Pontificia Universidad Católica Madre y Maestra
Facultad de Ciencias e Ingeniería
Escuela de Ingeniería en Computación y Telecomunicaciones
ICC-Metodología de la Investigación
Período 3-2024-2025
Prof.: Lisibonny Beato
Práctica II: Población, muestreo y mediciones

IDs y Nombres: Carolina Bencosme-10148929 y Manuel Rodríguez- 10150681

Con el problema de investigación que le fue asignado en la práctica anterior debe trabajar en los siguientes puntos:

a. Realizar las mejoras necesarias a la práctica I, tal y como le fueron indicadas en la corrección de la misma, y colocar aquí debajo de manera compacta los siguiente:

   a. Planteamiento del problema de investigación:
   Los algoritmos de ordenamiento son esenciales en la ciencia de la computación, ya que se utilizan ampliamente en estructuras de datos, bases de datos, motores de búsqueda, inteligencia artificial y otras aplicaciones críticas. Su rendimiento, sin embargo, no depende únicamente de su complejidad teórica (por ejemplo, $O(n^2)$ para Bubble Sort o $O(n \log n)$ para Merge Sort), sino que se ve influenciado por factores como el tamaño del arreglo de entrada, la distribución de los datos, el lenguaje de programación empleado y las características del entorno de ejecución. Estudios previos, como los de Knuth (1998) y Cormen et al. (2009), destacan que las implementaciones prácticas de estos algoritmos pueden variar significativamente en eficiencia según el contexto, lo que subraya la necesidad de evaluaciones empíricas en entornos controlados. En este sentido, surge la necesidad de comparar experimentalmente el rendimiento de algoritmos de ordenamiento tradicionales —Bubble Sort, Merge Sort, Quick Sort, Heap Sort y Counting Sort— en función de su tiempo de ejecución, considerando diferentes tamaños de entrada y lenguajes de programación ampliamente utilizados como Java, C++ y Python.

   Esta investigación tiene como objetivo evaluar y comparar el rendimiento de los algoritmos mencionados en términos de tiempo de ejecución al procesar arreglos de números enteros de tamaños crecientes (de 1,000 a 1,000,000 elementos), implementados en Java, C++ y Python, en un entorno controlado con datos generados aleatoriamente. El estudio aborda la problemática de seleccionar algoritmos óptimos para aplicaciones que manejan grandes volúmenes de datos, como sistemas de bases de datos o análisis de big data, donde el tiempo de ejecución impacta directamente la eficiencia y los costos computacionales. Al analizar el comportamiento de estos algoritmos bajo diferentes condiciones, esta investigación busca proporcionar una guía práctica para desarrolladores, identificando cuáles son más eficientes según el tamaño del arreglo y el lenguaje de

programación, contribuyendo así a la optimización de recursos en el desarrollo de software.

b. Hipótesis/pregunta de investigación seleccionada:
¿Cuál de los algoritmos de ordenamiento (Bubble Sort, Merge Sort, Quick Sort, Heap Sort y Counting Sort) ofrece un mejor rendimiento respecto al tiempo de ejecución al aumentar el tamaño del arreglo de entrada, cuando son implementados en distintos lenguajes de programación como Java, C++ y Python, y evaluados bajo un entorno controlado de pruebas con entradas de diferentes tamaños generadas aleatoriamente?

c. Variables dependientes, independientes y control, así como las escalas y/o niveles de como va a trabajarlas para su investigación.
   i. Variable dependiente:

      - Nombre: *Tiempo de ejecución del algoritmo.*

      - Tipo de dato: Numérico continuo.

      - Escala/Nivel: Escala de razón (medido en nanosegundos, milisegundos o segundos, según el tamaño del array).

   ii. Variable(s) independiente(s): Tamaño del array, Algoritmo de ordenamiento (ej. Bubble Sort, Merge Sort, Quick Sort).
      - *1. Tamaño del arreglo de entrada*

      - Tipo de dato: Numérico discreto.

