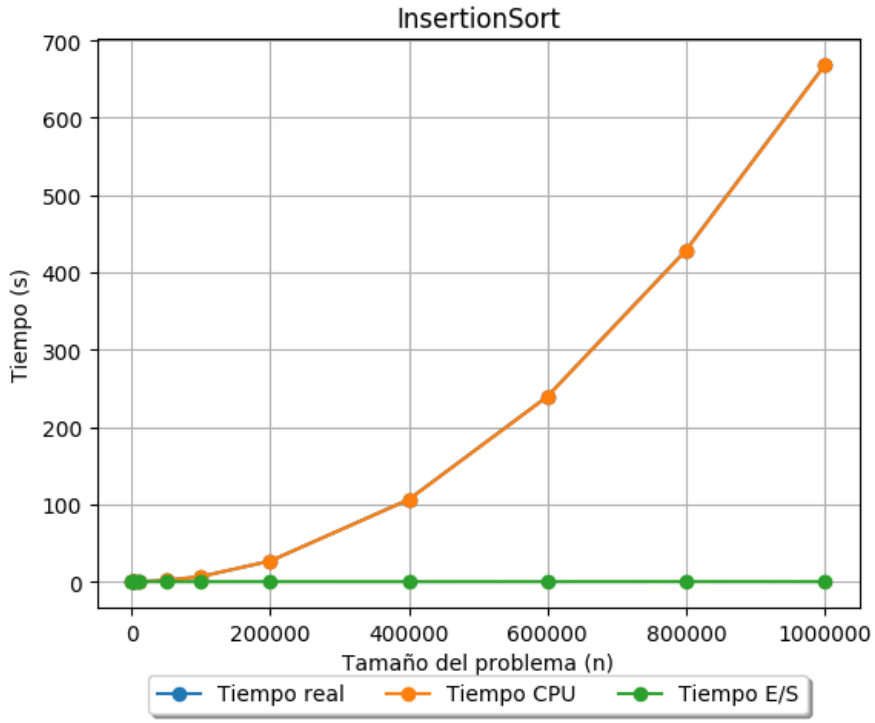
6.1.2. SimpleBubbleSort - BubbleSortv2



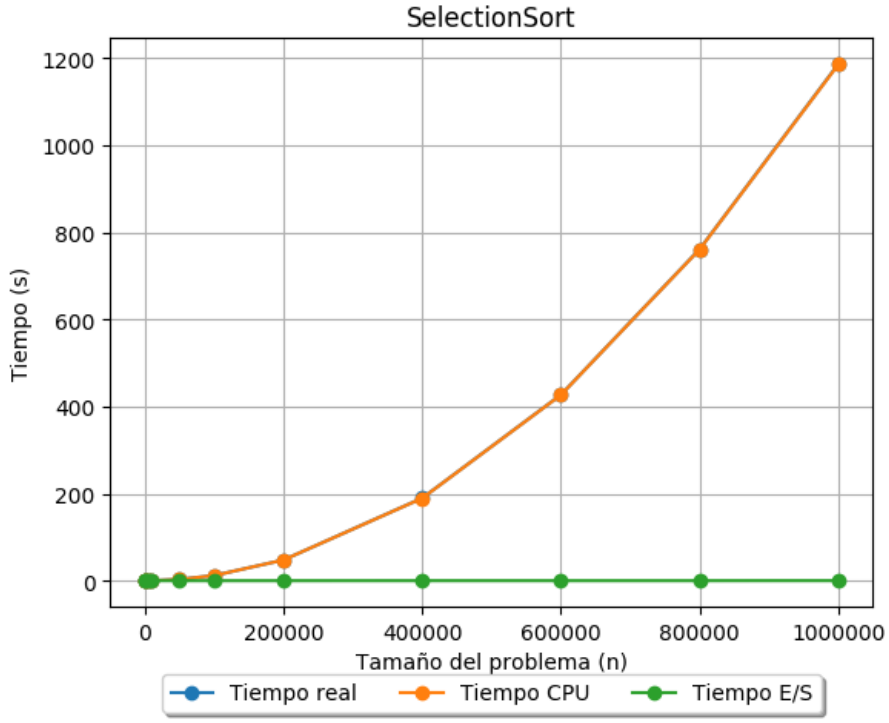
6.1.3. OptimizedBubbleSort - BubbleSortv3



