Universidad Americana UAM

Programación Estructurada, Grupo 1

Análisis de Eficiencia Computacional en Algoritmos de Ordenamiento y Búsqueda

Estudiantes

- Roberto Fabio Tercero Membreño (24014121)

Docente

MSc. José Durán

29 de noviembre de 2025

Índice

I. INTRODUCCIÓN 4

1. Planteamiento del Problema 4
2. Objetivo de la Investigación 4
3. Objetivos Específicos 4

II. METODOLOGÍA 5

1. Diseño de la Investigación 5
2. Enfoque de la Investigación. 5
3. Alcance de la Investigación 5
4. Procedimiento 5

III. MARCO CONCEPTUAL 6

IV. IMPLEMENTACIÓN DEL ALGORITMO 7

1. Algoritmos de Ordenamiento 7

Selection Sort 7

Merge Sort 8

2. Algoritmos de Búsqueda 11

Jump Search 11

Interpolation Search 13

V. ANÁLISIS A PRIORI 14

1. Selection Sort 14
2. Merge Sort 15
3. Jump Search 15
4. Búsqueda Interpolada 15

VI. ANÁLISIS A POSTERIORI 15

1. ANÁLISIS DEL MEJOR CASO 15

2. ANÁLISIS DEL CASO PROMEDIO 16

3. ANÁLISIS DEL PEOR CASO 16

VII. RESULTADOS 16

VIII. CONCLUSIONES 17

IX. REFERENCIAS BIBLIOGRÁFICAS 19

I. Introducción

En el ámbito del desarrollo de software de alto rendimiento, la selección adecuada de algoritmos es una decisión crítica que trasciende la mera funcionalidad. La presente investigación se centra en el análisis comparativo de la eficiencia computacional, abordando tanto la complejidad temporal como el consumo de recursos de memoria.

1. Planteamiento del Problema

Se describe la necesidad de evaluar qué tan eficientes son distintos algoritmos de búsqueda y ordenamiento, dado que su desempeño impacta directamente en el tiempo de ejecución y uso de recursos en aplicaciones reales.

2. Objetivo de la Investigación

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

3. Objetivos Específicos

1. Implementar algoritmos clásicos de ordenamiento (como Selection Sort y Merge Sort).
2. Implementar algoritmos de búsqueda (como Jump Search y Búsqueda Interpolada).
3. Evaluar el consumo de tiempo y memoria en diferentes escenarios.
4. Comparar los resultados obtenidos en función de la complejidad.

II. Metodología

1. Diseño de la Investigación

Experimental, ya que se manipulan estructuras de datos y algoritmos para medir su rendimiento mediante pruebas controladas.

2. Enfoque de la Investigación.

Cuantitativo, porque se recogen y analizan datos medibles como el tiempo de ejecución y el uso de memoria.

3. Alcance de la Investigación

Descriptivo y explicativo, pues se describen las características de los algoritmos y se explican sus comportamientos frente a diferentes entradas.

4. Procedimiento

- Selección de algoritmos a estudiar (Equipo Par: Selection Sort, Merge Sort, Jump Search, Búsqueda Interpolada).
- Codificación e implementación en lenguaje C# utilizando la plataforma .NET.
- Desarrollo de la herramienta AlgorithmLab para la generación de grandes volúmenes de datos.
- Ejecución con distintos tamaños de datos (500,000, 1,000,000 y 5,000,000 elementos).
- Medición de eficiencia mediante instrumentación de software (Stopwatch yProcess.PrivateMemorySize)

- Análisis comparativo.

III. Marco Conceptual

El fundamento teórico de esta investigación se basa en el análisis de la complejidad computacional. Un algoritmo se define como una secuencia finita de instrucciones bien definidas para resolver un problema.

El Análisis A Priori entrega una función que limita el tiempo de cálculo de un algoritmo. Consiste en obtener una expresión que indique el comportamiento del algoritmo en función de los parámetros que influyan, siendo aplicable en la etapa de diseño (di-algo-monica, 2012). Este análisis suele expresarse mediante la notación asintótica Big O, que clasifica los algoritmos según su tasa de crecimiento (Vaca Rodríguez, 2020)

Por otro lado, la Prueba A Posteriori (experimental) recoge estadísticas de tiempo y espacio consumidas por el algoritmo mientras se ejecuta. Según el principio de invarianza, el tiempo de ejecución de dos implementaciones distintas de un mismo algoritmo no diferirá más que en una constante multiplicativa (di-algo-monica, 2012).