      - Escala: Escala de razón.

      - Niveles previstos: 6 niveles de 1000 a 1000000 de elementos.

      - *2. Tipo de algoritmo de ordenamiento*

      - Tipo de dato: Categórico nominal.

- Niveles previstos: Bubble Sort, Merge Sort, Quick Sort, Heap Sort, Counting Sort.

- *3. Lenguaje de programación*

- Tipo de dato: Categórico nominal.

- Niveles previstos: Java, C++, Python.

- *4. Tipo de distribución del arreglo de entrada*

- Tipo: Categórico nominal

- Niveles:

- Alta repetición de valores

- Valores únicos

- Parcialmente ordenado

- Positivos y negativos

- Aleatorio puro

iii. Variable(s) a controlar**:**

Variable de Control	**Tipo de Dato**	**Nivel o condición específica de control**
Procesador (marca, modelo y frecuencia).	Categórico	Intel(R) Core(TM) i5-10500H CPU @ 2.50GHz at 12x4.5GHz (x86_64).
Memoria RAM	Categórico	16GB DDR4
Sistema Operativo	Categórico	Arch Linux

		(https://archlinux.org/) 6.15.4-arch2-1
Estado del sistema	Categórico	Sistema en reposo con procesos no esenciales desactivados, CPU y memoria monitoreadas para uso < 5%, y ejecución en modo de alto rendimiento sin interrupciones externas.
Versión del lenguaje de programación	Categórico	Java 21 (LTS), C++ 17 (estándar), Python 3.12 (última estable).
Versión del compilador/interprete	Categórico	OpenJDK 21 (LTS), GCC 13 (LTS), Python 3.12 (intérprete oficial).
Configuración de energía	Categórico	Modo de alto rendimiento activado
Número de hilos/cores utilizados	Numérico discreto	Se usará un solo hilo (monohilo) para asegurar igualdad en pruebas.
Repeticiones por prueba	Numérico discreto	10 ejecuciones por configuración para promediar.

b. Describa en detalle las características de la población de la que recolectará los datos para su investigación y especifique el método de muestreo que utilizará sobre dicha población para su investigación. Justifique, además, el por qué de la selección de dicho método.

Características de la población

IDs y Nombres: Carolina Bencosme-10148929 y Manuel Rodríguez- 10150681

La población de estudio está formada por todos los arreglos de números enteros generados aleatoriamente que serán utilizados como datos de entrada para evaluar el rendimiento de los algoritmos de ordenamiento (Bubble Sort, Merge Sort, Quick Sort, Heap Sort y Counting Sort) implementados en los lenguajes de programación Java, C++ y Python. Esta población es conceptualmente infinita, ya que los arreglos pueden generarse con cualquier combinación de valores, tamaños y configuraciones. Sin embargo, para delimitar el alcance de la investigación, se definen las siguientes características específicas:

- **Tamaño de los arreglos**:
  - Los arreglos varían en tamaño desde pequeños (1,000 elementos) hasta grandes (1,000,000 elementos), con incrementos definidos (por ejemplo, 1,000; 10,000; 50,000; 100,000; 500,000; 1,000,000).
  - Esta variación permite evaluar el rendimiento de los algoritmos en escenarios que van desde aplicaciones con datos pequeños hasta aplicaciones de big data, cubriendo casos de uso representativos en el desarrollo de software.
- **Tipo de datos**:
  - Los arreglos contienen números enteros de 32 bits, con valores en el rango de -2,147,483,648 a 2,147,483,647.
  - Este tipo de datos fue seleccionado por su simplicidad, compatibilidad con los tres lenguajes de programación, y su uso común en pruebas de algoritmos de ordenamiento, lo que facilita la reproducibilidad y comparación con estudios previos.
- **Diversidad de configuraciones**:
  - Los arreglos se generan aleatoriamente con una semilla fija para asegurar la reproducibilidad de los datos en cada prueba.
  - Se incluyen diferentes configuraciones de datos para simular casos reales:
    - **Arreglos con alta repetición de valores**: Por ejemplo, arreglos con muchos valores duplicados para evaluar algoritmos como Counting Sort, que se benefician de esta característica.
    - **Arreglos con valores únicos**: Para probar algoritmos sensibles a la diversidad de datos, como Quick Sort.
    - **Arreglos parcialmente ordenados**: Para simular datos que ya tienen cierto grado de orden, evaluando el impacto en algoritmos como Bubble Sort o Quick Sort.

