UNIVERSIDAD MICHOCANA DE SAN NICOLAS DE HIDALGO

FACULTAD DE INGENIERIA ELECTRICA

ANALISIS DE ALGORITMOS

SECCION 503

PROYECTO: ALGORITMOS DE ORDENAMIENTO

PROFESORA: DRA. VIOLETA MEDINA RIOS

INTEGRANTES:
IVAN CALDERON GARCIA (2337556E)

DIEGO CHAVEZ FERREIRA (2337534J)

LEON MAXIMILIANO GUZMAN LEYVA (1926162A)

1. Objetivo
   "El objetivo de este proyecto fue diseñar, implementar y evaluar un banco de pruebas para comparar el rendimiento y la estabilidad de 4 algoritmos de ordenamiento en C, representando diferentes clases de complejidad. Los algoritmos seleccionados fueron: Insertion Sort ($O(n^2)$), Merge Sort ($O(n\ log\ n)$), Counting Sort ($O(n)$), e Introsort (Híbrido)."
2. Metodologia:
   1. Ambiente de prueba
      Equipo I:
      CPU: AMD Ryzen 3 3250U with Radeon Graphics (4) @ 2.595GHz
      RAM: 16GB
      SISTEMA OPERATIVO: Ubuntu 22.04.5 LTS on Windows 10 x86_64
      GPU: AMD RADEON (TM) Graphics
      COMPILADOR: GCC
      Equipo II:

      CPU: AMD Ryzen 5 3500U with Radeon V

      RAM: 8GB

      SISTEMA OPERATIVO: Ubuntu 24.04.3 LTS x86_64

      GPU: AMD ATI Radeon Vega Series / Ra

      COMPILADOR: GCC

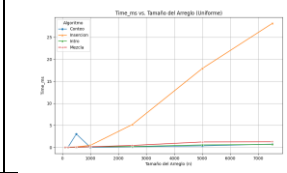
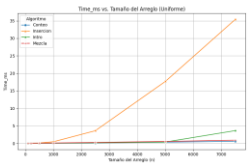
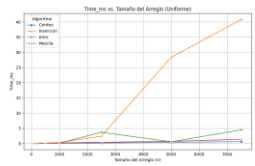
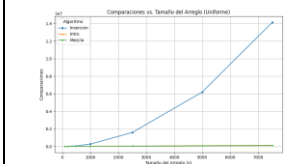
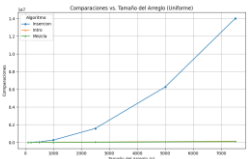
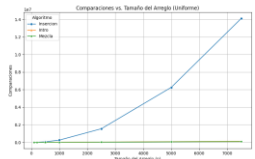
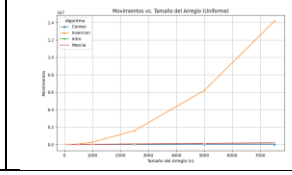
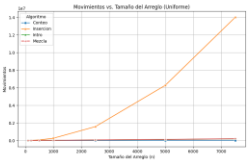
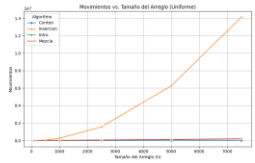
   2. Algoritmos implementados
      - Insertion Sort:
        Se recorre el arreglo desde el segundo elemento hasta el final. Para cada elemento, se compara con los anteriores y se va corriendo hacia la izquierda hasta que quede en su posición correcta. Es simple y eficiente para conjuntos pequeños o ya parcialmente ordenados.
      - Merge Sort:
        Se divide el arreglo en mitades hasta que queden subarreglos de un solo elemento. Luego, se van fusionando estas mitades ordenadamente: se comparan los elementos de cada mitad y se coloca el menor en el arreglo resultante. Es estable y siempre tiene un rendimiento de *O(n log n)*, pero usa memoria adicional.
      - Counting Sort:
        Este algoritmo no hace comparaciones. En su lugar, se cuenta la frecuencia de cada valor en el arreglo original. Luego, se reconstruye el arreglo ordenado usando esos conteos. Solo funciona cuando el rango de los valores (k) es razonable, pero puede ser muy rápido en esos casos.
      - Introsort:
        Combina lo mejor de Quicksort, Heapsort e Insertion Sort. Empieza con Quicksort, pero si la recursión se vuelve muy profunda (lo que

indicaría un caso peor), cambia a Heapsort para garantizar *O(n log n)*. En subarreglos pequeños, usa Insertion Sort por su bajo overhead.

