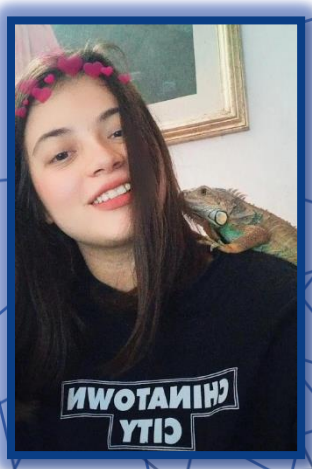
Practica Numero 1

PRUEBAS A POSTERIORI

Equipo:

- Mora Ayala José Antonio
- Jeon Jeong Paola
- Lemus Ruiz Mariana Elizabeth
- López López Oscar Manuel



INTRODUCCIÓN

Por lo general, el mismo problema puede tener Muchas soluciones con diferentes eficiencias (Velocidad de ejecución). En este ejercicio, utilizaremos y compararemos diferentes algoritmos de búsqueda. Un aspecto muy importante para considerar en el algoritmo es la velocidad de cada algoritmo. Es fácil encontrar un algoritmo para resolver el problema, pero si el algoritmo es demasiado lento, debe volver atrás y rediseñarlo. Entonces decimos que el propósito de estudiar los algoritmos de ordenación es preliminar, porque nos permite ilustrar la importancia de estudiar la eficiencia de los algoritmos, mostrando que no es intuitivo predecir cuánto tiempo de ejecución requieren.

Por lo general, el mismo problema puede tener Muchas soluciones con diferentes eficiencias En los cálculos, el ordenamiento de los datos juega un papel muy importante, ya sea como un fin en sí mismo o como parte de otros procesos más complejos. Se han desarrollado muchas tecnologías en este campo, cada una de las cuales tiene características específicas y tiene ventajas y desventajas en comparación con otras tecnologías.

DEFINICIÓN DEL PROBLEMA

Con base en el archivo de entrada proporcionado que tiene hasta 10,000,000 de números diferentes; ordenar bajo los siguientes 9 métodos de ordenamiento y comparar experimentalmente las complejidades aproximadas según se indica en las actividades a reportar

- Burbuja (Bubble Sort)
- Burbuja simple
- Burbuja optimizada
- Burbuja optimizada 2
- 2. Inserción (Insertion Sort)
- 3. Selección (selection sort)
- 4. Shell (Shell sort)
- 5. Ordenamiento con árbol binario de búsqueda (Tree Sort)
- 6. Ordenamiento por mezcla
- 7. Ordenamiento rápido

ORDENAMIENTO BURBUJA

Burbuja Simple

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 varías veces toda la lista hasta que no se necesiten más intercambios, pues en ese momento significara que la lista ya se encuentra ordenada

```
Algoritmo BurbujaSimple(A,n)
        para i=0 hasta n-2 hacer
                para j=0 hasta n-2 hacer
                        si (A[j]>A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                        fin si
                fin para
        fin para
fin Algoritmo
```

$$f_t(n) = n^2$$

Burbuja Optimizada

Implementa una mejora al método de ordenamiento burbuja. Este método recorre todo el arreglo comparando cada uno de los elementos con el elemento siguiente e intercambiándolo de ser necesario. Al finalizar la iteración el elemento mayor queda ubicado en la última posición, mientras los elementos menores ascienden una posición.

La mejora es que, 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.

```
Algoritmo BurbujaSimple(A,n)
        para i=0 hasta n-2 hacer
                para j=0 hasta (n-2)-i hacer
                        si (A[j]>A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                        fin si
                fin para
        fin para
fin Algoritmo
```

BURBUJA OPTIMIZADA 2

```
Algoritmo BurbujaOptimizada(A,n)
        cambios = SI
        i=0
        mientras i<= n-2 && cambios != NO hacer
                cambios = NO
                para j=0 hasta (n-2)-i hacer
                        si(A[j] < A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                                cambios = SI
                        fin si
                fin para
                 i= i+1
        fin mientras
fin Algoritmo
```

ORDENAMIENTO POR INSERCIÓN

Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

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

$$f_t(n) = n^2$$

ORDENAMIENTO POR SELECCIÓN

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.

Su funcionamiento es el siguiente:

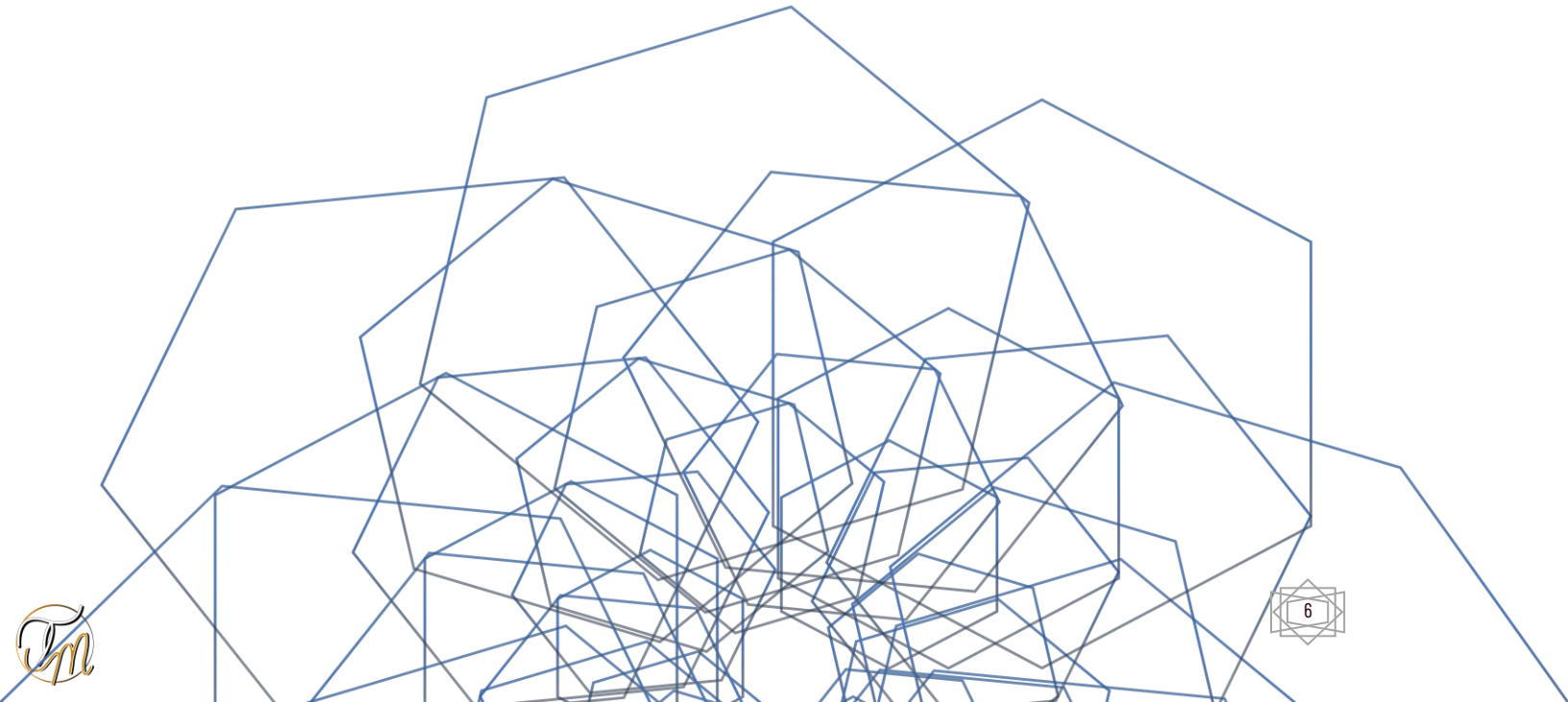
- Buscar el mínimo elemento de la lista
- Intercambiarlo con el primero
- Buscar el siguiente mínimo en el resto de la lista
- Intercambiarlo con el segundo

Y en general:

- Buscar el mínimo elemento entre una posición i y el final de la lista
- Intercambiar el mínimo con el elemento de la posición i

```
Algoritmo Seleccion(A,n)
        para k=0 hasta n-2 hacer
                p=k
                para i=k+1 hasta n-1 hacer
                        si A[i]<A[p] entonces
                                p=i
                        fin si
                fin para
                temp = A[p]
                A[p] = A[k]
                A[k] = temp
        fin para
fin Algoritmo
```

$$f_t(n) = n^2$$



ORDENAMIENTO SHELL

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

1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".

2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

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

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

$$f_t(n) = nlog(n)^2$$

ORDENAMIENTO POR ÁRBOL DE BÚSQUEDA

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. Remplazar el vector en desorden por el vector resultante de un recorrido InOrden del Árbol Binario, el cual entregara los números ordenados.

$$f_t(n) = nlog(n)$$

ORDENAMIENTO POR MEZCLA

```
Algoritmo Merge(A, p, q, r)
   l=r-p+1, i=p, j=q+1
   para k=0 hasta k>=l hacer
      si i<=q y j<=r entonces
         si A[i]<A[j] entonces
            C[k]=A[i]
            i++
         sino entonces
            C[k]=A[j]
            j++;
         fin si
      sino si i<=q entonces
         C[k]=A[i]
         i++
      sino entonces
         C[k]=A[j]
         j++
      fin si
   A[p-r]=C[]
fin Algoritmo
```

```
Algoritmo MergeSort(A, p, r)
   si p < r entonces
      q = parteEntera((p+r)/2)
      MergeSort(A, p, q)
      MergeSort(A, q+1,r)
      Merge(A, p, q, r)
   fin si
fin Algoritmo
```

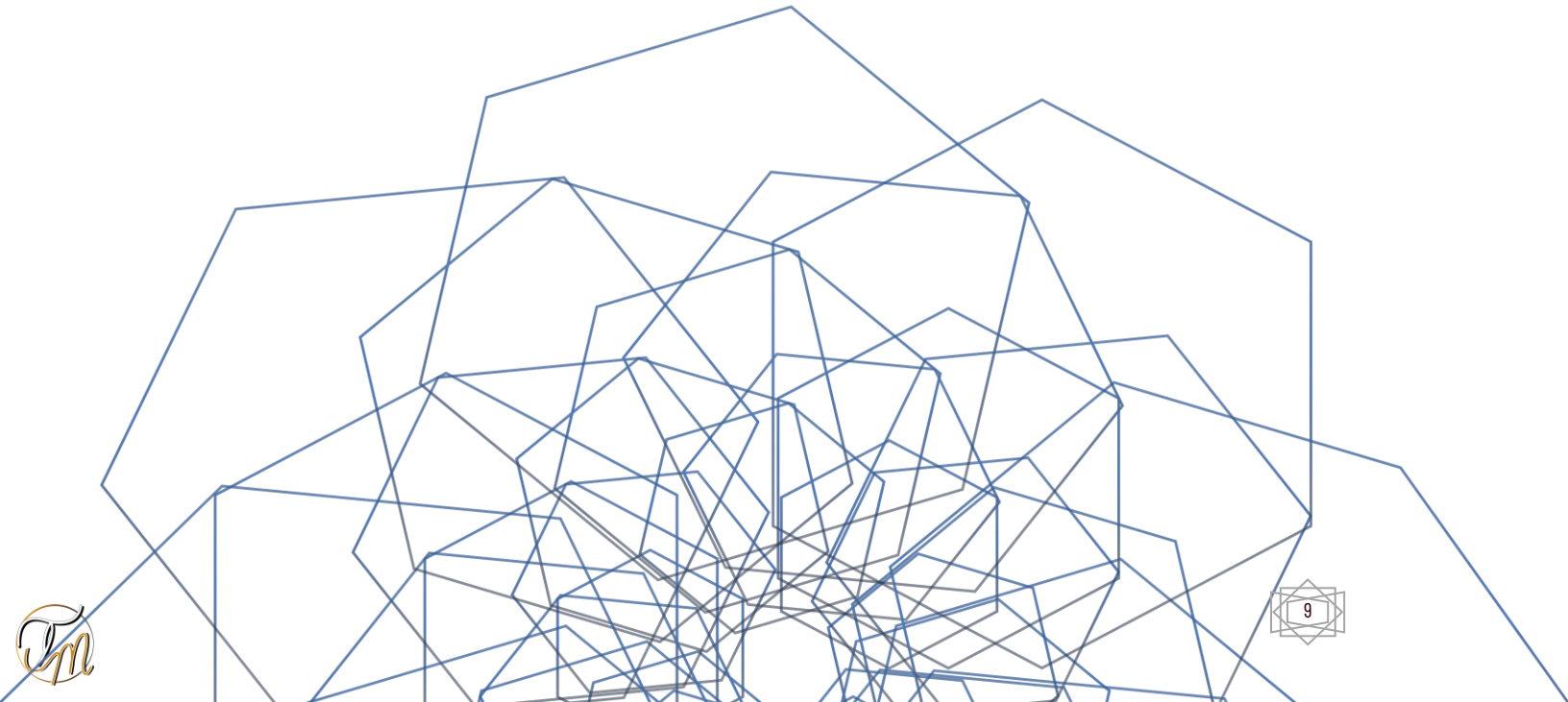
Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

- Si la longitud de la lista es 0 o 1, entonces ya está ordenada. En otro caso:
- Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
- Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
- Mezclar las dos sublistas en una sola lista ordenada.

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

- Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
- Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

$$\boldsymbol{f_t(n) = nlog(n)}$$



ORDENAMIENTO RÁPIDO (QUICKSORT)

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del conjunto de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

```
INICIO
Llenar(A)
Algoritmo quicksort(A,inf,sup)
i<-inf
j<-sup
x<-A[(inf+sup)div 2]
mientras i=<j hacer
  mientras A[i]< x hacer
     i<-i+1
  fin_mientras
  mientras A[j]>x hacer
     j<- j-1
  fin_mientras
  si i=<j entonces
    tam<-A[i]
    A[i]<-A[j]
    A[j]<-tam
    i=i+1
    j=j-1
  fin_si
fin_mientras
si inf<j
   llamar_a quicksort(A,inf,j)
fin_si
si i<sup
   llamar_a quicksort(A,i,sup)
fin_si
FIN
```

$$f_t(n) = nlog(n)$$

IMPLEMENTACIÓN

BURBUJA SIMPLE

```
void Burbuja(int *A, int n)
{
    int i, j, aux;
    uswtime(&utime0, &stime0, &wtime0);

    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1); j++)
        {
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
    uswtime(&utime1, &stime1, &wtime1);

    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}
```

BURBUJA OPTIMIZADA

```
void BurbujaOptimizada(int *A, int n)
{
    int i, j, aux;
    uswtime(&utime0, &stime0, &wtime0);
    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1) - i; j++)
        {
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
    uswtime(&utime1, &stime1, &wtime1);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
    // imprimirArreglo(A, n);
}
```

Burbuja Optimizada 2

```
void Burbuja2(int *A, int n)
{
    bool cambio = true;
    int i = 1;
    int j, aux;
    uswtime(&utime0, &stime0, &wtime0);

    while (i <= (n - 1) && cambio != false)
    {
        cambio = false;
        for (j = 0; j <= ((n - 1) - i); j++)
        {
            if (A[j + 1] < A[j])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
                cambio = true;
            }
        }
        i = i + 1;
    }
    uswtime(&utime1, &stime1, &wtime1);

    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}
```

Ordenamiento por Inserción

```
void Insercion(int *A, int n)
{
    int i, j, temp;
    uswtime(&utime0, &stime0, &wtime0);
    for (i = 0; i <= n - 1; i++)
    {
        j = i;
        temp = A[i];
        while ((j > 0) && (temp < A[j - 1]))
        {
            A[j] = A[j - 1];
            j--;
        }
        A[j] = temp;
    }
    uswtime(&utime1, &stime1, &wtime1);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
    // imprimirArreglo(A, n);
}
```

Ordenamiento por Selección

```
void Selection(int *A, int n)
{
    int i, k, p, temp;
    uswtime(&utime0, &stime0, &wtime0);
    for (k = 0; k < (n - 1); k++)
    {
        p = k;
        for (i = k + 1; i < (n); i++)
        {
            if (A[i] < A[p])
                p = i;
        }
        temp = A[p];
        A[p] = A[k];
        A[k] = temp;
    }
    uswtime(&utime1, &stime1, &wtime1);
    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}
```

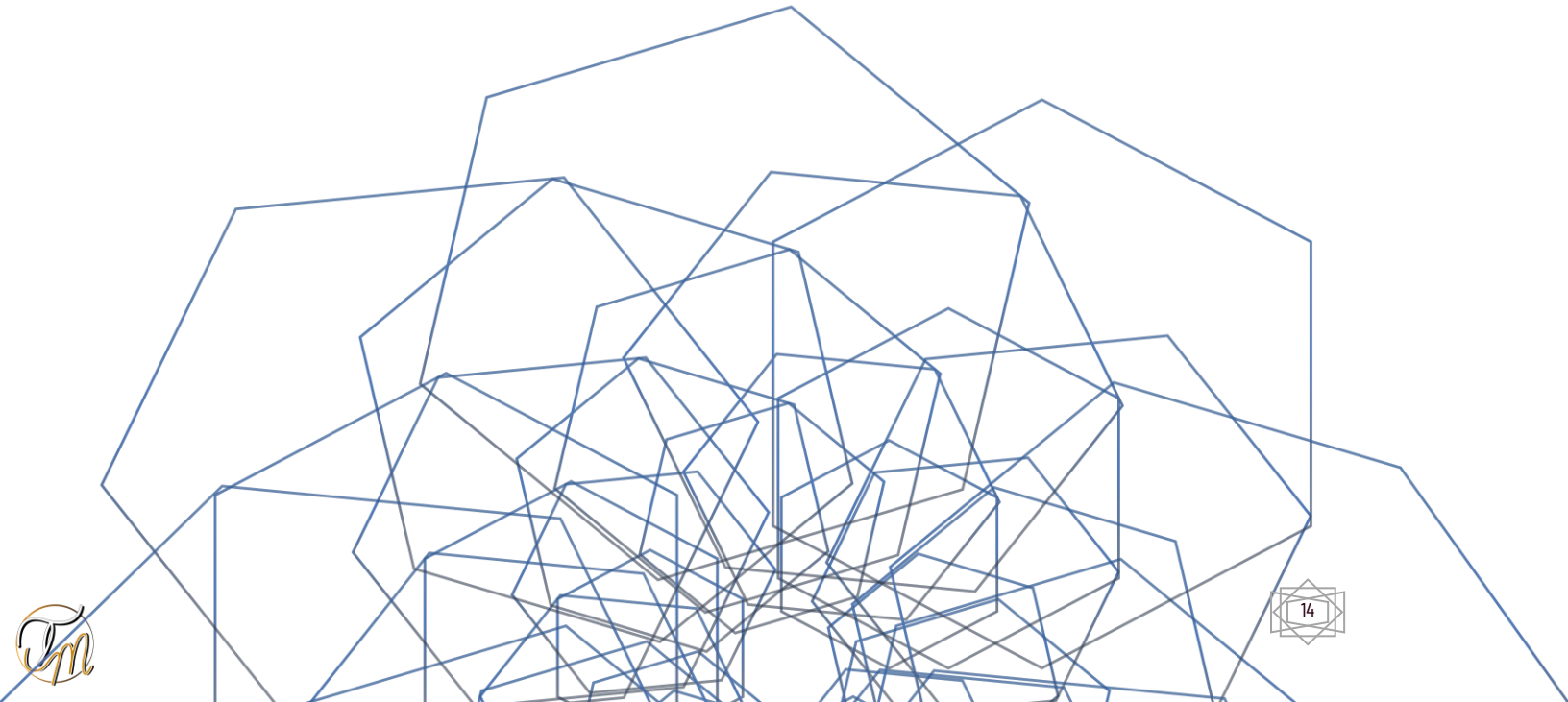
Ordenamiento Shell

```
void Shell(int *A, int n)
{
    int i, j, k, b, temp;
    uswtime(&utime0, &stime0, &wtime0);
    k = ceil(n / 2);
    while (k >= 1)
    {
        b = 1;
        while (b != 0)
        {
            b = 0;
            for (i = k; i <= n - 1; i++)
            {
                if (A[i - k] > A[i])
                {
                    temp = A[i];
                    A[i] = A[i - k];
                    A[i - k] = temp;
                    b = b + 1;
                }
```

```
            }
        }
        k = ceil(k / 2);
    }
    uswtime(&utime1, &stime1, &wtime1);

    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}
```

Ordenamiento por Árbol ABB

```
void ArbolBinario(int *A, int n)
{
    uswtime(&utime0, &stime0, &wtime0);
    A = InsertarABB(A, n);
    uswtime(&utime1, &stime1, &wtime1);
    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}
```



Ordenamiento por Mezcla (MergeSort)

```
int parteEntera(double n)
{
    double entero = floor(n);
    return (int)entero;
}

void MergeSort(int *A, int p, int r)
{

    if (p < r)
    {
        int q = parteEntera((p + r) / 2);
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        Merge(A, p, q, r);
    }
}

void Merge(int *A, int p, int q, int r)
{
    int l = r - p + 1, i = p, j = q + 1, k, *C;
    C = malloc(sizeof(int *) * l);

    for (k = 0; k <= l; k++)
    {
        if (i <= q && j <= r)
        {
            if (A[i] < A[j])
            {
                C[k] = A[i];
                i++;
            }
            else
            {
                C[k] = A[j];
                j++;
            }
        }
        else if (i <= q)
        {
            C[k] = A[i];
            i++;
        }
        else
        {
            C[k] = A[j];
            j++;
        }
    }
    for (i = p, j = 0; i <= r; i++, j++)
    {
```

```
        A[i] = C[j];
    }

    free(C);
}
```

