Análisis de Algoritmos de Ordenamiento y Arquitectura MIPS32

Brígido Noguera
Angel Besteiro

15 de julio de 2025

Resumen

Este informe detalla un análisis comparativo de los algoritmos de ordenamiento Bubblesort e Insertion Sort en el contexto de la arquitectura MIPS32. Se exploran las diferencias entre tipos de registros, el impacto de la memoria y las estructuras de control en el rendimiento, la complejidad computacional de los algoritmos, las fases del ciclo de ejecución de instrucciones, la frecuencia de uso de tipos de instrucciones, los efectos de las instrucciones de salto, las ventajas del modelo RISC, y la utilidad de las herramientas de depuración en MARS.

Índice

1. Diferencias entre los registros $tX y $sX y su aplicación práctica

Los registros $tX son conocidos como **registros temporales** ya que no conservan sus valores a través de un llamado a otra función. Esto significa que una función no necesita preocuparse por guardar y restaurar los valores originales de estos registros, ya que su contenido no se espera que persista fuera de la función actual.

Por otro lado, los registros $sX son registros que **salvan sus valores** a través del paso de funciones. Por lo tanto, si una función modifica estos registros, tiene la responsabilidad de asegurar que los valores originales sean restablecidos antes de retornar, para no afectar a la función llamadora.

En la práctica, esta distinción es crucial para muchos algoritmos y para la organización del código. Los registros temporales ($tX) pueden ser utilizados para propósitos transitorios, como índices variables desechables o resultados intermedios dentro de una función. Los registros salvados ($sX) se usan principalmente para variables cuyo valor debe preservarse a lo largo de las llamadas a subrutinas, como variables locales persistentes o direcciones base.

2. Diferencias entre los registros $vX y $aX y su aplicación práctica

Los registros de tipo $aX tienen como propósito principal servir como **medio para ingresar los datos de entrada a una subrutina**. Actúan como los parámetros que recibe una función al ser invocada.

Los registros $vX, por su parte, están destinados para las **salidas** de una función. El algoritmo retorna un valor, o valores, a través de estos registros una vez que la subrutina ha completado su ejecución.

Ambos tipos de registros son fundamentalmente importantes en el código de MIPS, ya que la comunicación y la operación entre funciones dependen directamente del uso de $aX para pasar parámetros y $vX para devolver resultados.

3. Impacto del uso de registros frente a memoria en el rendimiento de los algoritmos de ordenamiento

Las operaciones que involucran **registros** son intrínsecamente sumamente rápidas, lo que las hace ideales para operaciones donde los datos caben dentro de ellos. Esta característica es muy útil y eficiente cuando un algoritmo trabaja con conjuntos de datos pequeños, como vectores que pueden residir completamente en los registros.

Sin embargo, a medida que el tamaño de los datos aumenta y excede la capacidad de los registros, el algoritmo debe recurrir al uso de la **memoria**

principal. Las operaciones de memoria son considerablemente más lentas en comparación con las operaciones de registro. Por esta razón, el rendimiento de los algoritmos de ordenamiento (y de cualquier algoritmo intensivo en datos) se degrada significativamente a medida que el tamaño del arreglo crece y se requiere un mayor acceso a la memoria externa.

4. Impacto del uso de estructuras de control en la eficiencia de algoritmos en MIPS32

El uso de estructuras de control, como las instrucciones de salto (`branch`) y bucles, en MIPS32 tiene un impacto directo y, a menudo, negativo en la eficiencia del algoritmo debido a los **riesgos de control** en la **pipeline** (segmentación) de la CPU. Las penalizaciones causadas por los *stalls* (pausas) o, peor aún, por las **predicciones de rama incorrectas**, pueden degradar significativamente el rendimiento general. Cada vez que una instrucción de salto altera el flujo secuencial de ejecución, la pipeline puede necesitar ser vaciada y recargada, lo que implica una pérdida de ciclos de reloj.

