PRÁCTICA 01

Pruebas a posteriori

Integrantes
Manica Quintero Marco Antonio
Martínez Ramirez Serge Eduardo
Ramirez Sanchez Miriam Guadalupe
Villegas Gómez Alejandra

SIC MUNDUS CREATUS EST Análisis y Diseño de Algoritmos

Tabla de contenidos

Tabla de ilustraciones

Planteamiento del problema

En esta práctica ejecutaremos diferentes algoritmos de ordenamiento, con la finalidad de calcular su función de complejidad temporal correspondiente. A su vez, esto nos permitirá reconocer las diferencias entre estos diferentes métodos y las aplicaciones que tienen, así como su eficiencia para ciertos problemas concretos o generales.

ENTORNO DE PRUEBAS

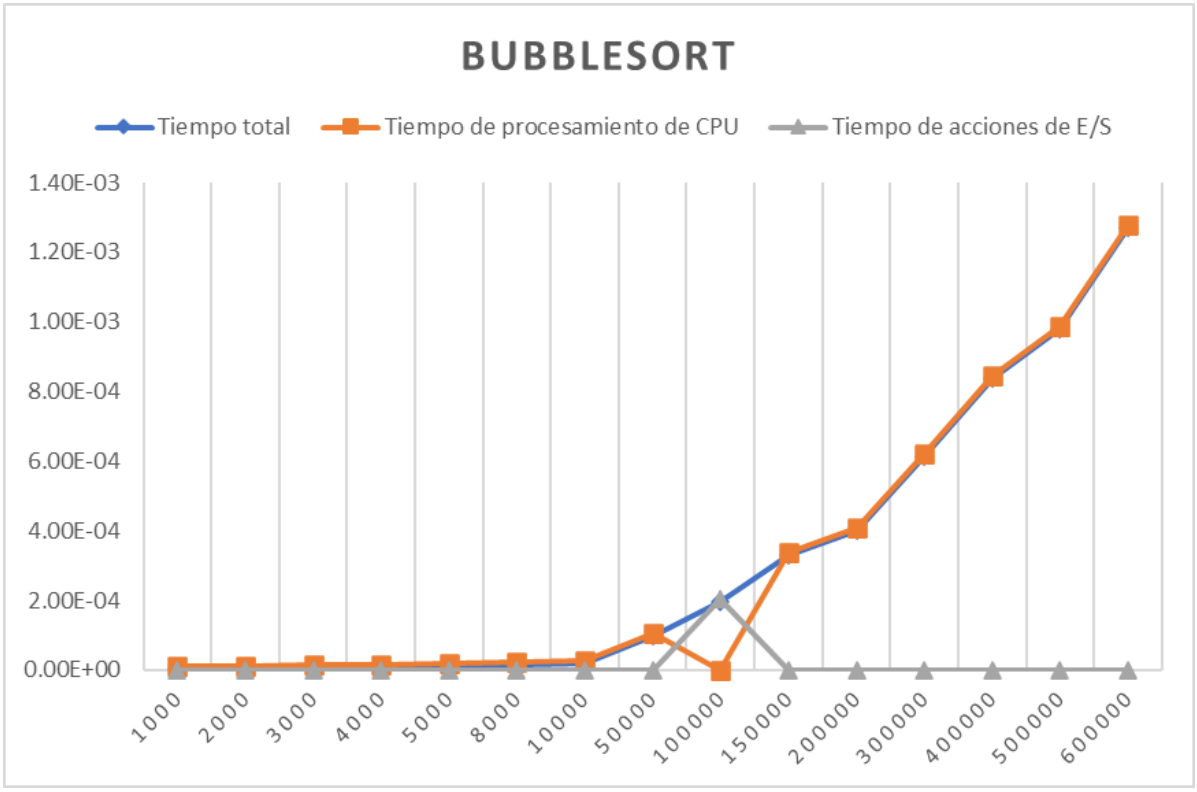
Dispositivo	LAPTOP-38E09JUU ASUS
Procesador	11th Gen Intel(R) Core (TM) i3-1115G4 @ 3.00GHz 3.00 GHz
RAM instalada	12.0 GB
Identificador de dispositivo	C6EF96BC-0486-4AEF-A67A-838012A9A1FB
Id. del producto	00342-41446-13904-AAOEM
Tipo de sistema	Sistema operativo de 64 bits, procesador basado en x64

Tiempos de ejecución

Algoritmo	Tiempo Real	Tiempo en CPU	Tiempo E/S	%CPU/Wall
Bubble Sort	0.000977 s	0.000986 s	0 s	100.81871 %
Burbuja Optimizada 1	0.001087 s	0.001097 s	0 s	100.83610 %
Burbuja Optimizada 2	0.000988 s	0.000996 s	0 s	100.71183 %
Insertion Sort	0.000977 s	0.000986 s	0 s	100.81871 %
Selection Sort	0.000978 s	0.000985 s	0 s	100.61834 %
Shell Sort	0.000978 s	0.000985 s	0 s	100.61834 %
Tree Sort	0.000977 s	0.000985 s	0 s	100.71646 %
Merge Sort	0.000946 s	0.000954 s	0 s	100.84087 %
Quick Sort	0.000977 s	0.000985 s	0 s	100.71646 %
Heap Sort	0.001021 s	0.001030 s	0 s	100.79638 %

Tabla 1. Tiempos de ejecución con 500000 números

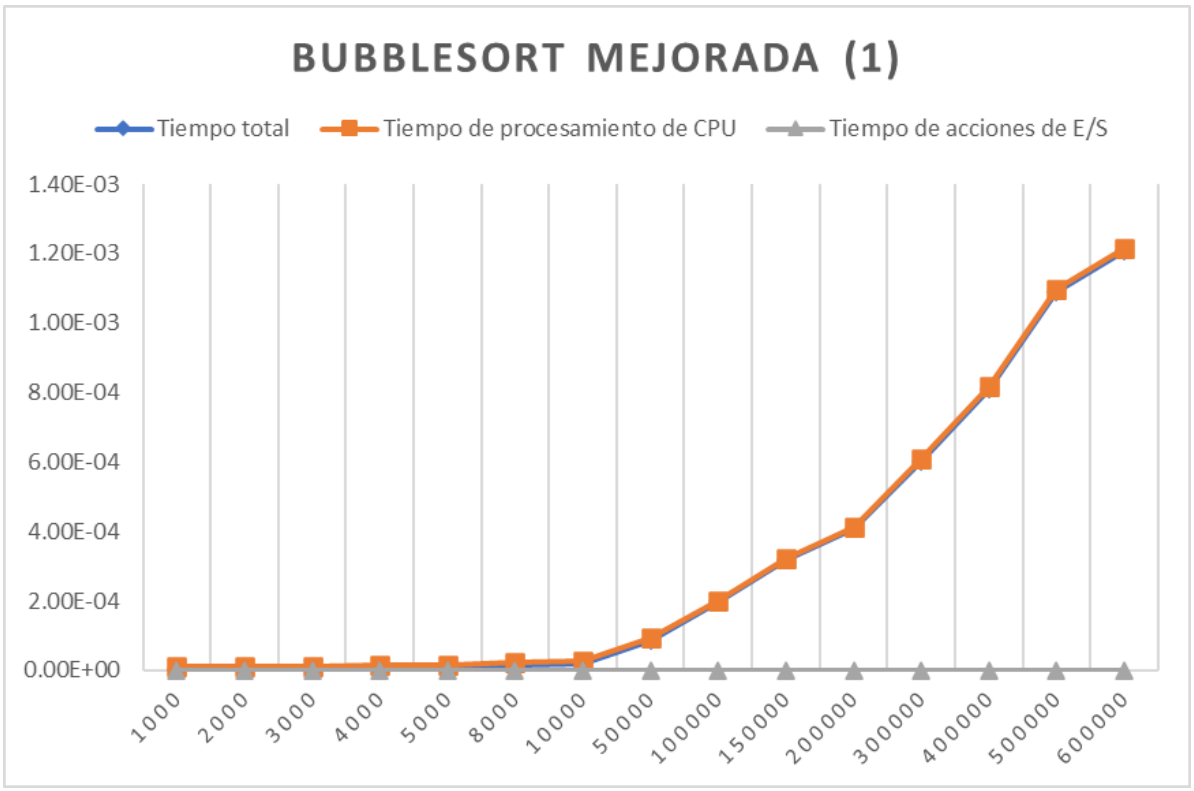
Códigos y gráficas con tiempo de ejecución



Grafica tiempo de ejecucion 1 Bubble Sort

```
void BurbujaS(int A[], int n) {
    int i, j, aux;
    for (i = 0; i <= n - 2; i++) {
        for (j = 0; j <= n - 2; j++)
            if (A[j] > A[j + 1]) {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
```

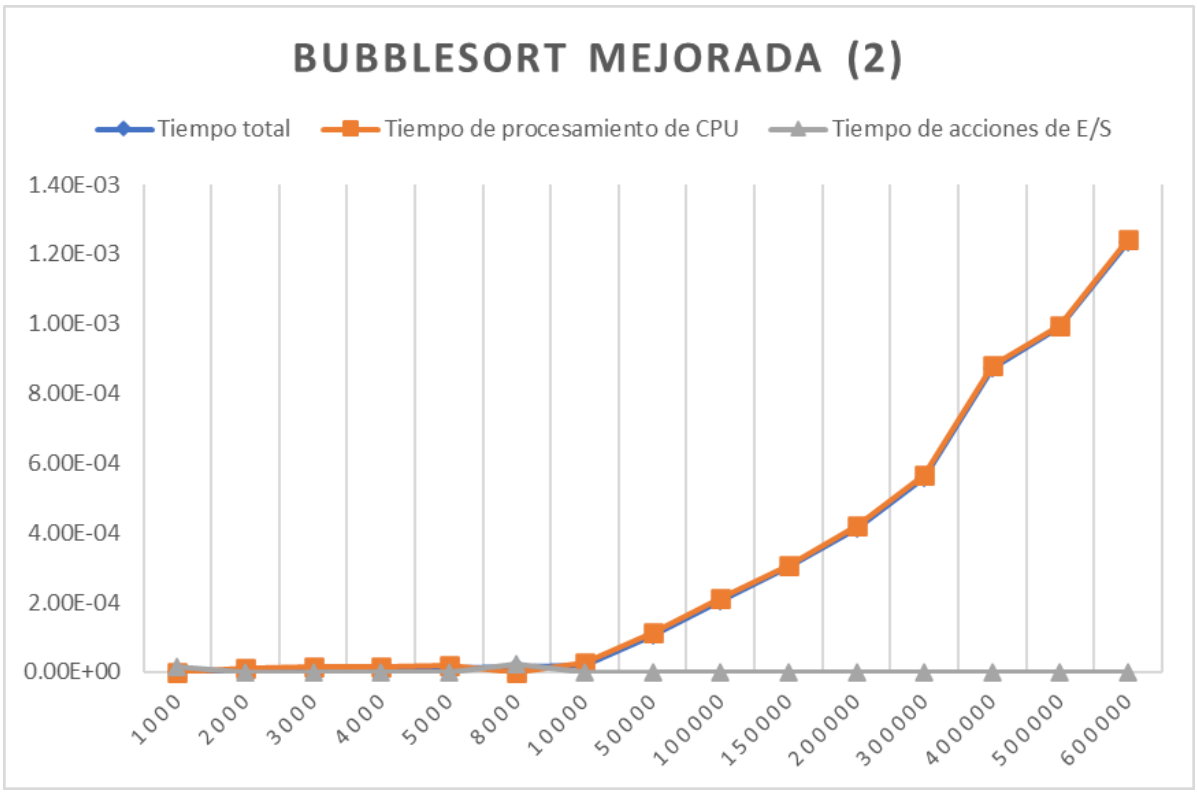
Código 1 Bubble Sort simple



Grafica tiempo de ejecucion 2 Bubble Sort Optimizada 1

```
void BurbujaO(int A[], int n) {
    int i, j, aux;
    for (i = 0; i <= n - 2; i++) {
        for (j = 0; j <= (n - 2)-i; j++) {
            if (A[j] > A[j + 1]) {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
}
```

