## **INSTITUTO POLITÉCNICO NACIONAL**

###### ESCUELA SUPERIOR DE CÓMPUTO

Asignatura:

Análisis de algoritmos

PROFESOR:

Franco Martínez Edgardo Adrián

GRUPO: 3CM1

Practica No. 1: Pruebas a posteriori (Algoritmos de ordenamientos)

**Equipo de los Contendientes**

Integrantes:

Morales Castellanos Adolfo Erik

Pérez Leal Rodolfo

Sánchez Huescas Jorge Luis



**Introducción**

El ordenamiento de datos cumple un papel muy importante ya sea como un fin en sí o como parte de otros procesos más complejos, es por eso que se han desarrollado diferentes algoritmos de ordenamiento, que como su nombre lo dice, ordenan información de una manera especial basándose en un criterio de ordenamiento, cada uno con características específicas que lo diferencian de los demás, entre los más famosos se encuentran: Ordenamiento de burbuja, Ordenamiento por inserción, Ordenamiento con árbol binario, Ordenamiento Shell, y Ordenamiento por selección.

**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**

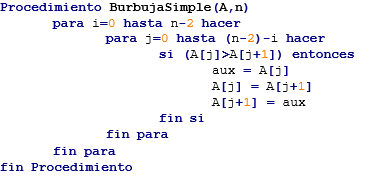
**Burbuja Simple**

El método de la burbuja es uno de los más simples, es tan fácil como comparar todos los elementos de una lista contra todos, si se cumple que uno es mayor o menor a otro, entonces los intercambia de posición.

Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado.

Para esto es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios.[1]

**Pseudocódigo**

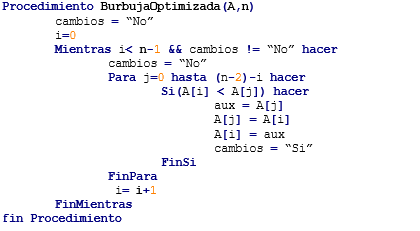


**Burbuja Optimizada**

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, reduciendo el número de iteraciones innecesarias si el arreglo ya fue ordenado totalmente.[1]

El algoritmo utiliza una bandera que cambia cuando se realiza algún intercambio y permanece igual cuando no se intercambia ningún valor pudiendo así detener el ciclo y terminar el proceso de ordenamiento cuando no se realicen intercambios, lo que indica que este ya está ordenado.[2]

**Pseudocódigo**

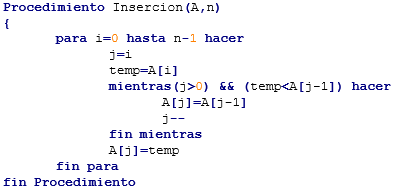
****

**Inserción (Insertion Sort)**

Este algoritmo se basa en hacer comparaciones, así que para que realice su trabajo de ordenación son imprescindibles dos cosas: un array o estructura similar de elementos comparables y un criterio claro de comparación, tal que dados dos elementos nos diga si están en orden o no.

En cada iteración del ciclo externo los elementos 0 a i forman una lista ordenada.[2]

**Pseudocódigo**

****

**Selección (Selection Sort)**

Este algoritmo consiste en lo siguiente:

·Buscar el elemento más pequeño del arreglo.

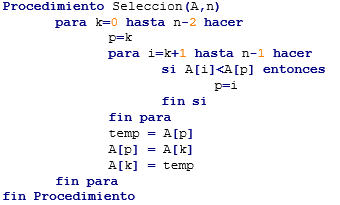
·Intercambiarlo con el elemento ubicado en la primera posición del arreglo

·Buscar el segundo elemento más pequeño de la lista.

· Intercambiarlo con el elemento que ocupa la segunda posición en la lista.

· Repites este proceso hasta que hayas ordenado toda la lista.[2]

**Pseudocódigo**

****

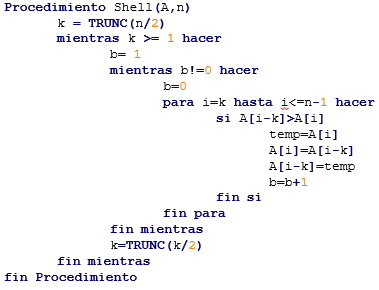
**Shell (Shell Sort)**

El método Shell es una versión mejorada del método de inserción. Este método también se conoce con el nombre de inserción con incrementos crecientes.

En el método de ordenación por inserción directa cada elemento se compara para su ubicación correcta en el arreglo, con los elementos que se encuentran en la parte izquierda del mismo. Si el elemento a insertar es más pequeño que el grupo de elementos que se encuentran a su izquierda es necesario efectuar entonces varias comparaciones antes de su ubicación.

Estas comparaciones se hacen a través de saltos de mayor tamaño que vayan disminuyendo, así los elementos quedarán ordenados en el arreglo más rápidamente.[2]

**Pseudocódigo**

****

**Ordenamiento con un árbol binario de búsqueda (Tree Sort)**

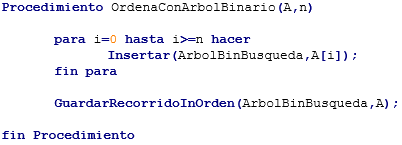
El ordenamiento con la ayuda de un árbol binario de búsqueda es muy simple debido a que solo requiere de dos pasos simples.

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.[2]

**Pseudocódigo**

****

**Implementación de los algoritmos**

**Burbuja Simple**

1. //Compilación: "gcc Insercion.c tiempo.o  -o Insercion
2. #include <stdio.h>
3. #include <stdlib.h>
5. **void** BurbujaSimple(**int** A[],**int** n);
6. **void** llenarArreglo(**int** A[], **int** n);
7. **void** mostrarArreglo(**int** A[], **int** n);
8. **void** main(**int** argc, **char** \*argv[]){
9. **int** n, \*a;
10. n = atoi(argv[1]);
11. a = (**int** \*)malloc(n\***sizeof**(**int**));//asignamos memoria para determinada n
12. llenarArreglo(a, n);//Llenamos el arreglo
13. BurbujaSimple(a, n);//Llamamos a la funcion
14. }//Fin de main
15. /\*////////////////////////////////////////////////
16. BurbujaSimple ordena los numeros comparando todos los elementos del arreglo,
17. compara todos los elementos contra todos
18. ////////////////////////////////////////////////\*/
19. **void** BurbujaSimple(**int** A[],**int** n){
20. **int** i,j,aux;
22. **for**(i=0;i<=(n-2);i++)//Primer ciclo
23. {
24. **for**(j=0;j<(n-2)-i;j++)//Segundo ciclo, que va hasta
25. {
26. **if**(A[j]>A[j+1]){//Si uno es mayor a otro los intercambia de pocision
27. aux=A[j];
28. A[j]=A[j+1];
29. A[j+1]=aux;
30. }//fin de si
31. }
32. }
33. }//FIn de BurbujaSimple

**Burbuja Optimizada**

1. //Compilación: "gcc Insercion.c tiempo.o  -o Insercion
2. #include <stdio.h>
3. #include <stdlib.h>
5. **void** BurbujaOptimizada(**int** A[],**int** n);
6. **void** llenarArreglo(**int** A[], **int** n);
7. **void** mostrarArreglo(**int** A[], **int** n);
9. **void** main(**int** argc, **char** \*argv[]){
10. **int** n, \*a;
11. n = atoi(argv[1]);
12. a = (**int** \*)malloc(n\***sizeof**(**int**));//asignamos memoria para determinada n
13. llenarArreglo(a, n);//Llenamos el arreglo
14. BurbujaOptimizada(a, n);//Llamamos a la funcion
15. }//Fin de main
17. /\*////////////////////////////////////////////////
18. BurbujaOptimizada ordena los numeros comparando los elementos del arreglo,
19. cuando el elemento mayor queda en su pocision no es necesario volver a comparar
20. ////////////////////////////////////////////////\*/
22. **void** BurbujaOptimizada(**int** A[],**int** n){
23. **int** i,j,aux,cambios;
24. i=0;
25. cambios=1;//Flag
26. **while**( (i<n-1) && cambios!=0 ){//Se realizara el while siempre y cuando i no supere el tamaño de n y cambios sea "No"
27. cambios=0;
28. **for**(j=0;j<n-1-i;j++){
29. **if**(A[j]>A[j+1]) {//Si el primero es mayor que el segundo los ordena
30. aux=A[j];
31. A[j]=A[j+1];
32. A[j+1]=aux;
33. cambios=1;
34. }//Fin de If
35. }//Fin de FOR
36. i=i+1;
37. }//Fin de While
39. }//Fin de Burbuja Optimizada

