UNIVERSIDAD AMERICANA
Facultad de Ingeniería y Arquitectura

Algoritmo y Estructura de Datos

Análisis de efectividad de los métodos hashing y mergesort en el manejo de datos

Estudiante:

- Michael Casco
- Daniel Gutierrez
- Joshua ordoñez
- Diego Rodriguez

Docente:

- Ing. Silvia Gigdalia Ticay López

Julio 7, 2025.

Tabla de Contenido

I. Introducción

El presente trabajo de investigación se enfoca en el estudio y aplicación de dos algoritmos fundamentales en informática: el método de búsqueda Hashing y el algoritmo de ordenamiento Merge Sort. Ambos algoritmos son esenciales para optimizar la gestión y procesamiento de datos, ya que Hashing permite acceder rápidamente a la información mediante funciones hash que asignan claves a posiciones específicas, mientras que Mergesort garantiza un ordenamiento eficiente y estable de grandes volúmenes de datos a través de la técnica divide y vencerás. La combinación de estos métodos ofrece una solución robusta para sistemas que requieren rapidez y precisión en el manejo de la información.

En este contexto, se propone el desarrollo de un sistema de inventarios que utilice Hashing para la búsqueda ágil de productos y Mergesort para el ordenamiento eficiente de los registros. Este sistema permitirá mejorar la gestión de inventarios, facilitando operaciones como la consulta rápida de productos y la organización ordenada de los datos para reportes y análisis. A través de esta investigación, se analizarán las características, ventajas, limitaciones y aplicaciones prácticas de ambos algoritmos, así como su implementación y evaluación en el contexto específico del sistema de inventarios, demostrando su relevancia y efectividad en entornos reales.

A través de este estudio, se busca proporcionar una visión integral que permita comprender la relevancia de estos algoritmos en el ámbito de la informática, así como ofrecer recomendaciones sobre su uso adecuado según las necesidades específicas de cada aplicación. La correcta selección e implementación de estos algoritmos puede impactar de manera significativa en el rendimiento global de los sistemas, optimizando tanto el uso de recursos como la experiencia del usuario final.

1) Planteamiento del problema

Se centra en la necesidad de optimizar los procesos fundamentales de búsqueda y ordenamiento de datos en sistemas computacionales, dada la creciente cantidad y complejidad de la información que se maneja en la actualidad. Los algoritmos tradicionales de búsqueda y ordenamiento, aunque funcionales, presentan limitaciones en cuanto a eficiencia y escalabilidad cuando se enfrentan a grandes volúmenes de datos o a requerimientos de acceso rápido y confiable. Esto genera un impacto negativo en el rendimiento de aplicaciones que dependen de estas operaciones, provocando tiempos de respuesta lentos y un uso ineficiente de los recursos computacionales.

En este contexto, surge la importancia de estudiar y aplicar algoritmos más avanzados y eficientes como el Hashing para la búsqueda y el Merge Sort para el ordenamiento. Hashing ofrece un acceso casi inmediato a los datos mediante funciones hash que asignan claves a posiciones específicas, mientras que Mergesort garantiza un ordenamiento estable y con complejidad temporal óptima de O(nlogn), incluso en los peores casos. Sin embargo, a pesar de sus reconocidas ventajas teóricas, es necesario analizar en detalle su comportamiento práctico, sus limitaciones y las condiciones bajo las cuales su implementación resulta más beneficiosa. Por ello, esta investigación busca profundizar en el estudio de estos dos algoritmos clave, evaluando sus características y desempeño para facilitar su correcta aplicación en sistemas que requieran eficiencia en búsqueda y ordenamiento.

2) Objetivo de investigación

Analizar e implementar el funcionamiento, las características y la eficiencia de los algoritmos Hashing para búsqueda y Merge Sort para ordenamiento, así como su aplicación práctica en sistemas que requieren optimización en el manejo de datos.

3) Objetivos específicos

Implementar el algoritmo de búsqueda Hashing y el algoritmo de ordenamiento Mergesort en un sistema que permita evaluar su desempeño en operaciones de búsqueda y ordenamiento.

Evaluar la eficiencia temporal y espacial de Hashing y Merge Sort mediante análisis teóricos y pruebas prácticas.

Identificar las ventajas y limitaciones de Hashing y Mergesort para su aplicación en sistemas informáticos.

II. Metodología

1) Diseño de investigación

¿Qué es la investigación experimental?