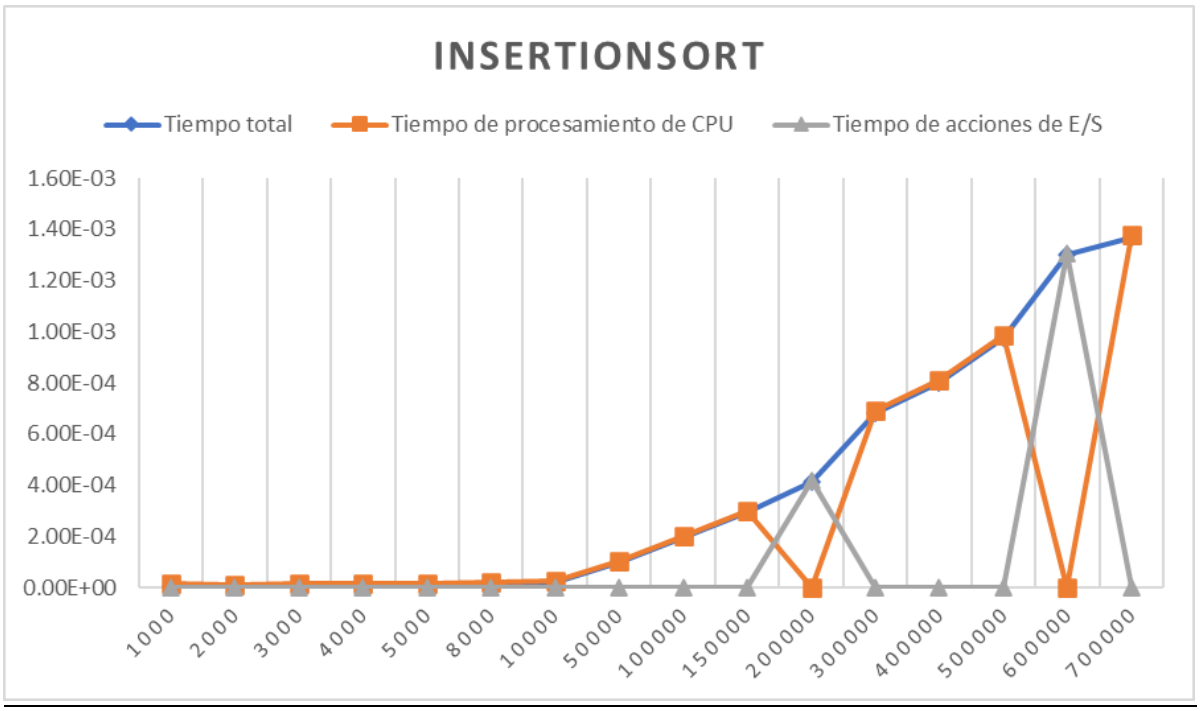
Código 2 Bubble Sort Optimizado 1



Grafica tiempo de ejecucion 3 Bubble Sort Optimizada 2

```c
void BurbujaOpti(int A[], int n) {
    int aux,j;
    int i=0;
    int cambios = 1;
    while (i<=n-2 && cambios != 0) {
        cambios = 0;
        for (j = 0; j <= (n - 2)-i; j++) {
            if (A[j] < A[j + 1]) {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
                cambios = 1;
            }
        }
        i=i+1;
    }
}
```

Código 3 Bubble Sort Optimizado 2

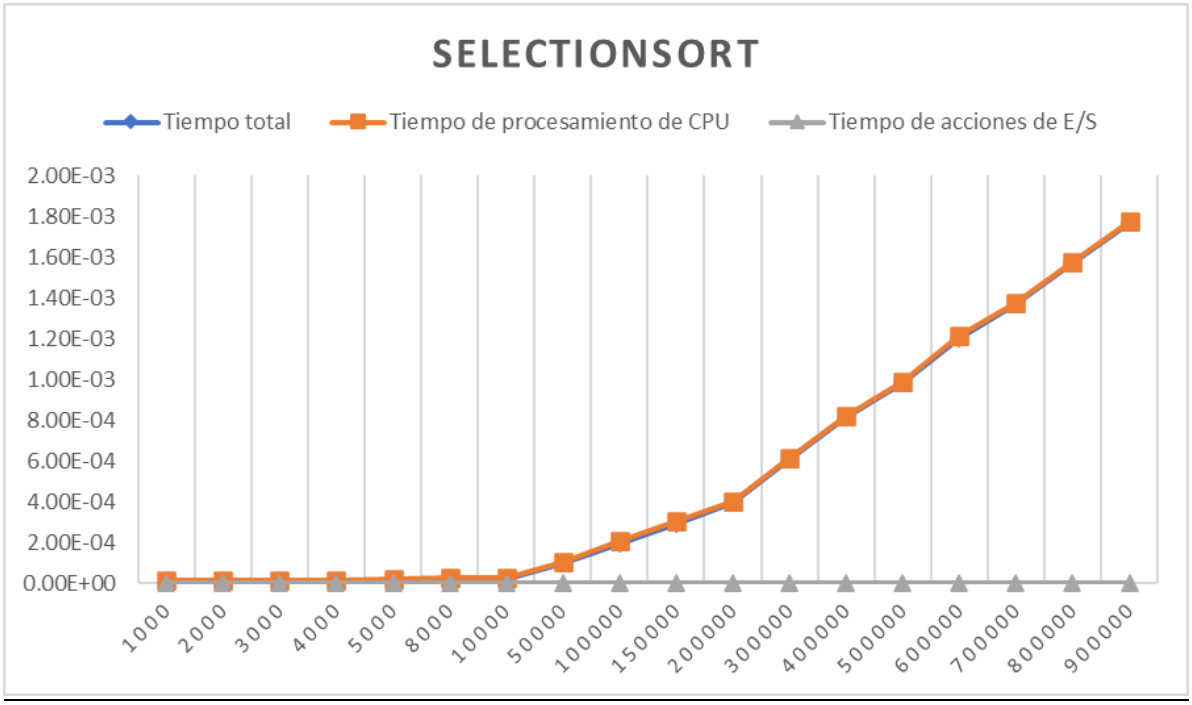


Grafica tiempo de ejecucion 4 Insertion Sort

```
for(i = 1; i < N; i++){
        clave = *(arreglo + i);
        j = i-1;

        while((j >= 0) && *(arreglo+j) > clave){
            *(arreglo + j + 1) = *(arreglo + j);
            j--;
        }
        *(arreglo + j + 1) = clave;
    }
```

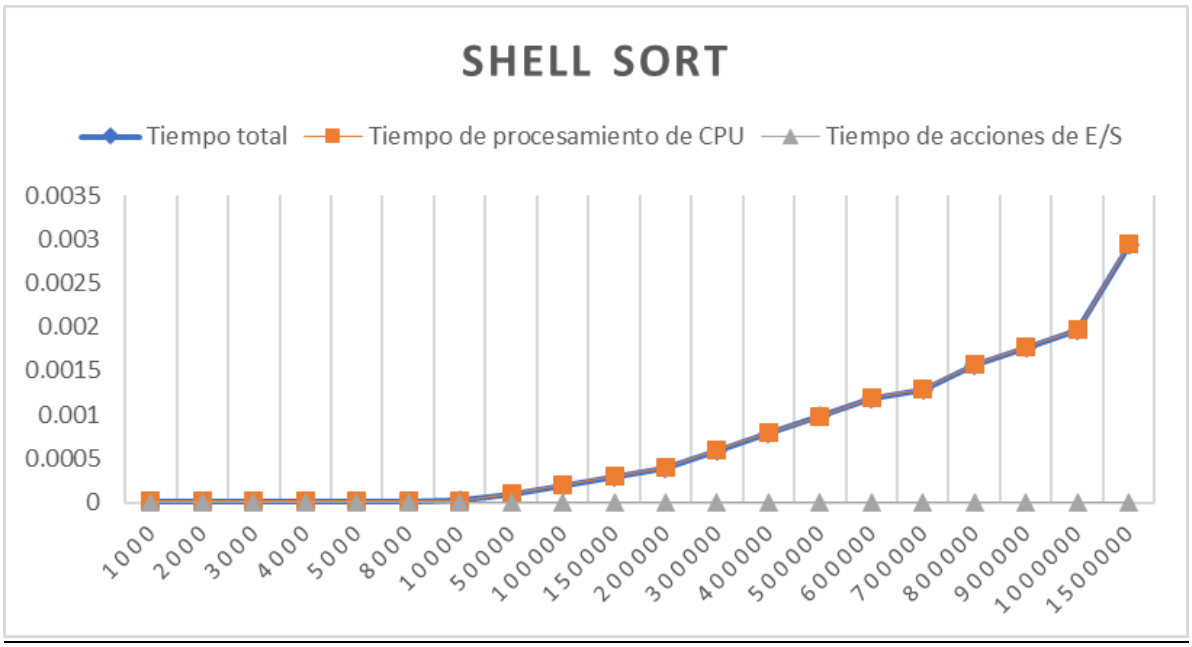
Código 4 Insertion Sort



Grafica tiempo de ejecucion 5 Selection Sort

```
void selectSort(int array[], int tam){
    int i, j, aux, minimo;
    for (i = 0; i < tam; i++) {
        minimo = i;
            for(j = i+1; j < tam; j++) {
                if(array[j] < array[minimo]) {
                    minimo = j;
                }
            }
    aux = array[i];
    array[i] = array[minimo];
    array[minimo] = aux;
    }
}
```

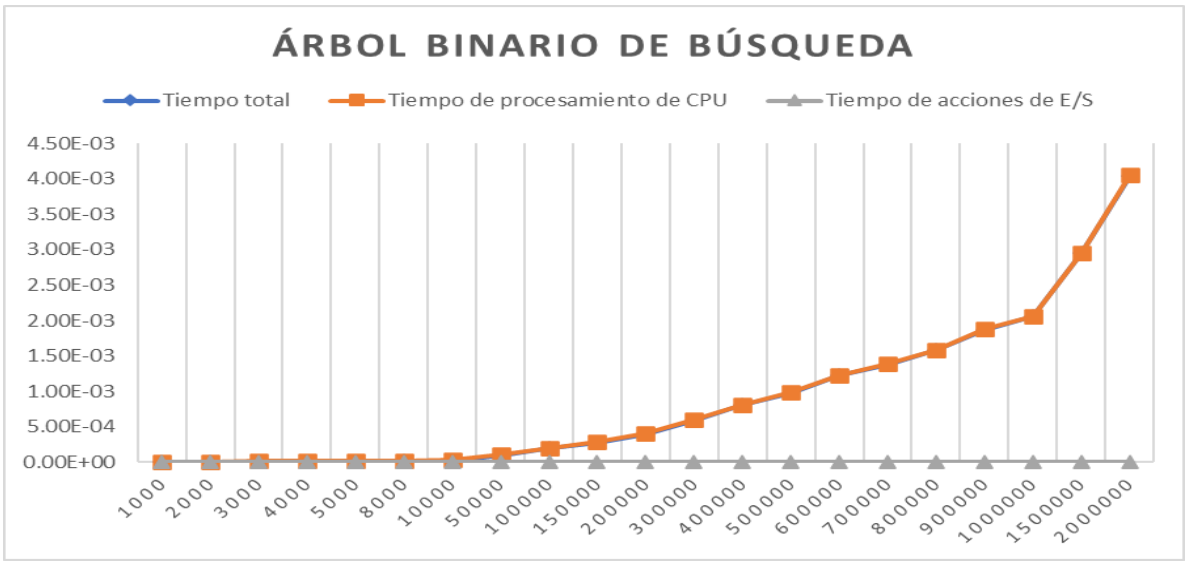
Código 5 Selection Sort



Grafica tiempo de ejecucion 6 Shell Sort

```c
void shellSort (int *a, int n) {
    int intervalo, i, j, k;
    intervalo = n / 2;
    while (intervalo > 0) {
        for (i = intervalo; i < n; i++) {
            j = i - intervalo;
            while (j >= 0) {
                k = j + intervalo;
                if (a[j] <= a[k])
                    j = -1;
                else {
                    int auxiliar = a[j];
                    a[j] = a[k];
                    a[k] = auxiliar;
                    j -= intervalo;
                }
            }
        }
        intervalo /= 2;
    }
}
```

