Actividad experimental: comparación empírica entre Merge Sort e Insertion Sort

Lic. Alex Maglio Neyra Herrada

(2025 MAES DIS) – Maestría en Ciencias de la Computación
Algoritmos y Estructura de Datos GA

Se utilizó Inteligencia Artificial con fines de apoyo, revisión y retroalimentación. La implementación, el análisis y el contenido fueron realizados por el autor.

1. Introducción

En el análisis de algoritmos, se estudia el comportamiento asintótico del tiempo de ejecución en función del tamaño de la entrada [2]. En particular, Insertion Sort presenta una complejidad temporal de $O(n^2)$ en el peor caso, mientras que Merge Sort ejecuta en $O(n \log n)$ en cualquier caso.

Desde un punto de vista teórico, esto implica que para valores suficientemente grandes de n, Merge Sort debería superar en velocidad de ejecución a Insertion Sort. Sin embargo, en implementaciones reales intervienen costos adicionales que no se contemplan en la notación asintótica.

Las diapositivas del curso del MIT [3] indican que, en la práctica, Merge Sort comienza a ser más rápido que Insertion Sort para tamaños de entrada mayores a aproximadamente $n > 30$. Este valor, denominado N_0, no es una ley, sino que depende de situaciones cómo: el entorno de ejecución, de la implementación concreta y principalmente de las características de los datos de entrada.

El objetivo de este trabajo es determinar empíricamente el valor de N_0 mediante un experimento controlado, comparando los tiempos de ejecución de Insertion Sort y Merge Sort bajo distintos escenarios de ordenamiento de los datos. Para ello, se implementaron ambos algoritmos en C++ y se diseñó un protocolo de medición reproducible que permite reducir el ruido y obtener estimaciones que intentan mitigar el ruido de la medición.

2. Implementación

Ambos algoritmos, Insertion Sort listado 1 y Merge Sort listado 2, fueron implementados en el lenguaje C++ con el objetivo de obtener el máximo rendimiento posible y un control explícito sobre el uso de memoria. La compilación se realizó utilizando la opción de optimización `-O2`, lo que permite al compilador aplicar transformaciones que reducen el tiempo de ejecución sin alterar la semántica del programa.

La generación de los datos de entrada se realizó mediante un generador pseudoaleatorio con semilla fija, de modo que cada ejecución del experimento utiliza exactamente los mismos arreglos de entrada. Esto garantiza que las comparaciones entre algoritmos se realizan sobre instancias idénticas del problema, eliminando una posible fuente de variabilidad y asegurando replicabilidad.

Con el fin de estudiar el impacto del grado de ordenamiento de los datos, se consideraron cuatro escenarios distintos: arreglos completamente ordenados, casi ordenados, aleatorios y en orden inverso. Esta elección se fundamenta en que el desempeño de Insertion Sort depende fuertemente del número de inversiones presentes en el arreglo, mientras que Merge Sort mantiene una complejidad $O(n \log n)$ independientemente de la distribución de los datos.

Para reducir el efecto del ruido producido por el sistema operativo, la jerarquía de memoria RAM o cache y otros factores externos, cada medición de tiempo fue repetida múltiples veces para cada tamaño de entrada y escenario, calculándose posteriormente el promedio de los tiempos obtenidos.

```cpp
void InsertionSort::insertion_sort(std::vector<int> &a)
{
    const size_t n = a.size();
    for (size_t i = 1; i < n; ++i)
    {
        int key = a[i];
        size_t j = i;
        while (j > 0 && a[j - 1] > key)
        {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = key;
    }
}
```

Listing 1: Fragmento de código sobre la implementación del algoritmo Insertion Sort en C++.

```cpp
void MergeSort::merge_sort_rec(
    std::vector<int> &a, std::vector<int> &buf,
    size_t left, size_t right)
{
    const size_t len = right - left;
    if (len <= 1)
        return;
    const size_t mid = left + len / 2;
    merge_sort_rec(a, buf, left, mid);
    merge_sort_rec(a, buf, mid, right);
    merge_ranges(a, buf, left, mid, right);
}
```

Listing 2: Fragmento de código sobre la implementación del algoritmo Merge Sort en C++.

