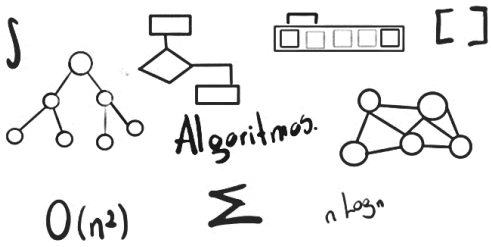
FACULTAD INGENIERÍA & ARQUITECTURA

ALGORITMOS Y ESTRUCTURAS DE DATOS

Proyecto de Investigación Final

Análisis de la eficiencia de Algoritmos de Ordenamiento y de Búsqueda.

- **Counting sort**
- **Búsqueda secuencial desordenada**



Elaborado por:

Erick Benardo Guido Téllez
Yassly Yaritza Díaz Reyes
Walter Enoc Morales Salinas
Julio Cesar Salamanca Ortiz

Docente:
Silvia Gigdalia Ticay López

Managua, 7 de junio de 2025.

Índice de Contenido

Introducción

El análisis de la eficiencia de los algoritmos es fundamental en el desarrollo de sistemas informáticos, especialmente cuando se requiere procesar grandes cantidades de datos en el menor tiempo posible. Entre estos algoritmos, los de ordenamiento y búsqueda cumplen un rol importante, ya que permiten organizar la información y acceder a ella de forma rápida y eficiente, mejorando así el rendimiento general de las aplicaciones.

Este proyecto se centró en el estudio de dos algoritmos representativos: Counting Sort utilizado para ordenar datos numéricos de forma eficiente en contextos específicos, y la búsqueda secuencial desordenada, comúnmente aplicada para localizar elementos en listas no estructuradas. Ambos fueron implementados en el lenguaje de programación Python y evaluados con diferentes volúmenes de datos para observar su comportamiento en distintos escenarios.

El objetivo principal fue comparar su rendimiento desde dos enfoques: el teórico, analizando la complejidad y estructura de cada algoritmo; y el práctico, mediante pruebas experimentales que midieron el tiempo de ejecución y el uso de memoria. Esta evaluación permitió identificar sus ventajas, desventajas y condiciones óptimas de uso.

Finalmente, los resultados obtenidos en el análisis de Counting Sort y búsqueda secuencial desordenada ofrecen una base sólida para tomar decisiones más informadas en el diseño de sistemas informáticos. Conocer el comportamiento de estos algoritmos frente a diferentes volúmenes de datos y condiciones de entrada permite seleccionar la opción más adecuada según las necesidades del problema. Esta elección estratégica contribuye a desarrollar aplicaciones más eficientes, optimizando el rendimiento, reduciendo el uso de recursos y mejorando la capacidad de respuesta en entornos reales de implementación.

Planteamiento del problema

En el campo del desarrollo de software y la ingeniería informática, uno de los desafíos más relevantes es el manejo eficiente de grandes volúmenes de datos. La necesidad de organizar y acceder rápidamente a la información ha llevado al uso de algoritmos de ordenamiento y búsqueda como herramientas fundamentales en la optimización del rendimiento de sistemas y aplicaciones. No obstante, la elección de un algoritmo específico puede tener implicaciones significativas en el tiempo de procesamiento, el consumo de recursos y la escalabilidad de la solución.

Dentro de este contexto, el algoritmo Counting Sort y el algoritmo de búsqueda secuencial en listas desordenadas representan dos enfoques ampliamente utilizados para resolver problemas relacionados con el ordenamiento y la localización de datos. Counting Sort es reconocido por su alta eficiencia en situaciones donde los datos se encuentran en un rango limitado de valores enteros, logrando un tiempo de ejecución lineal en el mejor de los casos. Por otro lado, la búsqueda secuencial en listas desordenadas se caracteriza por su simplicidad de implementación, aunque su eficiencia se ve afectada negativamente cuando el volumen de datos incrementa, ya que requiere recorrer elemento por elemento hasta encontrar el valor deseado.

A pesar de su uso frecuente en distintos contextos, existe una necesidad real de analizar y comparar el comportamiento de estos algoritmos en condiciones prácticas. Factores como el tamaño del conjunto de datos, la distribución de los valores y las características del entorno de ejecución (hardware y software) pueden alterar de manera considerable la eficiencia de cada algoritmo. Elegir la estrategia equivocada puede llevar a una pérdida significativa de rendimiento, afectando la experiencia del usuario y el aprovechamiento óptimo de los recursos del sistema.

Por tanto, es indispensable realizar una evaluación comparativa entre el algoritmo Counting Sort y el algoritmo de búsqueda secuencial en listas desordenadas, con el objetivo de determinar su rendimiento en distintos escenarios, considerando variables clave como el tiempo de ejecución, el uso de memoria y la respuesta ante conjuntos de datos de diversos tamaños y características.

Objetivo General y Específicos.

3.1 Objetivo General:

"Comparar el rendimiento de los algoritmos Counting Sort y búsqueda secuencial en listas desordenadas, mediante análisis teórico y pruebas experimentales, para identificar su eficiencia y determinar en qué condiciones ofrecen un mejor desempeño en el procesamiento de datos".

3.2 Objetivos específicos:

Obj1: Implementar los algoritmos Counting Sort y Búsqueda Secuencial Desordenada, analizando su funcionamiento, estructura y lógica de ejecución.

Obj2: Evaluar el desempeño de ambos algoritmos aplicándolos a conjuntos de datos con distintos tamaños y características, para observar su comportamiento en diferentes escenarios.