Código 6 Shell Sort



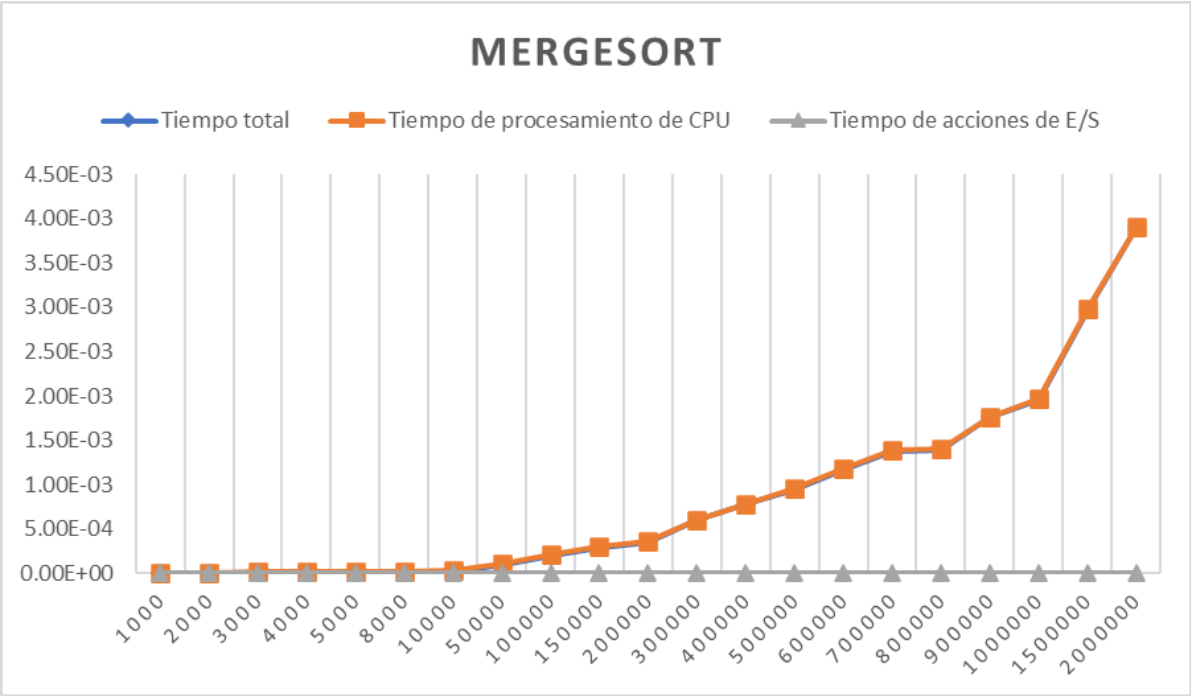
Grafica tiempo de ejecucion 7 Tree Sort

```c
void insertar(struct Nodo *nodo, int dato) {

    if (dato > nodo->dato) {
        if (nodo->derecha == NULL) {
            nodo->derecha = nuevoNodo(dato);

        } else {
            insertar(nodo->derecha, dato);
        }
    } else {
        if (nodo->izquierda == NULL) {
            nodo->izquierda = nuevoNodo(dato);
        } else {
            insertar(nodo->izquierda, dato);
        }
    }
}
```

Código 7 Tree Sort

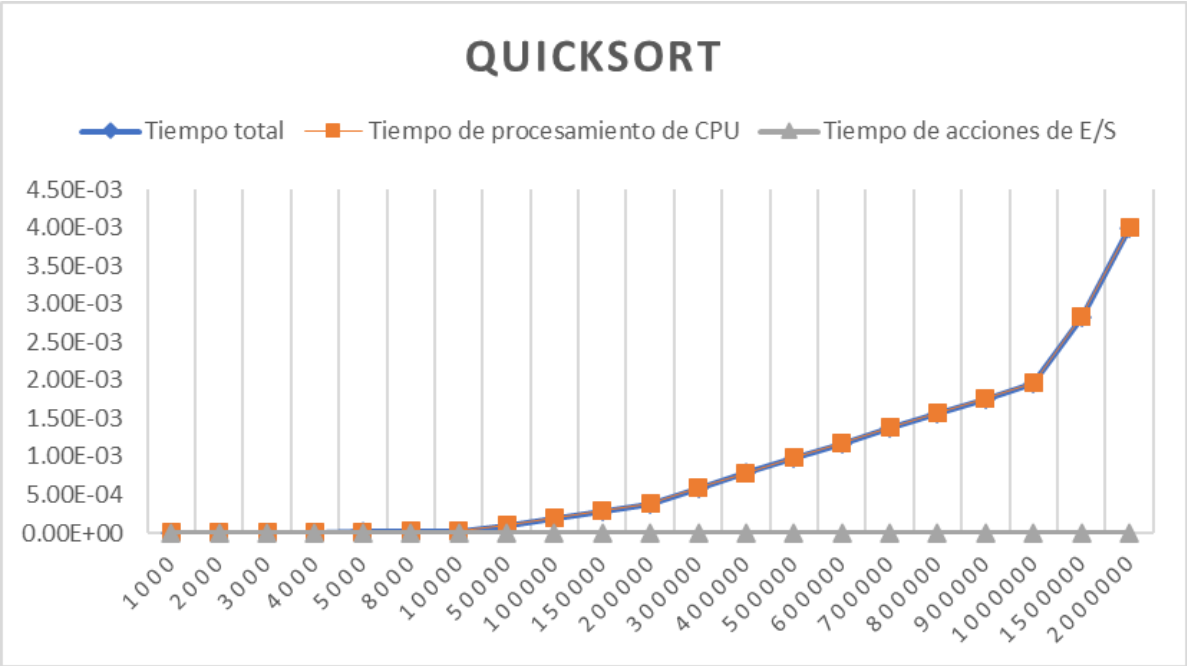


Grafica tiempo de ejecucion 8 Merge Sort

```
void mergeSort(int *a, int inicio, int final){
    if(inicio < final){
        int central = (inicio + final) / 2;
        mergeSort(a, inicio, central);
        mergeSort(a, central+1, final);
        mezcla(a, inicio, central, final
}
void mezcla(int *a, int inicio, int centro, int final){
    int i, j, k;
    int elementosIzq = centro - inicio + 1;
    int elementosDer = final - centro;
    int *izquierdo = (int*) malloc(elementosIzq*sizeof(int));
    int *derecho = (int*) malloc(elementosDer*sizeof(int));
    for(i = 0; i < elementosIzq; i++)
        izquierdo[i] = a[inicio+i];
    for(j = 0; j < elementosDer; j++)
        derecho[j] = a[centro + 1 + j];
    i = j = 0;    k = inicio;
    while(i < elementosIzq && j < elementosDer){
        if(izquierdo[i] <= derecho [j]){
            a[k] = izquierdo[i];  i++;
        }   else{
            a[k] = derecho[j]; j++;
        }
        k++;
```

```
    }
    while(j < elementosDer){
        a[k] = derecho[j];
        j++; k++;
    }
    while(i < elementosIzq){
        a[k] = izquierdo[i];
        i++; k++;
    }
    free(izquierdo);
    free(derecho);
}
```

Código 8 Merge Sort

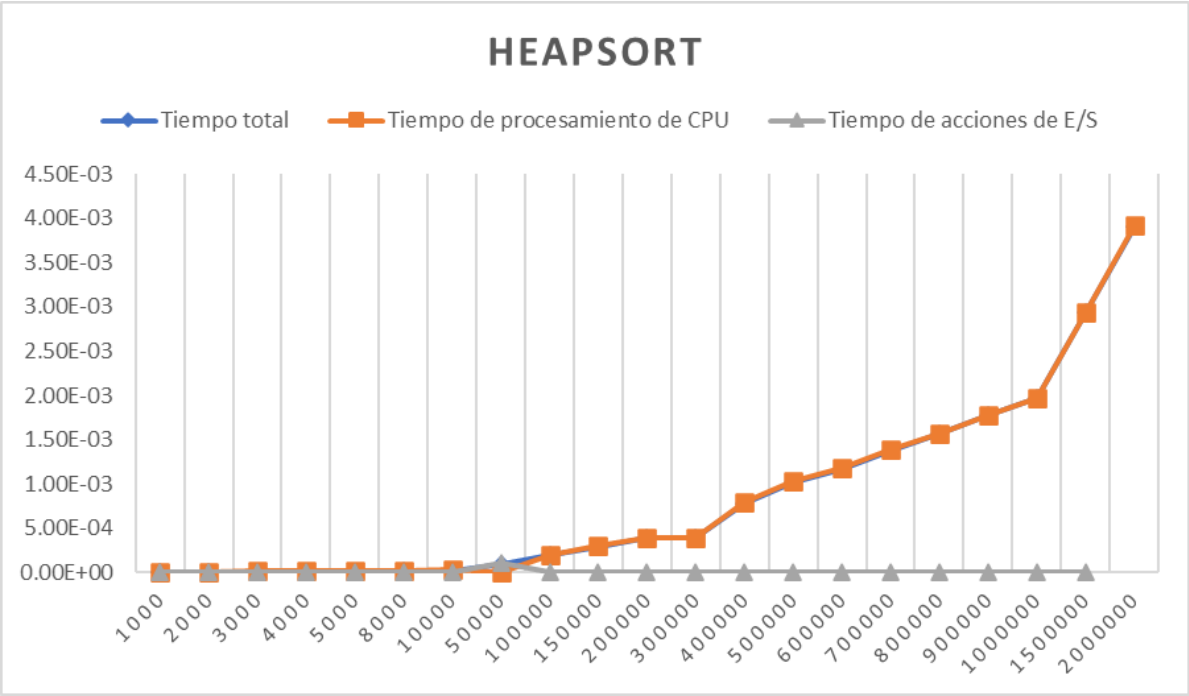


Grafica tiempo de ejecucion 9 Quick Sort

```
void quickSort(int *a, int primero, int ultimo){

    int i = primero, j = ultimo;
    int centro = (primero + ultimo) / 2;
    int pivote = a[centro];
    do {
        while(a[i] < pivote) i++;
        while(a[j] > pivote) j--;
        if(i <= j){
            int auxiliar;
            auxiliar = a[i];
            a[i] = a[j];
            a[j] = auxiliar;
            i++; j--;
        }
    }   while(i <= j);
    if(primero < j)
        quickSort(a, primero, j);
    if(ultimo > i)
        quickSort(a, i, ultimo);
}
```

