Tarea 1

Nathalie Alfaro Quesada B90221

***Resumen*— En este trabajo se analizaron seis algoritmos de ordenamiento: Inserción (Insertion), Selección (Selection), Mezcla (Mergesort), Montículos (Heapsort), Rápido (Quicksort) y Residuos (Radixsort).00., estos se programaron y durante el ordenamiento se les calculó su duración en milisegundos de ordenar números enteros en un arreglo de tamaño de 50.000, 100.000, 150.000 y 200.000, para luego graficarlos y observar su complejidad computacional. El algoritmo de Inserción recorre el arreglo insertando cada elemento en la posición adecuada del subarreglo ordenado, mientras que el de Selección busca el menor elemento y lo intercambia con el primer elemento no ordenado, el de Mezcla divide el arreglo en subarreglos a la mitad para ordenarlos y luego juntarlos en uno, Montículos monticulariza el árbol para luego ordenarlo quitando el máximo de la raíz e insertarlo al final, el Rápido selecciona un pivote de la lista y la divide en dos empezando a ordenar primero la izquierda y luego la derecha, y por último se tiene el de Residuos que ordena los elementos comparando sus cifras menos significativas con las más significativas y las guarda en una lista. Además, se observa que Inserción y Selección tienen una complejidad computacional de $O(n^2)$, mientras que Mezcla, Montículos y Rápido son ideales para arreglos de gran tamaño porque son $O(n\ log\ n)$, con excepción para el Rápido y el de Residuos presenta un tiempo de $O(n \cdot k)$.**

I. INTRODUCCIÓN

En este trabajo, se implementan seis algoritmos de ordenamiento Inserción (Insertion por su nombre en inglés), Selección (Selection), Mezcla (Mergesort), Montículos (Heapsort), Rápido (Quicksort) y Residuos (Radixsort), con el fin de tomarles el tiempo de duración a cada uno ordenando elementos en un arreglo de tipo Int de tamaño 50.000, 100.000, 150.000 y 200.000, para analizar su complejidad computacional, es importante conocer la mejor opción para optimizar procesos y así ahorrar recursos. El objetivo es implementar su respectivo código y ponerlo a prueba con grandes datos para comprender su funcionalidad y su eficiencia en diferentes casos.

II. METODOLOGÍA

Para lograr lo propuesto se consulta el libro de texto de Thomas H. Cormen y colaboradores, Introduction to Algorithms fourth edition, para comprender mejor los algoritmos y así poder programarlos en el lenguaje de programación C++ en el ambiente de desarrollo Visual Studio Code.

El código se muestra en los apéndices. La funcionalidad implementada fue hacer un arreglo de tipo Int de tamaño *n*, donde este tamaño iba cambiando de valor por 50.000, 100.000, 150.000 y 200.000. Los arreglos fueron rellenados con números aleatorios de tipo Int. Cuando ya se tiene el arreglo lleno, es momento de ordenarlo, entonces empezamos tomando el tiempo, llamamos al método de ordenamiento ya sea Insertion, Selection, Mergesort, Heapsort, Quicksort o Radixsort, pasando por parámetros el arreglo y el tamaño del arreglo, luego se toma el tiempo final para restarlos y ver la duración de ordenarlos, además se verifica que se haya ordenado correctamente, imprimiendo un mensaje de "Correcto" y haciendo la prueba de falla recomendada por el docente.

Cuadro I
TIEMPO DE EJECUCIÓN DE LOS ALGORITMOS.

		Tiempo (ms)					
		Corrida					
Algoritmo	Tam. (k)	1	2	3	4	5	Prom.
Selección	50	3077	3127	3105	3093	3086	3097.6
	100	11957	11988	11957	11867	11977	11949.2
	150	27022	27026	26971	27097	27059	27035
	200	47781	47762	47893	47825	47768	47805.8
Inserción	50	2745	2756	2768	2749	2757	2755
	100	11036	11056	11025	11061	11029	11041.4
	150	24631	24644	24605	24629	24661	24634
	200	**43724**	43830	43736	43058	43064	43482.4
Mezcla	50	7	6	8	10	6	7.4
	100	13	16	18	16	17	16
	150	25	29	26	29	25	26.8
	200	28	38	42	37	35	36
Montículos	50	10	15	9	11	9	10.8
	100	21	21	24	20	25	22.2
	150	34	32	33	37	32	33.6
	200	42	53	51	52	50	49.6
Rápido	50	22	18	19	19	23	20.2
	100	73	72	71	73	72	72.2
	150	144	144	141	153	144	145.2
	200	236	252	242	244	241	243
Residuos	50	11	14	13	11	15	12.8
	100	21	23	23	28	33	25.6
	150	36	43	43	42	41	41
	200	47	53	55	58	53	53.2

