



## Centro Universitario de Ciencias Exactas e Ingenierías (CUCEI)

### Actividad de aprendizaje 8: Métodos de ordenamiento recursivos

Víctor Agustín Díaz Méndez  
Ingeniería en Informática

Estructura de Datos I (Sección D12)  
Profesor: Dr. Gutierrez Hernandez Alfredo



## Problema

Problema: Haga un programa que genere valores enteros aleatorios entre 0 y 65,535 con los cuales rellene un arreglo de 65,536 elementos y luego los ordene tomando en cuenta el tiempo requerido para ello e informe una vez terminado el proceso de ordenación. El programa informará el tiempo necesario para ordenar el arreglo con los métodos de ordenamiento: Burbuja (mejorada), Shell, Inserción, Selección, Mezcla, y QuickSort; todos los tiempos en una sola pantalla para fines comparativos.

## Requerimientos:

- a) El estilo de programación debe ser Orientado a Objetos.
- b) El conjunto de elementos debe ser idéntico para cada caso de ordenamiento.
- c) Todos los métodos de ordenamiento deben ser métodos de la misma clase.

## Entregables:

Un solo documento PDF que contenga:

1. Caratula (Nombre de la actividad y datos del alumno).
2. Resumen personal del trabajo realizado, y forma en que fue abordado el problema.
3. Código fuente.
4. Impresiones de pantalla que muestren la ejecución satisfactoria del programa.

## Resumen:

La mayoría de los métodos de ordenamiento ya los había realizado anteriormente, lo que facilitó mucho las cosas. Así que solo tuve que adaptarlos para obtener el tiempo que tarda en ordenar cada método su arreglo. Esto fue sencillo, solo hice un método para cada método de ordenamiento que lo llamaba y media el tiempo que se tardaba en procesar el arreglo. El tiempo es retornado por la función.

Seguí mis pruebas los métodos de ordenamiento más rápidos son los recursivos. Obteniendo como resultado en mis pruebas que *Merge Sort* es ligeramente más rápido que *Quick Sort*. Los resultados pueden variar según las características de cada modelo de computadora como el procesador y la RAM.

Para facilitarme un poco las cosas hice un método que copia los valores de un arreglo a otro, para que así todos los arreglos a ordenar fueran iguales. También hice otro método para mostrarlos.



De todos los métodos de ordenamiento *inserción* y *burbuja* fueron los menos eficientes. *Selección* también demostró ser poco eficiente, aunque se tardó casi una tercera parte del tiempo que *burbuja* e *inserción*.



## Ejecución satisfactoria

```
Debug: ./Sort
vic@vic-Lenovo-B490:~/OneDrive/Escuela/2 semestre/EDA/EDA/p8/Sort/bin/Debug$ ./Sort
Burbuja
Insertando datos...
Ordenando...
Tiempo 29931 milisegundos

Insert
Insertando datos...
Ordenando...
Tiempo 29867 milisegundos

Quick Sort
Insertando datos...
Ordenando...
Tiempo 18 milisegundos

Merge Sort
Insertando datos...
Ordenando...
Tiempo 9 milisegundos

Selección
Insertando datos...
Ordenando...
Tiempo 10452 milisegundos

Shell
Insertando datos...
Ordenando...
Tiempo 25 milisegundos

vic@vic-Lenovo-B490:~/OneDrive/Escuela/2 semestre/EDA/EDA/p8/Sort/bin/Debug$
```



## Código fuente

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>

#include "sort.h"

#define A_SIZE 65536

using namespace std;

float getElapsedTimeInMs();
void mostrarArreglo(int* arr);
void llenaArreglo(int* arr);
void copiarArreglo(int* orig, int* cop);

int main()
{
    int arrBubble[A_SIZE],
        arrShell[A_SIZE],
        arrInsert[A_SIZE],
        arrSelect[A_SIZE],
        arrMerge[A_SIZE],
        arrQuick[A_SIZE];
    int startTime, elapsedTime;