Código 9 Quick Sort

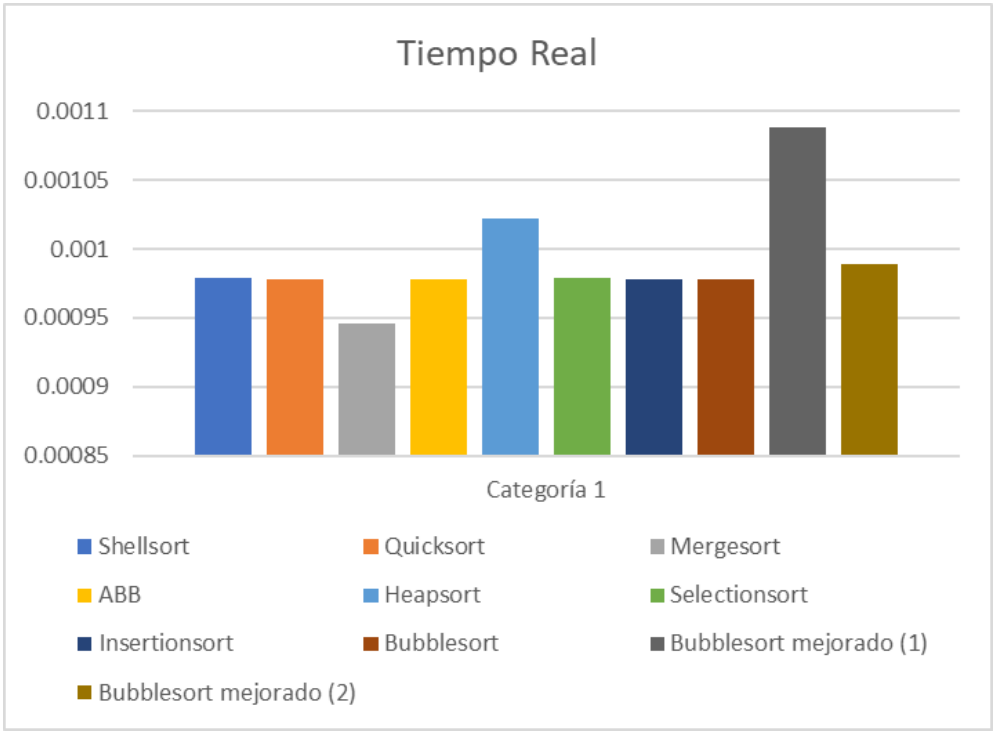


Grafica tiempo de ejecucion 10 Heap Sort

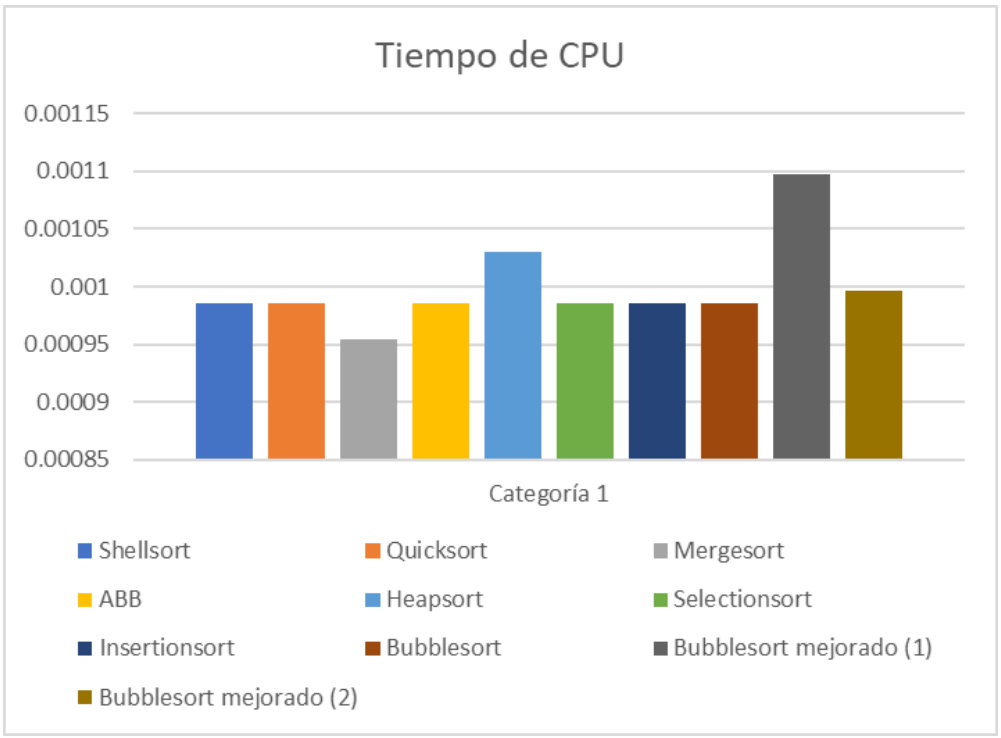
```
void monticulo(int arr[], int N, int i){
    int may = i;
    int izq = obtenerHijoIzquierdo(i);
    int der = obtenerHijoDerecho(i);
    if (izq < N && arr[izq] > arr[may]){
        may = izq;
    }
    if (der < N && arr[der] > arr[may]){
        may = der;
    }
    if (may != i) {
        intercambio(&arr[i], &arr[may]);
        monticulo(arr, N, may);
    }
}
void HeapSort(int A[], int n){
    int i;
    for (i = n / 2 - 1; i >= 0; i--)
        monticulo(A, n, i);
    for (i = n - 1; i >= 0; i--) {
        intercambio(&A[0], &A[i]);
        monticulo(A, i, 0);
    }
}
```

Código 10 Heap Sort

Comparación de tiempo real

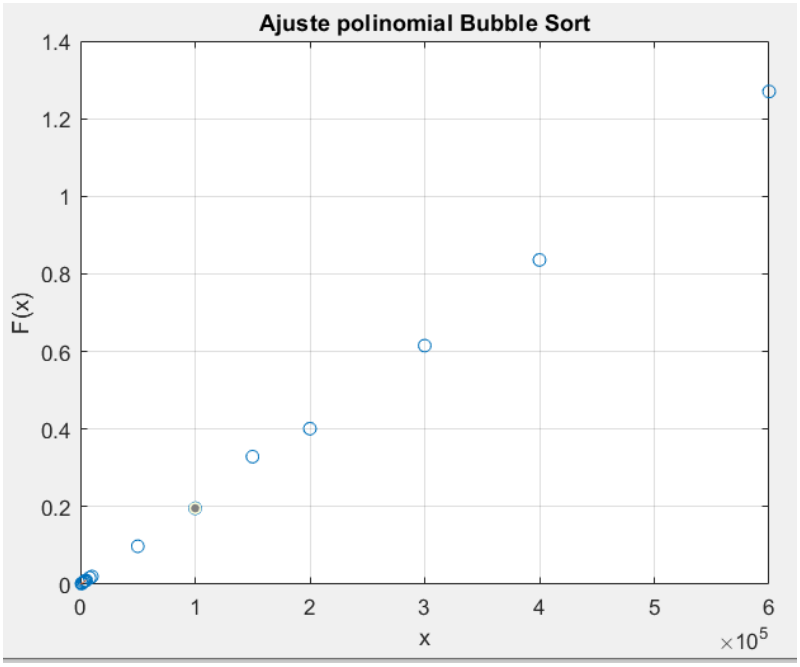


Grafica tiempo real 1

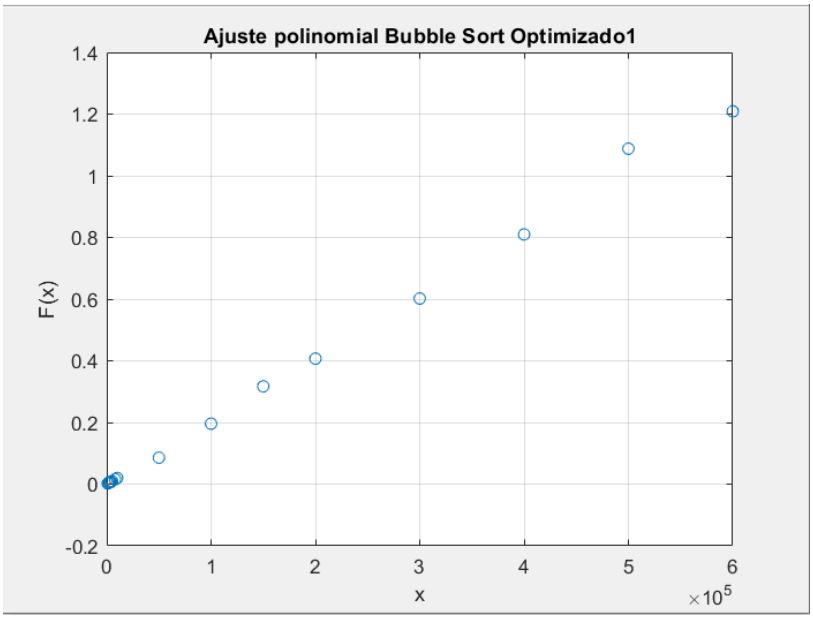


Grafica tiempo real 2

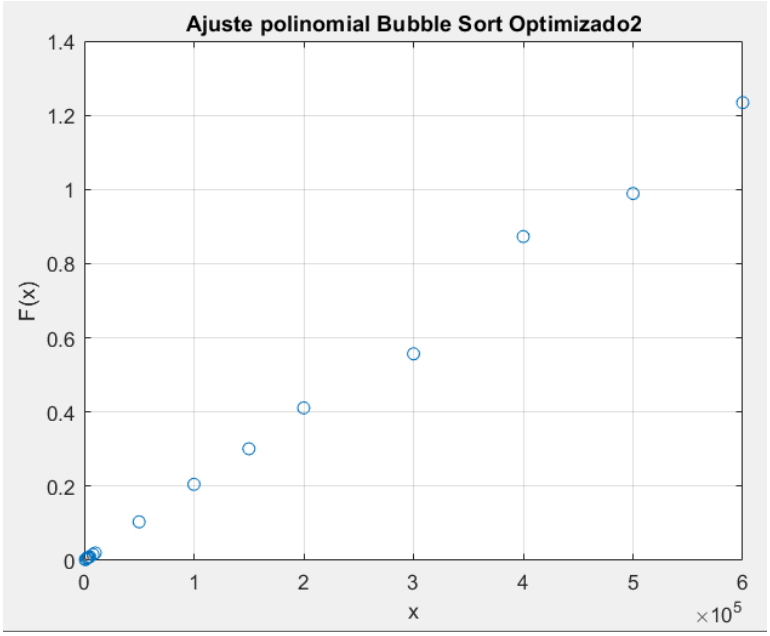
Gráficas de Matlab



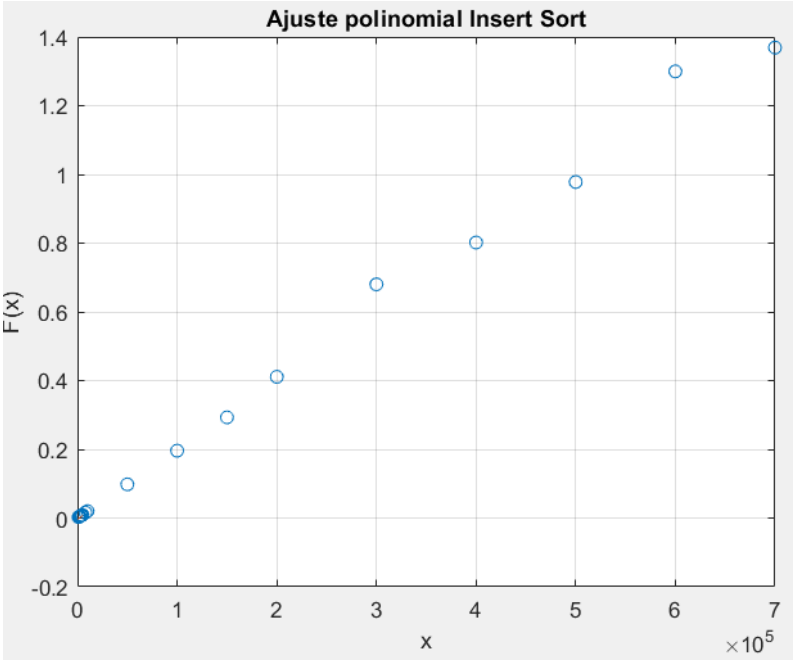
Ajuste polinomial en MatLab 1 Bubble Sort



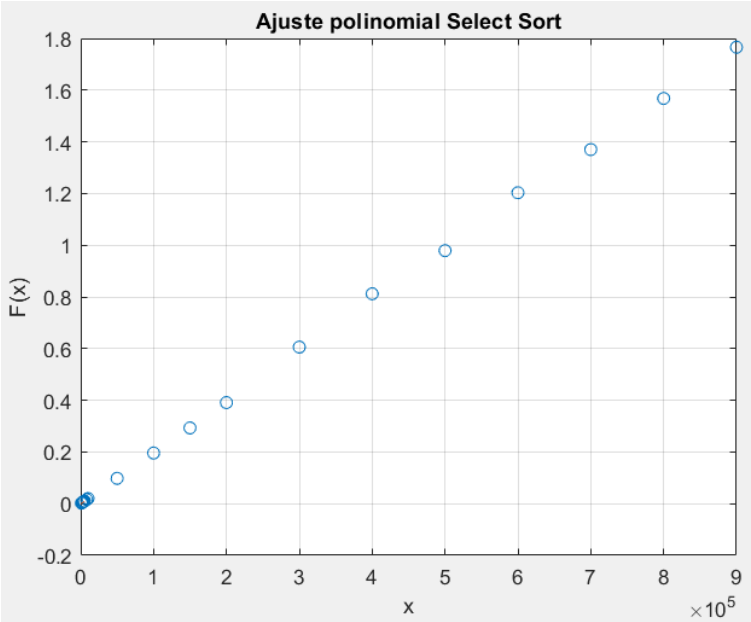
Ajuste polinomial en MatLab 2 Bubble Sort Optimizado 1