Ordenamiento QuickSort

```
void Quicksort2(int *A, int primero, int ultimo)
{
    int piv, i, j, central, aux;
    central = (primero + ultimo) / 2;
    piv = A[central], i = primero, j = ultimo;
    do
    {
        while (A[i] < piv)
        {
            i++;
        }
        while (A[j] > piv)
        {
            j--;
        }
        if (i <= j)
        {
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
            i++;
            j--;
        }
    } while (i <= j);
    if (primero < j)
    {
        Quicksort2(A, primero, j);
    }
    if (primero < ultimo)
    {
        Quicksort2(A, i, ultimo);
    }
}
```

ACTIVIDADES Y PRUEBAS

Especificaciones del equipo usado

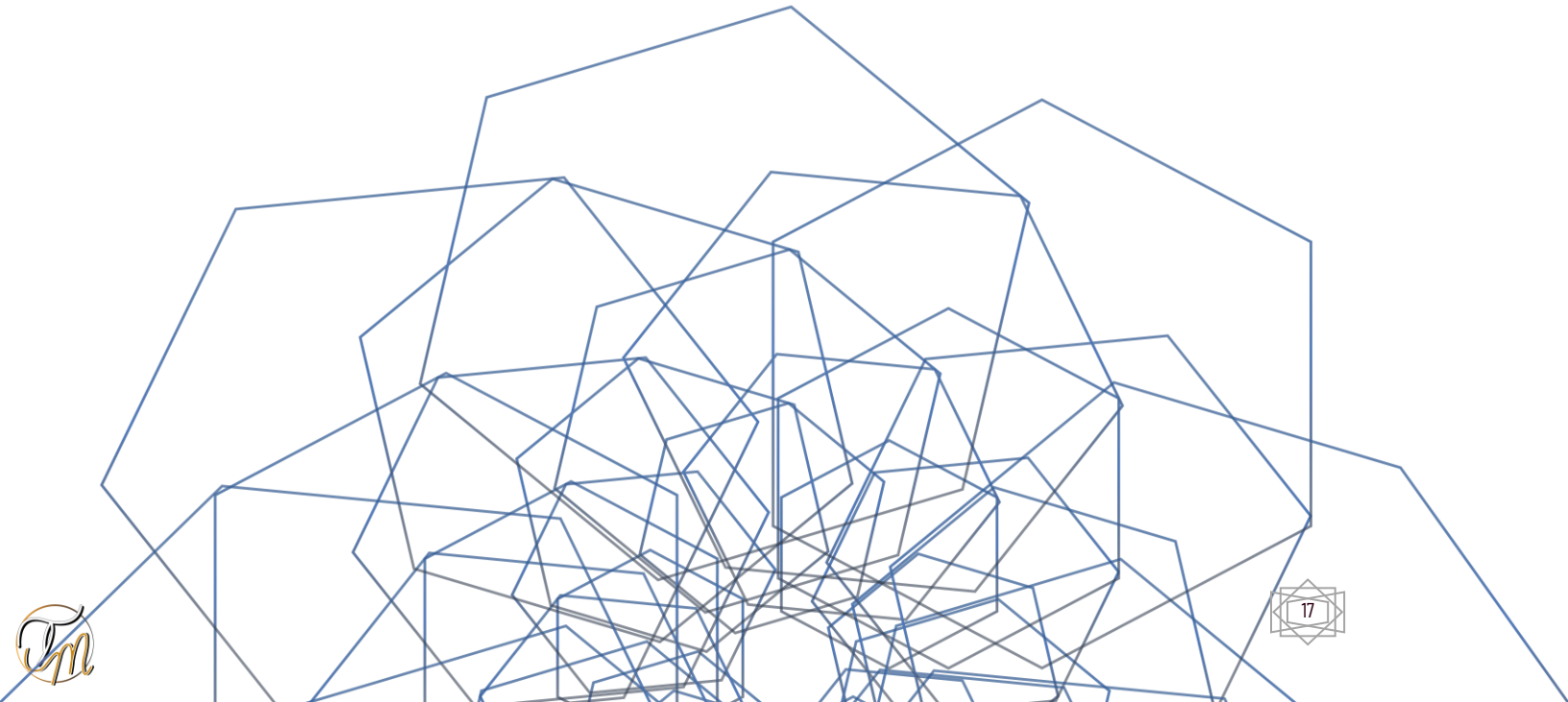
Para la realización de las pruebas de todos los algoritmos opté por usar el mismo equipo de cómputo, pues de esta forma las comparativas proporcionadas en este documento tendrán una relación justo, ya que habrán sido sometidas a prueba con las mismas especificaciones computacionales, claro que los resultados pueden variar en comparación de otros precisamente por esa cuestión, el rendimiento de cada equipo computacional.

A continuación, se listas las especificaciones del ordenador usado:

- Marca HP Notebook modelo g3 240
- 4 GB ram DDR3
- SSD 120 gb
- Intel Celeron (2 Núcleos)

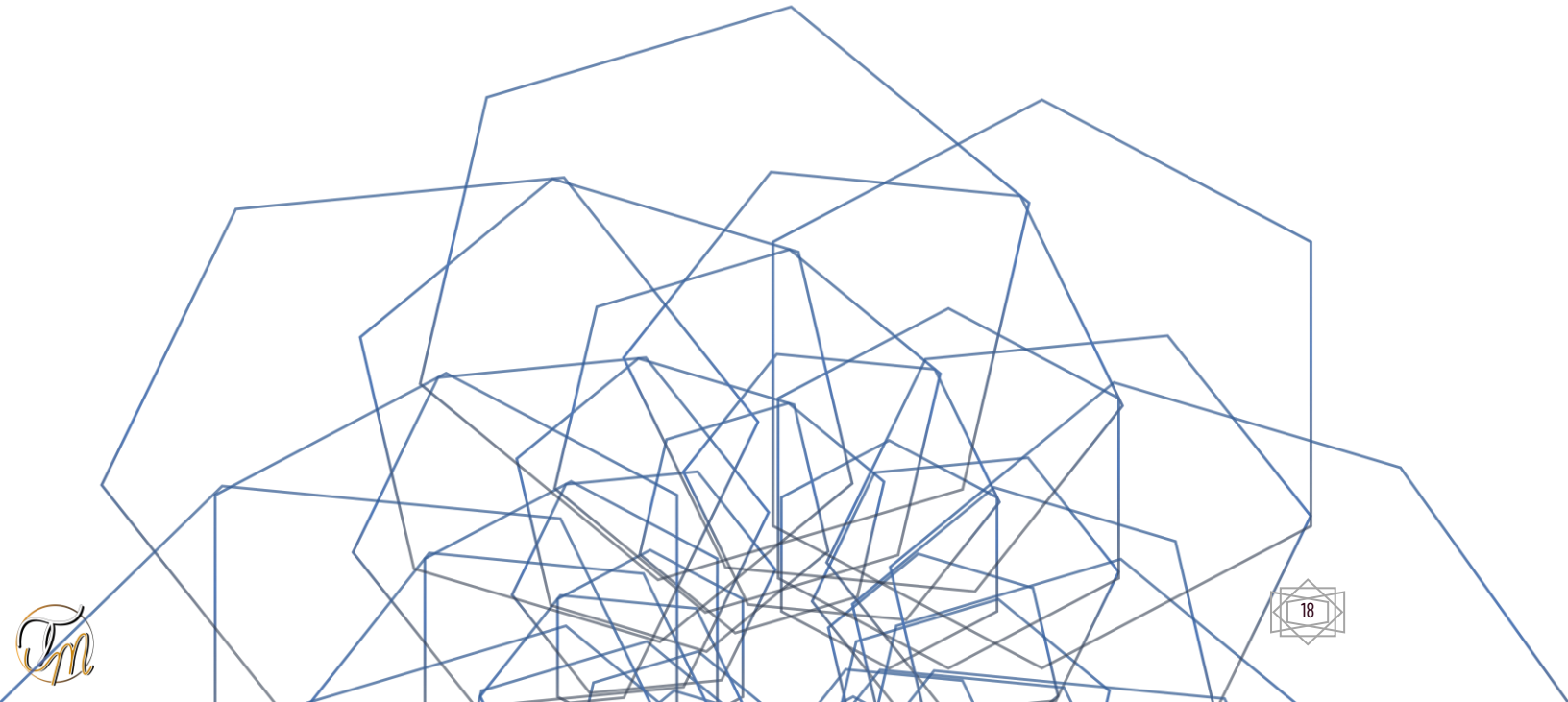
Sistema Operativo

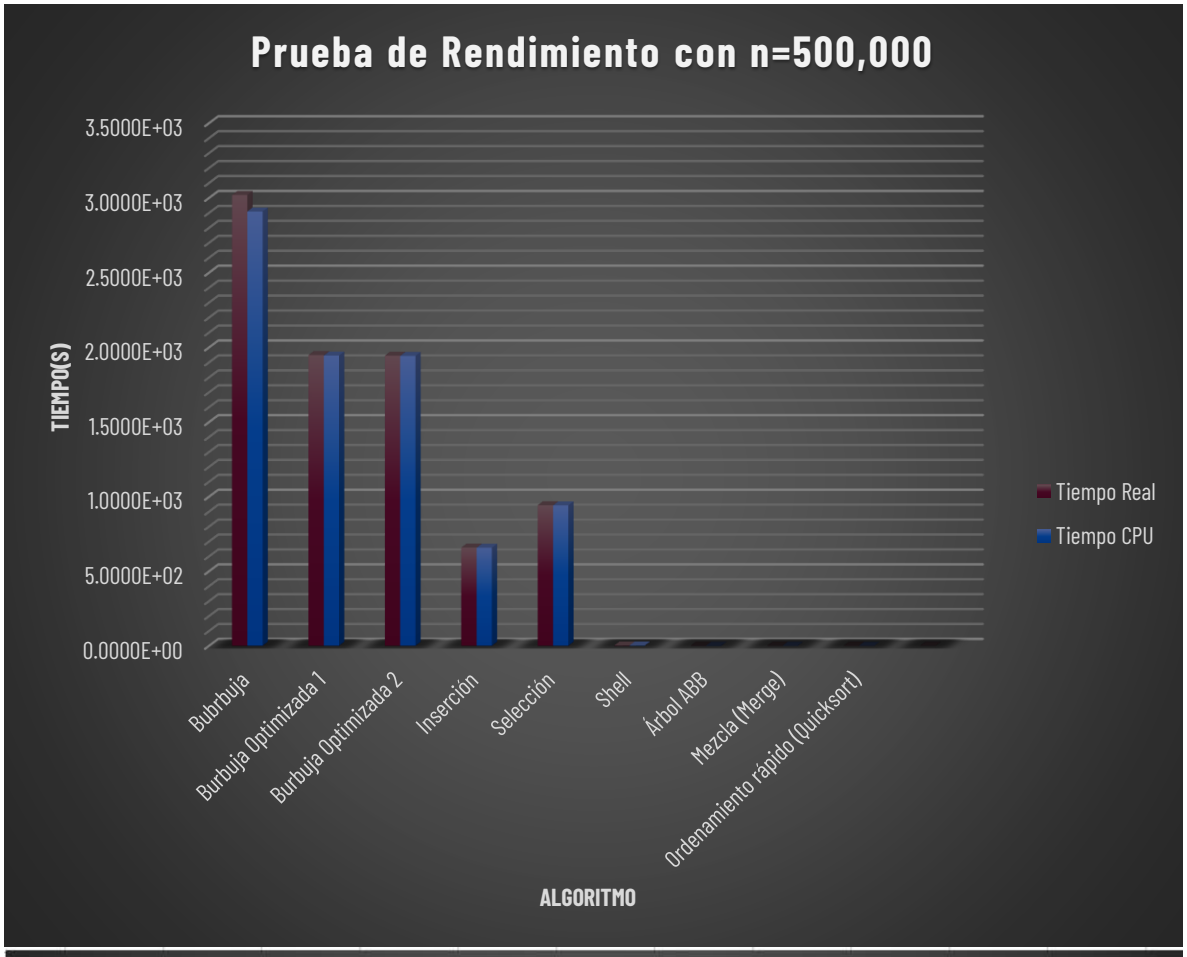
EL Sistema Operativo con el cual fueron robados todos los algoritmos en cuestión fue una distribución de Linux llamada: **Elementary OS**

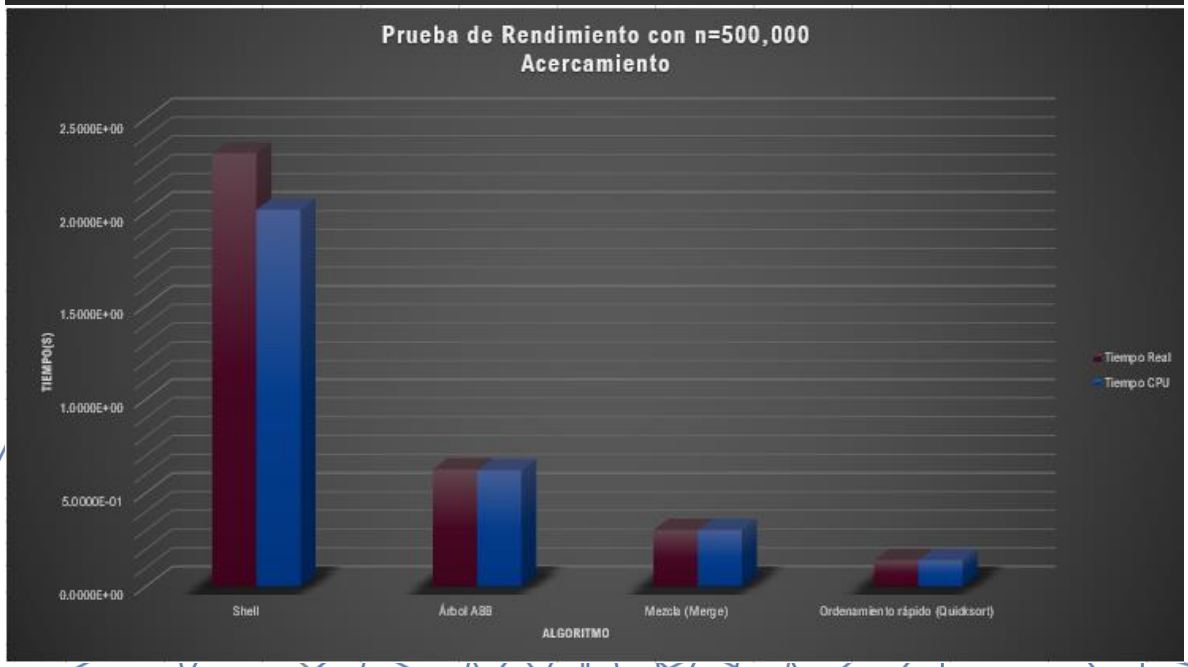


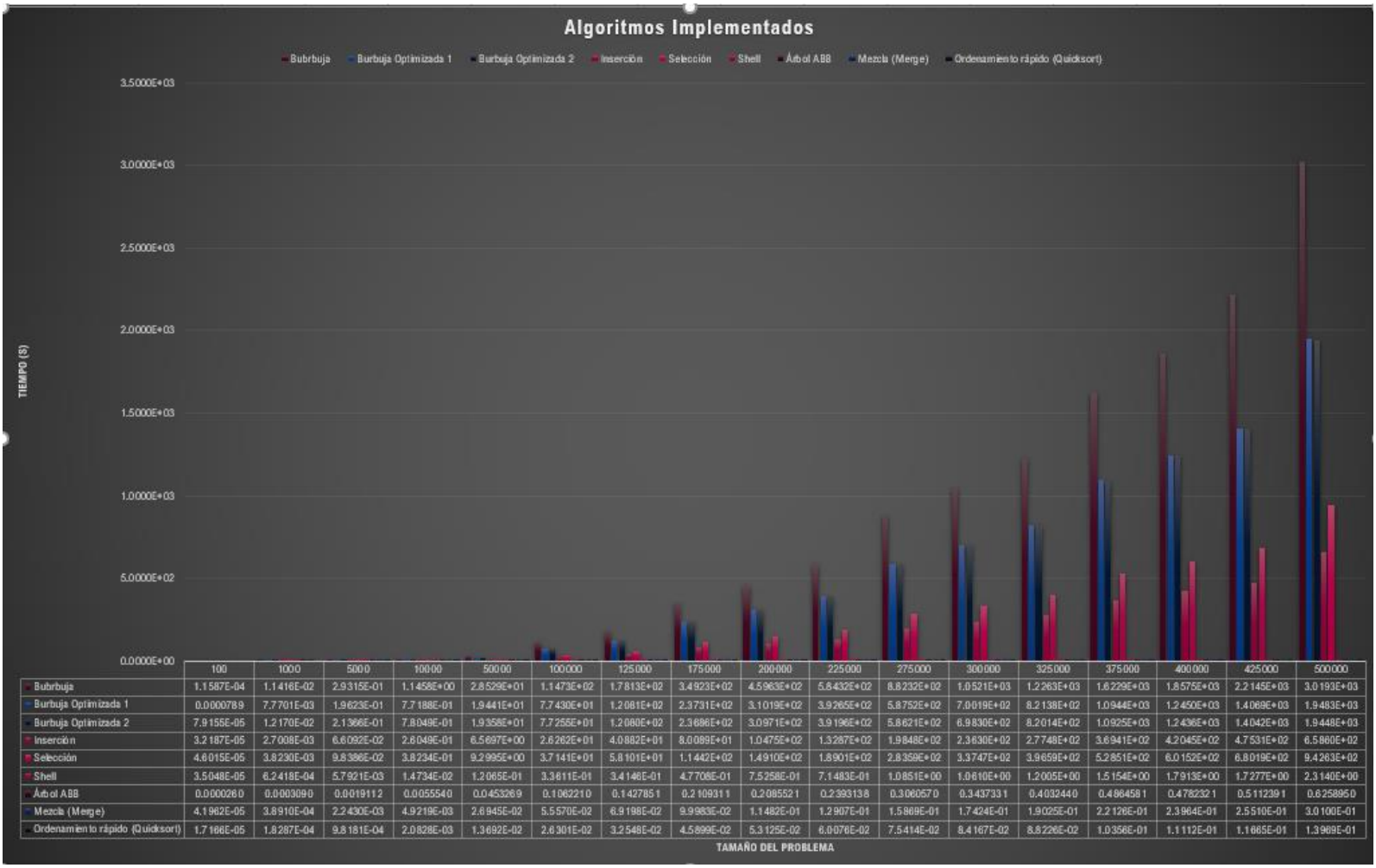
TIEMPO QUE TARDA CADA ALGORITMO EN EL ORDENAMIENTO DE N=500,000 NÚMEROS

n=500,000				
Algoritmo	Tiempo Real	Tiempo CPU	Tiempo E/S	%CPU/Wall
Burbuja	2.9227142401e+03 s	2.9180603180e+03 s	1.0237500000e+00 s	99.8757944918 %
Burbuja Optimizada 1	1.9413159690e+03 s	1.9408575040e+03 s	7.1963000000e-02 s	99.9800907221 %
Burbuja Optimizada 2	1.9448E+03	1.9429E+03	7.8424E-01	9.9945E+01
Inserción	6.5625005794e+02 s	6.5606074800e+02 s	6.8062000000e-02 s	99.9815241257
Selección	9.4223684502e+02 s	9.4045866800e+02 s	2.6084500000e-01 s	99.8389649031 %
Shell	1.9870009422e+00 s	1.9798090000e+00 s	4.1420000000e-03 s	99.8465052449 %
Árbol ABB	6.7595481873e-01 s	6.6078400000e-01 s	1.2636000000e-02 s	99.6250017523 %
Mezcla (Merge)	2.9991006851e-01 s	2.9889300000e-01 s	2.0900000000e-04 s	99.7305630598 %
Ordenamiento rápido (Quicksort)	1.4208817482e-01 s	1.4153900000e-01 s	2.3200000000e-04 s	99.7767760615 %




Prueba de Rendimiento con n=500,000
3.5000E+03
3.0000E+03
2.5000E+03
2.0000E+03
1.5000E+03
1.0000E+03
5.0000E+02
0.0000E+00
TIEMPO(S)
Tiempo Real
Tiempo CPU
Burbuja
Burbuja Optimizada 1
Burbuja Optimizada 2
Inserción
Selección
Shell
Árbol ABB
Mezcla (Merge)
Ordenamiento rápido (Quicksort)
ALGORITMO


Prueba de Rendimiento con n=500,000
Acercamiento
2.5000E+00
2.0000E+00
1.5000E+00
1.0000E+00
5.0000E-01
0.0000E+00
TIEMPO(S)
Tiempo Real
Tiempo CPU
Shell
Árbol ABB
Mezcla (Merge)
Ordenamiento rápido (Quicksort)
ALGORITMO



