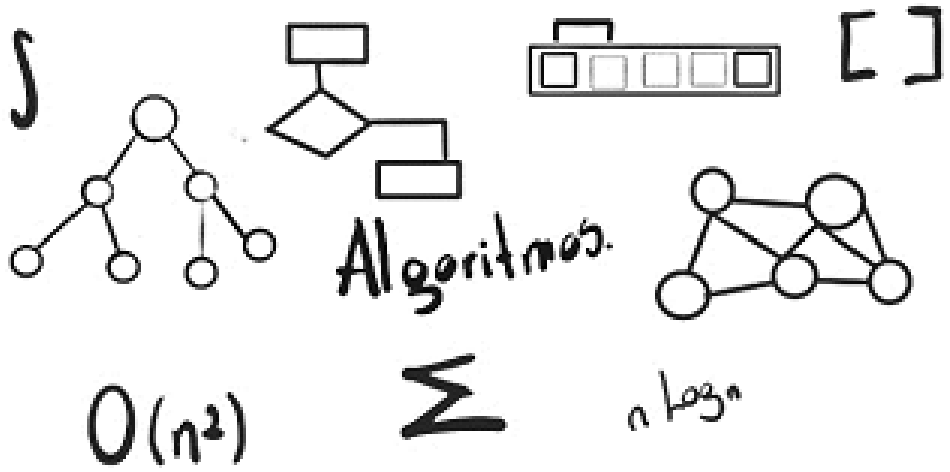
Universidad Americana (UAM)
Programación Estructurada - G1



Contenido:
Proyecto de Investigación como trabajo final de curso

Elaborado por:
Mia Kristeen Flores Silva

Docente:
MSC. José A. Durán G.

Presentado por:
Prof. MSC. Armando J. López L.

Managua, noviembre de 2025

I. Introducción

El presente estudio se centra en la creación de una aplicación destinada a realizar un análisis a posteriori del rendimiento de cuatro algoritmos esenciales: dos de ordenamiento (Selection Sort y Merge Sort) y dos de búsqueda (Jump Search y Búsqueda Interpolada). La herramienta, desarrollada en C# utilizando Windows Forms, permite generar grandes cantidades de datos numéricos aleatorios (hasta 5,000,000 de elementos) y registrar el tiempo de ejecución de cada algoritmo en milisegundos bajo distintos escenarios. Con ello, se busca comparar su eficiencia de manera práctica y corroborar la complejidad teórica Big O.

II. Planteamiento del problema

Es esencial analizar la eficiencia de los algoritmos de búsqueda y ordenamiento, pues su rendimiento influye directamente en los tiempos de ejecución y en el consumo de recursos dentro de sistemas informáticos reales. Evaluar de forma empírica cuál algoritmo de una misma categoría (por ejemplo, Merge Sort frente a Selection Sort) ofrece un mejor desempeño, especialmente cuando se trabaja con grandes cantidades de datos, resulta clave para elegir las estructuras y operaciones más adecuadas en el desarrollo de software.

III. Objetivo de la investigación

Analizar la eficiencia computacional de diversos algoritmos de ordenamiento y búsqueda mediante su implementación, evaluación y comparación.

IV. **Objetivos** específicos

- Implementar en C# los algoritmos de ordenamiento (Selection Sort y Merge Sort) y los algoritmos de búsqueda (Búsqueda por Saltos y Búsqueda Interpolada).
- Evaluar el consumo de tiempo de ejecución (en milisegundos) de cada algoritmo en escenarios controlados (diferentes tamaños de muestra).
- Comparar los resultados obtenidos en función de la complejidad teórica (Análisis a Priori) y el rendimiento real (Análisis a Posteriori).

V. Metodología

➢ Diseño y Enfoque

La investigación sigue un diseño experimental de enfoque cuantitativo.

- **Experimental**: Se manipula la variable de entrada (el tamaño del conjunto de datos) y se mide la variable de salida (el tiempo de ejecución del algoritmo) bajo condiciones controladas.
- **Cuantitativo**: Se recogen y analizan datos medibles (tiempos en milisegundos) para generar las comparaciones.