La investigación experimental es el tipo de investigación científica que obtiene datos a través de la experimentación, para luego cotejar tales datos con variables a fin de determinar las relaciones de causa y efecto en los fenómenos estudiados. (Equipo de Enciclopedia Significados, 2024)

Las variables están sujetas a manipulación según los objetivos del estudio. El investigador manipula la variable independiente (controlada intencionalmente) para observar cómo afecta a la variable dependiente (en la que se observa el efecto de la manipulación). (Equipo de Enciclopedia Significados, 2024)

Por otro lado, los investigadores también deben controlar las variables constantes. Estas son las que se mantienen igual a lo largo de un estudio para garantizar que cualquier influencia observada sobre la variable dependiente se deba exclusivamente a la manipulación de la variable independiente. (Equipo de Enciclopedia Significados, 2024)

Por ejemplo, cuando los investigadores prueban el efecto de un fertilizante sobre las plantas de leguminosas. A un grupo de plantas se les administra fertilizante (manipulación conducida, variable independiente). Otro grupo, de control, no recibirá el fertilizante. Las variables constantes deben ser que ambos grupos reciban la misma cantidad de agua, luz, y estén a la misma temperatura y tipo de suelo. (Equipo de Enciclopedia Significados, 2024)

Esta investigación es experimental, en la cual se estará evaluando el rendimiento de los algoritmos Hashing frente a otros métodos de búsqueda mediante la medición de tiempos de ejecución y análisis de resultados.

De igual forma, se comparará su uso en conjunto al algoritmo Merge Sort, midiendo los tiempos de duración en los diferentes escenarios (Mejor caso, peor caso, caso promedio), para determinar el método más adecuado para su implementación en distintos sistemas para lograr una mayor optimización y tiempo de respuesta.

2) Enfoque de la investigación

¿Qué es el enfoque metodológico cuantitativo?

La metodología cuantitativa es una forma de investigación que se basa en la medición y análisis estadístico de datos numéricos. Su objetivo es obtener información objetiva y precisa sobre un fenómeno o problema, a través de la recolección sistemática y controlada de información. En este tipo de investigación, se utilizan técnicas de muestreo para seleccionar una muestra representativa de la población y se aplican instrumentos estandarizados para recolectar los datos. (Rodriguez, n.d.)

El enfoque metodológico es cuantitativo, ya que se estarán utilizando datos numéricos y medibles como los tiempos de ejecución y el uso de memoria, los cuales se utilizarán para realizar las conclusiones objetivas.

3) Alcance de la investigación

¿Que es el alcance de investigacion?

El alcance del estudio depende de la estrategia de investigación. Así, el diseño, los procedimientos y otros componentes del proceso serán distintos en estudio con alcance exploratorio, descriptivo, correlacional o explicativo. Pero en la práctica, cualquier investigación puede incluir elementos de más de una de estos cuatro alcances. (Investigadores, 2022)

Esta investigación tiene un alcance descriptivo, explicativo y experimental, centrado en la implementación práctica de los algoritmos Hashing y Merge Sort dentro de un sistema

de inventario desarrollado en Python. El objetivo no es comparar múltiples métodos, sino analizar el comportamiento específico de estos dos algoritmos cuando se aplican en escenarios reales de búsqueda y ordenamiento de datos.

El sistema desarrollado permitirá realizar operaciones de búsqueda rápida mediante Hashing y ordenamiento eficiente de registros mediante Merge Sort, utilizando estructuras y volúmenes de datos representativos de un entorno real de inventario. Se evaluará el rendimiento del sistema a través de mediciones cuantitativas como tiempo de ejecución y uso de memoria.

Este estudio no contempla la comparación con otros algoritmos, ni se enfoca en optimizaciones avanzadas. En cambio, se enfoca en comprender cómo funcionan Hashing y Merge Sort en conjunto, sus ventajas, limitaciones y su impacto en la eficiencia general del sistema. De este modo, el alcance queda limitado a un solo sistema funcional orientado al inventario, como caso de aplicación práctica de ambos algoritmos.

4) Procedimiento

Se elaborará un programa con Merge Sort con cada escenario (Mejor caso, peor caso, caso promedio), para cada programa se realizarán pruebas con diversas cantidades de datos (1,000, 10,000, 100,000, 1,000,000+), estos resultados serán comparados los resultados de los otros algoritmos de búsqueda.

También se implementará un programa de búsqueda Hashing y se probará con diferentes volúmenes de datos (ej. 1,000, 10,000, 100,000 registros). Las pruebas se compararán con otro método de búsqueda como la búsqueda lineal o binaria para observar el comportamiento del hashing.

Variable independiente:

La variable independiente está representada por los algoritmos utilizados en el sistema:

- Hashing para realizar búsquedas rápidas de productos mediante claves únicas.

- Merge Sort para ordenar de forma estable los registros del inventario.

