UNIVERSIDAD AMERICANA

Facultad de Ingeniería y Arquitectura

Carrera de Ingeniería en Sistemas de Información

Algoritmos y Análisis de Datos

Proyecto de Investigación

Análisis de eficiencias de los algoritmos Selection Sort y Búsqueda Secuencial en Arreglos Ordenados, aplicados a Python

Integrantes:

Farid Eduardo Zuñiga Rico

Dennis Amaru Cruz Abrego

Emilio Fernando Meza Ortiz

José Nahum Espinoza Solano

Docente:

Silvia Gigdalia Ticay López

Managua, Lunes 7 de Julio del 2025

I. Introducción

Implementar los algoritmos **Selection Sort** y **Búsqueda Secuencial** en Python sobre arreglos permitirá disponer de versiones funcionales capaces de procesar conjuntos de distintos tamaños y características. Primero, desarrollaremos Selection Sort para ordenar datos de forma genérica y adaptamos la búsqueda secuencial para interrumpir su recorrido tan pronto como el elemento inspeccionado supere la clave buscada, aprovechando la ordenación previa. Esta fase asegurará que el código sea modular, legible y utilizable en distintos contextos de datos. Además, se incorporarán mecanismos de medición de rendimiento en cada implementación para facilitar el posterior análisis.

A continuación, analizaremos teórica y matemáticamente las complejidades temporal y espacial de ambos métodos en sus mejores, promedios y peores casos, considerando aspectos como estabilidad, operación in-place y sensibilidad a la disposición inicial de los datos. Posteriormente, evaluaremos empíricamente su desempeño registrando tiempos de ejecución y consumo de memoria en escenarios crecientes de tamaño de arreglo (por ejemplo 10^3, 10^4 y 10^5 elementos). Finalmente, contrastaremos los resultados prácticos con las predicciones teóricas para identificar umbrales de eficiencia y proponer pautas precisas sobre cuándo emplear cada algoritmo según las características del problema y las restricciones de recursos.

I.1. Planteamiento del problema

En el desarrollo de software, especialmente en aplicaciones que manejan grandes volúmenes de datos, la eficiencia de los algoritmos de búsqueda y ordenamiento es un factor determinante en el rendimiento general del sistema. Algoritmos poco eficientes pueden provocar tiempos de respuesta elevados y un uso excesivo de recursos, afectando la experiencia del usuario y la escalabilidad de las aplicaciones. (Asesoftware, 2024)

En este contexto, surge la necesidad de analizar y evaluar el comportamiento de algoritmos clásicos como Selection Sort para el ordenamiento, y la Búsqueda Secuencial en arreglos ya ordenados, utilizando el lenguaje de programación Python como medio de implementación.

A pesar de ser algoritmos simples y comúnmente enseñados en cursos introductorios, su análisis detallado permite comprender mejor sus ventajas, limitaciones y casos de uso adecuados. Esto resulta particularmente relevante en entornos donde la claridad del código, el bajo consumo de memoria o la simplicidad de implementación son más importantes que la velocidad pura.

Por tanto, esta investigación busca responder a la pregunta: **¿Qué tan eficientes son los algoritmos Selection Sort y Búsqueda Secuencial cuando se aplican a arreglos ordenados en Python, en términos de tiempo de ejecución y uso de recursos?**

I.2. Objetivo de la investigación

Analizar e identificar la eficiencia de los algoritmos Selection Sort y Búsqueda Secuencial (aplicado a arreglos ordenados), en términos de tiempo de ejecución y uso de recursos, aplicados a Python.

I.3. Objetivos específicos

- Implementar en Python los algoritmos Selection Sort y Búsqueda Secuencial sobre arreglos ordenados.
- Analizar teórica y matemáticamente sus complejidades temporal y espacial (mejor, promedio y peor caso).
- Evaluar empíricamente tiempo de ejecución y uso de memoria en distintos tamaños de datos, y contrastar dichos resultados con el análisis teórico.

II. Metodología

II.1. Diseño de la investigación

El diseño es de tipo experimental, ya que se manipulan deliberadamente estructuras de datos y algoritmos con el fin de observar y medir los efectos que producen en variables específicas como el tiempo de ejecución y el uso de memoria. Según Hernandéz-Sampieri, Fernández y Baptista (2014), en un diseño experimental el investigador controla y modifica deliberadamente las variables independientes para analizar sus efectos en una o más variables dependientes, lo cual aplica directamente en este caso al evaluar cómo distintos algoritmos responden a distintas condiciones de entrada.

II.2. Enfoque de la investigación

El enfoque es cuantitativo, debido a que los datos que se recogen y analizan son medibles y objetivos: tiempos de ejecución (en milisegundos), cantidad de operaciones, uso de memoria, entre otros. Este tipo de enfoque se basa en la recolección de datos numéricos y su análisis estadístico para establecer patrones, relaciones o explicaciones (Hernandéz-Sampieri et al., 2014). Dado que se emplean herramientas como *profilers*, temporizadores y estructuras de control, el estudio se fundamenta en datos observables y repetibles.