	100	1000	5000	10000	50000	100000	125000	175000	200000	225000	275000	300000	325000	375000	400000	425000	500000
Bubrbuja	1.1587E-04	1.1416E-02	2.9315E-01	1.1458E+00	2.8529E+01	1.1473E+02	1.7813E+02	3.4923E+02	4.5963E+02	5.8432E+02	8.8232E+02	1.0521E+03	1.2263E+03	1.6229E+03	1.8575E+03	2.2145E+03	3.0193E+03
Burbuja Optimizada 1	0.0000789	7.7701E-03	1.9623E-01	7.7188E-01	1.9441E+01	7.7430E+01	1.2081E+02	2.3731E+02	3.1019E+02	3.9265E+02	5.8752E+02	7.0019E+02	8.2138E+02	1.0944E+03	1.2450E+03	1.4069E+03	1.9483E+03
Burbuja Optimizada 2	7.9155E-05	1.2170E-02	2.1366E-01	7.8049E-01	1.9358E+01	7.7255E+01	1.2080E+02	2.3686E+02	3.0971E+02	3.9196E+02	5.8621E+02	6.9830E+02	8.2014E+02	1.0925E+03	1.2436E+03	1.4042E+03	1.9448E+03
Inserción	3.2187E-05	2.7008E-03	6.6092E-02	2.8049E-01	6.5697E+00	2.6262E+01	4.0882E+01	8.0089E+01	1.0475E+02	1.3287E+02	1.9848E+02	2.3630E+02	2.7748E+02	3.6941E+02	4.2045E+02	4.7531E+02	6.5860E+02
Selección	4.6015E-05	3.8230E-03	9.8386E-02	3.8234E-01	9.2995E+00	3.7141E+01	5.8101E+01	1.1442E+02	1.4910E+02	1.8901E+02	2.8359E+02	3.3747E+02	3.9659E+02	5.2851E+02	6.0152E+02	6.8019E+02	9.4263E+02
Shell	3.5048E-05	6.2418E-04	5.7921E-03	1.4734E-02	1.2065E-01	3.3611E-01	3.4146E-01	4.7708E-01	7.5258E-01	7.1483E-01	1.0851E+00	1.0610E+00	1.2005E+00	1.5154E+00	1.7913E+00	1.7277E+00	2.3140E+00
Árbol ABB	0.0000260	0.0003090	0.0019112	0.0055540	0.0453269	0.1062210	0.1427851	0.2109311	0.2085521	0.2393138	0.3060570	0.3437331	0.4032440	0.4864581	0.4782321	0.5112391	0.6258950
Mezcla (Merge)	4.1962E-05	3.8910E-04	2.2430E-03	4.9219E-03	2.6945E-02	5.5570E-02	6.9198E-02	9.9983E-02	1.1482E-01	1.2907E-01	1.5869E-01	1.7424E-01	1.9025E-01	2.2126E-01	2.3964E-01	2.5510E-01	3.0100E-01
Ordenamiento rápido (Quicksort)	1.7166E-05	1.8287E-04	9.8181E-04	2.0828E-03	1.3692E-02	2.6301E-02	3.2548E-02	4.5899E-02	5.3125E-02	6.0076E-02	7.5414E-02	8.4167E-02	8.8226E-02	1.0356E-01	1.1112E-01	1.1685E-01	1.3969E-01

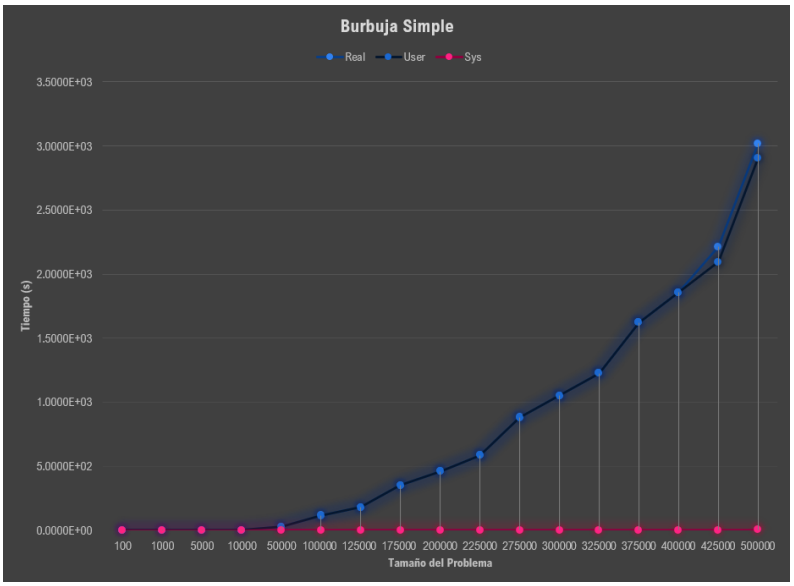
ACTIVIDAD 2

Realizar un análisis temporal para cada algoritmo ordenando los primeros: 100, 1000, 5000, 10000, 50000, 100000, 200000, 400000, 600000, 800000, 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000

Graficar el comportamiento de cada algoritmo

Debido a tiempos de cómputo altos se decidió cambiar los tamaños de problemas a unos más pequeños.

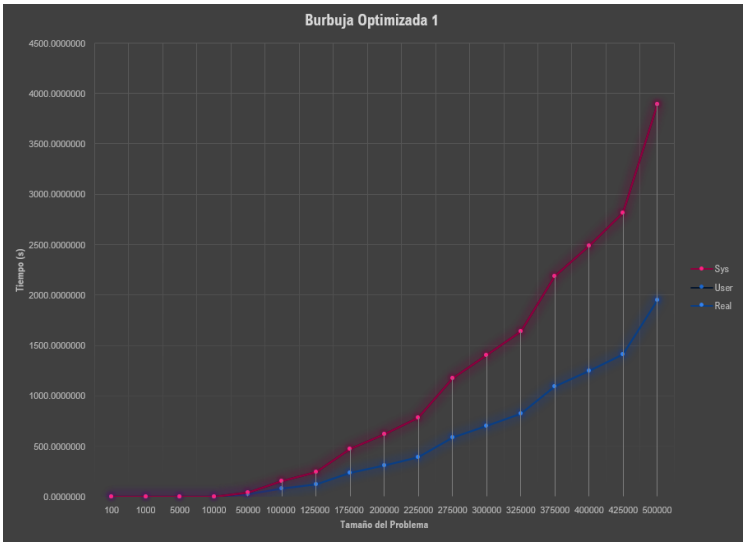
BURBUJA SIMPLE



Burbuja Simple				
n	Real	User	Sys	CPU/Wall
100	1.1587E-04	0.0000E+00	1.2700E-04	1.0960E+02
1,000	1.1416E-02	1.1425E-02	0.0000E+00	1.0008E+02
5,000	2.9315E-01	2.8873E-01	0.0000E+00	9.8493E+01
10,000	1.1458E+00	1.1431E+00	0.0000E+00	9.9762E+01
50,000	2.8529E+01	2.8519E+01	4.4740E-03	9.9982E+01
100000	1.1473E+02	1.1400E+02	3.1998E-02	9.9392E+01
125000	1.7813E+02	1.7801E+02	2.3938E-02	9.9946E+01
175000	3.4923E+02	3.4922E+02	4.8006E-02	1.0001E+02
200,000	4.5963E+02	4.5944E+02	8.7990E-02	9.9977E+01
225,000	5.8432E+02	5.8398E+02	2.4005E-01	9.9984E+01
275,000	8.8232E+02	8.8047E+02	3.2791E-01	9.9828E+01
300,000	1.0521E+03	1.0518E+03	2.3481E-02	9.9982E+01
325,000	1.2263E+03	1.2259E+03	1.1608E-01	9.9981E+01
375,000	1.6229E+03	1.6223E+03	2.9591E-01	9.9979E+01
400,000	1.8575E+03	1.8552E+03	2.7591E-01	9.9892E+01
425,000	2.2145E+03	2.0946E+03	9.6375E-01	9.4630E+01
500,000	3.0193E+03	2.9086E+03	2.7996E+00	9.6426E+01

Para este algoritmo la mejor aproximación es cuando no tomamos en cuenta los valores que se encuentran entre 2000 y 2500, aproximadamente 2300 segundos, ya que esto resulta una pequeña "salida" en lo que es la forma de la curva, pero si observamos, la gráfica sigue correctamente antes y después de los valores 2200 a 2300 segundos. Por otro lado, cuando se encuentra graficando los valores de 4.2 o 4.3, hay un fallo en la gráfica, lo cual concuerda con los valores de los segundos que también afectan a la forma de la curva.

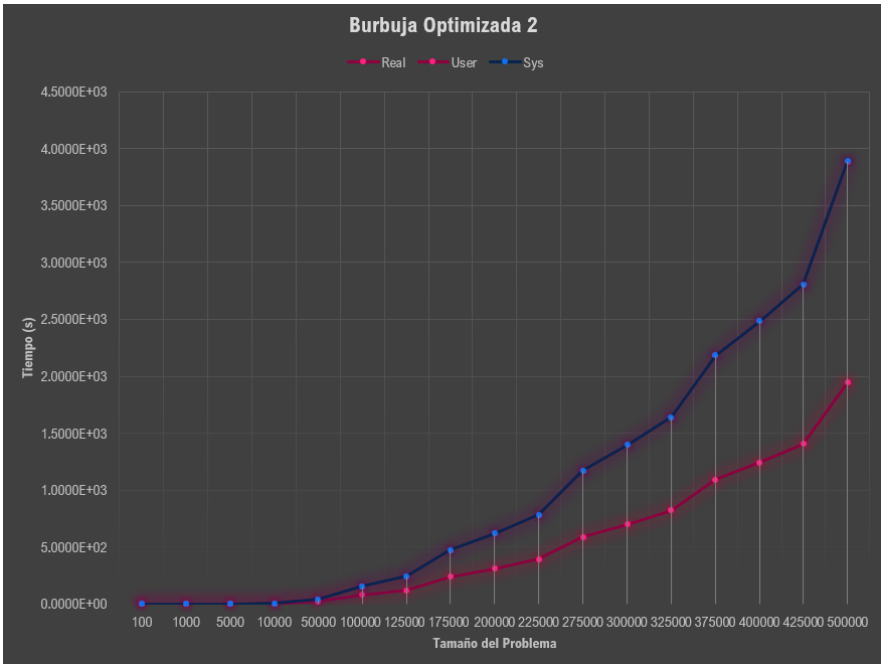
Burbuja Optimizada 1



Burbuja Optimizada				
n	Real	User	Sys	CPU/Wall
100	7.8917E-05	9.0000E-05	0.0000E+00	1.1404E+02
1,000	7.7701E-03	7.7770E-03	0.0000E+00	1.0009E+02
5,000	1.9623E-01	1.9404E-01	0.0000E+00	9.8882E+01
10,000	7.7188E-01	7.7128E-01	0.0000E+00	9.9923E+01
50,000	1.9441E+01	1.9399E+01	0.0000E+00	9.9786E+01
100000	7.7430E+01	7.7358E+01	1.9807E-02	9.9933E+01
125000	1.2081E+02	1.2073E+02	3.6163E-02	9.9971E+01
175000	2.3731E+02	2.3705E+02	9.1931E-02	9.9928E+01
200,000	3.1019E+02	3.0977E+02	2.1594E-01	9.9934E+01
225,000	3.9265E+02	3.9230E+02	2.0813E-01	9.9965E+01
275,000	5.8752E+02	5.8677E+02	3.2793E-01	9.9928E+01
300,000	7.0019E+02	6.9885E+02	4.4035E-01	9.9872E+01
325,000	8.2138E+02	8.2040E+02	4.6784E-01	9.9938E+01
375,000	1.0944E+03	1.0929E+03	6.5195E-01	9.9925E+01
400,000	1.2450E+03	1.2436E+03	8.2379E-01	9.9949E+01
425,000	1.4069E+03	1.4051E+03	7.4380E-01	9.9921E+01
500,000	1.9483E+03	1.9455E+03	1.4436E+00	9.9931E+01

Observando el seguimiento que la gráfica está mostrando, cada uno de los puntos de aproximación, generan una curva muy similar a la otra, con esto podemos decir que todos los puntos de aproximación, a menos de 0 a 500,000 valores, tienen un mismo patrón, es así que, para este algoritmo, cualquier aproximación que utilicemos, nos ayuda a sacar un resultado concreto.

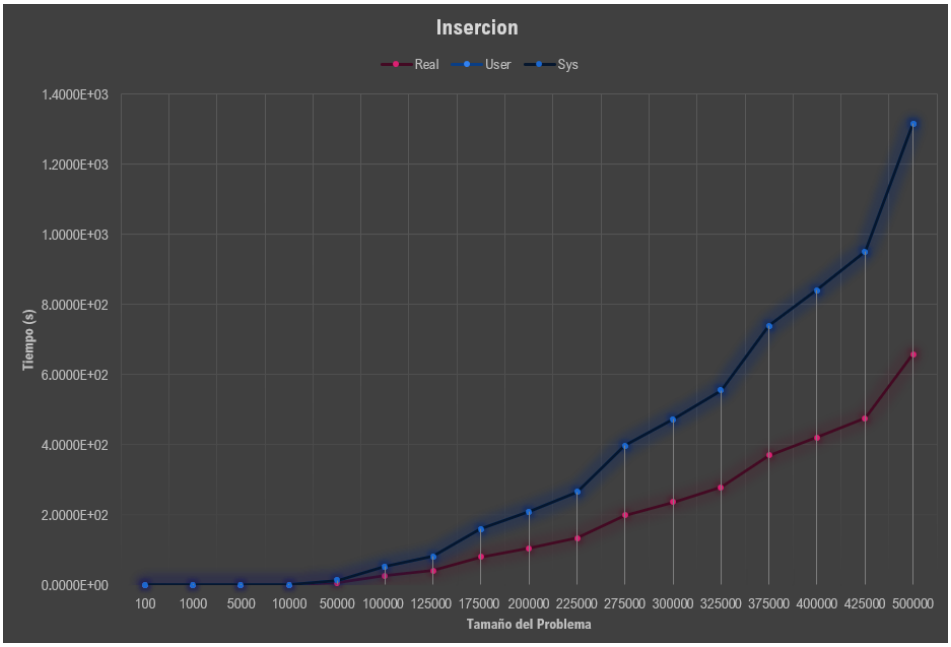
Burbuja Optimizada 2



Burbuja Optimizada 2				
n	Real	User	Sys	CPU/Wall
100	7.9155E-05	9.1000E-05	0.0000E+00	1.1496E+02
1,000	1.2170E-02	7.8330E-03	0.0000E+00	6.4363E+01
5,000	2.1366E-01	1.9541E-01	2.0300E-04	9.1555E+01
10,000	7.8049E-01	7.7001E-01	4.0140E-03	9.9172E+01
50,000	1.9358E+01	1.9336E+01	7.9980E-03	9.9927E+01
100000	7.7255E+01	7.7137E+01	8.7986E-02	9.9961E+01
125000	1.2080E+02	1.2066E+02	4.8467E-02	9.9922E+01
175000	2.3686E+02	2.3668E+02	5.5941E-02	9.9946E+01
200,000	3.0971E+02	3.0933E+02	1.1998E-01	9.9915E+01
225,000	3.9196E+02	3.9166E+02	1.3999E-01	9.9958E+01
275,000	5.8621E+02	5.8590E+02	9.9903E-02	9.9963E+01
300,000	6.9830E+02	6.9765E+02	1.6396E-01	9.9930E+01
325,000	8.2014E+02	8.1929E+02	2.7994E-01	9.9930E+01
375,000	1.0925E+03	1.0914E+03	4.0806E-01	9.9932E+01
400,000	1.2436E+03	1.2422E+03	4.0396E-01	9.9920E+01
425,000	1.4042E+03	1.4028E+03	4.9578E-01	9.9940E+01
500,000	1.9448E+03	1.9429E+03	7.8424E-01	9.9945E+01

Al observar ambas gráficas, sacamos la conclusión de que las aproximaciones que generó el sistema, ya que los puntos se encuentran más cerca a la gráfica que se encuentra a la izquierda, aunque no genere una curva "limpia" tiene mejor similitud a la gráfica que se generó sobre el algoritmo de burbuja optimizada 2.

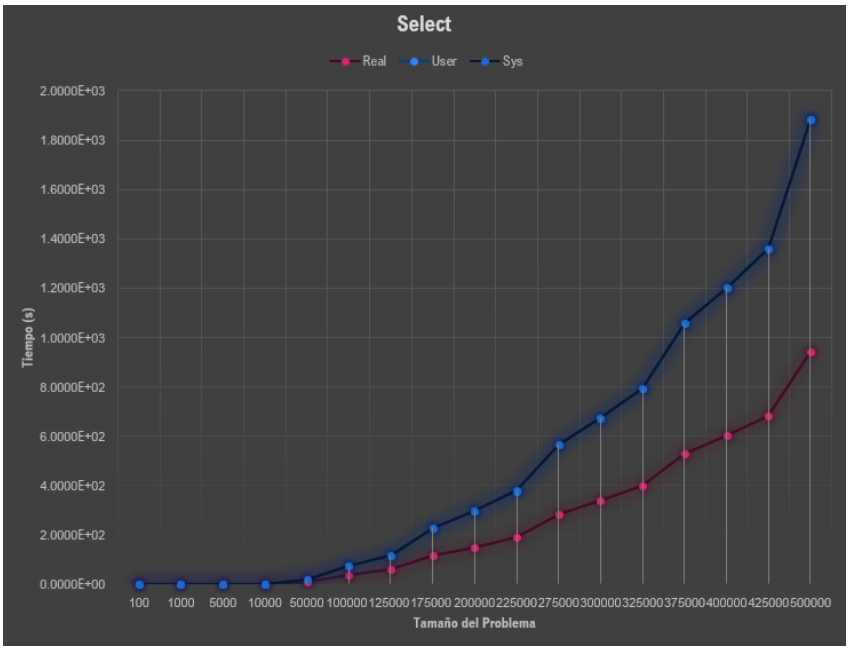
Inserción



Insercion				
n	Real	User	Sys	CPU/Wall
100	3.2187E-05	4.1000E-05	0.0000E+00	1.2738E+02
1,000	2.7008E-03	2.6790E-03	0.0000E+00	9.9193E+01
5,000	6.6092E-02	6.6098E-02	0.0000E+00	1.0001E+02
10,000	2.6049E-01	2.6050E-01	0.0000E+00	1.0000E+02
50,000	6.5697E+00	6.5667E+00	0.0000E+00	9.9955E+01
100000	2.6262E+01	2.6231E+01	2.0800E-04	9.9881E+01
125000	4.0882E+01	4.0859E+01	3.9990E-03	9.9956E+01
175000	8.0089E+01	8.0076E+01	3.9990E-03	9.9989E+01
200,000	1.0475E+02	1.0472E+02	1.0100E-04	9.9980E+01
225,000	1.3287E+02	1.3268E+02	4.7841E-02	9.9894E+01
275,000	1.9848E+02	1.9827E+02	1.5998E-01	9.9975E+01
300,000	2.3630E+02	2.3610E+02	3.9889E-02	9.9935E+01
325,000	2.7748E+02	2.7713E+02	1.2398E-01	9.9921E+01
375,000	3.6941E+02	3.6919E+02	6.0239E-02	9.9957E+01
400,000	4.2045E+02	4.1973E+02	5.0785E-01	9.9949E+01
425,000	4.7531E+02	4.7467E+02	4.6788E-01	9.9963E+01
500,000	6.5860E+02	6.5726E+02	9.2381E-01	9.9936E+01

Al igual que el algoritmo de burbuja y la de selección, las aproximaciones que mejor se adaptan a la curva generada por Matlab, son aquellas que el sistema generó, ya que los puntos que generan la gráfica de color rojo de la derecha, están más distantes de los puntos que genera la curva de azul.

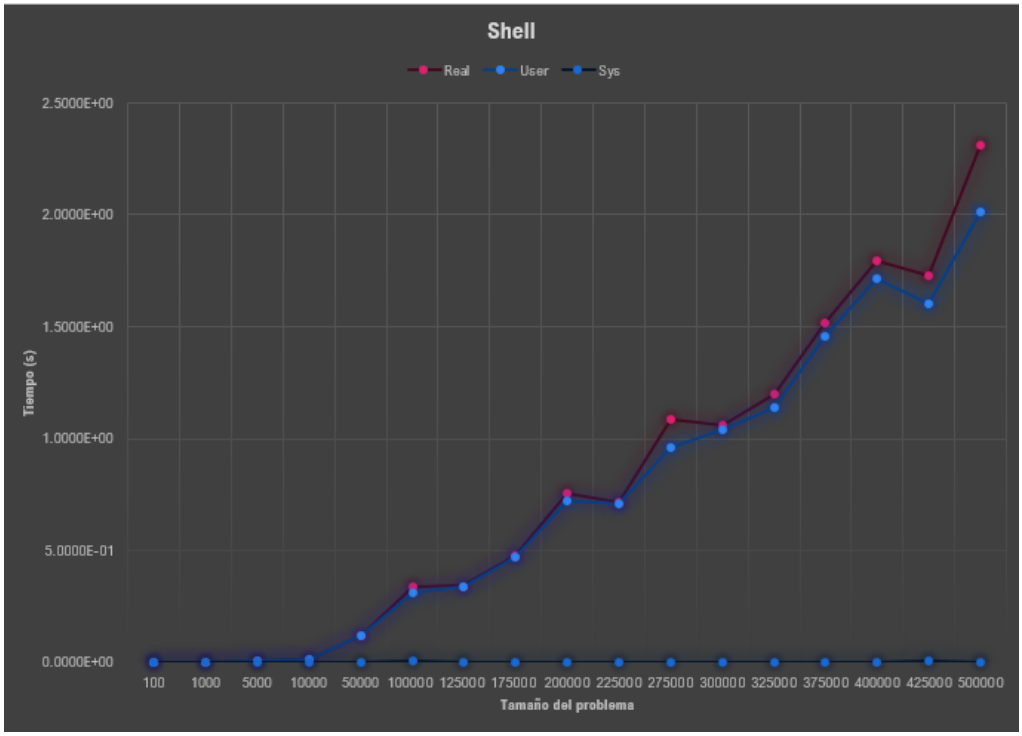
Selección



Selección				
n	Real	User	Sys	CPU/Wall
100	4.6015E-05	5.7000E-05	0.0000E+00	1.2387E+02
1,000	3.8230E-03	3.8370E-03	0.0000E+00	1.0037E+02
5,000	9.8386E-02	9.3284E-02	0.0000E+00	9.4814E+01
10,000	3.8234E-01	3.7419E-01	0.0000E+00	9.7867E+01
50,000	9.2995E+00	9.2973E+00	0.0000E+00	9.9977E+01
100000	3.7141E+01	3.7132E+01	4.2000E-03	9.9985E+01
125000	5.8101E+01	5.8076E+01	7.9990E-03	9.9972E+01
175000	1.1442E+02	1.1413E+02	1.5995E-02	9.9760E+01
200,000	1.4910E+02	1.4906E+02	1.5998E-02	9.9983E+01
225,000	1.8901E+02	1.8882E+02	1.1583E-01	9.9964E+01
275,000	2.8359E+02	2.8319E+02	8.4963E-02	9.9887E+01
300,000	3.3747E+02	3.3713E+02	7.1967E-02	9.9921E+01
325,000	3.9659E+02	3.9620E+02	1.8396E-01	9.9948E+01
375,000	5.2851E+02	5.2801E+02	1.7217E-01	9.9938E+01
400,000	6.0152E+02	6.0116E+02	9.9953E-02	9.9957E+01
425,000	6.8019E+02	6.7950E+02	3.5193E-01	9.9950E+01
500,000	9.4263E+02	9.4157E+02	4.8420E-01	9.9939E+01

Este caso no hay mucha diferencia con los algoritmos de inserción de burbuja optimizada 2, si observamos en la gráfica de la parte derecha con la gráfica de la parte izquierda, la curva azul, muestra una cercanía a los 900 segundos que tiene la gráfica de Matlab, cuando se encuentra en el valor de 500,000, mientras que la curva roja, la cual hace referencia a los tiempos reales, se encuentra mucho más debajo de los segundos que generan la curva de la izquierda.