- **Arreglos con valores positivos y negativos**: Para garantizar que los algoritmos manejen correctamente un rango amplio de números.

- Esta diversidad permite analizar el rendimiento desde múltiples perspectivas, reflejando escenarios prácticos en aplicaciones reales.

El metodo de muestro seleccionado es **Muestreo Aleatorio Simple**

Este muestreo quiere decir que cada posible arreglo de entrada tiene las mismas proabilidades de ser elegido, cumpliendo con las condiciones previamente explicadas. (tamaño y tipo de datos). Esta es la forma más idónea para la presente investigación, ya que se trabajará con datos generados artificialmente a través de funciones de generación aleatoria controlada en Python con random.randint() y luego el dataset será exportado en formato json o .csv para que pueda ser utilizado en todas las implementaciones (Python, Java y C++). Esto se debe a que es necesario controlar la aletoriedad para obtener mayor precisión y validez, por lo tanto, es mejor que cada algoritmo y lenguaje use los mismo datos. El Muestreo Aleatorio Simple (MAS) erradica el sesgo en los datos de entrada, lo que es fundamental para que los resultados de tiempo de ejecución solamente reflejen las diferencias atribuibles al algoritmo o lenguaje, y no a características particulares del arreglo. En caso de ser necesario, la semilla aleatoria fija en cada prueba hace posible recrear el mismo escenario de arreglos para todos los algoritmos y lenguajes, garantizando la equidad en la comparación.

c. Calcule (o determine) el tamaño de su muestra y valore si es factible para su estudio contar con los sujetos experimentales que necesita o si existe alguna limitación.

Para este caso, se partirá del *Benchmarking* que una práctica comúnmente utilizada a la hora de medir o comparar el rendimiento de sistemas, programas o algoritmos bajo características específicas y repetibles. En este caso particular, radica en ejecutar los algoritmos de ordenamiento previamente seleccionados en diversas repeticiones bajo el mismo escenario y medir su tiempo de ejecución, para comparar objetivamente cuál es más eficiente.

Pontificia Universidad Católica Madre y Maestra
Facultad de Ciencias e Ingeniería
Escuela de Ingeniería en Computación y Telecomunicaciones
ICC-Metodología de la Investigación
Período 3-2024-2025
Prof.: Lisibonny Beato
Práctica II: Población, muestreo y mediciones

IDs y Nombres: Carolina Bencosme-10148929 y Manuel Rodríguez- 10150681

Objetivo: Comparar el rendimiento (tiempo de ejecución) de cinco algoritmos de ordenamiento en tres lenguajes de programación sobre arreglos de diferentes tamaños.

CANTIDAD	ALGORITMO DE ORDENAMIENTO	LENGUAJE DE PROGRAMACIÓN	TAMAÑO DE ENTRADA	DISTRIBUCIÓN
	Bubble Sort	Java	50,000	Alta repetición de valores
	Merge Sort	C++	100,000	Valores Únicos
	Quick Sort	Python	200,000	Parcialmente ordenado
	Heap Sort		500,000	Positivos y negativos
	Counting Sort		750,000	Aleatorio Puro
			1,000,000	
TOTAL	5	3	6	5

Se utilizará un diseño de experimento basado en un enfoque factorial. Esto significa que se deben probar todas las combinaciones posibles de las variables independientes, para poder estudiar y analizar sus efectos tanto individuales como combinados sobre la variable dependiente: el tiempo de ejecución del algoritmo.