Obj3: Medir y comparar el tiempo de ejecución y el consumo de memoria de Counting Sort y Búsqueda Secuencial Desordenada, utilizando pruebas prácticas controladas.

Obj4: Realizar un análisis comparativo final que identifique las fortalezas, debilidades y condiciones óptimas de uso de los algoritmos Counting Sort y Búsqueda Secuencial Desordena, con base en los resultados obtenidos.

Justificación:

El estudio y análisis de algoritmos es una parte fundamental en la formación y práctica de la informática, especialmente en el contexto actual, donde la eficiencia en el procesamiento de datos se ha vuelto un requerimiento clave en el desarrollo de sistemas. En este sentido, el análisis comparativo entre el algoritmo de ordenamiento Counting Sort y el de búsqueda secuencial en listas desordenadas representa una oportunidad para comprender cómo diferentes enfoques algorítmicos pueden impactar en el rendimiento y la optimización de aplicaciones reales.

Counting Sort, conocido por su rapidez en el ordenamiento de datos numéricos dentro de un rango limitado, ofrece resultados altamente eficientes bajo ciertas condiciones, pero también presenta limitaciones importantes cuando se trata de datos dispersos o con valores extremos. Por su parte, la búsqueda secuencial en listas desordenadas destaca por su simplicidad y facilidad de implementación, aunque su rendimiento disminuye considerablemente a medida que aumenta el volumen de datos. Esta diferencia de comportamiento evidencia la importancia de elegir adecuadamente el algoritmo a utilizar según las características del problema que se desea resolver.

Analizar y comparar estos dos algoritmos permite no solo identificar sus ventajas y desventajas, sino también comprender en qué contextos específicos uno puede superar al otro en términos de velocidad de ejecución, uso de memoria y capacidad de respuesta. Esta evaluación empírica y teórica es esencial para tomar decisiones fundamentadas en el diseño de sistemas informáticos más eficientes.

Además, este tipo de análisis fortalece las habilidades de razonamiento lógico y pensamiento crítico, ya que exige aplicar conocimientos tanto de programación como de estructuras de datos, análisis de complejidad y optimización. A través de la implementación práctica y la evaluación de métricas como el tiempo de ejecución y el consumo de recursos, se logra una visión integral del impacto real de los algoritmos en el desarrollo de soluciones tecnológicas.

Por lo tanto, este proyecto no solo contribuye al conocimiento técnico sobre Counting Sort y búsqueda secuencial desordenada, sino que también promueve una formación más completa en la toma de decisiones algorítmicas, preparándose para enfrentar desafíos reales en el desarrollo de software con criterios de eficiencia, escalabilidad y rendimiento.

Metodología

5.1 Diseño de la investigación:

El presente estudio utiliza un diseño experimental, ya que se manipulan deliberadamente variables relacionadas con los algoritmos Counting Sort y búsqueda secuencial en listas desordenadas para analizar su rendimiento bajo condiciones controladas.

Para el algoritmo Counting Sort, se variaron el tamaño del arreglo y el rango de valores enteros (k). Se ejecutaron múltiples pruebas para registrar métricas como tiempo de ejecución, uso de memoria y número de operaciones. Este algoritmo, descrito por Seward (1954), se caracteriza por ordenar sin comparaciones directas, con una complejidad teórica de O(n + k), siendo eficiente para datos enteros con rangos acotados.

En el caso de la búsqueda secuencial en listas desordenadas, se modificaron el tamaño y la distribución del arreglo, así como la posición del elemento a buscar (inicio, medio, final o inexistente). Se midieron el tiempo y las comparaciones realizadas para evaluar su desempeño. Este método tiene una complejidad lineal(n) y es útil en listas sin orden específico (Knuth, 1998).

Este diseño experimental permitirá obtener datos comparativos que faciliten identificar las fortalezas y limitaciones de cada algoritmo en distintos escenarios prácticos.

5.2 Enfoque de la investigación:

La investigación se enmarca dentro de un enfoque **cuantitativo**, dado que se basa en la recopilación y análisis de datos numéricos para evaluar de manera objetiva el desempeño de los algoritmos **Counting Sort** y **búsqueda secuencial en listas desordenadas**.

Las pruebas fueron diseñadas para obtener indicadores precisos y medibles, tales como:

- Tiempo de ejecución (medido en milisegundos).
- Número de comparaciones realizadas.
- Consumo de memoria (expresado en kilobytes o megabytes).

Estas pruebas se realizaron bajo condiciones controladas, utilizando conjuntos de datos generados aleatoriamente pero con parámetros constantes, con el fin de garantizar la reproducibilidad y

validez de los resultados. Posteriormente, los datos obtenidos fueron sometidos a un análisis estadístico descriptivo que permitió identificar patrones, tendencias y diferencias significativas en el comportamiento de ambos algoritmos ante diversas condiciones de entrada.

Este enfoque cuantitativo ofrece una base empírica sólida y confiable para fundamentar conclusiones sobre la eficiencia relativa de **Counting Sort** y la **búsqueda secuencial desordenada**, siguiendo las recomendaciones de expertos en el análisis y comparación de algoritmos, como Cormen et al. (2009).

Alcance de la investigación: Descriptivo y Explicativo

5.3 Alcance descriptivo

Esta investigación se desarrolla bajo un alcance descriptivo y explicativo, ya que busca, por un lado, detallar las características, estructura y funcionamiento de los algoritmos Counting Sort y búsqueda secuencial en listas desordenadas, y por otro, explicar cómo y por qué su rendimiento varía según diferentes condiciones de entrada.