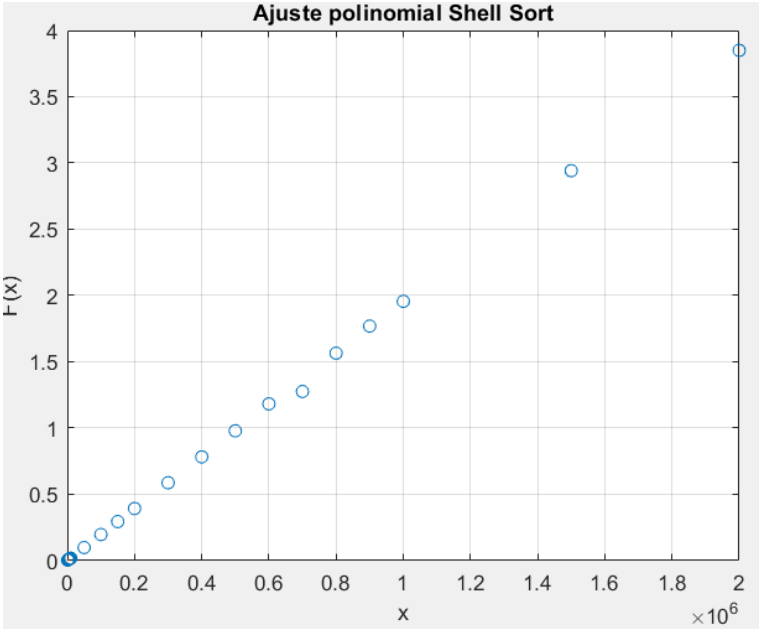
Ajuste polinomial en MatLab 3 Bubble Sort Optimizado 2



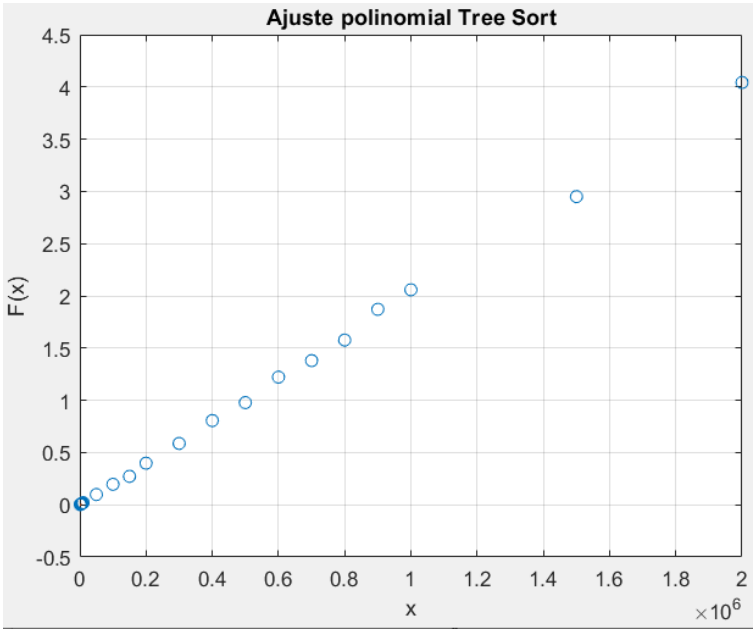
Ajuste polinomial en MatLab 4 Insert Sort



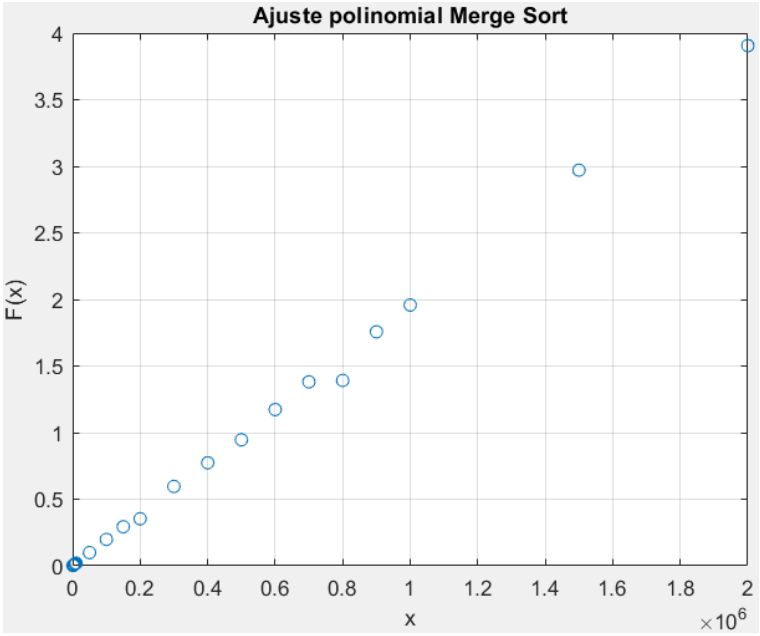
Ajuste polinomial en MatLab 5 Select Sort



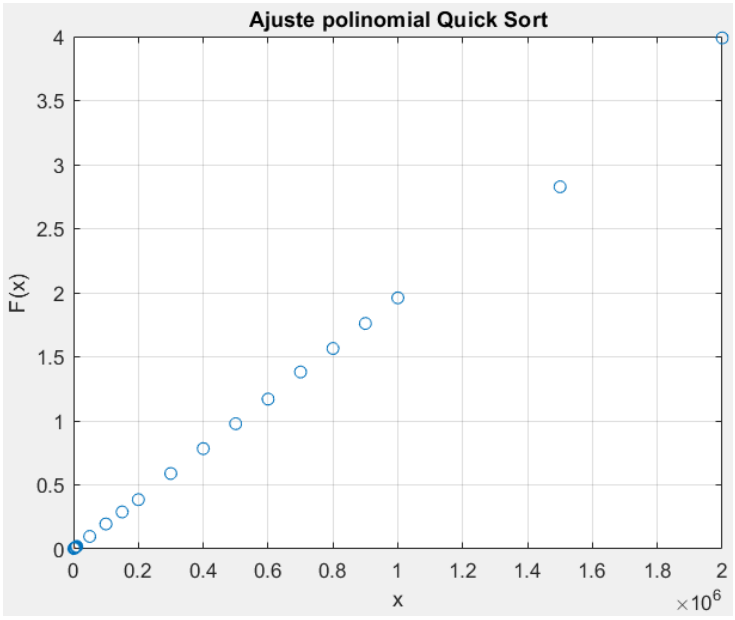
Ajuste polinomial en MatLab 6 Shell Sort



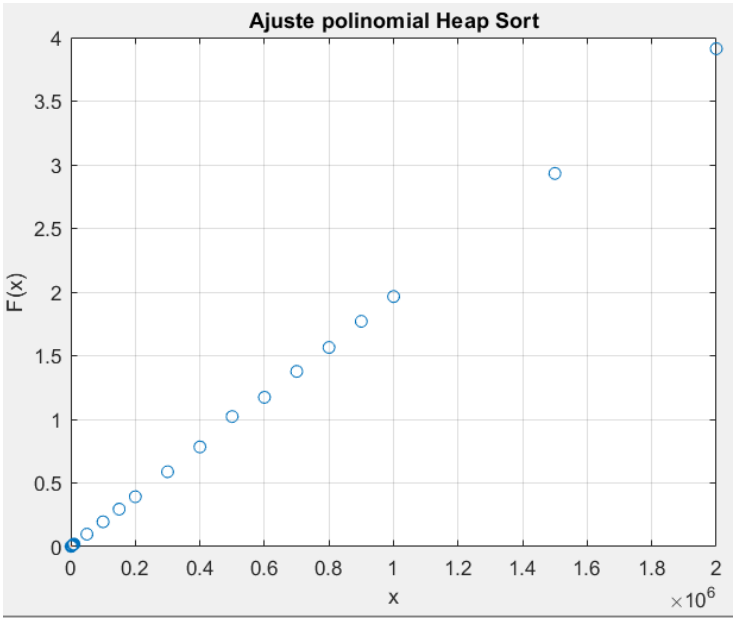
Ajuste polinomial en MatLab 7 Tree Sort



Ajuste polinomial en MatLab 8 Merge Sort



Ajuste polinomial en MatLab 9 Quick Sort



Ajuste polinomial en MatLab 10 Heap Sort

Pruebas con cada algoritmo

<table>
<tr><th>Algoritmo</th><th>Función Complejidad</th><th>Resultados</th></tr>
<tr><td>Bubble Sort</td><td>$n^2 - 2n + 1$</td><td>
<table>
<tr><th>Datos</th><th>Resultado algebraico</th><th>Resultado de ejecución</th></tr>
<tr><td>3,000,000</td><td>899.999e7</td><td>~</td></tr>
<tr><td>4,000,000</td><td>1.5999992e13</td><td>~</td></tr>
<tr><td>5,000,000</td><td>1.5999992e13</td><td>~</td></tr>
<tr><td>6,000,000</td><td>3.5999988e13</td><td>~</td></tr>
<tr><td>10,000,000</td><td>9.999998e13</td><td>~</td></tr>
</table>
</td></tr>
<tr><td>Bubble Sort Optimizado 1</td><td>$\frac{(n)^2}{2} - \frac{n}{2}$</td><td>
<table>
<tr><th>Datos</th><th>Resultado algebraico</th><th>Resultado de ejecución</th></tr>
<tr><td>3,000,000</td><td>4.4999985e+12</td><td>~</td></tr>
<tr><td>4,000,000</td><td>7.999998e+12</td><td>~</td></tr>
<tr><td>5,000,000</td><td>1.2499998e+13</td><td>~</td></tr>
<tr><td>6,000,000</td><td>1.7999997e+13</td><td>~</td></tr>
<tr><td>10,000,000</td><td>4.9999995e+13</td><td>~</td></tr>
</table>
</td></tr>
<tr><td>Bubble Sort Optimizado 2</td><td>$\frac{((n-1)\cdot n)}{2}$</td><td>
<table>
<tr><th>Datos</th><th>Resultado algebraico</th><th>Resultado de ejecución</th></tr>
<tr><td>3,000,000</td><td>4.4999985e+12</td><td>~</td></tr>
<tr><td>4,000,000</td><td>7.999998e+12</td><td>~</td></tr>
<tr><td>5,000,000</td><td>1.2499998e+13</td><td>~</td></tr>
<tr><td>6,000,000</td><td>1.7999997e+13</td><td>~</td></tr>
<tr><td>10,000,000</td><td>4.9999995e+13</td><td>~</td></tr>
</table>
</td></tr>
<tr><td>Insertion Sort</td><td>$\frac{((n-1)\cdot n)}{2}$</td><td>
<table>
<tr><th>Datos</th><th>Resultado algebraico</th><th>Resultado de ejecución</th></tr>
<tr><td>3,000,000</td><td>9e+12</td><td>~</td></tr>
<tr><td>4,000,000</td><td>1.6e+13</td><td>~</td></tr>
<tr><td>5,000,000</td><td>2.5e+13</td><td>~</td></tr>
<tr><td>6,000,000</td><td>3.6e+13</td><td>~</td></tr>
<tr><td>10,000,000</td><td>1e+14</td><td>~</td></tr>
</table>
</td></tr>
<tr><td>Selection Sort</td><td>$\frac{((n-1)\cdot n)}{2}$</td><td>
<table>
<tr><th>Datos</th><th>Resultado algebraico</th><th>Resultado de ejecución</th></tr>
<tr><td>3,000,000</td><td>4.4999985e+12</td><td>~</td></tr>
<tr><td>4,000,000</td><td>7.999998e+12</td><td>~</td></tr>
<tr><td>5,000,000</td><td>1.2499998e+13</td><td>~</td></tr>
<tr><td>6,000,000</td><td>1.7999997e+13</td><td>~</td></tr>
<tr><td>10,000,000</td><td>4.9999995e+13</td><td>~</td></tr>
</table>
</td></tr>
</table>