    Sort mySort;

    srand(time(NULL));

    llenaArreglo(arrBubble);
    copiarArreglo(arrBubble, arrInsert);
    copiarArreglo(arrBubble, arrQuick);
    copiarArreglo(arrBubble, arrSelect);
    copiarArreglo(arrBubble, arrShell);

    cout << "\tBurbuja" << endl
         << " Insertando datos..." << endl
         << " Ordenando..." << endl;
    elapsedTime = mySort.bubbleTime(arrBubble, A_SIZE);
    cout << " Tiempo " << elapsedTime << " milisegundos" << endl << endl;
```



```
cout << "\tInsert" << endl
    << " Insertando datos..." << endl
    << " Ordenando..." << endl;
elapsedTime = mySort.bubbleTime(arrInsert, A_SIZE);
cout << " Tiempo " << elapsedTime << " milisegundos" << endl << endl;

cout << "\tQuick Sort" << endl
    << " Insertando datos..." << endl
    << " Ordenando..." << endl;
elapsedTime = mySort.quickTime(arrQuick, 0, (A_SIZE-1));
cout << " Tiempo " << elapsedTime << " milisegundos" << endl << endl;

cout << "\tMerge Sort" << endl
    << " Insertando datos..." << endl
    << " Ordenando..." << endl;
elapsedTime = mySort.mergeTime(arrMerge, 0, (A_SIZE-1), A_SIZE);
cout << " Tiempo " << elapsedTime << " milisegundos" << endl << endl;

cout << "\tSelección" << endl
    << " Insertando datos..." << endl
    << " Ordenando..." << endl;
elapsedTime = mySort.selectTime(arrSelect, A_SIZE);
cout << " Tiempo " << elapsedTime << " milisegundos" << endl << endl;

cout << "\tShell" << endl
    << " Insertando datos..." << endl
    << " Ordenando..." << endl;
elapsedTime = mySort.shellTime(arrShell, A_SIZE);
cout << " Tiempo " << elapsedTime << " milisegundos" << endl << endl;

return 0;
}

void mostrarArreglo(int* arr){
    for (int i = 0; i < A_SIZE; i++) {
        cout << arr[i] << " ";
    }
}

void llenaArreglo(int* arr) {
    for (int i = 0; i < A_SIZE; i++) {
        arr[i] = rand() % A_SIZE;
    }
}
```



```
}
```

```
void copiarArreglo(int* orig, int* cop) {  
    for (int i = 0; i < A_SIZE; i++) {  
        cop[i] = orig[i];  
    }  
}
```



sort.h

```
#ifndef SORT_H
#define SORT_H

class Sort {
public:
    float bubbleTime(int*, int);
    float insertTime(int*, int);
    float quickTime(int*, int, int);
    float mergeTime(int*, int, int, int);
    float selectTime(int*, int);
    float shellTime(int*, int);

private:
    void bubble(int*, const int&);
    void insertSort(int*, const int&);
    void quick(int*, int, int);
    void mergeSort(int*, int, int, int);
    void mergeS(int*, int, int, int, int);
    void selectSort(int*, const int&);
    void shell(int*, const int&);

    void exchange(int& a, int& b);
    float getElapsedTimeInMs();

};

#endif // SORT_H
```





## sort.cpp

```
##include "sort.h"

#include <ctime>

float Sort::bubbleTime(int* arr, int s) {
    float startTime;
    startTime = getElapsedTimeInMs();
    bubble(arr, s);
    return getElapsedTimeInMs() - startTime;
}

float Sort::insertTime(int* arr, int s) {
    float startTime;
    startTime = getElapsedTimeInMs();
    insertSort(arr, s);
    return getElapsedTimeInMs() - startTime;
}

float Sort::quickTime(int* arr, int extI, int extD) {
    float startTime;
    startTime = getElapsedTimeInMs(); //Hora comienzo
    quick(arr, extI, extD);
    return getElapsedTimeInMs() - startTime; //Hora fin
}

float Sort::mergeTime(int* arr, int extI, int extD, int s) {
    float startTime = getElapsedTimeInMs(); //Hora comienzo
    mergeSort(arr, extI, extD, s);
    return getElapsedTimeInMs() - startTime; //Hora fin
}

float Sort::selectTime(int* arr, int s) {
    float startTime = getElapsedTimeInMs();
    selectSort(arr, s);
    return getElapsedTimeInMs() - startTime;
}

float Sort::shellTime(int* arr, int s) {
    float startTime = getElapsedTimeInMs();
    shell(arr, s);
    return getElapsedTimeInMs() - startTime;
}
```