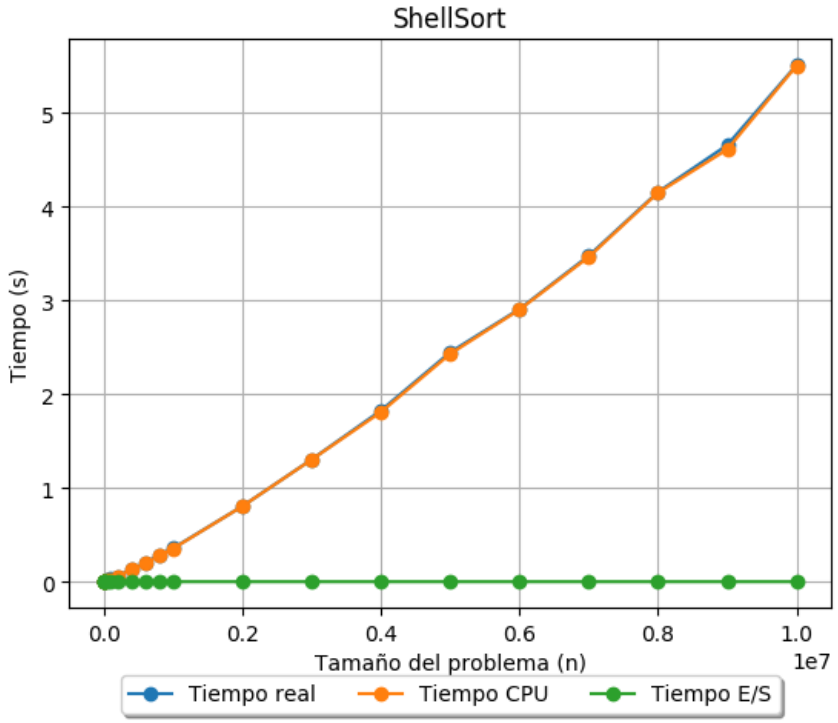
6.1.4. InsertionSort



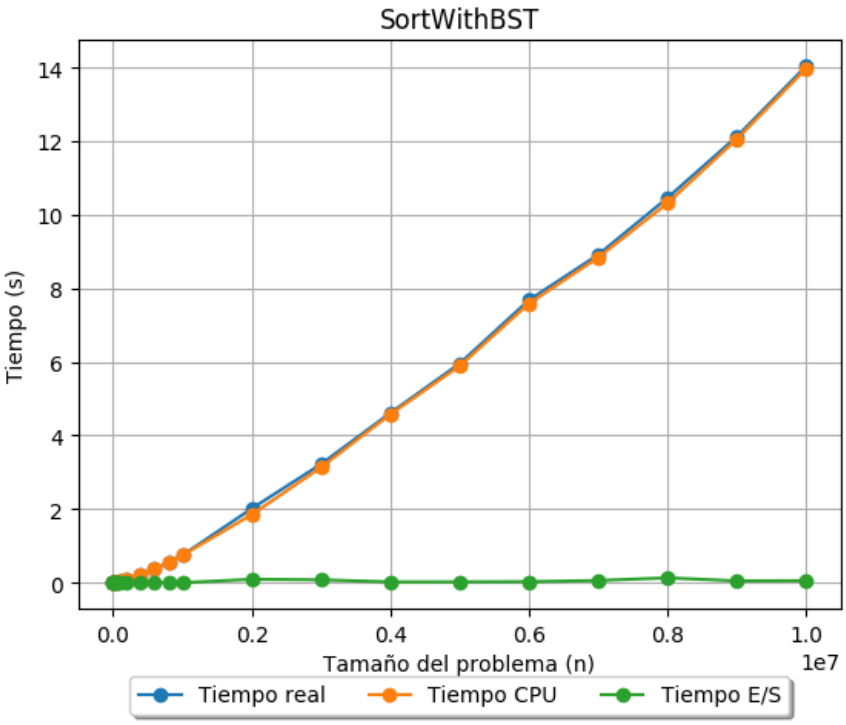
6.1.5. SelectionSort



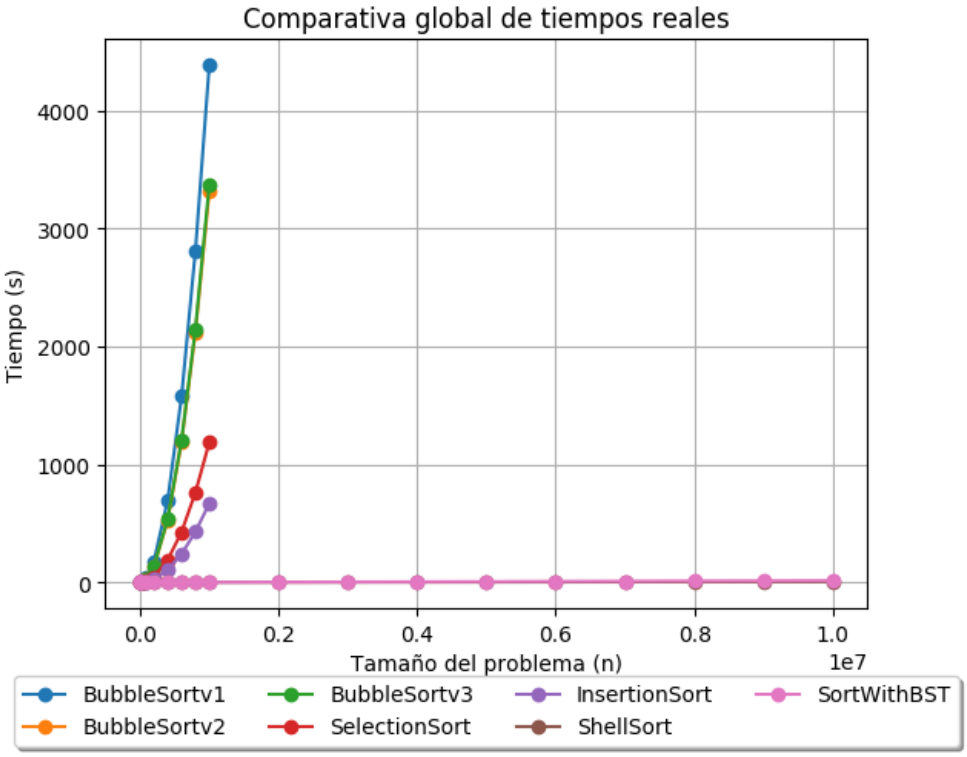
6.1.6. ShellSort

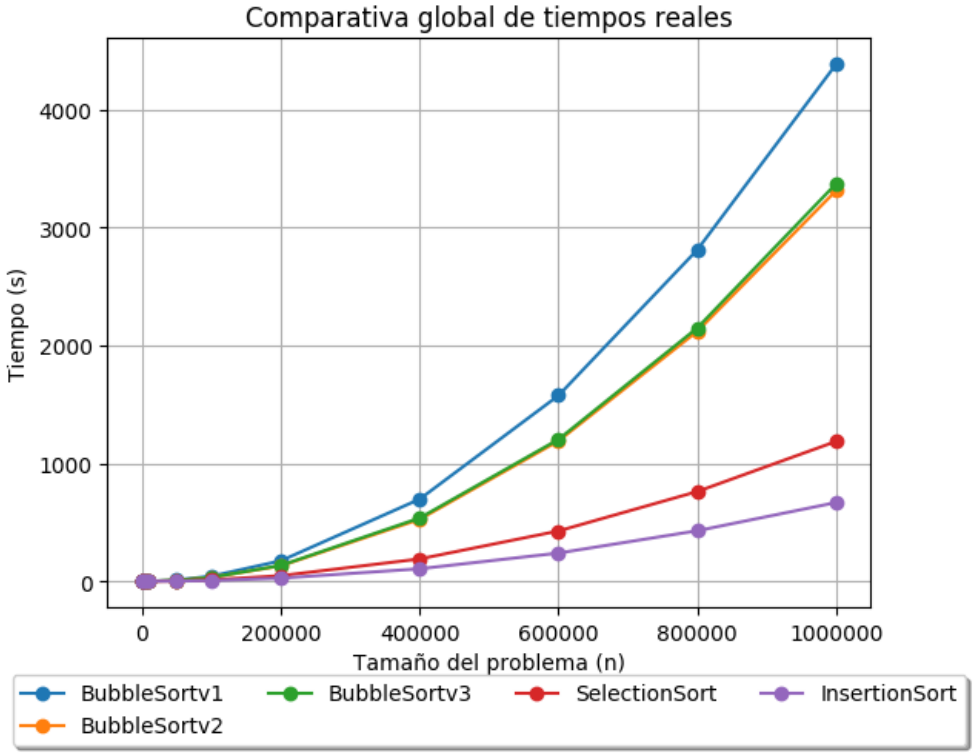


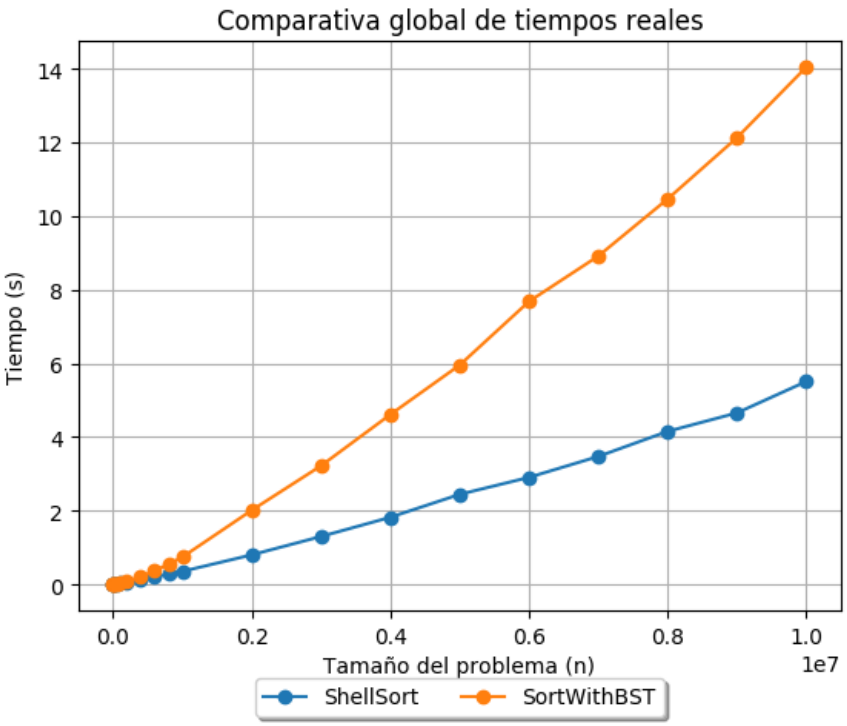
6.1.7. SortWithBST



6.2. Comparativa global de tiempos reales

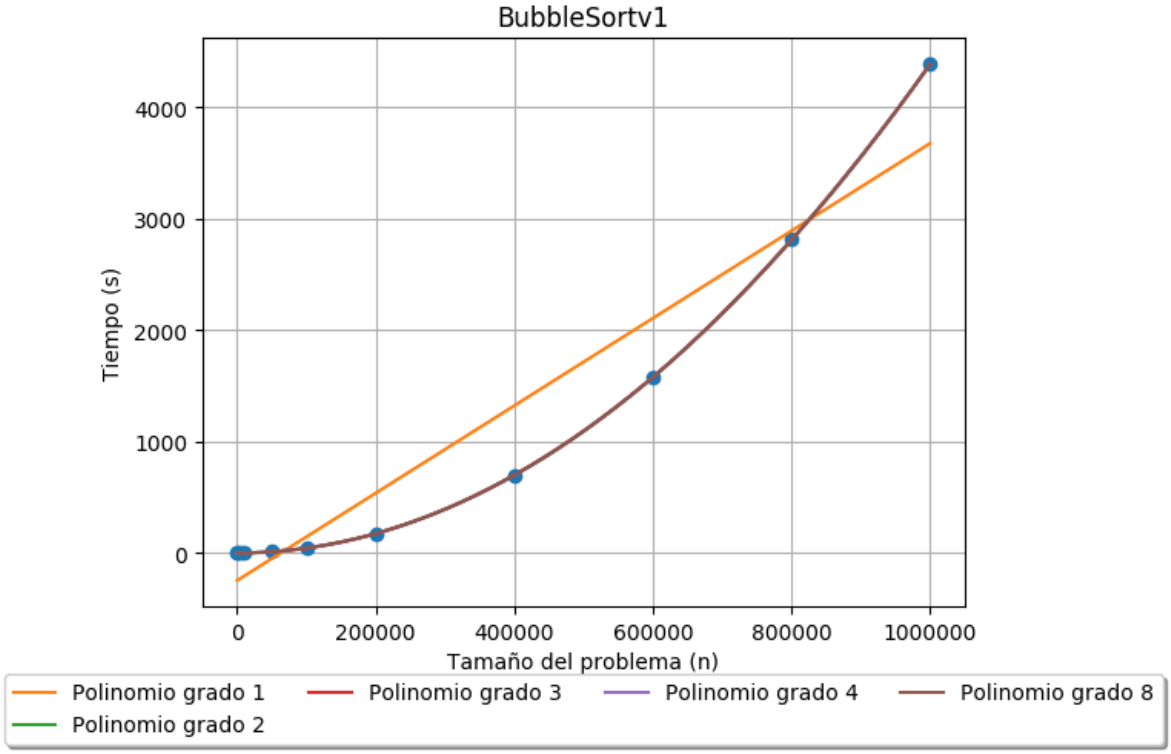





Comparativa global de tiempos reales
Tiempo (s)
0
2
4
6
8
10
12
14
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
ShellSort
SortWithBST

6.3. Aproximaciones polinomiales de cada algoritmo

6.3.1. SimpleBubbleSort - BubbleSortv1



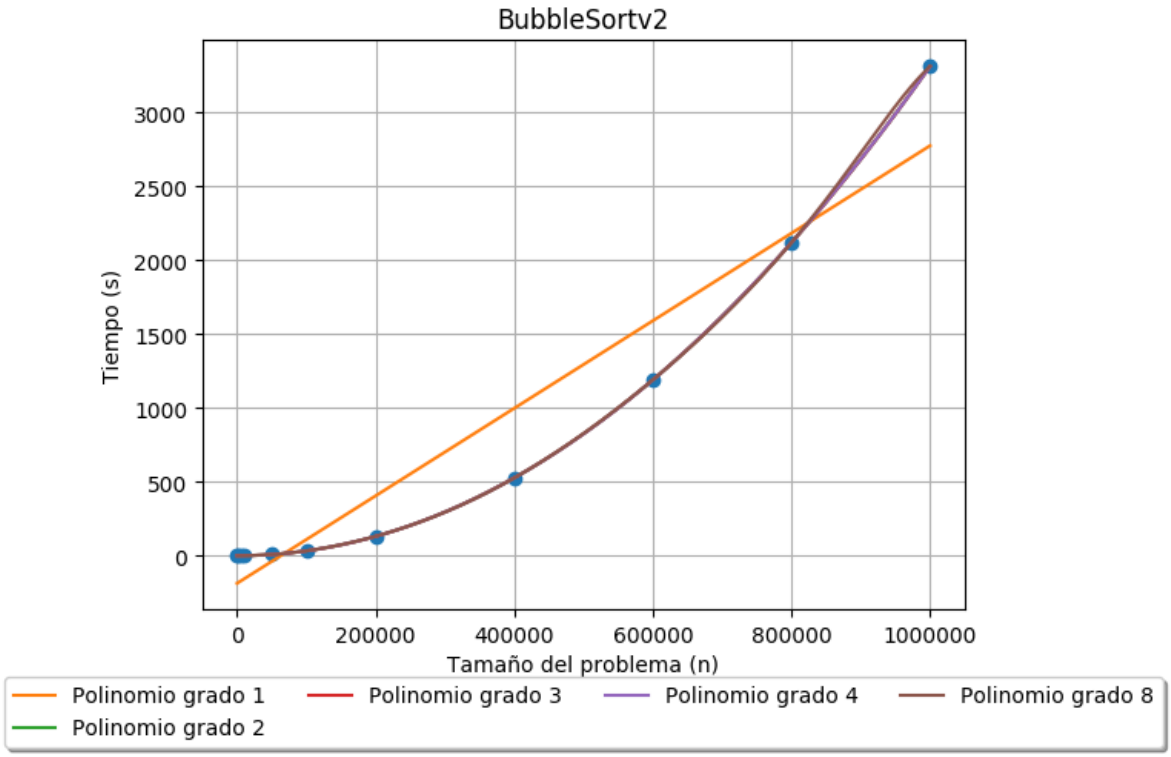
Aproximación Polinomial:

$$4.396143946219663e-09x^2 + -9.388438144401842e-06x^1 + 0.0653654511482518x^0$$

6.3.1.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	10989890.509007387s
100000000	43960500.683747634s
500000000	1099031292.401209s
1000000000	4396134557.846885s
5000000000	109903551713.36623s