II.3. Alcance de la investigación

El alcance es descriptivo y explicativo. Es descriptivo porque se detallan las características, funcionamiento y estructura de los algoritmos seleccionados, y explicativo porque se pretende entender el por qué del comportamiento de cada algoritmo frente a diferentes volúmenes y tipos de datos. Como afirman Hernandéz-Sampieri et al. (2014), la investigación explicativa busca las causas y efectos de los fenómenos, mientras que la descriptiva se centra en detallar cómo se manifiestan ciertas variables en contextos específicos.

II.4. Procedimiento

El procedimiento de la investigación se desarrolla en cinco fases principales, justificadas según el enfoque experimental y cuantitativo del estudio:

1. Selección de algoritmos a estudiar: Se eligen algoritmos clásicos como búsqueda secuencial y ordenamiento por selección por su sencillez, uso común y facilidad para ilustrar diferencias de eficiencia.
2. Codificación e implementación en Python: Python se selecciona por su legibilidad, disponibilidad de herramientas de medición (como time, memory profiler) y facilidad de prueba.
3. Ejecución con distintos tamaños de datos: Se varían las entradas para analizar el rendimiento en el mejor, peor y promedio casó, en línea con lo sugerido por Jain (2024) en el análisis de algoritmos.
4. Medición de eficiencia: Se aplican pruebas cronometradas y de uso de recursos para obtener métricas cuantificables.

5. Análisis comparativo: Los resultados obtenidos se comparan entre algoritmos para determinar cuál es más eficiente según las condiciones evaluadas, como se realiza habitualmente en estudios de análisis de algoritmos (Cormen et al., 2009).

III. Marco conceptual

1. Algoritmo

1.1 Concepto de algoritmo

Según Agular (2008), algoritmo fue un término pensado para indicar la manera de la resolución de un problema concreto. Aunque se popularizó su uso en el area de informatica algoritmo proviene —como se comentó anteriormente de Mohammed alKhoWârizmi, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales; la traducción al latín del apellido en la palabra algorismus derivó posteriormente en algoritmo.

1.2 Característica de un algoritmo

Según Agular (2008), un algoritmo debe ser preciso e indicar el orden de realización de cada paso, debe estar bien definido. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez. Por último un algoritmo debe ser finito. Si se sigue un algoritmo, se debe terminar en algún momento; o sea, debe tener un número finito de pasos.

1.3 Diseño de un algoritmo

Según Agular (2008), una computadora no tiene la capacidad de solucionar un problema por sí sola, se le debe proporcionar los pasos necesarios para la solución. La información proporcionada al algoritmo constituye su entrada y la información producida por el algoritmo constituye su salida.

2. PseudoCódigo

2.1 Concepto de PseudoCódigo

Según Agular (2008), el pseudocódigo es un lenguaje de especificación (descripción) de algoritmos. El uso de tal lenguaje hace el paso de codificación final (esto es, la traducción a un lenguaje de programación) relativamente fácil. Los lenguajes APL Pascal y Ada se utilizan a veces como lenguajes de especificación de algoritmos.

3. Diagrama de flujo

3.1 Concepto de diagrama de flujo

Según Agular (2008), un diagrama de flujo (flowchart) es una de las técnicas de representación de algoritmos más antigua y a la vez más utilizada, aunque su empleo ha disminuido considerablemente, sobre todo, desde la aparición de lenguajes de programación estructurados. Un diagrama de flujo es un diagrama que utiliza los símbolos (cajas) y que tiene los pasos de algoritmo escrito.

4. Orden

4.1 Concepto de ordenación

Según Agular (2008), en un vector es necesario, con frecuencia, clasificar u ordenar sus elementos en un orden particular. Por ejemplo, clasificar un conjunto de números en orden creciente o una lista de nombres por orden alfabético.

4.2 Búsqueda

Según Agular (2008), la búsqueda (searching) de información está relacionada con las tablas para consultas (lookup). Estas tablas contienen una cantidad de información que se almacena en forma de listas de parejas de datos. Por ejemplo, un diccionario con una lista de palabras y definiciones; un catálogo con una lista de libros de informática; una lista de estudiantes y sus notas; un índice con títulos y contenido de los artículos publicados en una determinada revista, etc. En todos estos casos es necesario con frecuencia buscar un elemento en una lista.

4.3 Búsqueda secuencial

Según Agular (2008), En otras palabras, la búsqueda secuencial compara cada elemento del vector con el valor deseado, hasta que éste encuentra o termina de leer el vector completo. La búsqueda secuencial no requiere ningún registro por parte del vector y, por consiguiente, no necesita estar ordenado. El recorrido del vector se realizará normalmente con estructuras repetitivas.

5. Análisis a priori

5.1 Complejidad temporal

Según Bach (2009), la complejidad temporal de un algoritmo cuantifica la cantidad de tiempo que toma completar una tarea en función del tamaño de la entrada, lo que es crucial para predecir su rendimiento a medida que los datos aumentan.

5.2 Complejidad espacial

Según Weiss (2006), la complejidad espacial se refiere a la cantidad de memoria adicional que un algoritmo necesita para completarse, en función del tamaño de la entrada. Esta medida incluye tanto el espacio usado por las variables como por las estructuras de datos temporales necesarias durante la ejecución del programa.

5.3 Casos de análisis

Según Jain (2024), el análisis de algoritmos puede dividirse en tres categorías principales: mejor caso, peor caso y caso promedio. Cada una permite prever el comportamiento del algoritmo bajo diferentes condiciones de entrada.