**Inserción (Insertion Sort)**

1. /\*Compilación: "gcc Insercion.c tiempo.o  -o Insercion
2. Ejecución: ./Insercion n < numeros10millones.txt >> salidaInsercion.txt
3. Codigo que utiliza el metodo de Inserción para ordenar numeros
5. \*/
7. #include <stdio.h>
8. #include <stdlib.h>
9. #include "tiempo.h"
10. //Declaración de finciones
11. **void** Insercion(**int** A[], **int** n);
12. **void** llenarArreglo(**int** A[], **int** n);
13. **void** mostrarArreglo(**int** A[], **int** n);
14. **void** mostrarTiempos(**double** wtime1, **double** wtime0, **double** utime1, **double** utime0, **double** stime1, **double** stime0);
16. /\*Función principal
17. Como argumentos recibe el tamaño del arreglo a ordenar
18. \*/
19. **void** main(**int** argc, **char** \*argv[]){
20. **int** n, \*a;  //Declaracion de la variable que define el tamaño del arreglo y un apuntador a entero para el arreglo
21. **double** utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para medición de tiempos
22. n = atoi(argv[1]);  //asignnación del tamaño del argumento por parametros
23. a = (**int** \*)malloc(n\***sizeof**(**int**));   //creacion del arreglo  de tamaño n
24. llenarArreglo(a, n);    //llama a la función para llenar el arreglo
25. uswtime(&utime0, &stime0, &wtime0); //obtener los tiempos iniciales del sitema
26. Insercion(a, n);    //llama a la función para ordenar por insercion
27. uswtime(&utime1, &stime1, &wtime1); //obtener los tiempos del sitema
28. printf("\nN = %d\n", n);
29. mostrarArreglo(a, n);   //llama a la función para mostrar el arreglo
30. printf("Tiempo total \tTiempo CPU\t Tiempo E/S \tCPU/WALL\n");
31. mostrarTiempos(wtime1, wtime0, utime1, utime0, stime1, stime0); //llama a la función para mostrar los tiempos de ejecución
32. }
34. /\*Funcion para ordenar por el metodo de insercion
35. recibe un arreglo y su tamaño
36. considera que los numeros a la izquierda de la posicion que recorre el arreglo estan ordenados
37. luego compara y lo inserta donde corresponda\*/
38. **void** Insercion(**int** A[], **int** n){
39. **int** i, j, temp;
40. **for**(i=1; i<n; i++){              //recorrer todos los elementos de arreglo
41. j=i-1;                      //empezar una posición antes de la actual
42. temp = A[i];                //guardar la posicion actual
43. **while**(j>=0 && temp<A[j]){ //recorrer a la izquierda y encontrar la posicion correcta
44. A[j+1]=A[j];            //ir cambiando de posicion la seccion "ordenada"
45. j--;                    //decremento en la posicion del arreglo
46. }
47. A[j+1]=temp;                //colocarlo en la posicion correcta
48. }
49. }
51. /\*Funcion para llenar un arreglo recibe el arreglo y su tamaño\*/
52. **void** llenarArreglo(**int** A[], **int** n){
53. **int** i;              //variable para ir rrecorrriendo el arreglo
54. **for**(i=0; i<n; i++){  //recorrer el arreglo
55. scanf("%d", &A[i]); //llenar los elementos del arreglo por entrada estandar
56. }
57. }
58. /\*Funcion para poder ver el arreglo ya ordenado recibe el arreglo y su tamaño\*/
59. **void** mostrarArreglo(**int** A[], **int** n){
60. **int** i;          //variable para ir recorriendo el arreglo
61. **for**(i=0; i<n; i++){  //recorrer el arreglo
62. printf("%d ", A[i]); //mostrar los miembros del arreglo
63. }
64. printf("\n");
65. }
66. /\*Funcion para mostrar los tiempos de ejecucion del programa
67. recibe los tiempos iniciales y finales obtenidos\*/
68. **void** mostrarTiempos(**double** wtime1, **double** wtime0, **double** utime1, **double** utime0, **double** stime1, **double** stime0){
69. //Cálculo del tiempo de ejecución del programa
70. /\*printf("%.10f \t %.10f \t %.10f \t %.10f",  wtime1 - wtime0, utime1 - utime0, stime1 - stime0, (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
71. printf("\n");\*/
72. printf("%.10e \t %.10e \t %.10e \t %.10f",  wtime1 - wtime0, utime1 - utime0, stime1 - stime0, (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
73. printf("\n");
74. }

**Selección (Selection Sort)**

/\*Compilación: "gcc Seleccion.c tiempo.o  -o seleccion

1. Ejecución: ./seleccion n < numeros10millones.txt >> salidaSeleccion.txt
2. Codigo que utiliza el metodo de Inserción para ordenar numeros\*/
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include "tiempo.h"
8. **void** Selecccion(**int** A[], **int** n);
9. **void** llenarArreglo(**int** A[], **int** n);
10. **void** mostrarArreglo(**int** A[], **int** n);
11. **void** mostrarTiempos(**double** wtime1, **double** wtime0, **double** utime1, **double** utime0, **double** stime1, **double** stime0);
12. /\*Función principal
13. Como argumentos recibe el tamaño del arreglo a ordenar
14. \*/
15. **void** main(**int** argc, **char** \*argv[]){
16. **int** n, \*a;  //Declaracion de la variable que define el tamaño del arreglo y un apuntador a entero para el arreglo
17. **double** utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para medición de tiempos
18. n = atoi(argv[1]); //asignnación del tamaño del argumento por parametros
19. a = (**int** \*)malloc(n\***sizeof**(**int**)); //creacion del arreglo  de tamaño n
20. llenarArreglo(a, n); //llama a la función para llenar el arreglo
21. uswtime(&utime0, &stime0, &wtime0); //obtener los tiempos iniciales del sitema
22. Selecccion(a, n); //llama a la función para ordenar por selecion
23. uswtime(&utime1, &stime1, &wtime1); //obtener los tiempos del sitema
24. printf("\nN = %d\n", n);
25. mostrarArreglo(a, n); //llama a la función para mortrar el arreglo
26. mostrarTiempos(wtime1, wtime0, utime1, utime0, stime1, stime0); //llama a la función para mostrar los tiempos de ejecución
27. }
28. /\*Funcion para ordenar por el metodo de seleccion
29. recibe un arreglo y su tamaño
30. Recorre el arreglo y busca el n-esimo menor y lo coloca en la posicion n\*/
31. **void** Selecccion(**int** A[], **int** n){
32. **int** k, p, i, temp;              //creacion de varibles para recorrer el arreglo y guardar temporalmente un valor
33. **for**(k=0; k<=n-2; k++){           //recorrer todo el arreglo
34. p=k;                        //guardar en p la posicion actual del arreglo
35. **for**(i=k+1; i<=n-1; i++){ //Recorrer el resto de elementos para buscar el menor
36. **if**(A[i]<A[p]){           //comparar si el elemento actual es mayor al del segundo recorrido
37. p=i;                //hacer el cambio deonde p es la posicion del menor
38. }
39. }
40. temp = A[p];                //realizar el cambio entre la posicion actual y el numero menor
41. A[p] = A[k];
42. A[k] = temp;
43. }
44. }
46. /\*Funcion para llenar un arreglo recibe el arreglo y su tamaño\*/
47. **void** llenarArreglo(**int** A[], **int** n){
48. **int** i;              //variable para ir rrecorrriendo el arreglo
49. **for**(i=0; i<n; i++){  //recorrer el arreglo
50. scanf("%d", &A[i]); //llenar los elementos del arreglo por entrada estandar
51. }
52. }
53. /\*Funcion para poder ver el arreglo ya ordenado recibe el arreglo y su tamaño\*/
54. **void** mostrarArreglo(**int** A[], **int** n){
55. **int** i;          //variable para ir recorriendo el arreglo
56. **for**(i=0; i<n; i++){  //recorrer el arreglo
57. printf("%d ", A[i]); //mostrar los miembros del arreglo
58. }
59. printf("\n");
60. }
61. /\*Funcion para mostrar los tiempos de ejecucion del programa
62. recibe los tiempos iniciales y finales obtenidos\*/
63. **void** mostrarTiempos(**double** wtime1, **double** wtime0, **double** utime1, **double** utime0, **double** stime1, **double** stime0){
64. //Cálculo del tiempo de ejecución del programa
65. /\*printf("%.10f \t %.10f \t %.10f \t %.10f",  wtime1 - wtime0, utime1 - utime0, stime1 - stime0, (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
66. printf("\n");\*/
67. printf("%.10e \t %.10e \t %.10e \t %.10f",  wtime1 - wtime0, utime1 - utime0, stime1 - stime0, (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
68. printf("\n");
69. }

