Práctica 1

Integrantes:

Buendía Velazco Abel

Carpio Becerra Erick Gustavo

Hernández Molina Leonardo Gaell

Velázquez Diaz Luis Francisco

Equipo:

Los tíos pelones

Grupo:

2CM1

ÍNDICE

MARCO TEORICO

Los algoritmos de ordenamiento son una serie de instrucciones que se le da a un programa con la finalidad de poder ordenar información de una manera especial, con la finalidad de hacer las cosas mucho más sencillas al momento de el manejo de datos. Pero para poder entrar más en materia al tema debemos profundizado acerca de que es un algoritmo el cual podemos definirlo cómo un conjunto de finito de pasos inherentes a un cómputo para resolver un problema. Es decir. Es una serie de instrucciones que tiene un inicio y un fin, las cuales son indispensables a la hora de poder resolver un problema computacional.

Además de tener una serie de instrucciones finitas, un algoritmo deberá ser capaz de resolver dicho problema en un tiempo razonable, ya que en la mayoría de las ocasiones no nos servirá tener la respuesta a un problema si esta nos va a tardar mucho tiempo para obtenerse, ya que dicha respuesta dejaría de sernos útil para resolver dicha disputa.

Como mencionábamos en el párrafo anterior, los algoritmos deben poder resolver un problema en un tiempo razonable, por ello y para poder saber con más exactitud cuanto es que nos podría tardar en obtener la respuesta de dicho planteamiento, podemos guiarnos por su orden de complejidad el cual además de medir el tiempo de ejecución, nos es muy útil para poder conocer la eficiencia o la memoria que se vaya a utilizar. Las complejidades de los algoritmos las podemos destacar en las siguientes:

- ***O (1)* Complejidad constante.**
- ***O (log n)* Complejidad logarítmica.**
- ***O (n)* Complejidad lineal.**
- ***O (n log n)* Complejidad “n log n”.**
- ***O(n^2)* Complejidad cuadrática.**
- ***O(n^3)* Complejidad cubica.**
- ***O (n^c)* c>1 Complejidad exponencial.**
- ***O (n!)* Complejidad factorial.**

Planteamiento del problema

En la práctica buscamos analizar los algoritmos de ordenamiento, tanto las complejidades de cada algoritmo, el tiempo que tarda en funcionar y el espacio en memoria que ocupan, para esto es necesario programar los algoritmos y ejecutarlos en una computadora sobre algunos ejemplares de prueba haciendo medidas para cierta cantidad de números (dados por el usuario).

Para poder realizar la practica usaremos material que nos proporcionó el maestro Edgardo Adrián Franco Martínez, que son los pseudocódigos de los métodos de ordenamiento a implementar y un documento texto que contiene 1,000,000 de números totalmente diferentes y aleatorios.

Las pruebas de ordenamiento se realizarán con los siguientes valores: 500, 1000, 5000, 10000, 20000, 50000, 100000, 200000 y 500000 números (se realizan todas las pruebas por algoritmo implementado), después de tener dichas pruebas se realizará la comparación de tiempos (por algoritmo), con su respectiva gráfica para ver cómo es su variación de tiempo, los métodos de ordenamiento a implementar son los siguientes:

- Burbuja (Bubble Sort).
- Inserción (Insertion Sort).
- Selección (Selection Sort).
- Shell (Shell Sort)
- Merge (Merge Sort).
- Quick (Quick Sort).

Diseño y funcionamiento de la solución

En el siguiente apartado explicaremos cada método de ordenamiento y agregaremos su respectivo pseudocódigo para poder ser programado.

Burbuja (Bubble Sort)

Este algoritmo funciona de manera tal en la que revisa cada elemento del arreglo con el siguiente, de tal manera que los ordenará, para esto los intercambia de posición si están en desorden, este revisa varias veces toda la lista hasta que no sean necesarios intercambios, cuando deje de revisarla significará que la lista estará ordenada.

Burbuja optimizada: Como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo el numero de comparaciones por iteración, además puede existir la posibilidad de realizar iteraciones de mas si el arreglo ya fue ordenado totalmente.

```
Procedimiento 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]) hacer
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                                cambios = "Si"
                        FinSi
                FinPara
                 i= i+1                          3 1 5 6 7 2 4 8
        FinMientras
fin Procedimiento
```

Inserción (Insertion Sort)

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 a 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.

```
Procedimiento 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 Procedimiento
```

Selección (Selection Sort).

Este método de ordenamiento se basa en buscar el mínimo elemento de la lista e intercambio con el primero, después busca el siguiente mínimo en el resto de la lista y lo intercambia con el segundo, y así sucesivamente.

```
Procedimiento 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 Procedimiento
```

Shell (Shell Sort)

Shell es una generalización del ordenamiento por inserción, teniendo encuenta dos observaciones:

- El ordenamiento por inserción es eficiente, si la entrada está “casi ordenada”.
- El ordenamiento por inserción es ineficiente, en general, porque mueve los valores solo una posición cada vez.

El algoritmo Shell mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga “pasos más grandes” hacia su posición esperada. Los pasos múltiples sobren los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del ordenamiento Shell es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

- Shell propone que se haga sobre el arreglo una serie de ordenaciones basadas en la inserción directa, pero dividiendo el arreglo original en varios sub-arreglos tales que cada elemento esté separado k elementos del anterior (a esta separación a menudo se le llama salto o gap).
- Se debe empezar con k=n/2, siendo n el número de elementos del arreglo, y utilizando siempre la división entera (TRUNC).

- Después iremos variando k haciéndolo más pequeño mediante sucesivas divisiones por 2, hasta llegar a k=1.

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

Merge (Merge Sort)

El ordenamiento por mezcla es un algoritmo recursivo que divide continuamente una lista por la mitad.

Si la lista está vacía o tiene un solo elemento, se ordena por definición (el caso base). Si la lista tiene más de un elemento, dividimos la lista e invocamos recursivamente un ordenamiento por mezcla para ambas mitades.

Una vez que las dos mitades están ordenadas, se realiza la operación fundamental, denominada mezcla. La mezcla es el proceso de tomar dos listas ordenadas, armas pequeñas y combinarlas en una sola lista nueva y ordenada.

```
Merge(A, p, q, r)
{
    l=r-p+1, i=p, j=q+1;
    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(k=p,i=0;k<=r;k++,i++)
        A[k]=C[i];
}
```

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

Quick (Quick Sort)

El algoritmo trabaja de la siguiente forma:

1. Elige un elemento del conjunto de elementos a ordenar, al que llamaremos pivote.
2. 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.
3. La lista queda separada en dos sub-listas, una formada por los elementos a la izquierda del pivote, y por otra por los elementos a su derecha.

4.- Repetir este proceso de forma recursiva, para cada sub-lista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos están ordenados.

```
Algoritmo QuickSort(A, p, r)
    si p < r entonces
        j = Pivot(A,p,r)
        QuickSort(A, p, j-1)
        QuickSort(A, j+1,r)
    fin si
fin Algoritmo
```

```
Algoritmo Intercambiar(A, i, j)
  temp= A[j]
  A[j]=A[i]
  A[i]=temp
fin Algoritmo
```

```
Algoritmo Pivot(A, p, r)
    piv=A[p], i=p+1, j=r
    mientras (i<j)
        mientras A[i]<= piv y i<r hacer
            i++
        mientras A[j]> piv hacer
            j--
        Intercambiar(A,i,j)
    fin mientras
    Intercambiar(A,p,j)
    regresar j
fin Algoritmo
```

Implementación de la solución

INSERCIÓN

En este tipo de ordenamiento 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 (en este punto, todos los elementos mayores se desplazaron hacia 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.

```c
void Inserccion(int *A,int n)
{
    int i, j, temp;

    for (i = 0; i <= n - 1; i++)
    {
        j = i;
        temp = A[i];
        //Seccion de recorrido de numeros en caso de que el termino anterior sea mayor
        while ((j > 0) && (temp < A[j - 1]))
        {
            A[j] = A[j - 1];
            j--;
        }
        //Asignacion del lugar que ocupara el numero
        A[j] = temp;
    }

}
```

Imagen: Algoritmo de inserción implementado en C

BURBUJA

Este revisa cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si estos se encuentran en el orden equivocado.

Es necesario que este revise la lista en varias ocasiones hasta que no se necesiten más intercambios, lo cual significará que la lista ya está ordenada.

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

Como al final de cada proceso, el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro elemento, esto reduce el número de comparaciones por iteración.

```c
void BurbujaOptimizada(int *A,int n)
{
    int i=0,aux, cambios = 1;
    int j,k;

    while(i<=(n-1) && cambios==1){
            cambios = 0;
            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;
                cambios = 1;

                }
            }

            i++;
    }

    //Voltear el arreglo O(n), ya que el pseudocodigo nos da el ordenamiento invertido
    for(i=0,j=n-1;i<n/2;i++,j--)
    {
        aux=A[i];
        A[i]=A[j];
        A[j]=aux;
    }


    return;
```

Imagen: Algoritmo de burbuja implementado en C

SELECCIÓN

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

Este algoritmo mejora ligeramente el algoritmo de la burbuja. En el caso de tener que ordenar un vector de enteros, esta mejora no es muy sustancial, pero cuando hay que ordenar un vector de estructuras más complejas, la operación de intercambiar los elementos sería más costosa en este caso.

Su funcionamiento se puede definir de forma general como:

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

```c
void Seleccion (int *A, int n){
    int k, i, p, aux;

    for (k = 0; k < n - 1; k++)
    {
        p = k; /*variable que guarda k para hacer la comparacion con el valor no
        acomodado mas a la izquierda en nuestro arreglo
        //recorre nuestro arreglo de acuerdo a k, el cual nos marca desde
        donde empezara a recorrerlo este ciclo*/
        for (i = k + 1; i < n; i++)
        {
            /*Compara si el ultimo valor no cambiada a la izquierda de nuestro
            arreglo es mayor al de los siguientes para encontrar el valor minimo*/
            if (A[i] < A[p])
                p = i; //guarda el valor minimo de la lista
        }

        /*Se cambian de posicion con ayuda de una variable auxiliar para acomodar el valor
        minimo encontrado en el primero mas a la izq que no haya sido acomodado antes*/
        aux = A[p];
        A[p] = A[k];
        A[k] = aux;
}
}
```

Imagen: Algoritmo de selección implementado en C

SHELL SORT

Este algoritmo mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Permite que un elemento haga “pasos mas grandes” hacía su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso de este ordenamiento es una simple inserción, pero para entonces, ya se garantizó que los datos del vector están “casi” ordenados.

Shell propone que se haga sobre el arreglo una serie de ordenaciones basadas en la inserción directa, pero dividiendo el arreglo original en varios sub - arreglos tales que cada elemento esté separado por k elementos del anterior.

```c
/*
void Algoritmo(int *arreglo,int size)
Recibe: int * Referencia/Dirección al arreglo A, int tamaño del arreglo
Devuelve: void (No retorna valor explicito)
Observaciones: Función que revierte el orden de los valores de A (voltea el arreglo)
y posteriormente pierde tiempo en razón de O(n^3)
*/
void Shell(int *A,int n){
int  i, k, b, temp;
    k = 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=k/2;
    }

    return;
}
```

Imagen: Algoritmo de Shell implementado en C

MERGE SORT

El ordenamiento por mezcla funciona de la siguiente manera:

- Si la longitud de la lista es 0 o 1, la lista ya está ordenada. Si esto no es así:
- Dividirá la lista desordenada en dos sublistas de aproximadamente la mitad de tamaño.
- Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
- Mezclar ambas sublistas para lograr una sola lista ordenada.