Por lo tanto, se tiene lo siguiente:

5 algoritmos × 3 lenguajes × 6 tamaños de entrada × 5 distribuciones = 450 combinaciones únicas.

Para garantizar validez estadística y reducir la variabilidad debida al entorno o aleatoriedad, se harán como mínimo 10 repeticiones por combinación.

Tamaño total de la muestra = 450 combinaciones × 10 repeticiones = 4500 ejecuciones.

Cada ejecución significa medir el tiempo de ejecución de un algoritmo con un arreglo de entrada específico y un lenguaje de programación específico en un ambiente controlado.

Los arreglos para las 450 combinaciones se crearán con antelación en Python y se almaenarán en archivos estructurados. Mas adelanta, se importarán en los distintos lenguajes, quitando variabilidades imprevistas en los datos y confirmando que las comparaciones demuestren únicamente las diferencias debidas al algoritmo, lenguaje o distribución.

d. Seleccione él o los instrumentos de medición que utilizará para recopilar los datos desde la muestra seleccionada y verique la confiabilidad y validez el/los mismo/s. Si desarrolla su propio instrumento debe también asegurar su validez y confiabilidad y mostrar lo realizado en detalle.

Por la naturaleza de la investigación, el utensilio principal para la medición, debe ser el registro de tiempo de ejecución de los algoritmos de ordenamiento (evaluado en nanosegundos, milisegundos o segundos, según el tamaño del arreglo). La variable independiente es cuantitativa continua (tiempo), por lo que la opción más viable para la investigación es utilizar herramientas que vienen ya dadas por los lenguajes de programación y a su vez, garantizan alta precisión en la medición del tiempo que tarda en ejecutarse un algoritmo de ordenamiento.

Instrumento para control de tiempo:

Java	C++	Python
System.nanoTime()	<chrono> con high_resolution_clock	time.perf_counter()
El tiempo es devuelto en	std::chrono::high_resolution_clock::now() es popularmente utilizado para encontrar	Tiene la mayor precisión disponible

nanosegundos, grandioso para mediciones de gran precisión.	diferencias de tiempo de manera precisa. Devuelve un objeto de tipo std::chrono::duration, cuya unidad puede variar según el cast, es decir, se puede obtener mediciones en nanosegundos, microsegundos o milisegundos.	cuando se trata de medir intervalos de tiempo, comúnmente utilizada en benchmaking. Devuelve el tiempo en segundos como un número de punto flotante (float).

Confiabilidad del instrumento:

Para mayor confiabilidad y que las mediciones no se vean comprometidas, se optó por ejecutar cada combinación 10 veces y luego, utilizar el promedio de los tiempos registrados.