```
void Sort::bubble(int* arr, const int& s) {
    int i = (s-1),
        j;
    bool seguir;
    do {
        seguir = false;
        j = 0;
        while ( j < i) {
            if (arr[j] > arr[j + 1]) {
                exchange(arr[j], arr[j + 1]);
                seguir = true;
            }

            j++;
        }

        i--;
    } while (seguir);
}
```

```
void Sort::insertSort(int* arr, const int& s) {
    int i = 1,
        j,
        last = s - 1;
    int aux;

    while (i <= last) {
        aux = arr[i];

        j = i;

        while (j > 0 and aux < arr[j-1]) {
            arr[j] = arr [j-1];
            j--;
        }
        if(i != j) {
            arr[j] = aux;
        }
        i++;
    }
}
```



```
}
```

```
void Sort::quick(int* arr, int extl, int extD) {  
    int i, j;  
  
    if (extl >= extD) {  
        return;  
    }  
  
    exchange(arr[(extl + extD) / 2], arr[extD]);  
  
    i = extl;  
    j = extD;  
  
    while (i < j) {  
        while (i < j and arr[i] <= arr[extD]) {  
            i++;  
        }  
  
        while (i < j and arr[j] >= arr[extD]) {  
            j--;  
        }  
  
        if (i != j) {  
            exchange(arr[i], arr[j]);  
        }  
    }  
  
    if (i != extD) {  
        exchange(arr[i], arr[extD]);  
    }  
  
    quick(arr, extl, (i - 1));  
    quick(arr, (i + 1), extD);  
}
```

```
void Sort::mergeSort(int* arr, int extl, int extD, int s) {  
    if (extl < extD) {  
        int m = (extl + extD) / 2;  
        mergeSort(arr, extl, m, s);  
        mergeSort(arr, m+1, extD, s);  
        mergeS(arr, extl, m, extD, s);  
    }
```



```
    }  
}  
  
void Sort::mergeS(int* arr, int extl, int m, int extD, int s) {  
    int i, j, k;  
    int temp[s];  
  
    for (i = extl; i <= extD; i++) {  
        temp[i] = arr[i];  
    }  
  
    i = extl;  
    j = m + 1;  
    k = extl;  
  
    while (i <= m and j <= extD) {  
        if (temp[i] <= temp[j]) {  
            arr[k++] = temp[i++];  
        } else {  
            arr[k++] = temp[j++];  
        }  
    }  
  
    while (i <= m) {  
        arr[k++] = temp[i++];  
    }  
}  
  
void Sort::selectSort(int* arr, const int& s)  
{  
    int last = s - 1;  
    int i = 0,  
        j;  
    int smaller;  
  
    while (i < last) {  
        smaller = i;  
  
        j = i + 1;  
        while (j <= last) {  
            if (arr[j] < arr [smaller]){  
                smaller = j;  
            }  
        }  
    }  
}
```



```
        j++;
    }

    if (smaller != i) {
        exchange(arr[i], arr[smaller]);
    }

    i++;
}
}

void Sort::shell(int* arr, const int& s)
{
    float fact = 3.0/4.0;
    int last = s - 1; //Ultimo indice
    int dif = last * fact;
    int i;

    while (dif > 0) {
        i = 0;

        while (i <= last - dif) {
            if(arr[i] > arr[i + dif]) {
                exchange(arr[i], arr[i + dif]);
            }
            i++;
        }
        dif = dif * fact;
    }
}

void Sort::exchange(int& a, int& b) {
    int c;

    c = a;
    a = b;
    b = c;
}

float Sort::getElapsedTimeInMs() {
    //Return clock() as ms
    //1 Second = 1000 milliseconds
}
```



```
    return clock()/(CLOCKS_PER_SEC/1000);  
}
```