A su vez, implementa 2 ideas principales para mejorar su eficacia y su tiempo de ejecución:

- Por obvias razones, una lista pequeña necesitará menos pasos para ordenarse que una lista de mayor tamaño.
- Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que partir dos listas desordenadas.

```c
void Merge(int *A, int p, int q, int r){ //A= Arreglo desordenado
    int *C; //Arreglo auxiliar           //P= Parte izquierda
                                         //Q= Tamaño del arreglo
    //Variables para los ciclos          //R= Parte derecha
    int l, i, j;
    l = r - p + 1;//Inicio del arreglo
    i = p; //Parte izquierda
    j = q + 1; //Parte derecha
    C = malloc(sizeof(int) * (l + 2)); //Espacio en memoria para guarda el arreglo que  vaya ordenando

    for (int 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++;
            }
        }
    }

    //Terminando de ordenar los numeros, se pasan los numeros del arreglo temporal al arreglo donde se van almacenar todos los numeros a ordenar
    int o = 0;
    for (int m = p; m <= r; m++)
    {
        A[m] = C[o];
        o++;
    }

}
```

Imagen: Algoritmo de Marge implementado en C

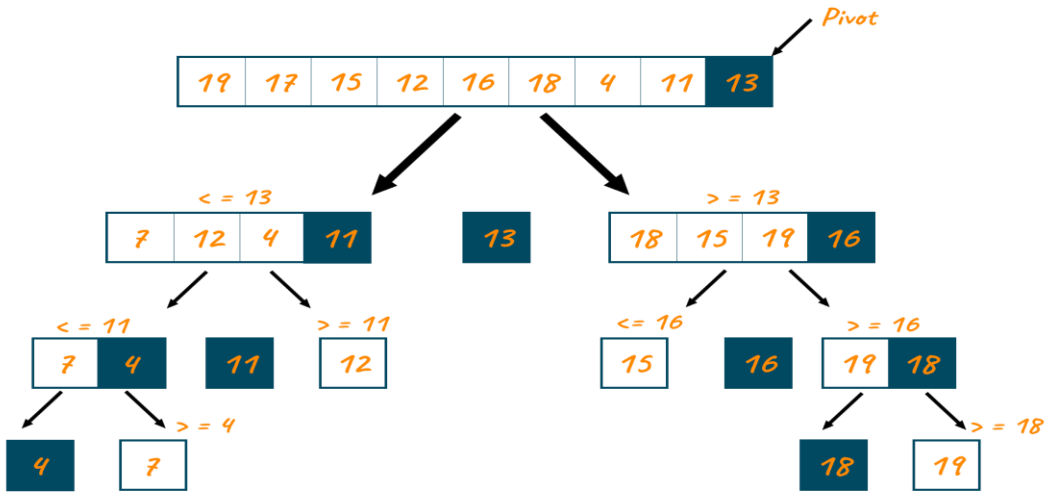
```c
void MergeSort(int *A, int p, int r){
    int q;
    if (p < r)
    {
        q = trunc((p + r) / 2); //Se obtiene la mitad (parte entera) del arreglo

        //Recursividad hasta llegar a 2 elementos
        MergeSort(A, p, q);     //Se ordenan de 0 a la mitad del arreglo
        MergeSort(A, q + 1, r); //Se ordenan de la mitad al final del arreglo
        Merge(A, p, q, r);      //Junta los mini arreglos que se van arreglando
    }
```

Imagen: Algoritmo de Marge2.0 implementado en C

QUICK SORT

Este algoritmo trabaja de la siguiente manera:

- Elige un elemento del conjunto a ordenar, al que llamaremos PIVOTE.
- Reorganizaremos los demás elementos de la lista a cada costado del pivote, de manera que de un lado queden todos los que sean menores que él, y del otro los mayores. Los elementos que sean iguales al pivote pueden ser colocados en cualquier lado, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponde de la lista ordenada.
- La lista queda separada en 2 sublistas, una formada por los elementos a la izquierda y otros a la derecha.
- Repite este proceso de forma recursiva para cada sublista, mientras éstas contengan más de un elemento. Una vez terminado el proceso todos los elementos estarán ordenados.



```c
void intercambiar(int *A, int i, int j){
    int temp;
    temp = A[j];
    A[j] = A[i];
    A[i] = temp;
}

int Pivot(int *A, int p, int r){
    int i = p + 1; //Segundo elemento del arreglo
    int j = r; //Tam del arreglo
    int piv = A[p]; //El pivote es la primer posicion del arreglo

    while (1)
    {
        while (A[i] < piv && i < r)
            i++;

        while (A[j] > piv)
            j--;

        if (i < j)
            intercambiar(A, i, j);

        else
        {
            intercambiar(A, p, j);
            return j;
        }
    }
}

void QuickSort(int *A,int p, int r){
    if (p < r)
    {
        int j = Pivot(A, p, r); //Se obtiene el pivote
        QuickSort(A, p, j - 1); //Se ordena de 0 al (pivote-1)
        QuickSort(A, j + 1, r); //Se ordena del pivote +1 hasta el final
    }
}
```

Imagen: Algoritmo de quick implementado en C

ACTIVIDADES

1. Entender el funcionamiento de cada uno de los algoritmos de ordenamiento a implementar.
2. Programar en C de forma separada cada uno de los algoritmos de ordenamiento.
3. Diseñar cada programa de manera que este sea capaz de recibir por argumento en el main() el tamaño del arreglo a ordenar.
4. Realizar pruebas de cada uno de los algoritmos para diferentes N'S y medir los tiempos de ejecución: 500 – 1000 – 5,000 – 10,000 – 50,000 – 100,000 – 200,000 – 500,000.

RESULTADOS OBTENIDOS EN LA EJECUCIÓN DE CADA UNO DE LOS ALGORITMOS.

INSERCION:

500 números

```
A[480]=2046404365
A[481]=2050294059
A[482]=2051915317
A[483]=2059110405
A[484]=2065894811
A[485]=2069141818
A[486]=2070411497
A[487]=2075455330
A[488]=2079048503
A[489]=2083630786
A[490]=2090025499
A[491]=2104337497
A[492]=2105472203
A[493]=2107886015
A[494]=2108162151
A[495]=2115141684
A[496]=2126220799
A[497]=2127921975
A[498]=2135997964
A[499]=2140056905

Tiempo medido: 0.000000000000000 segundos.
```

1000 números

```
A[980]=2108162151
A[981]=2109309777
A[982]=2112785333
A[983]=2114572597
A[984]=2115141684
A[985]=2117451796
A[986]=2119844088
A[987]=2126220799
A[988]=2127921975
A[989]=2135107075
A[990]=2135992376
A[991]=2135997964
A[992]=2139082770
A[993]=2139538255
A[994]=2140056905
A[995]=2140639991
A[996]=2144305891
A[997]=2144938480
A[998]=2144978749
A[999]=2147443379

Tiempo medido: 0.000000000000000 segundos.
```

5000 números

```
A[4980]=2141052447
A[4981]=2141625953
A[4982]=2142181636
A[4983]=2142603362
A[4984]=2143216287
A[4985]=2144305891
A[4986]=2144321335
A[4987]=2144687421
A[4988]=2144707086
A[4989]=2144938480
A[4990]=2144978749
A[4991]=2145079404
A[4992]=2145638446
A[4993]=2146062213
A[4994]=2146502522
A[4995]=2146810749
A[4996]=2147058635
A[4997]=2147352342
A[4998]=2147443379
A[4999]=2147458290

Tiempo medido: 0.025000000000000 segundos.
```

10,000 números

```
A[9980]=2144707086
A[9981]=2144870250
A[9982]=2144938480
A[9983]=2144978749
A[9984]=2145079404
A[9985]=2145373020
A[9986]=2145638446
A[9987]=2146062213
A[9988]=2146165608
A[9989]=2146202534
A[9990]=2146258922
A[9991]=2146473261
A[9992]=2146502522
A[9993]=2146781696
A[9994]=2146810749
A[9995]=2146932985
A[9996]=2147058635
A[9997]=2147352342
A[9998]=2147443379
A[9999]=2147458290

Tiempo medido: 0.069000000000000 segundos.
```

50,000 números

```
A[49980]=2146774272
A[49981]=2146781696
A[49982]=2146803268
A[49983]=2146810749
A[49984]=2146839753
A[49985]=2146877572
A[49986]=2146932985
A[49987]=2147058635
A[49988]=2147075090
A[49989]=2147106500
A[49990]=2147138326
A[49991]=2147220500
A[49992]=2147224321
A[49993]=2147244774
A[49994]=2147352342
A[49995]=2147373031
A[49996]=2147443379
A[49997]=2147445108
A[49998]=2147458290
A[49999]=2147465711

Tiempo medido: 1.664000000000000 segundos.
```

100,000 números

```
A[99980]=2147107736
A[99981]=2147128315
A[99982]=2147138326
A[99983]=2147170548
A[99984]=2147220500
A[99985]=2147224321
A[99986]=2147244774
A[99987]=2147253772
A[99988]=2147266441
A[99989]=2147272532
A[99990]=2147287994
A[99991]=2147352342
A[99992]=2147373031
A[99993]=2147395653
A[99994]=2147417281
A[99995]=2147433626
A[99996]=2147443379
A[99997]=2147445108
A[99998]=2147458290
A[99999]=2147465711

Tiempo medido: 6.608000000000000 segundos.
```

200,000 números

```
A[199980]=2147224321
A[199981]=2147239609
A[199982]=2147244774
A[199983]=2147253772
A[199984]=2147266441
A[199985]=2147272532
A[199986]=2147287994
A[199987]=2147294693
A[199988]=2147310359
A[199989]=2147352342
A[199990]=2147373031
A[199991]=2147395653
A[199992]=2147412455
A[199993]=2147417281
A[199994]=2147433626
A[199995]=2147437287
A[199996]=2147443379
A[199997]=2147445108
A[199998]=2147458290
A[199999]=2147465711

Tiempo medido: 26.504999999999999 segundos.
```

500,000 números

```
A[499980]=2147412455
A[499981]=2147414656
A[499982]=2147417281
A[499983]=2147419740
A[499984]=2147422546
A[499985]=2147423637
A[499986]=2147429538
A[499987]=2147430726
A[499988]=2147431463
A[499989]=2147433626
A[499990]=2147437287
A[499991]=2147443379
A[499992]=2147445108
A[499993]=2147446615
A[499994]=2147447520
A[499995]=2147450467
A[499996]=2147454139
A[499997]=2147458290
A[499998]=2147459301
A[499999]=2147465711

Tiempo medido: 165.080000000000010 segundos.
```

Gráfica que compara los tiempos del algoritmo de inserción

INSERCIÓN	
Cantidad de números a ordenar	Tiempo (en segundos)
500	0.000 segundos.
1,000	0.000 segundos.
5,000	0.0250 segundos.
10,000	0.0690 segundos.
50,000	1.6640 segundos.
100,000	6.6080 segundos.
200,000	26.5050 segundos.
500,000	165.800 segundos.

BURBUJA:

500 números

```
A[480]=2046404365
A[481]=2050294059
A[482]=2051915317
A[483]=2059110405
A[484]=2065894811
A[485]=2069141818
A[486]=2070411497
A[487]=2075455330
A[488]=2079048503
A[489]=2083630786
A[490]=2090025499
A[491]=2104337497
A[492]=2105472203
A[493]=2107886015
A[494]=2108162151
A[495]=2115141684
A[496]=2126220799
A[497]=2127921975
A[498]=2135997964
A[499]=2140056905

Tiempo medido: 0.000000000000000 segundos.
```

1000 números

```
A[980]=2108162151
A[981]=2109309777
A[982]=2112785333
A[983]=2114572597
A[984]=2115141684
A[985]=2117451796
A[986]=2119844088
A[987]=2126220799
A[988]=2127921975
A[989]=2135107075
A[990]=2135992376
A[991]=2135997964
A[992]=2139082770
A[993]=2139538255
A[994]=2140056905
A[995]=2140639991
A[996]=2144305891
A[997]=2144938480
A[998]=2144978749
A[999]=2147443379

Tiempo medido: 0.009000000000000 segundos.
```

5000 números

```
A[4980]=2141052447
A[4981]=2141625953
A[4982]=2142181636
A[4983]=2142603362
A[4984]=2143216287
A[4985]=2144305891
A[4986]=2144321335
A[4987]=2144687421
A[4988]=2144707086
A[4989]=2144938480
A[4990]=2144978749
A[4991]=2145079404
A[4992]=2145638446
A[4993]=2146062213
A[4994]=2146502522
A[4995]=2146810749
A[4996]=2147058635
A[4997]=2147352342
A[4998]=2147443379
A[4999]=2147458290

Tiempo medido: 0.130000000000000 segundos.
```

10,000 números

```
A[9980]=2144707086
A[9981]=2144870250
A[9982]=2144938480
A[9983]=2144978749
A[9984]=2145079404
A[9985]=2145373020
A[9986]=2145638446
A[9987]=2146062213
A[9988]=2146165608
A[9989]=2146202534
A[9990]=2146258922
A[9991]=2146473261
A[9992]=2146502522
A[9993]=2146781696
A[9994]=2146810749
A[9995]=2146932985
A[9996]=2147058635
A[9997]=2147352342
A[9998]=2147443379
A[9999]=2147458290

Tiempo medido: 0.545000000000000 segundos.
```