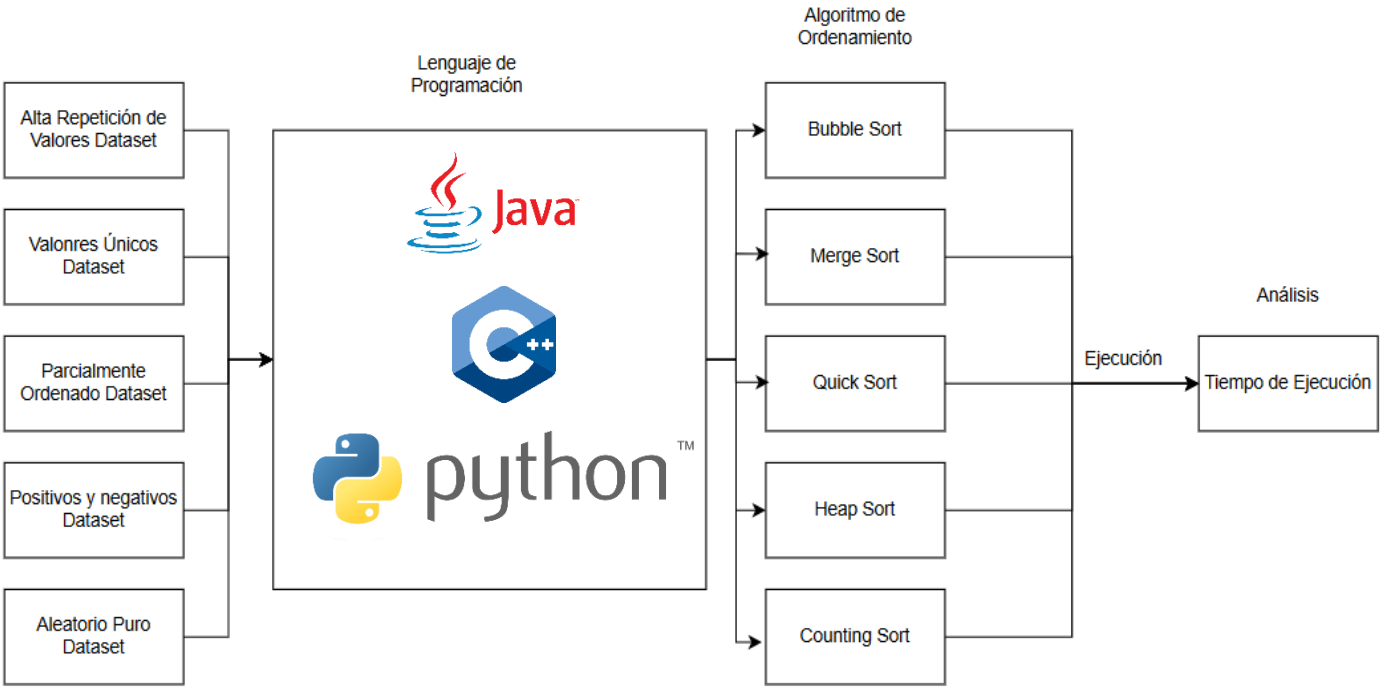
Condiciones controladas:

Todo el experimento será realizado en la misma máquina, con la configuración energética colocada en alto rendimiento, sin tener procesos en segundo plano y utilizando un único hilo.

Instrumento para control de aleatoriedad (Random Seed):

Python
random.seed(seed)
El módulo random permite fijar una semilla antes de generar números aleatorios. Una vez generados los arreglos con las

combinaciones deseadas (tamaño y distribución), se exportarán en formato .json o .csv para ser utilizados por las implementaciones en Java y C++.



Validez del instrumento:

Validez interna: Se está midiendo únicamente el tiempo de ejecución dentro de un entorno controlado, por lo que los cambios en los resultados serán dados por las variables independientes (algoritmo, lenguaje, tamaño, distribución) y no a factores externos.

Validez de contenido: Las funciones seleccionadas (System.nanoTime(), chrono, perf_counter()) existen con el propósito de medir intervalos de tiempo en entornos computacionales y se encuentran explicadas en las documentaciones oficiales de cada lenguaje.

1. Oracle. (2024). System.nanoTime — Java Platform SE 17. Oracle Corporation.

2. cppreference.com. (2024). std::chrono::high_resolution_clock — C++ reference documentation.

3. Python Software Foundation. (2024). time.perf_counter — Python 3.12.3 documentation.

Validez comparativa: Los tres instrumentos son respetables y comunmente usados dentro de sus lenguajes correxpondientes, y del mismo modo, han formado parte de estudios como los de Bhagat et al. (2024) los cuales forman parte de la revisión de literatura de esta investigación.

Bhagat, K., Das, A. K., Agrahari, S. K., Shah, S. A., Deepthi, R. T., & Ramasamy, G. (2024). *Cross-Language Comparative Study and Performance Benchmarking of Sorting Algorithms* [Manuscript]. SSRN.

e. Empiece ya a trabajar en su investigación. (En este punto no debe subir nada)