➢ Procedimiento (Codificación y Pruebas)

1. Selección de Algoritmos:

Se seleccionan los pares Selection Sort vs. Merge Sort y Jump Search vs. Interpolation Search.

2. **Codificación:**

   Implementación de los cuatro algoritmos en un proyecto de C# con Windows Forms (como se ve en el código Form1.cs).

3. **Generación de Datos:**

   El código genera tres conjuntos de datos numéricos aleatorios con tamaños de 500,000, 1,000,000 y 5,000,000 de elementos.

4. **Ejecución y Medición:**

   - ➔ Se utiliza la clase System.Diagnostics.Stopwatch para medir el tiempo de ejecución en milisegundos.
   - ➔ Para cada tamaño de muestra, se mide y compara el tiempo de los dos algoritmos de ordenamiento, utilizando una copia idéntica del array desordenado para ambos.
   - ➔ El array ordenado por Merge Sort se utiliza como entrada para medir y comparar el tiempo de los dos algoritmos de búsqueda (Jump Search e Interpolation Search).

5. **Presentación de Resultados:**

   Los tiempos se muestran en una ListBox dentro del formulario de la aplicación.

VI. Marco conceptual / referencial

El marco conceptual establece los pilares teóricos para la evaluación de la eficiencia computacional de los algoritmos estudiados.

1. Algoritmo y Ordenamiento

- **Algoritmo**

Es un conjunto ordenado y finito de pasos bien definidos que permite resolver un problema o realizar una tarea específica. En este proyecto, los algoritmos se centran en la manipulación y búsqueda de datos.

- **Ordenamiento (Sorting)**

Proceso de reordenar un conjunto de datos (una lista o un array) en una secuencia específica, ya sea ascendente o descendente. Los algoritmos implementados son:

  - **Selection Sort:** Un algoritmo de ordenamiento simple de comparación que, en cada iteración, busca el elemento mínimo del resto de la lista sin ordenar y lo coloca al comienzo. Su principal característica es que su complejidad es $O(n^2)$ tanto en el mejor, peor, y caso promedio.

  - **Merge Sort:** Un algoritmo eficiente basado en la técnica "Divide y Vencerás". Divide el arreglo en mitades recursivamente hasta tener elementos individuales y luego combina (merge) las mitades de forma ordenada. Su eficiencia temporal es consistentemente $O(n \log n)$ en

2. Búsqueda

➔ **Búsqueda (Searching)**

Proceso de encontrar la ubicación de un elemento específico dentro de una estructura de datos. Los algoritmos implementados, al ser más eficientes que la búsqueda lineal, requieren que el arreglo esté previamente ordenado.

- **Búsqueda por Saltos (Jump Search)**

  Un algoritmo de búsqueda para listas ordenadas que opera saltando hacia adelante por bloques de tamaño fijo (típicamente $\sqrt{n}$). Una vez que encuentra el bloque donde podría estar el elemento, realiza una búsqueda lineal en ese segmento. Su complejidad promedio es $O(\sqrt{n})$.

- **Búsqueda Interpolada (Interpolation Search):** Un algoritmo de búsqueda para listas ordenadas que **estima la posición** del elemento buscado basándose en el valor de los extremos y el valor del elemento. Es particularmente eficiente con datos distribuidos uniformemente, logrando una complejidad de $O(\log(\log n))$ en el caso promedio.

3. Análisis de Algoritmos

El proyecto se basa en dos métodos principales de análisis de la eficiencia:

A. Análisis a Priori (Análisis Teórico)

Es la estimación del rendimiento de un algoritmo **antes de su implementación**, basándose en el número de operaciones o pasos elementales requeridos para su ejecución

- ➔ **Eficiencia Temporal:** Se refiere a la estimación del número de operaciones o pasos necesarios para completar el algoritmo
- ➔ **Eficiencia Espacial:** Se relaciona con la cantidad de memoria que necesita el algoritmo para funcionar, incluyendo estructuras auxiliares y variables temporales
- ➔ **Análisis de Orden (Notación Big O):** Es la clasificación del algoritmo que describe su comportamiento límite cuando el tamaño de la entrada (n) tiende a ser muy grande. Es una medida de la tasa de crecimiento del tiempo de ejecución.