En el anterior Cuadro I se muestra que, para cada algoritmo de ordenamiento estudiado, se probó con un arreglo de tipo Int de tamaño 50.000, 100.000, 150.000 y 200.000, donde cada uno se ejecutó 5 veces para así obtener un promedio de los tiempos de duración de cada uno en milisegundos.

III. RESULTADOS

Los tiempos de ejecución de las 5 corridas de los algoritmos se muestran en el cuadro I. Cabe destacar que el hardware utilizado para estos procesos fue el Intel(R) Core(TM) i5-1035G1 de 4 Cores y 8 Logical processors con 12 GB de RAM.

En la figura 1 se muestra el gráfico del tiempo promedio en milisegundos de ordenar miles de datos por el algoritmo de Selección, probando el código por este tipo de ordenamiento se observa que en arreglos de tamaño grande se tarda en dar los resultados y así entre más grande sea. En el gráfico se observa que su curva tiende al peor

de los casos que es $O(n^2)$, haciendo este algoritmo ineficiente para ordenar datos grandes como miles.

Para la figura 2 se tiene el gráfico del tiempo promedio en milisegundos de ordenar miles de datos con el algoritmo de Inserción, cuando se prueba el código también se observa que se tarda un poco en dar el resultado, aunque es un poco más rápido que Selección, se comporta similar ya que su curva tiende al peor caso que es $O(n^2)$, que nos indica que es un algoritmo ineficiente para manejar miles de datos.

En la figura 3 y 4 se presenta el tiempo promedio en milisegundos al ordenar miles de datos por el algoritmo de ordenamiento por Mezcla y por Montículos, los resultados al ejecutar este código tienen una gran diferencia con respecto a los dos anteriores ya que da los resultados casi de inmediato y esto se debe a que su complejidad computacional en el mejor caso y en el peor caso es la misma, $O(n\ log\ n)$, lo que lo hace perfecto para ordenar miles de datos de una forma eficiente.

Aunque en la figura 5 que es el ordenamiento Rápido, también se observa que posee la misma complejidad computacional a Mezcla y Montículos, ya que es su caso promedio, pero cabe destacar que su complejidad computacional puede cambiar ya que su peor caso sería $O(n^2)$ que esto se puede dar cuando el arreglo ya está ordenado, pero esto no sucede en este caso.

También, para el último ordenamiento en la figura 6, por Residuos, se observa que su complejidad computacional se va a $O(n \cdot k)$, donde tenemos a n como el tamaño del arreglo y k el rango de elementos, con una curva más lineal.

En la figura 7 se puede observar la comparación entre todas las curvas de los seis algoritmos de ordenamiento estudiados sobre su tiempo de duración promedio en milisegundos ordenando un arreglo de tipo Int de cuatro tamaños distintos, en el caso de Inserción y Selección se observa que tienen mucha similitud ya que ambos tienden al mismo peor caso $O(n^2)$, mientras que Mezcla, Montículos y Rápido tienen mucha similitud en sus curvas ya que se ven que tienden a $O(n\ log\ n)$, lo cual deja en evidencia que son algoritmos bastante eficientes en el procesamiento de miles de datos. En cambio en Residuos vemos una gráfica más lineal que también es eficiente.