Es crucial distinguir entre los escenarios de ejecución:

- Mejor caso: Condiciones ideales donde el algoritmo realiza el mínimo de instrucciones (ej. lista ya ordenada).
- Peor caso: Situación menos favorable (ej. lista inversa), mostrando el máximo coste.
- Caso medio: Rendimiento típico con entradas aleatorias (di-algo-monica, 2012).

La eficiencia se mide en dos dimensiones: tiempo (operaciones ejecutadas) y espacio (memoria adicional requerida). Vaca Rodríguez (2020) destaca que un algoritmo eficiente maximiza el rendimiento minimizando estos recursos.

IV. Implementación del Algoritmo

Para la experimentación se desarrolló el software AlgorithmLab. A continuación, se presen tan los fragmentos clave de la lógica implementada en C#.

1. Algoritmos de Ordenamiento

Se implementó Selection Sort (ordenamiento directo) y Merge Sort (divide y vencerás).

Selection Sort

```
public static void SelectionSort<T>(T[] array) where T : IComparable
{
    ContadorIntercambios = 0;
    int n = array.Length;
    for (int i = 0; i < n - 1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
        {
            if (array[j].CompareTo(array[min_idx]) < 0)
```

```
                {
                    min_idx = j;
                }
            }

            if (min_idx != i)
            {
                T temp = array[min_idx];
                array[min_idx] = array[i];
                array[i] = temp;
                ContadorIntercambios++;
            }
        }
    }
```

Merge Sort

```
    public static void MergeSort<T>(T[] array) where T : IComparable
    {
        ContadorIntercambios = 0;
        if (array.Length <= 1) return;
```

```
            // Creamos el array auxiliar UNA SOLA VEZ aquí fuera para optimizar
memoria
            T[] auxiliar = new T[array.Length];
            MergeSortRecursivo(array, auxiliar, 0, array.Length - 1);
        }

        private static void MergeSortRecursivo<T>(T[] array, T[] auxiliar, int
left, int right) where T : IComparable
        {
            if (left < right)
            {
                int mid = left + (right - left) / 2;
                MergeSortRecursivo(array, auxiliar, left, mid);
                MergeSortRecursivo(array, auxiliar, mid + 1, right);
                Merge(array, auxiliar, left, mid, right);
            }
        }

        private static void Merge<T>(T[] array, T[] auxiliar, int left, int mid,
int right) where T : IComparable
        {
```

```
// Copiamos a auxiliar solo el segmento necesario
for (int i = left; i <= right; i++)
{
    auxiliar[i] = array[i];
}

int i_left = left;
int i_right = mid + 1;
int current = left;

while (i_left <= mid && i_right <= right)
{
    ContadorIntercambios++;
    // Comparamos desde el auxiliar, escribimos en el original
    if (auxiliar[i_left].CompareTo(auxiliar[i_right]) <= 0)
    {
        array[current] = auxiliar[i_left];
        i_left++;
    }
    else
```

```
            {
                array[current] = auxiliar[i_right];
                i_right++;
            }
            current++;
        }

        // Copiar el resto del lado izquierdo (el derecho ya está en su
sitio)
        int remaining = mid - i_left;
        for (int i = 0; i <= remaining; i++)
        {
            array[current + i] = auxiliar[i_left + i];
        }
    }
```

2. Algoritmos de Búsqueda

Se implementaron Jump Search y Búsqueda Interpolada.

Jump Search

```
    public static int JumpSearch<T>(T[] array, T x) where T : IComparable
    {
```

```
PasosRealizados = 0;

int n = array.Length;

int step = (int)Math.Floor(Math.Sqrt(n));

int prev = 0;



while (array[Math.Min(step, n) - 1].CompareTo(x) < 0)

{

    PasosRealizados++;

    prev = step;

    step += (int)Math.Floor(Math.Sqrt(n));

    if (prev >= n) return -1;

}



while (array[prev].CompareTo(x) < 0)

{

    PasosRealizados++;

    prev++;

    if (prev == Math.Min(step, n)) return -1;

}
```

```
        if (array[prev].CompareTo(x) == 0)
            return prev;


        return -1;
    }
```

Interpolation Search

```
    public static int InterpolationSearch(double[] array, double x)
    {
        PasosRealizados = 0;
        int lo = 0, hi = (array.Length - 1);


        while (lo <= hi && x >= array[lo] && x <= array[hi])
        {
            PasosRealizados++;
            if (lo == hi)
            {
                if (array[lo] == x) return lo;
                return -1;
            }
```

```
            // Fórmula de interpolación para estimar la posición
            int pos = lo + (int)(((double)(hi - lo) / (array[hi] -
array[lo])) * (x - array[lo]));

            if (array[pos] == x) return pos;

            if (array[pos] < x)
                lo = pos + 1;
            else
                hi = pos - 1;
        }
        return -1;
    }
```

V. Análisis a Priori

1. Selection Sort

- **Orden:** O(n^2). Realiza dos bucles anidados, comparando cada elemento con el resto.
- **Espacio:** O(1). Es un algoritmo *in-place*, no requiere memoria adicional significativa.