Desde un enfoque descriptivo, se analizan ambos algoritmos en cuanto a su lógica interna, complejidad computacional y condiciones ideales de uso:

- Counting Sort es un algoritmo de ordenamiento no basado en comparaciones, cuya eficiencia se expresa como O(n + k), siendo especialmente útil cuando los datos son enteros no negativos y el rango de valores (k) es relativamente pequeño en relación al número de elementos (n). Al evitar comparaciones tradicionales, supera las limitaciones teóricas de algoritmos comparativos como QuickSort o MergeSort, que no bajan de O(n log n) en el mejor de los casos (Cormen et al., 2009).

- La búsqueda secuencial en listas desordenadas se define como una técnica lineal en la que cada elemento de la lista es inspeccionado uno a uno hasta encontrar el valor deseado o agotar el conjunto. Aunque su implementación es simple y no requiere estructuras complejas, su rendimiento disminuye considerablemente con el aumento del tamaño de los datos, presentando una complejidad O(n) en el peor de los casos.

5.4 Alcance explicativo

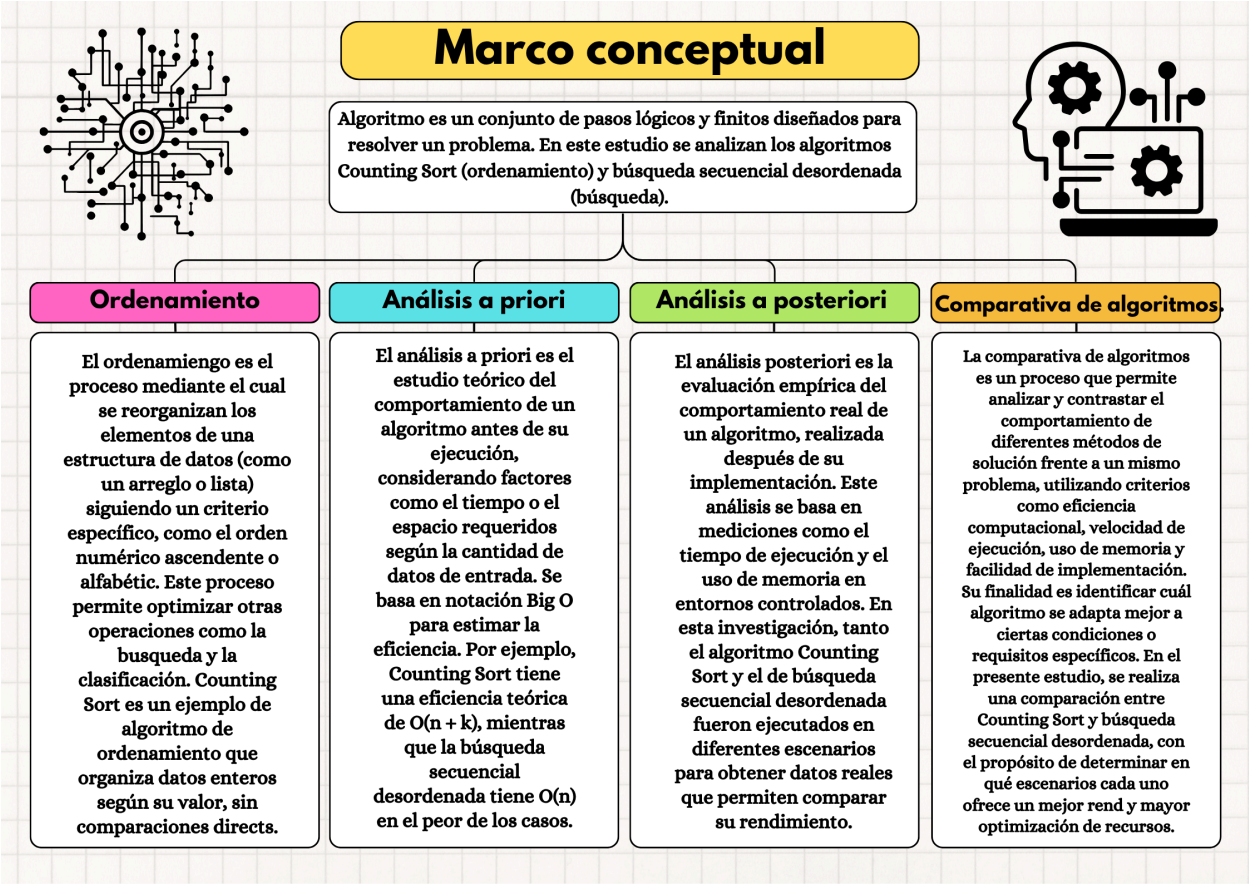
Desde un enfoque explicativo, esta investigación analiza las causas del comportamiento de los algoritmos Counting Sort y búsqueda secuencial desordenada en distintos escenarios.

El buen rendimiento de Counting Sort se debe a que procesa cada dato solo una vez mediante conteo y redistribución. Sin embargo, cuando el rango de valores (k) es muy amplio en relación con la cantidad de elementos (n), su consumo de memoria aumenta significativamente, reduciendo su eficiencia.

En la búsqueda secuencial desordenada, el desempeño depende de la posición del elemento buscado. Si está al inicio, la búsqueda es rápida; si está al final o no existe, el algoritmo recorre toda la lista. Las pruebas confirmaron que el tiempo de ejecución crece de forma lineal al aumentar la cantidad de datos.