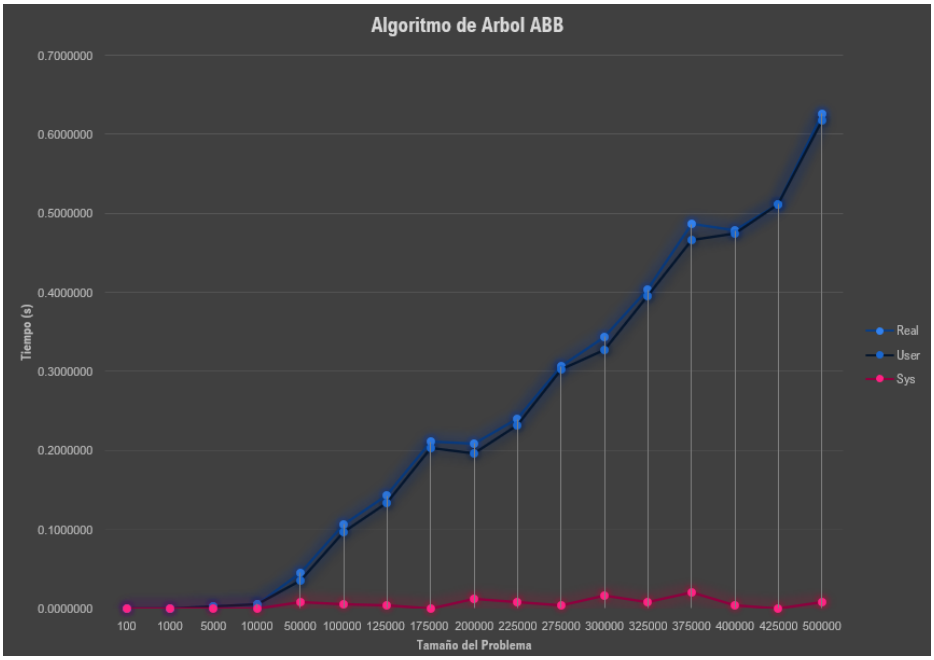
SHELL



shell				
n	Real	User	Sys	CPU/Wall
100	3.5048E-05	0.0000E+00	4.4000E-05	1.2554E+02
1,000	6.2418E-04	6.3400E-04	0.0000E+00	1.0157E+02
5,000	5.7921E-03	5.6810E-03	0.0000E+00	9.8081E+01
10,000	1.4734E-02	1.4579E-02	0.0000E+00	9.8948E+01
50,000	1.2065E-01	1.1992E-01	0.0000E+00	9.9390E+01
100000	3.3611E-01	3.1450E-01	3.8890E-03	9.4726E+01
125000	3.4146E-01	3.4011E-01	3.5700E-04	9.9710E+01
175000	4.7708E-01	4.7025E-01	0.0000E+00	9.8568E+01
200,000	7.5258E-01	7.2506E-01	0.0000E+00	9.6344E+01
225,000	7.1483E-01	7.1031E-01	2.9000E-04	9.9409E+01
275,000	1.0851E+00	9.6299E-01	2.6840E-03	8.8992E+01
300,000	1.0610E+00	1.0417E+00	2.1000E-05	9.8180E+01
325,000	1.2005E+00	1.1373E+00	0.0000E+00	9.4738E+01
375,000	1.5154E+00	1.4537E+00	0.0000E+00	9.5925E+01
400,000	1.7913E+00	1.7143E+00	5.8000E-05	9.5702E+01
425,000	1.7277E+00	1.6041E+00	3.9870E-03	9.3073E+01
500,000	2.3140E+00	2.0114E+00	7.2700E-04	8.6953E+01

La justificación para esta gráfica es un poco más complicada. La curva generada por Matlab, no concuerda con algunas coordenadas, ya que se sale un poco del patrón, si solo tomamos la forma de la curva, pero si nos enfocamos en los puntos "sueltos", la curva generaría unos "picos", los cuales se muestran en las gráficas de color rojo y azul, pero en este caso, las aproximaciones realizadas por el usuario son similares a las aproximaciones de Matlab. Si observamos el punto en el valor 500,000, esta se encuentra con el valor de segundos entre 2 y 2.5, lo cual la gráfica azul (usuario) representa una mejor cercanía.

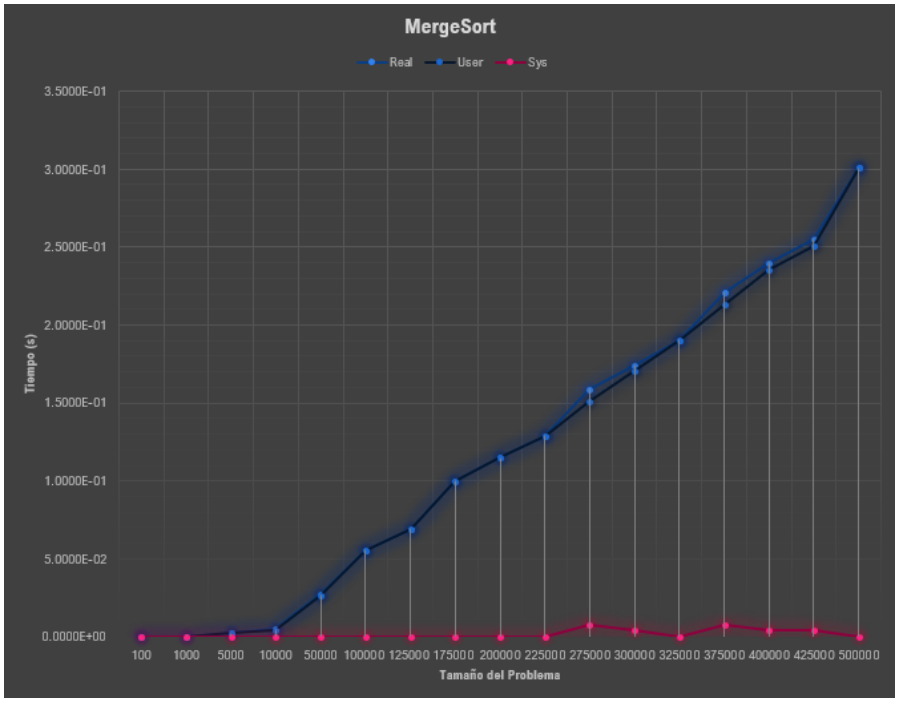
ÁRBOL ABB



Árbol				
n	Real	User	Sys	CPU/Wall
100	2.5988E-05	0.0000E+00	3.7000E-05	1.4238E+02
1,000	3.0899E-04	0.0000E+00	3.2100E-04	1.0389E+02
5,000	1.9112E-03	1.9210E-03	0.0000E+00	1.0051E+02
10,000	5.5540E-03	4.4270E-03	0.0000E+00	7.9709E+01
50,000	4.5327E-02	3.5060E-02	7.8310E-03	9.4626E+01
100000	1.0622E-01	9.5883E-02	5.0860E-03	9.5056E+01
125000	1.4279E-01	1.3341E-01	3.9040E-03	9.6167E+01
175000	2.1093E-01	2.0318E-01	0.0000E+00	9.6323E+01
200,000	2.0855E-01	1.9666E-01	1.1785E-02	9.9947E+01
225,000	2.3931E-01	2.3133E-01	7.9850E-03	1.0000E+02
275,000	3.0606E-01	3.0183E-01	4.2080E-03	9.9992E+01
300,000	3.4373E-01	3.2763E-01	1.5983E-02	9.9966E+01
325,000	4.0324E-01	3.9523E-01	7.9880E-03	9.9994E+01
375,000	4.8646E-01	4.6639E-01	1.9904E-02	9.9967E+01
400,000	4.7823E-01	4.7409E-01	4.0780E-03	9.9986E+01
425,000	5.1124E-01	5.1116E-01	7.9000E-05	9.9999E+01
500,000	6.2590E-01	6.1745E-01	7.9260E-03	9.9917E+01

Estas aproximaciones tienen los mismos efectos que en el algoritmo de Shell, pero en este caso, las aproximaciones realizadas en tiempo real (azul) y por el usuario (negro) son las que más se asemejan a la gráfica realizada por Matlab. Aunque las curvas de la parte de la derecha no tengan un patrón "liso", se pueden observar que en la gráfica de la izquierda hay algunas coordenadas que no siguen la forma de la curva, los cuales deberían de generan ciertos "picos" en la gráfica.

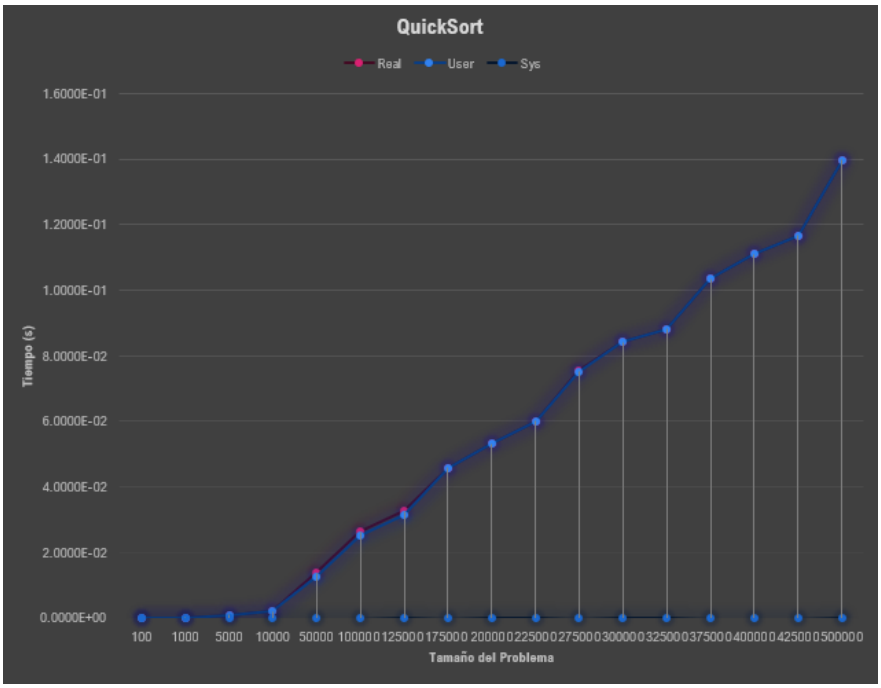
MergeSort



MergeSort				
n	Real	User	Sys	CPU/Wall
100	4.1962E-05	5.2000E-05	0.0000E+00	1.2392E+02
1,000	3.8910E-04	3.9800E-04	0.0000E+00	1.0229E+02
5,000	2.2430E-03	2.2520E-03	0.0000E+00	1.0040E+02
10,000	4.9219E-03	4.7020E-03	0.0000E+00	9.5532E+01
50,000	2.6945E-02	2.6473E-02	0.0000E+00	9.8249E+01
100000	5.5570E-02	5.5534E-02	0.0000E+00	9.9935E+01
125000	6.9198E-02	6.8879E-02	2.9800E-04	9.9970E+01
175000	9.9983E-02	9.9788E-02	1.5700E-04	9.9962E+01
200,000	1.1482E-01	1.1483E-01	0.0000E+00	1.0000E+02
225,000	1.2907E-01	1.2904E-01	0.0000E+00	9.9975E+01
275,000	1.5869E-01	1.5070E-01	7.9900E-03	9.9998E+01
300,000	1.7424E-01	1.7020E-01	3.9780E-03	9.9964E+01
325,000	1.9025E-01	1.9018E-01	0.0000E+00	9.9960E+01
375,000	2.2126E-01	2.1329E-01	7.9680E-03	1.0000E+02
400,000	2.3964E-01	2.3545E-01	4.0000E-03	9.9918E+01
425,000	2.5510E-01	2.5089E-01	4.1900E-03	9.9994E+01
500,000	3.0100E-01	3.0062E-01	6.5000E-05	9.9894E+01

Las aproximaciones realizadas por tiempo real y por el usuario, son más cercanas a las aproximaciones que se realizaron por Matlab, aunque, retomando las coordenadas que se encuentran fuera de la curva, son similares las curvas tanto de la derecha, como las de la izquierda.

QuickSort



QuickSort				
n	Real	User	Sys	CPU/Wall
100	1.7166E-05	2.7000E-05	0.0000E+00	1.5729E+02
1,000	1.8287E-04	1.9200E-04	0.0000E+00	1.0499E+02
5,000	9.8181E-04	9.8900E-04	0.0000E+00	1.0073E+02
10,000	2.0828E-03	2.0950E-03	0.0000E+00	1.0058E+02
50,000	1.3692E-02	1.2626E-02	0.0000E+00	9.2213E+01
100000	2.6301E-02	2.5324E-02	0.0000E+00	9.6286E+01
125000	3.2548E-02	3.1555E-02	7.8000E-05	9.7189E+01
175000	4.5899E-02	4.5905E-02	0.0000E+00	1.0001E+02
200,000	5.3125E-02	5.3078E-02	5.3000E-05	1.0001E+02
225,000	6.0076E-02	6.0046E-02	3.8000E-05	1.0001E+02
275,000	7.5414E-02	7.5143E-02	0.0000E+00	9.9641E+01
300,000	8.4167E-02	8.4124E-02	3.8000E-05	9.9994E+01
325,000	8.8226E-02	8.8188E-02	4.0000E-05	1.0000E+02
375,000	1.0356E-01	1.0357E-01	0.0000E+00	1.0000E+02
400,000	1.1112E-01	1.1111E-01	0.0000E+00	9.9996E+01
425,000	1.1665E-01	1.1660E-01	0.0000E+00	9.9952E+01
500,000	1.3969E-01	1.3966E-01	3.3000E-05	1.0000E+02

Tal vez uno de los algoritmos que mejor se adapta a la gráfica de Matlab. Con base a las gráficas, podemos observar que las aproximaciones en tiempo real y del usuario, casi se unen a excepción de un punto, sin embargo, la diferencia es mínima y a la hora de compararlos con la gráfica de la izquierda, no tiene tantas fallas como en los algoritmos anteriores. Como conclusión, podemos analizar que el algoritmo de quick sort, aunque es la más complicada a comparación de los otros algoritmos, tiene mucha más exactitud.

FUNCIONES DE COMPLEJIDAD TEMPORAL ENCONTRADAS

Burbuja Simple

$$f(x) = 34 \times 10^{-16}x^3 + 101 \times 10^{-10}x^2 + 123 \times 10^{-6}x^2 - 0.9982$$

Burbuja Optimizada 1

$$f(x) = 78 \times 10^{-10}x^2 - 13 \times 10^{-6}x + 0.2063$$

Burbuja Optimizada 2

$$f(x) = 1 \times 10^{-16}x^3 + 77 \times 10^{-10}x^2 - 79 \times 10^{-7}x + 0.1027$$