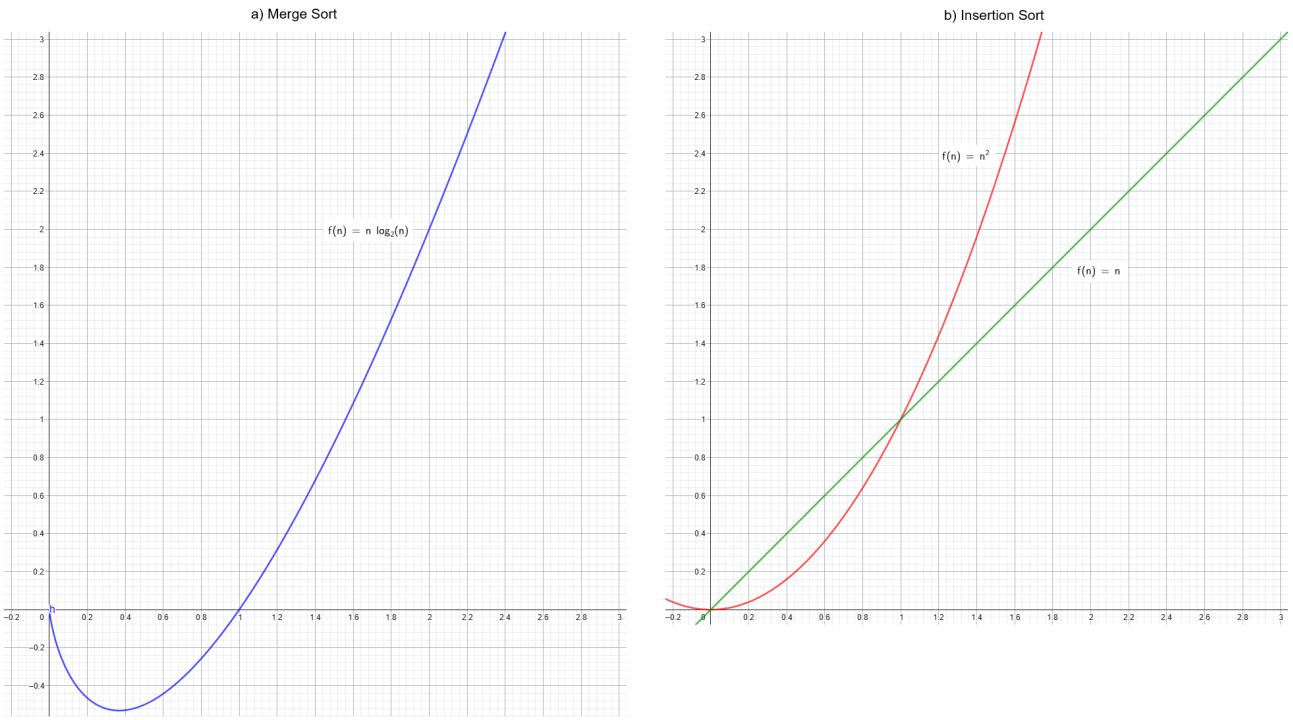

Figura 1: a) Complejidades $O(n)$ y $O(n^2)$ de Insertion Sort y su intersección. b) Complejidad $O(n \log n)$ de Merge Sort.

De esta forma se obtiene una estimación que de hasta cierto punto mitiga el ruido en la medición y representa el tiempo de ejecución real de cada algoritmo.

3. Resultados

Con el fin de interpretar adecuadamente los resultados empíricos, primero se presentan las complejidades teóricas de ambos algoritmos. La Figura figura 1 sección b) muestra las funciones $O(n)$ y $O(n^2)$ correspondientes al mejor y peor caso de Insertion Sort, respectivamente, y su intersección, que ilustra cómo el comportamiento del algoritmo depende fuertemente del número de inversiones presentes en los datos. Por su parte, la Figura figura 1 sección a) muestra la complejidad $O(n \log n)$ de Merge Sort, que se mantiene invariante frente al grado de ordenamiento de la entrada.

La Figura figura 2 sección b) presenta los tiempos de ejecución medidos para el escenario de arreglos casi ordenados. En este caso, se observa que el punto de cruce entre ambos algoritmos ocurre aproximadamente en $N_0 \approx 4300$. Este valor elevado se explica porque el número de inversiones en este escenario es muy bajo (alrededor del 1 % de n), y para tamaños pequeños ($n < 100$) incluso se mantiene cercano a una única inversión. Bajo estas condiciones, Insertion Sort se comporta de manera cercana a su mejor caso $O(n)$, lo que retrasa considerablemente el momento en que Merge Sort comienza a ser más eficiente.

En el escenario completamente aleatorio, mostrado en la Figura figura 2 sección c), el valor empírico observado es $N_0 \approx 58$. Este resultado es consistente con un caso promedio, donde el número de inversiones crece, haciendo que Insertion Sort se aproxime a su complejidad cuadrática mientras que Merge Sort mantiene su crecimiento $O(n \log n)$. Como consecuencia, el punto de cruce ocurre para valores de n mucho menores que en el caso casi ordenado.