Estos algoritmos son el foco principal del análisis, ya que se desea observar su impacto directo en el rendimiento del sistema.

Variable dependiente:

A. **Tiempo de ejecución**

Se analizará cuánto tarda el sistema en ejecutar búsquedas y ordenamientos con volúmenes de datos crecientes (1,000 a más de 100,000 registros). Este indicador permite evaluar la velocidad y escalabilidad de los algoritmos en escenarios reales.

B. **Tasa de éxito de búsqueda**

Se observará cuán efectiva es la búsqueda por claves mediante Hashing, especialmente en presencia de colisiones. Esto indicará la fiabilidad del algoritmo para recuperar correctamente la información.

C. Eficiencia General del sistema

Combinando los factores anteriores, se evaluará si el sistema responde de manera fluida, ordenada y precisa, incluso con grandes volúmenes de datos. Esta variable es clave para verificar la viabilidad práctica del uso conjunto de ambos algoritmos en una aplicación real.

D. Uso de memoria

Se medirá cuánta memoria consume el sistema durante los procesos de búsqueda y ordenamiento. Esto permitirá valorar la eficiencia espacial y detectar posibles excesos o cuellos de botella.

Instrumentos y herramientas

Se mantendrán constantes elementos como:

- El entorno de desarrollo (Python y Visual Studio Code),
- Las pruebas realizadas en un mismo equipo
- El diseño del sistema de inventario
- Las condiciones de entrada

Medición de eficiencias

Se utilizarán gráficos comparativos y medidas estadísticas básicas para interpretar los resultados de rendimiento.

III. Marco referencial

¿Qué es un Array?

"Un array es una estructura de datos que almacena una colección de elementos del mismo tipo, dispuestos en ubicaciones contiguas de memoria, y se accede a ellos mediante un índice numérico"

(Koffman & Wolfgang, 2011, p. 45).

¿Qué son algoritmos?

Un algoritmo es una serie de pasos claros, específicos y ordenados que nos ayudan a resolver un problema o realizar una tarea concreta. En el mundo de la informática, los algoritmos son clave para automatizar procesos y asegurarse de que los sistemas funcionen de manera eficiente. Es muy importante diseñarlos bien para obtener buenos resultados, tanto en rapidez como en uso de recursos.

¿Qué es el orden?

En los algoritmos, el orden se refiere a cómo organizamos los datos en un orden específico, ya sea de menor a mayor o viceversa. Esto ayuda a que las búsquedas, comparaciones y análisis sean mucho más rápidos y fáciles. Algoritmos de ordenamiento como Merge Sort o Quick Sort usan diferentes técnicas para ordenar los datos y, en realidad, son fundamentales en sistemas que manejan grandes cantidades de información.

¿Qué es el Análisis a priori?

El análisis a priori es una evaluación teórica que se realiza sobre un algoritmo antes de ponerlo en marcha. Consiste en estimar cómo se comportará en términos de rendimiento,

analizando cuánto tiempo y cuanta memoria puede necesitar, usando la notación Big O. Con esto, podemos entender cómo se comportará el algoritmo con diferentes tamaños de datos sin tener que correrlo realmente.

¿Qué es el Análisis a posteriori?

El análisis posterior a la ejecución, o análisis a posteriori, es cuando revisamos cómo salió un algoritmo después de usarlo. Se basa en medir qué tan bien funcionó en realidad, viendo cosas como cuánto tiempo tomó, cuánta memoria usó y cómo se comportó en diferentes situaciones (como en los casos mejor, promedio y peor). Esto ayuda a confirmar si las predicciones que hicimos antes, en el análisis a priori, eran correctas o si necesitamos ajustarlas.

¿Qué es la comparativa de algoritmos?

Comparar algoritmos es una forma de entender cuáles métodos funcionan mejor para resolver el mismo problema. Suele hacerse evaluando aspectos como qué tan rápido trabajan, si pueden adaptarse a diferentes tamaños de datos, qué tan estables son en distintos escenarios, qué tan fácil es usarlos y si funcionan bien en diferentes entornos. Hacer esta comparación ayuda a escoger el algoritmo que mejor se adapta a las necesidades específicas de cada sistema o proyecto.

¿Qué es un algoritmo de ordenamiento?

Según la conocida plataforma educativa en programacion FreeCodeCamp (2023) define un algoritmo de ordenamiento como: Los algoritmos de ordenación son un conjunto de instrucciones que toman un arreglo o lista como entrada y organizan los elementos en un orden particular.

Las ordenaciones suelen ser numéricas o una forma de orden alfabético (o lexicográfico), y pueden ser en orden ascendente (AZ, 0-9) o descendente (ZA, 9-0).