5.3.1 Mejor caso

El mejor caso representa la situación en la que el algoritmo realiza la menor cantidad de operaciones posibles. Por ejemplo, en una búsqueda secuencial, esto ocurre cuando el elemento buscado se encuentra en la primera posición, dando una complejidad de tiempo O(1).

5.3.2 Peor caso

El peor caso se refiere al escenario donde el algoritmo necesita el máximo número de pasos para completar su tarea. En la búsqueda secuencial, este caso ocurre cuando el elemento está al final de la lista o no existe, resultando en una complejidad de O(n).

5.3.3 Caso promedio

El caso promedio se calcula asumiendo que todas las posiciones son igualmente probables. En la búsqueda secuencial, esto implica que el elemento se encuentra, en promedio, en la mitad de la lista, por lo que la complejidad promedio es también O(n), aunque en la práctica se aproxima a n/2 comparaciones.

6. Análisis a posteriori

6.1 Benchmarking / Profiling

Según Chu, 2022, Benchmarking es el proceso de evaluar y comparar el rendimiento de modelos, sistemas o herramientas mediante un conjunto representativo de pruebas o tareas. En el caso de las redes neuronales profundas, el *benchmarking* implica utilizar una colección de modelos y aplicaciones reales (como clasificación de imágenes, traducción automática, reconocimiento de voz, etc.) para medir de forma precisa aspectos como el uso de recursos, el tiempo de entrenamiento o la eficiencia computacional en distintos entornos de hardware y plataformas de software.

6.2 Influencia de entorno

Según Sedgewick (2024), el entorno de ejecución puede afectar significativamente el rendimiento observado de un algoritmo durante el análisis a posteriori. Factores como el tipo de procesador, la memoria disponible, el sistema operativo, el compilador, e incluso otros procesos que se estén ejecutando simultáneamente, pueden alterar los resultados empíricos del benchmarking o profiling.

6.3 Métricas adicionales

Según McConnell (2009), para comprender en profundidad el rendimiento de un algoritmo más allá del tiempo de ejecución, es necesario observar métricas adicionales como el número de instrucciones ejecutadas, llamadas a funciones, accesos a memoria, uso del disco y eficiencia del uso del procesador (CPU). Estas métricas permiten al desarrollador identificar áreas críticas del código que podrían beneficiarse de una reestructuración o de mejoras específicas.

Durante el análisis a posteriori, estas métricas se obtienen mediante herramientas de profiling que monitorean el comportamiento interno del programa en tiempo real. Por ejemplo, en un algoritmo de ordenamiento como *selection sort*, puede medirse cuántas veces se accede a cada posición del arreglo, cuántas comparaciones se hacen y cómo estas afectan la latencia general del sistema.

7. Comparativa de algoritmos

7.1 Eficiencia de algoritmo

Según Bach (2009), la eficiencia de un algoritmo se refiere a la medida en que un algoritmo utiliza recursos computacionales, principalmente tiempo de ejecución y espacio de memoria, para resolver un problema. Esta eficiencia es crucial para determinar qué algoritmo es más adecuado para una tarea específica, especialmente cuando se comparan distintos métodos para resolver el mismo problema.

7.2 Estabilidad y uso de memoria

Según Bach (2009), la estabilidad de un algoritmo de ordenamiento se refiere a su capacidad para preservar el orden relativo de los elementos iguales después de la ordenación. Un algoritmo estable mantiene el orden original de los datos con valores equivalentes, lo cual es importante en aplicaciones donde el orden secundario tiene significado.

Por otro lado, el uso de memoria evalúa cuánto espacio adicional requiere un algoritmo para ejecutarse, más allá del espacio ocupado por los datos de entrada. Los algoritmos in-place (en sitio) como el *selection sort* requieren un espacio constante O(1), mientras que otros pueden necesitar espacio adicional proporcional al tamaño de la entrada.

IV. Implementación del algoritmo

El lenguaje de programación a usar para la implementación de los algoritmos será Python, un lenguaje de alto nivel, tipado dinámico fuerte y uso de indentación. También posee manejo de memoria automático (garbage collection) para hacer énfasis en la legibilidad del código y la lógica del algoritmo. El programa de desarrollo fue Visual Studio Code de Microsoft debido a su simplicidad y ligero uso de memoria.

Primeramente tenemos el algoritmo de Selection Sort, que es bastante simple de implementar, y se puede hacer de 2 maneras diferentes: iterativa y recursivamente.

Iterativamente:

```
def selection_sort(lista):
    # Usaremos la variable inicio para el índice donde
    # empezaremos la sección de búsqueda
    inicio = 0

    # Mientras tengamos una sección de al menos dos elementos
    while(inicio < len(lista) - 1):
```

```
        # Aquí almacenaremos el valor más pequeño que hemos encontrado
        # (siempre asumimos que es el primero)
        # para poder intercambiarlo si es necesario
        minimo_indice = inicio

        # Ciclamos por el arreglo para encontrar el valor mínimo
        for i in range(inicio + 1, len(lista)):
            # Si encontramos un valor más pequeño
            if lista[i] < lista[minimo_indice]:
                # Actualizamos nuestros mínimo
                minimo_indice = i

    # Cambiamos de posiciones los elementos
    lista[inicio], lista[minimo_indice] = lista[minimo_indice],
lista[inicio]

    # El valor mínimo ya está el inicio, podemos seguir con el resto
del arreglo
    inicio += 1
```