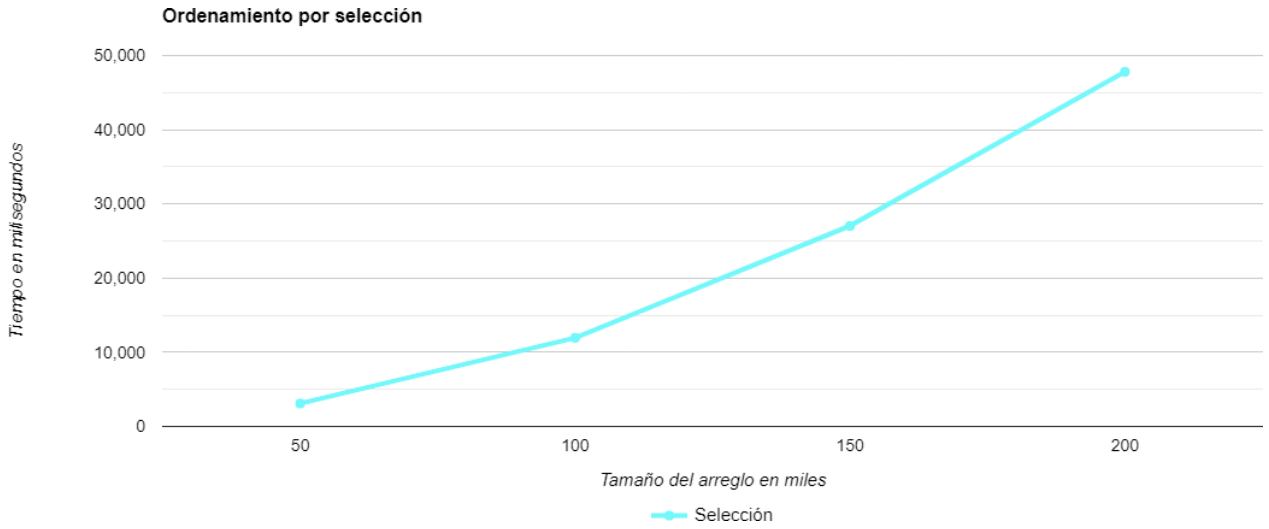

Figura 1. Tiempo promedio en milisegundos del ordenamiento por Selección de miles de datos.

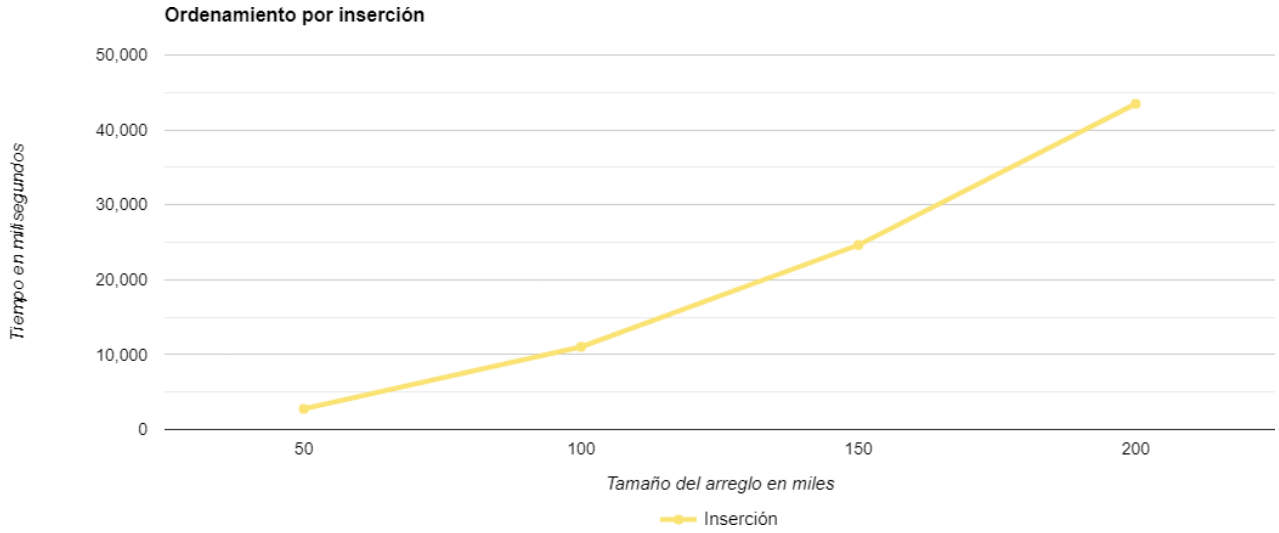

Figura 2. Tiempo promedio en milisegundos del ordenamiento por Inserción de miles de datos.

