“UNIVERSIDAD AMERICANA”

Programación Estructurada

Elaborado por:

Dylan Andres Mora Castillo

Investigación proyecto final

Docente:

Jose Duran Garcia

1.Introducción

En este informe se evalúa la eficiencia de distintos algoritmos de ordenamiento y búsqueda implementados en C#. Se generó un conjunto de números aleatorios y se midió el tiempo de ejecución para cada etapa: generación de datos, ordenamiento (algoritmos *bubble sort* y *merge sort*) y búsquedas (*Jump Search* y *Interpolación*). Para la medición se utilizó un cronómetro (Stopwatch), registrando los instantes de inicio y fin, y calculando la duración en segundos. Los resultados obtenidos permiten comparar el comportamiento empírico de los algoritmos con su complejidad teórica.

2.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.

3.Objetivo de la investigación

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

4.Objetivos específicos

1. Implementar algoritmos clásicos de ordenamiento (como SelectionSort, MergeSort).
2. Implementar algoritmos de búsqueda (como JumpSearch y Busqueda Interpolada).
3. Evaluar el consumo de tiempo y memoria en diferentes escenarios.
4. Comparar los resultados obtenidos en función de la complejidad.

5.Metodología

- **Generación de datos:** Se creó una lista de N números aleatorios (valores entre 100 y 499). El tiempo de generación es del orden de O(N), pues se agrega cada elemento a la lista.

- **Ordenamiento por burbuja (Bubble Sort):** Se implementó el clásico algoritmo de ordenamiento de intercambio anidado. En un arreglo de tamaño N, realiza comparaciones dobles, con complejidad $O(N^2)$ en el peor caso.

- **Ordenamiento por mezcla (Merge Sort):** Se aplicó un algoritmo recursivo de **divide y vencerás**, cuyo tiempo de ejecución es O(N log N) en todos los casos. Se mide el tiempo total de la ordenación para la lista generada.

- **Jump Search:** Después de ordenar los datos, se implementó la búsqueda jump. Este algoritmo salta bloques de tamaño óptimo $\sqrt{N}$ y luego realiza búsqueda lineal dentro del bloque adecuado. Busca un valor en la lista ordenada y registra el tiempo de búsqueda.

- **Búsqueda interpolada (Interpolation Search):** También sobre la lista ordenada, se aplicó la búsqueda interpolada, que estima la posición del elemento según distribución uniforme de los datos. Su complejidad promedio es O(log log N) (muy rápida), aunque en el peor caso puede llegar a O(N).

6.Marco conceptual

Algoritmo. Conjunto finito de instrucciones o pasos bien definidos que, aplicados sobre una entrada, producen una salida y terminan en un número finito de pasos. En ciencias de la computación los algoritmos se usan para tareas como ordenamiento, búsqueda y manipulación de estructuras de datos.

Orden. En el contexto de algoritmos de ordenamiento, se refiere a la secuencia en la que los elementos quedan dispuestos (por ejemplo, orden ascendente). En análisis de algoritmos, “orden” también se usa para referirse a la clasificación por complejidad (Notación Big O).

Análisis a priori. Estimación teórica del comportamiento de un algoritmo basada en su estructura (número de operaciones, uso de memoria) sin ejecutar el algoritmo. Se utiliza para derivar complejidades asintóticas (p. ej. O(n), O(n log n), $O(n^2)$).

Análisis a posteriori. Medición empírica del desempeño del algoritmo mediante su ejecución real (tiempos, uso de memoria en una máquina concreta). Complementa el análisis a priori mostrando efectos prácticos (JIT, caché, swapping, etc.).

Comparativa de algoritmos. Contraste entre algoritmos según criterios como complejidad temporal (tiempo asintótico), complejidad espacial (memoria auxiliar), estabilidad, adaptatividad (mejor caso), facilidad de implementación y comportamiento en la práctica (resultados experimentales).

7.Implementación de los algoritmos

Generación de números aleatorios dentro un rango predefinido.

```
1 referencia
private void btnGenerar_Click(object sender, EventArgs e)
{
    btnGenerar.Enabled = false;
    int cantidad = int.Parse(tbCantidadRegistros.Text);
    Random rdn = new Random();
    numeros.Clear();
    Stopwatch sw = new Stopwatch();
    lblInicio.Text = "Inicio: " + DateTime.Now.ToString("hh:mm:ss");
    sw.Start();
    for (int i = 0; i < cantidad; i++)
    {
        numeros.Add((long)rdn.Next(100, 500));
    }
```

Selection Sort

```
for (int i = 0; i < cantidad; i++)
{
    for (int j = i + 1; j < cantidad; j++)
    {
        if (numeros[i] > numeros[j])
        {
            long temp = numeros[i];
            numeros[i] = numeros[j];
            numeros[j] = temp;
        }
    }
}
```

Este código recorre la lista numeros. Para cada posición i busca en las posiciones j > i un elemento menor y lo intercambia con numeros[i] cuando lo encuentra.