50,000 números

```
A[49980]=2146774272
A[49981]=2146781696
A[49982]=2146803268
A[49983]=2146810749
A[49984]=2146839753
A[49985]=2146877572
A[49986]=2146932985
A[49987]=2147058635
A[49988]=2147075090
A[49989]=2147106500
A[49990]=2147138326
A[49991]=2147220500
A[49992]=2147224321
A[49993]=2147244774
A[49994]=2147352342
A[49995]=2147373031
A[49996]=2147443379
A[49997]=2147445108
A[49998]=2147458290
A[49999]=2147465711

Tiempo medido: 7.099000000000000 segundos.
```

100,000 números

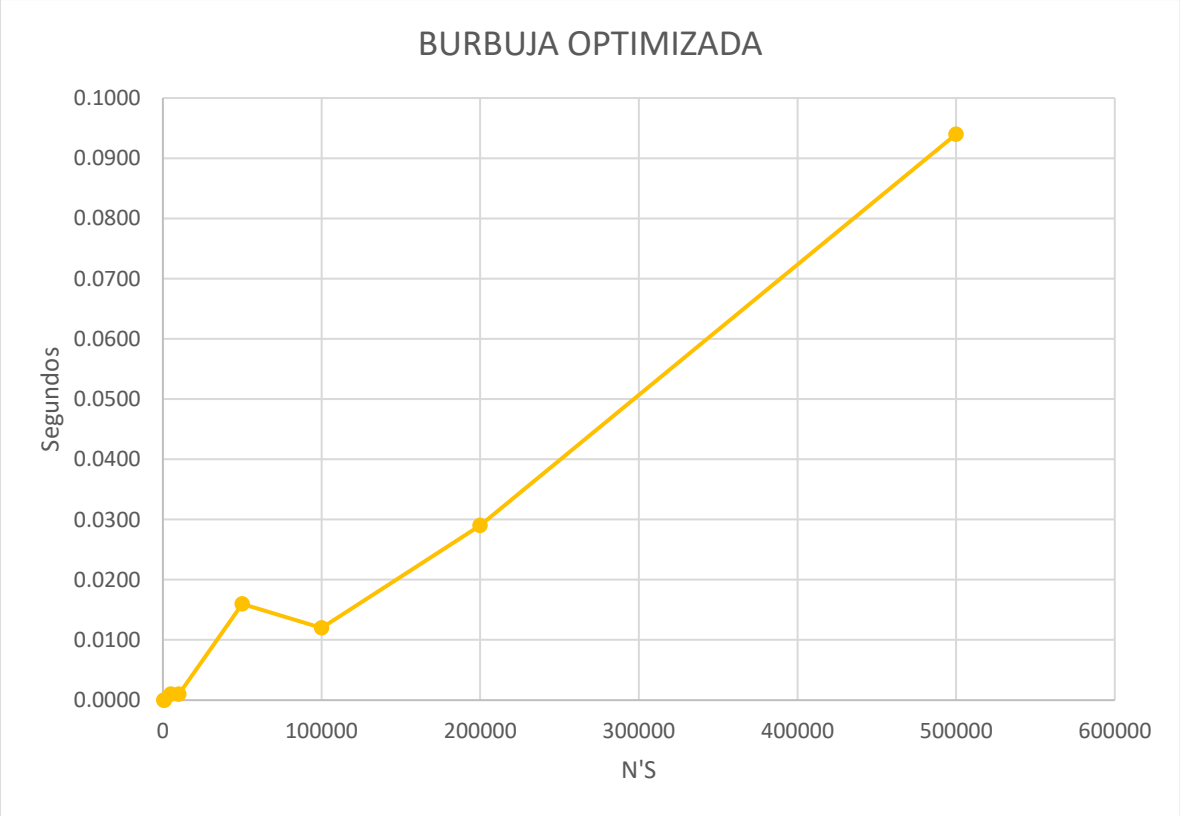
```
A[99980]=2147107736
A[99981]=2147128315
A[99982]=2147138326
A[99983]=2147170548
A[99984]=2147220500
A[99985]=2147224321
A[99986]=2147244774
A[99987]=2147253772
A[99988]=2147266441
A[99989]=2147272532
A[99990]=2147287994
A[99991]=2147352342
A[99992]=2147373031
A[99993]=2147395653
A[99994]=2147417281
A[99995]=2147433626
A[99996]=2147443379
A[99997]=2147445108
A[99998]=2147458290
A[99999]=2147465711

Tiempo medido: 28.771000000000001 segundos.
```

200,000 números

```
A[199980]=2147224321
A[199981]=2147239609
A[199982]=2147244774
A[199983]=2147253772
A[199984]=2147266441
A[199985]=2147272532
A[199986]=2147287994
A[199987]=2147294693
A[199988]=2147310359
A[199989]=2147352342
A[199990]=2147373031
A[199991]=2147395653
A[199992]=2147412455
A[199993]=2147417281
A[199994]=2147433626
A[199995]=2147437287
A[199996]=2147443379
A[199997]=2147445108
A[199998]=2147458290
A[199999]=2147465711

Tiempo medido: 121.652000000000000 segundos.
```

500,000 números

```
A[499980]=2147412455
A[499981]=2147414656
A[499982]=2147417281
A[499983]=2147419740
A[499984]=2147422546
A[499985]=2147423637
A[499986]=2147429538
A[499987]=2147430726
A[499988]=2147431463
A[499989]=2147433626
A[499990]=2147437287
A[499991]=2147443379
A[499992]=2147445108
A[499993]=2147446615
A[499994]=2147447520
A[499995]=2147450467
A[499996]=2147454139
A[499997]=2147458290
A[499998]=2147459301
A[499999]=2147465711

Tiempo medido: 757.644999999999980 segundos.
```



Burbuja optimizada	
Cantidad de números a ordenar	Tiempo (en segundos)
500	0.0000 segundos.
1,000	0.0090 segundos.
5,000	0.1300 segundos.
10,000	0.5450 segundos.
50,000	7.0990 segundos.
100,000	28.7710 segundos.
200,000	121.6520 segundos.
500,000	757.644999 segundos.

SELECCIÓN:

500 números

```
A[480]=2046404365
A[481]=2050294059
A[482]=2051915317
A[483]=2059110405
A[484]=2065894811
A[485]=2069141818
A[486]=2070411497
A[487]=2075455330
A[488]=2079048503
A[489]=2083630786
A[490]=2090025499
A[491]=2104337497
A[492]=2105472203
A[493]=2107886015
A[494]=2108162151
A[495]=2115141684
A[496]=2126220799
A[497]=2127921975
A[498]=2135997964
A[499]=2140056905

Tiempo medido: 0.000000000000000 segundos.
```

1,000 números

```
A[980]=2108162151
A[981]=2109309777
A[982]=2112785333
A[983]=2114572597
A[984]=2115141684
A[985]=2117451796
A[986]=2119844088
A[987]=2126220799
A[988]=2127921975
A[989]=2135107075
A[990]=2135992376
A[991]=2135997964
A[992]=2139082770
A[993]=2139538255
A[994]=2140056905
A[995]=2140639991
A[996]=2144305891
A[997]=2144938480
A[998]=2144978749
A[999]=2147443379

Tiempo medido: 0.001000000000000 segundos.
```

5,000 números

```
A[4980]=2141052447
A[4981]=2141625953
A[4982]=2142181636
A[4983]=2142603362
A[4984]=2143216287
A[4985]=2144305891
A[4986]=2144321335
A[4987]=2144687421
A[4988]=2144707086
A[4989]=2144938480
A[4990]=2144978749
A[4991]=2145079404
A[4992]=2145638446
A[4993]=2146062213
A[4994]=2146502522
A[4995]=2146810749
A[4996]=2147058635
A[4997]=2147352342
A[4998]=2147443379
A[4999]=2147458290

Tiempo medido: 0.030000000000000 segundos.
```

10,000 números

```
A[9980]=2144707086
A[9981]=2144870250
A[9982]=2144938480
A[9983]=2144978749
A[9984]=2145079404
A[9985]=2145373020
A[9986]=2145638446
A[9987]=2146062213
A[9988]=2146165608
A[9989]=2146202534
A[9990]=2146258922
A[9991]=2146473261
A[9992]=2146502522
A[9993]=2146781696
A[9994]=2146810749
A[9995]=2146932985
A[9996]=2147058635
A[9997]=2147352342
A[9998]=2147443379
A[9999]=2147458290

Tiempo medido: 0.104000000000000 segundos.
```

50,000 números

```
A[49980]=2146774272
A[49981]=2146781696
A[49982]=2146803268
A[49983]=2146810749
A[49984]=2146839753
A[49985]=2146877572
A[49986]=2146932985
A[49987]=2147058635
A[49988]=2147075090
A[49989]=2147106500
A[49990]=2147138326
A[49991]=2147220500
A[49992]=2147224321
A[49993]=2147244774
A[49994]=2147352342
A[49995]=2147373031
A[49996]=2147443379
A[49997]=2147445108
A[49998]=2147458290
A[49999]=2147465711

Tiempo medido: 3.183000000000000 segundos.
```

100,000 números

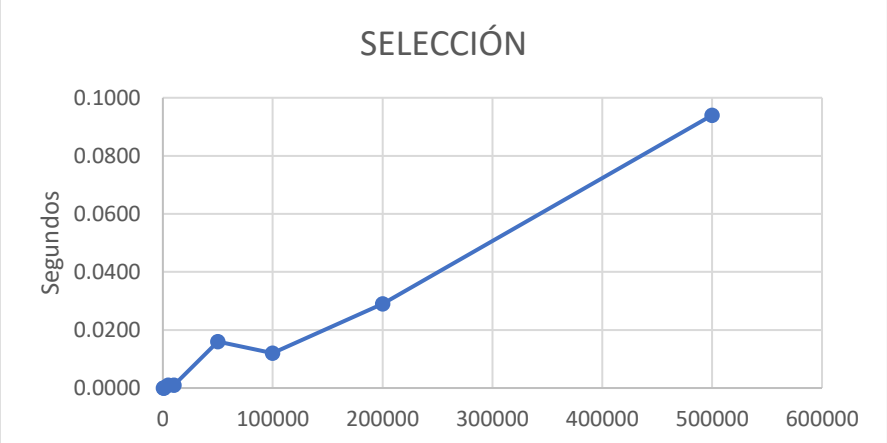
```
A[99980]=2147107736
A[99981]=2147128315
A[99982]=2147138326
A[99983]=2147170548
A[99984]=2147220500
A[99985]=2147224321
A[99986]=2147244774
A[99987]=2147253772
A[99988]=2147266441
A[99989]=2147272532
A[99990]=2147287994
A[99991]=2147352342
A[99992]=2147373031
A[99993]=2147395653
A[99994]=2147417281
A[99995]=2147433626
A[99996]=2147443379
A[99997]=2147445108
A[99998]=2147458290
A[99999]=2147465711

Tiempo medido: 12.878000000000000 segundos.
```

200,000 números

```
A[199980]=2147224321
A[199981]=2147239609
A[199982]=2147244774
A[199983]=2147253772
A[199984]=2147266441
A[199985]=2147272532
A[199986]=2147287994
A[199987]=2147294693
A[199988]=2147310359
A[199989]=2147352342
A[199990]=2147373031
A[199991]=2147395653
A[199992]=2147412455
A[199993]=2147417281
A[199994]=2147433626
A[199995]=2147437287
A[199996]=2147443379
A[199997]=2147445108
A[199998]=2147458290
A[199999]=2147465711

Tiempo medido: 39.326999999999998 segundos.
```

500,000 números

```
A[499980]=2147412455
A[499981]=2147414656
A[499982]=2147417281
A[499983]=2147419740
A[499984]=2147422546
A[499985]=2147423637
A[499986]=2147429538
A[499987]=2147430726
A[499988]=2147431463
A[499989]=2147433626
A[499990]=2147437287
A[499991]=2147443379
A[499992]=2147445108
A[499993]=2147446615
A[499994]=2147447520
A[499995]=2147450467
A[499996]=2147454139
A[499997]=2147458290
A[499998]=2147459301
A[499999]=2147465711

Tiempo medido: 291.632000000000010 segundos.
```



Selección	
Cantidad de números a ordenar	Tiempo (en segundos)
500	0.0000 segundos.
1,000	0.0010 segundos.
5,000	0.0300 segundos.
10,000	0.1040 segundos.
50,000	3.1830 segundos.
100,000	12.8780 segundos.
200,000	39.3269 segundos.
500,000	757.6449 segundos.