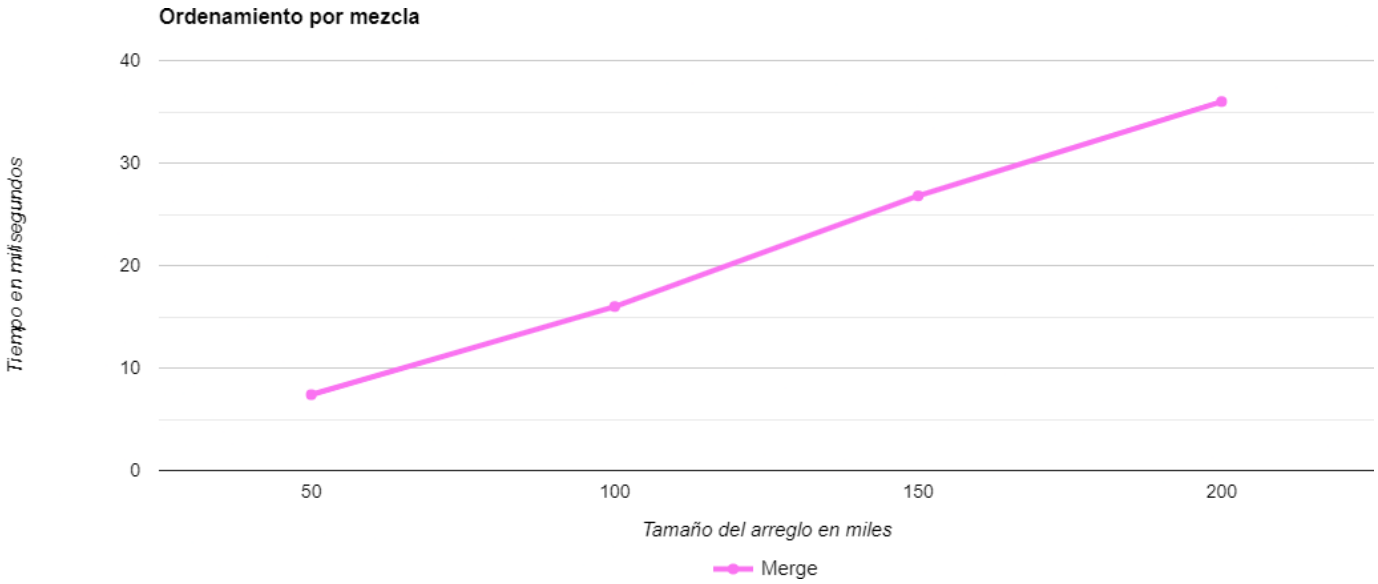

Figura 3. Tiempo promedio en milisegundos del ordenamiento por Mezcla de miles de datos.

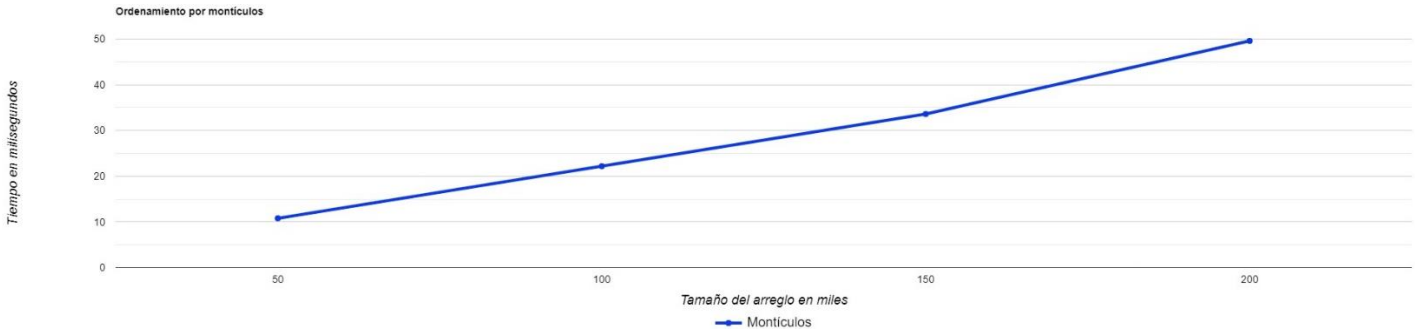

Figura 4. Tiempo promedio en milisegundos del ordenamiento por Montículos de miles de datos.