5. Diferencias en complejidad computacional entre Bubblesort e Insertion Sort y su afectación en MIPS32

Ambos algoritmos, **Bubblesort** e **Insertion Sort**, tienen una complejidad computacional de $O(n^2)$ en el peor caso y en el caso promedio. Sin embargo, existen diferencias prácticas importantes en el número de operaciones reales que realizan:

- **Bubblesort**: En cada pasada, recorre el arreglo y compara elementos adyacentes, intercambiándolos si están en el orden incorrecto. Incluso si el arreglo está casi ordenado, realiza múltiples pasadas y comparaciones redundantes, lo que lo hace ineficiente. El número de intercambios es una de sus mayores desventajas.
- **Insertion Sort**: Construye el arreglo ordenado de a un elemento a la vez. Toma cada elemento del arreglo de entrada y lo inserta en su posición correcta dentro de la parte ya ordenada del arreglo. Es muy eficiente para arreglos pequeños o casi ordenados, ya que el número de operaciones de comparación e intercambio es menor que en Bubblesort en estas situaciones.

En general, **Insertion Sort es preferible a Bubblesort** para la mayoría de los casos. Su mejor comportamiento en el caso promedio, un menor número de operaciones de intercambio y una mejor localidad de referencia lo hacen más eficiente en términos de ciclos de CPU y uso de caché. Esto resulta en un tiempo

de ejecución más rápido en el entorno MIPS32, ya que implica menos accesos a memoria y una ejecución más predecible dentro de la pipeline.

6. Fases del ciclo de ejecución de instrucciones en MIPS32 (Camino de Datos)

El ciclo de ejecución de instrucciones en MIPS32, a través de su camino de datos, se divide en cinco fases principales:

1. IF (Instruction Fetch - Búsqueda de Instrucción)

Consiste en: En esta primera fase, el procesador busca la siguiente instrucción a ejecutar de la memoria de instrucciones. El Contador de Programa (PC - *Program Counter*) contiene la dirección de memoria de la instrucción actual. El valor del PC se utiliza para acceder a la memoria de instrucciones y recuperar la instrucción de 32 bits. Simultáneamente, el PC se incrementa para apuntar a la siguiente instrucción en secuencia (PC = PC + 4, ya que las instrucciones MIPS son de 4 bytes).

Componentes principales del camino de datos: Contador de Programa (PC), Memoria de Instrucciones, sumador para PC+4.

2. ID (Instruction Decode / Register Fetch - Decodificación de Instrucción / Lectura de Registros)

Consiste en: La instrucción de 32 bits que se acaba de buscar se decodifica para determinar qué tipo de operación se va a realizar (ej. aritmética, lógica, carga/almacenamiento, salto). Al mismo tiempo, se leen los valores de los registros que la instrucción necesita como operandos (si es una instrucción tipo R o I). La decodificación implica identificar los campos de la instrucción (opcode, registros fuente, registro destino, inmediato, etc.). Los datos inmediatos (si existen) también se extienden con signo en esta etapa para convertirse a 32 bits.

Componentes principales del camino de datos: Unidad de Control, Banco de Registros (*Register File*), unidad de extensión de signo.

3. EX (Execute / Address Calculation - Ejecución / Cálculo de Dirección)

Consiste en: En esta fase, se realiza la operación principal de la instrucción. Para instrucciones aritméticas/lógicas (tipo R e I), la Unidad Lógico-Aritmética (ALU - *Arithmetic Logic Unit*) realiza la operación especificada (suma, resta, AND, OR, etc.) utilizando los operandos leídos en la fase ID. Para instrucciones de carga o almacenamiento (*load/store*), la ALU calcula la dirección de memoria efectiva sumando el valor de un registro base con el desplazamiento inmediato.

Para instrucciones de salto (*branches/jumps*), la ALU puede calcular la dirección de destino del salto.

Componentes principales del camino de datos: Unidad Lógico-Aritmética (ALU), unidades de suma/resta para direcciones de salto.

4. MEM (Memory Access - Acceso a Memoria)

Consiste en: Esta fase se utiliza para acceder a la memoria de datos si la instrucción lo requiere. Para instrucciones de carga (*load*), se lee un dato de la memoria de datos en la dirección calculada en la fase EX. Para instrucciones de almacenamiento (*store*), se escribe un dato (que proviene de un registro leído en la fase ID) en la memoria de datos en la dirección calculada en la fase EX. Las instrucciones que no requieren acceso a memoria (como las aritméticas/lógicas) simplemente pasan por esta fase sin realizar ninguna operación de memoria.

Componentes principales del camino de datos: Memoria de Datos.