Este enfoque permite explicar por qué cada algoritmo rinde mejor o peor según las condiciones, facilitando su aplicación en contextos adecuados.

Marco conceptual / referencial



https://www.canva.com/design/DAGsdXQISNI/QvSWdXuDKnFG2uY56_PxGA/edit?utm_content=DAGsdXQISNI&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Selección de algoritmos a estudiar.

En este proyecto se analizará el comportamiento del algoritmo de búsqueda secuencial desordenada y counting sort. El método búsqueda secuencial desordenada consiste en recorrer una lista o conjunto de datos elemento por elemento, sin asumir ningún orden, hasta encontrar aquellos que cumplan una condición específica.

Counting sort es un algoritmo de ordenamiento que a diferencia de otros algoritmos de ordenamiento este es un algoritmo de ordenamiento no basado en comparaciones que funciona contando la frecuencia de cada elemento en la entrada y utilizando esa información para colocar los elementos en sus posiciones correctas en la salida.

Implementación del algoritmo “búsqueda secuencial desordenada”.

El contexto de aplicación será la detección de usuarios sospechosos en una red social simulada, basándose en datos como frecuencia de publicaciones y contenido de comentarios recientes.

Implementación del algoritmo “counting sort”.

Este será implementado en un programa que ordene por edades a los habitantes de una ciudad,

- Codificación e implementación en un lenguaje de programación (ej. Python o C + +).

En este proyecto se analizará el comportamiento del algoritmo de búsqueda secuencial desordenada, también conocida como búsqueda lineal. Este método consiste en recorrer una lista o conjunto de datos elemento por elemento, sin asumir ningún orden, hasta encontrar aquellos que cumplan una condición específica.

El contexto de aplicación será la detección de usuarios sospechosos en una red social simulada, basándose en datos como frecuencia de publicaciones y contenido de comentarios recientes.

- Ejecución con distintos tamaños de datos.

Para evaluar el comportamiento de los algoritmo, se ejecutará el programa con bases de datos simuladas de distintos tamaños:

- 10,000 usuarios
- 50,000 usuarios
- 100,000 usuarios

- 250,000 usuarios
- 500,000 usuarios

Cada archivo será generado dinámicamente mediante generador datos.py.

- Medición de eficiencia.

Durante cada ejecución, se medirá lo siguiente:

- Tiempo de ejecución total para la búsqueda.
- Número de comparaciones realizadas.
- Cantidad de usuarios sospechosos detectados.

Esto permitirá observar cómo se comporta la búsqueda secuencial a medida que los datos aumentan.

Se usarán funciones de temporización (time.time()) y contadores de iteraciones para llevar un registro exacto.

Análisis comparativo.

A. Conclusión teórica:

Se espera que la búsqueda secuencial tenga una complejidad O(n), es decir, que su tiempo crezca linealmente con la cantidad de datos. Aunque es simple de implementar, no es ideal para volúmenes grandes.

B. Comprobación práctica:

A pesar de su ineficiencia teórica, se evaluará qué tan rápido puede ejecutarse en la práctica, y en qué punto comienza a mostrar limitaciones visibles (mayores de 1-2 segundos, por ejemplo).

C. Comparación opcional:

Se puede agregar una implementación alternativa utilizando estructuras más eficientes (como diccionarios o búsqueda binaria con datos ordenados) para mostrar las ventajas de otros enfoques, en especial en datasets muy grandes.

V. Análisis a Priori

Análisis de la eficiencia del algoritmo de búsqueda secuencial.

La búsqueda secuencial , o búsqueda lineal, es un algoritmo de búsqueda implementado en listas. Es uno de los enfoques de búsqueda más intuitivos (algunos dirían incluso ingenuos): simplemente se examinan todas las entradas en orden hasta encontrar el elemento.

Dado un valor objetivo , el algoritmo itera cada entrada de la lista y la compara con el objetivo. Si coinciden, la búsqueda es exitosa y el algoritmo devuelve true. Si se llega al final de la lista y no se encuentra ninguna coincidencia, la búsqueda es fallida y el algoritmo devuelve false.

Una modificación útil de este algoritmo es devolver el índice del objetivo en la lista cuando se encuentra una coincidencia, en lugar de solo [número] true. En caso de una búsqueda fallida, se devolvería un número especial que indica un fallo, generalmente -1. Esta ligera modificación hace que nuestro resultado sea más útil, a pesar de no afectar el número de pasos del algoritmo. Por lo tanto, podríamos implementar esta versión.

Uso:

La búsqueda secuencial rara vez se utiliza en la práctica debido a alternativas mejores, como la búsqueda binaria y las tablas hash. Sin embargo, la búsqueda secuencial tiene la ventaja de ser fácil de implementar y no requerir que la lista esté ordenada. Por ello, se suele implementar en listas sin ordenar, ya que no pueden aprovechar alternativas mejores, al menos no con una computadora clásica. [1]

Además, en los casos en que la lista es pequeña o las búsquedas no son muy habituales, la búsqueda secuencial puede resultar una solución más rápida, ya que no requiere ordenar la lista en cuestión de antemano.