Shell Sort	$n \ln(n)$	

Datos	Resultado algebraico	Resultado de ejecución
3,000,000	44742368.539	0.0058691 s
4,000,000	60,807,219.67	0.0072281 s
5,000,000	77,124,742.35	0.0097589 s
6,000,000	93,643,620.16	0.0113060 s
10,000,000	161,180,956.5	0.0194568 s

Tree Sort	$n \ln(n)$	

Datos	Resultado algebraico	Resultado de ejecución
3,000,000	44742368.539	0.0059199 s
4,000,000	60,807,219.67	0.0077540 s
5,000,000	77,124,742.35	0.0097711 s
6,000,000	93,643,620.16	0.0116589 s
10,000,000	161,180,956.5	0.0196599 s

Merge Sort	$n \ln(n)$	

Datos	Resultado algebraico	Resultado de ejecución
3,000,000	44742368.539	0.0052700 s
4,000,000	60,807,219.67	0.0074110 s
5,000,000	77,124,742.35	0.0098021 s
6,000,000	93,643,620.16	0.0117118 s
10,000,000	161,180,956.5	0.0196321 s

Quick Sort	$n \ln(n)$	

Datos	Resultado algebraico	Resultado de ejecución
3,000,000	44742368.539	0.0058469 s
4,000,000	60,807,219.67	0.0078217 s
5,000,000	77,124,742.35	0.0093779 s
6,000,000	93,643,620.16	0.0112969 s
10,000,000	161,180,956.5	0.0190160 s

Heap Sort	$n \ln(n)$	

Datos	Resultado algebraico	Resultado de ejecución
3,000,000	44742368.539	0.0058720 s
4,000,000	60,807,219.67	0.0078217 s
5,000,000	77,124,742.35	0.0097970 s
6,000,000	93,643,620.16	0.0116939 s
10,000,000	161,180,956.5	0.0189449 s

Cuestionario

¿Cuál de los 10 algoritmos es más fácil de implementar?

El algoritmo de ordenamiento burbuja es más fácil de codificar ya que solo consiste en dos bucles, uno interior y uno exterior. En el peor de los casos este algoritmo tiene complejidad $O(n^2)$. Es el algoritmo menos eficiente cuando la cantidad de datos por ordenar aumenta.

¿Cuál de los 10 algoritmos es más difícil de implementar?

El Árbol Binario de Búsqueda en cuestión de memoria, porque tenemos que hacer más uso de memoria con las estructuras de tipo nodo. Una vez que se tiene el ABB, se realiza el recorrido inorden asignando cada valor a un arreglo dinámico donde se encuentran los valores ordenados una vez que termina el recorrido.

También a la hora de codificar, ya ABB es una estructura de datos donde se implementa memoria dinámica, apuntadores y nodos.

¿Cuál algoritmo tiene menor complejidad temporal?

El ordenamiento QuickSort,l nombre que recibe este algoritmo se debe a su gran velocidad que tiene para ordenar una inmensa cantidad de datos. Es un algoritmo bastante eficiente. Pero en el peor de los casos, este algoritmo puede ser $O(n^2)$.

¿Cuál algoritmo tiene mayor complejidad temporal?

El ordenamiento burbuja, es el algoritmo que demora más tiempo en ordenar un arreglo de números. Es eficiente cuando el tamaño del arreglo es pequeño, pero mientras la cantidad de datos aumenta, el tiempo de ejecución también.

El problema de este algoritmo es la cantidad de comparaciones que tiene que hacer para ordenar los datos. A pesar de ser el más fácil de implementar, es el menos eficiente para ordenar una grande cantidad de datos.

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

Selection Sort, ya que solo utiliza una cantidad constante de memoria.

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

MergeSort, ya que necesita crear arreglos auxiliares para ir dividiendo y juntando las partes del arreglo original.

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

Si, realmente uno de los resultados que esperábamos era ver la gran eficiencia que tiene el algoritmo QuickSort, además de poder ver cómo es su comportamiento conforme la cantidad de datos aumenta.

También deducimos que el ordenamiento burbuja sería el más tardado en ordenar todos los datos.

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

Sí, utilizamos la última versión de Ubuntu y las especificaciones del sistema están al inicio de este reporte.

¿Facilito las pruebas mediante scripts u otras automatizaciones? ¿Como lo hizo?

Si, hicimos uso del lenguaje bash, que es la shell de Linux para automatizar las tareas. Hicimos un script que nos permite compilar cada uno de los códigos, otorgando el tamaño del arreglo para poder hacer las pruebas más fácilmente y no demorar tanto tiempo en estar compilando y ejecutando cada programa.

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

Saber automatizar las tareas con ayuda del lenguaje de la shell de Linux o en su defecto, en la shell de Windows, ya que realmente ayuda mucho en cuestión de tiempo poder hacer uso de scripts a la hora de realizar pruebas con los programas.
Tenes más equipos para poder realizar diversas pruebas de distintos algoritmos a la vez y poder avanzar más rápido, tener un equipo bien organizado y hacerlo con tiempo.

Anexo

```c
/*
VILLEGAS GOMEZ ALEJANDRA
ESCUELA SUPERIOR DE COMPUTO
MATERIA: Analisis y diseño de algoritmos
Algoritmo BubbleSort Simple
FECHA: 07/03/2023
*/

#include <stdio.h>          //libreria estandar de entrada y salida
#include <stdlib.h>         //libreria estandar  (para la memoria dinamica)

/*El ordenamiento burbuja simple, consta de comparar cada
elemento con el siguiente, intercambiando sus posiciones si
no estan ordenados correctamente;
iterando el numero de elementos a ordenar*/

void BurbujaS(int *A, int n);           //Prototipo de la funcion en que se
encuentra nuestro algoritmo

int main(int argc, char *argv[]) {
    int i, *A;                              //Declaramos la variable para
iterar y un apuntador que se utilizara para almacenar un arreglo de enteros
    if(argc!=2)
        exit (1);                           //Finalizar por error por
entrada no valida
    int n=atoi(argv[1]);                    //Convertir segundo argumento en
entero

    A = malloc(n*sizeof(int));              //Haciendo un arreglo dinamico
de N elementos y asignando el puntero a la variable A

    for(i=0;i<n;i++){
        scanf("%d",&A[i]);                  //Lee los valores de entrada y
los asigna al arreglo dinamico A
    }
    BurbujaS(A, n);                         //Llevar a cabo el ordenamiento
llamando a la funcion
    printf("BubbleSort finalizado");
    return 0;
}

void BurbujaS(int A[], int n) {                 //Funcion con A como arreglo
y n como posicion en el arreglo
```

```
    int i, j, aux;
    for (i = 0; i <= n - 2; i++) {              //Analiza desde la posicion
0 a la n-1 del arreglo
        for (j = 0; j <= n - 2; j++) {          //Analiza los elementos
adyacentes de 0 a n-1
            if (A[j] > A[j + 1]) {              //Si el elem analizado es
mayor al sig
                aux = A[j];                     //Guardamos el valor en el
aux
                A[j] = A[j + 1];                //Se realiza el cambio de
posicion, dando el valor del sig a A[j]
                A[j + 1] = aux;                 //Asignamos el valor
anterior guardado de A[j] a A[j+1]
            }
        }
    }
}
```

Código fuente 1. Bubble Sort

```
/*
VILLEGAS GOMEZ ALEJANDRA
ESCUELA SUPERIOR DE COMPUTO
MATERIA: Analisis y diseño de algoritmos
Algoritmo BubbleSort Optimizado 1
FECHA: 07/03/2023
*/

#include <stdio.h>          //libreria estandar de entrada y salida
#include <stdlib.h>         //libreria estandar  (para la memoria dinamica)

/*El ordenamiento burbuja, consta de comparar cada
elemento con el siguiente, intercambiando sus posiciones si
no estan ordenados correctamente;
Se observa que al final de cada iteracion el elemento mayor queda
correctamente posicionado, por lo que no es necesario volver a compararlo,
reduciendo asi las iteraciones */

void BurbujaO(int *A, int n);           //Prototipo de la funcion en que se
encuentra nuestro algoritmo

int main(int argc, char *argv[]) {
    int i, *A;                              //Declaramos la variable para
iterar y un apuntador que se utilizara para almacenar un arreglo de enteros
```

```
    if(argc!=2)
        exit (1);                           //Finalizar por error por
entrada no valida
    int n=atoi(argv[1]);                    //Convertir segundo argumento en
entero


    A = malloc(n*sizeof(int));              //Haciendo un arreglo dinamico
de N elementos y asignando el puntero a la variable A

    for(i=0;i<n;i++){
        scanf("%d",&A[i]);                  //Lee los valores de entrada y
los asigna al arreglo dinamico A
    }
    BurbujaO(A, n);                         //Llevar a cabo el ordenamiento
llamando a la funcion
    printf("Bubble1 finalizado");
    return 0;
}

void BurbujaO(int A[], int n) {                 //Funcion con A como arreglo
y n como posicion en el arreglo
    int i, j, aux;
    for (i = 0; i <= n - 2; i++) {              //Analiza desde la posicion
0 a la n-1 del arreglo
        for (j = 0; j <= (n - 2)-i; j++) {      //En esta primera
optimizacion se resta i (el numero de elementos que ya han sido ordenados en
las pasadas anteriores)
            if (A[j] > A[j + 1]) {              //Si el elem analizado es
mayor al sig
                aux = A[j];                     //Guardamos el valor en el
aux
                A[j] = A[j + 1];                //Se realiza el cambio de
posicion, dando el valor del sig a A[j]
                A[j + 1] = aux;                 //Asignamos el valor
anterior guardado de A[j] a A[j+1]
            }
        }
    }
}
```

Código fuente 2. Bubble Sort Optimizada 1

```
/*
VILLEGAS GOMEZ ALEJANDRA
ESCUELA SUPERIOR DE COMPUTO
```

```c
MATERIA: Analisis y diseño de algoritmos
Algoritmo BubbleSort Optimizado 2
FECHA: 07/03/2023
*/

#include <stdio.h>          //libreria estandar de entrada y salida
#include <stdlib.h>         //libreria estandar  (para la memoria dinamica)

/*El ordenamiento burbuja, consta de comparar cada elemento con el
siguiente,
intercambiando sus posiciones si no estan ordenados correctamente;
Se observa que al final de cada iteracion el elemento mayor queda
correctamente posicionado,
por lo que no es necesario volver a compararlo, reduciendo asi las
iteraciones.
Ademas, en esta segunda optimizacion no se ejecutara el bucle for hasta el
final en cada pasada,
pues se detiene la pasada tan pronto como no se detecten cambios en el
arreglo */

void BurbujaOpti(int *A, int n);            //Prototipo de la funcion en que
se encuentra nuestro algoritmo

int main(int argc, char *argv[]) {
    int i, *A;                              //Declaramos la variable para
iterar y un apuntador que se utilizara para almacenar un arreglo de enteros
    if(argc!=2)
        exit (1);                           //Finalizar por error por
entrada no valida
    int n=atoi(argv[1]);                    //Convertir segundo argumento en
entero


    A = malloc(n*sizeof(int));              //Haciendo un arreglo dinamico
de N elementos y asignando el puntero a la variable A

    for(i=0;i<n;i++){
        scanf("%d",&A[i]);                  //Lee los valores de entrada y
los asigna al arreglo dinamico A
    }
    BurbujaOpti(A, n);
    printf("Bubble2 finalizado");
    return 0;
}
```

```c
void BurbujaOpti(int A[], int n) {              //Funcion con A como arreglo
y n como posicion en el arreglo
    int aux,j;
    int i=0;
    int cambios = 1;                            //Bandera, para controlar si
se han realizado cambios en el arreglo durante la pasada.
    while (i<=n-2 && cambios != 0) {            //continuara mientras se
sigan realizando cambios en el arreglo
        cambios = 0;                            //cambios a 0 pues no se han
realizado cambios durante esa pasada
        for (j = 0; j <= (n - 2)-i; j++) {      //Se resta i a la llegada
(el numero de elementos que ya han sido ordenados en las pasadas anteriores)
            if (A[j] < A[j + 1]) {              //Si el elem analizado es
mayor al sig
                aux = A[j];                     //Guardar el valor en el
auxiliar
                A[j] = A[j + 1];                //Se realiza el cambio de
posicion, dando el valor del sig a A[j]
                A[j + 1] = aux;                 //Asignamos el valor
anterior guardado de A[j] a A[j+1]
                cambios = 1;                    // bandera a 1 para indicar
que se ha realizado al menos un cambio durante la pasada.
            }
        }
        i=i+1;                                  //Asegura que no se revisen
los elementos ya ordenados en la parte superior del arreglo durante las
pasadas posteriores
    }                                           //Si no se realizaron
cambios en el arreglo durante una pasada, el bucle while se detiene y la
función termina
}
```