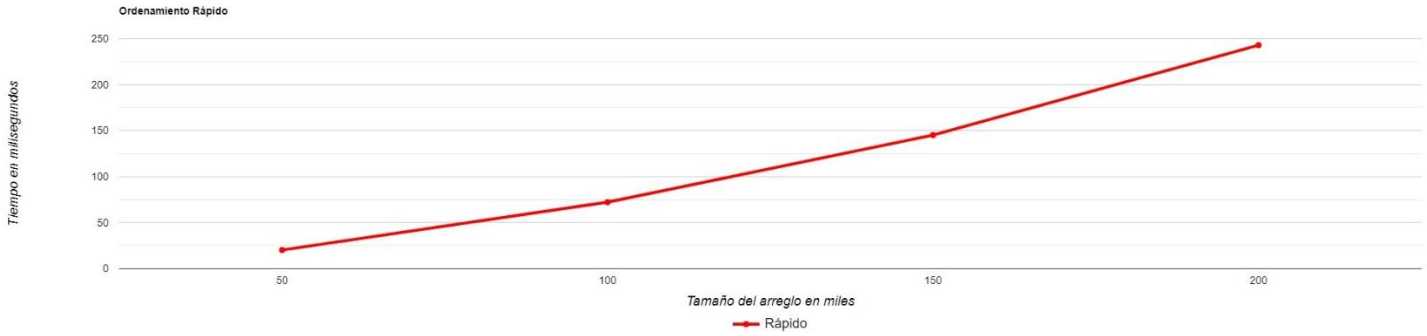

Figura 5. Tiempo promedio en milisegundos del ordenamiento Rápido de miles de datos.

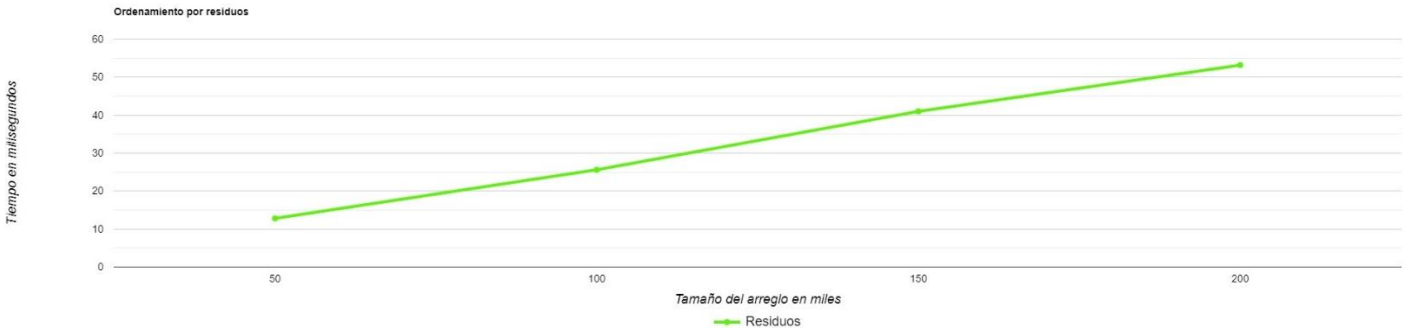

Figura 6. Tiempo promedio en milisegundos del ordenamiento por Residuos de miles de datos.