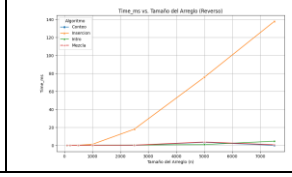
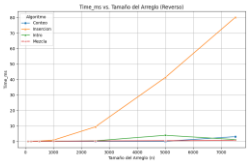
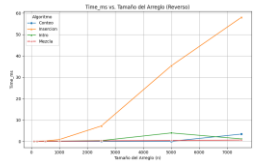
3. Datasets
   Tamaños (n): 100, 200, 500, 1000, 2500, 5000, 7500.
   Distribuciones:
   - **Uniforme (Aleatorio):** Números aleatorios.
   - **Ordenado:** Secuencia $1, 2, 3, \dots, n$
   - **Reverso:** Secuencia $n, n-1, \dots, 1$.
   - **Casi Ordenado:** Un arreglo ordenado con 5% de elementos intercambiados.
   - **Duplicados:** Números aleatorios en un rango pequeño (ej. 0 a n/10).

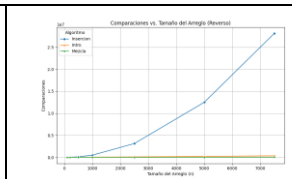
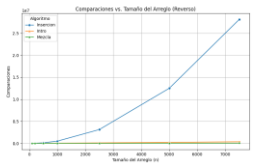
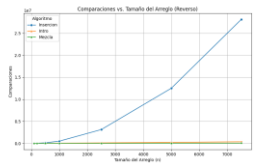
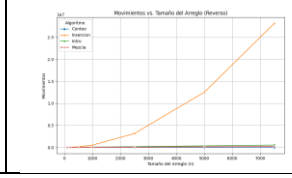
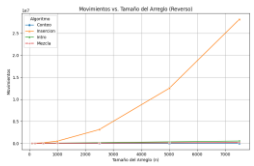
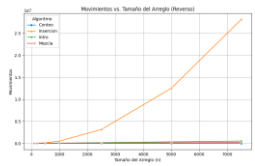
4. Métricas recopiladas
   Se corrió cada experimento 3 veces, promediando los resultados.
   Las métricas recopiladas por corrida fueron: time_ms, comparaciones, movimientos y estabilidad.

3. Resultados:
   Equipo I
   Uniforme

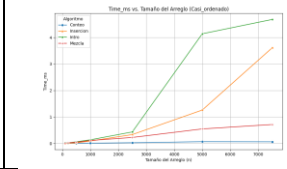
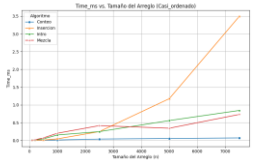
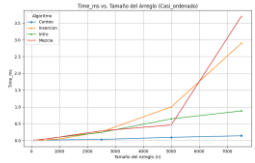
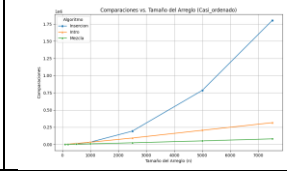
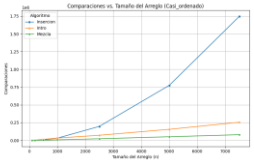
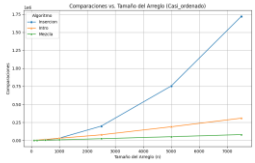
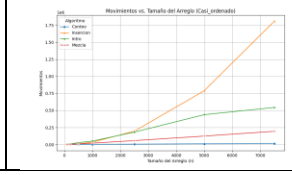
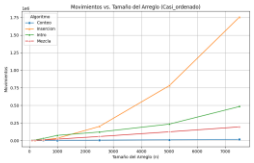
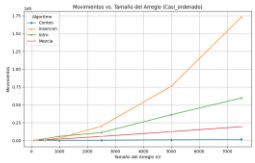
Pruebas	1	2	3
Tiempo de ejecución			
Numero de comparaciones			
Numero de movimientos			