SHELL SORT:

500 números

```
A[480]=2046404365
A[481]=2050294059
A[482]=2051915317
A[483]=2059110405
A[484]=2065894811
A[485]=2069141818
A[486]=2070411497
A[487]=2075455330
A[488]=2079048503
A[489]=2083630786
A[490]=2090025499
A[491]=2104337497
A[492]=2105472203
A[493]=2107886015
A[494]=2108162151
A[495]=2115141684
A[496]=2126220799
A[497]=2127921975
A[498]=2135997964
A[499]=2140056905

Tiempo medido: 0.000000000000000 segundos.
```

1,000 números

```
A[980]=2108162151
A[981]=2109309777
A[982]=2112785333
A[983]=2114572597
A[984]=2115141684
A[985]=2117451796
A[986]=2119844088
A[987]=2126220799
A[988]=2127921975
A[989]=2135107075
A[990]=2135992376
A[991]=2135997964
A[992]=2139082770
A[993]=2139538255
A[994]=2140056905
A[995]=2140639991
A[996]=2144305891
A[997]=2144938480
A[998]=2144978749
A[999]=2147443379

Tiempo medido: 0.000000000000000 segundos.
```

5,000 números

```
A[4980]=2141052447
A[4981]=2141625953
A[4982]=2142181636
A[4983]=2142603362
A[4984]=2143216287
A[4985]=2144305891
A[4986]=2144321335
A[4987]=2144687421
A[4988]=2144707086
A[4989]=2144938480
A[4990]=2144978749
A[4991]=2145079404
A[4992]=2145638446
A[4993]=2146062213
A[4994]=2146502522
A[4995]=2146810749
A[4996]=2147058635
A[4997]=2147352342
A[4998]=2147443379
A[4999]=2147458290

Tiempo medido: 0.004000000000000 segundos.
```

10,000 números

```
A[9980]=2144707086
A[9981]=2144870250
A[9982]=2144938480
A[9983]=2144978749
A[9984]=2145079404
A[9985]=2145373020
A[9986]=2145638446
A[9987]=2146062213
A[9988]=2146165608
A[9989]=2146202534
A[9990]=2146258922
A[9991]=2146473261
A[9992]=2146502522
A[9993]=2146781696
A[9994]=2146810749
A[9995]=2146932985
A[9996]=2147058635
A[9997]=2147352342
A[9998]=2147443379
A[9999]=2147458290

Tiempo medido: 0.005000000000000 segundos.
```

50,000 números

```
A[49980]=2146774272
A[49981]=2146781696
A[49982]=2146803268
A[49983]=2146810749
A[49984]=2146839753
A[49985]=2146877572
A[49986]=2146932985
A[49987]=2147058635
A[49988]=2147075090
A[49989]=2147106500
A[49990]=2147138326
A[49991]=2147220500
A[49992]=2147224321
A[49993]=2147244774
A[49994]=2147352342
A[49995]=2147373031
A[49996]=2147443379
A[49997]=2147445108
A[49998]=2147458290
A[49999]=2147465711

Tiempo medido: 0.039000000000000 segundos.
```

100,000 números

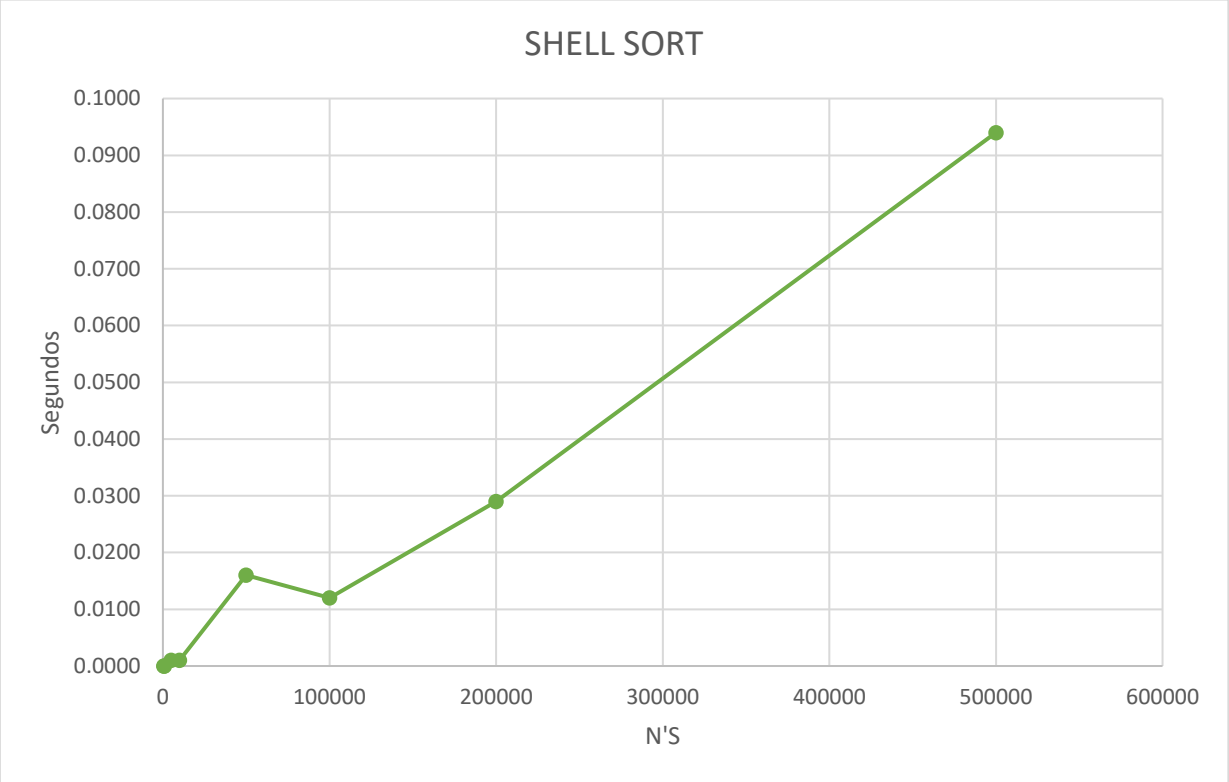
```
A[99980]=2147107736
A[99981]=2147128315
A[99982]=2147138326
A[99983]=2147170548
A[99984]=2147220500
A[99985]=2147224321
A[99986]=2147244774
A[99987]=2147253772
A[99988]=2147266441
A[99989]=2147272532
A[99990]=2147287994
A[99991]=2147352342
A[99992]=2147373031
A[99993]=2147395653
A[99994]=2147417281
A[99995]=2147433626
A[99996]=2147443379
A[99997]=2147445108
A[99998]=2147458290
A[99999]=2147465711

Tiempo medido: 0.101000000000000 segundos.
```

200,000 números

```
A[199980]=2147224321
A[199981]=2147239609
A[199982]=2147244774
A[199983]=2147253772
A[199984]=2147266441
A[199985]=2147272532
A[199986]=2147287994
A[199987]=2147294693
A[199988]=2147310359
A[199989]=2147352342
A[199990]=2147373031
A[199991]=2147395653
A[199992]=2147412455
A[199993]=2147417281
A[199994]=2147433626
A[199995]=2147437287
A[199996]=2147443379
A[199997]=2147445108
A[199998]=2147458290
A[199999]=2147465711

Tiempo medido: 0.194000000000000 segundos.
```

500,000 números

```
A[499980]=2147412455
A[499981]=2147414656
A[499982]=2147417281
A[499983]=2147419740
A[499984]=2147422546
A[499985]=2147423637
A[499986]=2147429538
A[499987]=2147430726
A[499988]=2147431463
A[499989]=2147433626
A[499990]=2147437287
A[499991]=2147443379
A[499992]=2147445108
A[499993]=2147446615
A[499994]=2147447520
A[499995]=2147450467
A[499996]=2147454139
A[499997]=2147458290
A[499998]=2147459301
A[499999]=2147465711

Tiempo medido: 0.576000000000000 segundos.
```



Shell Sort	
Cantidad de números a ordenar	Tiempo (en segundos)
500	0.0000 segundos.
1,000	0.0000 segundos.
5,000	0.0040 segundos.
10,000	0.0050 segundos.
50,000	0.0390 segundos.
100,000	0.1010 segundos.
200,000	0.1940 segundos.
500,000	0. segundos.

MERGE SORT:

500 números

```
A[480]=2046404365
A[481]=2050294059
A[482]=2051915317
A[483]=2059110405
A[484]=2065894811
A[485]=2069141818
A[486]=2070411497
A[487]=2075455330
A[488]=2079048503
A[489]=2083630786
A[490]=2090025499
A[491]=2104337497
A[492]=2105472203
A[493]=2107886015
A[494]=2108162151
A[495]=2115141684
A[496]=2126220799
A[497]=2127921975
A[498]=2135997964
A[499]=2140056905

Tiempo medido: 0.001000000000000 segundos.
```

1,000 números

```
A[980]=2108162151
A[981]=2109309777
A[982]=2112785333
A[983]=2114572597
A[984]=2115141684
A[985]=2117451796
A[986]=2119844088
A[987]=2126220799
A[988]=2127921975
A[989]=2135107075
A[990]=2135992376
A[991]=2135997964
A[992]=2139082770
A[993]=2139538255
A[994]=2140056905
A[995]=2140639991
A[996]=2144305891
A[997]=2144938480
A[998]=2144978749
A[999]=2147443379

Tiempo medido: 0.001000000000000 segundos.
```

5,000 números

```
A[4980]=2141052447
A[4981]=2141625953
A[4982]=2142181636
A[4983]=2142603362
A[4984]=2143216287
A[4985]=2144305891
A[4986]=2144321335
A[4987]=2144687421
A[4988]=2144707086
A[4989]=2144938480
A[4990]=2144978749
A[4991]=2145079404
A[4992]=2145638446
A[4993]=2146062213
A[4994]=2146502522
A[4995]=2146810749
A[4996]=2147058635
A[4997]=2147352342
A[4998]=2147443379
A[4999]=2147458290

Tiempo medido: 0.002000000000000 segundos.
```

10,000 números

```
A[9980]=2144707086
A[9981]=2144870250
A[9982]=2144938480
A[9983]=2144978749
A[9984]=2145079404
A[9985]=2145373020
A[9986]=2145638446
A[9987]=2146062213
A[9988]=2146165608
A[9989]=2146202534
A[9990]=2146258922
A[9991]=2146473261
A[9992]=2146502522
A[9993]=2146781696
A[9994]=2146810749
A[9995]=2146932985
A[9996]=2147058635
A[9997]=2147352342
A[9998]=2147443379
A[9999]=2147458290

Tiempo medido: 0.004000000000000 segundos.
```

50,000 números

```
A[49980]=2146774272
A[49981]=2146781696
A[49982]=2146803268
A[49983]=2146810749
A[49984]=2146839753
A[49985]=2146877572
A[49986]=2146932985
A[49987]=2147058635
A[49988]=2147075090
A[49989]=2147106500
A[49990]=2147138326
A[49991]=2147220500
A[49992]=2147224321
A[49993]=2147244774
A[49994]=2147352342
A[49995]=2147373031
A[49996]=2147443379
A[49997]=2147445108
A[49998]=2147458290
A[49999]=2147465711

Tiempo medido: 0.023000000000000 segundos.
```