**Shell (Shell Sort)**

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <math.h>
4. #include "tiempo.h"
6. **void** Shell(**int** A[],**int** n);
7. **void** llenarArreglo(**int** A[], **int** n);
8. **void** mostrarArreglo(**int** A[], **int** n);
9. **void** mostrarTiempos(**double** wtime1, **double** wtime0, **double** utime1, **double** utime0, **double** stime1, **double** stime0);
11. **int** main(**int** argc, **char** **const** \*argv[])
12. {   **int** n,b;                //Declaramos 3 variables del tipo entero
13. **int** \*A;                 //declaramos 1 variable del tipo apuntador a entero
14. **double** utime0, stime0, wtime0,utime1, stime1, wtime1; //Variables para medición de tiempos
15. n=atoi(argv[1]);        //Pedimos la cantidad de numero a ordenar
16. A = (**int** \*)malloc(n\***sizeof**(n));     //Reservamos memoria para el arreglo donde guardaremos los numeros
17. llenarArreglo(A, n);            //Mandamos a llamar la funcion llenarArreglo donde enviamos el arreglo y la cantidad de elementos a ordenar
18. uswtime(&utime0, &stime0, &wtime0);     //Activamos la funcion del tiempo
19. Shell(A,n);                     //Llamamos la funcion que nos ordenara el arreglo con los numeros
20. **for** (b = 0; b < n; b++)
21. {
22. printf("%d ",A[b]);             //Se imprime el arreglo ordenado
23. }
24. uswtime(&utime1, &stime1, &wtime1);//Se detiene la funcion tiempo
25. //mostrarArreglo(A, n);
26. printf("Tiempo total \tTiempo CPU\t Tiempo E/S \tCPU/WALL\n");      //Mostramos los tiempos
27. mostrarTiempos(wtime1, wtime0, utime1, utime0, stime1, stime0);
28. **return** 0;
29. }

32. **void** Shell(**int** A[],**int** n){              //Recibimos el arreglo a ordenar y el numero de elementos
33. **int** k,b,i,temp;                     //Declaramos 4 variables del tipo entero
34. k=trunc(n/2);                       //Asignamos a la variable k el valor del tamaño del arreglo divido en 2 sin valores flotantes mediante la funcion trunc
35. printf("k=%d\n",k);                 //Imprime el valor que le asignamos a k
36. **while**(k>=1){                     //Entramos en un ciclo donde mientras k sea mayor o igual 1, permaneceremos en el
37. b=1;                            //Asignamos a b el valor de 1
38. **while**(b!=0){                    //Entramos en un ciclo donde mientras b sea diferente de 1, permaneceremos en el
39. b=0;                        //Asignamos a b el valor de 0
40. **for** (i = k;i<=n-1;i++)       //Entramos a un for donde le asigamos el valor de i el de k, y permaneceremos en el for mientras i sea menor o igual al tamaño del arreglo menos 1
41. {
42. **if** (A[i-k]>A[i])     //si el valor del arreglo que se encuentra en la posicion i-k es mayor al de la posicon i se hara este proceso
43. {
44. temp=A[i];          //asignamos el valor de arreglo en la posicion i a una variable temporal
45. A[i]=A[i-k];        //intercambiamos los valores del arreglo en la posiocion i e i-k
46. A[i-k]=temp;
47. b=b+1;              //incrementamos a b
48. }
49. }
50. }
51. k=trunc(k/2);                   //Disminuimos el valor de k en k/2 redondiando el resultdo hacia abajo(piso) con la funcion trunc
52. }

55. }
56. **void** llenarArreglo(**int** A[], **int** n){
57. **int** i;
58. **for**(i=0; i<n; i++){
59. scanf("%d", &A[i]);
60. }
61. }
63. **void** mostrarArreglo(**int** A[], **int** n){
64. **int** i;
65. **for**(i=0; i<n; i++){
66. printf("%d ", A[i]);
67. }
68. printf("\n");
69. }
70. **void** mostrarTiempos(**double** wtime1, **double** wtime0, **double** utime1, **double** utime0, **double** stime1, **double** stime0){
71. //Cálculo del tiempo de ejecución del programa
72. /\*printf("%.10f \t %.10f \t %.10f \t %.10f",  wtime1 - wtime0, utime1 - utime0, stime1 - stime0, (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
73. printf("\n");\*/
74. printf("%.10e \t %.10e \t %.10e \t %.10f",  wtime1 - wtime0, utime1 - utime0, stime1 - stime0, (utime1 - utime0 + stime1 - stime0) / (wtime1 - wtime0));
75. printf("\n");
76. }

**Ordenamiento con árbol binario de búsqueda (Tree Sort)**

**arbolbb.h**

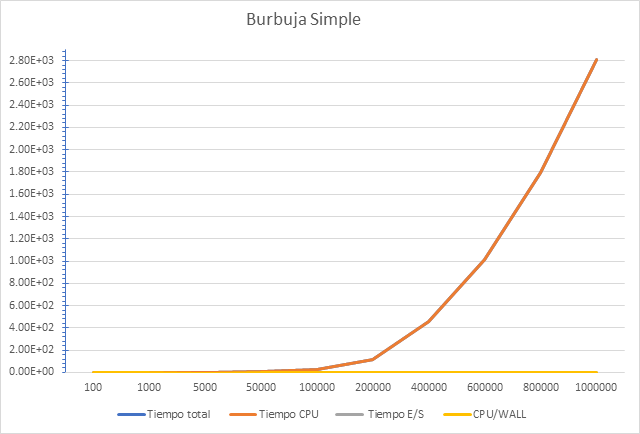
1. #include <stdio.h>
2. #include <stdlib.h>
3. #ifndef arbolbb
4. #define arbolbb
6. **typedef** **struct** nodo         //Creamos una estructura del tipo nodo que tiene como valores
7. {
8. **int** valor;          //Declaramos una variable del tipo entero
9. **struct** nodo\* izq;   //Declaramos un apuntador del tipo estructura a nodo
10. **struct** nodo\* der;   //Declaramos un apuntador del tipo estructura a nodo
11. }Nodo;
13. **typedef** Nodo Arbol;     //Re-definimos el nombre de la estructura Nodo a Arbol
15. Nodo\* CrearNodo(**int** valor);                     //Declaramos las funciones que utilizaremos en el documento tipo c
16. **void** Insertar(Nodo\*\* arbol, **int** valor);
17. **void** Preorden(Nodo\* arbol);
18. **void** GuardarRecorridoInOrden(Nodo\* arbol,**int** A[]);
19. **void** Postorden(Nodo\* arbol);
20. **void** OrdenaConArbolBinario(**int** A[],**int** n);
22. #endif

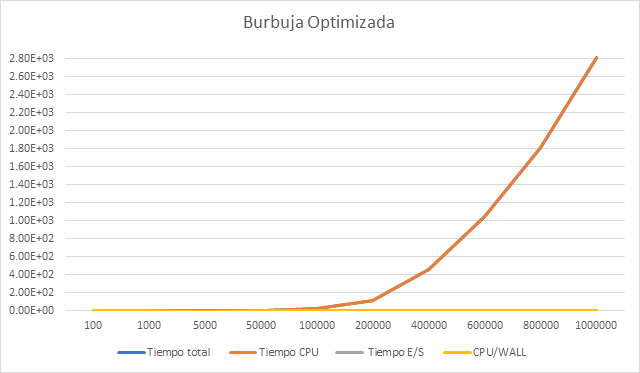
**arbolbb.c**

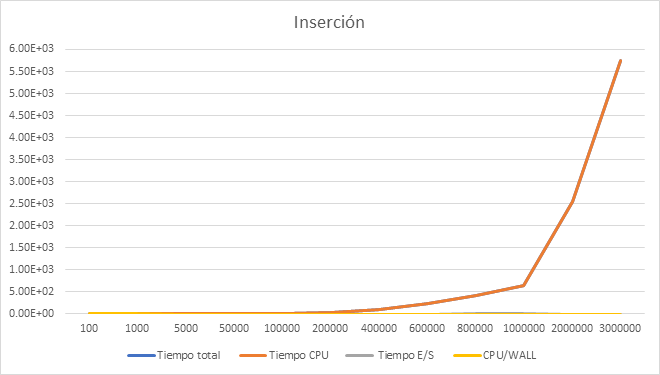
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include "arbolbb.h"
4. #include "tiempo.h"
6. Nodo\* CrearNodo(**int** valor);
7. **void** Insertar(Nodo\*\* arbol, **int** valor);
8. **void** Preorden(Nodo\* arbol);
9. **void** GuardarRecorridoInOrden(Nodo\* arbol,**int** A[]);
10. **void** Postorden(Nodo\* arbol);
11. **void** OrdenaConArbolBinario(**int** A[],**int** n);
12. **int** i=0;        //Se declara una variable global y se inicializa.