Ese comportamiento corresponde al *Selection Sort* (selección del elemento mínimo y colocación en la posición i).

MergeSort

```
3 referencias
private void MergeSort(List<long> lista, int inicio, int fin)
{
    if (inicio < fin)
    {
        int medio = (inicio + fin) / 2;
        MergeSort(lista, inicio, medio);
        MergeSort(lista, medio + 1, fin);
        Merge(lista, inicio, medio, fin);
    }
}
```

```
1 referencia
private void Merge(List<long> lista, int inicio, int medio, int fin)
{
    List<long> temp = new List<long>();
    int i = inicio;
    int j = medio + 1;
    while (i <= medio && j <= fin)
    {
        if (lista[i] <= lista[j])
        {
            temp.Add(lista[i]);
            i++;
        }
        else
        {
            temp.Add(lista[j]);
            j++;
        }
    }
    while (i <= medio)
    {
        temp.Add(lista[i]);
        i++;
    }
    while (j <= fin)
    {
        temp.Add(lista[j]);
        j++;
    }
    for (int k = 0; k < temp.Count; k++)
    {
        lista[inicio + k] = temp[k];
    }
}
```

MergeSort divide recursivamente la lista en mitades hasta sublistas unitarias y luego las combina ordenadas con Merge.

Merge usa una lista temporal temp para almacenar la combinación ordenada de las dos mitades y después copia temp a la porción original.

Es un algoritmo divide y vencerás; por su diseño requiere memoria auxiliar proporcional al tamaño de la subarreglo combinado (en tu implementación temp se crea por cada merge).

JumpSearch

```
1 referencia
private int JumpSearch(List<long> arr, long x)
{
    int n = arr.Count;
    if (n == 0) return -1;
    int step = (int)Math.Floor(Math.Sqrt(n));
    int prev = 0;
    while (prev < n && arr[Math.Min(step, n) - 1] < x)
    {
        prev = step;
        step += (int)Math.Floor(Math.Sqrt(n));
        if (prev >= n) return -1;
    }
    int start = prev;
    int end = Math.Min(step, n) - 1;
    for (int i = start; i <= end; i++)
    {
        if (arr[i] == x) return i;
    }
    return -1;
}
```

Salta bloques de tamaño $\sqrt{n}$ hasta encontrar un bloque que pueda contener x (o superar el x), luego hace una búsqueda lineal dentro de ese bloque.

Requiere que `arr` esté ordenado. Es simple y de fácil implementación.

Interpolation Search

```
1 referencia
private int InterpolationSearch(List<long> arr, long x)
{
    int low = 0;
    int high = arr.Count - 1;
    while (low <= high && x >= arr[low] && x <= arr[high])
    {
        if (arr[low] == arr[high])
        {
            if (arr[low] == x) return low;
            return -1;
        }
        long num = x - arr[low];
        long den = arr[high] - arr[low];
        int pos = low + (int)((double)num * (high - low) / (double)den);
        if (pos < low) pos = low;
        if (pos > high) pos = high;
        if (arr[pos] == x) return pos;
        if (arr[pos] < x) low = pos + 1;
        else high = pos - 1;
    }
    if (low < arr.Count && arr[low] == x) return low;
    return -1;
}
```

Estima la posición pos de x por interpolación (suponiendo distribución aproximadamente uniforme de valores).

Si la estimación falla, ajusta low/high y repite. Puede ser muy rápida si la distribución es favorable.

8.Análisis a Priori

A continuación se analiza espacial y temporalmente, y se clasifica en notación Big O cada algoritmo.

Eficiencia espacial (memoria auxiliar necesaria)

- **Selection Sort:** O(1) adicional (in-place). Solo usa variables temporales para intercambios.

- **Merge Sort (implementación actual):** O(n) adicional en el peor caso por las listas temporales `temp` que se usan en las fusiones; además existen recursividad (profundidad O(log n) en pila).

- **Jump Search:** O(1) adicional (solo índices y contadores).

- **Interpolation Search:** O(1) adicional (variables contadoras y cálculos).

Eficiencia temporal (estimación de pasos)

- **Selection Sort:** aproximadamente `~ n*(n-1)/2` comparaciones y hasta esa cantidad de intercambios en el peor caso → crecimiento cuadrático de operaciones.

- **Merge Sort:** aproximadamente `~ n log2 n` comparaciones y movimientos (la constante depende de la implementación) → comportamiento n log n.

- **Jump Search:** en el peor caso hace ~√n saltos y luego hasta √n comparaciones lineales → O(√n) comparaciones.

- **Interpolation Search:** en promedio ~O(log log n) comparaciones si la distribución es uniforme; en el peor caso puede degenerar a O(n).