6.3.2. SimpleBubbleSort - BubbleSortv2



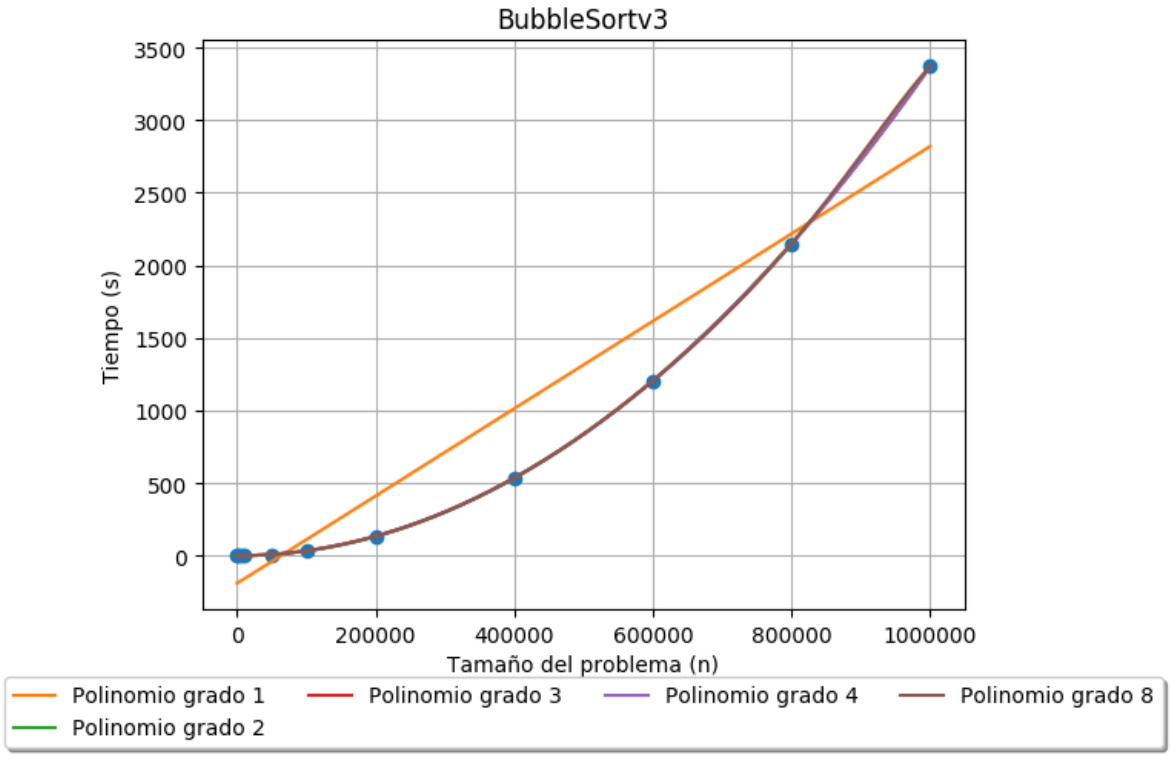
Aproximación Polinomial:

$$3.3264525132374845e-09x^2 + -1.3052159430450629e-05x^1 + 0.19869421499358572x^0$$

6.3.2.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	8315478.873816404s
100000000	33263220.115126017s
500000000	831606602.4283501s
1000000000	3326439461.2767477s
5000000000	83161247570.33865

6.3.3. OptimizedBubbleSort - BubbleSortv3



Aproximación Polinomial:

$$3.3940398651755237e-09x^2 + -2.6128637975465203e-05x^1 + 1.000789464342952x^0$$

6.3.3.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	8483794.2318295s
100000000	33937786.788747154s
500000000	848496902.9756827s
1000000000	3394013737.538338s
5000000000	84850865987.199s

6.3.4. InsertionSort

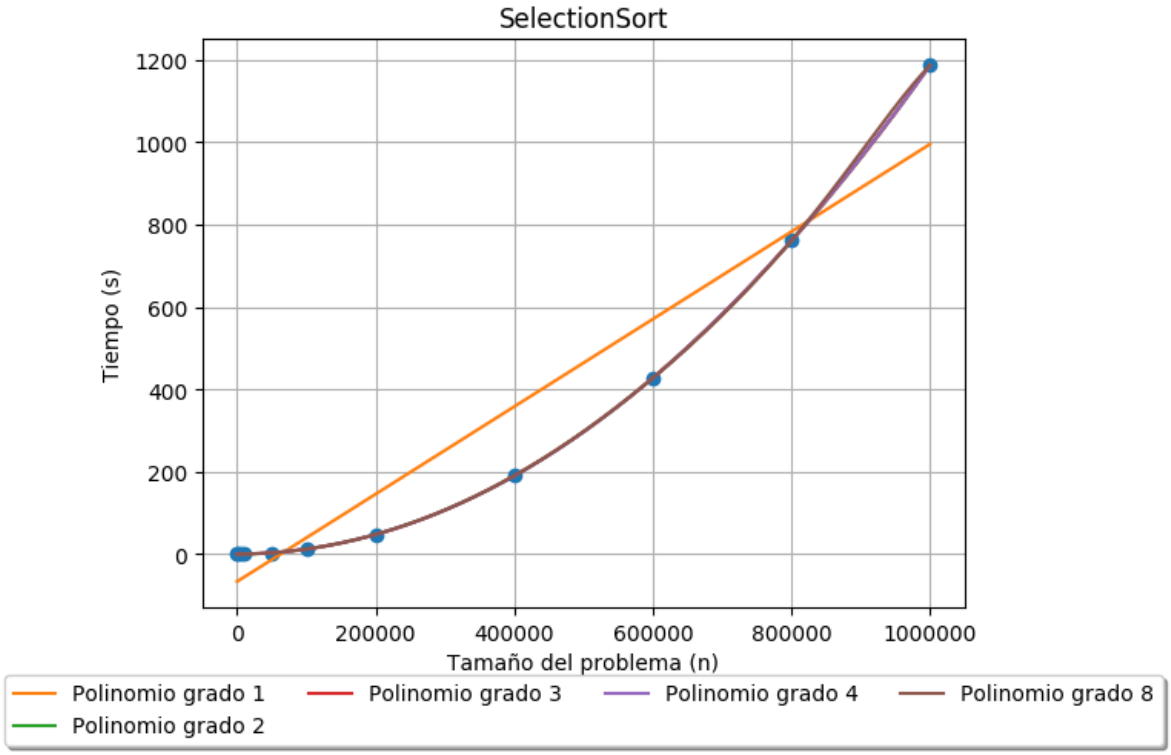
Aproximación Polinomial:

$$6.702139500160778e-10x^2 + -1.7789811241034799e-06x^1 + 0.0463881726464695x^0$$

6.3.4.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	1675445.9723721617s
100000000	6701961.64843654s
500000000	167552598.0598456s
1000000000	670212171.081342s
5000000000	16755339855.542713s

6.3.5. SelectionSort



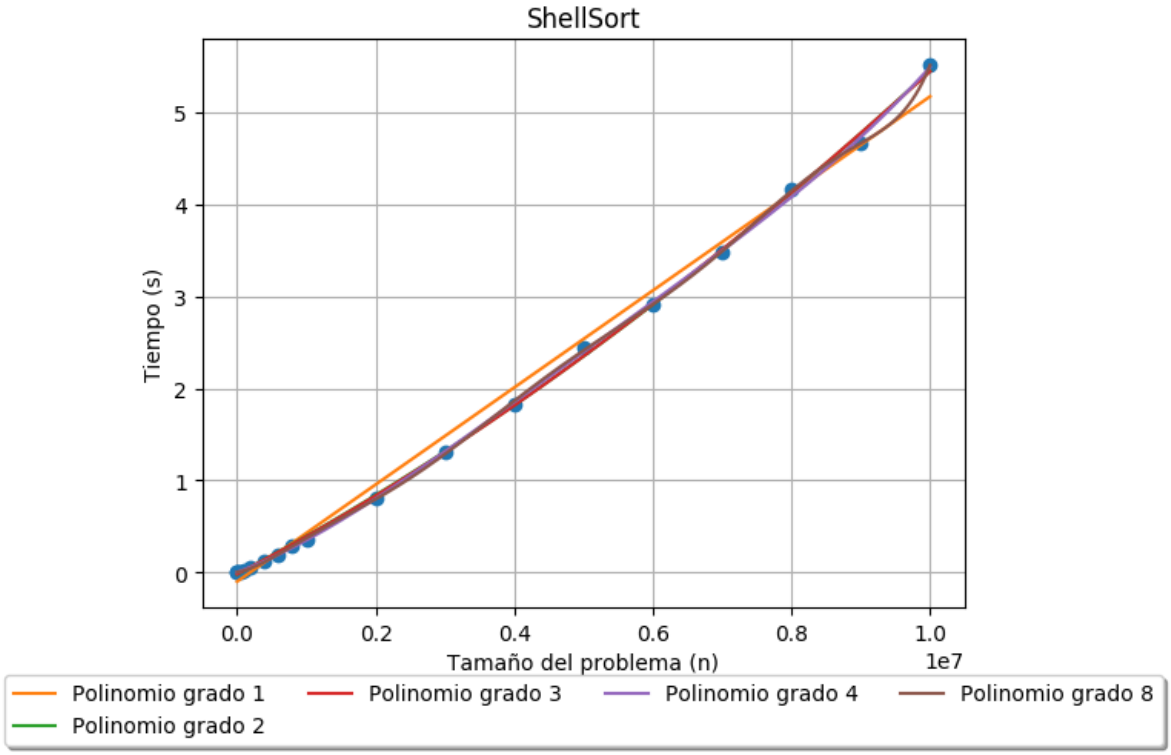
Aproximación Polinomial:

$$1.1861111531727285e-09x^2 + 1.0550677080305921e-06x^1 + -0.029197355142516 92x^0$$

6.3.5.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	2965330.6071198676s
100000000	11861217.009300733s
500000000	296528315.79783875s
1000000000	1186112208.211239s
5000000000	29652784104.627556s