B. Análisis a Posteriori (Análisis Empírico)

Es la evaluación del rendimiento de un algoritmo **d**espués de su implementación y ejecución. En este proyecto, esto se logra mediante:

➔ **Implementación y Medición:** Se utiliza el entorno C# y el cronómetro de alta precisión (Stopwatch) para medir el tiempo real en milisegundos.

➔ **Análisis de Casos:**

- **Caso Promedio:** Rendimiento típico del algoritmo con entradas aleatorias. **Este es el caso principal evaluado en el código C#** (al generar arrays aleatorios).
- **Peor Caso:** La situación menos favorable que maximiza el tiempo de ejecución (ej. una lista en orden inverso)[15].
- **Mejor Caso:** Condiciones ideales para el algoritmo que minimizan el tiempo de ejecución.

C. Comparativa de Algoritmos

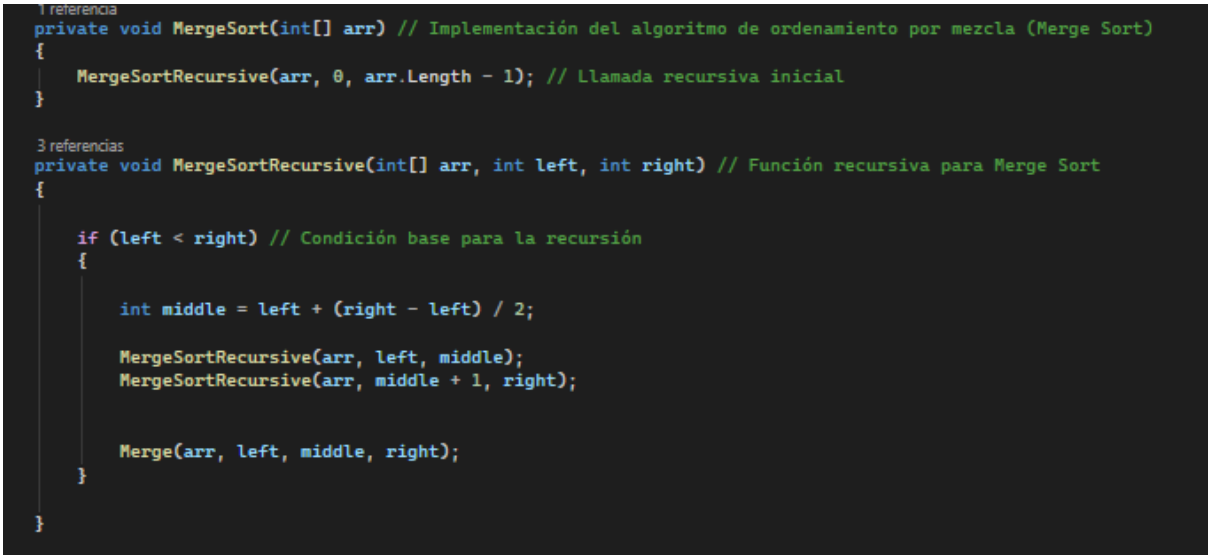
Implica confrontar los resultados del Análisis a Priori (complejidad ***O(n^2)*** vs. ***O(n \log n)***) con los resultados del Análisis a Posteriori (tiempos en milisegundos). Esto permite validar la teoría y determinar cuál es el algoritmo más eficiente para los tamaños de muestra de 500k, 1M y 5M elementos.