Código fuente 3. Bubble Sort Optimizado 2

```c
/*
MÁNICA QUINTERO MARCO ANTONIO
ESCUELA SUPERIOR DE COMPUTO
MATERIA: Analisis y diseño de algoritmos
Algoritmo Insert Sort
FECHA: 14/03/2023
*/

#include <stdio.h> // Incluye la librería estándar de entrada/salida
#include <stdlib.h> // Incluye la librería estándar de funciones generales

void Insercion(int *A, int n); // Declaración de la función Insercion
```

```c
int main(int argc, char *argv[]) { // Función principal que recibe argumentos desde la línea de comandos

    if(argc!=2) // Verifica que se haya pasado un argumento
        exit (1);

    int n=atoi(argv[1]); // Convierte el argumento pasado como una cadena de caracteres a un entero

    int *A; // Declaración del puntero A
    int i;

    A = malloc(n*sizeof(int)); // Reserva memoria dinámicamente para un arreglo de enteros de tamaño n

    for(i=0;i<n;i++){ // Ciclo que pide al usuario que ingrese los n elementos del arreglo
        scanf("%d",&A[i]);
    }

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

    //for (i = 0; i < n; i++) { // Ciclo que imprime los elementos del arreglo ya ordenados
        //printf("%d\n ", A[i]);
   // }

    printf("InsertionSort finalizado");
    free(A); // Libera la memoria reservada para el arreglo
    return 0; // Retorna 0 si todo fue exitoso
}

void Insercion(int *A, int n) { // Función que ordena el arreglo por el método de inserción
    int i, j, temp;
    for (i = 0; i < n; i++) { // Ciclo que recorre el arreglo
        j = i;
        temp = A[i];
        while ((j > 0) && (temp < A[j - 1])) { // Ciclo que intercambia el elemento actual con los elementos anteriores que sean mayores a él
            A[j] = A[j - 1];
            j--;
        }
```

```
        A[j] = temp; // Coloca el elemento actual en la posición correcta en
el arreglo
    }
}
```

Código fuente 4. Insertion Sort

```
// SERGE EDUARDO MARTÍNEZ RAMÍREZ
// ESCUELA SUPERIOR DE CÓMPUTO
// MATERIA: Análisis y diseño de algoritmos
// FECHA: 13/03/2023

//Librerias

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

//Prototipo de funciones

void mostrarArreglo (int a[], int tam);
void selectSort(int array[], int tam);

/*
Funcion principal o main, recibe como parametro de entrada el numero de
datos a ordenar y una cadena con ellos, utiliza un arreglo dinamico para
guardar los
datos si la memoria fue asignada correctamente. Procede a llamar
a la funcion selectSort donde se ejecutara el algoritmo de ordenamiento y
por ultimo nos mostrara los datos ordenados
*/

int main(int n, char *cadena[]) {
    if(n !=2)
        exit (1);
    int nElemento=atoi(cadena[1]);
    int *arreglo;
    arreglo = malloc(nElemento *sizeof(int));
        if (arreglo == NULL) {
            printf("Sin memoria");
            exit(1);
        } else{
            for(int i=0;i<nElemento;i++){
                scanf("%d", &arreglo[i]);
            }
            selectSort(arreglo, nElemento);
```

```
        }
    mostrarArreglo(arreglo, nElemento);
    return 0;
}

/*
La funcion selectSort consiste en buscar el mínimo elemento de la lista e
intercambiarlo
con el primer elemento, busca de nuevo el elemento mínimo en el resto de la
lista y
lo intercambia con el segundo.
*/

void selectSort(int array[], int tam){
    int i, j, aux, minimo;
    for (i = 0; i < tam; i++) {
        minimo = i;
            for(j = i+1; j < tam; j++) {
                if(array[j] < array[minimo]) {
                    minimo = j;
                }
            }
    aux = array[i];
    array[i] = array[minimo];
    array[minimo] = aux;
    }
}
/*
La funcion selectSort imprime los elementos del arreglo ya ordenado
con ayuda de un ciclo for para iterar sobre el
*/
void mostrarArreglo (int a[], int tam) {
    //for(int i = 0; i < tam; i++)
        //printf("%d \n", a[i]);
        printf("SelectionSort finalizado");
}

//Fin del programa
```

Código fuente 5. Selection Sort

```
/**
// SERGE EDUARDO MARTÍNEZ RAMÍREZ
// ESCUELA SUPERIOR DE CÓMPUTO
// MATERIA: Análisis y diseño de algoritmos
```

```c
// FECHA: 13/03/2023
// ALGORITMO SHELLSORT:
//      El ordenamiento Shell, es un algoritmo de ordenamiento eficiente que mejora el ordenamiento
        por inserción, porque  divide la lista original en varias sublistas pequeñas y ordenarlas por separado
        a medida de que el intervalo va decreciendo.

        En el peor de los casos, la complejidad de este algoritmo es n^2 en el mejor de los casos es O(n log^2 n) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
// PROTOTIPOS DE FUNCIONES
void shellSort (int *a, int n);
void mostrarArreglo (int *a, int n);
void pedir(int *a, int n);

// Función de ShellSort
void shellSort (int *a, int n) {
    int intervalo, i, j, k;
    //Se divide la lista
    intervalo = n / 2;
    // Mientras el intervalo sea mayor a cero
    while (intervalo > 0) {
        // Se itera desde la mitad de la sublista hasta el final
        for (i = intervalo; i < n; i++) {
            // Se toma el valor que se encuentra al inicio de cada sublista
            j = i - intervalo;
            while (j >= 0) {
                k = j + intervalo;
                // Se commparan los valores, si es menor, se sale del bucle porque no hay que hacer intercambio
                if (a[j] <= a[k])
                    j = -1; // Se sale del búcle si esto se cumple

                // En caso de que no, entonces se realiza el intercambio
                else {
                    int auxiliar = a[j];
                    a[j] = a[k];
                    a[k] = auxiliar;
                    j -= intervalo;
                }
            }
```

```
        }
        // Se reduce a la mitad en cada iteración, hasta que intervalo teng
el valor de 1.
        intervalo /= 2;
    }
}

// Se encarga de mostrar el arreglo ordenado
void mostrarArreglo (int *a, int n) {
    //int i;
    //for (i = 0; i < n; i++)
        //printf("%d\n", a[i]);
    //printf("\n");
    printf("ShellSort finalizado");
}

// Se encarga de recibir la entrada de datos a ordenar
void pedir(int *a, int n) {
    int i;
    for(i = 0; i < n; i++)
        scanf("%d", &a[i]);
}

int main(int argc, char *argv[]) {
    // Haciendo un arreglo dinámico de N elementos
    if(argc!=2){
        exit(1);
        }
    argc=atoi(argv[1]);
    int *arreglo = (int*) malloc(argc * sizeof(int));
    pedir(arreglo, argc);
    shellSort(arreglo, argc);
    mostrarArreglo(arreglo, argc);
    free(arreglo); // Se libera la memoria
    return 0;
}
```

Código fuente 6. Shell Sort

```
/*
Codigo por Miriam G Ramirez Sanchez
Escuela Superior de Computo
Curso: Analisis y diseno de algoritmos
Algoritmo de ordenamiento Arbol Binario
fecha: 14/03/2023
```

```
*/

//librerias

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

/*
Tipo de dato abstracto Nodo. Cada nodo contiene un dato
en este caso de tipo entero y un apuntador a siguiente, en este caso
a izquierda y derecha en donde en izquierda encontraremos valores menores al
dato del nodo
y en la derecha valores menores a este
*/

struct Nodo {
    int dato;
    struct Nodo *izquierda;
    struct Nodo *derecha;
};

/*
Creamos un apuntador al nuevo nodo y le pasamos como parametro el valor del
dato del nodo anterior y definimos que el TDA
Nodo sea dinamico. Declaramos un apuntador del TDA Nodo y apuntara al dato
recibido del parametro, este sera la raiz del arbol
*/

struct Nodo *nuevoNodo(int dato) {
    size_t tamanioNodo = sizeof(struct Nodo);
    struct Nodo *nodo = (struct Nodo *) malloc(tamanioNodo);
    nodo->dato = dato;
    nodo->izquierda = nodo->derecha = NULL;
    return nodo;
}

//Prototipo de funciones

int datoI(struct Nodo *raiz);
void insertar(struct Nodo *nodo, int dato);
void inorden(struct Nodo *nodo);


/*
```

```
Funcion principal de dos argumentos, un entero y un apuntador de tipo char a
el arreglo cadena.
Estos parametros me ayudaran a recibir datos desde la consola de un archivo
de texto
, si la funcion no tiene ambos parametros, valida y automaticamente sale del
programa.
La funcion recibe un primer dato con el que inicia el arbol, el peor de los
casos seria que el primer dato
fuese el mas pequeno del arreglo ya que lo asigna como raiz y no incluimos
una funcion que haga que el
arbol siempre este balanceado.
*/
int main(int n, char *cadena[]) {

    int raizDato,i;
       if(n!=2){
            exit(1);
        }
    int el=atoi(cadena[1]);
    scanf("%d", &raizDato);
    struct Nodo *raiz = nuevoNodo(raizDato);
        for(i=1;i<el;i++){//Ingresara los demas nodos, tomando como
comparacion al Nodo padre
                datoI(raiz);
        }inorden(raiz);
    return 0;
}

/*
La funcion insertar es una funcion donde la operacion basica es la
comparacion, ya que
define en donde insertara el valor del dato, recibe como parametro el
apuntador a un nodo y
el valor entero del dato y empieza a hacer comparaciones. Si el dato es
mayor que la raiz
del nodo, se insertara a la derecha, creando dos posibles escenarios, si no
hay un nodo existente lo crea
y asigna el dato, de lo contario solamente lo asigna. Ocurre lo contrario
cuando es menor, ya que se
agregara a la izquierda y realizara la misma evaluacion.
*/
void insertar(struct Nodo *nodo, int dato) {

    if (dato > nodo->dato) {
        if (nodo->derecha == NULL) {
```

```
            nodo->derecha = nuevoNodo(dato);

        } else {
            insertar(nodo->derecha, dato);
        }
    } else {
        if (nodo->izquierda == NULL) {
            nodo->izquierda = nuevoNodo(dato);
        } else {
            insertar(nodo->izquierda, dato);
        }
    }
}

/*
En esta funcion recursiva realizamos el recorrido inorden de un arbol
binario. Recibe como parametro un apuntador y un entero auxiliar
este apuntador me permite ir a las diferentes posiciones del arreglo donde
estaran mis datos ordenados y
y el entero auxiliar me permite definir el tamano de este arreglo, ya que es
dinamico y en base a esto primero validara que el arbol
no este vacio y procedera a realizar iteraciones mientras asigna valores a
los elementos del arreglo
*/
void inorden(struct Nodo *nodo) {
    if (nodo != NULL) {
        inorden(nodo->izquierda);
        //printf("%d \n", nodo->dato);
        inorden(nodo->derecha);
    }
    printf("Ordenamiento ABB finalizado");
}

/*
La funcion datoI es la que nos ayuda a delcarar el incio del arbol, ya que
necesitamos una raiz, aunque esta funcion nos limita
a que exista un desbalance el arbol que no es lo mas correcto en cuestiones
algoritmicas, ya que se puede optimizar pero para este
caso no implementamos una funcion que rebalancee. Toma como parametro un
apuntador a raiz en donde se quedara el valor guardado y
retorna 0 ya que no realiza operaciones que devuelvan informacion
*/
int datoI(struct Nodo *raiz){
    int dato1;
    scanf("%d", &dato1);
```

```
    insertar(raiz, dato1);

    return 0;
}

//Fin del programa
```