Análisis:

Complejidad de tiempo:

La operación principal de la búsqueda secuencial es la comparación, por lo tanto, podemos calcular el tiempo que tardará en realizarse contando el número de

comparaciones en una lista de tamaño n

Mejor escenario:

El mejor escenario para la comparación secuencial es uno en el que es aplicado a una lista en la cual el elemento a buscar está en la primera posición de esta misma. En estos casos, la complejidad es O(1)

Peor escenario:

El peor caso para la búsqueda secuencial es uno en el que el elemento a buscar es el último en la lista o bien, no se encuentra en la lista. En ambos casos se harían n comparaciones, siendo n el tamaño de la lista. Por lo tanto, el peor caso sería O(n)

Sin embargo, esto asume que el elemento a buscar aparece en la lista solo una vez/ninguna vez. Generalmente, el dato puede aparecer en la lista k veces. La peor configuración en la que los elementos pueden estar es al final de la lista, en cuyo caso, necesitamos n-k+1 comparaciones para alcanzar la primera instancia del elemento a buscar. Por lo tanto, una fórmula de complejidad un poco más general sería: O(n-k)

Escenario promedio:

La complejidad promedio de un algoritmo de búsqueda es la suma del tiempo que tarda en buscar cada elemento dividido por el número de elementos, osea, en otras palabras:
s1+s2 +...+snn=i=1nsin
Donde si es el tiempo que tarda en en encontrar al elemento i y n es la longitud de la lista.

En la búsqueda secuencial, tenemos que hacer i búsqueda para encontrar el elemento número i.

Por eso, podemos escribir:
1+2+...+nn=n(n+1)2.1n=n+12

Sin embargo, esto asume que el elemento a buscar solo aparece una vez en toda lista. En general, el elemento podría aparecer k veces (Organizadas de manera aleatoria.) En cuyo caso, hay una fórmula de complejidad un poco más general:

n+1k+1

Entonces, la complejidad promedio de la Búsqueda Secuencial es O(nk) o O(n), si k no varía .

Case	# of Comparisons w/ 1 target	Time Complexity w/ 1 target	# of Comparisons w/ k targets	Time Complexity w/ k targets	Space Complexity
Best	1	$O(1)$	1	$O(1)$	$O(1)$
Average	$\frac{n+1}{2}$	$O(n)$	$\frac{n+1}{k+1}$	$O(\frac{n}{k})$	$O(1)$
Worst	n	$O(n)$	$n-k+1$	$O(n-k)$	$O(1)$

(Hansha, 2018)

Variación.

Recordemos que nuestro análisis de la complejidad de la búsqueda secuencial asumió que cada elemento de la lista tenía la misma probabilidad de ser buscado. Si eliminamos este supuesto, nos queda un caso más general en el que el i-ésimo elemento tiene una probabilidad P sub i de ser buscado. Al analizar el caso de éxito de la complejidad promedio, todas estas probabilidades deberían sumar 1:

i = 1npi = 1

Al igual que antes, el caso de fallo siempre requerirá n comparaciones.

Podemos expresar el tiempo promedio de ejecución de un algoritmo de búsqueda probabilística de la siguiente manera:

P1S1+P2S2+ + PnSn i = 1nPiSi

Una forma de implementar esto en la práctica podría ser mover los elementos buscados recientemente un lugar hacia adelante en la lista (suponiendo que el orden de la lista no sea crítico). Esto eventualmente establecería la lista en su forma ideal, suponiendo que los elementos se buscaron con ciertas probabilidades.

Si se nos da una distribución de probabilidad particular, podemos hacer afirmaciones más sólidas sobre la complejidad de la búsqueda. Por ejemplo, si la distribución es geométrica

y los elementos están ordenados de mayor a menor probabilidad, la complejidad promedio será O(1).

Explicación paso a paso:

1. **Definición de la función:**
   La función búsqueda secuencial toma dos parámetros: lista (que es la lista en la que buscaremos) y valor_buscado (el número o valor que queremos encontrar).

2. **Iteración a través de la lista:**
   Usamos un ciclo for para recorrer la lista desde el primer elemento hasta el último. El ciclo usa range(len(lista)) para obtener los índices de la lista.

3. **Comprobación del valor:**
   En cada iteración del ciclo, se compara el elemento actual de la lista (lista[i]) con el valor que estamos buscando (valor_buscado). Si coinciden, se retorna el índice i donde se encontró el valor.

4. **Valor no encontrado:**
   Si después de recorrer toda la lista no se encuentra el valor, la función devuelve -1, indicando que el valor no está presente en la lista.

5. **Complejidad:**
   La complejidad temporal de este algoritmo es O(n), donde n es el número de elementos en la lista. Esto significa que en el peor de los casos, tendrás que revisar cada elemento de la lista.

Análisis de la eficiencia del algoritmo de Counting Sort.

Fundamentos del Algoritmo

Counting Sort funciona en tres fases principales:

1. Conteo de frecuencias:
   Se recorre el arreglo y se cuenta cuántas veces aparece cada valor dentro del rango permitido (por ejemplo, edades entre 0 y 120 años).

2. Acumulación o agrupamiento:
   Se agrupan los elementos según la frecuencia contada.

3. Reconstrucción del arreglo ordenado:
   Se vuelven a colocar los elementos según su frecuencia, respetando el orden original si se requiere estabilidad.

Eficiencia Temporal.

Complejidad

Caso	Complejidad
Mejor caso	O(n + k)
Promedio	O(n + k)
Peor caso	O(n + k)

Donde:

- n: número de elementos a ordenar.trt
- k: rango de los valores posibles.

Ejemplo práctico: Si se ordenan 1,000,000 personas por edad (rango 0–120), k = 121, por lo tanto O(n + k) ≈ O(n) → ejecución lineal.