Inserción

$$f(x) = 26 \times 10^{-10}x^2 - 57 \times 10^{-7}x + 0.1231$$

Selección

$$f(x) = 37 \times 10^{-10}x^2 - 14 \times 10^{-6}x + 0.2192$$

Shell

$$f(x) = 926.222205 \ln x - 0.000011$$

Árbol ABB

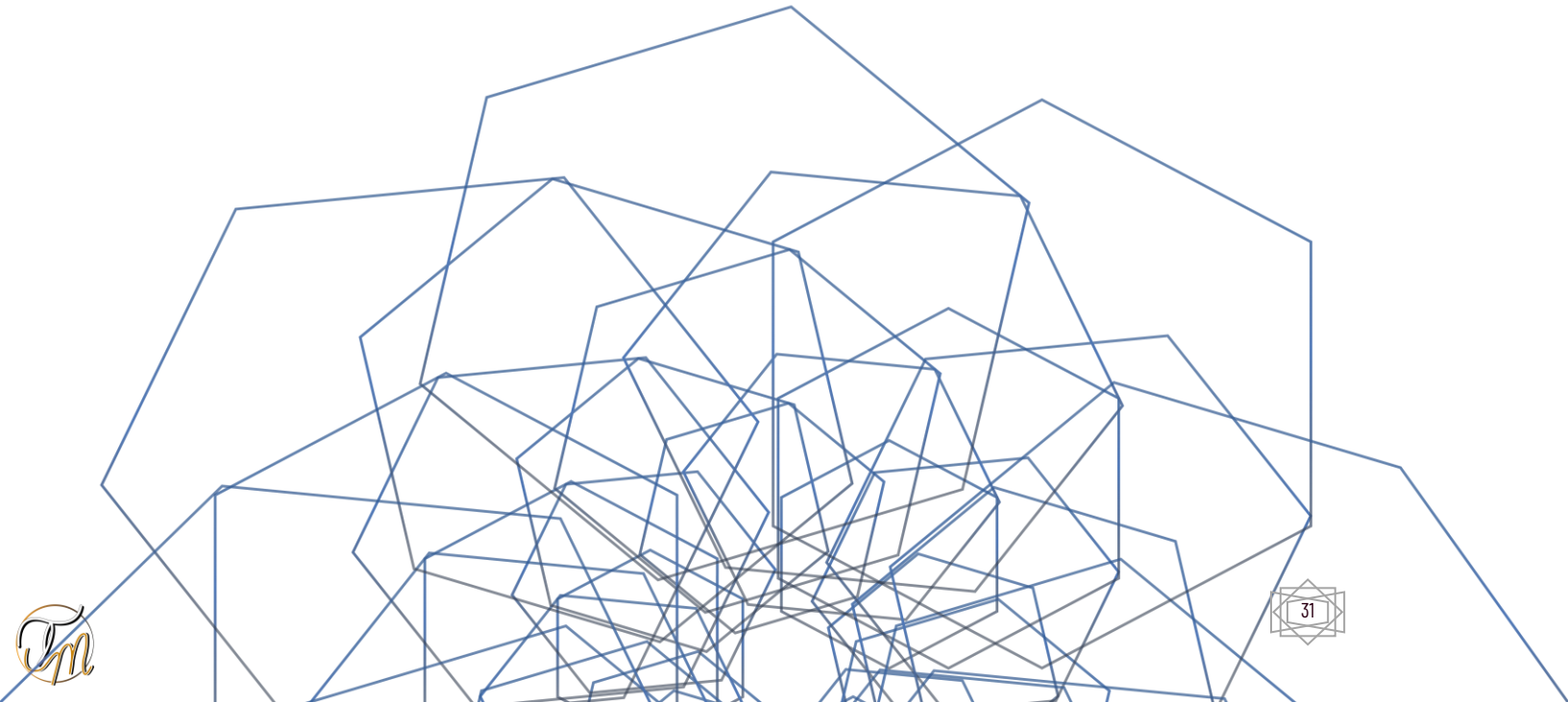
$$f(x) = -21.431314 \ln x + 6.552377$$

Merge Sort

$$f(x) = 0.000141 \ln x - 0.005888$$

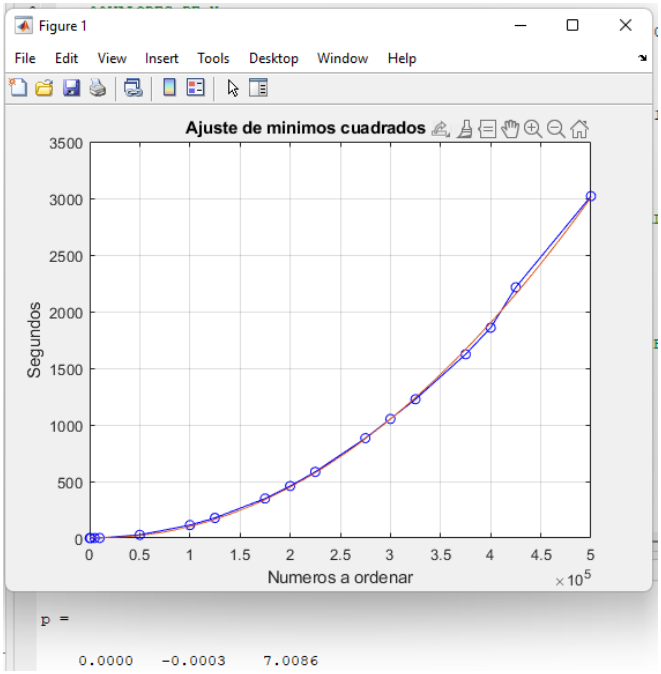
Quick Sort

$$f(x) = -0.346912 \ln x - 0.002619$$

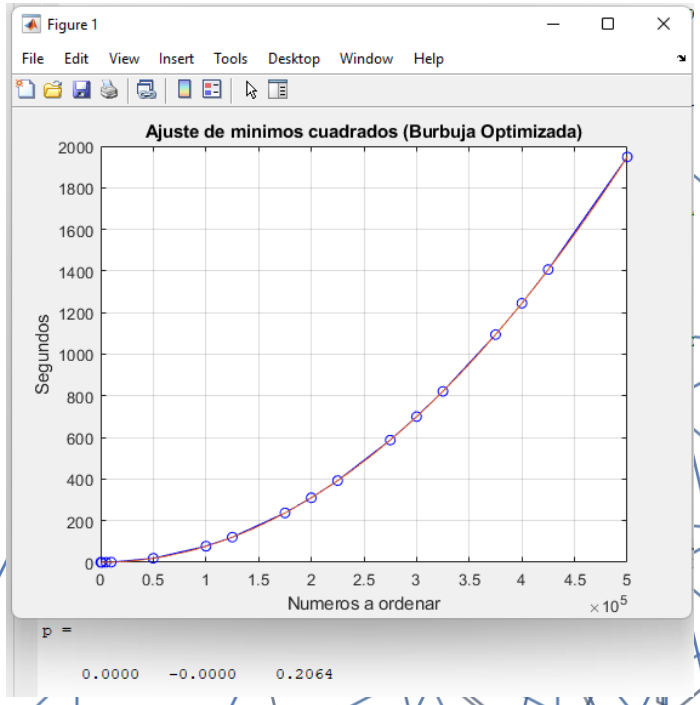


APROXIMACIÓN A CURVAS

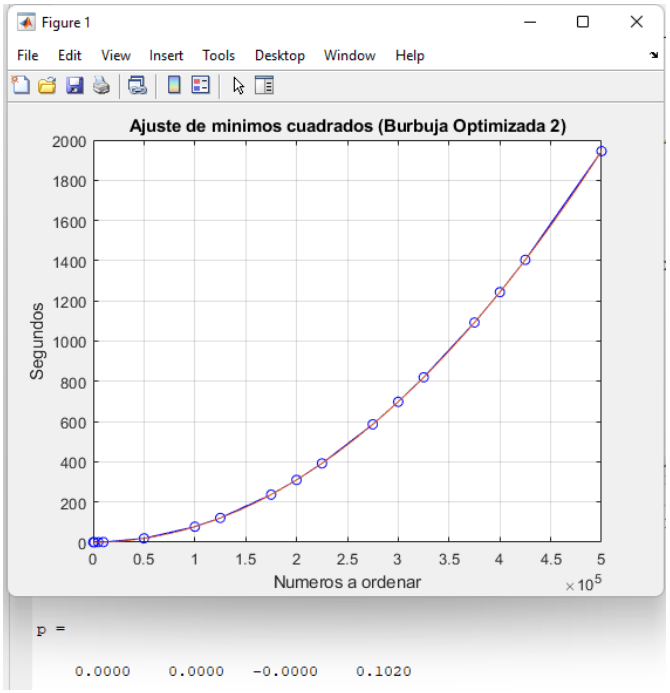
BURBUJA SIMPLE



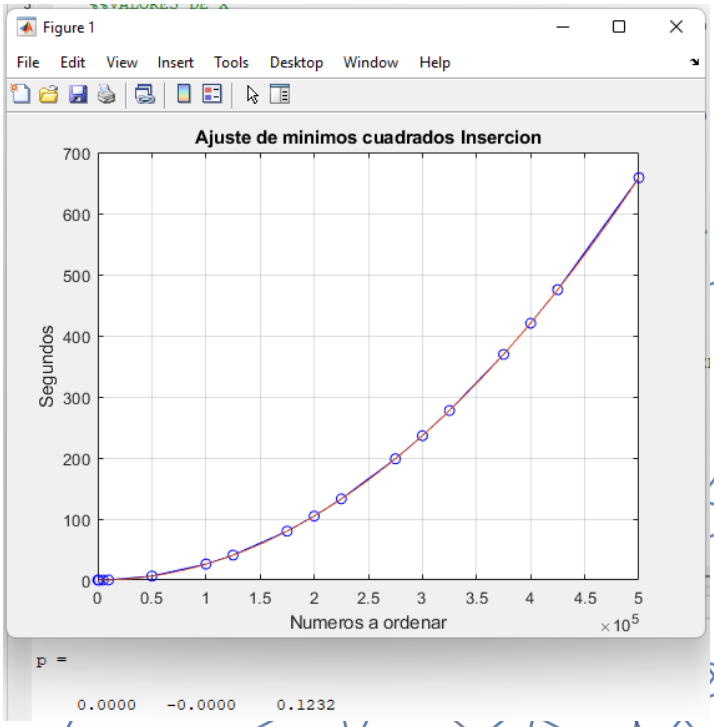
BURBUJA OPTIMIZADA 1



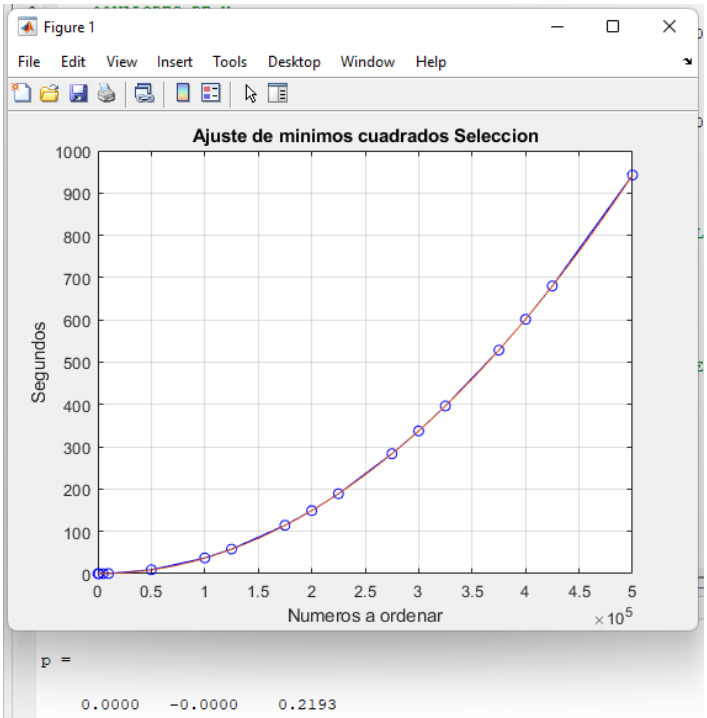
Burbuja Optimizada 2



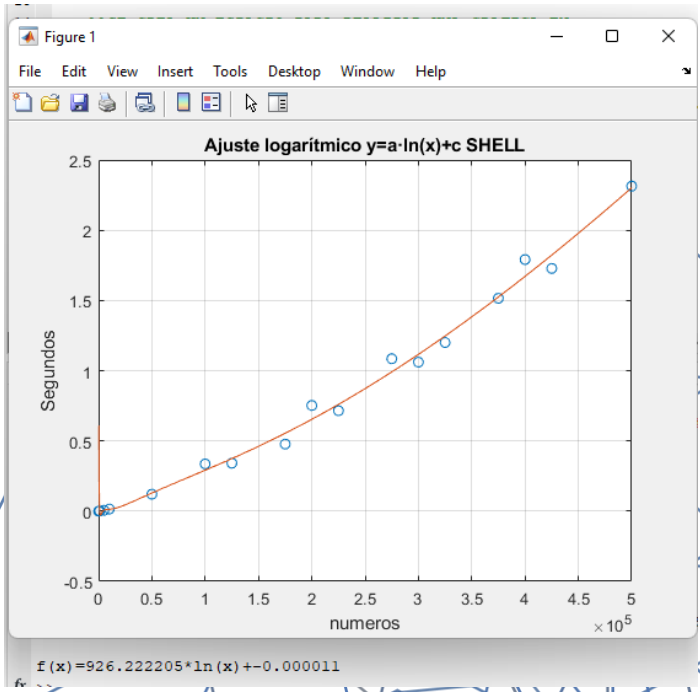
Inserción



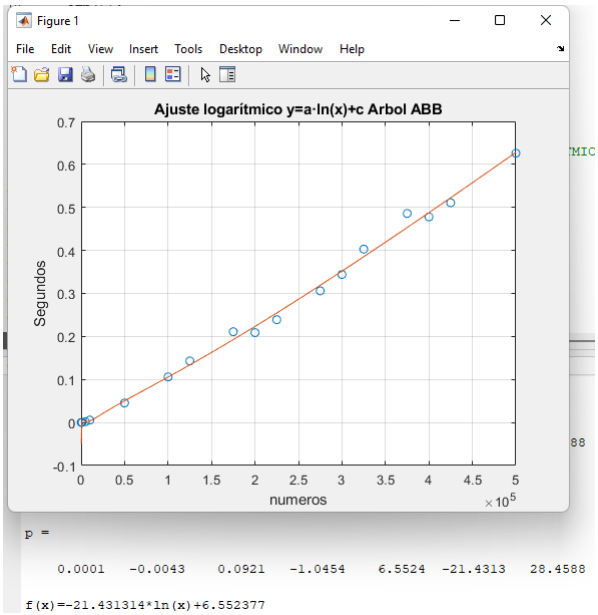
SELECCIÓN



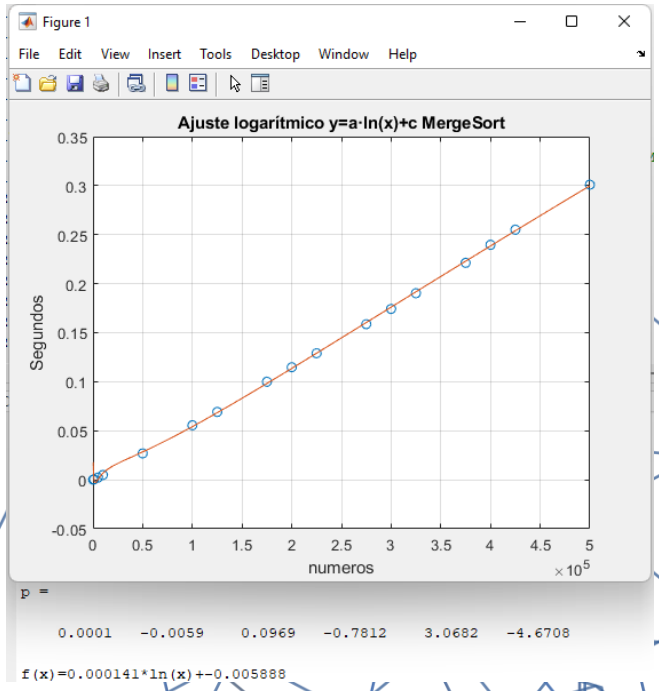
SHELL



ÁRBOL ABB



MERGESORT



QuickSort

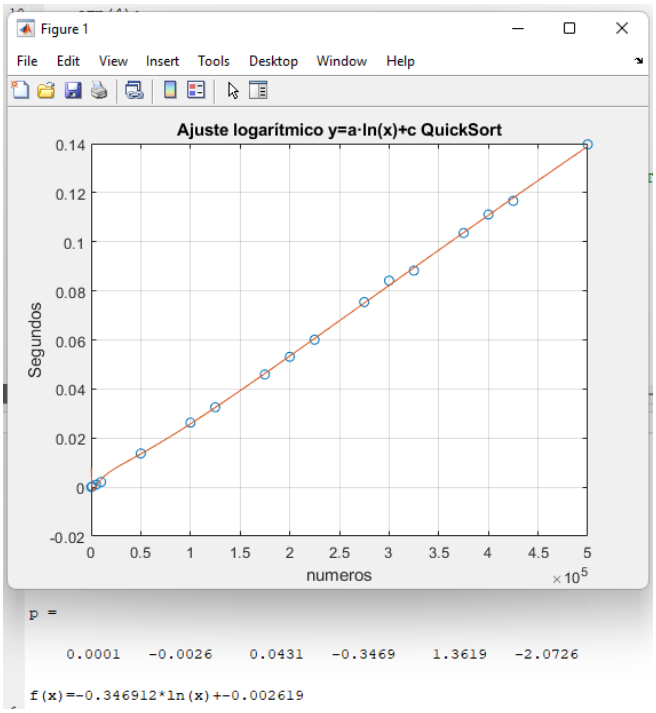
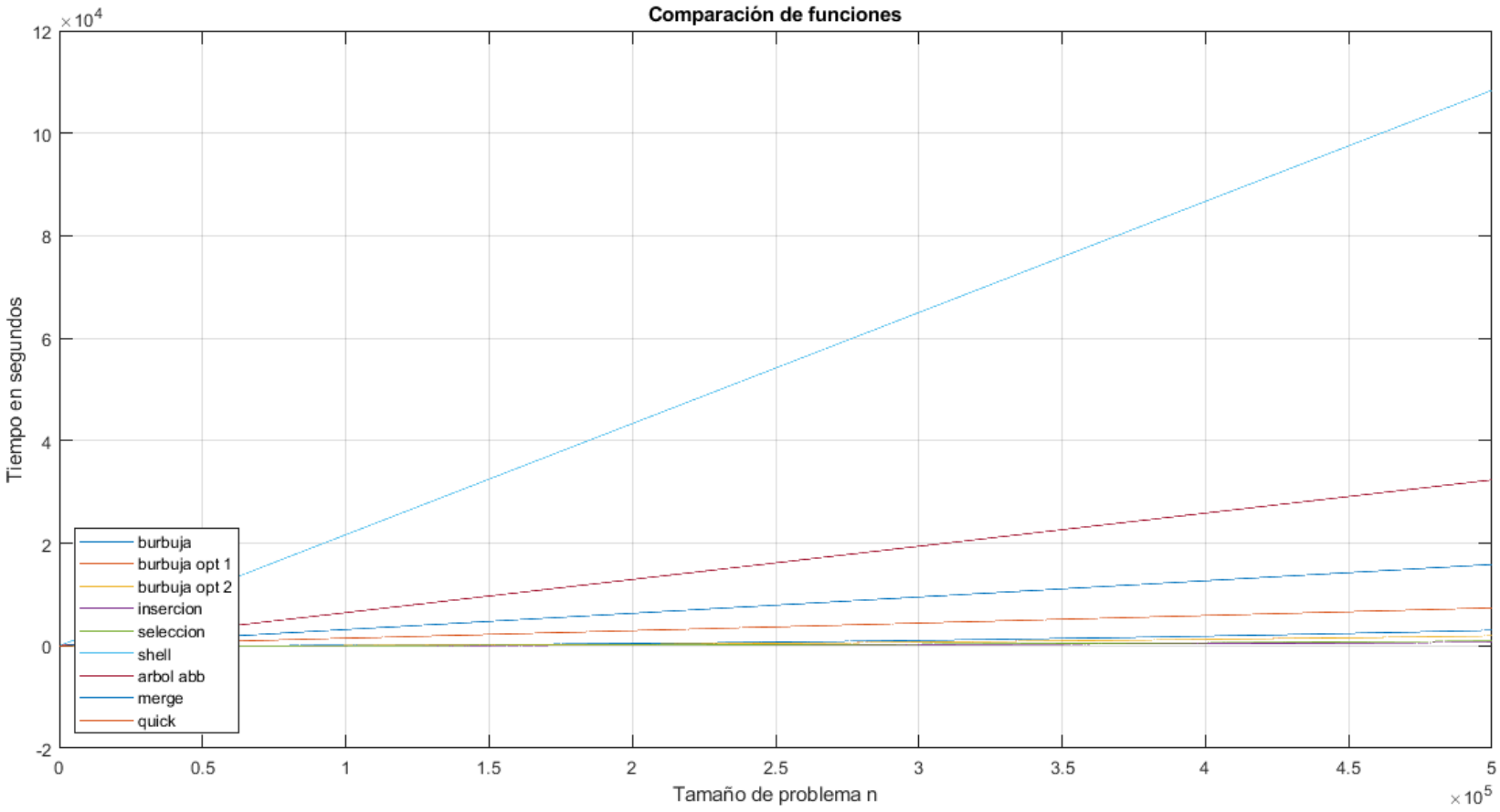


Figure 1
File Edit View Insert Tools Desktop Window Help
Ajuste logarítmico y=a·ln(x)+c QuickSort
Segundos
numeros
×10^5
p =
0.0001 -0.0026 0.0431 -0.3469 1.3619 -2.0726
f(x)=-0.346912*ln(x)+-0.002619

GRÁFICA COMPARATIVA ENTRE LOS 9 ALGORITMOS



Código en MatLab generado:

```
clc
%%X's:
x=[100 1000 5000 10000 50000 100000 125000 175000 200000 225000 275000
300000 325000 375000 400000 425000 500000];
%%Valores Y burbuja:
y1=[1.1587E-04 1.1416E-02 2.9315E-01 1.1458E+00 2.8529E+01 1.1473E+02
1.7813E+02 3.4923E+02 4.5963E+02 5.8432E+02 8.8232E+02 1.0521E+03
1.2263E+03 1.6229E+03 1.8575E+03 2.2145E+03 3.0193E+03];
%%Valores Y burbuja opt 1
y2=[7.8917E-05 7.7701E-03 1.9623E-01 7.7188E-01 1.9441E+01 7.7430E+01
1.2081E+02 2.3731E+02 3.1019E+02 3.9265E+02 5.8752E+02 7.0019E+02
8.2138E+02 1.0944E+03 1.2450E+03 1.4069E+03 1.9483E+03];
%%Valores Y burbuja opt 2
y3=[0.000079155 0.012170076 0.213660002 0.780486107 19.35835981
77.25503421 120.8048251 236.8600502 309.7122211 391.9615159 586.2148209
698.3011532 820.1439228 1092.517701 1243.641719 1404.164403
1944.795895];
%%Valores Y Insercion
y4=[3.2187E-05 2.7008E-03 6.6092E-02 2.6049E-01 6.5697E+00 2.6262E+01
4.0882E+01 8.0089E+01 1.0475E+02 1.3287E+02 1.9848E+02 2.3630E+02
2.7748E+02 3.6941E+02 4.2045E+02 4.7531E+02 6.5860E+02];
%%Valores Y Seleccion
y5=[4.6015E-05 3.8230E-03 9.8386E-02 3.8234E-01 9.2995E+00 3.7141E+01
5.8101E+01 1.1442E+02 1.4910E+02 1.8901E+02 2.8359E+02 3.3747E+02
3.9659E+02 5.2851E+02 6.0152E+02 6.8019E+02 9.4263E+02];
%%Valores Y Shell
y6=[3.5048E-05 6.2418E-04 5.7921E-03 1.4734E-02 1.2065E-01 3.3611E-01
3.4146E-01 4.7708E-01 7.5258E-01 7.1483E-01 1.0851E+00 1.0610E+00
1.2005E+00 1.5154E+00 1.7913E+00 1.7277E+00 2.3140E+00];
%%Valores Y Arbol ABB
y7=[2.60E-05 3.09E-04 1.91E-03 5.55E-03 4.53E-02 1.06E-01 1.43E-01
2.11E-01 2.09E-01 2.39E-01 3.06E-01 3.44E-01 4.03E-01 4.86E-01 4.78E-01
5.11E-01 6.26E-01];
%%Valores Y Merge
y8=[4.1962E-05 3.8910E-04 2.2430E-03 4.9219E-03 2.6945E-02 5.5570E-02
6.9198E-02 9.9983E-02 1.1482E-01 1.2907E-01 1.5869E-01 1.7424E-01
1.9025E-01 2.2126E-01 2.3964E-01 2.5510E-01 3.0100E-01];
%%Valores Y Quick
y9=[1.7166E-05 1.8287E-04 9.8181E-04 2.0828E-03 1.3692E-02 2.6301E-02
3.2548E-02 4.5899E-02 5.3125E-02 6.0076E-02 7.5414E-02 8.4167E-02
8.8226E-02 1.0356E-01 1.1112E-01 1.1665E-01 1.3969E-01];

%%Grados
n1=3;
n2=2;
n3=3;
n4=2;
n5=2;

%%polinomios
p1 = polyfit(x, y1, n1);
p2 = polyfit(x, y2, n2);
p3 = polyfit(x, y3, n3);
p4 = polyfit(x, y4, n4);
p5 = polyfit(x, y5, n5);
```

```
%%logaritmos
p6 = polyfit(log(x),y6,1);
p7 = polyfit(log(x),y7,1);
p8 = polyfit(log(x),y8,1);
p9 = polyfit(log(x),y9,1);

%%espacio de puntos
xi = linspace(0,500000,10000);
z1 = polyval(p1, xi);
z2 = polyval(p2, xi);
z3 = polyval(p3, xi);
z4 = polyval(p4, xi);
z5 = polyval(p5, xi);
z6 = polyval(p6, xi);
z7 = polyval(p7, xi);
z8 = polyval(p8, xi);
z9 = polyval(p9, xi);

%%grafica
figure(1);
plot(xi,z1,'-',xi,z2,'-',xi,z3,'-',xi,z4,'-',xi,z5,'-',xi,z6,'-
',xi,z7,'-',xi,z8,'-',xi,z9,'-'),grid;
ylabel('Tiempo en segundos');
xlabel('Tamaño de problema n');
legend({'burbuja','burbuja opt 1','burbuja opt
2','insercion','seleccion','shell','arbol
abb','merge','quick'},'Location','southwest')
title('Comparación de funciones');
```

APROXIMACIONES POR ALGORITMO CON SU FUNCION

Burbuja

Burbuja Simple					
N	15000000	20000000	50000000	100000000	500000000
Resultado	27677272499.7668	49204040000.3618	307525250007.5018	1230101000033.0017	3.07525 E13

Burbuja Optimizada 1

Burbuja Optimizada 1					
N	15000000	20000000	50000000	100000000	500000000
Resultado	1754805.2063	3119740.2063	19499350.2063	77998700.2063	1949993500.2063