Ahora que entendemos que comprendemos que es un algoritmo de ordenamiento debemos de saber que existen diferentes tipos de algoritmos como: Selection Sort, Bubble Sort o Merge Sort. Pero específicamente

¿Qué es Notación Big O?

"La notación Big O describe el comportamiento de crecimiento del tiempo de ejecución de un algoritmo en función del tamaño de la entrada. Es una herramienta para expresar la complejidad algorítmica en el peor de los casos"
(Cormen et al., 2009, p. 25).

¿Qué es Merge Sort?

Según W3Schools (s.f) lo define como: "El algoritmo Merge Sort es un algoritmo de divide y vencerás que ordena una matriz dividiéndola primero en matrices más pequeñas y luego reconstruyendo la matriz de la manera correcta para que quede ordenada".

¿Qué es "Divide y Vencerás"?

según Frias, S (2021):

La metodología de divide y vencerás se conforma por tres procesos. Dividir: Descomponer el problema en sub-problemas del mismo tipo. Este paso involucra descomponer el problema original en pequeños sub-problemas. Cada sub-problema debe representar una parte del problema original. Por lo general, este paso emplea un enfoque recursivo para dividir el problema hasta que no es posible crear un sub-problema más.

Vencer: Resolver los sub-problemas recursivamente. Este paso recibe un gran conjunto de sub-problemas a ser resueltos. Generalmente a este nivel, los problemas se resuelven por sí solos. Y por ultimo Combinar: Combinar las respuestas apropiadamente. Cuando los sub-problemas son resueltos, esta fase los combina recursivamente hasta que estos formulan la solución al problema original. Este enfoque algorítmico trabaja recursivamente y los pasos de conquista y fusión trabajan tan a la par que parece un sólo paso.

¿Qué es Recursividad?

según Morales (s.f):

Se llama recursividad a un proceso mediante el que una función se llama a sí misma de forma repetida, hasta que se satisface alguna determinada condición. El proceso se utiliza para computaciones repetidas en las que cada acción se determina mediante un resultado anterior. Se pueden escribir de esta forma muchos problemas iterativos.

¿Qué es Merge (fusión)?

“La operación de merge combina dos secuencias ordenadas en una sola secuencia ordenada, lo cual es esencial en la segunda etapa del algoritmo Merge Sort” (Cormen et al., 2009, p. 35).

¿Qué es Complejidad Temporal?

Según Rivera, J. (2021):

La complejidad temporal es el número de operaciones que realiza un algoritmo para completar su tarea (considerando que cada operación dura el mismo tiempo). El algoritmo que realiza la tarea en el menor número de operaciones se considera el más eficiente en términos de complejidad temporal. Sin embargo, la complejidad espacial y temporal se ve afectada por factores como el sistema operativo y el hardware, pero no los incluiremos en discusión.

¿Qué es Complejidad Espacial?

Según la empresa de soluciones de comercio electrónico Engati (s.f):

La complejidad espacial es, en esencia, una medida de la cantidad total de memoria que los algoritmos u operaciones necesitan para ejecutarse según el tamaño de su entrada. Es, en esencia, la cantidad de espacio de memoria que el algoritmo requiere para resolver una instancia del problema computacional en función de las características de la entrada.

¿Qué es Hashing?

"Hashing es una técnica para almacenar y acceder a elementos usando una clave transformada por una función hash en un índice específico de una tabla"
(Tanenbaum & Bos, 2015, p. 129).

¿Qué es Tabla Hash?

"Una tabla hash es una estructura de datos que permite una búsqueda, inserción y eliminación eficientes mediante el uso de una función hash"
(Cormen et al., 2009, p. 256).

¿Que es una Función Hash?

"Una función hash convierte claves en índices de una tabla. Su efectividad depende de la distribución uniforme de las claves"
(Malpica et al., 2014, p. 143).

¿Qué es una colisión?

"Una colisión ocurre cuando dos claves diferentes son asignadas al mismo índice. Las colisiones deben ser gestionadas para garantizar la eficiencia del hashing"

(Koffman & Wolfgang, 2011, p. 210).

¿Qué son Resolución de Colisiones?

"Las colisiones pueden resolverse mediante técnicas como el encadenamiento (listas enlazadas en cada posición) o direccionamiento abierto"

(Tanenbaum & Bos, 2015, p. 131).

¿Qué es Factor de Carga?

"El factor de carga (load factor) afecta el rendimiento de una tabla hash. Es la razón entre el número de elementos y el tamaño de la tabla"

(Cormen et al., 2009, p. 260).