Reverso

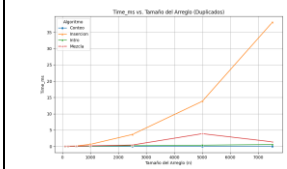
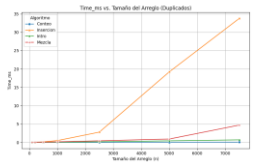
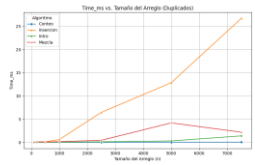
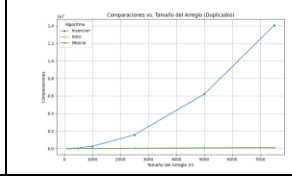
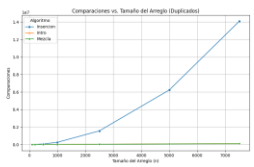
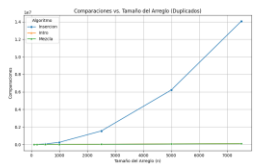
Pruebas	1	2	3
Tiempo de ejecución			

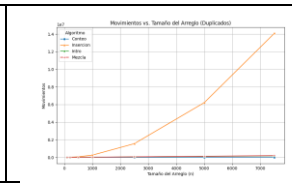
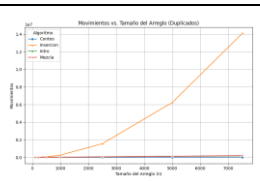
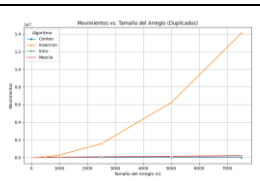
Numero de comparaciones			
Numero de movimientos			

Casi ordenado

Pruebas	1	2	3
Tiempo de ejecución			
Numero de comparaciones			
Numero de movimientos			

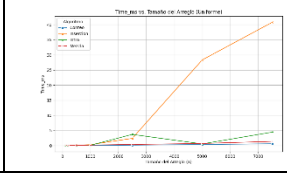
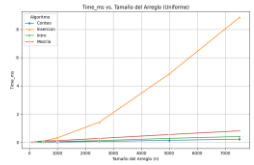
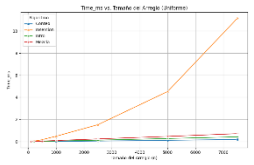
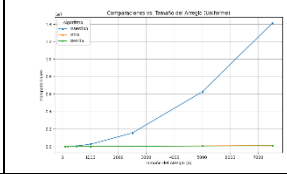
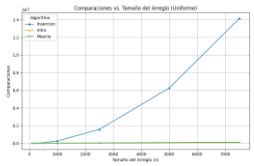
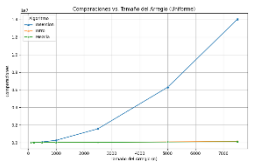
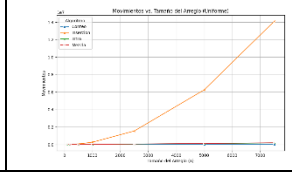
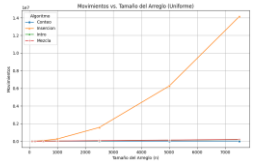
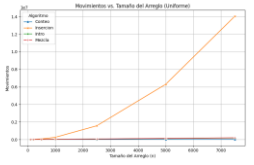
Duplicados