100,000 números

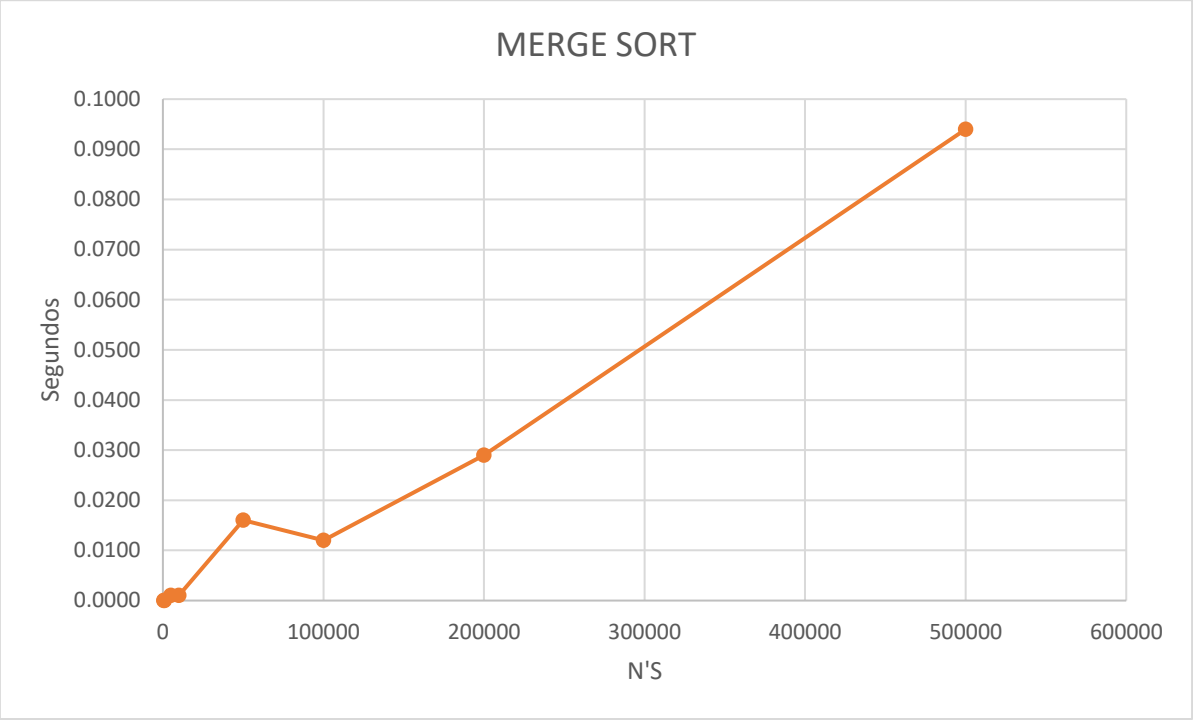
```
A[99980]=2147107736
A[99981]=2147128315
A[99982]=2147138326
A[99983]=2147170548
A[99984]=2147220500
A[99985]=2147224321
A[99986]=2147244774
A[99987]=2147253772
A[99988]=2147266441
A[99989]=2147272532
A[99990]=2147287994
A[99991]=2147352342
A[99992]=2147373031
A[99993]=2147395653
A[99994]=2147417281
A[99995]=2147433626
A[99996]=2147443379
A[99997]=2147445108
A[99998]=2147458290
A[99999]=2147465711

Tiempo medido: 0.035000000000000 segundos.
```

200,000 números

```
A[199980]=2147224321
A[199981]=2147239609
A[199982]=2147244774
A[199983]=2147253772
A[199984]=2147266441
A[199985]=2147272532
A[199986]=2147287994
A[199987]=2147294693
A[199988]=2147310359
A[199989]=2147352342
A[199990]=2147373031
A[199991]=2147395653
A[199992]=2147412455
A[199993]=2147417281
A[199994]=2147433626
A[199995]=2147437287
A[199996]=2147443379
A[199997]=2147445108
A[199998]=2147458290
A[199999]=2147465711

Tiempo medido: 0.066000000000000 segundos.
```

500,000 números

```
A[499980]=2147412455
A[499981]=2147414656
A[499982]=2147417281
A[499983]=2147419740
A[499984]=2147422546
A[499985]=2147423637
A[499986]=2147429538
A[499987]=2147430726
A[499988]=2147431463
A[499989]=2147433626
A[499990]=2147437287
A[499991]=2147443379
A[499992]=2147445108
A[499993]=2147446615
A[499994]=2147447520
A[499995]=2147450467
A[499996]=2147454139
A[499997]=2147458290
A[499998]=2147459301
A[499999]=2147465711

Tiempo medido: 0.149000000000000 segundos.
```



Merge Sort	
Cantidad de números a ordenar	Tiempo (en segundos)
500	0.0010 segundos.
1,000	0.0010 segundos.
5,000	0.0020 segundos.
10,000	0.0040 segundos.
50,000	0.0230 segundos.
100,000	0.0350 segundos.
200,000	0.0660 segundos.
500,000	0.1490 segundos.

QUICK SORT:

500 números

```
A[480]=2046404365
A[481]=2050294059
A[482]=2051915317
A[483]=2059110405
A[484]=2065894811
A[485]=2069141818
A[486]=2070411497
A[487]=2075455330
A[488]=2079048503
A[489]=2083630786
A[490]=2090025499
A[491]=2104337497
A[492]=2105472203
A[493]=2107886015
A[494]=2108162151
A[495]=2115141684
A[496]=2126220799
A[497]=2127921975
A[498]=2135997964
A[499]=2140056905

Tiempo medido: 0.000000000000000 segundos.
```

1,000 números

```
A[980]=2108162151
A[981]=2109309777
A[982]=2112785333
A[983]=2114572597
A[984]=2115141684
A[985]=2117451796
A[986]=2119844088
A[987]=2126220799
A[988]=2127921975
A[989]=2135107075
A[990]=2135992376
A[991]=2135997964
A[992]=2139082770
A[993]=2139538255
A[994]=2140056905
A[995]=2140639991
A[996]=2144305891
A[997]=2144938480
A[998]=2144978749
A[999]=2147443379

Tiempo medido: 0.000000000000000 segundos.
```

5,000 números

```
A[4980]=2141052447
A[4981]=2141625953
A[4982]=2142181636
A[4983]=2142603362
A[4984]=2143216287
A[4985]=2144305891
A[4986]=2144321335
A[4987]=2144687421
A[4988]=2144707086
A[4989]=2144938480
A[4990]=2144978749
A[4991]=2145079404
A[4992]=2145638446
A[4993]=2146062213
A[4994]=2146502522
A[4995]=2146810749
A[4996]=2147058635
A[4997]=2147352342
A[4998]=2147443379
A[4999]=2147458290

Tiempo medido: 0.001000000000000 segundos.
```

10,000 números

```
A[9980]=2144707086
A[9981]=2144870250
A[9982]=2144938480
A[9983]=2144978749
A[9984]=2145079404
A[9985]=2145373020
A[9986]=2145638446
A[9987]=2146062213
A[9988]=2146165608
A[9989]=2146202534
A[9990]=2146258922
A[9991]=2146473261
A[9992]=2146502522
A[9993]=2146781696
A[9994]=2146810749
A[9995]=2146932985
A[9996]=2147058635
A[9997]=2147352342
A[9998]=2147443379
A[9999]=2147458290

Tiempo medido: 0.001000000000000 segundos.
```

50,000 números

```
A[49980]=2146774272
A[49981]=2146781696
A[49982]=2146803268
A[49983]=2146810749
A[49984]=2146839753
A[49985]=2146877572
A[49986]=2146932985
A[49987]=2147058635
A[49988]=2147075090
A[49989]=2147106500
A[49990]=2147138326
A[49991]=2147220500
A[49992]=2147224321
A[49993]=2147244774
A[49994]=2147352342
A[49995]=2147373031
A[49996]=2147443379
A[49997]=2147445108
A[49998]=2147458290
A[49999]=2147465711

Tiempo medido: 0.016000000000000 segundos.
```

100,000 números

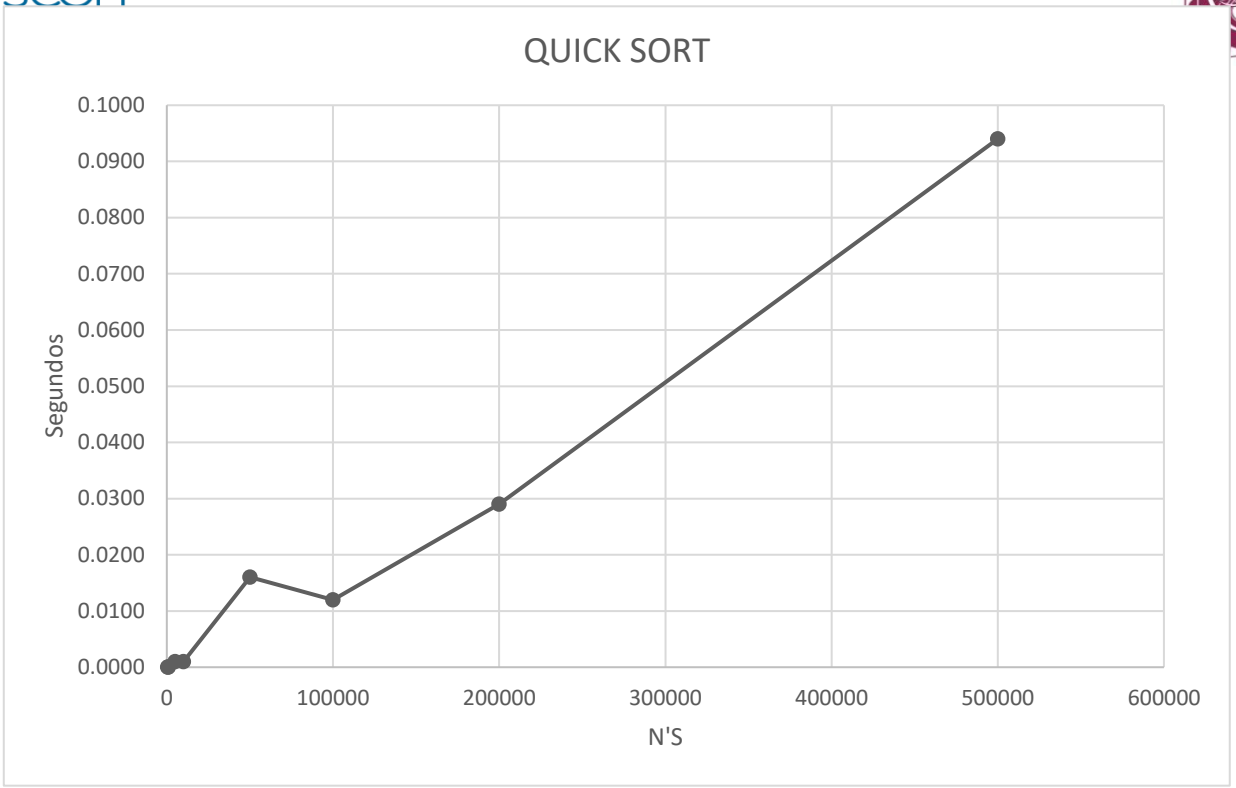
```
A[99980]=2147107736
A[99981]=2147128315
A[99982]=2147138326
A[99983]=2147170548
A[99984]=2147220500
A[99985]=2147224321
A[99986]=2147244774
A[99987]=2147253772
A[99988]=2147266441
A[99989]=2147272532
A[99990]=2147287994
A[99991]=2147352342
A[99992]=2147373031
A[99993]=2147395653
A[99994]=2147417281
A[99995]=2147433626
A[99996]=2147443379
A[99997]=2147445108
A[99998]=2147458290
A[99999]=2147465711

Tiempo medido: 0.012000000000000 segundos.
```

200,000 números

```
A[199980]=2147224321
A[199981]=2147239609
A[199982]=2147244774
A[199983]=2147253772
A[199984]=2147266441
A[199985]=2147272532
A[199986]=2147287994
A[199987]=2147294693
A[199988]=2147310359
A[199989]=2147352342
A[199990]=2147373031
A[199991]=2147395653
A[199992]=2147412455
A[199993]=2147417281
A[199994]=2147433626
A[199995]=2147437287
A[199996]=2147443379
A[199997]=2147445108
A[199998]=2147458290
A[199999]=2147465711

Tiempo medido: 0.029000000000000 segundos.
```

500,000 números

```
A[499980]=2147412455
A[499981]=2147414656
A[499982]=2147417281
A[499983]=2147419740
A[499984]=2147422546
A[499985]=2147423637
A[499986]=2147429538
A[499987]=2147430726
A[499988]=2147431463
A[499989]=2147433626
A[499990]=2147437287
A[499991]=2147443379
A[499992]=2147445108
A[499993]=2147446615
A[499994]=2147447520
A[499995]=2147450467
A[499996]=2147454139
A[499997]=2147458290
A[499998]=2147459301
A[499999]=2147465711

Tiempo medido: 0.094000000000000 segundos.
```



QUICK SORT	
Cantidad de números a ordenar	Tiempo (en segundos)
500	0.0000 segundos.
1,000	0.0000 segundos.
5,000	0.0010 segundos.
10,000	0.0010 segundos.
50,000	0.0160 segundos.
100,000	0.0120 segundos.
200,000	0.0290 segundos.
500,000	0.1490 segundos.