6.3.6. ShellSort



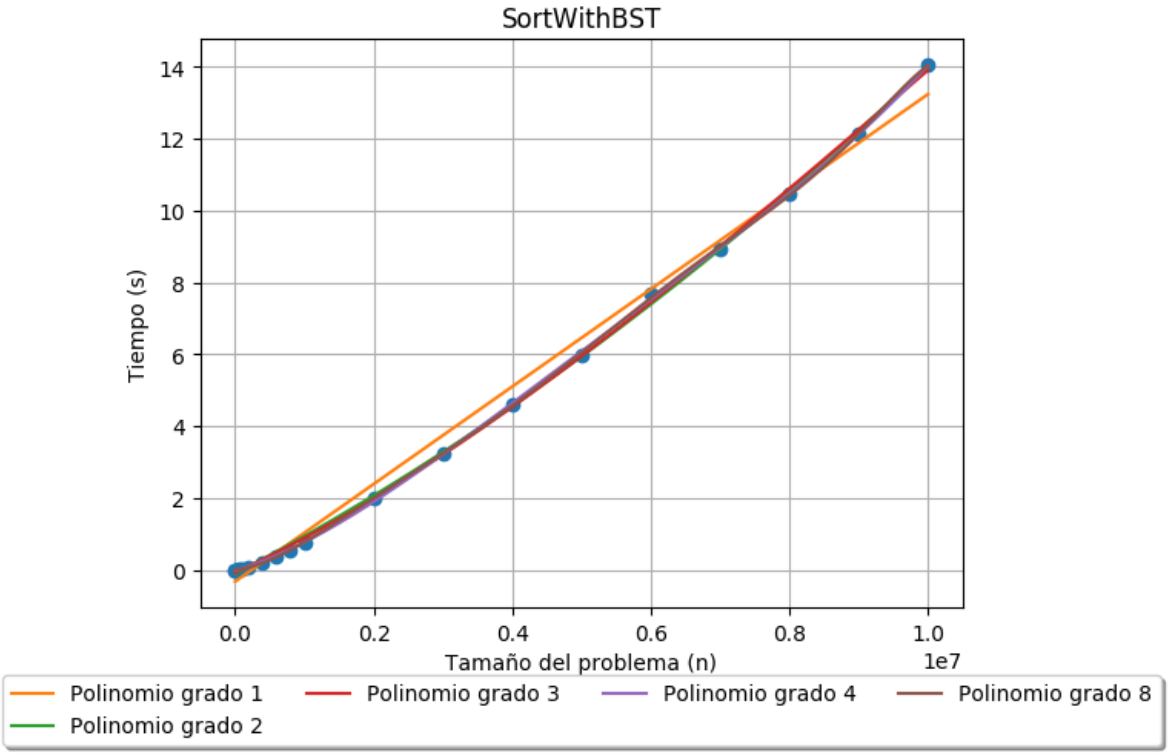
Aproximación Polinomial:

$$5.267236243879228e-07x^{1}+-0.09826119976612042x^{0}$$

6.3.6.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	26.237920019630018s
100000000	52.574101239026156s
500000000	263.26355099419527s
1000000000	526.6253631881566s
5000000000	2633.519860739848s

6.3.7. SortWithBST



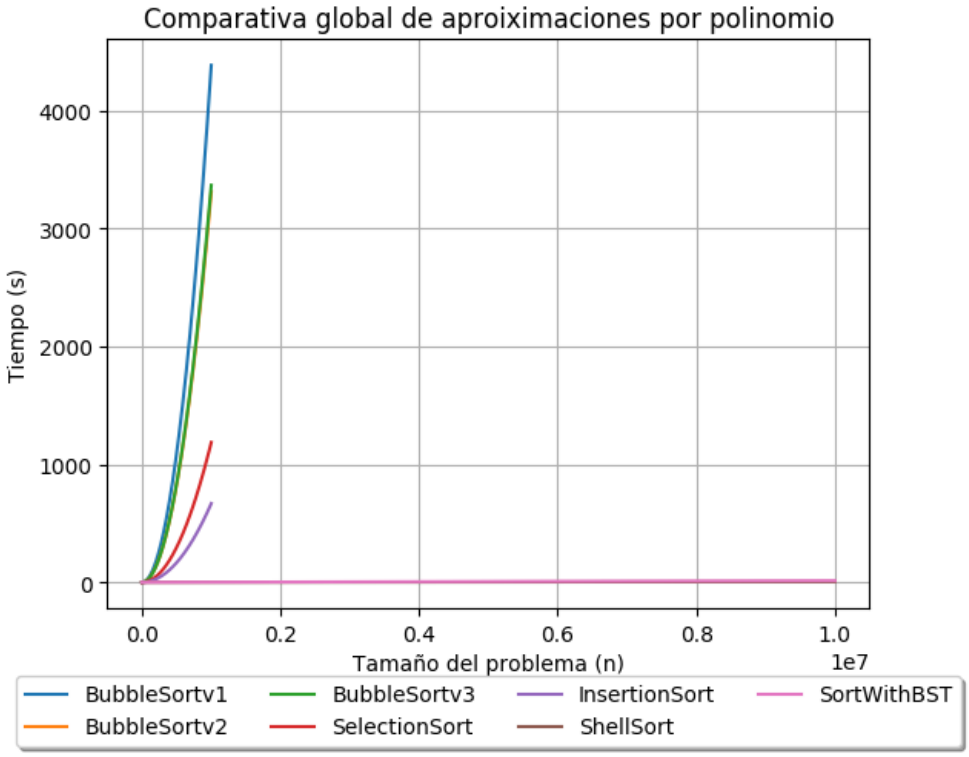
Aproximación Polinomial:

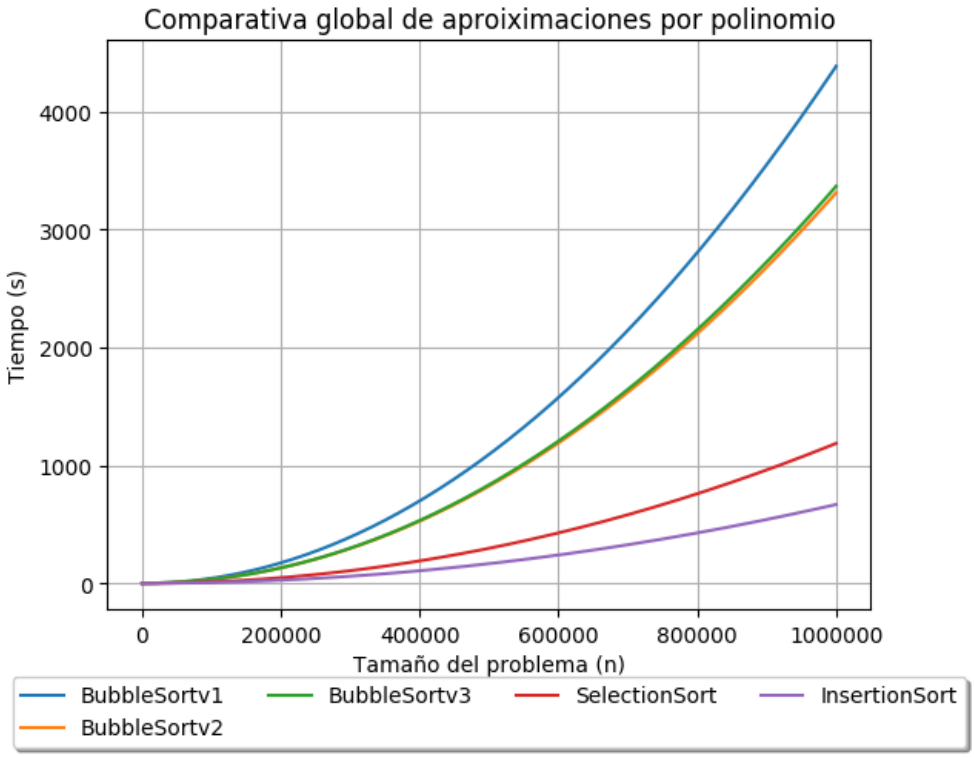
$$1.355784237283041e-06x^{1}+-0.3235820378072996x^{0}$$

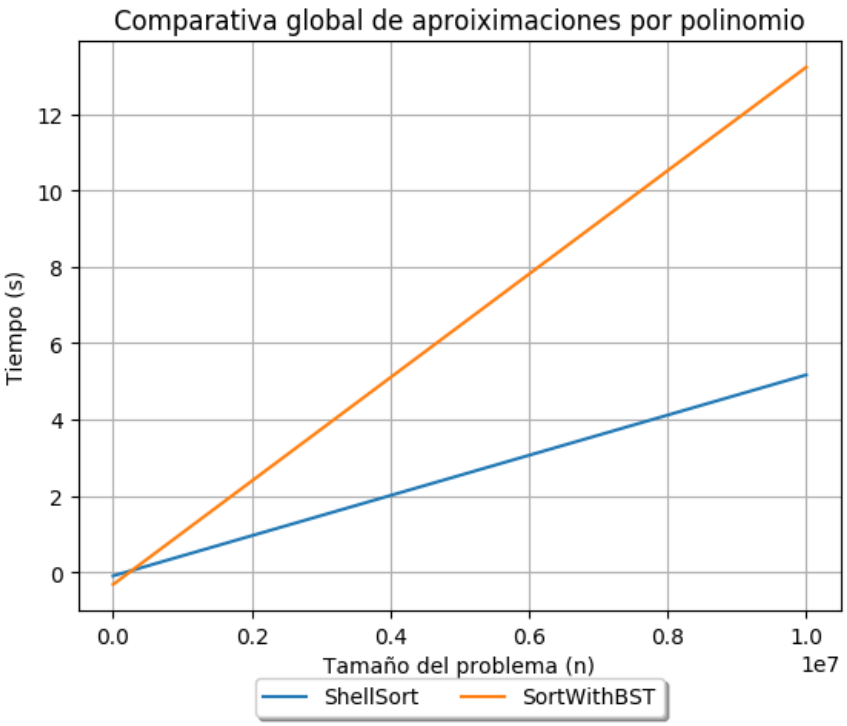
6.3.7.1. Tiempos reales basados en la aproximación polinomial

Tamaño del arreglo a ordenar (n)	Tiempo (s)
50000000	67.46562982634475s
100000000	135.2548416904968s
500000000	677.5685366037131s
1000000000	1355.4606552452337s
5000000000	6778.597604377397s

6.4. Comparativa global por polinomio






Comparativa global de aproximaciones por polinomio
Tiempo (s)
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
ShellSort
SortWithBST

6.5. Preguntas

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

   Burbuja Simple, una vez que se comprende como funciona el algoritmos, es mínima o nula la abstracción necesaria para desarrollar el código.
   En resumen, es una estructura simple, un código corto y es fácil de recordar.

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

   El Árbol Binario, ya que se necesitan implementar demasiadas funciones para lograr que funcione bien, sin mencionar que se tiene que tener un buen concepto de recursividad si es que se requiere la implementación de esta forma, sin embargo, si se quiere una implementación iterativa como en esta practica aumenta considerablemente la dificultad.
   En resumen, a diferencia de la Burbuja simple, al implementar el Árbol Binario se se quiere un buen nivel de abstracción para desarrollarlo de manera correcta.

3. **¿Cuál algoritmo tiene menor complejidad temporal?**

   Shell.
   De acuerdo a las pruebas realizadas Shell para un `n = 10,000,000` tardo entre 5 y 9 seg.

4. **¿Cuál algoritmo tiene mayor complejidad temporal?**

   Burbuja en cualquiera de sus versiones.
   De acuerdo a la estimación creada en clase, para un `n = 10,000,000` tardaría aproximadamente una semana.
   El ordenamiento de burbuja tiene una complejidad $\Omega(n^2)$ igual que ordenamiento por selección.