Pruebas	1	2	3
Tiempo de ejecución			
Numero de comparaciones			

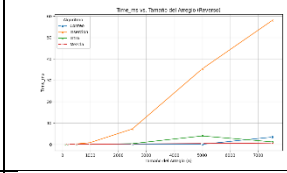
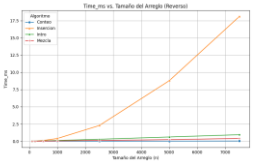
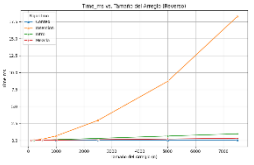
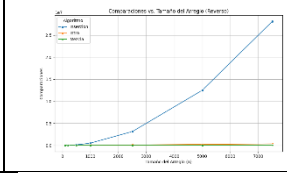
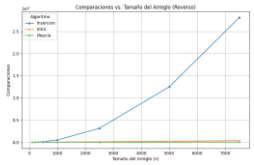
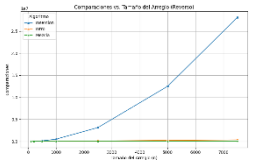
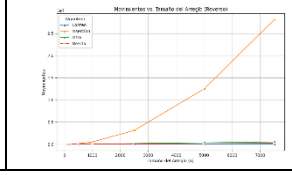
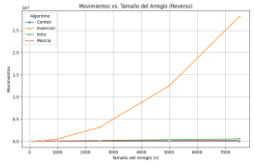
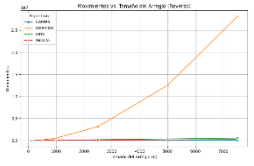
Numero de movimientos			

Equipo II

Uniforme

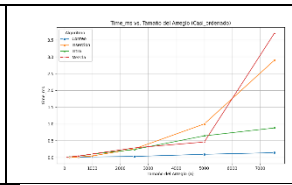
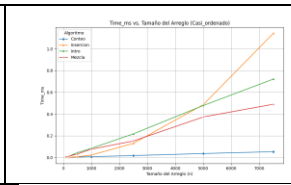
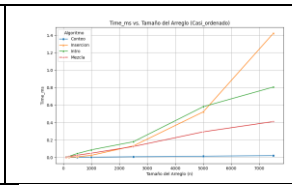
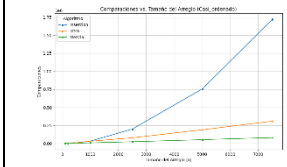
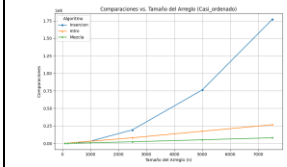
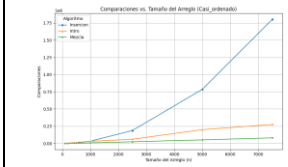
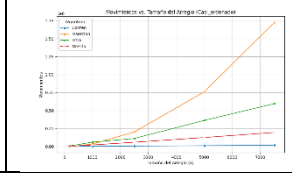
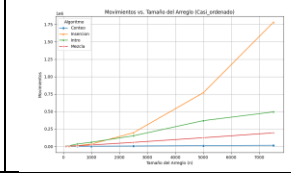
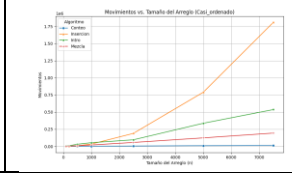
Pruebas	1	2	3
Tiempo de ejecución			
Numero de comparaciones			
Numero de movimientos			