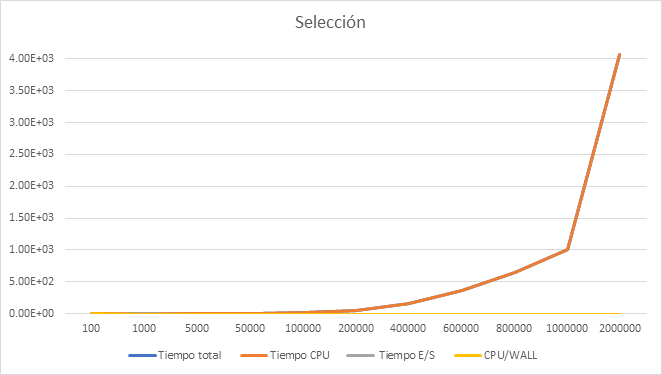
15. **int** main(**int** argc, **char** **const** \*argv[])
16. {
17. **int** x,\*A,n;     // se crean 2 variables del tipo entero y una del tipo apuntador a entero
19. /\*printf("Numeros N\n");        // se piden el numero de numeros que quiere ordenar
20. scanf("%d",&n);\*/
21. n = atoi(argv[1]);          //Tamaño del arreglo a crear
22. A = (**int** \*)malloc(n\***sizeof**(**int**));   // Creamos el Arreglo
23. **for** (**int** x = 0; x < n; ++x)
24. {
25. scanf("%d",&A[x]);          //Los ingresamos en el arreglo
26. }
28. OrdenaConArbolBinario(A,n);
30. **return** 0;
31. }
33. Nodo\* CrearNodo(**int** valor){
34. Nodo\* nuevoNodo = (Nodo \*) malloc(**sizeof**(Nodo)); //Se crea un apuntador del tipo Nodo y se le asigna memoria dinamicamnete
35. nuevoNodo->valor = valor;        //Por medio del operador flecha, asignamos el valor que recibe la funcion al valor del Nodo
36. nuevoNodo->izq = nuevoNodo->der = NULL;   //Igualamos los valores de los apuntadores a NULL
37. **return** nuevoNodo;       // retornamos el Nodo
38. }
40. **void** Insertar(Nodo\*\* arbol, **int** valor){     // Para insertar un valor al arbol, la funcion insertar recibe un doble apuntador (por la recursividad) y el valor a ingresar.
41. **if** (\*arbol == NULL) //Preguntamos si el Nodo apunta a null (si el arbol esta vacio)
42. {
43. \*arbol = CrearNodo(valor);  //Mandamos a llamar la funcion CrearNodo y le mandamos el valor a ingrear al árbol
45. } **else** {
46. **int** valorRaiz = (\*arbol)->valor;     //Creamos un entero donde guardaremos el valor de la raiz
47. **if** (valor < valorRaiz)           //Verificamos si el el valor de la raiz es mayor que el valor que se quiere insertar
48. {
49. Insertar(&(\*arbol)->izq,valor);      //Mandamos recursivamente el valor a ingresar y el nodo izquierdo por ser menor el valor a ingresar
50. } **else**{
51. Insertar(&(\*arbol)->der,valor);      //Mandamos recursivamente el valor a ingresar y el nodo derecho por ser mayor el valor a ingresar
52. }
53. }
54. }
56. **void** GuardarRecorridoInOrden(Nodo\* arbol,**int** A[]){      //Se recibe el arbol y el valor a ingresar.
57. **if** (arbol == NULL)      //verificamos que el nodo que se envio este vacio para verificar que sea el ultimo nodo hijo
58. {
59. //  printf("-");
60. } **else** {
61. GuardarRecorridoInOrden(arbol->izq,A);       //Se envia recursivamente el nodo izquierdo
62. A[i]= arbol->valor;              // guardamos en el arreglo el valor más a la izquierda del arbol(menor)
63. i++;                            // incrementamos la variable
64. GuardarRecorridoInOrden(arbol->der,A);       //Se envia recursivamente el nodo izquierdo
65. }
66. }
68. **void** OrdenaConArbolBinario(**int** A[],**int** n){          //Se recibe el arreglo a ordenar junto con el tamaño del mismo
69. Nodo \*ArbolBinBusqueda = NULL;                  //Se crea un arbol vacio
70. **int** i;
71. **for** (i = 0; i <n; ++i)
72. {
73. Insertar(&ArbolBinBusqueda,A[i]);           //Mandamos a llamar la funcion Insertar para llenar el arbol con los valores del arreglo
74. }
75. GuardarRecorridoInOrden(ArbolBinBusqueda,A);    //Llamamos la funcion que nos ordenara el arreglo inorden
76. }