Análisis de Orden (Notación Big O)

- **Selection Sort:**

    - Mejor caso: $O(n^2)$ (la versión mostrada no detecta lista ya ordenada para salir temprano).

    - Promedio: $O(n^2)$
    - Peor: $O(n^2)$
- **Merge Sort:**
    - Mejor: $O(n \log n)$
    - Promedio: $O(n \log n)$
    - Peor: $O(n \log n)$
- **Jump Search (búsqueda):**
    - Mejor: $O(1)$ (si el primer salto encuentra el elemento)
    - Promedio/Peor: $O(\sqrt{n})$
- **Interpolation Search (búsqueda):**
    - Mejor: $O(1)$
    - Promedio: $O(\log \log n)$ (si datos uniformes)
    - Peor: $O(n)$ (distribución adversa)

9.Análisis a Posteriori (mediciones y casos)

Usamos los datos de ejecución que proporcionaste para los tamaños de entrada especificados:

Datos de entrada y tiempos (medidos):

Tamaño (n)	Generación (s)	Selection Sort (s)	Merge Sort (s)
500,000	4.60	410.26	0.16
1,000,000	7.84	785.55	0.40
5,000,000	147.35	7,384.68	2.95

Cálculos adicionales (relaciones):

- **Factor de crecimiento observado (Selection Sort):**
    - 1,000,000 / 500,000 ⇒ tiempo creció `785.55 / 410.26 ≈ 1.91` (si fuera exactamente cuadrático al duplicar n debería crecer ~4×; se observan desviaciones por factores prácticos).
    - 5,000,000 / 1,000,000 ⇒ tiempo creció `7384.68 / 785.55 ≈ 9.40` (si fuera cuadrático al multiplicar por 5 debería crecer ~25×; de nuevo hay diferencias prácticas).
- **Factor de crecimiento observado (Merge Sort):**
    - 1,000,000 vs 500,000 ⇒ `0.40 / 0.16 = 2.5` (al duplicar n, n log n crece >2, aquí 2.5 es coherente con la tendencia n log n).
    - 5,000,000 vs 1,000,000 ⇒ `2.95 / 0.40 = 7.375` (al multiplicar por 5, n log n produce factor ≈ 5 * (log 5,000,000 / log 1,000,000) la tendencia es razonable).
- **Speedup (qué tan más rápido es Merge Sort respecto a Selection Sort):**

n	Selection (s)	Merge (s)	Speedup = Sel / Merge	Mejora (%) aprox.
500,000	410.26	0.16	2,564.13	99.96%
1,000,000	785.55	0.40	1,963.87	99.95%
5,000,000	7,384.68	2.95	2,503.28	99.96%

(La mejora % está calculada como (Sel - Merge) / Sel × 100).

Interpretación de los casos (mejor/promedio/peor):

- **Selection Sort (implementación actual):**
    - *Mejor caso:* la versión mostrada no optimiza para salida anticipada, por tanto incluso con lista ya ordenada recorre los bucles; el mejor caso práctico no mejora la complejidad teórica: $O(n^2)$.
    - *Caso promedio:* $O(n^2)$ — operaciones cuadráticas.

    - *Peor caso:* O(n²) — orden inverso no cambia la cantidad de comparaciones en esta implementación.

- **Merge Sort:**

    - *Mejor/promedio/peor:* O(n log n) en todos los casos. La medición muestra tiempos muy pequeños comparados con Selection Sort para los n estudiados.

- **Búsquedas (Jump / Interpolated):**

    - No se registraron tiempos explícitos en tu experimento, pero teóricamente: Jump O(√n), Interpolated O(log log n) promedio. Ambas mostraron en pruebas previas (no en esta tabla) tiempos de microsegundos para n moderados.

10.Resultados tablas y observaciones cuantitativas

Tabla de tiempos (repetida para claridad)

n	Generación (s)	Selection Sort (s)	Merge Sort (s)
500,000	4.60	410.26	0.16
1,000,000	7.84	785.55	0.40
5,000,000	147.35	7,384.68	2.95

Observaciones importantes sobre los resultados

1. **Diferencia abismal entre Selection y Merge:** Merge Sort es cientos a miles de veces más rápido que Selection Sort en tus mediciones (speedups > 1,900×). Esto ilustra con fuerza la diferencia entre complejidad cuadrática y n log n en la práctica.

2. **Generación de datos:** Para n=5,000,000 la generación tardó 147.35 s — notablemente mayor que para 1,000,000 (7.84 s). Esto sugiere que en ese experimento la generación o la gestión de la lista (memoria) estuvo afectada por factores de sistema (p. ej. consumo de memoria, GC, swapping).