   Cuando una lista ya está ordenada, a diferencia del ordenamiento por inserción que pasará por la lista una vez y encontrará que no hay necesidad de intercambiar las posiciones de los elementos, el método de ordenación por burbuja está forzado a pasar por dichas comparaciones, lo que hace que su complejidad sea cuadrática en el mejor de los casos. Esto lo cataloga como el algoritmo más ineficiente que existe, aunque para muchos programadores sea el más sencillo de implementar.

5. **¿Cuál algoritmo tiene menor complejidad espacial?**
   ¿Por qué?

   Árbol Binario, si tomamos en cuenta únicamente sus operaciones básicas, son 2 y las variables que ocupa son `i, n` y un arreglo de `A [n+1]`.

6. **¿Cuál algoritmo tiene mayor complejidad espacial?**
   ¿Por qué?

De acuerdo al análisis que hicimos Burbuja Optimizada, Selección y Shell, son los más complejos espacialmente, utilizan un arreglo de `A[n]` donde eso equivale a n y sus otras variables, por ejemplo selección: `k, n , i, y temp` dando una complejidad espacial de `n+5`.

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

   En general si, por ejemplo, los algoritmos que tenían más operaciones básicas eran los más complejos por ejemplo, inserción este algoritmo para cantidades grandes en un peor caso tendría que estar siempre regresándose y recorriendo todo el arreglo, desde el análisis del algoritmo ya mostraba complejidad y fue demostrada cuando se hizo la prueba para cantidades grandes, pero, durante las pruebas nos dimos cuenta que Shell supero al Árbol Binario hablando de la rapidez para un `n = 10,000,000` ya que el Árbol Binario tardo aproximadamente de 12 a 14 segundos mientras que Shell lo hizo tardándose entre 5 y 9 segundos.

8. **¿Sus resultados experimentales difieren mucho de los del resto de los equipos? ¿A que se debe?**

   A grandes rasgos no. Todos o la mayoría de equipos seguimos los mismos pseudo códigos proporcionados por el profesor, por lo tanto, los resultados obtenidos tomando en cuenta únicamente el código, podría decirse que fue el mismo.
   Realizando un analizáis mas detallado, es decir, tomando en cuenta la computadora utilizada para realizar las pruebas es mas difícil dar una respuesta acertada, durante la revisión casi todos los equipos utilizamos las computadoras del laboratorio y no existió mucha diferencia.

   En conclusión, podemos decir que no existieron muchas diferencias. Las que existieron fueron causadas por alguna optimización en el código o por algún cambio de computadora al realizar las pruebas, es decir, utilizar un procesador mas "mas potente".

9. **¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?**

   Podría decirse que si, una vez que quedaron desarrollados los algoritmos procedimos a probarlos inmediatamente en una sola computadora para poder tener una referencia comparativa entre los algoritmos. Durante el tiempo de ejecución se trato de frenar todos los procesos ajenos a nuestra prueba, siempre y cuando fueron procesos que permitían ser cancelados sin afectar en nada, pero, al no ser TODOS los procesos no podemos decir que fue un entorno controlado al 100 % .
   En conclusión, se mantuvo un entorno controlado lo mas que se pudo. Las pruebas realizadas fueron bajo el sistema operativo Ubuntu el cual es una distribución de Linux basada en la arquitectura de Debian.
   Al inicio de la seccion **Tabla de datos experimentales** se muestran las especificaciones de la computadora utlizada.

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

a) Hagan sus scripts, pues aunque parezca más tedioso y crean que es más rápido anotar toda la información solicitada (que es mucha) a mano, es más eficiente por si se requiere cambiar algún algoritmo o los valores de entrada. Al menos que el script guarde los tiempos en algún archivo de texto con un formato que quieran.

b) Prueben sus algoritmos con arreglos pequeños antes de buscar en los 10 millones para ver si realmente los programaron bien.

c) Hagan sus programas generales, es decir, que reciban cualquier tamaño de subarreglo y el algoritmo a usar.

d) Usen Linux, pues en Windows no están la bibliotecas para medir los tres tiempos.

e) Cuando corran sus algoritmos, procuren cerrar todas las tareas en segundo plano para que los tiempos obtenidos sean lo más apegados a la realidad posible.

7. Errores Detectados.

Por un lado, no hemos detectado errores al momento de correr los algoritmos que no hallamos sabido cómo resolver, bueno, sí, solo uno.

El Tiempo.

A la hora de medir el tiempo de los algoritmos tenemos un grave problema, y es que a la hora de usar las funciones estándar proporcionadas por el profesor para medir el tiempo tenemos que sus medicaciones son incorrectas en el sentido de que sabemos que la ecuación para calcular el porcentaje de uso de la CPU está dada por:

$$CPUWall = (UserTime + SysTime)/RealTime$$

Pero al momento de intentar medir este porcentaje con búsquedas ridículamente rápida tenemos que nos dan valores sobre el 100 % lo que indica que las mediciones de tiempo son imprecisas si trabajamos con intervalos pequeños de tiempo.

Por otro lado no "errores" en sí, pero cosas que podrían ocasionar un error en los programas son:

- Que no se sigan las indicaciones de los parámetros de consola, es decir:
  - Darme más casos de los que tengo en el archivo.
  - Darme más tamaño para el arreglo que números.
  - Darme rutas incorrectas a la hora de abrir los ficheros.
- Que no se ingresen los parámetros de consola.
- Que no se tenga configurada la versión de python3.6 sino la 3.5 que es la estándar en este momento.
- Que no existan los directorios donde estarán las salidas y las gráficas.

8. Posibles Mejoras.

Las posibles mejoras que podemos hacer a estos algoritmos son sencillas:

- Podemos crear una interfaz de entrada de comandos mucho más sencilla, como una que no requiera pasar la cantidad de casos a buscar, sino un simple path al archivo, así como un path al archivo de datos del arreglo.
- Podemos asegurarnos de los errores que podría traer la entrada de datos, por ejemplo al asegurarnos de que estamos leyendo enteros o que el path existe.
- Podemos usar una documentación acorde al lenguaje, más exactamente usar algo como el estándar para documentar en C++11/14.
- Podríamos haber hecho la organización del script Make.py mucho más documentada y separar en funciones el código.

9. Conclusiones.

- **Gilberto Irving Oledo Enriquez.**
  Mis conclusiones respecto a a está practica son las siguientes, decidí dividirlas en varios puntos para hacer más entendible cuáles son.

  - Desarrollo del código.

    El desarrollo del código fue sencillo, pues los pseudo códigos (a excepción del árbol binario) fueron dados por el profesor, por lo tanto, únicamente requerían traducirlos a lenguaje c, sin embargo decidimos realizar algunas modificaciones a fin de obtener tiempos menores para n's muy grandes (por ejemplo, 10000000).

    Una de las modificaciones que en lo personal me gustó añadir, fue un corrimiento de bits en la implementación de shell, esto fue porque en pseudo código se tenían que realizar dos operaciones, una era n/2 y otra era TRUNC para garantizar que el arreglo sería dividido en dos y dejar como resultados un entero, sin embargo, un corrimiento de bits de la forma $>> 1$ nos garantiza eso desde el primer momento, además de que es bien sabido que para el procesador de una computadora es mucho menos costoso el operar a nivel bit que el realizar una división esto sin añadir el TRUNC ya mencionado.

    Otra optimización que agregue y que sin duda ayudó mucho además de que aprendí a implementar, fueron las funciones inline, ya que cuando se le da más valor valor al tiempo de ejecución y no a la memoria utilizada son una excelente opción.

    Regresando a la implementación de los algoritmos, el más difícil fue el árbol binario, aunque en estrictas de datos se había trabajado con el, está vez se requería una implantación sin recursividad y esto desde mi punto de vista aumentó la dificultad, pero, ayudándonos de una pila o stack pudimos cumplir el objetivo.

  - Pruebas.

    Para realizar las pruebas tratamos de optimizar todo, es decir crear un archivo en c que permitiera correr el algoritmo deseado, con la n que se requiriera y mostrando los números ordenados en un .txt por lo tanto no fue difícil testear el funcionamiento, y además, de acuerdo a la teoría vista en clase sabíamos cuáles algoritmos tardarían más y cuáles menos, esto los ayudó para saber que n asignar a cada uno y obtener los resultados en un tiempo razonable.

    Fue posible probar con una n = 10000000 con distintos algoritmos por ejemplo shell y árbol binario los cuales tardaron 5 y 12 segundos respectivamente. Para otro tipo algoritmos como burbuja (en sus 3 variantes) se utilizaron n's más pequeñas y de igual forma se obtuvieron buenos resultados respecto a la salida de los números ordenados, respecto al tiempo los resultados no fueron tan agradables, pero, era lo esperado, repito, por lo visto en clase.

    El obtener las gráficas de primera instancia fue difícil ya que pretendíamos utilizar phyton para automatizar todo, se tuvieron que investigar muchas cosas sobre la sintaxis del len-

guaje ya que ninguno de los 3 integrantes sabía utilizar el lenguaje a detalle, finalmente después de algunas maldiciones con el código se logró el objetivo y se obtuvieron tanto las gráficas mostradas anteriormente, como los polinomios de cada algoritmo.

- Uso de uno u otro algoritmo.

  Esta es la parte más interesante de las conclusiones, es increíble cómo funciona el análisis de un algoritmo, es decir, a simple vista al ver que todos cumples el objetivo -ordenar números- se pensaría que es indiferente cual se use en algún proyecto real (o incluso en alguna práctica) pero cuando nos adentramos a analizar y ver que no son lo mismo nos damos cuenta de la importancia y la complejidad de esta disciplina y entendemos cómo aunque las computadoras de hoy en día son muy rápidas, el tomar una mala decisión puede jugar en nuestra contra.

  Añadido a esto, entendimos que la complejidad de entender cómo funciona un algoritmo y la complejidad de su implementación no va de la mano con su eficiencia al realizar su función, el ejemplo más claro es el ordenamiento con burbuja, el pensar cómo trabaja es simple, el convertir el algoritmo a pseudo código y después traducirlo a algún lenguaje de programación de igual forma es fácil pero si funcionamiento es pésimo para tamaños de problema muy grandes, del lado contrario tenemos el ordenamiento con un árbol binario, el entender la idea de su funcionamiento requiere de poner más atención y su implementación no se diga, en ocasiones NO es trivial, y sin embargo, fue el segundo algoritmo más rápido.

En conclusión, la práctica me gusto, ayuda a entender cosas sencillas que funcionan de base para entender futuros temas y futuros análisis.