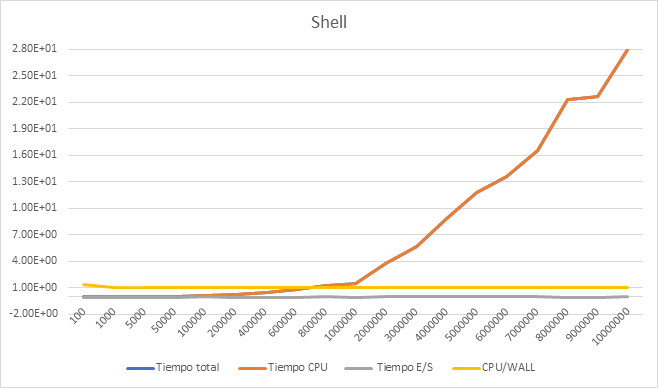
**Actividades y pruebas**

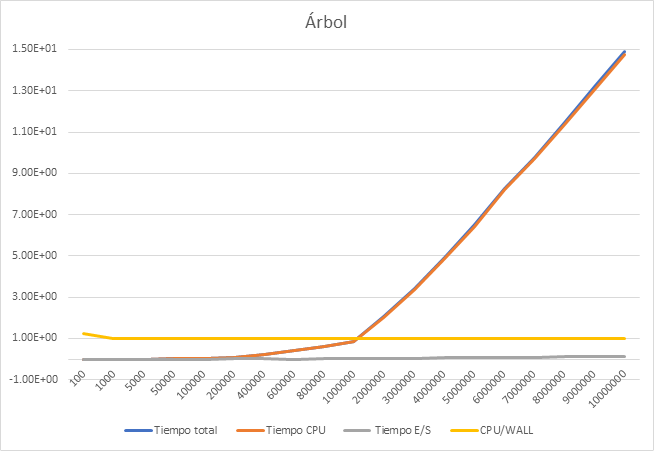
****

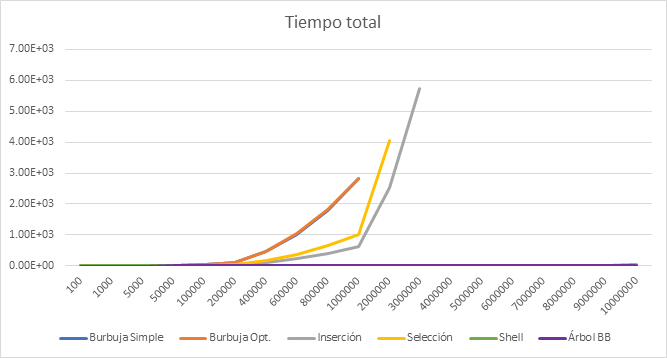
****

****

****

****

****



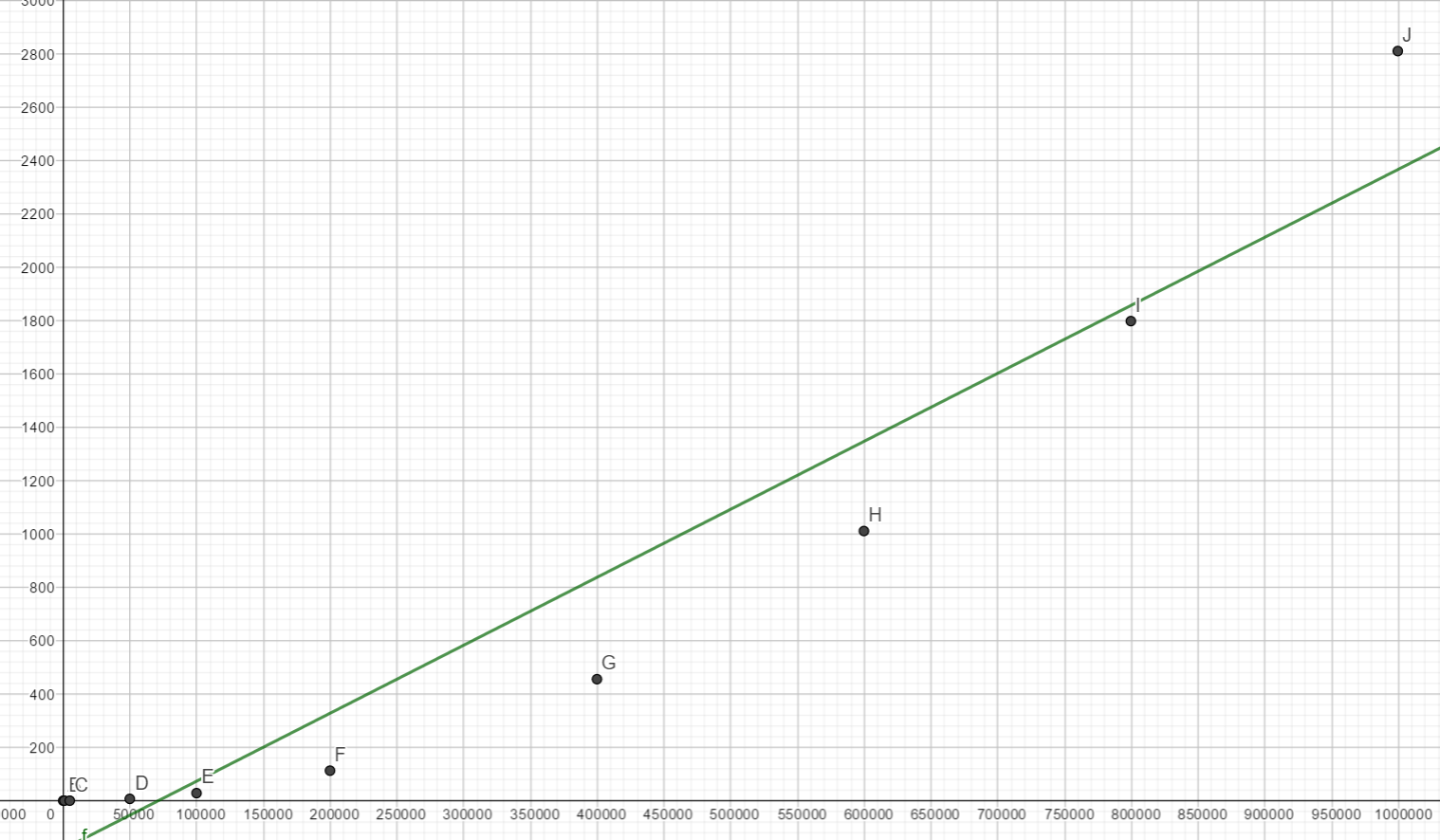
AJUSTE POLINOMIAL

Algoritmo de burbuja simple

Para el algoritmo de burbuja los polinomios obtenidos son:

Grado 1:

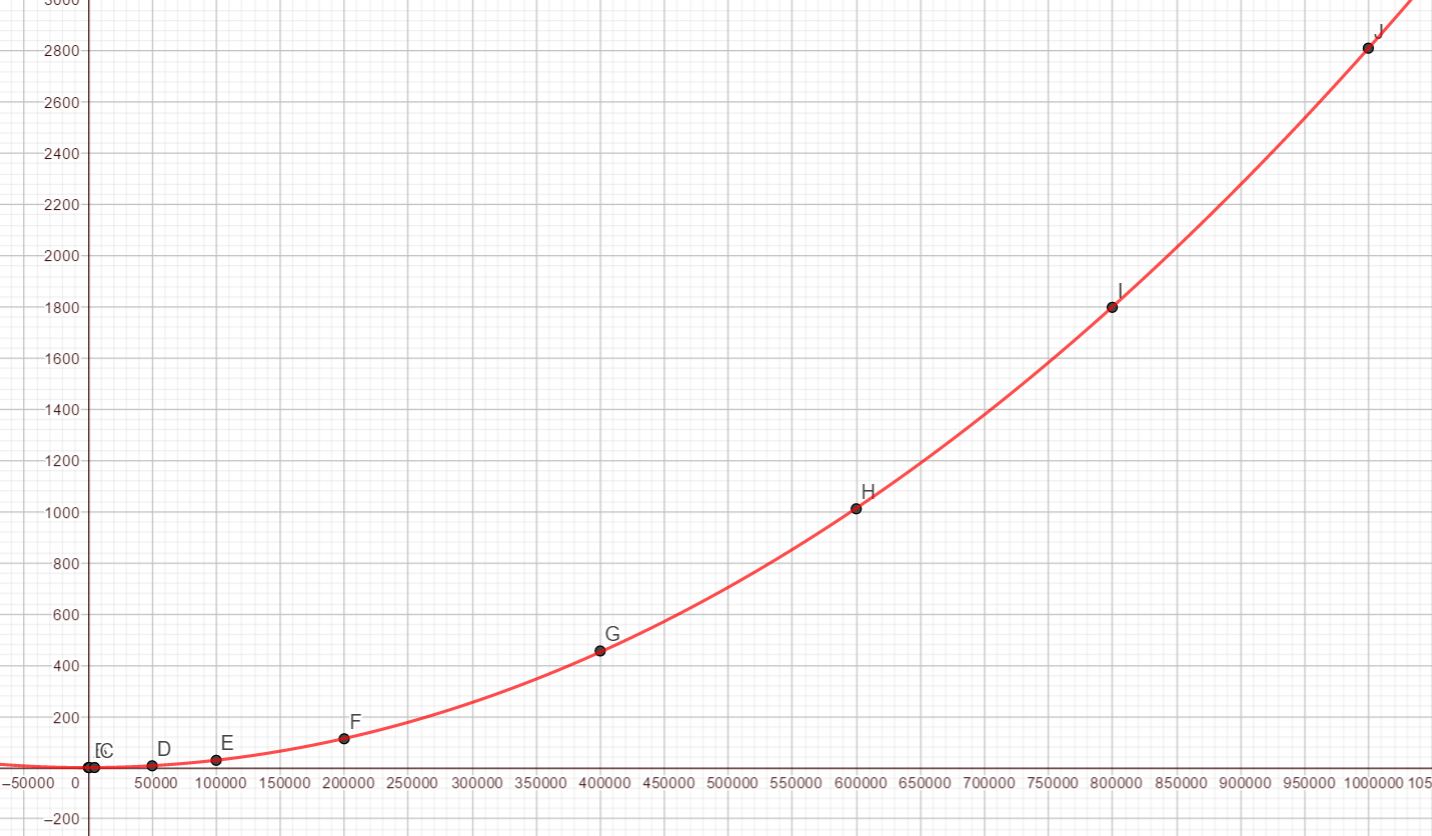




Podemos notar que no es conveniente escoger un polinomio de grado 1 para este algoritmo pues solo se aproxima a los valores reales en los extremos.

Grado 2:





Podemos notar que para los valores de n grandes el polinomio se acerca lo suficiente para considerar que es una buena aproximación.

Para polinomios mayores la aplicación le asigna coeficientes de 0, lo que hace que sea el polinomio de orden 2.

En base al polinomio se calculó la siguiente tabla.

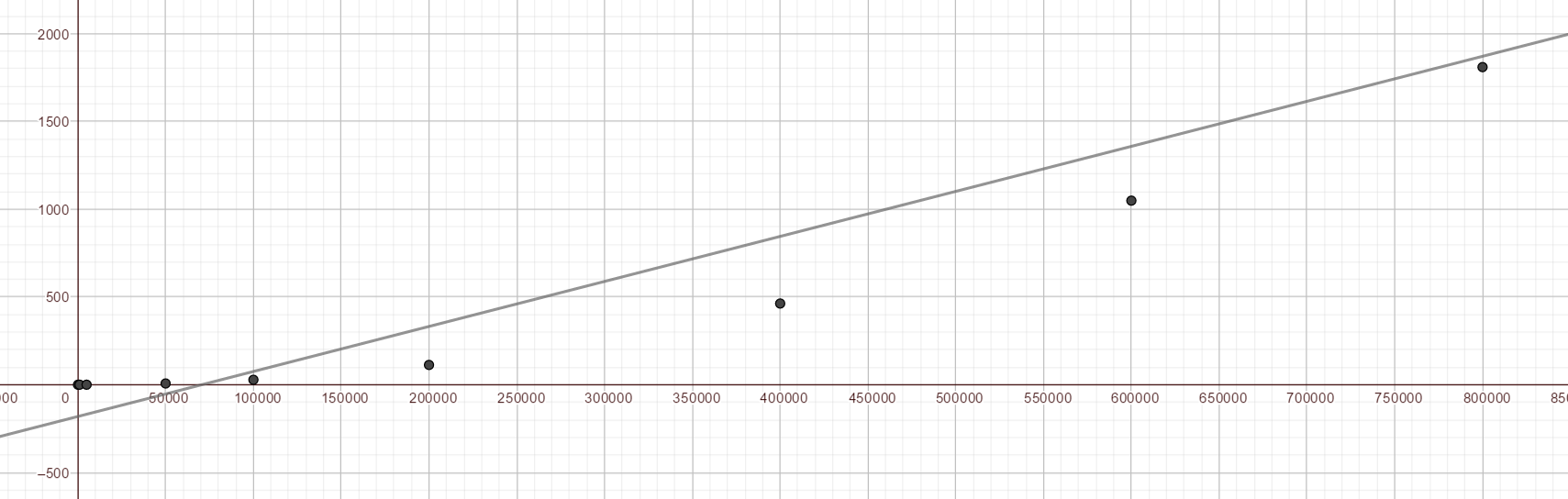
|  |  |
| --- | --- |
| Tamaño de problema (n) | Tiempo (Segundos) |
| 50000000 |  |
| 100000000 |  |
| 500000000 |  |
| 1000000000 |  |