Burbuja Optimizada 2

Burbuja Optimizada 2					
N	15000000	20000000	50000000	100000000	500000000
Resultado	2069881.6027Si	3879842.1027	31749605.1027	176999210.1027	14424996050.1027

Inserción

Inserción					
N	15000000	20000000	50000000	100000000	500000000
Resultado	584914.6231	1039886.1231	6499715.1231	25999430.1231	649997150.1231

Selección

Selección					
N	15000000	20000000	50000000	100000000	500000000
Resultado	832290.2192	1479720.2192	9249300.2192	36998600.2192	924993000.2192

Shell

Shell					
N	15000000	20000000	50000000	100000000	500000000
Resultado	15304.48878	15570.94630	16419.63512	17061.64343	18552.34055

Árbol

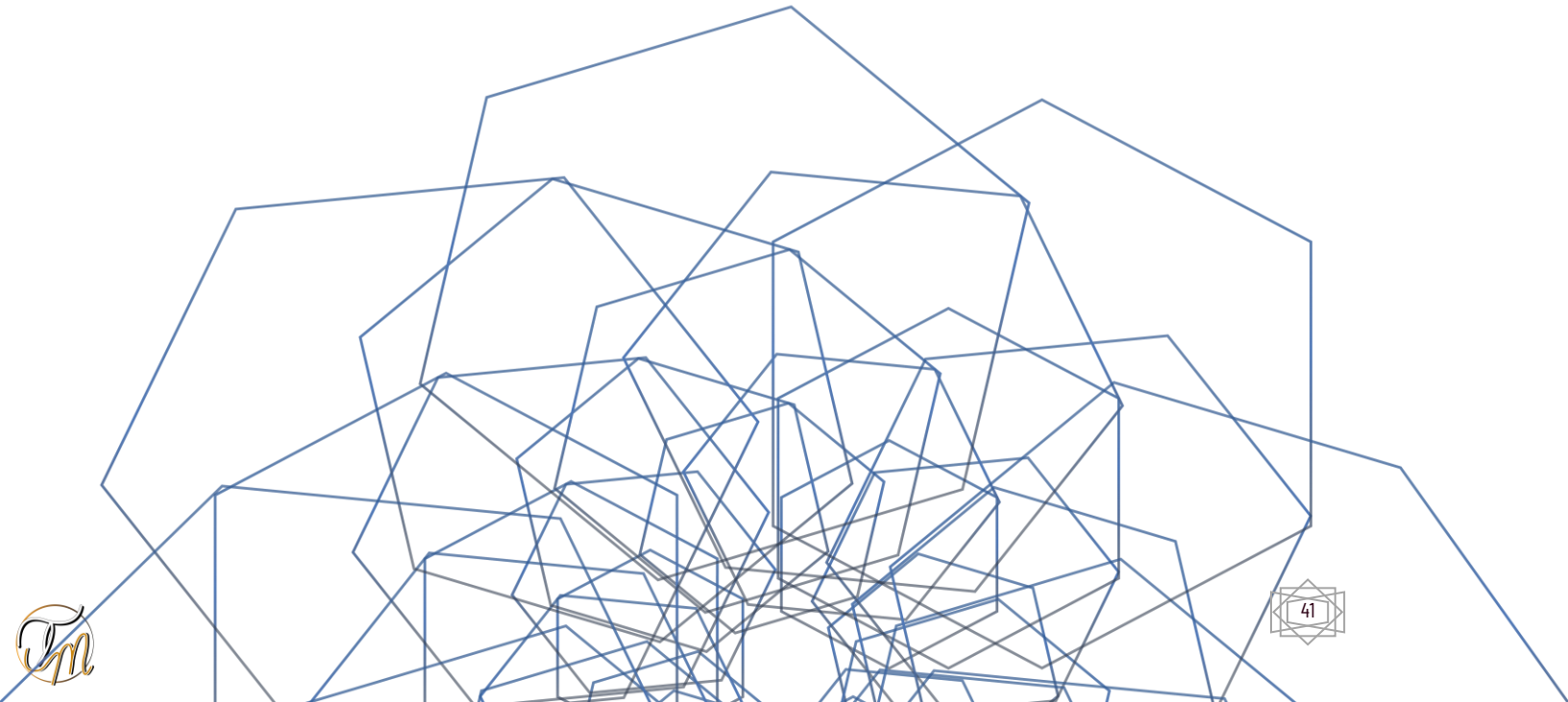
Árbol ABB					
N	15000000	20000000	50000000	100000000	500000000
Resultado	347.56924	353.73464	373.37196	388.22701	422.71938

MergeSort

Merge Sort					
N	15000000	20000000	50000000	100000000	500000000
Resultado	0.00357	0.00353	0.00340	0.00330	0.00308

QuickSort

Quick Sort					
N	15000000	20000000	50000000	100000000	500000000
Resultado	5.73464	5.83443	6.15230	6.39275	6.95106



CUESTIONARIO

- ¿Cuál de los 8 algoritmos es más fácil de implementar?

Desde mi perspectiva fue Shell seguido de la versión de QuickSort implementada en este trabajo

- ¿Cuál de los 8 algoritmos es el más difícil de implementar?

Árbol, pues conlleva una complejidad un poca mas avanzada, aparte de conocimientos acerca de árboles de cierta manera bien establecidos para poder saber como realizar los ordenamientos de forma correcta, así como un cierto nivel de abstracción para comprender como es la manera en que esta operando el algoritmo y como es que se debe ir recorriendo el árbol y sus ramas.

- ¿Cuál algoritmo tiene menor complejidad temporal?

Shell

- ¿Cuál algoritmo tiene mayor complejidad temporal?

Inserción

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

Desde mi punto de vista ese sería el árbol debido a que solo ocupara una n cantidad de nodos, los cuales dependen del tamaño del arreglo

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

Burbuja Optimizada y Shell dado que están basados en arreglo de A[n] proporcionando n y sus otras variables

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

Si, 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

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

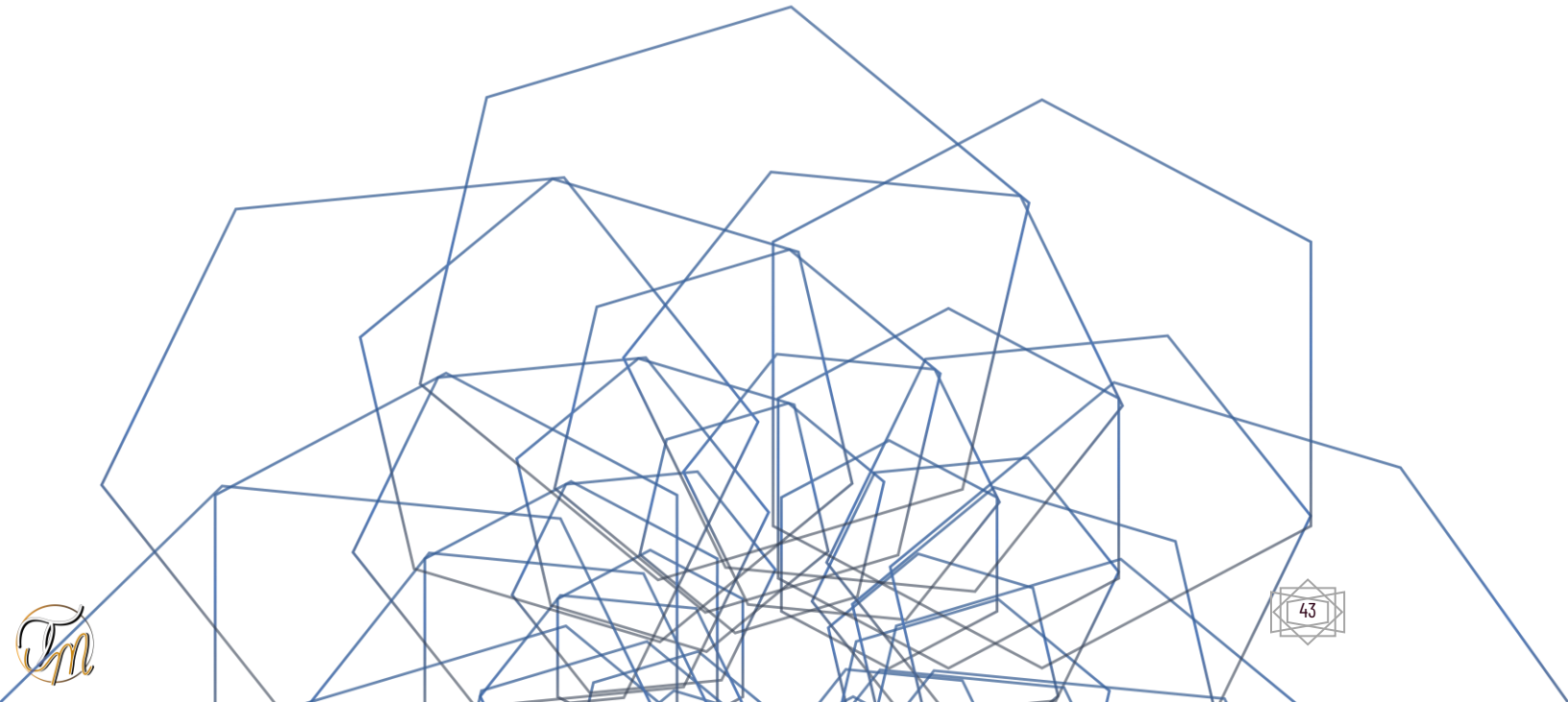
El único factor que fue controlado fue la métrica de que serían sometidos a prueba en el mismo equipo de computo

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

Si, de una forma bastante eficiente de hecho, pues proporciono mayor velocidad al tiempo de realizar las ejecuciones pertinentes para cada algoritmo, sin la necesidad de ir ejecutando uno a uno o ir modificando los valores que se tendrían que tomar (cantidad de elementos del archivo o la opción del menú)

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

Que realmente se tomen un tiempo pertinente para poder analizar cada algoritmo, pues comprenderlos facilita muchísimo la implementación de los mismos, así como el análisis que se tiene que realizar acerca de cada uno de ellos



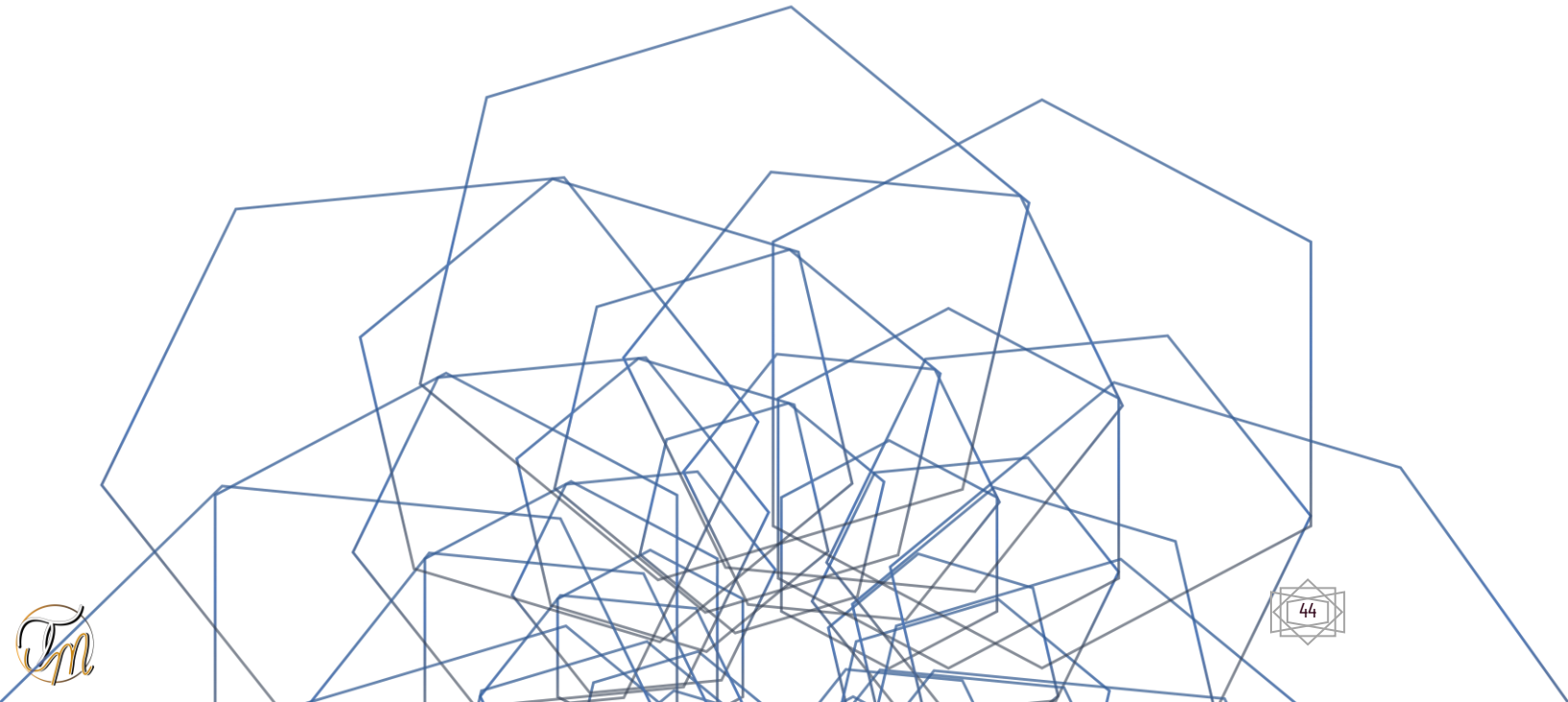
ERRORES DETECTADOS

Como tal no existió la presencia de algún error como tal, solo simples fallas en la lógica de algunos de los seudocódigos que se nos fueron proporcionados, aunque nada que no pudiese ser detectado de una forma sencilla.

Aunque cabe destacar que al inicio de la implementación del código con la forma en la que lo hicimos notamos que era posible que la persona que quisiera ejecutar el programa no tuviera el conocimiento de cómo hacerlo (ya que nosotros solicitamos que al momento de ser ejecutado el programa se manden los 2 parámetros necesarios de opción de ordenamiento y cantidad de elementos a ordenar), por lo cual en caso de que no manden los parámetros solicitamos que elijan una de las opciones mediante consola y definimos una cantidad de elementos en específico (100 para ser exactos)

¿OPTIMIZACIONES?

Me parece que los algoritmos funcionan bastante bien y no dudo que se les pudiese implementar algún tipo de mejora, pero considero que eso conlleva a un nivel de experiencia mucho más alto, así como mayores conocimientos (que, desde mi perspectiva, aun no poseo), o tal vez los algoritmos ya están establecidos de esta forma y no han sido modificados por que han llegado a su mejor rendimiento.



CONCLUSIONES

Antonio:

Me parece que la practica ha sido una muy buena forma de ponernos a analizar las cosas y el funcionamiento de los algoritmos que se nos solicito implementar, pues a pesar de que se nos proporcionaron los seudocódigos estos contenían algunos errores que provocaban un funcionamiento erróneo, lo cual no esta para nada mal, pues uno mismo se debe fijar como es que funciona primero el algoritmo para después poderlo llevar a una implementación práctica de código, personalmente opte por hacerlo a mano e ir probando con arreglo y valores pequeños para poder comprender como es que se realiza el ordenamiento del arreglo en cuestión.

Oscar:

Mediante esta práctica pude percatarme por primera vez de cuánto le "cuesta" a nuestra computadora ejecutar un código, pues comúnmente no nos preocupamos por cuántas veces se repita una iteración o cuántas operaciones se hacen en una sola línea de código porque ya estamos acostumbrados a que nuestros ordenadores soporten trabajos de carga muy superiores a un ciclo for doble, sin embargo me percaté que es importantísimo prestar atención a nuestras implementaciones de solución ya que posiblemente en el momento de su elaboración solo las estemos diseñando para un tamaño de problema controlado, es decir, que sabemos de ante mano que en nuestros casos de prueba se ejecutará sin problemas, pero no nos preguntamos qué pasaría si aumentamos el tamaño de problema, y muy posiblemente en todos los casos resulte que nuestro código no está optimizado para tamaños superiores a lo planeado. Es por esta razón que aprendí a poner empeño de ahora en adelante a que mis soluciones no se centren en tamaños específicos y que sean lo más optimizadas posibles para los casos de mayor tamaño.

Mariana:

Mas que nada con esta practica llegue a la comprensión respectiva a los algoritmos de ordenamiento y las operaciones que estos realizan como es el caso de la orden, la búsqueda y mezcla, basándonos principalmente en el criterio que se le da a este, además de que consideramos que un factor importante en el algoritmo de búsqueda o clasificación es su complejidad computacional y cómo se depende de la cantidad de datos a procesar, por lo que finalmente esta práctica nos mostró de una forma explicativa la importancia de la optimización de los algoritmos para números grandes de datos.

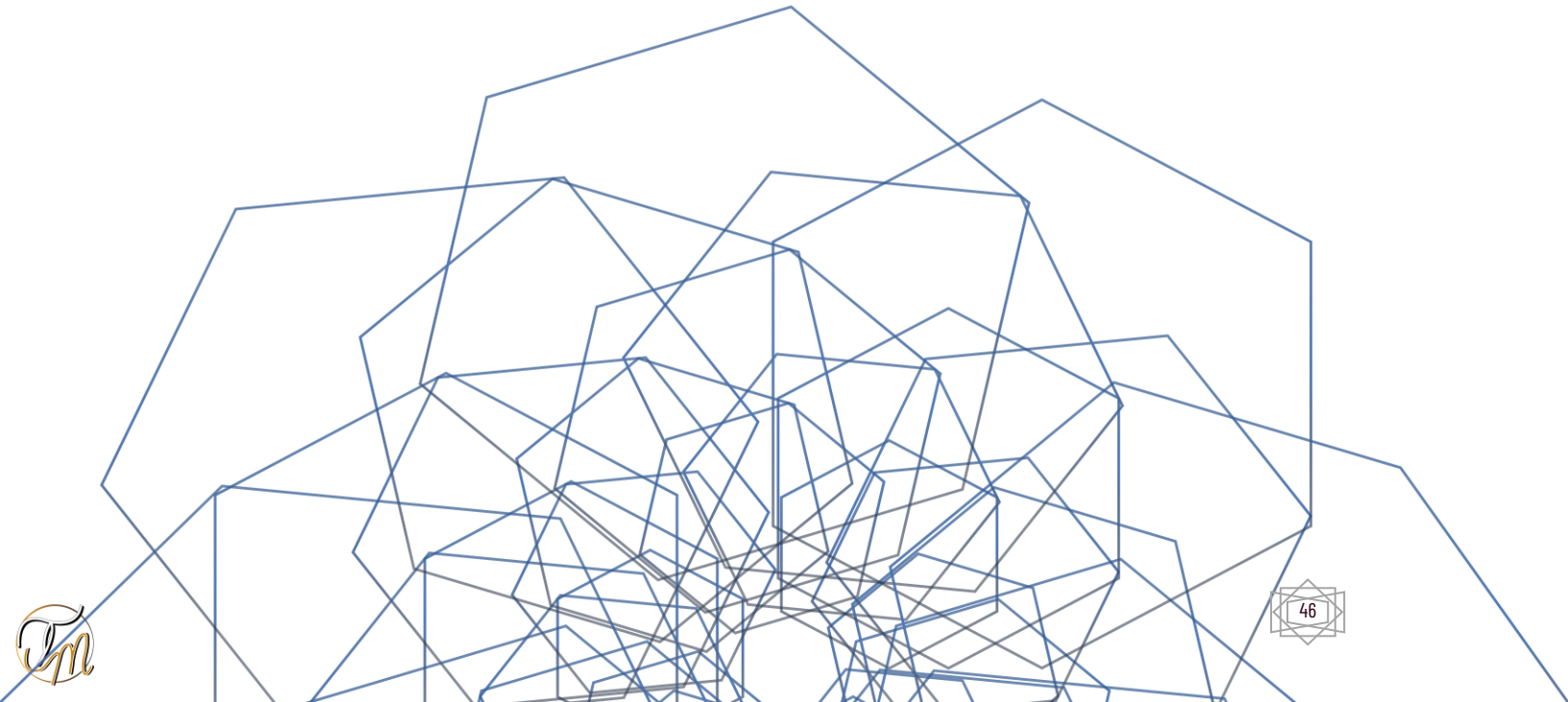
Paola:

A través de esta práctica pudimos observar cómo funcionan los diferentes algoritmos de ordenamiento, y aunque cada una tenga el mismo enfoque, todas tienen sus formas de resolver el problema de ordenamiento, por ejemplo, tomando en cuenta el algoritmo de ordenamiento de burbuja, trabajamos con 3 diferentes funciones que tienen el mismo objetivo. El algoritmo del ordenamiento de burbuja 1 no es la misma que la de la burbuja optimizada 2 y la de selección, no es la misma que la de árbol ABB, dando como ejemplo, que hay veces en la que un problema se puede resolver de distintas maneras.

Por otro lado, a la hora de generar las gráficas, tomamos en cuenta el tamaño del problema como eje "x" y los tiempos reales para el eje "y", todos estos datos los obtuvimos a través de los análisis que realizamos para

esta práctica, es decir, los tiempos que se obtuvieron a la hora de ejecutar los programas, fueron los datos necesarios para generar las gráficas. Es por eso por lo que los resultados que nos hayan salido en los puntos anteriores nos ayudan a seguir con lo que faltaba de la práctica.

Por último, gracias a esta práctica pudimos observar qué algoritmo conviene más, ya sea a su exactitud, su rapidez, complejidad, etc. Y con base a esto también pudimos sacar conclusiones del por qué uno es mejor que otro con hechos que pueden ser verificados con los resultados que obtuvimos en cada uno de los puntos de la práctica.