- **Carrillo Balcazar, Eduardo Yair.**
  Como pudimos observar en esta práctica cada uno de los algoritmos al ser de orden n^2 al tratarse de algoritmos de ordenamiento, existe gran diferencia en la eficiencia de cada uno, cada uno desarrollando un tiempo diferente de acuerdo a esta eficiencia. También dentro de los parámetros que influyen para el tiempo en que se tarda cada algoritmo, depende a la computadora donde se esta corriendo este algoritmo, de los programas que este ejecutando el computador, entre otras cosas más.

- **Rojas Alvarado, Luis Enrique.**
  En ésta práctica simplemente se requería ver los tiempos que tardaba en ordenar tantos números a un algoritmo. Simplemente con base en los pseudocódigos proporcionados por el profesor, se hacen sus funciones en c para que al programa principal por medio de parámetros se indique el número de algoritmos, el tamaña del arreglo y los números a ordenar que vendrán de un archivo de texto de 10 millones de números.

  La práctica fue relativamente sencilla puesto que la parte de tomar los tiempos antes de que inicie el algoritmo sólo es llamar a las funciones que ya nos había proporcionado el profesor con anterioridad y la parte difícil fue buscar una manera no recursiva de recorrer el árbol.

Anexos

A. Código de fuente original.

A.1. BubbleSort.c

```
/*****************************************
 *    Instituto Politecnico Nacional
 *    Escuela Superior de Computo
 *    Analisis de Algoritmos
 *    Fecha: 21/02/2018
 *    Autores:
 *            Oledo Enriquez Gilberto Irving
 *            Carrillo Balcazar Eduardo Yair
 *            Rojas Alvarado Luis Enrique
 *     Version: 1.0
 *     Equipo: Fuerza.exe
*****************************************/

/*=====================================================================
=====================      FUNCIONES     ============================
=====================================================================*/

/*****************************************
 * Nombre de la funcion: BubbleSortv1
 * Descripcion:
 *             Este es el algoritmo (bubble sort) más trivial, sin optimización nada, el algoritmo en bruto.
 *             Para cada elemento en el Array, compruebe si hay 2 elementos contiguos que están en el orden incorrecto,
 *             si es valido, intercambia.
 * Parametros:
 *         Parametro Data         Un puntero a la matriz de int para ordenar.
 *         Parametro DataSize  El tamaño de la matriz de datos.
 * Return:
         Nada.
 *****************************************/

void BubbleSortv1(int Data[], int DataSize) {                  //=== BubbleSortv1 ======
    for (int i = 0; i < DataSize - 1; ++i) {                   //Hacer esto (Size - 1) veces
        for (int j = 0; j < DataSize - 1; ++j) {               //Hacer esto (Size - 1) veces
            if (Data[j] > Data[j + 1])                         //Si necesitamos intercambiar
                Swap(&Data[j], &Data[j + 1]);                  //Intercambiamos!
        }
    }
}



/*****************************************
 * Nombre de la funcion: BubbleSortv2
```

```
44  * Descripcion:
45  *             Esta es la optimización más importante, reducir el segundo bucle porque los
    ↪  últimos
46  *             elementos i ya están en su lugar.
47  * Parametros:
48  *         Parametro Data         Un puntero a la matriz de int para ordenar.
49  *         Parametro DataSize  El tamaño de la matriz de datos.
50  * Return:
51          Nada.
52  ****************************************/
53
54  void BubbleSortv2(int Data[], int DataSize) {                    //=== BubbleSortv2 ======
55      for (int i = 0; i < DataSize - 1; i++) {                     //Hacer esto (Size - 1)
        ↪  veces
56          for (int j = 0; j < DataSize - i - 1; j++) {             //Los últimos (i) están
            ↪  ordenados
57              if (Data[j] > Data[j + 1])                           //Si necesitamos
                ↪  intercambiar
58                  Swap(&Data[j], &Data[j + 1]);                    //Intercambiamos!
59          }
60      }
61  }
62
63
64  /****************************************
65   * Nombre de la funcion: BubbleSortv3
66   * Descripcion:
67   *             Esta es la siguiente optimización más importante, romper la búsqueda si ya
    ↪  está ordenada.
68   * Parametros:
69   *         Parametro Data         Un puntero a la matriz de int para ordenar.
70   *         Parametro DataSize  El tamaño de la matriz de datos.
71   * Return:
72          Nada.
73   ****************************************/
74  void BubbleSortv3(int Data[], int DataSize) {                     //=== BubbleSortv3
    ↪  ======
75
76      for (int i = 0; i < DataSize - 1; i++) {                     //Hacer esto (Size - 1)
        ↪  veces
77          bool Swapped = false;                                    //Supongamos que no hay
            ↪  intercambio
78
79          for (int j = 0; j < DataSize - i - 1; j++) {             //Los últimos (i) están
            ↪  ordenados
80              if (Data[j] > Data[j + 1]) {                         //¿Necesitamos
                ↪  intercambiar?
81                  Swap(&Data[j], &Data[j + 1]);                    //Intercambiamos!
82                  Swapped = true;                                  //Intercambio es TRUE
83              }
84          }
85
86          if (!Swapped) break;                                     //No los cambies, estan
            ↪  ordenados!
87      }
```

```
88 }
```

A.2. InsertionSort.c

```
1  /*****************************************
2   *    Instituto Politecnico Nacional
3   *    Escuela Superior de Computo
4   *    Analisis de Algoritmos
5   *    Fecha: 21/02/2018
6   *    Autores:
7   *            Oledo Enriquez Gilberto Irving
8   *            Carrillo Balcazar Eduardo Yair
9   *            Rojas Alvarado Luis Enrique
10  *     Version: 1.0
11  *     Equipo: Fuerza.exe
12  *****************************************/
13
14
15  /*=====================================================================
16  ====================      FUNCIONES     ============================
17  =====================================================================*/
18
19  /****************************************
20   * Nombre de la funcion: InsertionSort
21   * Descripcion:
22   *             Este es un algoritmo de clasificación realmente intuitivo, para ello decimos
    ↪  que crearemos un nuevo
23   *             subarray. Entonces el subarray [0, 1] ya está ordenado, ahora:
24   *             Llevamos el siguiente elemento contiguos al subarreglo, lo ponemos donde
    ↪  pertenece y movemos todos los
25   *             otros 1 lugar.
26   *             Ahora nuestro nuevo subarreglo es de [0, 2], repetimos (n-1) y listo.
27   * Parametros:
28   *         Parametro Data         Un puntero a la matriz de int para ordenar.
29   *         Parametro DataSize  El tamaño de la matriz de datos.
30   * Return:
31           Nada.
32   ****************************************/
33
34  void InsertionSort(int Data[], int DataSize){                    //= InsertionSort ===
35
36      for (int i = 1; i < DataSize; ++i) {                         //Recorre cada elemento
37
38          int j = i;                                               //A[0..i-1] ya esta
            ↪  ordenado
39          int Temp = Data[i];                                      //Buscamos en siguiente
            ↪  elemento
40
41          while (j > 0 && Temp < Data[j - 1]) {                    //Mover los elementos
            ↪  del subarreglo
42              Data[j] = Data[j - 1];                               //un elemento adelante
43              j--;                                                 //hasta que encontremos
                ↪  el lugar correcto
44          }
```

```
45
46          Data[j] = Temp;                                         //Ponemos ahi Temp
47      }
48  }
```

A.3. SelectionSort.c

```
1   /*****************************************
2    *    Instituto Politecnico Nacional
3    *    Escuela Superior de Computo
4    *    Analisis de Algoritmos
5    *    Fecha: 21/02/2018
6    *    Autores:
7    *            Oledo Enriquez Gilberto Irving
8    *            Carrillo Balcazar Eduardo Yair
9    *            Rojas Alvarado Luis Enrique
10   *     Version: 1.0
11   *     Equipo: Fuerza.exe
12  *****************************************/
13
14
15  /*=========================================================================
16  ====================      FUNCIONES     ============================
17  ===========================================================================*/
18
19  /****************************************
20   * Nombre de la funcion: InsertionSort
21   * Descripcion:
22   *             Seleccionamos el valor más bajo para cada índice
23   *            Así que, al final, encontramos los datos más pequeños,
24   *            luego los ponemos al principio, al lado solo tenemos
25   *            que ordenar la matriz a partir de la siguiente posición,
26   *            por lo que hacemos lo mismo otra vez.
27   * Parametros:
28   *         Parametro Data         Un puntero a la matriz de int para ordenar.
29   *         Parametro DataSize  El tamaño de la matriz de datos.
30   * Return:
31           Nada.
32   ****************************************/
33
34
35  void SelectionSort(int Data[], int DataSize){                   //== SelectionSort ==
36
37      for (int i = 0; i < DataSize - 1; ++i) {                    //Para cada indice
38          int TiniestIndex = i;                                   //Guardar el indice
            ↪ actual
39          for (int j = i + 1; j < DataSize; ++j)                  //Comprobar si es mas
            ↪ pequeño
40              if (Data[j] < Data[TiniestIndex])                   //Si existe más pequeño
41                  TiniestIndex = j;                               //cambiamos el indice
42
43          Swap(&Data[TiniestIndex], &Data[i]);                    //Intercambiar los datos
            ↪ más pequeños
44      }
```

```
45 }
```

A.4. ShellSort.c

```
1  /*****************************************
2   *    Instituto Politecnico Nacional
3   *    Escuela Superior de Computo
4   *    Analisis de Algoritmos
5   *    Fecha: 21/02/2018
6   *    Autores:
7   *            Oledo Enriquez Gilberto Irving
8   *            Carrillo Balcazar Eduardo Yair
9   *            Rojas Alvarado Luis Enrique
10  *     Version: 1.0
11  *     Equipo: Fuerza.exe
12  *****************************************/
13
14
15  /*=====================================================================
16  ====================      FUNCIONES     ============================
17  =====================================================================*/
18
19  /*****************************************
20   * Nombre de la funcion: ShellSort
21   * Descripcion:
22   *             ShellSort es principalmente una variación de Insertion Sort.
23   *            En la ordenación por inserción, movemos los elementos solo una
24   *            posición adelante. Cuando un elemento tiene que moverse mucho más
25   *            adelante, hay muchos movimientos involucrados. La idea de shellSort
26   *            es permitir el intercambio de elementos lejanos.
27   *            Shell propone que se haga sobre el arreglo una serie de
28   *            ordenaciones basadas en la inserción directa, pero dividiendo
29   *            el arreglo original en varios sub-arreglos, tales que cada
30   *            elemento esté separado k elementos del anterior.
31   *            Esto a traves de k=n/2, siendo n el número de elementos del arreglo.
32   *            Después iremos variando k haciéndolo más pequeño
33   *            mediante sucesivas divisiones por 2, hasta llegar a k=1.
34   * Parametros:
35   *         Parametro Data         Un puntero a la matriz de int para ordenar.
36   *         Parametro DataSize  El tamaño de la matriz de datos.
37   * Return:
38           Nada.
39   *****************************************/
40
41  void ShellSort(int Data[], int DataSize) {                          //=== ShellSort ======
42      int Gap = DataSize >> 1;                                        //Consegimos la division
     ↪  entera n/2
43
44      while (Gap > 0) {                                               //Hasta que la separacon
     ↪  sea 1
45
46          for(int i = Gap; i < DataSize; i++) {                       //Para cada subarray
47
```

```
48              int j = i;                                          //Sea j = i para
                ↪  modificar j
49              int Temporal = Data[i];                             //Temp. sera el
                ↪  siguiente
50
51              while (j >= Gap && Temporal < Data[j - Gap]) {      //Cambiar antes
                ↪  gap-sort
52                  Data[j] = Data[j - Gap];                        //hasta el lugar
                    ↪  correcto
53                  j -= Gap;                                       //se encuentra para
                    ↪  Data[i]
54              }
55
56              Data[j] = Temporal;                                 //Temp. Encuentra su
                ↪  lugar
57          }
58
59          Gap >>= 1;                                              //Reducir gap a la
            ↪  mitad
60      }
61  }
```