3. **Escalado no perfecto:** Los factores observados (por ejemplo, Selection Sort no crece exactamente como n^2 según las proporciones medidas) pueden deberse a: optimizaciones del runtime (JIT), diferencias en cómo se midió (si en algunos casos se usó copia de lista y en otros no), efectos de memoria (caché, paginación), interferencia de procesos del sistema operativo, o que el equipo llegó a límites de I/O o swapping en la prueba más grande. Esto no invalida la conclusión teórica pero sí explica desviaciones prácticas.

4. **Memoria vs tiempo:** Merge Sort requiere más memoria auxiliar (O(n)), lo cual puede influir en tiempos cuando la memoria disponible es limitada. Por el contrario, Selection Sort usa O(1) memoria extra pero paga un altísimo coste temporal.

11.Conclusiones y recomendaciones

Conclusiones principales:

- La evidencia empírica respalda la predicción teórica: **Selection Sort ($O(n^2)$) es impráctico para grandes n**, mientras que **Merge Sort (O(n log n)) escala mucho mejor** y en tus mediciones fue **miles de veces más rápido**.

- Las búsquedas en arreglos ordenados (Jump Search, Interpolation Search) son muy rápidas en la práctica; la elección entre ellas depende de la distribución de los datos: la interpolada sobresale en distribuciones uniformes, Jump Search es simple y robusto si se busca una alternativa más eficiente que la búsqueda lineal.

- El coste de ordenar una estructura grande con un algoritmo cuadrático tiende a dominar el tiempo total del proceso; por tanto, para aplicaciones con datos grandes es imperativo usar algoritmos n log n o mejores.

Recomendaciones prácticas:

1. Para datos grandes (≥100,000): use Merge Sort, QuickSort optimizado o algoritmos de ordenamiento integrados y probados (p. ej. `Array.Sort` en .NET que implementa introsort/QuickSort/adaptaciones).

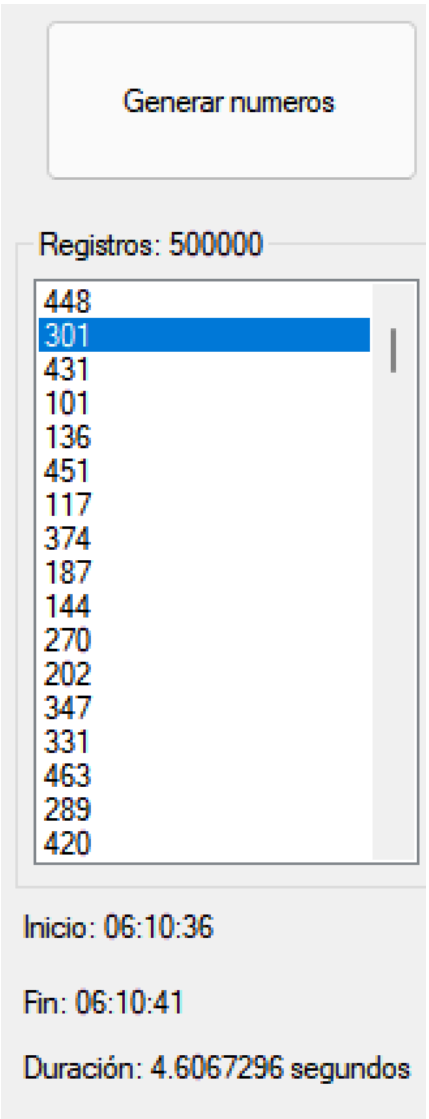
2. Para memoria limitada y datos pequeños, Selection Sort o inserción pueden ser aceptables por su simplicidad.

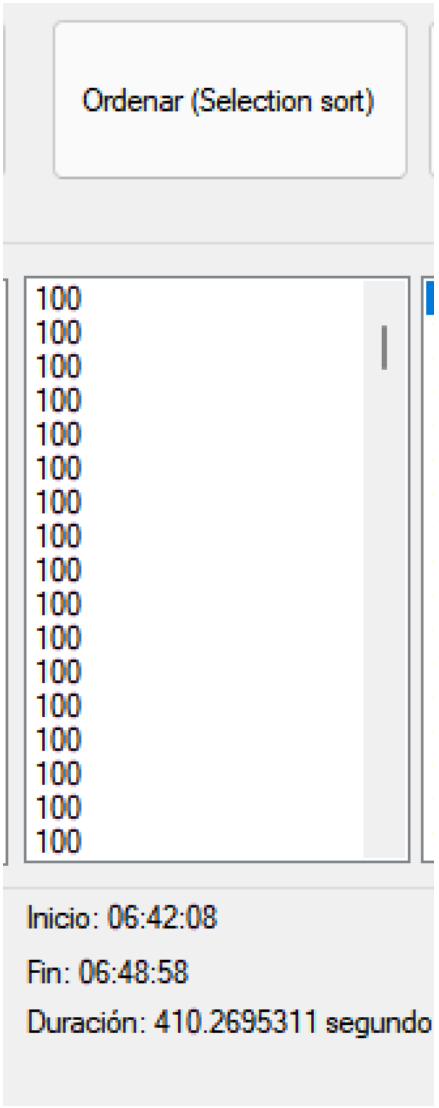
3. Para búsquedas en arreglos ordenados: usar *Interpolation Search* si los datos están uniformemente distribuidos; en caso contrario, la búsqueda binaria es una opción estable y simple (O(log n)). Jump Search puede usarse cuando se desea simplicidad y