Algoritmo burbuja optimizada

Para el algoritmo de burbuja optimizado se tienen los siguientes polinomios:

Grado 1:

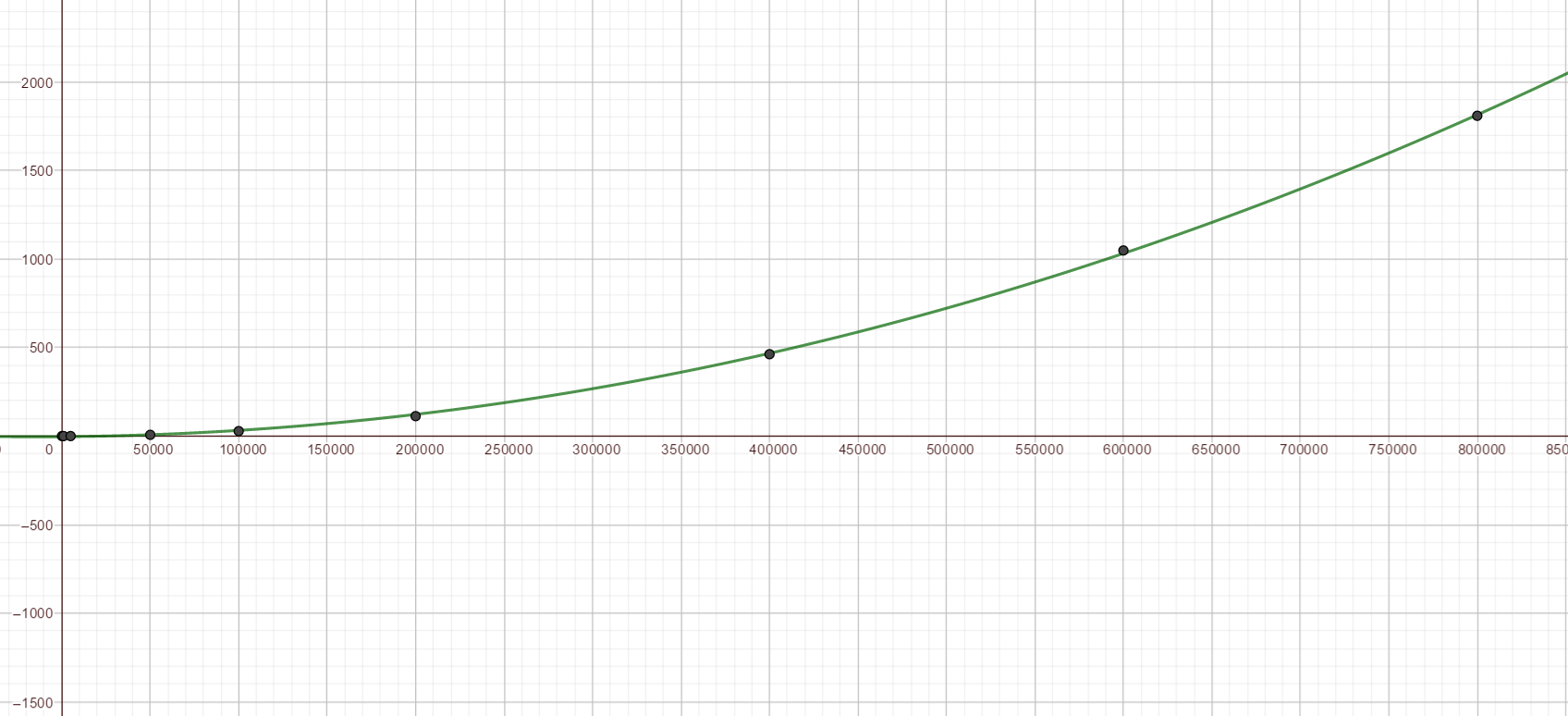




Podemos notar que no es conveniente escoger un polinomio de grado 1 para este algoritmo pues solo se aproxima a los valores reales en los extremos.

Grado 2:





Podemos notar que para los valores de n grandes el polinomio se acerca lo suficiente para considerar que es una buena aproximación.

Para polinomios mayores la aplicación le asigna coeficientes de 0, lo que hace que sea el polinomio de orden 2.

En base al polinomio se calculó la tabla:



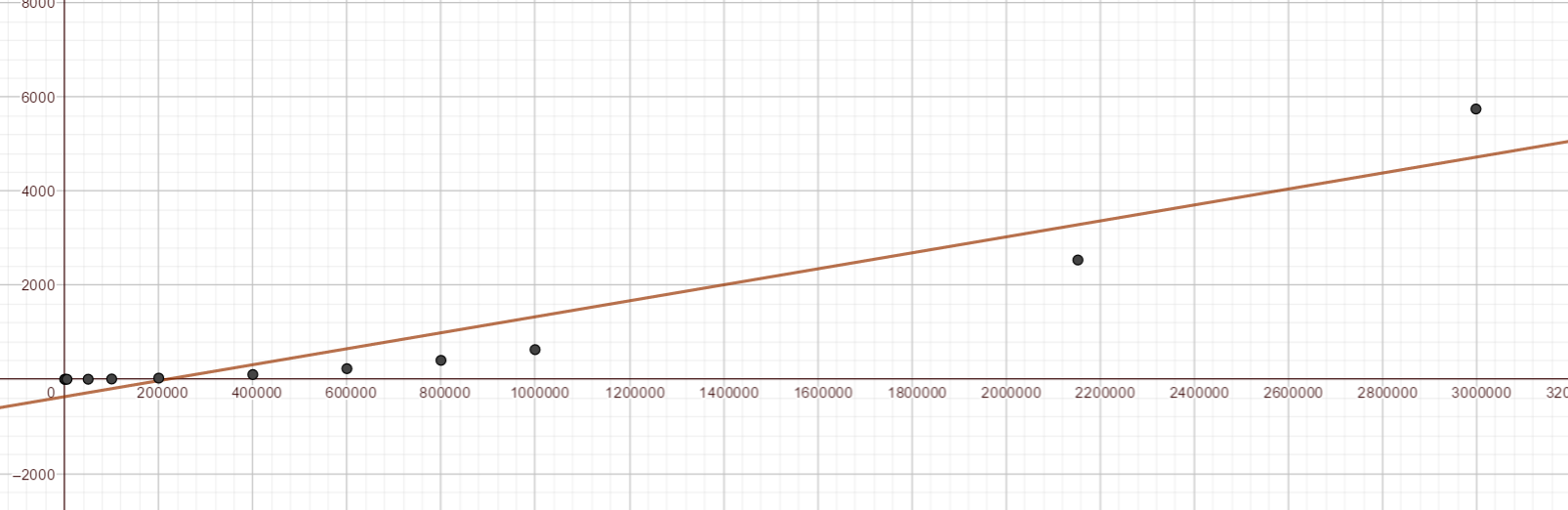
|  |  |
| --- | --- |
| Tamaño de problema (n) | Tiempo (Segundos) |
| 50000000 |  |
| 100000000 |  |
| 500000000 |  |
| 1000000000 |  |

Algoritmo de Inserción

Para el algoritmo de Inserción se tienen los siguientes polinomios.