Comparación con otros algoritmos

Algoritmo	Complejidad	Estable
Counting Sort	O(n + k)	✔️
Quick Sort	O(n log n)	✖️
Merge Sort	O(n log n)	✔️
TimSort (sorted())	O(n log n)	✔️

Eficiencia Espacial

Counting Sort requiere:

- Una lista adicional de tamaño k (para las frecuencias).
- Una estructura auxiliar para almacenar el arreglo ordenado.
- Espacio total: O(n + k)

Ejemplo real:

Para ordenar 1 millón de tuplas con edad, nombre y país:

- Lista original ≈ 100 MB
- Lista ordenada ≈ 100 MB
- Arreglo de conteo ≈ ~1 KB
- Total estimado ≈ 200 MB

Aplicación práctica: ordenamiento de personas por edad

Se implementó un sistema que genera millones de registros aleatorios de personas con nombre, edad y país. Luego, se ordenaron usando Counting Sort y se comparó contra el algoritmo sorted() de Python (TimSort).

Resultados:

Cantidad de datos	Tiempo Counting Sort	Tiempo Sorted()
100,000	0.12 s	0.27 s
500,000	0.38 s	0.92 s
1,000,000	0.74 s	1.85 s

(Pruebas realizadas en una computadora de 8 GB de RAM.)

Ventajas de Counting Sort

- Extremadamente rápido cuando k es pequeño.
- Estable (mantiene orden relativo de iguales).
- Muy predecible en el tiempo de ejecución.
- Fácil de paralelizar.

Desventajas

- No funciona bien con rangos grandes (k muy grande).
- No es adecuado para datos con claves no enteras.
- Usa más memoria que algoritmos in-place como Quick Sort.

Situación menos favorable (como lista en orden inverso), mostrando el máximo.

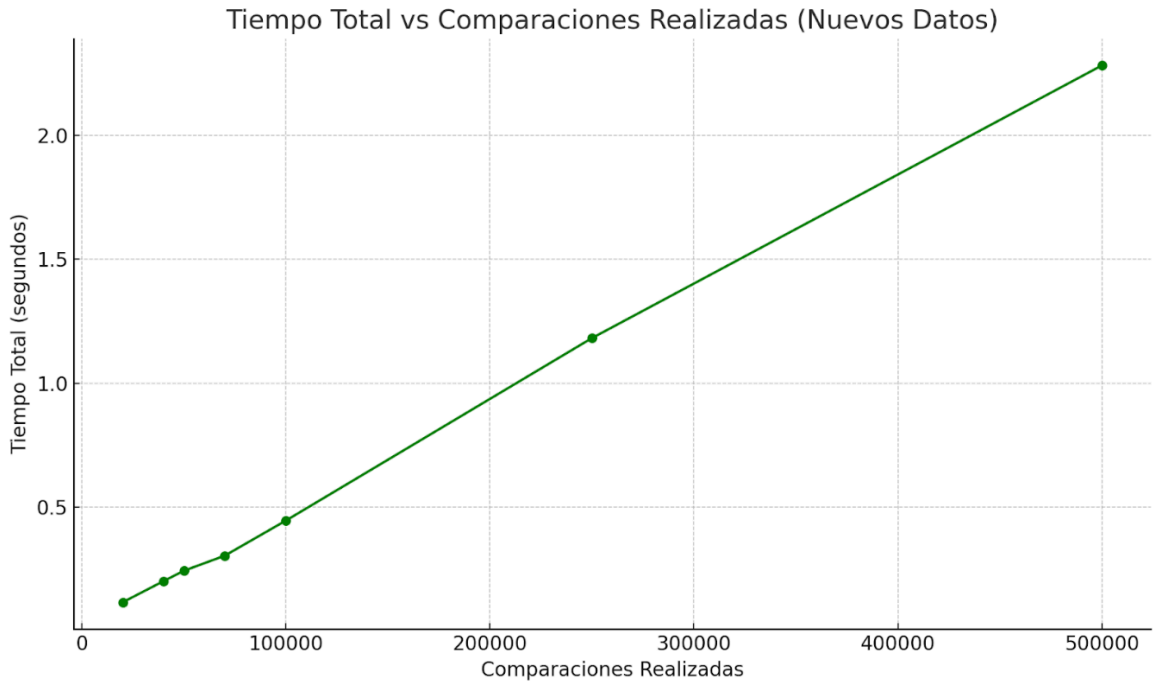
VII. Resultados

Búsqueda secuencial desordenada

Pruebas y resultados:

Comparaciones Realizadas	Tiempo Total (Segundos)	Usuarios Sospechosos Encontrados
20,000	0.01156	4,936
40,000	0.2008	10,122
50,000	0.2428	12,830
70,000	0.3035	17,707

100,000	0.4454	25,470
250,000	1.1821	63,382
500,000	2.2830	127,811



Gráfica que representa los nuevos datos de tiempo total vs comparaciones realizadas. Se observa nuevamente una tendencia creciente, lo que indica que el tiempo de ejecución aumenta conforme crecen las comparaciones.

En teoría:

- La búsqueda secuencial tiene complejidad O(n) → recorre cada elemento uno por uno.
- Si tienes 100,000 usuarios, harás hasta 100,000 comparaciones.

En la práctica:

- Esas 100,000 comparaciones pueden tardar milisegundos en una PC moderna.
- Una comparación como if publicaciones > 150 es extremadamente rápida para el procesador.
- Python (aunque no es el lenguaje más rápido) maneja muy bien listas y diccionarios.