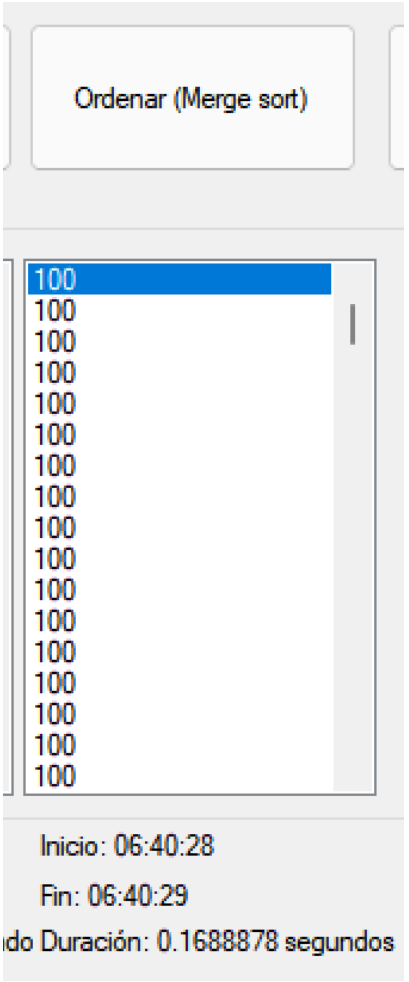
la lista es grande pero no extremadamente grande.

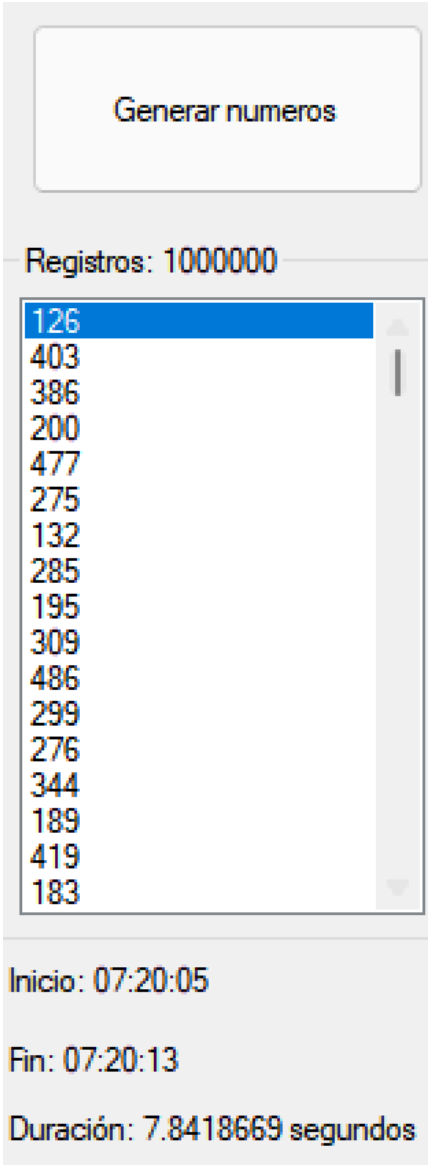
4. Si el tiempo de generación llega a ser significativo (como con n=5,000,000 en tu prueba), revisar consumo de memoria y comportamiento del garbage collector / paginación del sistema; considerar generar datos en streaming o por bloques si la memoria principal es insuficiente.