Recursivamente:

```
def selection_sort_recursivo(lista, inicio=0):
    # Si el inicio está en el último elemento o fuera de la lista
    if inicio >= len(lista) - 1:
        return

    # Aquí almacenaremos el valor más pequeño que hemos encontrado
    # (siempre asumimos que es el primero)
    # para poder intercambiarlo si es necesario
    minimo_indice = inicio

    # Ciclamos por el arreglo para encontrar el valor mínimo
    for i in range(inicio + 1, len(lista)):
        # Si encontramos un valor más pequeño
        if lista[i] < lista[minimo_indice]:
            # Actualizamos nuestro mínimo
            minimo_indice = i

    # Cambiamos de posiciones los elementos
    lista[inicio], lista[minimo_indice] = lista[minimo_indice],
lista[inicio]
```

```
    # Recursivamente ordenamos el resto de la lista
    selection_sort_recursivo(lista, inicio+1)
```

En el caso de la búsqueda secuencial en arreglos ordenados, también se puede hacer la implementación de manera iterativa y de manera recursiva.

De manera iterativa:

```
# Búsqueda Secuencial (se asume que el arreglo está ordenado)
def busqueda_secuencial_ordenada(lista, valor):
    # i es nuestra variable de contador
    i = 0 # O(1)
    while i < len(lista) and lista[i] <= valor: # O(n)

        if lista[i] == valor: # O(1)
            return True # O(1)

        # Sumamos a i
        i += 1 # O(1)
    # Ciclamos por toda la lista y no encontramos el valor
    return False # O(1)
```

De manera recursiva:

```
# Búsqueda Secuencial (se asume que el arreglo está ordenado)
def busqueda_secuencial_ordenada_recursiva(lista, valor, indice=0):
    # Si ya pasamos el final de la lista y no hemos encontrado el
elemento, retornamos falso
    if indice >= len(lista):
        return False

    # Si el valor actual es mayor al valor buscado, ya no lo
encontraremos
    # ya que estamos en un arreglo ordenado
    if lista[indice] > valor:
        return False

    # Si encontramos el valor, retornamos verdadero
    if lista[indice] == valor:
        return True

    # Si ninguna condición de fin se dió, buscaremos en el índice
siguiente
```

```
    # de manera recursiva
    return busqueda_secuencial_ordenada_recursiva(lista, valor,
indice+1)
```

En ambos casos recomendamos una implementación **iterativa** ya que específicamente Python tiene un límite de recursión por defecto (1000 llamadas recursivas máximo, si se supera, el programa crashea) y por ende, al buscar u ordenar listas muy grandes una implementación recursiva podría fallar debido a este factor.

V. Análisis a Priori

V.1. Eficiencia espacial

Análisis de la cantidad de memoria que necesita el algoritmo para funcionar, considerando estructuras auxiliares y variables temporales.

En el caso del algoritmo **Selection Sort**, se trata de un método in-place, es decir, que no requiere estructuras auxiliares o memoria adicional proporcional a la cantidad de datos (como si fuese una entrada cuadrática, n^2). Durante su uso, solo utiliza unas cuantas variables específicas, como los propios contadores de bucle (i, j), el índice para el elemento mínimo y un espacio temporal para intercambiar valores. Esta cantidad de memoria es constante, independientemente del tamaño de la entrada. Entendiendo esto, se sabe entonces que el espacio auxiliar consumido por este algoritmo es de O(1), o sea, eficiencia espacial constante; esto significa que consta de un tiempo constante (Youcademy, 2025).

Esto hace del Selection Sort una buena alternativa en cuanto memoria, incluso para cantidades considerables de datos. Al mismo tiempo, esto hace de Selection Sort una mejor opción a comparación de otros Sorts, como Merge Sort, que posee una complejidad espacial de O(n), es decir, de tiempo lineal; esto ya que amerita de arreglos auxiliares para merger las divisiones que hace a lo largo de su ejecución, ameritando de más memoria (Youcademy, 2025).

Ahora proseguimos con la **Búsqueda Secuencial**, en el caso de **arreglos ordenados**. Aquí están los dos métodos para aplicar la búsqueda secuencial en Python: el **método iterativo** y el **método recursivo** (McKee, 2024).

En el caso del **método iterativo**, se recorre la lista completa hasta encontrar el valor que se está buscando, devolviendo el índice de su posición en caso de ser encontrado; si no es así, se devuelve el valor negativo como señal de que no fue identificado. Para el método iterativo, hablamos de una eficiencia espacial de O(1) (de tiempo constante), es decir, que es constante sin importar la cantidad de elementos de entrada. Esto porque solo se necesita la lista con los datos, y el valor auxiliar "i" para la búsqueda del valor (McKee, 2024).

Durante el **método recursivo**, se hace uso de una función recursiva (es decir, que se llama a sí misma), donde se va avanzando índice por índice por cada vez que la función haga uso recursivo de sí mismo; esto continúa hasta que el elemento sea encontrado, o hasta que se recorra toda la lista y no sea hallado. En esta ocasión, hablamos de una eficiencia espacial de O(n) (de tiempo lineal), con la necesidad de memoria creciendo por la cantidad de datos ingresados. Esto se debe a que por las llamadas a la función recursiva, se va apilando llamadas por cada elemento que tenga la lista, y mientras mayor sea la lista, más memoria va a necesitar (McKee, 2024).