Errores detectados

En general, no se detectó ningún error en los códigos que limite el funcionamiento de estos, sin embargo, tuvimos que hacer ligeros cambios al momento de implementar el código del algoritmo de ordenamiento QuickSort con respecto a su pseudocódigo mostrado en el material adjunto a la práctica; únicamente fue necesario modificar la función de Pivote, puesto a que las demás (QuickSort e Intercambiar), se implementaron al pie de la letra. Dichas modificaciones se muestran a continuación.

Pseudocódigo de la función Pivote:

```
Algoritmo Pivot(A, p, r)
    piv=A[p], i=p+1, j=r
    mientras (i<j)
          mientras A[i]<= piv y i<r hacer
               i++
          mientras A[j]> piv hacer
               j--
          Intercambiar(A,i,j)
    fin mientras
    Intercambiar(A,p,j)
    regresar j
fin Algoritmo
```

Implementación de la función Pivote:

```
88  int Pivot(int *A, int p, int r){
89      int i = p + 1; //Segundo elemento del arreglo
90      int j = r; //Tam del arreglo
91      int piv = A[p]; //El pivote es la primer posicion del arreglo
92
93      while (1)
94      {
95          while (A[i] < piv && i < r)
96              i++;
97
98          while (A[j] > piv)
99              j--;
100
101         if (i < j)
102             intercambiar(A, i, j);
103
104         else
105         {
106             intercambiar(A, p, j);
107             return j;
108         }
109     }
110 }
```

Como se puede observar en la comparativa, la diferencia entre lo que plantea el pseudocódigo y nuestra implementación, radica en el ciclo while.

Mientras que en el pseudocódigo se condiciona y al final del ciclo regresa el pivote, en nuestra implementación, el ciclo while siempre se ejecuta hasta que cae en el if que hace retornar el pivote y por lo tanto rompe el ciclo.

Estas modificaciones fueron hechas debido a que la transcripción del pseudocódigo original al código de implementación no ordenaba los números correctamente, entonces se optó por esta nueva solución, la cual a priori no nos causó problemas.

Entre otros ligeros problemas que tuvimos durante la implementación del código, uno de los más relevantes fue que al momento de llamar a la función del algoritmo en el main, nos salía un warning porque no estábamos declarando el arreglo que recibe como argumento la función de forma correcta. Este error se ejemplifica a continuación.

Forma incorrecta:

```
Inserccion(*A,n);
```

Forma correcta:

```
Inserccion(A,n);
```

Esto ya que en nuestra declaración de variables ya estábamos estableciendo que A es un apuntador a una dirección de memoria que contiene enteros, como podemos ver a continuación.

```
19      //Variables para el algoritmo
20      int i,j,n,*A;
```

Entonces, al haber establecido antes que “A” es un apuntador a una dirección de memoria, ya no hace falta poner el apuntador; porque nosotros teorizamos que, entonces nos estaríamos refiriendo a lo que contiene esa dirección de memoria y no a la dirección de memoria en sí.

No obstante, después de un par de revisiones al código fue como pudimos darle solución a este error de implementación y sacar nuestras conclusiones de su origen.

Posibles mejoras

Burbuja Optimizada

Dentro del if, con intercambiar el operador (<) por (>) basta para directamente descartar la segunda parte de la función, cuya labor es invertir el arreglo. Esto debido a que el pseudocódigo ordena de mayor a menor y nosotros lo requerimos de menor a mayor.

```
//Algoritmo de ordenamiento "Burbuja optimizada"
void BurbujaOptimizada(int *A,int n)
{
    int i=0,aux, cambios = 1;
    int j,k;

    while(i<=(n-1) && cambios==1){
            cambios = 0;
            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;
                cambios = 1;

                }
            }

            i++;
    }

    //Voltear el arreglo O(n), ya que el pseudocodigo nos da el ordenamiento invertido
    for(i=0,j=n-1;i<n/2;i++,j--)
    {
        aux=A[i];
        A[i]=A[j];
        A[j]=aux;
    }


    return;
}
```

Intercambiar (<) por (>)

Fin del algoritmo

Parte innecesaria

Merge Sort

Al consultar material para el mejor entendimiento de los algoritmos de ordenamiento, nos dimos cuenta de que el código autodocumentado nos puede hacer más fácil la tarea de tanto entender el algoritmo de ordenamiento, como de encontrar errores en él.

Si bien esta no es una mejora de eficiencia para el algoritmo, al tratarse de uno un poco más difícil de entender en principio que los anteriores de complejidad n^2, creemos que el incluir nombres de variables como “inicio”, “mitad” o “fin” desde el pseudocódigo facilitan la lectura del código tanto de quien lo programa como de quien quiera hacer modificaciones en él.

Ejemplo de estas modificaciones se pueden ver a continuación:

Pseudocódigo de MergeSort:

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

```
Merge(A
{
```

Pseudocódigo sugerido para MergeSort:

```
MergeSort(A,inicio,fin){
    if(inicio<fin){
        mitad = parteEntera((inicio+fin)/2);
        MergeSort(A,inicio,mitad);
        MergeSort(A,mitad+1,fin);
        Merge(A, inicio, mitad, fin);
    }
}
```

Conclusiones

Buendía Velazco Abel:

En lo personal, hacer la implementación y el análisis de los algoritmos de ordenamiento es algo muy útil, ya que teníamos expresadas las complejidades de cada uno teóricamente y gracias a que se hizo la implementación, logramos ver cómo es que se comportan las complejidades en razón de tiempo, en realidad donde más nos dimos cuenta cual fue la diferencia de tiempos fue cuando realizamos las pruebas desde 50,000 números, ya que desde ahí algunos algoritmos comienzan a tardar mucho más que otros (como lo es burbuja optimizada) y al igual me sorprendió que hay algoritmos que a pesar de la cantidad de números que da el usuario, estos son ordenados muy rápidamente (como lo es Quick Sort y Merge Sort), así que para mí el algoritmo que es más eficiente son Quick Sort y Merge Sort, ya que por sus complejidades de estos ordenan en un tiempo sorpresivamente muy corto (en menos de un segundo), cabe mencionar que la prueba que más tardo fue el algoritmo de burbuja, ya que a su complejidad y el número de comparaciones que hace al ordenar lo hace muy tardado, aunque en realidad es un buen método, así que para mí los métodos de ordenamiento que más conviene para ordenar grandes cantidades de números.

Carpio Becerra Erick Gustavo:

Puedo concluir que los algoritmos de ordenamiento forman una parte muy importante para la manipulación de información en el mundo de la computación. En la práctica me pude dar cuenta de que la complejidad del algoritmo que se usa no debe ser tratada a la ligera, ya que pudimos ver algunos de los algoritmos más ineficientes (los de complejidad n2) como el bubble sort, y conforme la entrada de números crecía, estos tardaban cada vez más y más tiempo en ordenar; así como los algoritmos de complejidad logarítmica, los cuales dejaban en ridículo a los anteriores. Me pareció impresionante como un bubble sort puede tardar hasta minutos en ordenar una entrada muy grande de números, mientras que el quick sort menos de un segundo con esta misma entrada.

En cuanto a la implementación de los códigos, algo a lo que casi no le tomaba importancia y ahora me doy cuenta de cuan crucial es, es la incorporación de comentarios describiendo el código, así como del uso de variables y funciones que formen el llamado “código autodocumentado”, pues esto nos ayuda tanto para estructurar nuestro código como también para corregir errores y modificar el código en general.

Hernández Molina Leonardo Gael:

Realizar esta práctica sin lugar a duda reforzó todos mis conocimientos previamente adquiridos en la materia, ya que además de comprender perfectamente el funcionamiento de cada algoritmo propuesto, pude analizar y

entender el por qué a un algoritmo le lleva más o menos tiempo ordenar cierta cantidad de números a pesar de que son exactamente los mismos números y la misma cantidad. Además aprendí a compilar desde consola, así como ejecutar códigos desde la misma, cosa la cual a principio no me parecía algo muy útil ya que al haber estado usando la ejecución que cada IDE ofrece respectivamente pues llegaba a parecer algo innecesario ejecutar desde consola, nada más lejos de la realidad ya que ejecutar en consola además de que es la manera más "Correcta" por decirlo así, podemos contar con otras opciones las cuales por el momento fue pasarle parámetros con comandos, cosa que podemos hacer desde el mismo compilador de la IDE pero utilizando comandos como FOPEN, haciendo que nuestro código tenga unas cuantas líneas de más o incluso

pudiendo generarnos alguna clase de conflicto ya que me encontré con el problema de que mi compilador no tenía la librería para poder utilizar estos comandos, así que utilizar CMD fue una herramienta mucho más fácil y sobre todo útil.

Cómo mencionaba anteriormente, logré entender el por qué a un algoritmo le tarda más o menos tiempo ejecutar un programa, y todo esto depende de la complejidad de nuestro algoritmo, es decir, que tan complejo o que tan "difícil" le resulta a la computadora ejecutar cierta cantidad de instrucciones, porque en el caso de burbuja, al tener dos ciclos for, lograba hacer que la cantidad de instrucciones que leyera fuera de n cuadrado, cosa que al tener 5 números pues no era muy relevante ya que cómo sabemos, la computadora es capaz de procesar 100000000 de instrucciones por segundo, lo cual comparado con 5 al cuadrado = 25 pues no es nada, pero ¿qué pasaría si en vez de 5 operaciones son 10000000? Entonces sería 10000000 al cuadrado, cosa que al calcularlo nos damos cuenta de que no es un tiempo precisamente razonable, en comparación a otros algoritmos que al hacer uso de la recursividad o pivotes pueden lograr casos cómo O(N log N) los cuales son la mejor complejidad que podemos llegar a encontrar al menos en un algoritmo de ordenamiento.

Velázquez Díaz Luis Francisco:

Para concluir, puedo decir que esta práctica me ayudó mucho para reforzar los conocimientos adquiridos en clase, a su vez, gracias a esta nos pudimos dar cuenta que es muy esencial conocer la implementación de estos algoritmos ya que, con ellos, la manipulación de la información se vuelve más fácil y práctica.

A mí parecer de todos los algoritmos implementados, los más útiles son Quick Sort y Merge Sort, aunque estos son más difíciles de entender y de implementar, la rapidez con la que organizan la información es mucho mayor a los de los demás algoritmos.

Haciendo más pruebas que las solicitadas, me pude dar cuenta que varios de los algoritmos ya tardaban bastante en reflejar un resultado, lo cual hacía muy tardado el proceso, sin embargo, había otros que lo realizaban en menos de un segundo, lo que me pareció algo impresionante y a destacar.

También, pude encontrarme con varios problemas a la hora de realizar la práctica, uno de ellos es que yo jamás había ejecutado en cmd y aunque es algo muy fácil, comenzar a hacerlo por primera vez considero que me genero un retraso importante a lo largo del trabajo.

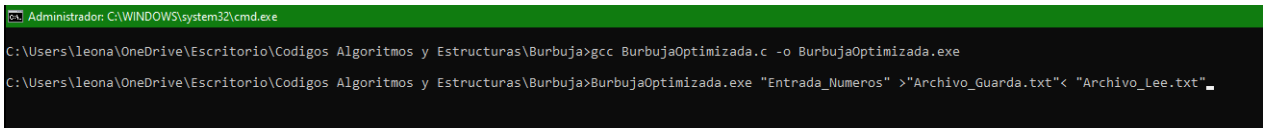
Considero que los resultados obtenidos son muy buenos y nos ayudó a cumplir todos los objetivos esperados.