5. WB (Write Back - Escritura en Registros)

Consiste en: En esta fase final, el resultado de la operación se escribe de nuevo en el Banco de Registros. Para instrucciones aritméticas/lógicas y de carga, el resultado de la ALU (en el caso de operaciones) o el dato leído de la memoria (en el caso de cargas) se escribe en el registro destino especificado por la instrucción. Las instrucciones de almacenamiento y salto, que no producen un resultado para ser escrito en un registro, no realizan ninguna operación en esta fase.

Componentes principales del camino de datos: Banco de Registros.

7. Tipos de instrucciones más frecuentes en la práctica (R, I, J) y su razón

En los algoritmos de ordenación como Bubblesort e Insertion Sort, los tipos de instrucciones más frecuentemente utilizadas en MIPS32 serían las de **tipo I (Immediate)** y **tipo R (Register)**. Las instrucciones de **tipo J (Jump)** se usarían con mucha menos frecuencia.

La razón es que las operaciones fundamentales de estos algoritmos implican:

- **Comparaciones y operaciones aritméticas/lógicas**: Realizadas entre valores en registros (tipo R, como `add`, `sub`, `slt`) o entre un registro y un valor inmediato (tipo I, como `addi`, `andi`).
- **Carga y almacenamiento de datos**: Esenciales para acceder a elementos del arreglo en memoria. Las instrucciones `lw` (load word) y `sw` (store word) son de tipo I y son el corazón de cualquier operación que manipule datos en un arreglo. Bubblesort, con sus numerosos intercambios, generará muchas parejas `lw/sw`. Insertion Sort, aunque con menos

intercambios directos, sigue usando `lw` y `sw` para leer el .[el]emento claveτ desplazar elementos.

- **Saltos condicionales**: Para controlar los bucles y las estructuras `if/else` (`beq`, `bne`), que son también instrucciones de tipo I.

Las instrucciones de tipo J, que son saltos incondicionales a una dirección de memoria específica (`j`, `jal`), son menos frecuentes en el cuerpo principal de los algoritmos de ordenamiento, siendo más comunes para llamadas a funciones o para saltos a secciones de código distantes que no son parte del flujo iterativo principal.

8. Impacto en el rendimiento por abuso de instrucciones de salto (tipo J) en lugar de instrucciones lineales

El abuso de instrucciones de salto (tipo J) y, en general, de cualquier tipo de salto (incluyendo las condicionales tipo I como `beq` y `bne`) en lugar de un flujo de instrucciones lineales tiene un impacto significativo y negativo en el rendimiento de un procesador MIPS32, especialmente en arquitecturas modernas con **pipelining** (segmentación) y **caché**.

1. Interrupción del Pipeline (Segmentación)

Los procesadores modernos, incluido el MIPS32 en sus implementaciones más comunes, utilizan una técnica llamada *pipelining* (segmentación) para ejecutar instrucciones de manera más eficiente. Esto significa que varias instrucciones están en diferentes etapas de ejecución simultáneamente (búsqueda, decodificación, ejecución, acceso a memoria, escritura en registros). Cuando ocurre un salto, el procesador no sabe qué instrucción ejecutar a continuación hasta que la dirección de destino del salto es calculada y la condición es evaluada. Esto crea un riesgo de controlτ a menudo resulta en:

- **Stalls (paros)**: El pipeline debe detenerse y esperar a que la instrucción de salto se complete y la dirección de destino sea conocida. Esto introduce burbujas en el pipeline, donde las unidades funcionales están ociosas.
- **Flushing (vaciado)**: Las instrucciones que ya se habían buscado o decodificado siguiendo el flujo secuencial, pero que resultan ser las incorrectas debido al salto, deben ser descartadas. El pipeline debe ser vaciado y luego rellenado con las instrucciones de la nueva dirección de destino, lo que consume ciclos de reloj.

2. Fallos en la Predicción de Saltos

Los procesadores avanzados emplean **predictores de saltos** para mitigar la penalización del pipeline. Estos circuitos intentan adivinar si un salto condicional será tomado o no, y si lo será, cuál será su destino, para poder empezar a buscar instrucciones en esa dirección de forma anticipada.

- **Predicciones correctas**: Si el predictor acierta, la penalización es mínima o nula.
- **Predicciones incorrectas**: Si el predictor falla, el costo es mucho mayor. El pipeline debe ser vaciado y se incurren en los *stalls* y el *flushing* mencionados anteriormente, ya que el procesador debe revertir su estado y comenzar a buscar las instrucciones correctas desde la dirección real del salto. Esto puede causar una degradación significativa del rendimiento.