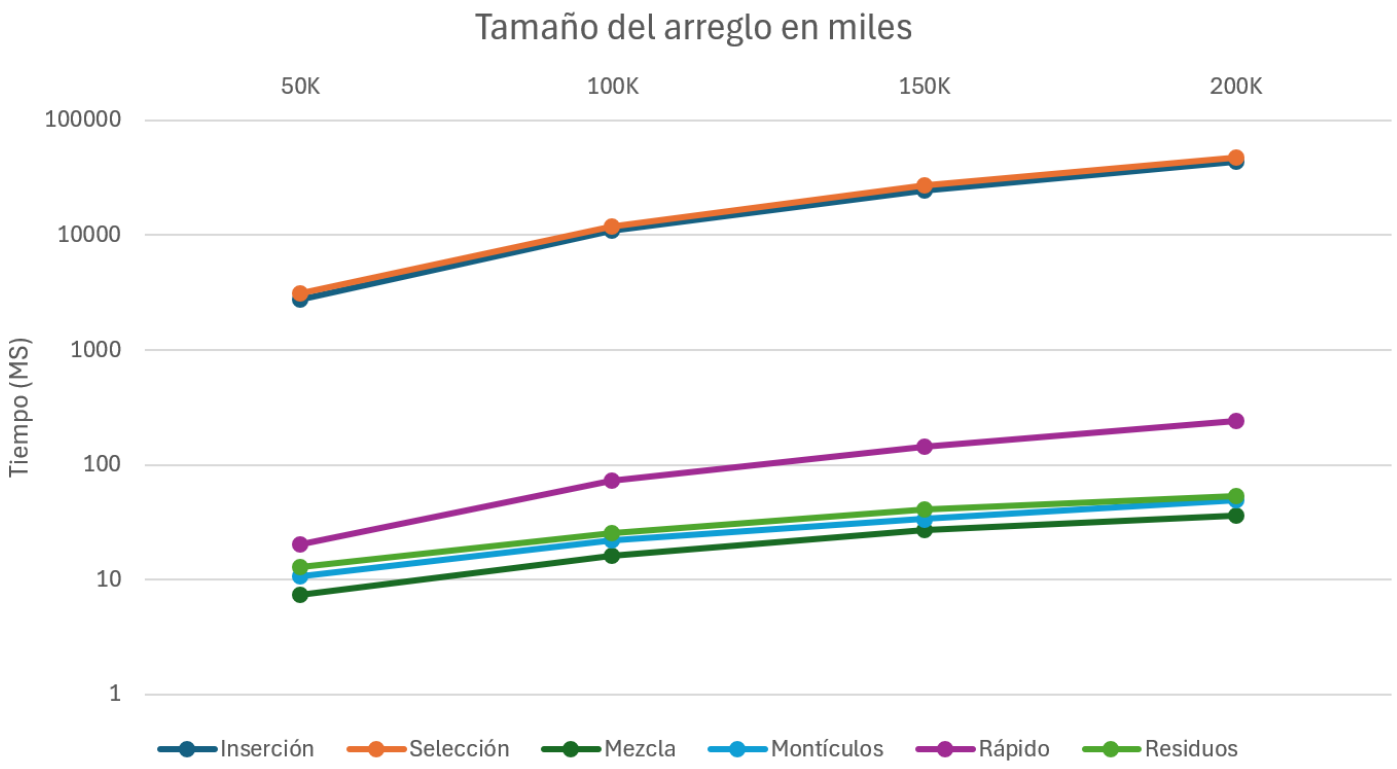

Figura 7. Gráfico comparativo de los tiempos promedio de ejecución en milisegundos de los algoritmos de ordenamiento de miles de datos por Selección, Inserción, Mezcla, Montículos, Rápido y Residuos.

IV. CONCLUSIONES

A partir de los resultados obtenidos en este análisis, se observa que el algoritmo de Selección y de Inserción poseen cierta similitud en su comportamiento y sus tiempos de ejecución ya que tienen forma parabólica, aunque el de Inserción es un poco más rápido que el de Selección.