Reverso

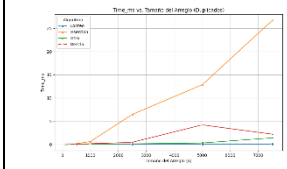
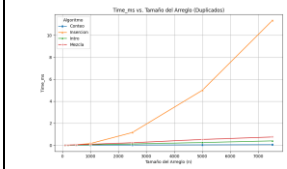
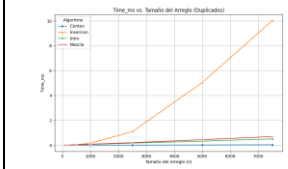
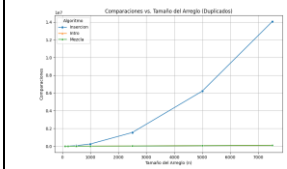
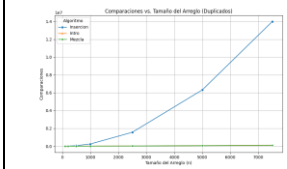
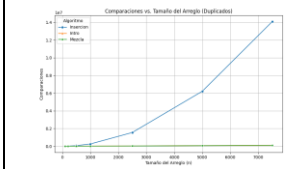
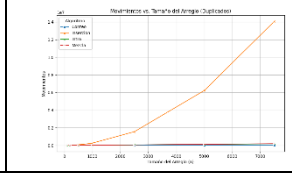
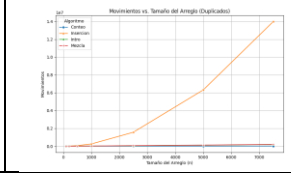
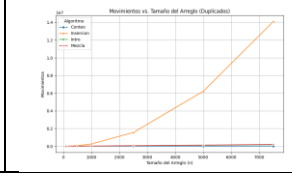
Pruebas	1	2	3
Tiempo de ejecución			
Numero de comparaciones			
Numero de movimientos			

Casi ordenado

Pruebas	1	2	3

Tiempo de ejecución			
Numero de comparaciones			
Numero de movimientos			

Duplicados

Pruebas	1	2	3
Tiempo de ejecución			
Numero de comparaciones			
Numero de movimientos			

Prueba de estabilidad

Algoritmo	¿Es estable (Teoría)?
Insertion Sort	Si
Merge Sort	Si
Counting Sort	Si
Introsort	No

4. Discusión Critica
    1. Análisis $O(n^2)$ vs $O(n \log n)$:

- En los casos "Uniforme" y "Duplicados", las gráficas de tiempo deben mostrar que Insertion Sort (cuadrática, $O(n^2)$) crece muchísimo más rápido que Merge Sort e Introsort (log-lineales, $O(n \log n)$). La línea de Insertion Sort debe "dispararse" hacia arriba.

2. El peor caso: Distribución “Reverso”:
    - Aquí es donde Introsort brilla. La gráfica de tiempo de Insertion Sort será la peor de todas ($O(n^2)$).
    - La gráfica de Merge Sort se mantendrá estable, mostrando su robustez $O(n \log n)$.
    - Lo más interesante: Si hubiéramos usado un Quicksort "simple", se habría degradado a $O(n^2)$ (igual que Insertion). Pero tu gráfica de Introsort no debería hacer eso; debería verse muy parecida a la de Merge Sort, demostrando que su mecanismo de seguridad (cambiar a Heapsort) funcionó y evitó el peor caso.
3. El mejor caso: Distribución “Casi ordenado”
    - ¡Aquí los papeles se invierten! La gráfica de Insertion Sort debería ser la más rápida de todas (incluso más que Merge e Intro). Esto se debe a que su mejor caso es $O(n)$.
    - Merge Sort e Introsort no se benefician tanto de los datos casi ordenados; su tiempo será similar al del caso uniforme.
4. El especialista: Counting Sort
    - En todas las gráficas (tiempo, movimientos), Counting Sort debe ser el ganador absoluto.
    - Debería ser una línea casi recta y muy baja, demostrando su complejidad lineal $O(n)$.
    - En la gráfica de comparaciones, no debe aparecer, ya que no las usa. Esto prueba que opera de forma fundamentalmente diferente.
5. Conclusiones más recomendaciones

Iván Calderón García:

El benchmark demostró que no existe un algoritmo universalmente "mejor", sino que la elección depende del escenario.