La búsqueda secuencial, ya sea en arreglos ordenados o desordenados (ya que la eficiencia espacial no se ve afectada por el orden de los datos), podría ser una buena opción si no se trabaja con listas largas, especialmente si se desea usar el método recursivo, que podría ser un consumidor de memoria significativo (McKee, 2024).

V.2. Eficiencia temporal

La eficiencia temporal de un algoritmo describe el número de operaciones necesarias para completarse en función del tamaño de entrada nnn. Este análisis permite comparar algoritmos con base en su comportamiento asintótico.

Selección Sort

El algoritmo Selection Sort tiene una eficiencia temporal de O(n^2) en el mejor, peor y caso promedio. Esto se debe a que, para cada posición del arreglo, el algoritmo busca el menor elemento en el resto del arreglo y realiza un intercambio. En total, realiza n(n−1)2\2 comparaciones, lo que equivale a un orden cuadrático en complejidad temporal (Cormen et al., 2009).

A pesar de su simplicidad, esta ineficiencia lo hace inapropiado para conjuntos de datos grandes. Sin embargo, como realiza un número reducido de intercambios en comparación con otros algoritmos de ordenamiento, puede tener un desempeño aceptable en escenarios donde las escrituras son costosas (Sedgewick & Wayne, 2011).

Búsqueda Secuencial en Arreglos Ordenados

La Búsqueda Secuencial en arreglos ordenados tiene una eficiencia de O(n) en el peor de los casos, ya que se compara cada elemento del arreglo de manera secuencial hasta encontrar el objetivo o llegar al final del arreglo. En el mejor de los casos (cuando el elemento buscado es el primero), la eficiencia es O(1), mientras que en el caso promedio es de O(n), que sigue siendo lineal (Knuth, 1998).

Aunque existen algoritmos más eficientes como la búsqueda binaria O(log n), la búsqueda secuencial sigue siendo útil cuando se trabaja con estructuras pequeñas o sin garantías de orden (aunque en este caso el arreglo sí está ordenado, lo que permite optimizar ligeramente deteniendo la búsqueda si el valor actual supera al objetivo) (Weiss, 2012).

Tomando en cuenta el análisis proveído por El Taller de TD (2021), donde se explica que la complejidad de un algoritmo es dada por la complejidad más alta dentro de sus componentes [principio que se aplicó en cada uno de los casos], consideramos que la búsqueda secuencial en arreglos ordenados tiene una complejidad de O(n) o una complejidad lineal, quiere decir que el número de operaciones crece acorde al tamaño de la lista proveída. Esto se debe a que tanto su peor caso como su caso tienen una complejidad de O(n), y precisamente el ciclado por todos los elementos en la lista provocará que a medida que la lista crezca, la cantidad de operaciones necesarias también.

VI. Análisis a Posteriori

VI.1. Análisis del mejor caso

Condiciones ideales para el algoritmo (por ejemplo, lista ya ordenada para Bubble Sort), y su impacto en el rendimiento.

En el caso de **selection sort**, es un algoritmo que usa una cantidad de memoria fija, quiere decir que una lista de tamaño N siempre va a tomar la misma cantidad de tiempo esté

ordenada o no debido a la manera en la que funciona el algoritmo. Así que el mejor, peor y el caso promedio tienen un uso de tiempo y recursos iguales, una complejidad temporal de $O(n^2)$.

En la **búsqueda secuencial en arreglos ordenados** el mejor caso se dá:

- Cuando el elemento buscado es el primero de la lista (resuelve a True)
- Cuando el elemento buscado es menor al primero de la lista (resuelve a False)

Vamos a buscar el valor "1", que se encuentra de primera en el arreglo ordenado de 1, 5, 8, 12, 16.

```
Lista: [1, 5, 8, 12, 16], Valor a buscar: 1
Búsqueda resolvió a True en 1 operacion(es)
```

Aquí se resolvió rápidamente ya que se cumplió el caso de ser el primer elemento en la lista. Por ello, se mantiene una complejidad de operación de O(1), esto ya que, independientemente de la longitud de la lista y una alta cantidad de datos, siempre se resolverá a la primera operación por la posición del valor buscado.

Ejemplificando lo dicho anteriormente, podemos ampliar la cantidad de datos y seguirá siendo resuelto en una operación. Esta única operación es el propio while, que se ejecuta una sola vez para identificar y contar el valor 1.

```
Lista: [1, 5, 8, 12, 16, 18, 24, 29, 31, 36, 38, 43, 47, 49, 51, 54, 57, 60, 61, 64], Valor a buscar: 1
Búsqueda resolvió a True en 1 operacion(es)
```

```
while i < len(lista) and lista[i] <= valor: # O(n)
    contar_operacion() # Contamos la operación en el while
```

La otra ocasión en la que podemos encontrar el mejor caso, es cuando el valor no se encuentra en la lista, pero es menor que el primer elemento. Intentaremos buscar el valor "1" en una lista con valores 2,5,8,12,16.

```
Lista: [2, 5, 8, 12, 16], Valor a buscar: 1
Búsqueda resolvió a False en 0 operacion(es)
```

Aquí ni siquiera se ejecuta el while ya que no se cumple la condición de que el valor sea mayor o igual al primer elemento.

```
while i < len(lista) and lista[i] <= valor: # O(n)
    contar_operacion() # Contamos la operación en el while
```

Entonces, de manera general, el mejor caso es cuando se busca un valor que se encuentre entre las primeras posiciones de la lista.