A.5. SortWithBST.c

```
1   /*****************************************
2    *    Instituto Politecnico Nacional
3    *    Escuela Superior de Computo
4    *    Analisis de Algoritmos
5    *    Fecha: 21/02/2018
6    *    Autores:
7    *            Oledo Enriquez Gilberto Irving
8    *            Carrillo Balcazar Eduardo Yair
9    *            Rojas Alvarado Luis Enrique
10   *     Version: 1.0
11   *     Equipo: Fuerza.exe
12  *****************************************/
13
14
15  #include "TreeAuxFunction.c"
16
17  /*=========================================================================
18  ====================      FUNCIONES     ============================
19  ===========================================================================*/
20
21  /***************************************
22   * Nombre de la funcion: ShellSort
23   * Descripcion:
24   *             Los datos se colocan en un arbol binario de busqueda.
25   * Parametros:
26   *         Parametro Data         Un puntero a la matriz de int para ordenar.
27   *         Parametro DataSize  El tamaño de la matriz de datos.
28   * Return:
29           Nada.
30   ***************************************/
```

```
31
32 void SortWithBST(int Data[], int DataSize) {                    //=== SortWithBST
   ↪  ======
33     Node* Tree = NULL;                                         //Empezar con un árbol
       ↪  vacío
34
35     for(int i = 0; i < DataSize; i++)                          //Para cada elemento
36         IterativeInsertBST(&Tree, Data[i]);                    //insertarlo en el
           ↪  arbol
37
38     IterativeCreateInOrder(&Tree, Data, DataSize);             //Recorrido inOrden
39 }
```

A.6. TreeAuxFunction.c

```
1  /*****************************************
2   *    Instituto Politecnico Nacional
3   *    Escuela Superior de Computo
4   *    Analisis de Algoritmos
5   *    Fecha: 21/02/2018
6   *    Autores:
7   *            Oledo Enriquez Gilberto Irving
8   *            Carrillo Balcazar Eduardo Yair
9   *            Rojas Alvarado Luis Enrique
10  *     Version: 1.0
11  *     Equipo: Fuerza.exe
12  *****************************************/
13
14 /*****************************************
15  * Declaracion de un arbol binario
16  *****************************************/
17 typedef struct Node {                                          //=== Nodo ===
18    int  NodeItem;                                              //Puntero a los datos
       ↪  reales.
19    struct Node *Left;                                          //Puntero al nodo
       ↪  izquierdo.
20    struct Node *Right;                                         //Puntero al nodo
       ↪  derecho.
21 } Node;                                                       //Llamamos a esta
   ↪  estructura un nodo.
22
23 typedef Node BinaryTree;                                      //Nuevo nombre misma
   ↪  funcionalidad
24
25
26
27 /*****************************************
28  * Nombre de la funcion: CreateBinaryTree
29  * Descripcion:
30  *             Crea un puntero a un árbol binario que tenga los datos que damos.
31  * Parametros:
32  *         Parametro NewItem   Los datos que el nodo guardará.
33  * Return:
34  *         (Nodo *) Un puntero al nuevo nodo.
```

```
35  ****************************************/
36
37  extern inline Node* CreateBinaryTree(int NewItem) {                    // ==== CreateBinaryTree
    ↪  ==
38      Node *NewNode = (Node*) malloc(sizeof(Node));                      //Reserva de memoria
        ↪  para el nodo
39      NewNode->NodeItem = NewItem;                                       //Se proteje esto
40      NewNode->Left = NULL;                                              //Talvez sea una hoja
41      NewNode->Right = NULL;                                             //Talvez sea una hoja
42      return NewNode;                                                    //Se regresa puntero del
        ↪  nuevo nodo
43  }
44
45  /****************************************
46   * Nombre de la funcion: IterativeInsertBST
47   * Descripcion:
48   *             Crea un puntero a un árbol de BÚSQUEDA binario que tenga los datos que se le
    ↪  porporcionaron,
49   *            tenga en cuenta que este algoritmo NO recursivo.
50   * Parametros:
51   *         Parametro Tree       Un puntero a un puntero a una estructura de árbol.
52   *        Parametro NewItem    Los datos a insertar.
53   * Return:
54           Nada.
55   ****************************************/
56
57  void IterativeInsertBST(BinaryTree **Tree, int NewItem) {              // ====
    ↪  IterativeInsertBST ==
58      Node **NewNode = Tree;                                             //Empezamos en la raiz
59
60      while (*NewNode != NULL)                                           //Mientras no estén en
        ↪  una hoja
61          NewNode = (NewItem < (*NewNode)->NodeItem)?                    //Tenemos que movernos a
            ↪  la derecha
62              &((*NewNode)->Left): &((*NewNode)->Right);                 //Mover a la izquierda o
                ↪  derecha
63
64      (*NewNode) = CreateBinaryTree(NewItem);                            //Crear un nodo en una
        ↪  hoja
65  }
66
67
68  /****************************************
69   * Nombre de la funcion: IterativeInsertBST
70   * Descripcion:
71   *             Se rellena con los datos proporcionados, tenga en cuenta que se realiza un
72   *            recorrido inOrden
73   * Parametros:
74   *         Parametro Tree       Un puntero a un puntero a una estructura de árbol.
75   *        Parametro Data[]    Esto es un puntero a una dimensión de tamaño de datos ya
    ↪  reservada
76   * Return:
77           Nada.
78   ****************************************/
79
```

```
80  void IterativeCreateInOrder(Node **Tree, int *Data, int DataSize) { //= Crear array del
    ↪  arbol ==
81      Node *ActualNode = *Tree;                                       //Ahora, usamos un nodo
        ↪  temporal
82
83      Node **Stack = calloc(DataSize, sizeof(Node*));                 //Crear una pila como
84
85      int StackPointer = -1, i = 0;                                   //Variables axiliares
        ↪
86
87      do {                                                            //Hacer lo siguiente:
88
89          while (ActualNode != NULL) {                                //Mientras no seamos una
            ↪  hoja
90              Stack[++StackPointer] = ActualNode;                     //Añadir el nodo a la
                ↪  pila
91              ActualNode = ActualNode->Left;                          //Mover el nodo a la
                ↪  izquierda
92          }
93
94          if (StackPointer >= 0) {                                    //Ahora, mientras no
            ↪  esté vacío
95              ActualNode = Stack[StackPointer--];                     //Empezamos a sacar los
                ↪  datos
96              Data[i++] = ActualNode->NodeItem;                       //Ahora, agregamos al
                ↪  array de datos
97              ActualNode = ActualNode->Right;                         //Nos movemos a la
                ↪  derecha
98          }
99      }
100     while (ActualNode != NULL || StackPointer >= 0);                //Hacer esto mientras se
        ↪  pueda
101
102     free(Stack);                                                    //Fin Stack
103 }
```

A.7. AuxFunctions.c

```
1   /*****************************************
2    *    Instituto Politecnico Nacional
3    *    Escuela Superior de Computo
4    *    Analisis de Algoritmos
5    *    Fecha: 21/02/2018
6    *    Autores:
7    *            Oledo Enriquez Gilberto Irving
8    *            Carrillo Balcazar Eduardo Yair
9    *            Rojas Alvarado Luis Enrique
10   *     Version: 1.0
11   *     Equipo: Fuerza.exe
12  *****************************************/
13
14
15  /*==================================================================
16  ====================      FUNCIONES     ============================
```

```
17 ============================================================================*/
18
19 /****************************************
20  * Nombre de la funcion: Swap
21  * Descripcion:
22  *             Función en línea para no crear una nueva llamada a la pila de funciones para
    ↪  intercambiar elementos.
23  * Parametros:
24  *         Parametro A     Un puntero a un int.
25  *         Parametro B     Un puntero a un int.
26  * Return:
27          Nada.
28  ****************************************/
29 extern inline void Swap(int *A, int *B) {
30     int Temporal;                                                    //Crear una var temporal
31
32     Temporal = *A;                                                   //Tem <- A
33     *A = *B;                                                         //A   <- B
34     *B = Temporal;                                                   //B   <- Tem
35 }
36
37 /****************************************
38  * Nombre de la funcion: PrintArray
39  * Descripcion:
40  *             Función en línea para no crear una nueva llamada a la pila de funciones para
    ↪  imprimir la matriz.
41  * Parametros:
42  *         Parametro Data              Un puntero a la matriz de int para ordenar.
43  *         Parametro DataSize          El tamaño de la matriz de datos
44  *        Parametro FileName         Un puntero a un archivo para escribir la matriz.
45  * Return:
46          Nada.
47  ****************************************/
48
49 extern inline void fPrintArray(
50     int Data[], int DataSize, FILE * FileName) {                    //Parámetros largos
51
52     for (int i = 0; i < DataSize; ++i)                               //Elemento foreach:
53         fprintf(FileName, "%i \n", Data[i]);                         //Imprimirlo!
54
55 }
```