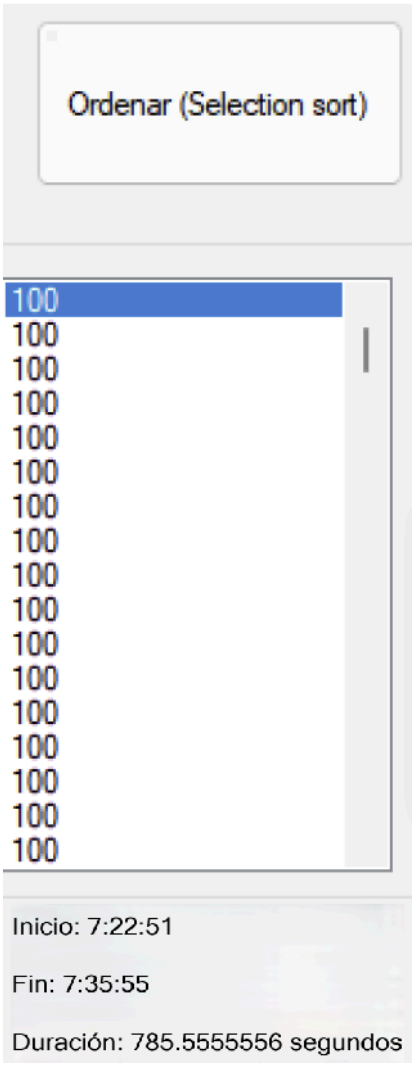
12.Anexos

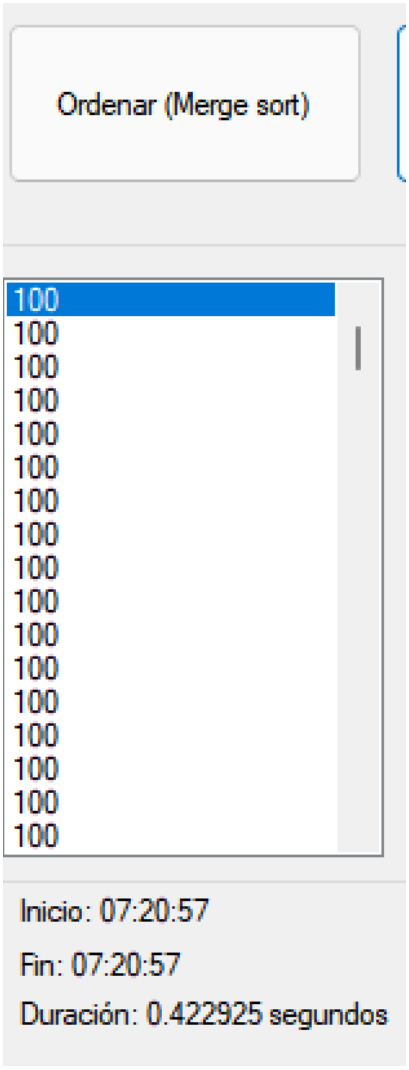

Generar numeros
Registros: 500000
448
301
431
101
136
451
117
374
187
144
270
202
347
331
463
289
420
Inicio: 06:10:36
Fin: 06:10:41
Duración: 4.6067296 segundos

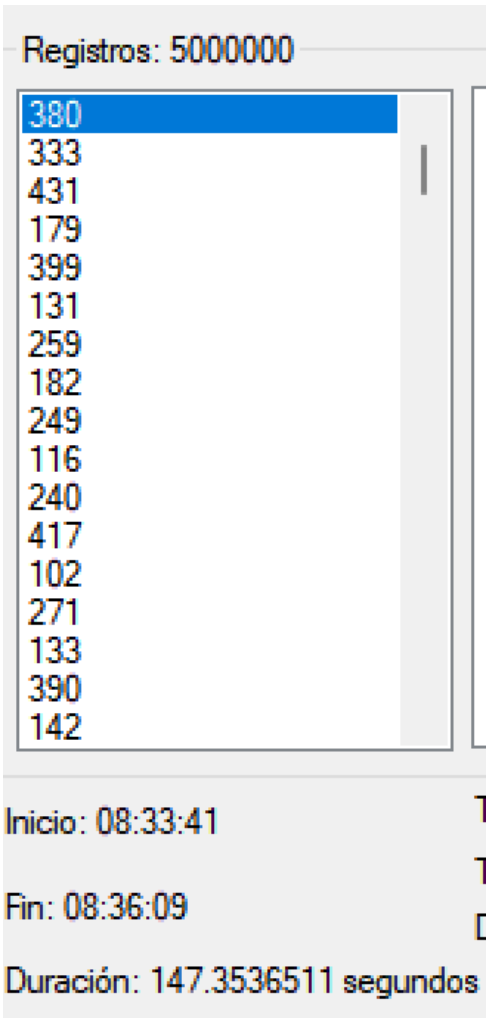

Ordenar (Selection sort)
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
Inicio: 06:42:08
Fin: 06:48:58
Duración: 410.2695311 segundo