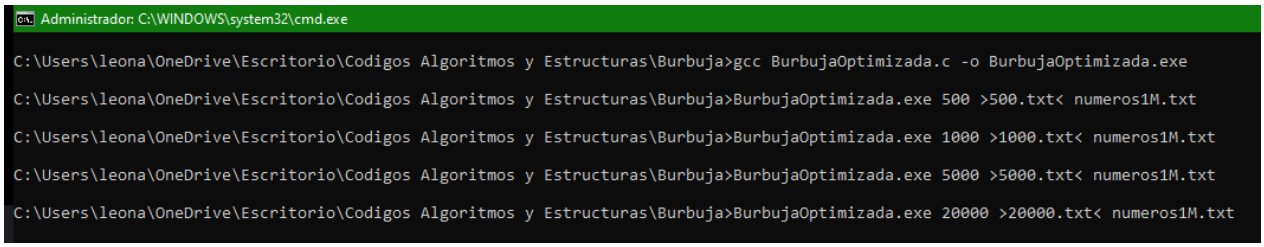
Burbuja optimizada

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
/*
                INTEGRANTES
    /*********************************************
        Algoritmo: Burbuja optimizada
            Integrantes:
        Equipo: Los tios pelones
        Grupo: 2CM1
        Buendía Velazco Abel
        Cadena Becerra Erick Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
            Grupo: 2CM1
    *********************************************/
    Compilación: gcc BurbujaOptimizada.c -o "Nombre_Ejecutable"
    Ejecución: "Nombre_Ejecutable.exe" (Entrada de numeros) > "Archivo_receptor.txt" < "Nombre_Archivo_a_Ordenar.txt"
*/


//Definición de valores bool
#define TRUE 1
#define FALSE 0

//Prototipo de la función
void BurbujaOptimizada(int *A,int n);

int main(int argc, char* argv[])
{
    //Variables para la medición de tiempos
    clock_t t_inicio, t_final;
    double t_intervalo;
    //Variables para el algoritmo
    int i,j,n,*A;
    //Si no recibe al menos dos argumentos en cmd, salimos del programa
    if (argc!=2)
    {
        printf("\nIndique el tamanio de n - Ejemplo: [user@equipo]$ %s 100\n",argv[0]);
        exit(1);
    }

    //Convierte la cadena de caracteres a numeros enteros
    n=atoi(argv[1]); //n
    //Reserva espacio en memoria
    A=malloc(n*sizeof(int));
    if(A==NULL)
    {
        printf("\nError al intentar resevar memoria para %d elementos\n",n);
        exit(1);
    }

    //Leer de la entrada estandar los n valores y colocarlos en el arreglo de numeros
    for(i=0;i<n;i++)
        scanf("%d",&A[i]);
    //Inicia medición del tiempo
    t_inicio = clock();
    //Llamar al algoritmo
    BurbujaOptimizada(A,n);
    //Termina medición del tiempo
    t_final = clock();
    //Cálculo del tiempo y enviar mensaje a salida estandar con la medición
    t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
    //Enviar a la salida estandar el arreglo final
    printf("\n\nOrdenamiento por burbuja optimizada:");
    for(i=0;i<n;i++)
        printf("\nA[%d]=%d",i,A[i]);
    //Imprime el resultado del tiempo medido con respecto la ejecución del algoritmo
    printf("\n\nTiempo medido: %.15f segundos.", t_intervalo);
    return 0;
}
/*
void BurbujaOptimizada(int *A,int n)
Recibe: int * Referencia/Dirección al arreglo A, int tamaño del arreglo
Devuelve: void (No retorna valor explicito)
Observaciones: Proyecto que realiza el ordenamiento de un arreglo en orden de complejidad  O(n^2)
*/

//Algoritmo de ordenamiento "Burbuja optimizada"
```

```c
//Algoritmo de ordenamiento "Burbuja optimizada"
void BurbujaOptimizada(int *A,int n)
{
    int i=0,aux, cambios = 1;
    int j,k;

    while(i<=(n-1) && cambios==1){
            cambios = 0;
            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;
                cambios = 1;
                }
            }
            i++;
    }

    //Voltear el arreglo O(n), ya que el pseudocodigo nos da el ordenamiento invertido
    for(i=0,j=n-1;i<n/2;i++,j--)
    {
        aux=A[i];
        A[i]=A[j];
        A[j]=aux;
    }


    return;
}
```

Script de ejecución y compilación





Ordenamiento por inserción

Código

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
/*
                INTEGRANTES
    /*********************************************
        Algoritmo: Burbuja optimizada
            Integrantes:
        Equipo: Los tíos melones
        Grupo: 2CM1
        Bucndía Velazco Abel
        Cantú Becerra Erick Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
            Grupo: 2CM1
    *********************************************/
    Compilación: gcc BurbujaOptimizada.c -o "Nombre_Ejecutable"
    Ejecución: "Nombre_Ejecutable.exe" (Entrada de numeros) > "Archivo_receptor.txt" < "Nombre_Archivo_a_Ordenar.txt"
*/
//Definición de valores bool
#define TRUE 1
#define FALSE 0
//Prototipo de la función
void Inserccion(int *A,int n);

int main(int argc, char* argv[])
{
    //Variables para la medición de tiempos
    clock_t t_inicio, t_final;
    double t_intervalo;
    //Variables para el algoritmo
    int i,j,n,*A;
    //Si no recibe al menos dos argumentos en cmd, salimos del programa
    if (argc!=2)
    {
```

```
        printf("\nIndique el tamanio de n - Ejemplo: [user@equipo]$ %s 100\n",argv[0]);
        exit(1);
    }
    //Convierte la cadena de caracteres a numeros enteros
    n=atoi(argv[1]); //n
    //Reserva espacio en memoria
    A=malloc(n*sizeof(int));
    if(A==NULL)
    {
        printf("\nError al intentar reservar memoria para %d elementos\n",n);
        exit(1);
    }
    //Leer de la entrada estandar los n valores y colocarlos en el arreglo de números
    for(i=0;i<n;i++)
        scanf("%d",&A[i]);
    //Inicia medición del tiempo
    t_inicio = clock();
    //Llamar al algoritmo
    Inserccion(A,n);
    //Termina medición del tiempo
    t_final = clock();
    //Cálculo del tiempo y enviar mensaje a salida estandar con la medición
    t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
    //Enviar a la salida estandar el arreglo final
    printf("\n\nOrdenamiento por insercion:");
    for(i=0;i<n;i++)
        printf("\nA[%d]=%d",i,A[i]);
    //Mostrar el tiempo medido
    printf("\n\nTiempo medido: %.15f segundos.", t_intervalo);
    return 0;
}

void Inserccion(int *A,int n)
Recibe: int * Referencia/Dirección al arreglo A, int tamaño del arreglo
Devuelve: void (No retorna valor explicito) dado que solo cambia de posicion los valores
Observaciones: Función que revierte el orden de los valores de A (voltea el arreglo)
y posteriormente pierde tiempo en razón de O(n²)
*/
void Inserccion(int *A,int n)
{
    int i, j, temp; //Variables para el programa (iteradores y auxiliares)

    for (i = 0; i <= n - 1; i++)
    {
        j = i;
        temp = A[i];
        //Ejecion de recorrido de numeros en caso de que el termino anterior sea mayor
        while ((j > 0) && (temp < A[j - 1]))
        {
            A[j] = A[j - 1];
            j--;
        }
        //Asignacion del lugar que ocupará el número
        A[j] = temp;
    }

}
```

Script de ejecución y compilación

```
Seleccionar Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>gcc Insercion.c -o Insercion.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>Insercion.exe "Entrada_Numeros.txt" > "Archivo_Guarda.txt" <"Archivo_Lectura.txt"
```

```
Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>gcc Insercion.c -o Insercion.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>Insercion.exe 500 >500.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>Insercion.exe 1000 >1000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>Insercion.exe 5000 >5000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Insercion>Insercion.exe 20000 >20000.txt< numeros1M.txt
```

Algoritmo por Selección

Código

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include <conio.h>
/*
                INTEGRANTES
    /*****************************************
        Algoritmo: Burbuja optimizada
            Integrantes:
        Equipo: Los tios pelones
        Grupo: 2CM1
        Buendia Velazco Abel
        Carmio Becerra Erick Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
            Grupo: 2CM1
    *****************************************
    Compilación: gcc BurbujaOptimizada.c -o "Nombre_Ejecutable"
    Ejecución: "Nombre_Ejecutable.exe" (Entrada de numeros) > "Archivo_receptor.txt" < "Nombre_Archivo_a_Ordenar.txt"
*/
//Definición de valores bool
#define TRUE 1
#define FALSE 0
//Prototipo de la función
void Seleccion (int *A, int n);

int main(int argc, char* argv[])
{
    //Variables para la medición de tiempos
    clock_t t_inicio, t_final;
    double t_intervalo;
    //Variables para el algoritmo
    int i,j,n,*A;
    //Si no recibe al menos dos argumentos en cmd, salimos del programa
    if (argc!=2)


        printf("\nIndique el tamanio de n - Ejemplo: [user@equipo]$ %s 100\n",argv[0]);
        exit(1);
    }
    //Convierte la cadena de caracteres a numeros enteros
    n=atoi(argv[1]); //n
    //Reserva espacio en memoria para n enteros
    A=malloc(n*sizeof(int));
    if(A==NULL)
    {
        printf("\nError al intentar reservar memoria para %d elementos\n",n);
        exit(1);
    }

    //Leer de la entrada estandar los n valores y colocarlos en el arreglo de números
    for(i=0;i<n;i++)
        scanf("%d",&A[i]);
    //Inicia medición del tiempo
    t_inicio = clock();
    //Llamar al algoritmo
    Seleccion(A,n);
    //Termina medición del tiempo
    t_final = clock();
    //Cálculo del tiempo y enviar mensaje a salida estandar con la medición
    t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
    //Enviar a la salida estandar el arreglo final
    printf("\n\nOrdenamiento por Seleccion:");
    for(i=0;i<n;i++)
        printf("\nA[%d]=%d",i,A[i]);
    //Mostrar el tiempo medido
    printf("\n\nTiempo medido: %.15f segundos.", t_intervalo);
    return 0;
}

void Seleccion (int *A, int n)
Recibe: int * Referencia/Dirección al arreglo A, int tamaño del arreglo
Devuelve: void (No retorna valor explicito)
Observaciones: Función Ordena n enteros en razón de O(n^2)
*/

void Seleccion (int *A, int n){
    int k, i, p, aux; //Variables para el algoritmo (Iteradores, auxiliares, etc.)
    for (k = 0; k < n - 1; k++)
    {
        p = k; //variable que guarda k para hacer la comparacion con el valor no acomodado mas a la izquierda en nuestro arreglo
        //recorre nuestro arreglo de acuerdo a k, el cual nos marca desde donde empezara a recorrerlo este ciclo
        for (i = k + 1; i < n; i++)
        {
            //Comara si el ultimo valor no cambiado a la izquierda de nuestro arreglo es mayor al de los siguientes para encontrar el valor minimo
            if (A[i] < A[p])
                p = i; //guarda el valor minimo de la lista
        }
        //Se cambian de posicion con ayuda de una variable auxiliar para acomodar el valor minimo encontrado en el primero mas a la izq que no haya sido ac
        aux = A[p];
        A[p] = A[k];
        A[k] = aux;
}
}
```

Script de ejecución y compilación

```
Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>gcc Seleccion.c -o Seleccion.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>Seleccion.exe "Entrada_Numeros.txt" > "Archivo_Guarda.txt " < "Archivo_Lectura.txt"
```