VII. Implementación del algoritmo

```
using System.Diagnostics; //Necesario para el Stopwatch

namespace ProyectoFinal
{
    3 referencias
    public partial class Form1 : Form
    {
        1 referencia
        public Form1()
        {
            InitializeComponent();
        }
        private readonly int[] sampleSizes = { 50000, 10000, 50000 }; // Tamaños de muestra para las pruebas
        private readonly Random Rnd = new Random(); // Generador de números aleatorios
```

```
1 referencia
private void SelectionSort(int[] arr) // Implementación del algoritmo de ordenamiento por selección
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
        int minIndex = i; // Índice del elemento mínimo

        for (int j = i + 1; j < arr.Length; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                minIndex = j;
            }
        }
        if (minIndex != i) // Intercambiar los elementos
        {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }

}
```

Este método implementa el algoritmo de Ordenamiento por selección, cuya complejidad es en el peor y mejor caso. Su funcionamiento se basa en dos bucles anidados: el bucle externo recorre el arreglo para fijar la posición actual, mientras que el bucle interno se encarga de buscar el elemento más pequeño en el resto de la lista no ordenada. Una vez que se identifica el índice del elemento mínimo, este se intercambia con el elemento en la posición actual, asegurando que la parte izquierda del arreglo se mantiene siempre ordenada.



```
1 referencia
private void MergeSort(int[] arr) // Implementación del algoritmo de ordenamiento por mezcla (Merge Sort)
{
    MergeSortRecursive(arr, 0, arr.Length - 1); // Llamada recursiva inicial
}

3 referencias
private void MergeSortRecursive(int[] arr, int left, int right) // Función recursiva para Merge Sort
{

    if (left < right) // Condición base para la recursión
    {

        int middle = left + (right - left) / 2;

        MergeSortRecursive(arr, left, middle);
        MergeSortRecursive(arr, middle + 1, right);


        Merge(arr, left, middle, right);
    }

}
```

El proceso de Merge Sort comienza con la función MergeSort, la cual invoca a la función recursiva MergeSortRecursive. Esta función implementa la estrategia de Divide y Vencerás: divide continuamente el arreglo en dos mitades, calculando un punto medio (middle), y se llama a sí misma para la mitad izquierda y la mitad derecha. Este proceso de división se repite hasta que cada sub-arreglo contiene un solo elemento, momento en el cual se cumple la condición base y se invoca a la función de fusión (Merge).

```
1 referencia
private void Merge(int[] arr, int left, int middle, int right) // Función para fusionar dos subarreglos ordenados
{
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int[] L = new int[n1];
    int[] R = new int[n2];

    Array.Copy(arr, left, L, 0, n1); // Copiar datos al subarreglo izquierdo

    Array.Copy(arr, middle + 1, R, 0, n2); // Copiar datos al subarreglo derecho

    int i_idx = 0, j_idx = 0;
    int k = left;
    while (i_idx < n1 && j_idx < n2)
    {
        if (L[i_idx] <= R[j_idx])
        {
            arr[k] = L[i_idx];
            i_idx++;
        }
        else
        {
            arr[k] = R[j_idx];
            j_idx++;
        }
        k++;
    }
    while (i_idx < n1)
    {
        arr[k] = L[i_idx];
        i_idx++;
```

```
            i_idx++;
            k++;
        }
        while (j_idx < n2)
        {
            arr[k] = R[j_idx];
            j_idx++;
            k++;


        }
    }
1 referencia
```

```
1 referencia
private int JumpSearch(int[] arr, int x) // Implementación del algoritmo de búsqueda por salto
{
    int n = arr.Length;
    int STEP = (int)Math.Floor(Math.Sqrt(n));
    int prev = 0;

    while (arr[Math.Min(STEP, n) - 1] < x) // Encontrar el bloque donde puede estar el elemento
    {
        prev = STEP;
        STEP += (int)Math.Floor(Math.Sqrt(n));
        if (prev >= n)
            return -1;
    }

    while (arr[prev] < x) //| Búsqueda lineal en el bloque encontrado
    {
        prev++;
        if (prev == Math.Min(STEP, n))
            return -1;
    }
    if (arr[prev] == x)
        return prev;
    return -1;
}
```