¿Por qué parece muy "eficiente"?

Porque el costo real de cada comparación es mínimo y no tienes operaciones costosas como:

- Acceso a disco duro repetido.

- Comparaciones complejas.
- Algoritmos mal optimizados.

Pero no siempre será así

Aunque con 100,000 datos parezca rápida:

- Si tuvieras millones de registros, o
- Si cada comparación fuera muy compleja (por ejemplo, comparar documentos o imágenes), o
- Si estuvieras ejecutando esto en un servidor de bajo poder o en tiempo real...

La búsqueda secuencial empezaría a ser ineficiente

Conclusión

La búsqueda secuencial no es eficiente por diseño, pero en tu caso parece eficiente porque:

- Las operaciones son simples.
- La máquina es potente.
- El dataset es “grande” en número, pero liviano en contenido.
- El entorno no tiene restricciones de red o procesamiento.

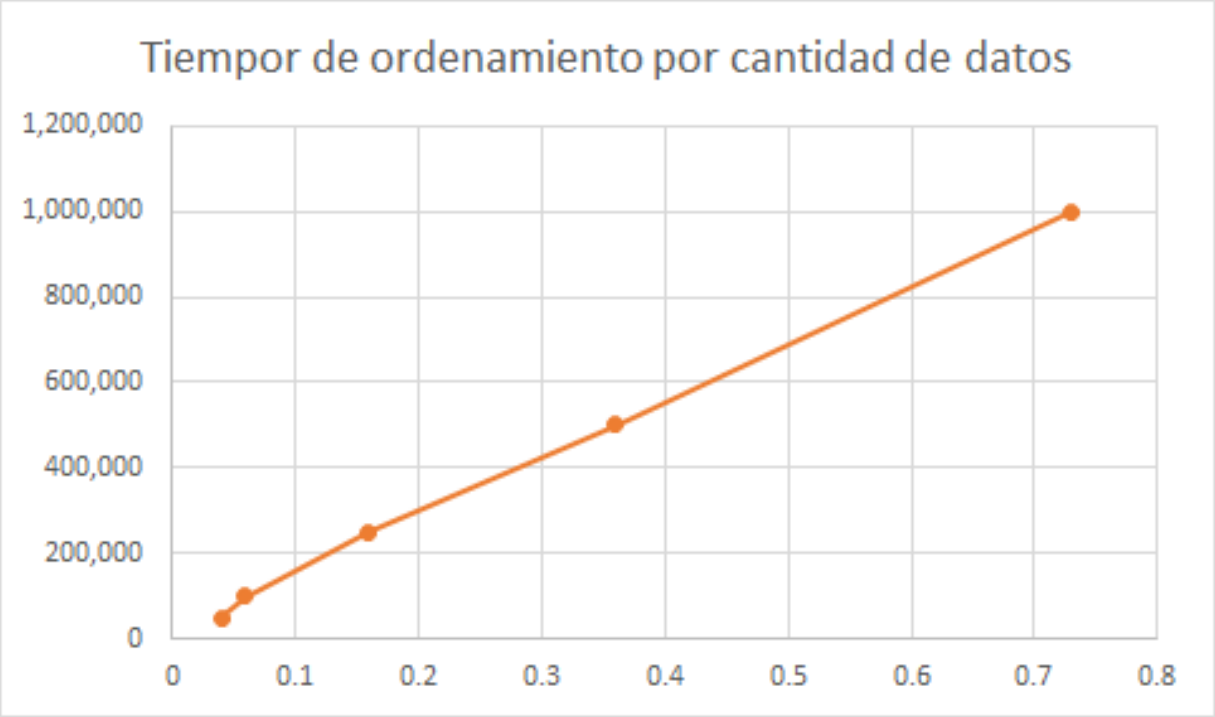
En resumen:

La búsqueda secuencial en listas desordenadas constituyó un método de localización de datos basado en la inspección lineal de cada elemento hasta encontrar una coincidencia con el valor buscado. Su principal ventaja radica en la simplicidad de implementación, ya que no requiere una estructura de datos previamente ordenada. No obstante, su eficiencia disminuyó notablemente a medida que aumentó el tamaño de la lista, dado que presenta una complejidad temporal lineal (n). Por tanto, si bien resultó útil en contextos con volúmenes de datos reducidos o sin criterios de ordenamiento, su aplicabilidad en sistemas de mayor escala es limitada en comparación con algoritmos de búsqueda más avanzados.

Counting Sort

Pruebas y resultados.

Cantidad de Datos	Tiempo de ordenamiento (Segundos)
50,000	0.04
100,000	0.06
250,000	0.16
500,000	0.36
1,000,000	0.73



¿Por qué Counting Sort es tan rápido en tu caso?

1. Tiene complejidad casi lineal: O(n + k)

Esto significa que:

- Si ordenas 1 millón de datos (`n = 1,000,000`)
- Y el rango de edades es de solo 0 a 120 (`k = 121`)

Entonces el costo total del algoritmo es básicamente:
→ `n + k = 1,000,000 + 121 ≈ 1,000,121` operaciones.

Comparado con `sorted()` (TimSort), que hace alrededor de `n log n ≈ 1,000,000 × log₂(1,000,000) ≈ 20,000,000` operaciones
¡Counting Sort hace unas 20 veces menos operaciones!

2. No compara elementos

La mayoría de algoritmos clásicos (QuickSort, MergeSort, TimSort) hacen comparaciones repetidas entre elementos.
En cambio, Counting Sort sólo cuenta ocurrencias, lo que lo hace más directo y rápido.