2. **Merge Sort**
    - **Orden:** O(*n* log *n*). Divide el problema logarítmicamente y mezcla linealmente.
    - **Espacio:** O(n). Requiere un array auxiliar del mismo tamaño que la entrada para realizar la mezcla.
3. **Jump Search**
    - **Orden:** O($\sqrt{n}$). Salta bloques de tamaño $\sqrt{n}$ y luego hace una búsqueda lineal.
    - **Espacio:** O(1).
4. **Búsqueda Interpolada**
    - **Orden:** O(log(log *n*)) en caso promedio (datos uniformes) y O(*n)* en el peor caso.
    - **Espacio:** O(1).

VI. Análisis A Posteriori

A continuación, se detalla el comportamiento observado durante la experimentación con *AlgorithmLab.*

1. **Análisis del mejor caso**

Para **Merge Sort**, el mejor caso (datos ya ordenados) mostró una mejora notable en tiempo (0.348s para 500k datos vs 0.444s en aleatorio), aunque la complejidad teórica sigue siendo O(*n* log *n*). La **Búsqueda Interpolada** fue instantánea (0.015s) para encontrar datos en distribuciones uniformes.

2. Análisis del caso promedio

Con datos aleatorios, **Selection Sort** demostró ser inviable para grandes volúmenes. En la prueba de 500,000 elementos, el algoritmo fue detenido tras 10 minutos de ejecución sin finalizar, confirmando la ineficiencia cuadrática $O(n^2)$. **Merge Sort,** en cambio, mantuvo tiempos estables y bajos (8.8s para 5 millones de datos).

3. Análisis del peor caso

En el escenario inverso, **Merge Sort** no sufrió degradación significativa de rendimiento (8.544s para 5M), validando su robustez. Sin embargo, su consumo de memoria (pico de 38.46 MB) fue consistente con la teoría $O(n)$, superior a algoritmos *in-place*.

VII. Resultados

Las siguientes tablas resumen los datos obtenidos mediante la herramienta *AlgorithmLab.*

Tabla 1

Comparativa de Ordenamiento (Merge Sort)

Tamaño (N)	Distribución	Tiempo (s)	Memoria Pico (MB)
500,000	Aleatorio	0.444	2.05
500,000	Ordenado	0.348	2.04
500,000	Inverso	0.379	0.00*

1,000,000	Aleatorio	1.019	0.08*
1,000,000	Ordenando	0.781	0.08*
1,000,000	Inverso	0.968	0.08*
5,000,000	Aleatorio	8.822	38.46
5,000,000	Ordenando	6.143	38.32
5,000,000	Inverso	8.544	38.32

*Valores bajos debido a optimización del GC o medición post-pico.

En **Selection Sort,** en la prueba con 500,000 datos, el proceso fue abortado tras superar los 10 minutos de ejecución, demostrando su inviabilidad para *N* grande.

Tabla 2

Comparativa de Búsqueda (Datos Ordenados)

Tamaño (N)	**Algoritmo**	**Tiempo (s)**	**Pasos/Saltos**	**Memoria (MB)**
500,000	Jump Search	0.004	682	0.08
500,000	Interpolada	0.015	3	3.70
1,000,000	Jump Search	0.005	1,486	0.24
1,000,000	Interpolada	<0.001	4	0.24
5,000,000	Jump Search	0.005	2,404	0.25
5,000,000	Interpolada	<0.001	4	38.54

VIII. Conclusiones

Tras el análisis experimental, se concluye:

1. **Ineficiencia de $O(n^2)$:** Selection Sort es inapropiado para volúmenes de datos industriales (> 100,000 elementos). Su crecimiento cuadrático lo hace inutilizable en tiempo real.
2. **Supremacía de Merge Sort:** Merge Sort demostró ser altamente eficiente y estable en tiempo $O(n \log n)$, independiente de la distribución inicial de los datos. Sin embargo, esto se logra a expensas de un mayor consumo de RAM, como se evidenció en la prueba de 5 millones de datos (38MB).
3. **Eficiencia de Búsqueda:** La Búsqueda Interpolada es superior a Jump Search en datos uniformemente distribuidos, logrando encontrar elementos en tan solo 3 o 4 pasos en un universo de 5 millones. Jump Search, aunque más lento ($O(\sqrt{n})$), es más consistente si la distribución no es uniforme.

IX. Referencias Bibliográficas

di-algo-monica. (2012, 23 de mayo). 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