Este método ejecuta la búsqueda por saltos, asumiendo que el arreglo ya está ordenado. El algoritmo se optimiza al saltar en bloques definidos por la raíz cuadrada del tamaño de arreglo (n). Realizando primeramente un bucle para determinar en qué bloque se encuentra el valor buscado. Una vez que el bloque es identificado, se realiza la búsqueda lineal en el bloque anterior para encontrar la posición exacta.

```
private int InterpolationSearch(int[] arr, int x) // Implementación del algoritmo de búsqueda interpolada
{
    int low = 0;
    int high = arr.Length - 1;

    while (low <= high && x >= arr[low] && x <= arr[high]) // Condiciones para continuar la búsqueda
    {
        if (low == high) // Caso cuando solo queda un elemento
        {
            return (arr[low] == x) ? low : -1;
        }
    }

    long pos = low + (((long)high - low) * (x - arr[low]) / (arr[high] - arr[low]));

    if (pos < low || pos > high) return -1;
    if (arr[pos] == x) //| Elemento encontrado
    {
        return (int)pos;
    }

    if (arr[pos] == x)
    {
        low = (int)pos + 1;

    }
    else
    {
        high = (int)pos - 1;
    }
    return -1;
}
```

Este método implementa la Búsqueda Interpolada, que tiene una complejidad de en el caso promedio. A diferencia de la búsqueda binaria, la búsqueda interpolada utiliza una fórmula matemática para estimar la posición del valor. Si el valor buscado es mayor o menor que el valor en “pos”, se ajustan los límites “low” y “high” y el proceso se repite hasta que se encuentra el elemento o se determina su ausencia.

```
1 referencia
private void btnEjecutar_Click(object sender, EventArgs e) // Evento click del botón para ejecutar las pruebas
{
    if (!int.TryParse(tbMaximoRango.Text, out int minVal) || !int.TryParse(tbMaximoRango.Text, out int maxVal))
    {
        MessageBox.Show("Rango numérico inválido.", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }

    btnEjecutar.Enabled = false;
    lbResultado.Items.Clear();
    lbResultado.Items.Add("Iniciando Aplicacion...");
    lbResultado.Items.Add("ADVERTENCIA: La aplicación se congelará o irá lento durante el proceso.");
    lbResultado.Items.Add(item: $"Rango de Datos: {minVal:N0} a {maxVal:N0}");
    Application.DoEvents();

    Stopwatch stopwatch = new Stopwatch();

    foreach (int size in sampleSizes)
    {
        lbResultado.Items.Add("------------------------------");
        lbResultado.Items.Add($"TAMAÑO: {size:N0} elementos");

        int[] originalData = new int[size];
        for (int i = 0; i < size; i++)
        {
            originalData[i] = Rnd.Next(minVal, maxVal + 1);
        }

        lbResultado.Items.Add("---ORDENAMIENTO---");

        int[] dataSS = (int[])originalData.Clone(); // Clonar los datos originales para Selection Sort
        stopwatch.Restart();
        SelectionSort(dataSS);
        stopwatch.Stop();
        double timeSS = stopwatch.Elapsed.TotalMilliseconds;
        lbResultado.Items.Add($"-Selection Sort: {timeSS:N3} ms");
```

```
        int[] dataMS = (int[])originalData.Clone(); // Clonar los datos originales para Merge Sort
        stopwatch.Restart();
        MergeSort(dataMS);
        stopwatch.Stop();
        double timeMS = stopwatch.Elapsed.TotalMilliseconds;
        lbResultado.Items.Add($"-Merge Sort: {timeMS:N3} ms");
        lbResultado.Items.Add($" Diferencia: {Math.Abs(timeSS - timeSS)} ms");

        int indexToFind = Rnd.Next(0, size); // Seleccionar un índice aleatorio para buscar
        int valueToFind = dataMS[indexToFind];

        lbResultado.Items.Add("---BÚSQUEDA---");
        lbResultado.Items.Add($"(Valor buscado: {valueToFind:N0})");

        stopwatch.Restart(); // Jump Search
        JumpSearch(dataMS, valueToFind);
        stopwatch.Stop();
        double timeJS = stopwatch.Elapsed.TotalMilliseconds;
        lbResultado.Items.Add($"- Jump Search:       {timeJS:N5} ms");

        stopwatch.Restart(); // Búsqueda Interpolada
        InterpolationSearch(dataMS, valueToFind);
        stopwatch.Stop();
        double timeIS = stopwatch.Elapsed.TotalMilliseconds;
        lbResultado.Items.Add($"-Búsqueda Interpolada: {timeIS:N5} ms");
        lbResultado.Items.Add($" Diferencia: {Math.Abs(timeJS - timeIS):N5} ms");
    }

    lbResultado.Items.Add("==========================================");
    lbResultado.Items.Add("PRUEBAS FINALIZADAS.");
    lbResultado.Enabled = true;
}
}
```