- **Insertion Sort ($O(n^2)$):** Confirmó ser el más lento en datos aleatorios y en reversa. Sin embargo, fue **el más rápido de todos** en el caso casi ordenado.
- **Merge Sort y Introsort ($O(n \log n)$):** Fueron los más estables y escalables. Introsort probó su valor al evitar el peor caso $O(n^2)$ en la prueba reversa, comportándose tan bien como Merge Sort.
- **Counting Sort (O(n)):** Fue, sin competencia, el más rápido en todas las pruebas de tiempo. Su rendimiento lineal se debe a que no usa comparaciones, pero su uso se limita a enteros en rangos conocidos.

Recomendaciones:

- **Para uso general:** Introsort es la mejor opción. Es rápido en el caso promedio y tiene un "seguro" que evita los peores casos, haciéndolo muy robusto.
- **Si la estabilidad es crucial:** Si necesitas que los elementos iguales mantengan su orden original, Merge Sort es la opción más segura y confiable con buen rendimiento.
- **Si los datos son enteros en un rango conocido:** No hay competencia. Counting Sort es la opción más rápida, pero solo funciona para este caso específico.
- **Si los datos ya están "casi ordenados":** Aunque suene raro, Insertion Sort es una opción excelente y súper rápida para este escenario.

Diego Chávez Ferreira:

Bueno dentro de este proyecto estuvimos haciendo diferentes pruebas relacionadas con la herramienta llamada benchmark, nos decepciono el hecho de que creíamos que encontraríamos un algoritmo ideal para todos los casos, pero no fue de esa manera.

Los algoritmos que implementamos fueron: insertion, merge y counting.

Cada uno pudimos encontrar diferentes formas de implementarlos dependiendo del caso y cada uno demostró ser eficiente de diferentes formas y haciendo una comparación general de los 3, podemos concluir que counting sort ha sido el más rápido solamente en enteros, el segundo a analizar es el insertion que como explicamos este no llega ser el algoritmo ideal en todos los casos ya que es bastante ineficiente cuando se trata de datos totalmente desordenados, pero es útil en el caso casi ordenado y por ultimo está el merge siendo este como un algoritmo bueno y estable, puede no ser el mejor o más rápido pero demostrando ser estable y escalable.

RECOMENDACION:

Bueno después de analizar nuestros algoritmos y ver cuales llegan a ser los pros y contras, en mi recomendación es que el algoritmo insertion es el más útil en general gracias a su modo de implementación, ya que este mismo posee un seguro que podría ayudarnos en casos complicados

León Maximiliano Guzmán Leyva:

El valor real de este proyecto fue ver cómo las ideas abstractas de "complejidad" se convierten en resultados tangibles y medibles. El *benchmark* no solo midió la velocidad, también nos ayudó a validar lo teórico.

Fue fascinante observar cómo el Insertion Sort literalmente explotaba en las gráficas de tiempo cuando se enfrentaba a datos aleatorios o inversos, demostrando por qué la complejidad cuadrática es un problema real. Sin embargo, fue igual de impactante verlo transformarse en el algoritmo más rápido de todos en la prueba casi ordenada, justificando por qué sigue siendo relevante.

Al mismo tiempo, Introsort demostró su diseño inteligente, mientras un Quicksort simple se habría rendido en la prueba reversa, el *benchmark* mostró que Introsort se protegió y mantuvo un rendimiento de primer nivel.

Finalmente, Counting Sort nos dio la imagen más clara, su línea casi plana en las gráficas de tiempo fue la prueba visual de que los algoritmos lineales operan en un nivel de eficiencia completamente diferente al no depender de comparaciones.

RECOMENDACIONES

- Para uso General: Usar introsort**.** Es el más rápido en promedio y tiene protección contra el peor caso (como en datos en reverso).
- Si los Datos ya están "Casi Ordenados": Usa insertion sort, fue el más rápido en este escenario específico.