En cambio, en ordenamiento por Mezcla, Montículos, Rápido y Residuos se observa una curva diferente con respecto a los algoritmos antes mencionados ya que este presenta un tiempo de ejecución casi lineal y todos fueron probados con la misma cantidad de datos lo que muestra claramente al compararlos que estos últimos son más eficientes para el ordenamiento de miles de datos.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

```
void seleccion(int *A, int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min = i;
        for (int j = i + 1; j < n; j++)
        {
            if (A[j] < A[min])
            {
                min = j;
            } // Fin if
        }     // Fin for
        if (min != i)
        {
            cambio(&A[i], &A[min]);
        } // Fin if
    }     // Fin for
}         // Fin seleccion
```

Algoritmo 1. Ordenamiento por Selección

```
void cambio(int *a, int *b)
{
    int temporal = *b;
    *b = *a;
    *a = temporal;
} // Fin cambio
```

Algoritmo 2. Función para hacer cambios

```
void insercion(int *A, int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while (j > 0 && A[j - 1] > A[j])
        {
            cambio(&A[j], &A[j - 1]);
            j--;
        } // Fin while
    }     // Fin for
}         // Fin insercion
```

Algoritmo 3. Ordenamiento por Inserción

```
void mergesort(int *A, int n)
{
    mergeRecursivo(A, 0, n - 1);
} // Fin mergesort
void mergeRecursivo(int *A, int inicio, int final)
{
    if (inicio < final)
    {
        int divido = inicio + (final - inicio) / 2;
        mergeRecursivo(A, inicio, divido);
        mergeRecursivo(A, divido + 1, final);
        merge(A, inicio, divido, final);
    } // Fin if
}     // Fin mergeRecursivo
```

```
void merge(int *A, int izquierda, int divido, int derecha)
{
    int tamano1 = divido - izquierda + 1;
    int tamano2 = derecha - divido;

    int arreglo1[tamano1], arreglo2[tamano2];

    for (int i = 0; i < tamano1; i++)
    {
        arreglo1[i] = A[izquierda + i];
    } // Fin for

    for (int j = 0; j < tamano2; j++)
    {
        arreglo2[j] = A[divido + 1 + j];
    } // Fin for

    int i = 0;
    int j = 0;
    int k = izquierda;

    while (i < tamano1 && j < tamano2)
    {
        if (arreglo1[i] <= arreglo2[j])
        {
            A[k] = arreglo1[i];
            i++;
        }
        else
        {
            A[k] = arreglo2[j];
            j++;
        } // Fin if
        k++;
    } // Fin while

    while (j < tamano2)
    {
        A[k] = arreglo2[j];
        j++;
        k++;
    } // Fin while
}     // Fin merge
```

Algoritmo 4. Ordenamiento por Mezcla.

```
/* Algoritmo de ordenamiento no comparativo que funciona separando los
elementos a ordenar en "dígitos" y ordenándolos de manera sucesiva en
función de cada uno */
void radixsort(int *A, int n)
{
    // Arreglo temporal para almacenar resultados intermedios
    auto *arregloTemporal = new int[n];
    int largoBit = sizeof(int) * 8;
    int contador[2];
    int prefijo[2];

    // Máscara para manejo de números negativos
    int mascara = 1;
    int numerosNegativos = 0;
    int numerosPositivos = 0;
    unsigned int valorBit;

    // Unión que contiene float e int
    union ufi
    {
        float f;
        int i;
    } ufi;

    // Itera de bit menos significativo a más significativo
    for (int shift = 0; shift < largoBit; ++shift)
    {
        // Inicializa arreglo
        for (int i = 0; i < 2; ++i)
        {
            contador[i] = 0;
        } // Fin for

        // Cuenta cuántas veces aparece el valor del bit en el arreglo
        for (int i = 0; i < n; ++i)
        {
            ufi.f = A[i];
            valorBit = (ufi.i >> shift) & mascara;
            contador[valorBit]++;
            // Cuenta número negativos durante primera iteración
            if (shift == 0 && A[i] < 0)
            {
                numerosNegativos++;
            } // Fin if
        }

        // Cuenta números positivos
        if (shift == 0)
        {
            numerosPositivos = n - numerosNegativos;
        } // Fin if

        // Calcula prefijos acumulativos en "contador"
        prefijo[0] = 0;
        for (int i = 1; i < 2; ++i)
        {
            prefijo[i] = prefijo[i - 1] + contador[i - 1];
        } // Fin for
```

```
        // Distribuye números en A basándose en valor de bit actual
        for (int i = 0; i < n; ++i)
        {
            ufi.f = A[i];
            valorBit = (ufi.i >> shift) & mascara;
            int index = prefijo[valorBit]++;
            // Ajusta índices para manejar negativos en última iteración
            if (shift == largoBit - 1)
            {
                if (A[i] < 0)
                {
                    index = numerosPositivos - (index - numerosNegativos) - 1;
                }
                else
                {
                    index += numerosNegativos;
                } // Fin if
            } // Fin if
            // Almacena elemento en arreglo temporal (en posición calculada)
            arregloTemporal[index] = A[i];
        } // Fin for
        // Copia elementos de "arregloTemporal" en A
        memcpy(A, arregloTemporal, sizeof(int) * n);
    } // Fin for
    delete[] arregloTemporal; // Libera memoria
} // Fin radixsort
```