En contraste, un flujo de instrucciones lineal permite que el pipeline funcione de manera óptima, ya que la siguiente instrucción es siempre predecible (la siguiente en memoria). Por lo tanto, un código con menos saltos y un flujo más lineal será generalmente más eficiente en términos de ciclos de CPU.

9. Ventajas del modelo RISC de MIPS en la implementación de algoritmos simples como los de ordenamiento

El modelo RISC (*Reduced Instruction Set Computer*), ejemplificado por la arquitectura MIPS, ofrece varias ventajas significativas al implementar algoritmos simples como los de ordenamiento (por ejemplo, Bubblesort, Insertion Sort):

- **Conjunto de Instrucciones Simplificado y Uniforme**: MIPS tiene un conjunto de instrucciones pequeño, fijo y simple. Esto facilita la decodificación de instrucciones y permite una implementación de **pipeline eficiente**. Para algoritmos con bucles intensivos como los de ordenamiento, cada instrucción se ejecuta rápidamente a través del pipeline sin interrupciones complejas.
- **Registros Abundantes y Uso Intensivo de Registros**: MIPS proporciona una gran cantidad de registros de propósito general (32 en total). Las operaciones entre registros son extremadamente rápidas, lo que reduce la necesidad de acceder a la memoria (que es más lenta). Los algoritmos de ordenamiento se benefician enormemente al mantener datos y contadores dentro de los registros siempre que sea posible, minimizando los cuellos de botella de memoria.
- **Arquitectura Load/Store**: En MIPS, solo las instrucciones explícitas de carga (`lw`) y almacenamiento (`sw`) acceden a la memoria de datos. Esto

simplifica la unidad de control y permite que la ALU opere exclusivamente con datos en registros. Esta clara separación mejora la eficiencia del pipeline y la predicción de rendimiento, especialmente en bucles donde se manipulan elementos de un arreglo.

- **Formato de Instrucción Fijo**: Todas las instrucciones MIPS tienen un tamaño fijo de 32 bits, lo que simplifica la fase de búsqueda y decodificación de instrucciones en el pipeline.
- **Mayor Velocidad de Reloj Posible**: La simplicidad de las instrucciones y la estructura del pipeline permiten a los procesadores RISC operar a frecuencias de reloj más altas, lo que se traduce en más instrucciones ejecutadas por segundo.
- **Compiladores Optimizados**: La regularidad y simplicidad de la arquitectura RISC facilitan la creación de compiladores altamente optimizados que pueden generar código muy eficiente, aprovechando al máximo los registros y la pipeline para algoritmos repetitivos.

Para algoritmos como Bubblesort e Insertion Sort, que se caracterizan por operaciones repetitivas, comparaciones y accesos a la memoria dentro de bucles, los principios RISC de MIPS se traducen en:

- **Ejecución de instrucciones más rápida**: Debido a instrucciones simples y una segmentación eficiente.
- **Reducción de cuellos de botella de memoria**: Gracias a la arquitectura de carga/almacenamiento y los amplios registros, lo que minimiza los accesos lentos a la memoria principal.
- **Optimización más sencilla**: Tanto para programadores humanos como para compiladores.

Estos factores contribuyen a un rendimiento más predecible y, a menudo, superior para este tipo de algoritmos en procesadores RISC MIPS en comparación con lo que se podría lograr en una arquitectura CISC con velocidades de reloj similares pero conjuntos de instrucciones más complejos y una segmentación menos directa.

10. Uso del modo paso a paso (step, step into) en la verificación del funcionamiento del algoritmo

La herramienta de **modo paso a paso** (`step`, `step into`) en entornos de depuración como MARS es fundamental para la verificación del funcionamiento de un algoritmo. Una vez ensamblado el código, esta funcionalidad permite

ejecutar el programa **línea a línea** en lugar de ejecutar todo el código de una sola vez.

Esto es tremendamente conveniente cuando se quiere analizar el funcionamiento detallado de un programa. Permite no solo observar un resultado erróneo al final, sino **visualizar cómo se construye el resultado a partir de la lógica interna del algoritmo**. Es posible inspeccionar el contenido de las variables (registros y memoria) línea a línea, observar la transferencia de datos, y entender el flujo de control del programa.