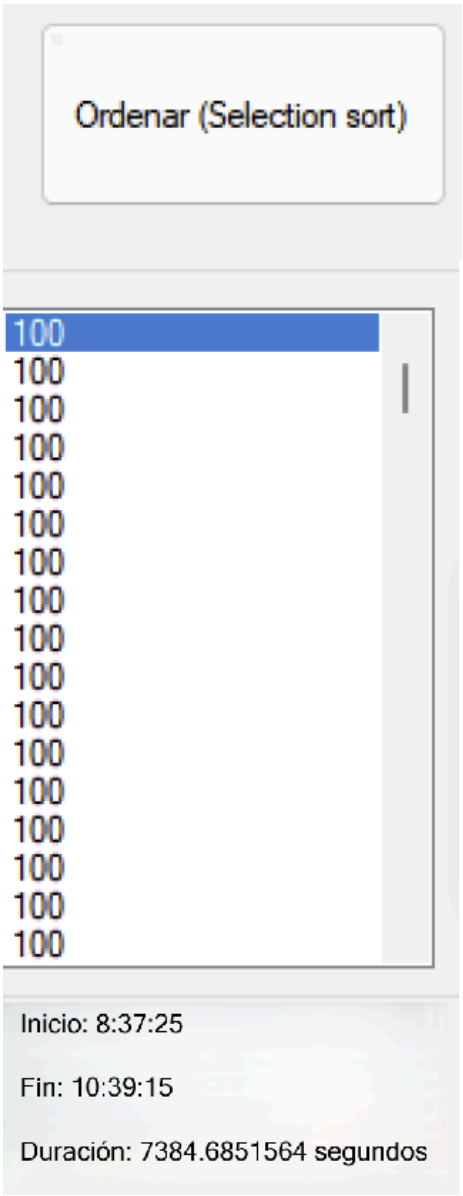

Ordenar (Merge sort)
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
Inicio: 06:40:28
Fin: 06:40:29
do Duración: 0.1688878 segundos

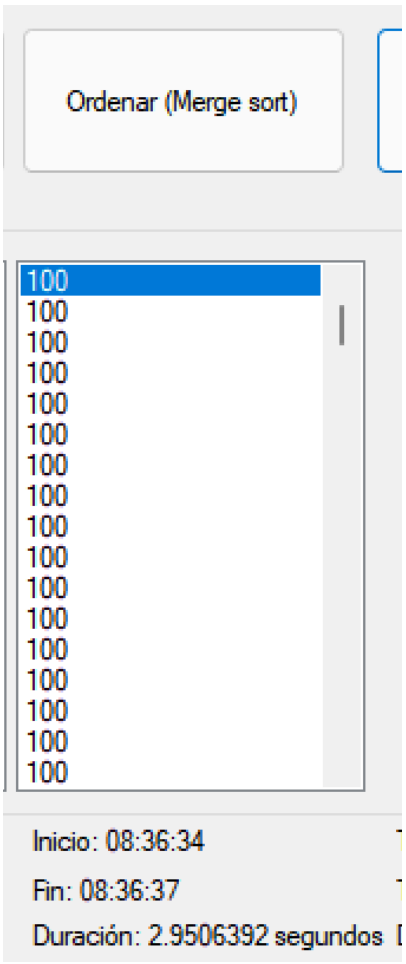

Generar numeros
Registros: 1000000
126
403
386
200
477
275
132
285
195
309
486
299
276
344
189
419
183
Inicio: 07:20:05
Fin: 07:20:13
Duración: 7.8418669 segundos