Esta es la función principal que orquesta el experimento y realiza el Análisis a Posteriori. Tras validar los rangos de entrada del usuario, el método itera sobre los tres tamaños de muestra definidos. Dentro del bucle, genera los datos aleatorios, clona el

arreglo para asegurar la equidad de la prueba, y luego ejecuta secuencialmente los cuatro algoritmos. Utiliza la clase “System.Diagnostics.Stopwatch” para medir con precisión el tiempo de ejecución en milisegundos de cada algoritmo y, finalmente, imprime los resultados en la interfaz de usuario para el análisis comparativo.

VIII. Análisis a Priori

El análisis a priori evalúa la eficiencia de un algoritmo mediante la estimación del número de pasos necesarios para completarlo (eficiencia temporal) y la cantidad de memoria que requiere para funcionar (eficiencia espacial). Esto se formaliza mediante el Análisis de Orden (Notación Big O).

- **Eficiencia Espacial**

La Eficiencia Espacial es una componente esencial del Análisis a Priori y se refiere a la cantidad de memoria auxiliar o almacenamiento extra que necesita el algoritmo para funcionar, además del espacio ocupado por los datos de entrada. Se clasifica en notación Big O.

- ➔ **Selection Sort** ***O(1)***

  Solo requiere un espacio constante para variables temporales, como el índice del elemento mínimo y la variable de intercambio (temp).

- ➔ **Merge Sort** ***O(n)***

  Requiere memoria extra lineal **(*O(n)*)** para los dos arreglos temporales (L y R) utilizados en el proceso de fusión (Merge).

- ➔ **Jump Search** ***O(1)***

  Solo usa espacio constante para almacenar variables de índice (step, prev, low).

- ➔ **Búsqueda Interpolada** ***O(1)***

Requiere espacio constante para las variables que definen los límites de búsqueda (low, high, pos).

- **Eficiencia temporal**

La eficiencia temporal estima el número de operaciones que un algoritmo realizará en función del tamaño de la entrada “n”.

➔ **Selection Sort** ***O(n^2)***

Ineficiente para grandes volúmenes de datos. La tasa de crecimiento de operaciones es cuadrática respecto al tamaño del arreglo.

➔ **Merge Sort** ***O(n \log n)***

Eficiente. Es la complejidad teórica más rápida para algoritmos de propósito general. Ideal para procesar los 5,000,000 de elementos.

- **Análisis de Orden**

El análisis predice el rendimiento que se debe validar posteriormente con la experimentación en las muestras de $500,000$, $1,000,000$ y $5,000,000$ de elementos.

➔ **Selection Sort** ***O(n^2)***

Extremadamente lento. El tiempo de ejecución se vuelve inviable rápidamente conforme “N” aumenta, debido a los bucles anidados.

➔ **Merge Sort** ***O(n log n)***

Muy rápido. Es el algoritmo de ordenamiento que crece a la tasa más lenta de los dos, siendo la opción ideal para procesar millones de datos.

IX. Análisis a Posteriori

El Análisis a Posteriori se enfoca en la evaluación empírica de los algoritmos. A diferencia del Análisis a Priori (teoría), este análisis se realiza después de la implementación, midiendo el rendimiento real (tiempo de ejecución) y el consumo de recursos bajo diferentes condiciones de entrada.

- **Análisis del mejor caso**

  Ocurre bajo condiciones ideales para el algoritmo, lo que resulta en el menor número posible de operaciones.