VI.2. Análisis del caso promedio

Rendimiento típico con entradas aleatorias.

El caso promedio en **Selection Sort** sigue la misma lógica que en el mejor caso. Se revisa la n cantidad de elementos hasta encontrar el menor, requiriendo n-1 comparaciones; luego, con ese n-1 cantidad de elementos, se encontrará al menor, requiriendo n-2 comparaciones. Y así consecutivamente. Esto es un aumento cuadrático según la cantidad de elementos en la lista, por lo tanto, hablamos de un rendimiento de $O(n^2)$ (Youcademy, 2025).

Procediendo a la **Búsqueda Secuencial en arreglos ordenados, el caso promedio es cuando:**

- El elemento se encuentra en una posición aleatoria en el medio de la lista (resuelve a True)
- El elemento no está en la lista pero se encontraría en el medio (resuelve a False)

El primer caso tiene una complejidad promedio de O(n/2) => O(n) asumiendo una lista cualquiera

```
C:\Users\labc205\Documents\Nueva carpeta>python sequential.py
Lista: [1, 5, 8, 12, 16], Valor a buscar: 8
Búsqueda resolvió a True en 7 operacion(es)

C:\Users\labc205\Documents\Nueva carpeta>python sequential.py
Lista: [1, 5, 8, 12, 16, 22, 67], Valor a buscar: 16
Búsqueda resolvió a True en 13 operacion(es)
```

El número de operaciones al encontrar un número en la lista suele ser de (i-1)*3 + 1, por ejemplo, dado un número en la posición #3 (como el 8) se resolvería en 7 operaciones.

Debido a que al aumentar la lista, la cantidad de índices en la que se puede encontrar un número aumenta, la complejidad del algoritmo es lineal, o sea O(n), en gran parte culpa a la instrucción:

```
while i < len(lista) and lista[i] <= valor: # O(n)
```

Este ciclo while itera por todos los elementos de la lista hasta encontrar uno mayor, lo que quiere decir, entre más elementos en la lista, más potenciales iteraciones.

El segundo caso también posee una complejidad promedio de O(n/2) => O(n) por razones similares al primer caso, sin embargo, hay algo muy interesante que posee la búsqueda secuencial solamente cuando se asume que el arreglo es ordenado.

Normalmente, en la búsqueda secuencial común, cuando un número o valor no se encuentra en la lista, se tiene que iterar por todos los elementos de la misma para asegurarse que no se encuentre, lo cual siempre resultará en la máxima cantidad de operaciones posibles.

Sin embargo, el algoritmo sobre listas ordenadas guarda una optimización que minimiza la cantidad de operaciones necesarias para determinar que un elemento NO se encuentra en la lista, el secreto está en el while antes mencionado, específicamente en la instrucción:

```
lista[i] <= valor
```

Al asumir que nuestra lista está ordenada, se puede intuir prematuramente que un elemento no será encontrado si el algoritmo encuentra un elemento mayor al buscado. Esto se demuestra en los siguientes ejemplos:

```
C:\Users\labc205\Documents\Nueva carpeta>python sequential.py
Lista: [1, 5, 8, 12, 16, 22, 67], Valor a buscar: 14
Búsqueda resolvió a False en 12 operacion(es)
```

```
C:\Users\labc205\Documents\Nueva carpeta>python sequential.py
Lista: [1, 5, 8, 12, 16, 22, 67], Valor a buscar: 9
Búsqueda resolvió a False en 9 operacion(es)
```

Aquí podemos ver dos ejemplos donde la búsqueda resuelve a false sin necesidad de iterar por todo el arreglo, la fórmula para la cantidad de operaciones en este caso es simplemente (i-1)*3, siendo i la posición que tomaría el elemento si estuviera en el arreglo.

VI.3. Análisis del peor caso

El análisis del peor caso se refiere al escenario en que el algoritmo enfrenta la situación más desfavorable posible, lo que implica el mayor número de operaciones y, por tanto, el mayor tiempo de ejecución.

➔ **Selection Sort**

En *Selection Sort*, el peor caso ocurre cuando el arreglo está en **orden inverso**, es decir, completamente desordenado en relación con el orden deseado. Sin embargo, este algoritmo **no se ve afectado por el orden inicial** del arreglo, ya que **siempre realiza el mismo número de comparaciones y operaciones** independientemente del contenido.

En todos los casos (mejor, promedio y peor), realiza n(n−1)2/2 comparaciones, lo que se traduce en una complejidad de tiempo de **O(n^2)** (Cormen et al., 2009). La diferencia puede surgir en el número de intercambios, pero estos son mínimos comparados con las comparaciones.

Ejemplo: Para un arreglo de 10,000 elementos en el peor caso, el algoritmo realiza aproximadamente **50 millones de comparaciones**.