```
Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>gcc Seleccion.c -o Seleccion.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>Seleccion.exe 500 >500.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>Seleccion.exe 1000 >1000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>Seleccion.exe 5000 >5000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>Seleccion.exe 20000 >20000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Seleccion>
```

Algoritmo Shell

Código

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
/*
                INTEGRANTES
    ***********************************************
        Algoritmo: Burbuja optimizada
            Integrantes:
        Equipo: Los tios pelones
        Grupo: 2CM1
        Buendia Velazco Abel
        Carmio Rezerra Erick Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
            Grupo: 2CM1
    ***********************************************
    Compilación: gcc BurbujaOptimizada.c -o "Nombre_Ejecutable"
    Ejecución: "Nombre_Ejecutable.exe" (Entrada de numeros) > "Archivo_receptor.txt" < "Nombre_Archivo_a_Ordenar.txt"
*/
//Definición de valores bool
#define TRUE 1
#define FALSE 0
//Prototipo de la función
void Shell(int *A,int n);
int main(int argc, char* argv[])
{
    //Variables para la medición de tiempos
    clock_t t_inicio, t_final;
    double t_intervalo;
    //Variables para el algoritmo
    int i,j,n,*A;
    //Si no recibe al menos dos argumentos en cmd, salimos del programa
    if (argc!=2)
    {
        exit(1);
    }
    //Convierte la cadena de caracteres a numeros enteros
    n=atoi(argv[1]); //n
    //Reserva espacio en memoria para n números  enteros
    A=malloc(n*sizeof(int));
    if(A==NULL)
    {
        printf("\nError al intentar reservar memoria para %d elementos\n",n);
        exit(1);
    }

    //Leer de la entrada estandar los n valores y colocarlos en el arreglo de números
    for(i=0;i<n;i++)
        scanf("%d",&A[i]);
    //Inicia medición del tiempo
    t_inicio = clock();
    //Llamar al algoritmo
    Shell(A,n);
    //Termina medición del tiempo
    t_final = clock();
    //Calculo del tiempo y enviar mensaje a salida estandar con la medición
    t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
    //Enviar a la salida estandar el arreglo final
    printf("\n\nOrdenamiento por Shell:");
    for(i=0;i<n;i++)
        printf("\nA[%d]=%d",i,A[i]);
    //Mostrar el tiempo medido
    printf("\n\nTiempo medido: %.15f segundos.", t_intervalo);
    return 0;
}
/*
void Shell(int *A,int n)
```

```c
void Shell(int *A,int n)
Recibe: int * Referencia/Dirección al arreglo A, int tamaño del arreglo
Devuelve: void (No retorna valor explicito)
Observaciones: Función que ordena una serie de enteros con el algoritmo shell en razón de O(n^2)
*/
void Shell(int *A,int n){
int  i, k, b, temp; //Variables para el algoritmo(Iteradores, auxiliares)
    k = n/2; //Guarda el resultado de dividir el arreglo a la mitad
        while(k>=1){
            b=1;
            while(b!=0){
                b=0;
                for(i=k;i<=n-1;i++){
                    if(A[i-k]>A[i]){ //Intercambia los valores para poder ordenar el arreglo
                        temp=A[i];
                        A[i]= A[i-k];
                        A[i-k]=temp;
                        b=b+1;
                    }
                }
            }
            k=k/2; //Vuelve a dividir a la mitad el arreglo
    }

    return;
}
```

Script de ejecución y compilación

```
Administrador: C:\WINDOWS\system32\cmd.exe
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>gcc Shell.c -o Shell.exe
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>Shell.exe "Entrada_Numeros.txt" > "500.txt" < numeros1M.txt
```

```
Administrador: C:\WINDOWS\system32\cmd.exe
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>gcc Shell.c -o Shell.exe
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>Shell.exe 500 >500.txt< numeros1M.txt
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>Shell.exe 1000 >1000.txt< numeros1M.txt
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>Shell.exe 5000 >5000.txt< numeros1M.txt
C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Shell>Shell.exe 20000 >20000.txt< numeros1M.txt
```

Algoritmo de ordenamiento por Mezcla o Merge

Código

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include <string.h>
#include <math.h>
/*
                INTEGRANTES
    /*****************************************
        Algoritmo: Burbuja optimizada
            Integrantes:
        Equipo: Los tios melones
        Grupo: 2CM1
        Bucadia Velazco Abel
        Carrio Becerra Erick Gustavo
        Hernandez Molina Leonardo Gadil
        Velazquez Diaz Luis Francisco
            Grupo: 2CM1
    *****************************************
    Compilación: gcc BurbujaOptimizada.c -o "Nombre_Ejecutable"
    Ejecución: "Nombre_Ejecutable.exe" (Entrada de numeros) > "Archivo_receptor.txt" < "Nombre_Archivo_a_Ordenar.txt"
*/
//Definición de valores bool
#define TRUE 1
#define FALSE 0
//Prototipo de la función
void Merge(int *A, int p, int q, int r);
void MergeSort(int *A, int p, int r);

int main(int argc, char* argv[])
{
    //Variables para la medición de tiempos
    clock_t t_inicio, t_final;
    double t_intervalo;
    //Variables para el algoritmo
    int i,j,n,*A;
```

```
//Si no recibe al menos dos argumentos en cmd, salimos del programa
if (argc!=2)
{
    printf("\nIndique el tamanio de n - Ejemplo: [user@equipo]$ %s 100\n",argv[0]);
    exit(1);
}
//Convierte la cadena de caracteres a n numeros enteros
n=atoi(argv[1]); //n
//Apartar memoria para n números  enteros
A=malloc(n*sizeof(int));
if(A==NULL)
{
    printf("\nError al intentar reservar memoria para %d elementos\n",n);
    exit(1);
}
//Leer de la entrada estandar los n valores y colocarlos en el arreglo de números
for(i=0;i<n;i++)
    scanf("%d",&A[i]);
//Inicia medición del tiempo
t_inicio = clock();
//Llamar al algoritmo
MergeSort(A, 0, n-1);
//Termina medición del tiempo
t_final = clock();
//Calcular del tiempo y enviar mensaje a salida estandar con la medición
t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
//Enviar a la salida estandar el arreglo final
printf("\n\nOrdenamiento por Merge:");
for(i=0;i<n;i++)
    printf("\nA[%d]=%d",i,A[i]);
//Mostrar el tiempo medido
printf("\n\nTiempo medido: %.15f segundos.", t_intervalo);
return 0;


void Merge(int *A, int p, int q, int r)
Recibe: int * Referencia/Dirección al arreglo A, int tamaño del arreglo
Devuelve: void (No retorna valor explícito)
Observaciones: Algoritmo que recibe un arreglo para ordenarlo asegurando en todos los casos una razón de O(nlogn)
*/
void Merge(int *A, int p, int q, int r){ //A= Arreglo desordenado
    int *C; //Arreglo auxiliar           //P= Parte izquierda
                                         //Q= Tamaño del arreglo
    //Variables para los ciclos          //R= Parte derecha
    int l, i, j;
    l = r - p + 1;//Inicio del arreglo
    i = p; //Parte izquierda
    j = q + 1; //Parte derecha
    C = malloc(sizeof(int) * (l + 2)); //Espacio en memoria para guarda el arreglo que  vaya ordenando

    for (int 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++;
           }
       }
    }
    //Terminando de ordenar los numeros, se pasan los numeros del arreglo temporal al arreglo donde se van almacenar todos los numeros a ordenar
    int o = 0;
    for (int m = p; m <= r; m++)
    {
        A[m] = C[o];
        o++;
    }

}
void MergeSort(int *A, int p, int r){
    int q;
    if (p < r)
    {
        q = trunc((p + r) / 2); //Se obtiene la mitad (parte entera) del arreglo
        //Recursividad hasta llegar a 2 elementos
        MergeSort(A, p, q);     //Se ordenan de 0 a la mitad del arreglo
        MergeSort(A, q + 1, r); //Se ordenan de la mitad al final del arreglo
        Merge(A, p, q, r);      //Junta los mini arreglos que se van arreglando
    }
}
```

Script de ejecución y compilación

```
Seleccionar Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>gcc Merge.c -o Merge.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>Merge.exe "Entradas_Numeros" >"Archivo_Guarda.txt"<"Archivo_Lectura.txt"
```

```
Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>gcc Merge.c -o Merge.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>Merge.exe 500 >500.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>Merge.exe 1000 >1000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>Merge.exe 5000 >5000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Merge>Merge.exe 20000 >20000.txt< numeros1M.txt
```

Algoritmo de ordenamiento Rápido o Quick Sort

Código

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include <string.h>
/*
                INTEGRANTES
    /*********************************************
        Algoritmo: Burbuja optimizada
            Integrantes:
        Equipo: Los tios pelones
        Grupo: 2CM1
        Rueda Velazco Abel
        Carbo Becerra Erick Gustavo
        Hernandez Molina Leonardo Gaell
        Velazquez Diaz Luis Francisco
            Grupo: 2CM1
    *********************************************
    Compilación: gcc BurbujaOptimizada.c -o "Nombre_Ejecutable"
    Ejecución: "Nombre_Ejecutable.exe" (Entrada de numeros) > "Archivo_receptor.txt" < "Nombre_Archivo_a_Ordenar.txt"
*/
//Definición de valores bool
#define TRUE 1
#define FALSE 0
//Encabezado de la función
void intercambiar(int *A, int i, int j);
int Pivot(int *A, int p, int r);
void QuickSort(int *A,int p, int r);
int main(int argc, char* argv[])
{
    //Variables para la medición de tiempos
    clock_t t_inicio, t_final;
    double t_intervalo;
    //Variables para el algoritmo
    int i,j,n,*A;
 //Si no recibe al menos dos argumentos en cmd, salimos del programa
 if (argc!=2)
 {
     printf("\nIndique el tamanio de n - Ejemplo: [user@equipo]$ %s 100\n",argv[0]);
     exit(1);
 }
 //Convierte la cadena de caracteres a n numeros enteros
 n=atoi(argv[1]); //n
 //Reserva espacio en memoria para n números  enteros
 A=malloc(n*sizeof(int));
 if(A==NULL)
 {
     printf("\nError al intentar reservar memoria para %d elementos\n",n);
     exit(1);
 }
 //Leer de la entrada estandar los n valores y colocarlos en el arreglo de números
 for(i=0;i<n;i++)
     scanf("%d",&A[i]);
 //Inicia medición del tiempo
 t_inicio = clock();
 //Llamar al algoritmo
 QuickSort(A, 0, n-1);
 //Termina medición del tiempo
 t_final = clock();
 //Cálculo del tiempo y enviar mensaje a salida estandar con la medición
 t_intervalo = (double)(t_final - t_inicio) / CLOCKS_PER_SEC;
 //Enviar a la salida estandar el arreglo final
 printf("\n\nOrdenamiento por Quick:");
 for(i=0;i<n;i++)
     printf("\nA[%d]=%d",i,A[i]);
 //Mostrar el tiempo medido
 printf("\n\nTiempo medido: %.15f segundos.", t_intervalo);
 return 0;
```

```
/*
oid intercambiar(int *A, int i, int j){
Recibe: int * Referencia/Dirección al arreglo A, Mitad izquierda y mitad derecha
Devuelve: void (No retorna valor explicito)
Observaciones: Funcion que ordena valores dados de un arreglo en razón de nLogn
*/
//Función que intercambia los valores de una posición a otra
void intercambiar(int *A, int i, int j){
    int temp;
    temp = A[j];
    A[j] = A[i];
    A[i] = temp;
}
//Función que sirve de pivote para poder arreglar el arreglo
int Pivot(int *A, int p, int r){
    int i = p + 1; //Segundo elemento del arreglo
    int j = r; //Tam del arreglo
    int piv = A[p]; //El pivote es la primer posicion del arreglo
    while (1)
    {
        while (A[i] < piv && i < r)
            i++;
        while (A[j] > piv)
            j--;
        if (i < j)
            intercambiar(A, i, j);
        else
        {
            intercambiar(A, p, j);
            return j;
        }
    }

//Algoritmo QuickSort que ordena ambas mitades haciendo uso del pivote
void QuickSort(int *A,int p, int r){
    if (p < r)
    {
        int j = Pivot(A, p, r); //Se obtiene el pivote
        QuickSort(A, p, j - 1); //Se ordena de 0 al (pivote-1)
        QuickSort(A, j + 1, r); //Se ordena del pivote +1 hasta el final
    }
}
```

Script de ejecución y compilación

```
Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>gcc Quick.c -o Quick.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>Quick.exe "Entradas_Numero" >"Archivos_Guarda.txt"< "Archivos_Lectura.txt"
```

```
Administrador: C:\WINDOWS\system32\cmd.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>gcc Quick.c -o Quick.exe

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>Quick.exe 500 >500.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>Quick.exe 1000 >1000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>Quick.exe 5000 >5000.txt< numeros1M.txt

C:\Users\leona\OneDrive\Escritorio\Codigos Algoritmos y Estructuras\Quick>Quick.exe 20000 >20000.txt< numeros1M.txt
```

Bibliografia

Navarro, A. (2020) Algoritmos de ordenamiento, Juncotic. Available at: https://juncotic.com/algoritmos-de-ordenamiento/ (Accessed: November 5, 2022).

Navarro, A. (2016) Ordenamiento por inserción – Algoritmos de ordenamiento, Juncotic. Available at: https://juncotic.com/ordenamiento-por-insercion-algoritmos-de-ordenamiento (Accessed: November 5, 2022).

"Merge Sort Algorithm - GeeksforGeeks". GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/ (accedido el 10 de noviembre de 2022).

"Bubble Sort Algorithm - GeeksforGeeks". GeeksforGeeks. https://www.geeksforgeeks.org/bubble-sort/ (accedido el 10 de noviembre de 2022).