Grado 1:

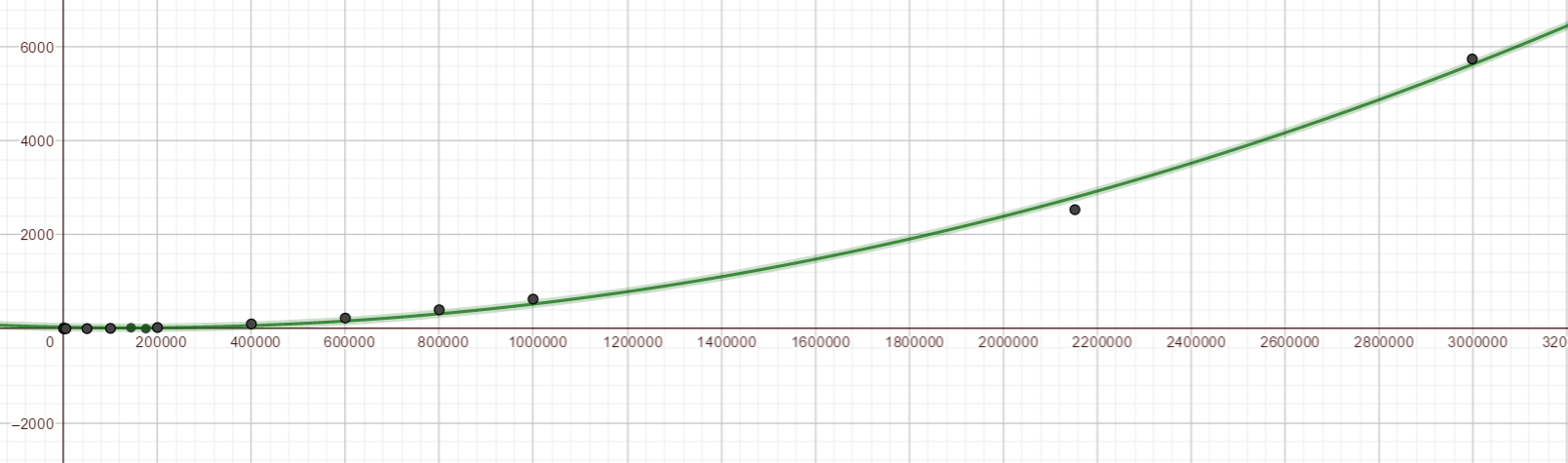




Podemos notar que no es conveniente escoger un polinomio de grado 1 para este algoritmo pues solo se aproxima a los valores reales en los extremos.

Grado 2:





Podemos notar que para los valores de n grandes el polinomio se acerca lo suficiente para considerar que es una buena aproximación.

Para polinomios mayores la aplicación le asigna coeficientes de 0, lo que hace que sea el polinomio de orden 2.

En base al polinomio  se calculó la tabla:

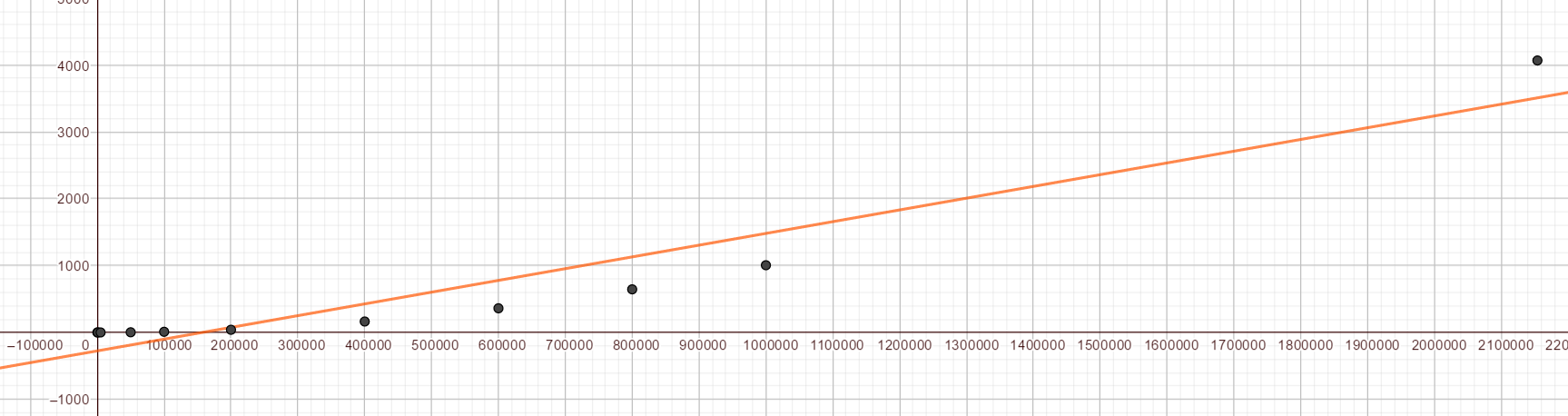
|  |  |
| --- | --- |
| Tamaño de problema (n) | Tiempo (Segundos) |
| 50000000 |  |
| 100000000 |  |
| 500000000 |  |
| 1000000000 |  |

Algoritmo selección

Para el algoritmo de selección se tiene lo siguiente

Grado 1:

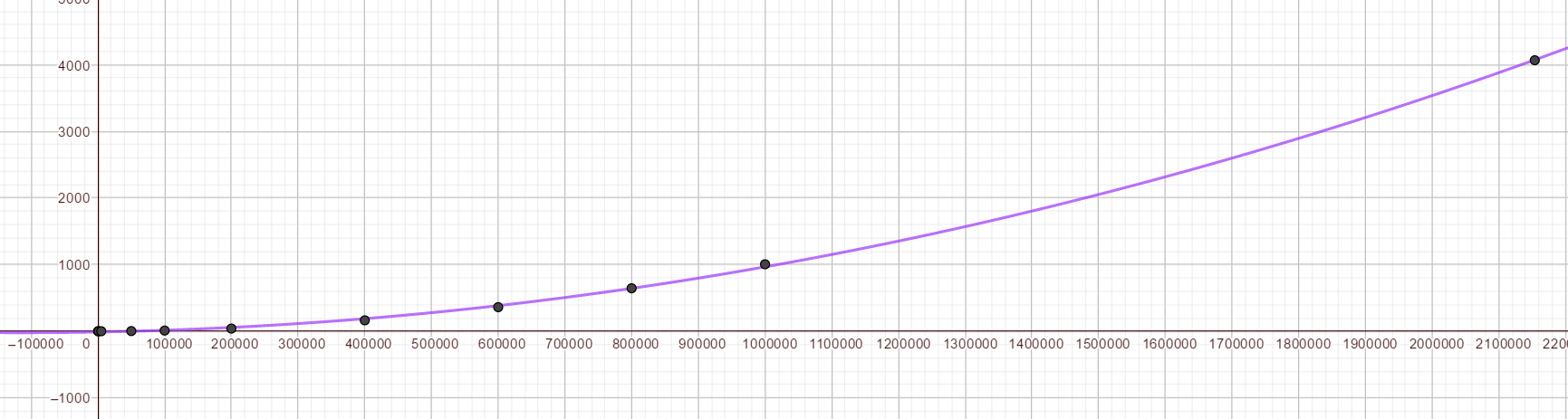




Podemos notar que no es conveniente escoger un polinomio de grado 1 para este algoritmo pues solo se aproxima a los valores reales en los extremos.

Grado 2:





Podemos notar que para los valores de n grandes el polinomio se acerca lo suficiente para considerar que es una buena aproximación.

Para polinomios mayores la aplicación le asigna coeficientes de 0, lo que hace que sea el polinomio de orden 2.

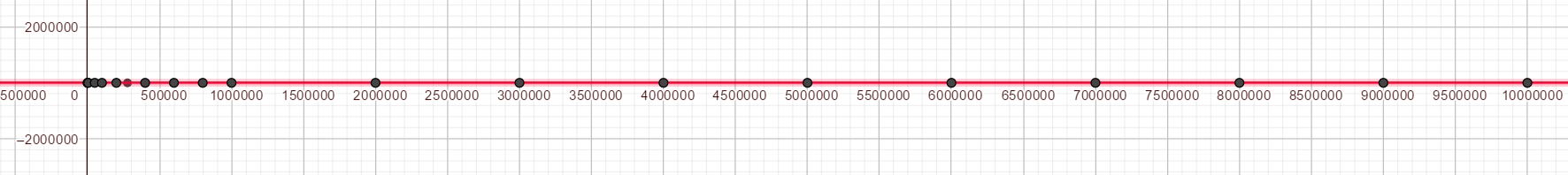
En base al polinomio  se calculó la tabla:

|  |  |
| --- | --- |
| Tamaño de problema (n) | Tiempo (Segundos) |
| 50000000 |  |
| 100000000 |  |
| 500000000 |  |
| 1000000000 |  |

Algoritmo Shell

Grado 1:





Podemos notar que para los valores de n grandes el polinomio se acerca lo suficiente para considerar que es una buena aproximación.