La Figura figura 2 sección d) corresponde al escenario de arreglos ordenados en forma inversa, que representa el peor caso para Insertion Sort. Aquí se observa que el valor de cruce es aproximadamente $N_0 \approx 30$ apartado 3, en concordancia con lo sugerido en las diapositivas del curso del MIT [3]. En este caso, Insertion Sort incurre en su complejidad $O(n^2)$ completa, mientras que Merge Sort conserva su comportamiento $O(n \log n)$, lo que hace que este último supere rápidamente al primero.

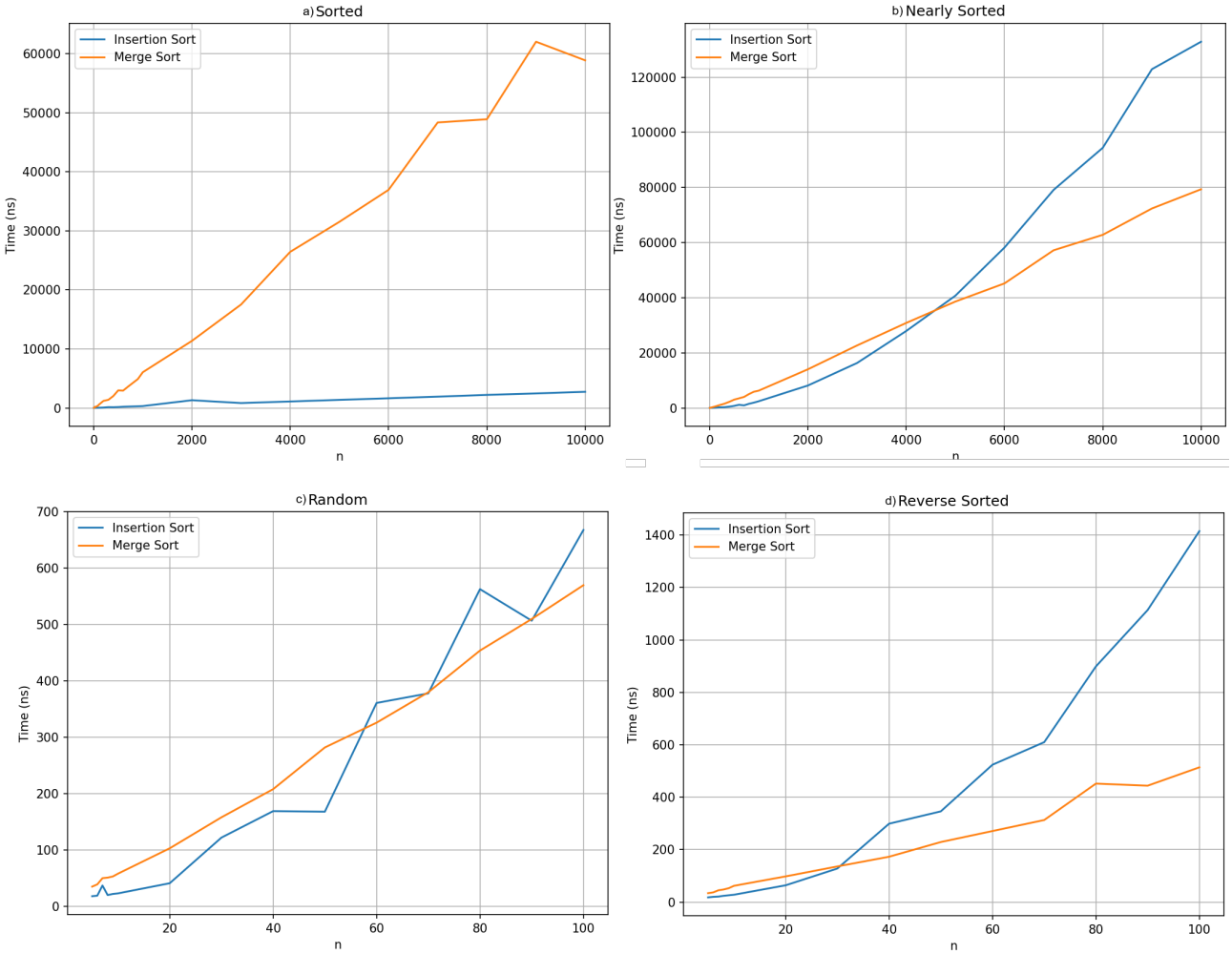

Figura 2: a) Tiempo de ejecución vs. n para el escenario completamente ordenado. b) Tiempo de ejecución vs. n para el escenario casi ordenado. c) Tiempo de ejecución vs. n para el escenario aleatorio. d) Tiempo de ejecución vs. n para el escenario en orden inverso.