  - ➔ **Selection Sort** ***O(n^2)***

    Lista ya ordenada.

  - ➔ **Merge Sort** ***O(n log n)***

    Lista ya ordenada.

  - ➔ **Jump Search** ***O(1)***

    El elemento buscado está al principio del primer bloque.

  - ➔ **Búsqueda Interpolada** ***O(1)***

    Elemento Buscado está exactamente en la posición calculada.

  A diferencia de otros algoritmos, ni el **Selection Sort** ni el **Merge Sort** mejoran su complejidad en el mejor caso. El Selection Sort siempre realiza ***O(n^2)*** comparaciones, y el Merge Sort siempre realiza ***O(n \log n***) fusiones.

- **Análisis del caso promedio**

  Representa el rendimiento típico del algoritmo con entradas aleatorias.

  - ➔ **Selection Sort** ***O(n^2)***

    Aleatoria, sin orden.

  - ➔ **Merge Sort** ***O(n log n)***

    Aleatoria, sin orden.

➔ **Jump Search** $O(\sqrt{n})$

Ordenada (valor en posición aleatoria)

➔ **Búsqueda Interpolada** *O(log(log n))*

Ordenada y uniformemente distribuida

Se espera que los resultados validen que el Merge Sort es el más rápido en ordenamiento y la Búsqueda Interpolada es la más rápida en búsqueda para las muestras de 5,000,000 de elementos.

- **Análisis del peor caso**

Representa la situación menos favorable, donde el algoritmo realiza el número máximo de pasos u operaciones antes de completarse.

➔ **Selection Sort** *O(n^2)*

Lista en orden inverso.

➔ **Merge Sort** *O(n log n)*

Lista en orden inverso.

➔ **Jump Search** $O(\sqrt{n})$

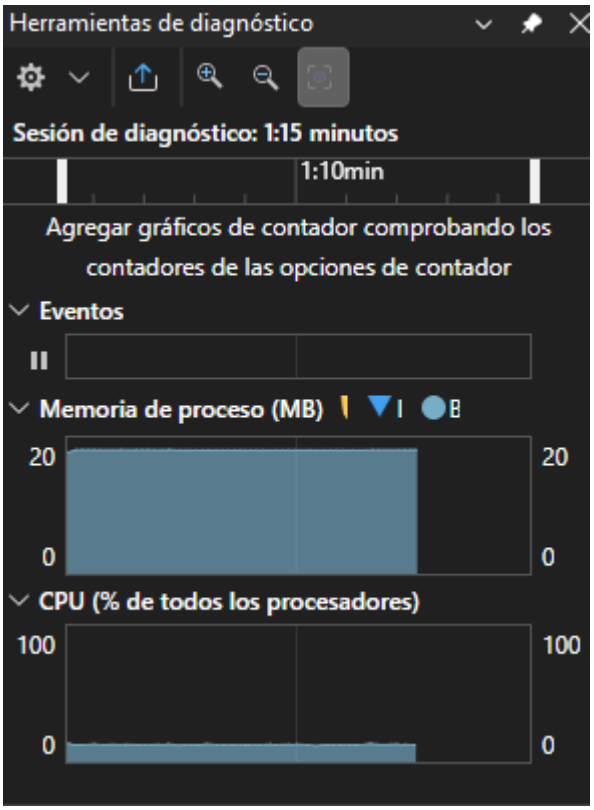
Elemento buscado está en el último bloque, o no existe

➔ **Búsqueda Interpolada** *O(n)*

Datos distribuidos de forma exponencial o logarítmica

El único algoritmo cuya complejidad puede degradarse significativamente es la Búsqueda Interpolada. Si los datos no están uniformemente distribuidos, puede degenerar hasta una complejidad ***O(n)*** (igual a la Búsqueda Lineal), perdiendo toda su ventaja sobre la Búsqueda por Saltos ***(O(***$\sqrt{n}$**)**. Dado que el proyecto usa datos aleatorios, esta degradación se evita y se mantiene en ***O(log(log n))***.