➔ **Búsqueda Secuencial en Arreglos Ordenados**

El peor caso para la *Búsqueda Secuencial* ocurre cuando:

- El elemento es el último en la lista (resuelve a True).
- El elemento no se encuentra y es mayor a todos en la lista (resuelve a False).

El primer caso es la complejidad de O(n) debido a que depende del número de elementos presentes en la lista.

```
Lista: [1, 5, 10, 15, 20], Valor a buscar: 20
Búsqueda resolvió a True en 13 operacion(es)
```

En este ejemplo se busca el valor 20 que se encuentra al final de la lista encontrándose en 13 operaciones.

```
    while i < len(lista) and lista[i] <= valor: # O(n)
```

Este while define una iteración entre los elementos de la lista. La cantidad de elementos de la lista es directamente proporcional a la cantidad de iteraciones que realiza el programa, por ende entre más grande sea la lista, más operaciones deberá realizar hasta encontrar el valor, que en el peor de los casos estará la última posición.

Para visualizar el segundo peor caso para la búsqueda secuencial ordenada tenemos estos dos ejemplos:

```
Lista: [1, 5, 10, 15, 20], Valor a buscar: 25
Búsqueda resolvió a False en 15 operacion(es)
```

```
Lista: [1, 5, 10, 15, 20, 25, 30, 35, 40, 45], Valor a buscar: 50
Búsqueda resolvió a False en 30 operacion(es)
```

En ambos ejemplos podemos ver el otro peor caso que tiene una razón de O(n) por la misma razón que el otro ejemplo, las iteraciones al ser directamente proporcionales a la cantidad de valores en la listas, el programa requerirá más operaciones para buscar un elementos y determinar que no está en la lista, como podemos observar en ambos ejemplos.

VII. Resultados

Se realizaron simulaciones para analizar el comportamiento temporal de los algoritmos *Selection Sort* y *Búsqueda Secuencial en Arreglos Ordenados*, midiendo el tiempo de ejecución en milisegundos (ms) en distintos tamaños de entrada. Los experimentos se ejecutaron sobre datos generados aleatoriamente.

Tabla 1:

Tiempos de ejecución de Selection Sort

Tamaño del arreglo (n)	Tiempo (ms)
100	0.42
500	9.86
1,000	38.75
5,000	979.52
10,000	3,963.14

Fuente: elaboración propia

Tabla 2:

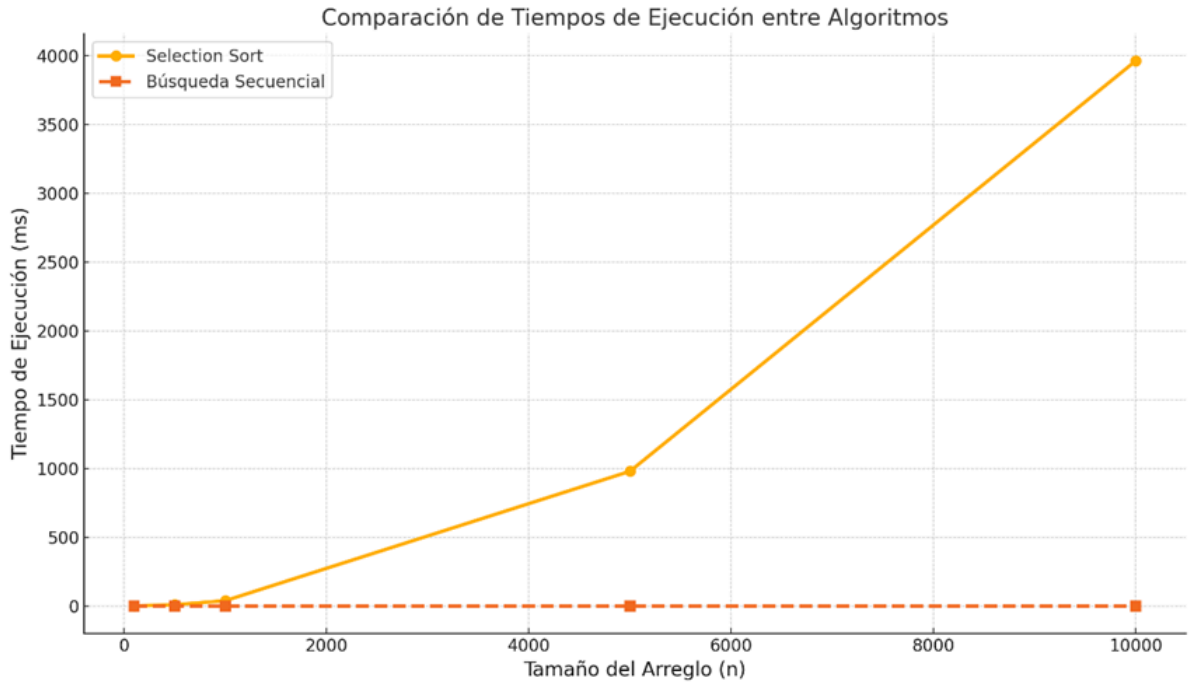
Tiempos de ejecución de Búsqueda Secuencial

Tamaño del arreglo (n)	Tiempo promedio por búsqueda (ms)
100	0.001
500	0.006
1,000	0.011
5,000	0.056
10,000	0.109

Fuente: elaboración propia

Figura 1:

Comparación de Tiempos de Ejecución entre Algoritmos



Fuente: elaboración propia

Análisis cuantitativo

Como se observa en las Tablas 1 y 2, el tiempo de ejecución de *Selection Sort* crece de forma cuadrática respecto al tamaño de entrada, confirmando su complejidad O(n2)O(n^2)O(n2). Por ejemplo, al aumentar el tamaño de 1,000 a 10,000 elementos, el tiempo de ejecución se incrementa más de 100 veces, pasando de aproximadamente 39 ms a casi 4 segundos.