Dicha práctica también nos permite abordar mejor cierto tipo de errores, como lo son las **estructuras iterativas con falla en su condición de parada**, lo que normalmente llamamos bucles infinitos. Al estar acotado el funcionamiento del algoritmo por la cantidad de pasos que queramos realizar, podemos ver de manera detallada cómo los bucles no terminan, facilitando la identificación de la condición errónea.

11. Herramienta de MARS más útil para observar registros y detectar errores lógicos

Cuando se ensambla el código en MARS, se despliega una pantalla con múltiples tablas e indicadores, que son herramientas que nos permiten monitorear distintas etapas y procesos de nuestro programa. En este caso, la herramienta más útil para observar los registros y detectar errores lógicos es el **desplegable que contiene a todos los registros y su valor durante la ejecución del programa**, junto con el **código en tiempo real señalado por líneas**, indicando exactamente dónde se encuentra el programa actualmente.

Esta combinación permite al programador:

- **Rastrear los valores de los registros**: Ver cómo los datos cambian en cada registro después de la ejecución de cada instrucción. Esto es crucial para verificar si las operaciones aritméticas, lógicas o de carga/almacenamiento están produciendo los resultados esperados.
- **Seguir el flujo de ejecución**: La indicación de la línea de código actual en tiempo real permite comprender el camino que toma el programa a través de bucles y saltos, lo cual es vital para depurar errores de lógica en estructuras de control.

Esta capacidad de observar el estado interno del procesador y el flujo de ejecución en detalle nos ayudó significativamente a detectar errores lógicos y problemas durante la ejecución de los algoritmos.

12. Representación del camino de datos para una instrucción tipo R: `add`

Consideremos la instrucción `add $rd, $rs, $rt`, donde `$rd` es el registro destino, y `$rs`, `$rt` son los registros fuente.

1. Búsqueda de la Instrucción (Instruction Fetch - IF):

- **Program Counter (PC)**: El PC contiene la dirección de memoria de la instrucción actual (`add`).
- **Memory (Instruction Memory)**: La dirección del PC se envía a la memoria de instrucciones.
- **Instruction Register (IR)**: La instrucción (`add`) se lee de la memoria y se almacena en el Registro de Instrucciones (IR).
- **PC + 4**: El PC se incrementa en 4 (para la siguiente instrucción) y el nuevo valor se almacena de nuevo en el PC.

2. Decodificación de la Instrucción y Búsqueda de Registros (Instruction Decode/Register Fetch - ID/RF):

- **Instruction Register (IR)**: La instrucción almacenada en el IR se decodifica. Para una instrucción `add` (tipo R), se identifican los campos *opcode*, *rs*, *rt*, *rd*, *shamt* y *funct*.
- **Register File (Banco de Registros)**:
  - Los campos *rs* y *rt* (números de registro fuente) se utilizan para leer los valores correspondientes del banco de registros.
  - Los valores leídos de los registros *rs* y *rt* se almacenan en búferes temporales (por ejemplo, Read Data 1 y Read Data 2).

3. Ejecución (Execute - EX):

- **ALU (Unidad Aritmético Lógica)**:
  - Los valores de los registros *rs* y *rt* (obtenidos en la etapa anterior) se envían como entradas a la ALU.
  - La ALU realiza la operación de adición (`add`) con estos dos operandos.
  - El resultado de la adición se genera en la salida de la ALU.

4. Acceso a Memoria (Memory Access - MEM):

- Para una instrucción `add` (tipo R), esta etapa no realiza ninguna operación de acceso a memoria (ni lectura ni escritura de datos). Sin embargo, la instrucción pasa por esta etapa del pipeline.

5. Escritura de Resultado (Write Back - WB):

- **Register File (Banco de Registros)**:
  - El resultado de la ALU (la suma de los operandos) se envía de vuelta al banco de registros.
  - El campo *rd* (número de registro destino) de la instrucción se utiliza para especificar el registro donde se escribirá el resultado.
  - El valor del resultado se escribe en el registro *rd* del banco de registros.

13. Representación del camino de datos para una instrucción de tipo I: `lw`

Consideremos la instrucción `lw $rt, offset($rs)`, donde `$rt` es el registro destino (donde se cargará el dato), `$rs` es el registro base que contiene parte de la dirección, y `offset` es el desplazamiento inmediato.