A.8. TestSortAlgorithms.c

```
1  /****************************************
2   *    Instituto Politecnico Nacional
3   *    Escuela Superior de Computo
4   *    Analisis de Algoritmos
5   *    Fecha: 21/02/2018
6   *    Autores:
7   *            Oledo Enriquez Gilberto Irving
8   *            Carrillo Balcazar Eduardo Yair
9   *            Rojas Alvarado Luis Enrique
10  *     Version: 1.0
```

```
 * compile "gcc -std=c11 Time.c TestSortAlgorithms.c -o TestSortAlgorithms"
 * run "./TestSortAlgorithms n NumAlgorithm OutputPlace < Input10Million.txt"
 *     Equipo: Fuerza.exe
*****************************************/



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

#include "Time.h"

#include "AuxFunctions.c"
#include "BubbleSort.c"
#include "SelectionSort.c"
#include "InsertionSort.c"
#include "ShellSort.c"
#include "SortWithBST.c"


/*=========================================================================
====================      FUNCIONES     ==============================
=========================================================================*/

/****************************************
 * Nombre de la funcion: MAIN
 * Descripcion:
 *             Este es el programa de control para verificar los algoritmos
 * Parametros:
 *         parametro argc      Tamaño de elementos de argv
 *         parametro argv[0]   Nombre del programa
 *         parametro argv[1]   n a probar (10Million.txt)
 *         parametro argv[2]   Numero de algoritmo
 * Return:
         0 SI todo es correcto.
 ****************************************/

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

    // === Leer y capturar datos  ====
    if (argc < 3) exit(0);                              //Una simple verificacion

    int DataSize = atoi(argv[1]);                       //Obtenemos DataSize
    int Algorithm = atoi(argv[2]);                      //Obtenemos el Algoritmo a probar

    double UserTimeStart, SysTimeStart, WallTimeStart;  //Variables de Inicio
    double UserTimeEnd, SysTimeEnd, WallTimeEnd;        //Variables de FIn

    FILE * FileName = fopen (argv[3], "w");             //Abrir el archivo

    int *OriginalData =
            (int*) malloc(DataSize*sizeof(int));        //Reserva de memoria

```

```
66      for (int i = 0; i < DataSize; ++i)                  //Para cada numero
67          scanf("%i", &OriginalData[i]);                  //Obtenemos el numero
68
69
70      uswtime(&UserTimeStart,
71              &SysTimeStart,
72              &WallTimeStart);                            //Comienza el conteo
73
74      if (Algorithm == 0)                                 //0 es Bubble Sort
75          BubbleSortv1(OriginalData, DataSize);           //Bubble Sort
76      else if (Algorithm == 1)                            //1 es Bubble Sort v2
77          BubbleSortv2(OriginalData, DataSize);           //Bubble Sort v2
78      else if (Algorithm == 2)                            //2 es Bubble Sort v3
79          BubbleSortv3(OriginalData, DataSize);           //Bubble Sort v3
80      else if (Algorithm == 3)                            //3 es SelectionSort
81          SelectionSort(OriginalData, DataSize);          //SelectionSort
82      else if (Algorithm == 4)                            //4 es InsertionSort
83          InsertionSort(OriginalData, DataSize);          //InsertionSort
84      else if (Algorithm == 5)                            //5 es ShellSort
85          ShellSort(OriginalData, DataSize);              //ShellSort
86      else if (Algorithm == 6)                            //6 es SortWithBST
87          SortWithBST(OriginalData, DataSize);            //SortWithBST
88
89      uswtime(
90          &UserTimeEnd,
91          &SysTimeEnd,
92          &WallTimeEnd);                                  //Fin del conteo
93
94
95      double RealTime = WallTimeEnd - WallTimeStart;      //Se obtiene la diferencia
96      double UserTime = UserTimeEnd - UserTimeStart;      //Se obtiene la diferencia
97      double SysTime  = SysTimeEnd - SysTimeStart;        //Se obtiene la diferencia
98
99      printf("%.10f %.10f %.10f", RealTime, UserTime, SysTime);
100
101
102     fPrintArray(OriginalData, DataSize, FileName);      //Ahora, ordenados
103     fclose(FileName);                                   //Cierre del archivo
104
105     return 0;                                           //FIN
106 }
```

A.9. Make.py

```
1  import os
2  import subprocess
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  '''
7  /*****************************************
8   *    Instituto Politecnico Nacional
9   *    Escuela Superior de Computo
10  *    Analisis de Algoritmos
```

```
 *    Fecha: 21/02/2018
 *    Autores:
 *            Oledo Enriquez Gilberto Irving
 *            Carrillo Balcazar Eduardo Yair
 *            Rojas Alvarado Luis Enrique
 *     Version: 1.0
 *     Equipo: Fuerza.exe
 * @run     "python3.6 Make.py"
 * @require numpy, matplotlib
 */
*****************************************/
'''

DataSize = [
    100, 1000, 5000, 10000, 50000, 100000,
    200000, 400000, 600000, 800000, 1000000, 2000000,
    3000000, 4000000, 5000000, 6000000, 7000000,
    8000000, 9000000, 10000000
]

DataExtra = [50000000, 100000000, 500000000,
1000000000, 5000000000]

Algorithms = [
    "BubbleSortv1",
    "BubbleSortv2",
    "BubbleSortv3",
    "SelectionSort",
    "InsertionSort",
    "ShellSort",
    "SortWithBST"
]

Degrees = [1, 2, 3, 4, 8]

ProgramName = "TestSortAlgorithms"
Input       = "Input10Million.txt"
Flags       = "-std=c11 Time.c"

os.system("reset")
os.system(F"gcc {Flags} {ProgramName}.c -o {ProgramName}")

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

x_real_all = []
y_real_all = []
```

```
x_poly_all = []
y_poly_all = []
poly_all   = []

file = open("../outputs/info.txt", "w")

for NumAlgorithm in range(0, len(Algorithms)):

    AlgorithmName = Algorithms[NumAlgorithm]

    x = [];
    y_real = [];
    y_cpu = [];
    y_es = [];

    for n in DataSize:

        if NumAlgorithm >= 5 or n <= 1000000:

            OutputPlace = f"../outputs/Out-{AlgorithmName}-N={n}"

            print(f"Running Algorithm {AlgorithmName} for n = {n}")

            OutputProgram = subprocess.check_output(
                f'./{ProgramName} {n} {NumAlgorithm} {OutputPlace} < {Input}',
                shell = True,
                universal_newlines = True)

            Data = [float(i) for i in OutputProgram.split()]
            RealTime = Data[0]
            UserTime = Data[1]
            SysTime = Data[2]
            CPUWall = (UserTime + SysTime) / RealTime;

            x.append(n)
            y_real.append(RealTime)
            y_cpu.append(UserTime)
            y_es.append(SysTime)

            print(f"Real:     {RealTime}s")
            print(f"User:     {UserTime}s")
            print(f"Sys:      {SysTime}s")
            print(f"CPU/Wall: {CPUWall * 100}%")
            print("")

    graficar([x, x, x], [y_real, y_cpu, y_es], ["Tiempo real", "Tiempo CPU", "Tiempo E/S"],
        "Tamaño del problema (n)", "Tiempo (s)", AlgorithmName,
        f"../graphics/{AlgorithmName}-ExperimentalTimes.png", 'o')

    file.write("===" + AlgorithmName + "===\n")
    file.write(" Real times:\n")
    for i in range(0, len(x)):
        file.write(f"({x[i]}, {y_real[i]})\n")
    file.write(" User times:\n")
    for i in range(0, len(x)):
```

```
121         file.write(f"({x[i]}, {y_cpu[i]})\n")
122     file.write(" Sys times:\n")
123     for i in range(0, len(x)):
124         file.write(f"({x[i]}, {y_es[i]})\n")
125     file.write("\n")
126 
127     x_real_all.append(x)
128     y_real_all.append(y_real)
129 
130     polynomials_x = []
131     polynomials_y = []
132     for degree in Degrees:
133         xp = np.linspace(0, x[-1], 1000)
134         polynomials_x.append(xp)
135         polynomial = np.poly1d(np.polyfit(x, y_real, degree))
136         evaluations = polynomial(xp)
137         polynomials_y.append(evaluations)
138         if (NumAlgorithm <= 4 and degree == 2) or (NumAlgorithm >= 5 and degree == 1):
139             x_poly_all.append(xp)
140             y_poly_all.append(evaluations)
141             poly_all.append(polynomial)
142     plt.plot(x, y_real, 'o')
143     graficar(polynomials_x, polynomials_y, [f"Polinomio grado {Degrees[i]}" for i in
      ↪  range(0, len(Degrees))],
144         "Tamaño del problema (n)", "Tiempo (s)", AlgorithmName,
145         f"../graphics/{AlgorithmName}-Polynomials.png", '')
146     print(end = "\n\n")
147 
148 file.close()
149 
150 info_poly = open("../outputs/polynomials.txt", "w")
151 for i in range(0, len(Algorithms)):
152     info_poly.write(Algorithms[i] + "\n")
153     info_poly.write("+".join([str(poly_all[i].c[j]) + "x^" + str(poly_all[i].order - j) for
      ↪  j in range(0, poly_all[i].order + 1)]) + "\n")
154     for j in range(0, len(DataExtra)):
155         info_poly.write(f"{DataExtra[j]}: {str(poly_all[i](DataExtra[j]))}s\n")
156     info_poly.write("\n")
157 info_poly.close()
158 
159 graficar(x_real_all[0:7], y_real_all[0:7], Algorithms[0:7], "Tamaño del problema (n)",
    ↪  "Tiempo (s)",
160     "Comparativa global de tiempos reales", "../graphics/globalComparativeTimes1.png", 'o')
161 
162 graficar(x_real_all[0:5], y_real_all[0:5], Algorithms[0:5], "Tamaño del problema (n)",
    ↪  "Tiempo (s)",
163     "Comparativa global de tiempos reales", "../graphics/globalComparativeTimes2.png", 'o')
164 
165 graficar(x_real_all[5:7], y_real_all[5:7], Algorithms[5:7], "Tamaño del problema (n)",
    ↪  "Tiempo (s)",
166     "Comparativa global de tiempos reales", "../graphics/globalComparativeTimes3.png", 'o')
167 
168 
169 graficar(x_poly_all[0:7], y_poly_all[0:7], Algorithms[0:7], "Tamaño del problema (n)",
    ↪  "Tiempo (s)",
```

```
170     "Comparativa global de aproximaciones por polinomio",
        ↪ "../graphics/globalComparativePolynomial1.png", '')
171
172 graficar(x_poly_all[0:5], y_poly_all[0:5], Algorithms[0:5], "Tamaño del problema (n)",
    ↪ "Tiempo (s)",
173     "Comparativa global de aproximaciones por polinomio",
        ↪ "../graphics/globalComparativePolynomial2.png", '')
174
175 graficar(x_poly_all[5:7], y_poly_all[5:7], Algorithms[5:7], "Tamaño del problema (n)",
    ↪ "Tiempo (s)",
176     "Comparativa global de aproximaciones por polinomio",
        ↪ "../graphics/globalComparativePolynomial3.png", '')
```

B. Compilación y ejecución

- El script completo:
  `python3.6 Make.py` (requiere las bibliotecas `matplotlib y numpy`).
- Únicamente el programa principal.
  `gcc -std=c11 Time.c TestSortAlgorithms.c -o TestSortAlgorithms"`
  `run "./TestSortAlgorithms n NumAlgorithm OutputPlace <Input10Million.txt"`

Referencias

[1] J. Hidalgo, «C++ Con Clase,» 9 Septiembre 2000. [En línea]. Available: http://c.conclase.net/orden/?. [Último acceso: 01 Marzo 2019].