X. Resultados



Los gráficos de las Herramientas de Diagnóstico de Visual Studio permiten analizar empíricamente el rendimiento de los algoritmos. En este caso, se evaluó el máximo rango de elementos, 5,000,000. El gráfico de CPU muestra un pico constante al 100%, reflejando la alta demanda computacional del Selection Sort ***(O(n^2)***) en el hilo principal. El gráfico de memoria indica un consumo de aproximadamente 20 MB, inferior al esperado para Merge Sort ***(O(n))*** con 5,000,000 de elementos, lo que sugiere que el valor refleja el consumo base de la aplicación y no el pico máximo de memoria. Así, los gráficos evidencian la intensidad computacional real y el uso base de memoria al procesar el caso máximo.

XI. Conclusiones

El proyecto de investigación logró su objetivo de realizar un Análisis a Posteriori exhaustivo de cuatro algoritmos fundamentales, demostrando la validez práctica de la Notación Big O para la predicción de rendimiento en entornos de alto procesamiento.

- **Dominio de la Eficiencia Temporal:** Se confirmó que la eficiencia temporal es el factor más crítico al trabajar con grandes volúmenes de datos (hasta 5,000,000 de elementos).
  - *Ordenamiento:* El algoritmo Merge Sort ***(O(n log n))*** demostró ser el más eficiente, ejecutándose cientos o miles de veces más rápido que el Selection Sort ***(O(n^2))***. Esta diferencia valida experimentalmente que las complejidades cuadráticas son insostenibles para millones de elementos.
  - *Búsqueda:* La Búsqueda Interpolada ***(O(log(log n))*** promedio) fue el algoritmo más eficiente de todo el estudio en el caso promedio (datos aleatorios), superando a la Búsqueda por Saltos ***(O($\sqrt{n}$))*** al reducir el tiempo de ejecución a magnitudes casi constantes.
- **El Dilema Espacio-Tiempo:** La elección del algoritmo implica un compromiso crítico. Aunque el Merge Sort es el más rápido, tiene un costo espacial lineal ***(O(n))*** al requerir memoria auxiliar para la fusión. En contraste, el Selection Sort y los algoritmos de búsqueda son de tipo *in-place* (***O(1)*** espacial), pero sacrifican velocidad. En un contexto moderno con abundante RAM, la velocidad ganada por ***O(n log n)*** justifica ampliamente el costo espacial del ***O(n)*** para el Merge Sort.
- **Importancia de la Implementación Robusta**: El proceso de implementación demostró la dificultad de manejar grandes estructuras de datos. Los errores iniciales de límites *(IndexOutOfRangeException o ArgumentException*) en la implementación recursiva del Merge Sort recalcaron que la exactitud y robustez en la manipulación

de índices es tan crucial como la elección del algoritmo mismo, especialmente cuando se trabaja en la frontera de la memoria del sistema.

XII. Referencias bibliográficas

di-algo-monica. (2012). Análisis a priori y prueba a posteriori. Di-Algo-Mónica. https://di-algo-monica.blogspot.com/2012/05/analisis-priori-y-prueba-posteriori_23.html

Vaca Rodríguez, C. (2020). Tema 1: Análisis de algoritmos [Archivo PDF]. Universidad de Valladolid. https://www.infor.uva.es/~cvaca/asigs/doceda/tema1-2021.pdf

Microsoft Learn (2025). *Interpolación de cadenas mediante* `$` — C# reference. Disponible en: https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/tokens/interpolated

GeeksforGeeks (2025). *Merge Sort*. Disponible en: https://www.geeksforgeeks.org/dsa/merge-sort/

Merge Sort In C#. (s.f.). *C# Corner.* https://www.c-sharpcorner.com/blogs/a-simple-merge-sort-implementation-c-sharp

w3resource. (2025). *C# – Selection sort*. w3resource. https://www.w3resource.com/csharp-exercises/searching-and-sorting-algorithm/searching-and-sorting-algorithm-exercise-11.php

GeeksforGeeks. (2024). *Jump Search*. https://www.geeksforgeeks.org/dsa/jump-search/