1. Búsqueda de la Instrucción (Instruction Fetch - IF):

- **Program Counter (PC)**: Contiene la dirección de memoria de la instrucción `lw`.
- **Memory (Instruction Memory)**: La dirección del PC se envía a la memoria de instrucciones.
- **Instruction Register (IR)**: La instrucción `lw` se lee de la memoria y se almacena en el Registro de Instrucciones (IR).
- **PC + 4**: El PC se incrementa en 4 para apuntar a la siguiente instrucción.

2. Decodificación de la Instrucción y Búsqueda de Registros (Instruction Decode/Register Fetch - ID/RF):

- **Instruction Register (IR)**: La instrucción `lw` (tipo I) se decodifica. Se identifican los campos *opcode*, *rs*, *rt* (registro destino), e *immediate* (constante de 16 bits).
- **Register File (Banco de Registros)**:
  - El campo *rs* (registro base) se utiliza para leer su valor del banco de registros. Este valor se almacenará temporalmente (por ejemplo, en Read Data 1).
  - El campo *immediate* se extiende a 32 bits (extensión de signo) para su uso en la etapa de ejecución.

3. Ejecución (Execute - EX):

- **ALU (Unidad Aritmético Lógica)**:
  - La ALU recibe dos entradas: el valor del registro *rs* (dirección base) y el valor *immediate* extendido a 32 bits.
  - La ALU realiza una operación de suma para calcular la dirección de memoria efectiva: Dirección_Efectiva = Valor_rs+Immediate_Extendido.
  - El resultado de esta suma es la dirección de la palabra en memoria que queremos cargar.

4. Acceso a Memoria (Memory Access - MEM):

- **Memory (Data Memory)**:
  - La dirección efectiva calculada por la ALU se envía a la memoria de datos.
  - Se activa la señal de control de lectura (*MemRead*).
  - La palabra de 32 bits almacenada en esa dirección de memoria se lee y se almacena en un búfer temporal (*Read Data from Memory*).

5. Escritura de Resultado (Write Back - WB):

- **Register File (Banco de Registros)**:
  - El dato leído de la memoria de datos (en la etapa MEM) se envía de vuelta al banco de registros.
  - El campo *rt* (registro destino de la instrucción `lw`) se utiliza para especificar el registro donde se escribirá el dato.
  - El valor leído de la memoria se escribe en el registro *rt* del banco de registros.

14. Justificación de la elección del algoritmo alternativo

Nosotros elegimos el **Insertion Sort** como algoritmo alternativo al Bubblesort debido a varias razones:

- **Facilidad de comprensión y ensamblaje**: Nos pareció un algoritmo relativamente fácil de entender en su lógica y, por ende, de ensamblar en MIPS32.
- **Eficacia en su tarea**: Es un algoritmo de ordenamiento simple pero efectivo para conjuntos de datos pequeños.

- **Similitud con Bubblesort**: Ambos algoritmos tienen una complejidad en el peor caso de $O(n^2)$, lo que los hace comparables en términos de comportamiento asintótico. Sin embargo, la lógica interna y la ejecución demuestran diferencias prácticas.
- **Eficiencia en casos específicos**: Aunque $O(n^2)$ en el peor caso, Insertion Sort es muy eficiente para arreglos pequeños o para aquellos que ya están casi ordenados. En estas situaciones, requiere menos operaciones de intercambio que Bubblesort.

Ambos algoritmos son útiles en contextos de arreglos no extremadamente grandes y ambos son igualmente ineficientes cuando la cantidad de elementos del arreglo se vuelve masiva.

15. Análisis de los datos

Como conclusión, analizamos ambos algoritmos, Bubblesort e Insertion Sort, en términos de su funcionamiento y rendimiento con arreglos de tamaños comunes y arreglos extremadamente grandes. Los dos algoritmos demostraron ser ciertamente parecidos en su complejidad asintótica ($O(n^2)$), siendo eficientes en primera instancia para datos pequeños. Sin embargo, rápidamente pierden eficiencia al tener que ejecutar múltiples instrucciones y realizar numerosos accesos a memoria para arreglos extremadamente grandes, confirmando las predicciones de la complejidad computacional. La elección de Insertion Sort se justificó por su relativa facilidad de implementación y su mejor rendimiento en el caso promedio o con datos parcialmente ordenados, en comparación con Bubblesort.