ANEXOS CÓDIGOS

INSTRUCCIONES DE COMPILACIÓN Y EJECUCIÓN

El archivo que debe ser compilado es: **mainwin.c o mainLin.c,** esto dependerá de**l sistema operativo que se este usando al momento de la ejecución.**

Se solicita que se haga de la siguiente forma Windows:

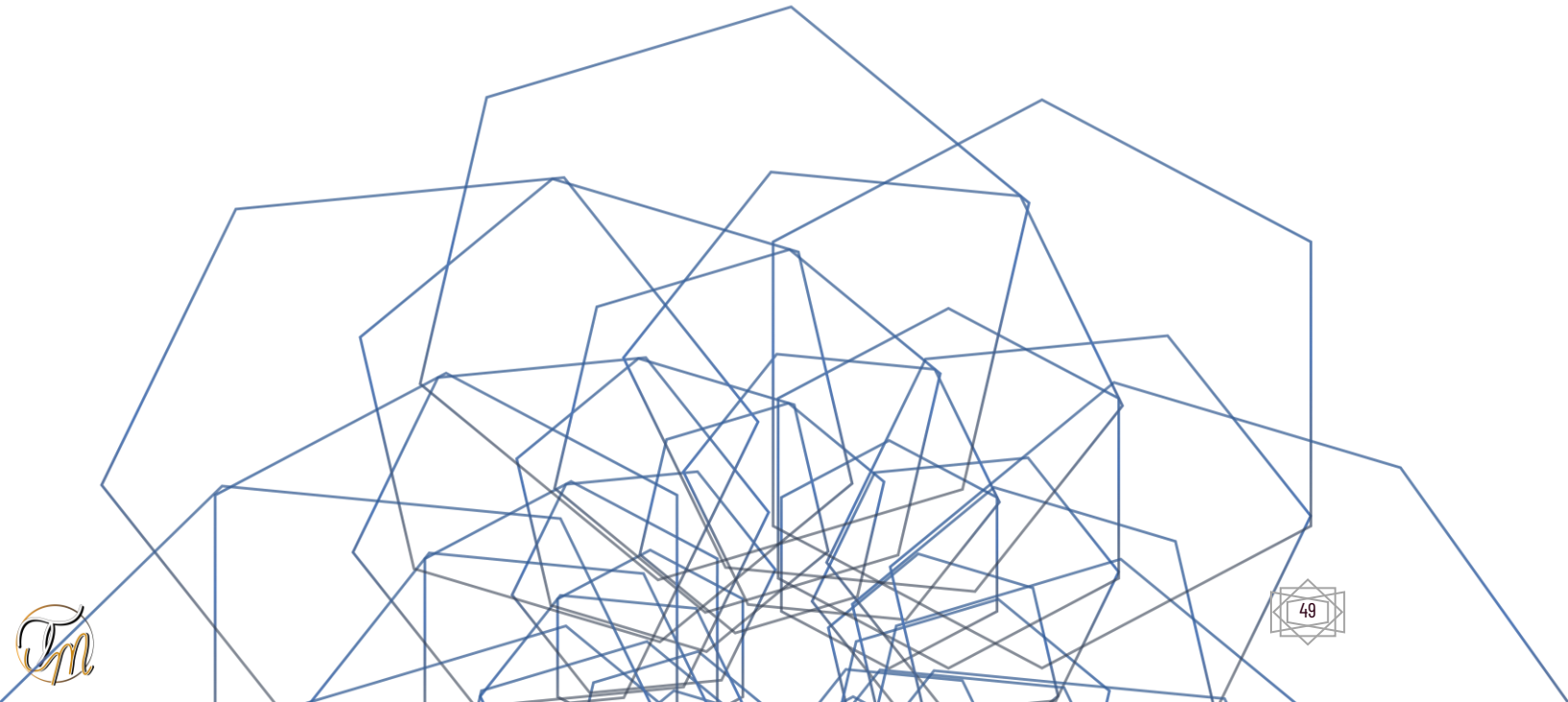
- gcc mainWin.c -o mainWin
- .\mainWin (numero de algoritmo de ordenamiento 1-9) (cantidad de elementos a ordenar)
    - 1.Burbuja
    - 2.BurbujaOpt
    - BurbujaOpt2
    - 4.Insercion
    - 5.Seleccion
    - 6.Shell
    - 7.Arbol
    - 8.Merge
    - 9.Quick
- LINUX:
- gcc mainLin.c -lm -o mainLin
    - Se debe hacer inclusión de lm debido a que empleamos la librería de math.h
- ./mainWin (numero de algoritmo de ordenamiento 1-9) (cantidad de elementos a ordenar)
    - 1.Burbuja
    - 2.BurbujaOpt
    - BurbujaOpt2
    - 4.Insercion
    - 5.Seleccion
    - 6.Shell
    - 7.Arbol
    - 8.Merge
    - 9.Quick

En caso de no mandar los parametros el mismo programa de pedirá que tu ingreses una opción y por defecto de proporcionara un archivo con tiempos de 100 elementos ordenados en caso de Linux o bien un archivo de 100 elementos ordenados con Windows.

MAINLIN.C

```
// Programa Principal para ser ejecutado en Linux
// Si este archivo es visualizado desde Windows marcara un error con el in
clude
// de la medicion de tiempos, pues estamos ocupando funciones exclusivas d
e
// Linux (Por favor pasa a mainWin)
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "./MediciondeTiempos/tiempo.h"
// Para Linux debe ser solo con un punto
#include "./MediciondeTiempos/tiempo.c"
#include "./Funciones/Funciones.c"
#include "./Funciones/Funciones.h"

// Tomara los argumentos que se coloquen al momento de ejecutar, en caso d
e ejecutar sin argumentos
// pedira que ingreses una opcion del 1 al 9 y los elementos a ordenar por
 defecto sera de 100 numeros

int main(int argc, char const *argv[])
{
    int *Arreglo;
    int n,opc;
    if ((atoi(argv[1]))==0)
    {
        printf("Te faltan argumentos\n");
        printf("Ingresa una opcion: (Solo del 1 al 9)\n");
        printf("1.Burbuja\n2.BurbujaOpt\n3.BurbujaOpt2\n4.Insercion\n5.Sel
eccion\n6.Shell\n7.Arbol\n8.Merge\n9.Quick\n");
        scanf("%d",&opc);
        n=100;
```

```
    }
    else{
    opc=atoi(argv[1]);
    n=atoi(argv[2]);
    }
    Arreglo = (int *)malloc(n * sizeof(int));
    LeerArchivo(Arreglo, n);
    MenuSeleccion(Arreglo, n,opc);

    return 0;
}
```



MAINWIN.C

```c
// Programa Principal para ser ejecutado en Windows
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "./MediciondeTiempos/tiempo.h"
#include "./Funciones/FuncionesWin.c"
#include "./Funciones/Funciones.h"
// numeros = Arreglo leido
// n = tamaño del arreglo

// Tomara los argumentos que se coloquen al momento de ejecutar, en caso d
e ejecutar sin argumentos
// pedira que ingreses una opcion del 1 al 9 y los elementos a ordenar por
 defecto sera de 100 numeros

int main(int argc, char const *argv[])
{
    int *Arreglo;
    int n,opc;
    if ((atoi(argv[1]))==0)
    {
        printf("Te faltan argumentos\n");
        printf("Ingresa una opcion: (Solo del 1 al 9)\n");
        printf("1.Burbuja\n2.BurbujaOpt\n3.BurbujaOpt2\n4.Insercion\n5.Sel
eccion\n6.Shell\n7.Arbol\n8.Merge\n9.Quick\n");
        scanf("%d",&opc);
        n=100;
    }
    else{
    opc=atoi(argv[1]);
    n=atoi(argv[2]);
    }
    Arreglo = (int *)malloc(n * sizeof(int));
    LeerArchivo(Arreglo, n);
    MenuSeleccion(Arreglo, n,opc);

    return 0;
}
```

FUNCIONESWIN.C

```c
// Programa Principal de implementacion de funciones para ser usadas en Wi
ndows
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Funciones.h"
#include "../Arbol/Arbol.h"
#include "../Arbol/Arbol.c"
#include <stdbool.h>


// Uso: LeerArchivo(int *A,int n);
// Realiza la lectura del archivo de 10 millones de numeros
// A: Arreglo de numeros no vacio (Obligatorio)
// n: Cantidad de elementos que queremos tomar (Obligatorio)
int *LeerArchivo(int *A, int n)
{
    int i;
    FILE *numeros;
    numeros = fopen("numeros10millones.txt", "r");
    if (numeros == NULL)
    {
        puts("Error en la apertura del archivo");
    }
    for (i = 0; i < n; i++)
    {
        fscanf(numeros, "%d", &A[i]);
    }
    fclose(numeros);
}


    // Uso: Burbuja(int *A,int n);
    // Realiza el ordenamiento de tipo burbuja
    //A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)
void Burbuja(int *A, int n)
{
    int i, j, aux;
    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1); j++)
        {
            if (A[j] > A[j + 1])
            {
```

```
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
    imprimirArreglo(A, n);
}
    // Uso: BurbujaOptimizada(int *A,int n);
    // Realiza el ordenamiento de tipo burbuja Optimizada 1
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)
void BurbujaOptimizada(int *A, int n)
{
    int i, j, aux;
    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1) - i; j++)
        {
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
    imprimirArreglo(A, n);
}
    // Uso: imprimeArreglo (A,n);
    // Imprime el arreglo ordenado
    // A: Arreglo de elementos no vacío (Obligatorio)
    // n: longitud del arreglo
void Burbuja2(int *A, int n)
{
    bool cambio = true;
    int i = 1;
    int j, aux;
    while (i <= (n - 1) && cambio != false)
    {
        cambio = false;
        for (j = 0; j <= ((n - 1) - i); j++)
        {
            if (A[j + 1] < A[j])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
                cambio = true;
            }
        }
        i = i + 1;
```

```
    }
    imprimirArreglo(A, n);
}
    // Uso: Insercion(A,n);
    // Realiza el ordenamiento de tipo Insercion
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)
void Insercion(int *A, int n)
{
    int i, j, temp;
    for (i = 0; i <= n - 1; i++)
    {
        j = i;
        temp = A[i];
        while ((j > 0) && (temp < A[j - 1]))
        {
            A[j] = A[j - 1];
            j--;
        }
        A[j] = temp;
    }
    imprimirArreglo(A, n);
}
    // Uso: Selection(A,n)
    // Realiza el ordenamiento de la forma de Seleccion
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)
void Selection(int *A, int n)
{
    int i, k, p, temp;
    for (k = 0; k < (n - 1); k++)
    {
        p = k;
        for (i = k + 1; i < (n); i++)
        {
            if (A[i] < A[p])
                p = i;
        }
        temp = A[p];
        A[p] = A[k];
        A[k] = temp;
    }
    imprimirArreglo(A, n);
}
    // Uso: Shell(A,n);
    // Realiza el ordenamiento de Tipo Shell
    // A: Arreglo de elementos no vacío (Obligatorio)
    // n: longitud del arreglo
void Shell(int *A, int n)
{
    int i, j, k, b, temp;
    k = ceil(n / 2);
```

```
    while (k >= 1)
    {
        b = 1;
        while (b != 0)
        {
            b = 0;
            for (i = k; i <= n - 1; i++)
            {
                if (A[i - k] > A[i])
                {
                    temp = A[i];
                    A[i] = A[i - k];
                    A[i - k] = temp;
                    b = b + 1;
                }
            }
        }
        k = ceil(k / 2);
    }
    imprimirArreglo(A, n);
}

int parteEntera(double n)
{
    double entero = floor(n);
    return (int)entero;
}
    // Uso: MergeSort(A,0,n-1)
    // Realiza el ordenamiento de elementos de forma recursiva
    // A: Arreglo de elementos no vacio
    // p : Primera posicion
    // r = n-
1 (tamaño del arreglo menos una posicion, por la forma en que se indexa)
void MergeSort(int *A, int p, int r)
{
    if (p < r)
    {
        int q = parteEntera((p + r) / 2);
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        Merge(A, p, q, r);
    }
}
    // A: Arreglo de elementos no vacio
    // p: posicion del primer elemento
    // q: Parte entera obtenida a partir de la division de p/r realizada e
n la implementacion de la funcion
    // r: ultima posicion
void Merge(int *A, int p, int q, int r)
{
    int l = r - p + 1, i = p, j = q + 1, k, *C;
    C = malloc(sizeof(int *) * l);
```

```
    for (k = 0; k <= l; k++)
    {
        if (i <= q && j <= r)
        {
            if (A[i] < A[j])
            {
                C[k] = A[i];
                i++;
            }
            else
            {
                C[k] = A[j];
                j++;
            }
        }
        else if (i <= q)
        {
            C[k] = A[i];
            i++;
        }
        else
        {
            C[k] = A[j];
            j++;
        }
    }
    for (i = p, j = 0; i <= r; i++, j++)
    {
        A[i] = C[j];
    }
    free(C);
}
    // Uso: QuickSort2(A,0,n-1)
    // Realiza el ordenamiento mediante QuickSort, el cual funciona de man
era recursiva, recibe un arreglo
    // no vacio, asi como la cantidad de elementos que contiene -
1 (debido a que buscamos obtener la
    // utlima posicion)
    // A: Arreglo de elementos no vacio
    // i  (primer elemento) con respecto al cual queremos realizar el orde
namiento (en este caso 0)
    // n -1 : Tamaño del arreglo - 1
void Quicksort2(int *A, int primero, int ultimo)
{
    int piv, i, j, central, aux;
    central = (primero + ultimo) / 2;
    piv = A[central], i = primero, j = ultimo;
    do
    {
        while (A[i] < piv)
        {
```

```
            i++;
        }
        while (A[j] > piv)
        {
            j--;
        }
        if (i <= j)
        {
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
            i++;
            j--;
        }
    } while (i <= j);
    if (primero < j)
    {
        Quicksort2(A, primero, j);
    }
    if (primero < ultimo)
    {
        Quicksort2(A, i, ultimo);
    }
}
    // Recibe un arreglo y tamaño del arreglo

void ArbolBinario(int *A, int n)
{
    A = InsertarABB(A, n);
    imprimirArreglo(A, n);
}

// Permite realizar la seleccion de alguno de los algoritmos conforme al p
arametro que fue
// seleccionado al momento de ejecutar el programa
void MenuSeleccion(int *A, int n, int opc)
{
    switch (opc)
    {
    case 1:
        puts("Metodo de Burbuja");
        Burbuja(A, n);
        break;
    case 2:
        puts("Metodo de Burbuja Optimizada 1");
        BurbujaOptimizada(A, n);
        break;
    case 3:
        puts("Metodo de Burbuja Optimizada 2\n");
        Burbuja2(A, n);
        break;
    case 4:
```

```
        puts("Metodo de Insercion");
        Insercion(A, n);
        break;
    case 5:
        printf("Metodo de Seleccion\n");
        Selection(A, n);
        break;
    case 6:
        puts("Metodo de Shell");
        Shell(A, n);
        break;
    case 7:
        puts("Ordenamiento basado en ABB");
        ArbolBinario(A, n);
        break;
    case 8:
        puts("Metodo de MergeSort");
        MergeSort(A, 0, n - 1);
        imprimirArreglo(A, n);
        break;
    case 9:
        puts("Metodo de QuickSort");
        Quicksort2(A, 0, n - 1);
        imprimirArreglo(A, n);
        break;
    default:
        break;
    }
}
    // Uso: imprimeArreglo (A,n);
    // Imprime el arreglo ordenado
    // A: Arreglo de elementos no vacío (Obligatorio)
    // n: longitud del arreglo
void imprimirArreglo(int *A, int n)
{
    int aux = 1;
    for (int i = 0; i < n; i++)
    {
        printf("%d. %d,\n ", aux, A[i]);
        aux++;
    }
}
```

FUNCIONES.H

```
// Archivo de cabecera de las funciones
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
#ifndef Funciones
#define Funciones
// Documentar Funciones
int *LeerArchivo(int *A, int n);
    //
    // Realiza la lectura del archivo de 10 millones de numeros
    // Uso: LeerArchivo(int *A,int n);
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos que queremos tomar (Obligatorio)

void MenuSeleccion(int *A, int n,int opc);
//Declaracion cabecera de Funciones
void Burbuja(int *numeros, int n);
    // Realiza el ordenamiento de tipo burbuja
    // Uso: Burbuja(int *A,int n);
    //A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)
void BurbujaOptimizada(int *A, int n);
    // Realiza el ordenamiento de tipo burbuja Optimizada 1
    // Uso: BurbujaOptimizada(int *A,int n);
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)
void Burbuja2(int *A, int n);
    // Imprime el arreglo ordenado
    // Uso: imprimeArreglo (A,n);
    // A: Arreglo de elementos no vacío (Obligatorio)
    // n: longitud del arreglo

void Insercion(int *A, int n);
    // Realiza el ordenamiento de tipo Insercion
    // Uso: Insercion(A,n);
```

```
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)

void Shell(int *A, int n);
    // Realiza el ordenamiento de Tipo Shell
    // Uso: Shell(A,n);
    // A: Arreglo de elementos no vacío (Obligatorio)
    // n: longitud del arreglo

void imprimirArreglo(int *A,int n);
    // Imprime el arreglo ordenado
    // Uso: imprimeArreglo (A,n);
    // A: Arreglo de elementos no vacío (Obligatorio)
    // n: longitud del arreglo

void Selection(int *A, int n);
    // Realiza el ordenamiento de la forma de Seleccion
    // Uso: Selection(A,n)
    // A: Arreglo de numeros no vacio (Obligatorio)
    // n: Cantidad de elementos del arreglo (Obligatorio)

void Quicksort2(int *A, int i, int n);
    // Realiza el ordenamiento mediante QuickSort, el cual funciona de man
era recursiva, recibe un arreglo
    // no vacio, asi como la cantidad de elementos que contiene -
1 (debido a que buscamos obtener la
    // utlima posicion)
    // A: Arreglo de elementos no vacio
    // i  (primer elemento) con respecto al cual queremos realizar el orde
namiento (en este caso 0)
    // n -1 : Tamaño del arreglo - 1
    // Uso: QuickSort2(A,0,n-1)

void MergeSort(int *A, int p, int r);
    // Realiza el ordenamiento de elementos de forma recursiva
    // Uso: MergeSort(A,0,n-1)
    // A: Arreglo de elementos no vacio
    // p : Primera posicion
```

```
    // r = n-
1 (tamaño del arreglo menos una posicion, por la forma en que se indexa)

void Merge(int *A, int p, int q, int r);

    // A: Arreglo de elementos no vacio
    // p: posicion del primer elemento
    // q: Parte entera obtenida a partir de la division de p/r realizada e
n la implementacion de la funcion
    // r: ultima posicion

int parteEntera(double n);
    // Recibe un numero y lo manda a piso (redondeo hacia abajo) para obte
ner un valor de tipo entero

void ArbolBinario (int *A,int n);
    // Recibe un arreglo y tamaño del arreglo
void ImprimirTiempos(double utime0, double stime0, double wtime0, double u
time1, double stime1, double wtime1);
    // Imprime los tiempos que tomo en ser ejecutado el programa de forma
 decimal y de forma exponencial
    // Uso: ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1
)
    // utime0,stime0,wtime0 : Vairables del tiempo inicial (deberia ser 0)
    // utime1, stime1, wtime1: Variables de tiempo final
#endif //Funciones
```

FUNCIONES.C

```c
// Funciones para ser usado en Linux
// Si este archivo es visualizado desde Windows marcara un error con el in
clude
// de la medicion de tiempos, pues estamos ocupando funciones exclusivas d
e
// Linux (Por favor pasa a mainWin)
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Funciones.h"
#include "../Arbol/Arbol.h"
#include "../Arbol/Arbol.c"
#include <stdbool.h>
//Variables para medición de tiempos
double utime0, stime0, wtime0, utime1, stime1, wtime1;
int *LeerArchivo(int *A, int n)
{
    int i;
    FILE *numeros;
    numeros = fopen("numeros10millones.txt", "r");
    if (numeros == NULL)
    {
        puts("Error en la apertura del archivo");
    }
    for (i = 0; i < n; i++)
    {
        fscanf(numeros, "%d", &A[i]);
    }
    fclose(numeros);
}

void Burbuja(int *A, int n)
{
    int i, j, aux;
    uswtime(&utime0, &stime0, &wtime0);

    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1); j++)
        {
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
```

```
                A[j + 1] = aux;
            }
        }
    }
    uswtime(&utime1, &stime1, &wtime1);

    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}

void BurbujaOptimizada(int *A, int n)
{
    int i, j, aux;
    uswtime(&utime0, &stime0, &wtime0);
    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1) - i; j++)
        {
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
    uswtime(&utime1, &stime1, &wtime1);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
    // imprimirArreglo(A, n);
}

void Burbuja2(int *A, int n)
{
    bool cambio = true;
    int i = 1;
    int j, aux;
    uswtime(&utime0, &stime0, &wtime0);

    while (i <= (n - 1) && cambio != false)
    {
        cambio = false;
        for (j = 0; j <= ((n - 1) - i); j++)
        {
            if (A[j + 1] < A[j])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
                cambio = true;
            }
        }
        i = i + 1;
```

```
    }
    uswtime(&utime1, &stime1, &wtime1);

    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}

void Insercion(int *A, int n)
{
    int i, j, temp;
    uswtime(&utime0, &stime0, &wtime0);
    for (i = 0; i <= n - 1; i++)
    {
        j = i;
        temp = A[i];
        while ((j > 0) && (temp < A[j - 1]))
        {
            A[j] = A[j - 1];
            j--;
        }
        A[j] = temp;
    }

    uswtime(&utime1, &stime1, &wtime1);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
    // imprimirArreglo(A, n);
}

void Selection(int *A, int n)
{
    int i, k, p, temp;
    uswtime(&utime0, &stime0, &wtime0);
    for (k = 0; k < (n - 1); k++)
    {
        p = k;
        for (i = k + 1; i < (n); i++)
        {
            if (A[i] < A[p])
                p = i;
        }
        temp = A[p];
        A[p] = A[k];
        A[k] = temp;
    }
    uswtime(&utime1, &stime1, &wtime1);
    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}

void Shell(int *A, int n)
{
    int i, j, k, b, temp;
```