Algoritmo 5. Ordenamiento por Residuos

```c
//  Algoritmo de ordenamiento Quicksort
void quicksort(int *A, int n)
{
    actualQuickSort(A, 0, n);
} // Fin quicksort
```

```c
/* Divide el arreglo en dos subarreglos alrededor de un pivote
y coloca los elementos menores a su izquierda y los mayores a su derecha */
int particion(int *A, int inicio, int fin)
{
    // Selecciona el último elemento del arreglo como pivote
    int pivote = A[fin];
    // Inicializa el índice del elemento más pequeño, que se incrementará
    //cuando se encuentre un elemento menor que el pivote
    int i = (inicio - 1);

    // Itera sobre el subarreglo desde el índice de inicio hasta el índice de fin - 1
    for (int j = inicio; j <= fin - 1; j++)
    {
        if (A[j] < pivote) // Si actual menor que pivote
        {
            i++; // Incremente el índice del elemento mas pequeno
            cambio(&A[i], &A[j]);
        } // Fin if
    } // Fin for
    cambio(&A[i + 1], &A[fin]);
    return (i + 1); // Indice del pivote
} // Fin particion

/* Ordenar los subarreglos izquierdo y derecho del pivote, hasta que todos
los subarreglos sean de longitud uno o cero, lo que significa que
el arreglo está ordenado */
void actualQuickSort(int *A, int inicio, int fin)
{
    // Verifica si el índice de inicio es menor que el índice final, lo
    //que indica que hay más de un elemento en el subarreglo
    if (inicio < fin)
    {
        // Divide el arreglo en dos subarreglos y obtiene el índice del elemento pivote
        int medio = particion(A, inicio, fin);
        // Ordenar el subarreglo izquierdo del pivote
        actualQuickSort(A, inicio, medio - 1);
        // Ordenar el subarreglo derecho del pivote
        actualQuickSort(A, medio + 1, fin);
    } // Fin if
} // Fin actualQuickSort
```

Algoritmo 6. Ordenamiento Rápido

```
void heapSort(int *A, int n)
{
    // Hace el monticulo
    for (int i = (n / 2 - 1); i >= 0; i--)
    {
        monticulos(A, n, i);
    } // Fin for

    // Ordena el monticulo
    for (int i = (n - 1); i > 0; i--)
    {
        cambio(&A[0], &A[i]);
        monticulos(A, i, 0); // Restaura la propiedad del montículo en el subárbol restante
    } // Fin for
} // Fin heapSort
// Fin class
#dif
```

```
// Garantiza que la propiedad del montículo se cumpla en un subárbol con raíz en el nodo i
void monticulos(int *A, int n, int i)
{
    // Selecciona el nodo actual como el máximo
    int maximo = i;
    int izquierda = (2 * i) + 1;
    int derecha = (2 * i) + 2;

    // Comparación con el hijo izquierdo
    if (izquierda < n && A[izquierda] > A[maximo])
    {
        maximo = izquierda;
    } // Fin if

    // Comparación con el hijo derecho
    if (derecha < n && A[derecha] > A[maximo])
    {
        maximo = derecha;
    } // Fin if

    // Si el máximo no es el nodo actual, se intercambia y se realiza la recursión
    if (maximo != i)
    {
        cambio(&A[i], &A[maximo]);
        monticulos(A, n, maximo);
    } // Fin if
} // Fin monticulos
```

Algoritmo 7. Ordenamiento por montículos

REFERENCIAS

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.