Para polinomios mayores la aplicación le asigna coeficientes de 0, lo que hace que sea el polinomio de orden 1.

En base al polinomio  se calculó la tabla:

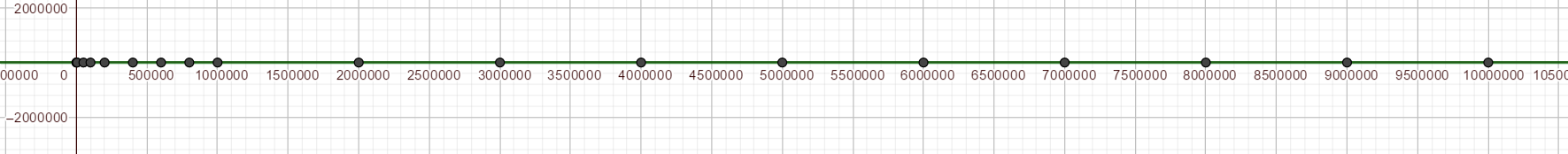
|  |  |
| --- | --- |
| Tamaño de problema (n) | Tiempo (Segundos) |
| 50000000 | 131.7704566 |
| 100000000 | 264.2754566 |
| 500000000 | 1324.315457 |
| 1000000000 | 2649.365457 |

Árbol de búsqueda binaria

En el caso del árbol de búsqueda binario obtenemos los siguientes polinomios.

Grado 1:





Podemos notar que para los valores de n grandes el polinomio se acerca lo suficiente para considerar que es una buena aproximación.

Para polinomios mayores la aplicación le asigna coeficientes de 0, lo que hace que sea el polinomio de orden 1.

En base al polinomio  se calculó la tabla:

|  |  |
| --- | --- |
| Tamaño de problema (n) | Tiempo (Segundos) |
| 50000000 | 73.05503516 |
| 100000000 | 146.4900352 |
| 500000000 | 733.9700352 |
| 1000000000 | 1468.320035 |

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

Burbuja Simple

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

Shell

iii. ¿Cuál algoritmo tiene menor complejidad temporal? Shell

iv. ¿Cuál algoritmo tiene mayor complejidad temporal? Burbuja Simple

v. ¿Cuál algoritmo tiene menor complejidad espacial? Burbuja e inserción ¿Por qué? Porque solo se ocupa el arreglo donde se almacenan los datos y una variable auxiliar temporal que nos ayuda a desplazar los elementos del arreglo de un lado a otro.

vi. ¿Cuál algoritmo tiene mayor complejidad espacial? Árbol BB ¿Por qué? Porque para cada dato insertado en el arreglo se tenía que crear un nodo del árbol, el cual es una estructura que tiene un valor entero y 2 apuntadores a nodos.

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

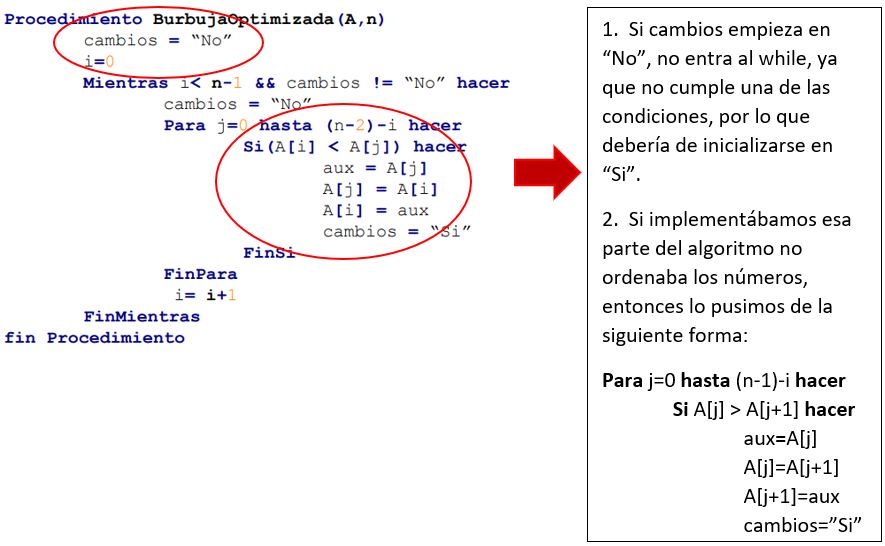
si ¿Por qué? porque como lo vimos en clase, cada algoritmo manejaba su información de distintas maneras, los resultados que obtuvimos en la experimentación fueron variados, pero se comprobó la eficiencia de cada algoritmo.

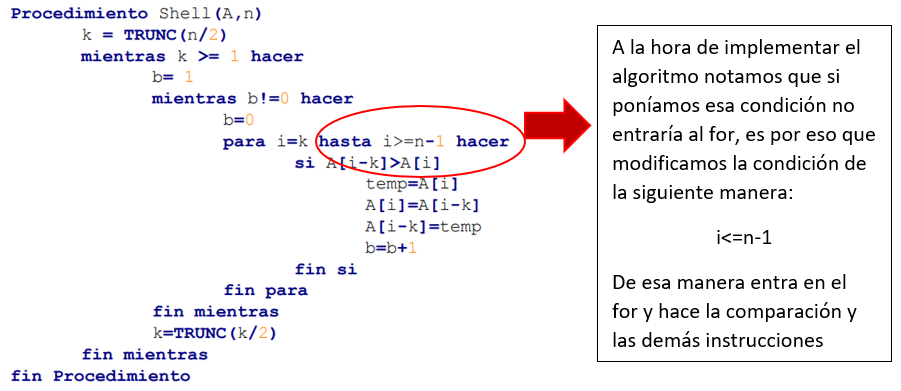
viii. ¿Sus resultados experimentales difieren mucho de los del resto de los equipos? Sí ¿A qué se debe? A las distintas características de cada computadora (Procesador, Memoria RAM)

ix. ¿Existió un entorno controlado para realizar las pruebas experimentales? Si ¿Cuál fue? Se finalizaron todos los procesos externos y solo se dejaron activos los procesos del sistema.

x. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica? Entender los algoritmos al 100% antes de implementarlos, ya que si llegara a existir un error en el código, será más fácil identificarlo.

**Errores detectados**





**Posibles mejoras**

**Conclusiones**

*Morales Castellanos Adolfo Erik*  
La práctica se realizó con resultados positivos, ya que a pesar de las dificultades que se presentaron, supimos cómo resolver cada uno de los problemas. De igual manera se observó la velocidad y el desempeño que tiene cada algoritmo al medir el tiempo que tardaba cada uno de los procesos del algoritmo, observando la efectividad de cada algoritmo demostrando que el shell llega a ser más rápido que un árbol binario de búsqueda.

*Pérez Leal Rodolfo*

El desarrollo de la práctica fue de utilidad para verificar que a pesar de que existen distintos algoritmos de ordenamiento, no es lo mismo ocupar cualquiera de ellos, pues esto depende mucho de cómo se requiera implementar, en especial para un tamaño de problema muy grande algoritmos como burbuja o selección no son tan factibles, pues al realizar varias comparaciones se demora mucho a diferencia de Shell o un árbol que son más eficientes. A pesar de que se presentaron ciertos problemas en el desarrollo este fue relativamente sencillo porque los algoritmos ocupados no eran muy difíciles de implementar una vez entendido su funcionamiento.

*Sanchez Huescas Jorge Luis*

Los algoritmos de ordenamiento los podemos implementar cuando requerimos ordenar cierta información bajo un criterio en específico, existen muchos métodos de ordenamiento y en esta práctica analizamos 6 de ellos y llegamos a la conclusión de que hay algunos mejores que otros en cuanto al ahorro de recursos, ya sea para ahorrar memoria o tiempo pues para 10 millones de números; que fue la mayor cantidad de números que ordenamos, es necesario escoger un algoritmo óptimo. En lo personal tuve algunas dificultades en implementar los algoritmos, pero una vez entendido qué hace el algoritmo fue fácil de resolver.

**Bibliografía**

# 

|  |  |
| --- | --- |
| [1] | E. A. F. Martínez, «Web Personal de Edgardo Adrian Franco Martinez,» 2010. [En línea]. Available: http://www.eafranco.com/. [Último acceso: 10 marzo 2018]. |
| [2] | A. G. S. M. G. A. J. Exposito Lopez Daniel, «Arboles Binarios de Busqueda,» Universidad de Granada, [En línea]. Available: http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/arb\_BB.htm. [Último acceso: 10 marzo 2018]. |
| [3] | «El rincón de Zerial,» [En línea]. Available: https://blog.zerial.org/. [Último acceso: 10 marzo 2018]. |

[1]

[2]

[3]