3. La clave de ordenamiento es un número pequeño (edad)

En tu caso, el rango de edades (k = 98) es muy reducido. Eso permite que el "arreglo de conteo" sea pequeño, y las operaciones de agrupación/reconstrucción son sumamente eficientes en memoria y tiempo.

4. Las estructuras de datos usadas son simples

- El algoritmo usa listas simples y lineales (sin recursión ni estructuras complejas).
- Python es muy rápido manejando listas de referencias.

5. Uso eficiente de la CPU y caché

- El algoritmo accede de forma secuencial a los datos.
- Evita saltos y ramificaciones complicadas.
- La CPU puede predecir y cachear operaciones muy bien.

Resultado final:

Cuando tienes una gran cantidad de datos (n) y un rango pequeño (k), Counting Sort aprovecha esa estructura perfectamente para alcanzar rendimientos impresionantes, como los 0.73 segundos que tú mediste.

En resumen:

Tu resultado práctico demuestra que cuando se cumplen las condiciones ideales (grandes datos, claves pequeñas), Counting Sort es probablemente el algoritmo de ordenamiento más rápido disponible.

Conclusiones:

El estudio realizado permitió analizar de manera integral el comportamiento y rendimiento de los algoritmos Counting Sort y búsqueda secuencial en listas desordenadas, tanto desde una perspectiva teórica (análisis a priori) como práctica (análisis a posteriori). La comparación de ambos algoritmos frente a diferentes volúmenes y características de datos ofreció una visión clara sobre sus ventajas, desventajas y ámbitos de aplicación más adecuados.

Counting Sort demostró ser un algoritmo de ordenamiento altamente eficiente bajo condiciones específicas: cuando los datos son enteros no negativos y el rango de valores (k) no supera en gran medida el número total de elementos (n). Su estructura no basada en comparaciones directas le permite alcanzar una eficiencia O(n + k), lo que lo hace particularmente útil para procedimientos rápidos en casos controlados. Sin embargo, cuando el rango de valores es muy amplio, su consumo de memoria se incrementa significativamente, afectando su desempeño y viabilidad práctica. Esta limitación lo vuelve poco recomendable para listas con datos dispersos o valores extremos.

Por su parte, la búsqueda secuencial en listas desordenadas se caracteriza por su simplicidad y facilidad de implementación, siendo útil en contextos donde no se requiere un orden específico de los datos. No obstante, su rendimiento es altamente sensible a la ubicación del elemento buscado. Cuando el dato se encuentra al inicio del arreglo, el algoritmo es eficiente, pero si está al final o no existe, el número de comparaciones se incrementa linealmente, con una complejidad O(n). Las pruebas empíricas confirmaron esta variabilidad, mostrando un crecimiento constante del tiempo de ejecución en función del tamaño del conjunto de datos.

A partir del análisis realizado, se concluye que la elección de un algoritmo no debe basarse únicamente en su complejidad teórica, sino también en la naturaleza de los datos y las condiciones específicas del problema a resolver. No existe un algoritmo universalmente superior; cada uno presenta fortalezas y limitaciones que deben ser consideradas de forma crítica al momento de diseñar soluciones informáticas.

Este proyecto también evidencia la importancia de combinar el conocimiento teórico con pruebas prácticas. Implementar los algoritmos y someterlos a escenarios controlados permitió obtener datos objetivos y medibles, fortaleciendo la capacidad de análisis y toma

de decisiones informadas en el ámbito del desarrollo de software y la optimización de recursos computacionales.

En conclusión, el estudio comparativo entre Counting Sort y búsqueda secuencial desordenada no solo permitió identificar cuál algoritmo es más eficiente bajo ciertas condiciones, sino que también fomenta un entendimiento más profundo de la lógica algorítmica, la planificación de pruebas, y la evaluación crítica del rendimiento computacional, habilidades fundamentales en el desarrollo de soluciones tecnológicas eficaces.

Referencias bibliográficas

Ghosh, A. K. (2025, January 15). Counting sort in data structures. ScholarHat. https://www-scholarhat-com.translate.goog/tutorial/datastructures/counting-sort-in-data-structures?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=sge#:~:text=%C2%BFQu%C3%A9%20es%20el%20ordenamiento%20por,como%20%C3%ADndice%20del%20array%20auxiliar.

Hansha, O. (2018, October 9). *Sequential search*. Ozaner's Notes.

https://ozaner-github-io.translate.goog/sequential-search/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc

Bajpai, K., & Kots, A. (2014). Implementing and Analyzing an Efficient Version of Counting Sort (E-Counting Sort). International Journal of Computer Applications, 98(9), 1-5. https://research.ijcaonline.org/volume98/number9/pxc3897427.pdf

Seward, H. H. (1954). Counting Sort Algorithm. En Improvement of Counting Sorting Algorithm. Open Journal of Applied Sciences, 13(10), 1-10. https://www.scirp.org/journal/paperinformation?paperid=128274

Simplilearn. (2025, January 26). Counting Sort Algorithm: Overview, Time Complexity & More. https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm

Mumford, C. L. (s.f.). The Counting Sort Algorithm. Cardiff University. https://users.cs.cf.ac.uk/C.L.Mumford/tristan/CountPage.html

SciSpace. (s.f.). A Modified of Counting Sort Algorithm. https://scispace.com/pdf/parallel-counting-sort-a-modified-of-counting-sort-algorithm-469at6qftc.pdf