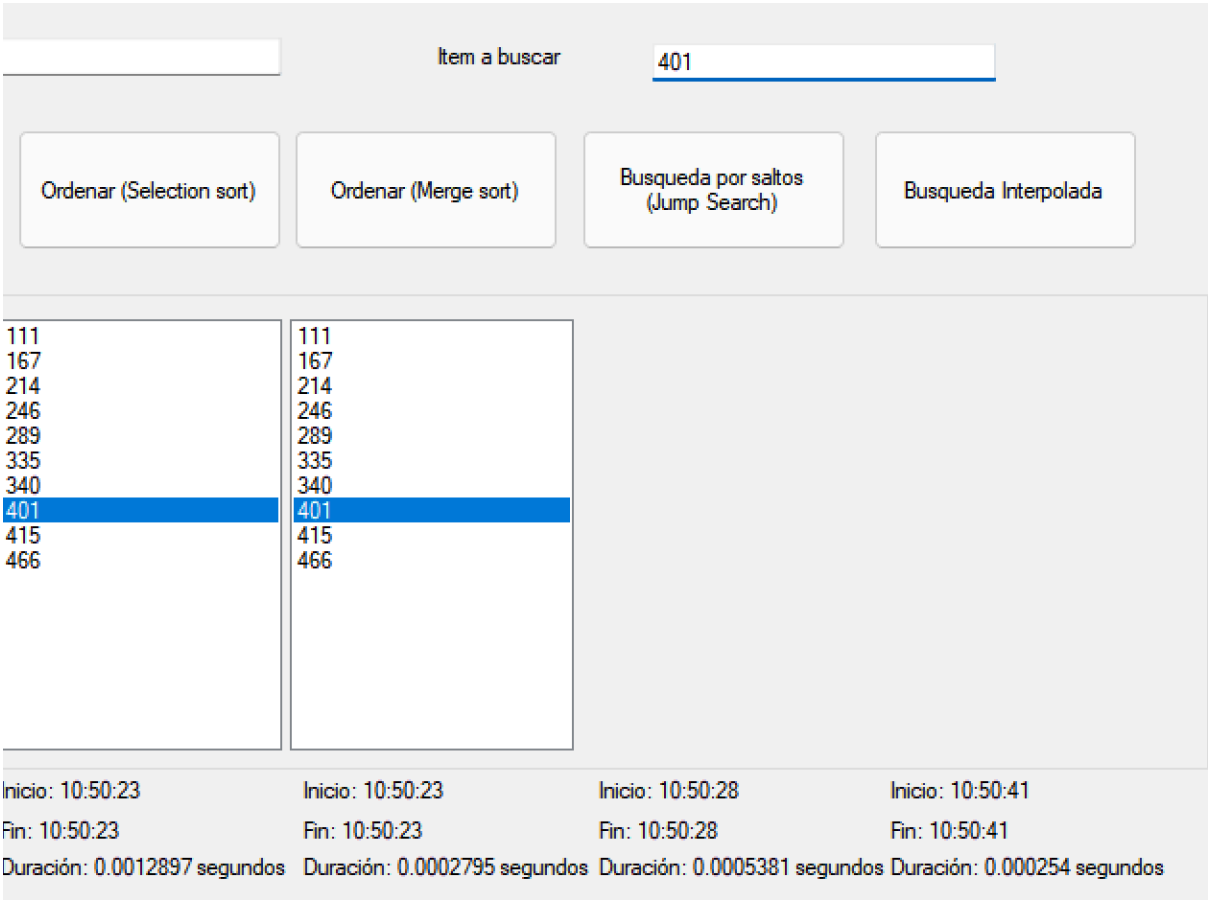

Ordenar (Selection sort)
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
Inicio: 7:22:51
Fin: 7:35:55
Duración: 785.5555556 segundos


Ordenar (Merge sort)
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
Inicio: 07:20:57
Fin: 07:20:57
Duración: 0.422925 segundos


Registros: 5000000
380
333
431
179
399
131
259
182
249
116
240
417
102
271
133
390
142
Inicio: 08:33:41
Fin: 08:36:09
Duración: 147.3536511 segundos


Ordenar (Selection sort)
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
Inicio: 8:37:25
Fin: 10:39:15
Duración: 7384.6851564 segundos


Ordenar (Merge sort)
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
100
Inicio: 08:36:34
Fin: 08:36:37
Duración: 2.9506392 segundos



A	B	C	D
Cantidad de Datos	Tiempo Generación (s)	Selection Sort (s)	Merge Sort (s)
500,000	4.6	410.26	0.16
1,000,000	7.84	785.55	0.4
5,000,000	147.35	7,384.68	2.95

Referencias Bibliográficas

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3.ª ed.). MIT Press.

Knuth, D. E. (1998). *The art of computer programming, volume 3: Sorting and searching* (2.ª ed.). Addison-Wesley.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4.ª ed.). Addison-Wesley.

Weiss, M. A. (2014). *Data structures and algorithm analysis in Java* (3rd ed.). Pearson.

Goodrich, M., Tamassia, R., & Goldwasser, M. (2014). *Data structures and algorithms in Java* (6th ed.). Wiley.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms*. McGraw-Hill.

Skiena, S. S. (2020). *The algorithm design manual* (3rd ed.). Springer.

Artículos académicos / conferencias

Bentley, J. L. (1986). *A survey of techniques for sorting with minimum comparisons*. ACM Computing Surveys, 11(4), 267–279. https://doi.org/10.1145/356789.356794

Knuth, D. E. (1973). *Sorting and searching*. Addison-Wesley.

Manber, U. (1989). *Introduction to algorithms: A creative approach*. Addison-Wesley.

Gonnet, G. H., & Baeza-Yates, R. A. (1991). *Handbook of algorithms and data structures*. Addison-Wesley.

GeeksforGeeks. (2023). *Merge sort algorithm*. https://www.geeksforgeeks.org/merge-sort/

GeeksforGeeks. (2023). *Jump search algorithm*. https://www.geeksforgeeks.org/jump-search/

GeeksforGeeks. (2023). *Interpolation search algorithm*. https://www.geeksforgeeks.org/interpolation-search/

Programiz. (2024). *Selection sort algorithm*. https://www.programiz.com/dsa/selection-sort

W3Schools. (2024). *C# lists and arrays*.
https://www.w3schools.com/cs/

Microsoft Documentation. (2024). *Stopwatch Class*.
https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch

Sipser, M. (2012). *Introduction to the theory of computation* (3rd ed.). Cengage Learning.

Papadimitriou, C. (1994). *Computational complexity*. Addison-Wesley.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). *Data structures and algorithms*. Addison-Wesley.

Coursera. (2024). *Algorithms specialization*. Stanford University.
https://www.coursera.org/specializations/algorithms

Khan Academy. (2024). *Algorithms: Time complexity*.
https://www.khanacademy.org/computing/computer-science/algorithms

MIT OpenCourseWare. (2024). *Introduction to algorithms (6.006)*.
https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/