Código fuente 7. Tree Sort

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
void mergeSort(int *a, int inicio, int final);
void mezcla(int *a, int inicio, int centro, int final);
int main(int argc, char *argv[]){
    if(argc!=2)
        exit (1);
    int n=atoi(argv[1]);
    int *arreglo;
    int i;
    arreglo = malloc(n*sizeof(int));//creación del arreglo dinámico
    for(i=0;i<n;i++){
        scanf("%d",&arreglo[i]);//lee los datos del archivo
numeros10millones.txt
    }
    mergeSort(arreglo, 0, n-1);//llamado a la función merge, utiliza el
arreglo y su tamaño
    //for(i=0; i < n; i++)
        //printf("%d\n", arreglo[i]);//impresión de los números ordenados
        printf("MergeSort finalizado");
    free(arreglo);//liberar el espacio del arreglo
    return 0;
}

void mergeSort(int *a, int inicio, int final){
    if(inicio < final){
        int central = (inicio + final) / 2;
        mergeSort(a, inicio, central); // Ordena la primera mitad de la
lista
        mergeSort(a, central+1, final); // Ordena la segunda mitad de la
lista
        mezcla(a, inicio, central, final); // Fusiona las dos sublistas
ordenadas
    }
}
```

```
void mezcla(int *a, int inicio, int centro, int final){//funcion de mezclar
arreglos
    int i, j, k;
    int elementosIzq = centro - inicio + 1;
    int elementosDer = final - centro;

    int *izquierdo = (int*) malloc(elementosIzq*sizeof(int));
    int *derecho = (int*) malloc(elementosDer*sizeof(int));

    for(i = 0; i < elementosIzq; i++)
        izquierdo[i] = a[inicio+i];

    for(j = 0; j < elementosDer; j++)
        derecho[j] = a[centro + 1 + j];

    i = j = 0;
    k = inicio;

    while(i < elementosIzq && j < elementosDer){
        if(izquierdo[i] <= derecho [j]){
            a[k] = izquierdo[i];
            i++;
        }   else{
            a[k] = derecho[j];
            j++;
        }
        k++;
    }

    /* En caso de que alguno de los arreglos se quede sin elementos
     para comparar, debe de vaciar los arreglos */
    while(j < elementosDer){
        a[k] = derecho[j];
        j++; k++;
    }

    while(i < elementosIzq){
        a[k] = izquierdo[i];
        i++; k++;
    }

    free(izquierdo);
    free(derecho);
}
```

Código fuente 8. Merge Sort

```c
/**
// SERGE EDUARDO MARTÍNEZ RAMÍREZ
// ESCUELA SUPERIOR DE CÓMPUTO
// MATERIA: Análisis y diseño de algoritmos
// FECHA: 13/03/2023
// ALGORITMO QUICKSORT:
//      Es un algoritmo "divide y vencerás" que utilza la estrategia de
partir la lista para poder ordenarla,
        tomando un elemento denominado "pivote" que nos sirve para partir la
lista en dos nuevas sublistas. En la sublista
        izquierda tenemos los valores que son menores al valor del pivote, y
del lado derecho tenemos los valores que son
        mayores al valor del pivote. En cada llamada a la función se va
partiendo de esta manera hasta que las sublistas
        tienen el tamaño de 1 o 0. */
//
#include <stdio.h>
#include <stdlib.h>

// PROTOTIPOS DE FUNCIONES
void quickSort(int *a, int primero, int ultimo);
void pedir(int *a, int n);
void mostrarArray(int *a, int ns);

// FUNCIÓN PRINCIPAL
int main(int x, char *cadena[]){
    // Si el programma no recibe dos argumentos a la hora de correrlo, va a
haber un error.
    if (x!=2)
        exit(1);
    int *arreglo;
    int i;
    // La cadena recibida, se convierte a un entero para poder asignar el
tamaño del arreglo
    int n = atoi(cadena[1]);
    arreglo = malloc(n * sizeof(int));
    // Otra causa de error, es que no se pueda asignar la cantidad de
memoria solicitada
    if(arreglo == NULL) {
        printf("No se pudo asignar memoria correctamente\n");
        exit(1);
    }
```

```
    for(i =  0; i < n; i++)
        scanf("%d", &arreglo[i]);
    quickSort(arreglo, 0, n-1);
    //for(i = 0; i < n; i++)
        //printf("%d\n", arreglo[i]);
    printf("QuickSort finalizado");
    free(arreglo);
    system("pause");
    return 0;
}

/**
 * Descripción: Es una función recursiva encargada de ordenar los elementos
de un arreglo
 * de tamaño N, recibe el arreglo, la primer y última posición del arreglo.
 * Regresael arreglo ordenado ascendentemente
*/

void quickSort(int *a, int primero, int ultimo){
    // Tomando la primera y última posición de la lista
    int i = primero, j = ultimo;
    // Tomando el valor central de la lista, este será el pivote.
    int centro = (primero + ultimo) / 2;
    int pivote = a[centro];
    do {
        while(a[i] < pivote) i++; // Mientras el elemento a[i] sea menor que
el valor del pivote
        while(a[j] > pivote) j--; // Mientras el elemento a[j] sea mayor que
el valor del pivote

        // Si i < j, entonces se realiza un intercambio de valores entre la
posición i y la posición j
        if(i <= j){
            /* Una vez tenemos los dos valores, los intercambiamos para
tener en la sublista
            izquierda los valores menores que pivote y en la sublista
derecha los valores mayores que pivote*/
            int auxiliar; // Creando variable auxiliar para realizar un
intercambio de valores
            // Realizando el intercambio de valores
            auxiliar = a[i];
            a[i] = a[j];
            a[j] = auxiliar;
            i++; j--;
        }
```

```
    }   while(i <= j); // Esto se repite mientras i sea menor o igual que j

    if(primero < j)
    //Se repite el proceso para la sublista izquierda
        quickSort(a, primero, j);
    if(ultimo > i)
    // Se repite el proceso para la sublista derecha
        quickSort(a, i, ultimo);
}
```

Código fuente 9. Quick Sort

```
/*
Codigo por Miriam G Ramirez Sanchez
Escuela Superior de Computo
Curso: Analisis y diseno de algoritmos
Algoritmo de ordenamiento heapSort
fecha: 14/03/2023
*/

//librerias

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

//Prototipo de funciones

void intercambio(int* a, int* b);
int obtenerHijoIzquierdo(int i);
int obtenerHijoDerecho(int i);
void monticulo(int arr[], int N, int i);
void HeapSort(int A[], int n);
void imprimeDatos(int arr[], int N);

/*
Funcion principal de dos argumentos, un entero y un apuntador de tipo char a
el arreglo cadena.
Estos parametros me ayudaran a recibir datos desde la consola de un archivo
de texto
, si la funcion no tiene ambos parametros, valida y automaticamente sale del
programa.
El arreglo que estamos usando es un arreglo dinamico por lo que tambien
pedimos que nos valide
si la asigancion de memoria fue correcta, si esto sucede procede a recibir
los datos que ordenara
```

```
y a agregarlos al arreglo para despues enviar por parametro a la funcion
HeapSort, incluyendo el numero
de elementos que ordenara y la funcion guardaDatos los colocara ya ordenados
en el arreglo
*/

int main(int n, char *cadena[]){
    if(n!=2){
        exit(1);
    }
        int opc =atoi(cadena[1]);
    int a;
    int *arr;
    arr = malloc(opc * sizeof(int));
        if (arr == NULL) {
            printf("Sin memoria");
            exit(1);
        }
        else{
        for(int i=0;i<opc;i++){
                    scanf("%d", &arr[i]);//recibe los datos a ordenar
            }
            HeapSort(arr,opc);
            printf("HeapSort finalizado");
            imprimeDatos(arr,opc);//Regresa el arreglo ordenado
        }
        return 0;
}

/*
La funcion de intercambio es una funcion auxiliar para poder cumplir con el
paso del
algoritmo de heapSort ya que realiza un intercambio del elemento mas a la
izquiera y lo pone en el
primer elemento del arreglo para despues poder re equilibrar el arbol y que
cumpla
que siempre este completo, igual es auxiliar para que siempre se cumpla que
el elemento mayor
es la raiz del arbol
*/
void intercambio(int* a, int* b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
/*
La funcion obtenerHijoIzquierdo y obtenerHijoDerecho nos brindan la posicion
exacta
en el arreglo de los respectivos hijos, esto se puede calcular atraves de
una expresion algebraica
ya que en el algoritmo heapSort se implementa un arbol binario completo,
esto quiere decir
que siempre tendra dos hijos y en caso de no hacerlo, estaran lo mas a la
izquierda posible, y
el número de nodos totales cumple una fórmula específica (2^(h+1) - 1) en
función de la altura del arbol
*/
int obtenerHijoIzquierdo(int i) {
    return 2 * i + 1;
}

int obtenerHijoDerecho(int i) {
    return 2 * i + 2;
}
/*
La funcion monticulo forma pequenos monticulos como su nombre lo dice para
facilitar el
trabajo de la funcion HeapSort, en este algoritmo de ordenamiento lo mas
importante es esta funcion
ya que de ella depende si es mas o menos eficiente el ordenamiento.
la estructura de monticulo es un pequeno arbol binario completo y debe de
tener siempre
una raiz y sus respectivos dos nodos hijos, esta funcion en ayuda de
intercambio hacen que en cada 'pasada'
que realice el algoritmo el arbol se vuelva a balancear y este completo.
*/
void monticulo(int arr[], int N, int i){

    int may = i;//raiz //padre
    //Hijos
    int izq = obtenerHijoIzquierdo(i);
    int der = obtenerHijoDerecho(i);
//Hace las comparaciones para que el arbol siempre sea completo
    if (izq < N && arr[izq] > arr[may]){
        may = izq;
    }

    if (der < N && arr[der] > arr[may]){
        may = der;
    }
```

```c
//Reacomoda la raiz ya que el heapSort remueve el mayor elemento y lo
//sustituye por el ultimo elemento del arreglo
    if (may != i) {

        intercambio(&arr[i], &arr[may]);
        monticulo(arr, N, may);
    }
}
/*
La funcion HeapSort es en donde se lleva a cabo todo el algoritmo, uniendo
las funciones
anteriormente descritas, recibe por parametro un arreglo de datos y un
entero que contiene
el tamano del arreglo ya que es una implementacion dinamica, en el primer
ciclo
manda a llamar a la funcion monticulo para ordenar los datos y en el segundo
ciclo
elimina el elemento mayor y lo sustituye por el que esta mas a la izquierda
con ayuda
de la funcion de intercambio y rebalancea el arbol
*/
void HeapSort(int A[], int n){
    int i;
    for (i = n / 2 - 1; i >= 0; i--){
        monticulo(A, n, i);
    }
    for (i = n - 1; i >= 0; i--) {
        intercambio(&A[0], &A[i]);
        monticulo(A, i, 0);
    }
}
/* Esta funcion imprime los valores del conjunto ya ordenados
*/

void imprimeDatos(int arr[], int N){
    int i;
    for (i = 0;i < N; i++){
       // printf("%d \n", arr[i]);
    }
}

//Fin del programa
```

Código fuente 10. Heap Sort

Bibliografía

Bash. https://www.youtube.com/watch?v=np-8ivtLRwI&list=PLDbrnXa6SAzUsIAqsjVOeyagmmAvmwsG2

Comparación de algoritmos. https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/

Funciones en Bash. https://www.delftstack.com/es/howto/linux/functions-in-bash/ https://www.delftstack.com/es/howto/linux/functions-in-bash/

Heapsort. https://es.wikipedia.org/wiki/Heapsort