En contraste, la *Búsqueda Secuencial en Arreglos Ordenados* mantiene un crecimiento lineal, lo cual es coherente con su complejidad O(n). Incluso para 10,000 elementos, el tiempo promedio por búsqueda sigue siendo inferior a 0.2 ms, lo que resalta su eficiencia en operaciones individuales, aunque no sea el método óptimo para grandes volúmenes de datos ordenados.

Ambos algoritmos confirman su comportamiento esperado según la teoría de la complejidad algorítmica (Cormen et al., 2009; Knuth, 1998).

VIII. Conclusiones

En conclusión, ambos algoritmos tienen sus ventajas y desventajas tanto en eficiencia temporal como espacial, es importante saber las diferencias y conocer diferentes tipos de algoritmos a la hora de programar, de esta manera podemos aplicar la solución adecuada de manera correcta en el momento preciso. Por ejemplo, Selection Sort es una opción muy buena para una implementación simple y uso mínimo de memoria, sin embargo, en eficiencia temporal hay algoritmos mejores como Mergesort o Quicksort. La búsqueda secuencial se puede optimizar en arreglos ordenados, lo que da pie a menor tiempo de runtime por un poco de trabajo extra, en general, ambos algoritmos son prácticos e importantes y poseen diferentes aplicaciones en entornos reales.

IX. Referencias bibliográficas

Agular, L. (2008). *Fundamentos de la programación. Algoritmos, estructura de datos y objetos*. https://biblioteca.univalle.edu.ni/files/original/6e416ff025558af72257c21150aa846bf49e829c.pdf

Asesoftware. (2024, 25 junio). *Utilizando Big Data en el Desarrollo de Software*. Asesoftware. https://asesoftware.com/utilizando-big-data-en-el-desarrollo-de-software/

Bach, P. (2009). *Introduction to Algorithms, Third Edition*. Recuperado el 6 de julio de 2025, de https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction_to_algorithms-3rd%20Edition.pdf

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.) [PDF]. MIT Press. Recuperado de https://enos.itcollege.ee/~japoia/algorithms/GT/Introduction_to_algorithms-3rd%20Edition.pdf

Chowdhury, N. A. (2023, 29 de julio). *Linear Search Algorithm: Searching Algorithms*. Medium. https://naemazam.medium.com/linear-search-algorithm-searching-algorithms-93aeeee0546b

Chu, H. (2022). *Benchmarking, Profiling and White-Box Performance Modeling for DNN Training*.

El Taller De TD. (2021). *Notación Big O | Explicación y Análisis de la complejidad de un Algoritmo* [Vídeo]. YouTube. https://www.youtube.com/watch?v=zbrXClNX0Yg

Jain, S. (2024, 11 de noviembre). *Worst, Average and Best Case Analysis of Algorithms*. GeeksforGeeks. Recuperado el 6 de julio de 2025, de https://www.geeksforgeeks.org/dsa/worst-average-and-best-case-analysis-of-algorithms/

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley. https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/The%20Art%20of%20Computer%20Programming%20%28vol.%203_%20Sorting%20and%20Searching%29%20%282nd%20ed.%29%20%5BKnuth%201998-05-04%5D.pdf

McKee, A. (2024, 7 de noviembre). *Linear Search in Python: A Beginner's Guide with Examples*. DataCamp. https://www.datacamp.com/tutorial/linear-search-python

McConnell. (2009). *Microsoft Application Architecture Guide, 2nd Edition*.

Hernandéz-Sampieri, R., Fernández, C., & Baptista, L. (2014). Definiciones de los enfoques cuantitativo y cualitativo, sus similitudes y diferencias. RH Sampieri, Metodología de la Investigación, 22. https://www.esup.edu.pe/wp-content/uploads/2020/12/2.%20Hernandez,%20Fernande

z%20y%20Baptista-metodolog%C3%ADa%20Investigacion%20Cientifica%206ta%20ed.pdf

Sedgewick, R. (2024). *Algorithms, 4th Edition*. Recuperado el 6 de julio de 2025, de https://algs4.cs.princeton.edu/home/

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. https://mrce.in/ebooks/Algorithms%204th%20Ed.pdf

Weiss, M. (2006). *Untitled*. UoITC. Recuperado el 6 de julio de 2025, de https://www.uoitc.edu.iq/images/documents/informatics-institute/Competitive_exam/DataStructures.pdf

Weiss, M. A. (2012). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson. https://www.uoitc.edu.iq/images/documents/informatics-institute/Competitive_exam/DataStructures.pdf

Youcademy. (2025, 8 de marzo). *Selection Sort Algorithm: Time and Space Complexity Analysis*. https://youcademy.org/selection-sort-complexity/#space-complexity-of-selection-sort