n	Insertion Mean	Merge Mean	Ratio	Faster
5	18	34	0.552589	Insertion Sort
6	20	37	0.541965	Insertion Sort
7	21	45	0.475483	Insertion Sort
8	24	48	0.496459	Insertion Sort
9	26	53	0.498368	Insertion Sort
10	28	62	0.448556	Insertion Sort
20	64	98	0.660918	Insertion Sort
30	128	136	0.946778	Insertion Sort
40	299	173	1.72833	Merge Sort
50	346	229	1.51011	Merge Sort
60	524	271	1.93302	Merge Sort
70	610	313	1.94841	Merge Sort
80	899	452	1.98822	Merge Sort
90	1114	444	2.50364	Merge Sort
100	1415	514	2.75183	Merge Sort

Cuadro 1: Fragmento de los resultados en el escenario donde los datos están ordenados de forma invertida.

Finalmente, la Figura figura 2 sección a) muestra el escenario de arreglos completamente ordenados. En este caso, Insertion Sort opera en su mejor caso $O(n)$, mientras que Merge Sort continúa ejecutándose en $O(n \log n)$. Como resultado, no se observa ningún valor finito de N_0: Insertion Sort supera sistemáticamente a Merge Sort para todos los tamaños de entrada considerados.

En conjunto, estos resultados confirman que el valor de N_0 no es una constante universal, sino una magnitud que depende de manera crítica del grado de ordenamiento de los datos, lo que concuerda con la diferencia fundamental entre la dependencia de Insertion Sort en el número de inversiones y la complejidad estable de Merge Sort.

4. Conclusiones

En este trabajo se realizó una evaluación empírica del punto de cruce N_0 entre Insertion Sort y Merge Sort bajo distintos escenarios de ordenamiento de los datos. Los resultados muestran que el valor de N_0 no es una constante universal, sino que depende de manera crítica de la estructura de la entrada y, en particular, del número de inversiones presentes en el arreglo.

En el escenario de peor caso para Insertion Sort (arreglos en orden inverso), se obtuvo un valor $N_0 \approx 30$, lo cual coincide con la estimación presentada en las diapositivas del MIT [3]. Sin embargo, al considerar escenarios más favorables para Insertion Sort, como arreglos casi ordenados o completamente ordenados, el valor de N_0 aumenta significativamente o incluso deja de existir, reflejando el comportamiento cercano a $O(n)$ de dicho algoritmo en estos casos. Por el contrario, Merge Sort mantiene su complejidad $O(n \log n)$ independientemente de la distribución de los datos.

Estos resultados ponen de manifiesto una limitación importante del análisis puramente asintótico: la notación Big-O describe correctamente el crecimiento de los algoritmos, más no su costo computacional real, no captura las constantes ocultas ni la influencia de las características específicas de la entrada.

Los resultados deben interpretarse bajo el modelo RAM, en el cual las operaciones básicas y los accesos a memoria tienen costo uniforme. En este contexto, Insertion Sort se beneficia de ser un algoritmo que trabaja directamente con los datos, mientras que Merge Sort tiene sobrecostos por el uso de memoria auxiliar, lo que explica las diferencias observadas en el valor empírico de N_0.

Esta observación explica la motivación detrás de algoritmos híbridos como *TimSort* [1], utilizados en bibliotecas estándar modernas. TimSort combina Insertion Sort y Merge Sort con el objetivo de explotar el hecho de que Insertion Sort es extremadamente eficiente en secuencias pequeñas o casi ordenadas, mientras que Merge Sort garantiza un buen comportamiento en el caso general. De este modo, TimSort logra un rendimiento que se aproxima al mejor de ambos mundos, adaptándose dinámicamente a la estructura de los datos.

En conclusión, el experimento confirma que la afirmación del MIT [3] sobre $N_0 \approx 30$ es válida bajo condiciones de entrada desfavorables para Insertion Sort, pero que en escenarios más estructurados dicho punto de cruce puede desplazarse de manera significativa.

Referencias

[1] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort. *arXiv preprint arXiv:1805.08612*, 2018.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[3] MIT OpenCourseWare. Lecture 3: Insertion sort and merge sort. `https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/resources/lecture-3-insertion-sort-merge-sort/`, 2011.