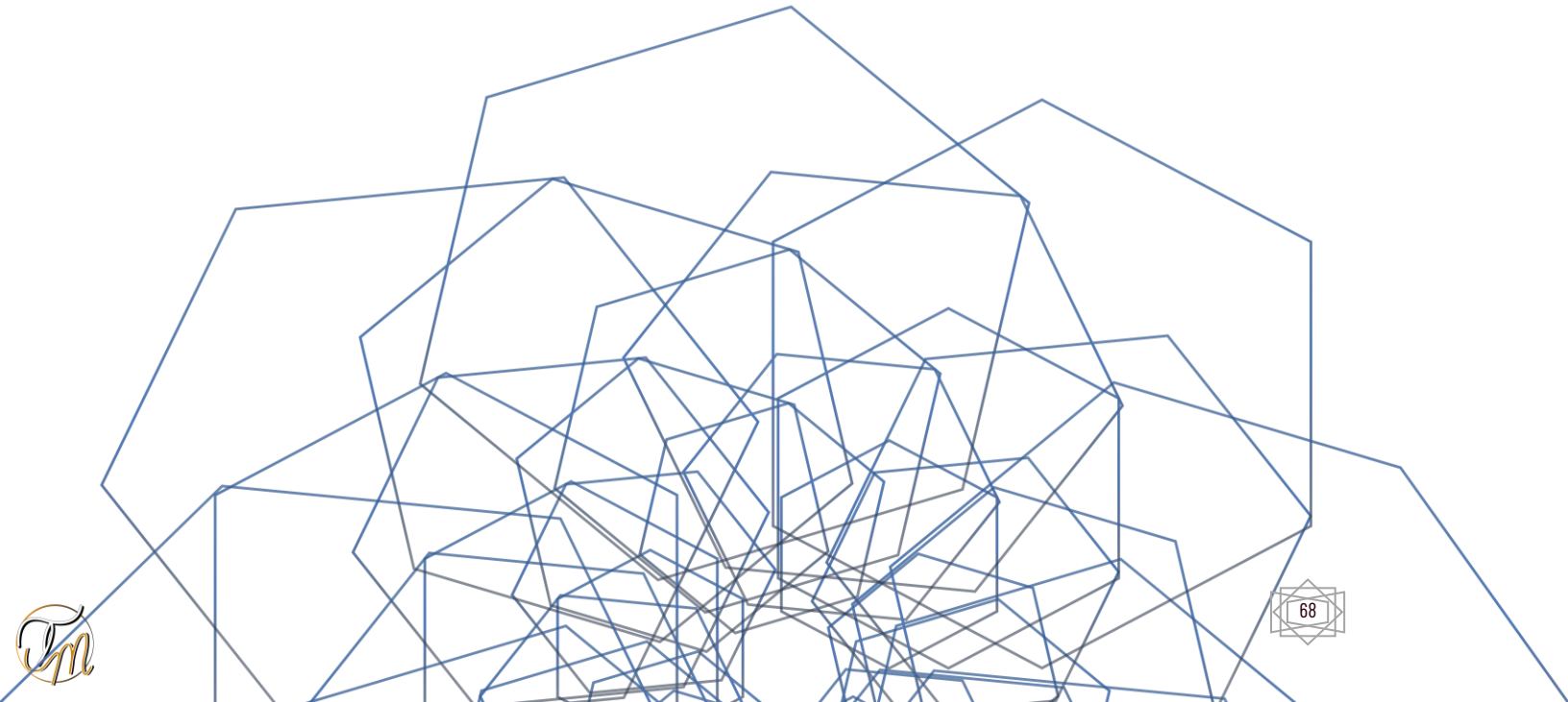
```
    uswtime(&utime0, &stime0, &wtime0);
    k = ceil(n / 2);
    while (k >= 1)
    {
        b = 1;
        while (b != 0)
        {
            b = 0;
            for (i = k; i <= n - 1; i++)
            {
                if (A[i - k] > A[i])
                {
                    temp = A[i];
                    A[i] = A[i - k];
                    A[i - k] = temp;
                    b = b + 1;
                }
            }
        }
        k = ceil(k / 2);
    }
    uswtime(&utime1, &stime1, &wtime1);
    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}

int parteEntera(double n)
{
    double entero = floor(n);
    return (int)entero;
}

void MergeSort(int *A, int p, int r)
{

    if (p < r)
    {
        int q = parteEntera((p + r) / 2);
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        Merge(A, p, q, r);
    }
}

void Merge(int *A, int p, int q, int r)
{
    int l = r - p + 1, i = p, j = q + 1, k, *C;
    C = malloc(sizeof(int *) * l);

    for (k = 0; k <= l; k++)
    {
        if (i <= q && j <= r)
```

```
        {
            if (A[i] < A[j])
            {
                C[k] = A[i];
                i++;
            }
            else
            {
                C[k] = A[j];
                j++;
            }
        }
        else if (i <= q)
        {
            C[k] = A[i];
            i++;
        }
        else
        {
            C[k] = A[j];
            j++;
        }
    }
    for (i = p, j = 0; i <= r; i++, j++)
    {
        A[i] = C[j];
    }

    free(C);
}

void Quicksort2(int *A, int primero, int ultimo)
{
    int piv, i, j, central, aux;
    central = (primero + ultimo) / 2;
    piv = A[central], i = primero, j = ultimo;
    do
    {
        while (A[i] < piv)
        {
            i++;
        }
        while (A[j] > piv)
        {
            j--;
        }
        if (i <= j)
        {
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
            i++;
```

```
            j--;
        }
    } while (i <= j);
    if (primero < j)
    {
        Quicksort2(A, primero, j);
    }
    if (primero < ultimo)
    {
        Quicksort2(A, i, ultimo);
    }
}

void ArbolBinario(int *A, int n)
{
    uswtime(&utime0, &stime0, &wtime0);
    A = InsertarABB(A, n);
    uswtime(&utime1, &stime1, &wtime1);
    // imprimirArreglo(A, n);
    ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
}

void MenuSeleccion(int *A, int n, int opc)
{
    switch (opc)
    {
    case 1:
        puts("Metodo de Burbuja");
        Burbuja(A, n);
        break;
    case 2:
        puts("Metodo de Burbuja Optimizada 1");
        BurbujaOptimizada(A, n);
        break;
    case 3:
        puts("Metodo de Burbuja Optimizada 2\n");
        Burbuja2(A, n);
        break;
    case 4:
        puts("Metodo de Insercion");
        Insercion(A, n);
        break;
    case 5:
        printf("Metodo de Seleccion\n");
        Selection(A, n);
        break;
    case 6:
        puts("Metodo de Shell");
        Shell(A, n);
        break;
    case 7:
        puts("Ordenamiento basado en ABB");
```

```
        ArbolBinario(A, n);
        break;
    case 8:
        puts("Metodo de MergeSort");
        uswtime(&utime0, &stime0, &wtime0);
        MergeSort(A, 0, n - 1);
        uswtime(&utime1, &stime1, &wtime1);
        ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);
        // imprimirArreglo(A, n);
        break;
    case 9:
        puts("Metodo de QuickSort");
        uswtime(&utime0, &stime0, &wtime0);
        Quicksort2(A, 0, n - 1);
        uswtime(&utime1, &stime1, &wtime1);
        ImprimirTiempos(utime0, stime0, wtime0, utime1, stime1, wtime1);

        // imprimirArreglo(A, n);
        break;
    default:
        break;
    }
}

void imprimirArreglo(int *A, int n)
{
    int aux = 1;
    for (int i = 0; i < n; i++)
    {
        printf("%d. %d,\n ", aux, A[i]);
        aux++;
    }
}

void ImprimirTiempos(double utime0, double stime0, double wtime0, double u
time1, double stime1, double wtime1)
{
    printf("\n");
    printf("real (Tiempo total)  %.10f s\n", wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10f s\n", utime1 - uti
me0);
    printf("sys (Tiempo en acciónes de E/S)  %.10f s\n", stime1 - stime0);
    printf("CPU/Wall   %.10f %% \n", 100.0 * (utime1 - utime0 + stime1 - s
time0) / (wtime1 - wtime0));
    printf("\n");

    // Mostrar los tiempos en formato exponecial
    printf("\n");
    printf("real (Tiempo total)  %.10e s\n", wtime1 - wtime0);
    printf("user (Tiempo de procesamiento en CPU) %.10e s\n", utime1 - uti
me0);
    printf("sys (Tiempo en acciónes de E/S)  %.10e s\n", stime1 - stime0);
```

```
    printf("CPU/Wall   %.10f %% \n", 100.0 * (utime1 - utime0 + stime1 - s
time0) / (wtime1 - wtime0));
    printf("\n");
}
```

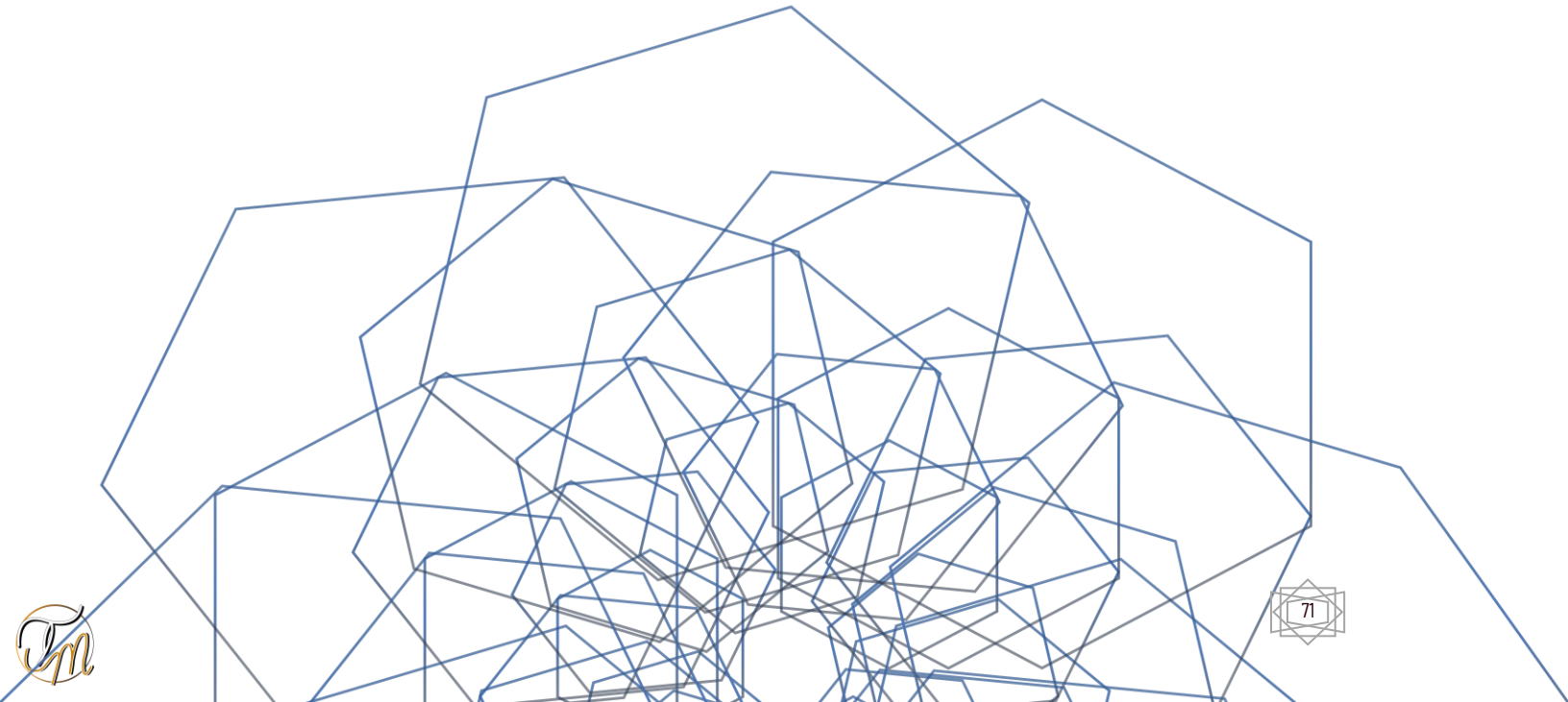


ARBOL.C

```
// Arbol de Busqueda Binaria
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio,
// 3CM12
#include <stdlib.h>
#include "Arbol.h"

// *A = Arreglo de lista de numeros
// n Longitud del arreglo
// DatoRecibido = Numeros de la lista obtenidos mediante el for

int * InsertarABB (int * A, int n)
{
    // Declarando raiz del arbol
    arbolBinario *Raiz;
    //Reservando memoria
    Raiz = (arbolBinario *) malloc (sizeof(arbolBinario));
    // Estableciendo la raiz como vacia
    Raiz = NULL;
    for (int i = 0; i < n; ++i)
    // Recorriendo la lista para ir insertando cada numero en el arbol
        Insertar(&Raiz,A[i]);

    A = Inorden(Raiz,A);
}

void Insertar(arbolBinario **Raiz,int DatoRecibido)
{
    //Se crea un nodo auxiliar
    arbolBinario **Auxiliar = NULL;
     //Se iguala el auxiliar numeros la raiz
    Auxiliar = Raiz;
    // Si no hay dato se crea un nodo nuevo
    if (*Raiz == NULL)
        *Raiz = InicializarArbol(DatoRecibido);
    else
    {

        // Bucle para buscar un nodo que este vacio y poder proceder a una
        // nueva insercion de dato
        while(*Auxiliar != NULL)
        {
            // Si el numero a ingresar es menor en Auxiliar
            if (DatoRecibido < (*Auxiliar)->dato)
            // Se va al nodo izquierdo
                Auxiliar = &((*Auxiliar)->izq);
                //Si el numero a ingresar es mayor en Auxiliar
            else if (DatoRecibido > (*Auxiliar)->dato)
```

```
                Auxiliar = &((*Auxiliar)->der); //Se va al nodo derecho
        }
        // Ya se llego a la posicion adecuada, agrega el nuevo nodo
        // Cuando llegamos a la posicion donde no había dato podemos agreg
ar
        // el nodo nuevo (Auxiliar ya sabe si debe ir a la izquierda o der
echa)
        // Gracias al if que pusimos
        *Auxiliar = InicializarArbol (DatoRecibido);
    }
}


/*

Funcion: Recibe un arbol binario Raiz y una posicion A, realiza el recorri
do izquierdo -> raiz -> derecho a partir de la posicion A.
Requerimientos: El arbol binario A es no vacio y la posicion A es una posi
cion valida.
*/
int * Inorden (arbolBinario * Raiz, int *A)
{
    int i = 0;
    arbolBinario *Auxiliar,*Recorrido;
    Auxiliar = Raiz;
    while (Auxiliar !=NULL)
    {
        if (Auxiliar->izq == NULL)
        {
            A[i] = Auxiliar->dato;
            Auxiliar = Auxiliar->der;
            i++;
        }
        else
        {
            Recorrido = Auxiliar->izq;
            while(Recorrido->der !=NULL && Recorrido->der != Auxiliar)
                Recorrido = Recorrido->der;
            if(Recorrido->der == NULL)
            {
                Recorrido->der = Auxiliar;
                Auxiliar = Auxiliar->izq;
            }
            else
            {
                Recorrido->der = NULL;
                A[i] = Auxiliar->dato;
                Auxiliar = Auxiliar->der;
                i++;
            }
        }
    }
```

```
    return A;
}


// Inicializa un arbol, reservando memoria y asignando primer elemento a D
ato
//                                  Recibe un dato
// arbolbinario * IniciarlizarAtblo (int NumDato)
arbolBinario * InicializarArbol (int numDato)
{
    arbolBinario *A = malloc(sizeof(nodo));
    // Asignando el DatoRecibido a dato del aebol
    A->dato= numDato;
    //Sin nodo izquierdo
    A->izq=NULL;
    //Sin nodo derecho
    A->der=NULL;
    // Regresamos el Arreglo
    return A;
}
```



ARBOL.H

```h
#ifndef Arbol
#define Arbol
#define TRUE 1
#define FALSE 0

// Definiendo estructura del arbol
typedef struct nodo
{
    struct nodo *izq, *der;
    int dato;
}nodo;

typedef nodo arbolBinario;

// *A = Arreglo de lista de numeros
// n Longitud del arreglo
// DatoRecibido = Numeros de la lista obtenidos mediante el for

// Funcion devuelve un antero (Recibe un Arreglo, Recibe un dato numerico
)
int * InsertarABB(int *A, int n);
// Inserta elementos en el arbol (Recibe un Arbol binario, Recibe un dato
 numerico)
void Insertar(arbolBinario **Raiz,int DatoRecibido);
// Cuando el arbol se encuentra vacio inicializa uno nuevo (Recibe un dat
o numerico)
arbolBinario * InicializarArbol (int numDato);
// Colocara en orden los elementos del arreglo (Recibe un arbol binario,a
rreglo de numeros)
int * Inorden (arbolBinario *Raiz, int *numeros);
#endif
```

ANEXO SCRIPTS

Script General Linux

```
#!/bin/bash
source Burbuja.sh
source BurbujaOptimizada.sh
source BurbujaOptimizada2.sh
source Insercion.sh
source MergeSort.sh
source QuickSort.sh
source Seleccion.sh
source Shell.sh
source ArbolABB.sh
```

Script Burbuja.sh

```
#!/bin/bash
echo Hola Estas apunto de ejecutar el programa:
cd ..
gcc mainLin.c -lm -o mainLin

./mainLin 1 100 >BurbujaT.txt
./mainLin 1 1000 >>BurbujaT.txt
./mainLin 1 5000 >>BurbujaT.txt
./mainLin 1 10000 >>BurbujaT.txt
./mainLin 1 50000 >>BurbujaT.txt
./mainLin 1 100000 >>BurbujaT.txt
./mainLin 1 125000 >>BurbujaT.txt
./mainLin 1 175000 >>BurbujaT.txt
./mainLin 1 200000 >>BurbujaT.txt
./mainLin 1 225000 >>BurbujaT.txt
./mainLin 1 275000 >>BurbujaT.txt
./mainLin 1 300000 >>BurbujaT.txt
./mainLin 1 325000 >>BurbujaT.txt
./mainLin 1 375000 >>BurbujaT.txt
./mainLin 1 400000 >>BurbujaT.txt
./mainLin 1 425000 >>BurbujaT.txt
./mainLin 1 500000 >>BurbujaT.txt
# ./mainLin 1 600000 >>BurbujaT.txt
# ./mainLin 1 800000 >>BurbujaT.txt
# ./mainLin 1 1000000 >>BurbujaT.txt
# ./mainLin 1 2000000 >>BurbujaT.txt
# ./mainLin 1 3000000 >>BurbujaT.txt
# ./mainLin 1 4000000 >>BurbujaT.txt
# ./mainLin 1 5000000 >>BurbujaT.txt
# ./mainLin 1 6000000 >>BurbujaT.txt
# ./mainLin 1 7000000 >>BurbujaT.txt
# ./mainLin 1 8000000 >>BurbujaT.txt
# ./mainLin 1 9000000 >>BurbujaT.txt
# ./mainLin 1 10000000 >>BurbujaT.txt
```

(El resto de Script solo varía en el primer parámetro, el cual determina la opción del menú que se seleccionó, así como en el nombre de salida de cada archivo)

Script General Windows

```
Arbol0.bat
Burbuja.bat
Burbuja0.bat
Burbuja02.bat
Insercion0.bat
Merge0.bat
Quick0.bat
Seleccion0.bat
Shell0.bat
pause
```

Script Arbol0.bat

```
cd ..
gcc mainWin.c -lm -o mainWin

.\mainWin 7 100 >Arbol0.txt
.\mainWin 7 1000 >>Arbol0.txt
.\mainWin 7 5000 >>Arbol0.txt
.\mainWin 7 10000 >>Arbol0.txt
.\mainWin 7 50000 >>Arbol0.txt
.\mainWin 7 100000 >>Arbol0.txt
.\mainWin 7 200000 >>Arbol0.txt
.\mainWin 7 400000 >>Arbol0.txt
.\mainWin 7 600000 >>Arbol0.txt
.\mainWin 7 800000 >>Arbol0.txt
.\mainWin 7 1000000 >>Arbol0.txt
.\mainWin 7 2000000 >>Arbol0.txt
.\mainWin 7 3000000 >>Arbol0.txt
.\mainWin 7 4000000 >>Arbol0.txt
.\mainWin 7 5000000 >>Arbol0.txt
.\mainWin 7 6000000 >>Arbol0.txt
.\mainWin 7 7000000 >>Arbol0.txt
.\mainWin 7 8000000 >>Arbol0.txt
.\mainWin 7 9000000 >>Arbol0.txt
.\mainWin 7 10000000 >>Arbol0.txt

pause
```

BIBLIOGRAFÍA

[1] E. A. F. Martinez, «eafranco.eakdemy,» 6 Septiembre 2021. [En línea]. Available: https://eafranco.eakdemy.com/mod/scorm/player.php. [Último acceso: 2021 Septiembre 17].

[2] T. H. C. C. E. L. R. L. R. C. S. Thomas H.. Cormen, Introduction To Algorithms, ilustrada, reimpresa ed., M. Press, Ed., United States, 2001.