IV. Implementación de algoritmo

```python
import random
import time
from merge_sort import merge_sort
from busqueda_binaria import busqueda_binaria
from hash_busqueda import HashBusqueda

# Función: generar_producto_realista
# Crea un producto con nombre realista del supermercado,
# cantidad aleatoria entre 1 y 100, y código numérico único.

def generar_producto_realista(codigo_num):
    nombres_productos = [
        "Leche", "Pan", "Huevos", "Arroz", "Fideos", "Azúcar", "Aceite", "Sal",
        "Detergente", "Jabón", "Shampoo", "Galletas", "Cereal", "Queso", "Mantequilla",
        "Yogur", "Refresco", "Agua", "Papel Higiénico", "Servilletas", "Carne", "Pollo",
        "Pescado", "Manzanas", "Bananas", "Tomates", "Papas", "Cebolla", "Zanahorias"
    ]
    nombre = random.choice(nombres_productos)
    cantidad = random.randint(1, 100)
    return {'codigo': codigo_num, 'nombre': nombre, 'cantidad': cantidad}

# Función: guardar_inventarios_txt
# Guarda el inventario original y el inventario ordenado en un archivo de texto.
# El archivo se sobrescribe cada vez que se actualiza o visualiza.

def guardar_inventarios_txt(inventario_original, inventario_ordenado, nombre_archivo):
    with open(nombre_archivo, 'w', encoding='utf-8') as f:
        # Sección 1: inventario original
        f.write("INVENTARIO ORIGINAL:\n")
        for producto in inventario_original:
            f.write(f"{producto}\n")
        # Sección 2: inventario ordenado
        f.write("\nINVENTARIO ORDENADO POR CÓDIGO:\n")
        for producto in inventario_ordenado:
            f.write(f"{producto}\n")
    # Este archivo servirá como referencia visual para el usuario
```

Para la implementación de ambos algoritmos elaboramos un programa de inventario, muestra los datos mediante un archivo .txt donde muestra los productos antes y después de ser ordenados.

```
# Función: mostrar_desde_txt
# Lee y muestra el contenido del archivo de inventario al usuario.

def mostrar_desde_txt(nombre_archivo):
    try:
        with open(nombre_archivo, 'r', encoding='utf-8') as f:
            print(f.read())
    except FileNotFoundError:
        print("El archivo de inventario no existe aún.")

# Inicializamos
inventario = []
tabla_hash = HashBusqueda()

try:
    tam = int(input("Ingrese el tamaño del inventario: "))
except ValueError:
    print("Entrada inválida.")
    exit()
```

En esta parte del código se inicializa el inventario y el algoritmo de búsqueda.

```
print("\nSeleccione el tipo de caso:")
print("1. Mejor caso (ordenado)")
print("2. Caso promedio (aleatorio)")
print("3. Peor caso (orden inverso)")

try:
    caso = int(input("Ingrese una opción (1-3): "))
except ValueError:
    print("Entrada inválida.")
    exit()

# Crea los datos base
inventario = [generar_producto_realista(i + 1) for i in range(tam)]

# Ordena según el caso
if caso == 1:
    inventario.sort(key=lambda x: x['codigo'])                    # Mejor caso: ordenado
elif caso == 2:
    random.shuffle(inventario)                                    # Caso promedio: aleatorio
elif caso == 3:
    inventario.sort(key=lambda x: x['codigo'], reverse=True)      # Peor caso: orden inverso
else:
    print("Opción inválida. Se usará caso promedio por defecto.")
    random.shuffle(inventario)

# Guarda copia del inventario tal como fue generado
inventario_original = inventario.copy()

# Ordena con Merge Sort para uso posterior
inventario_ordenado = merge_sort(inventario.copy(), clave=lambda x: x['codigo'])


# Se guarda el inventario en un archivo de texto
guardar_inventarios_txt(inventario_original, inventario_ordenado, "inventario.txt")
print("Inventario generado y guardado en 'inventario.txt'.\n")
# Se trabaja con la versión ordenada por defecto
inventario = inventario_ordenado
```

Generamos los datos según el caso deseado y se crean los registros de cada producto, se guarda una copia del inventario original en el archivo .txt, se ordena y se guardó en el mismo archivo el inventario ya ordenado. Finalmente actualizamos para trabajar con el inventario ordenado.

```python
# Se cargan los productos en la tabla hash para búsqueda rápida
for prod in inventario:
    tabla_hash.insertar(prod['codigo'], prod)

while True:
    print("\nMenú:")
    print("1. Buscar producto")
    print("2. Visualizar inventario")
    print("3. Actualizar producto")
    print("4. Salir")

    try:
        opcion = int(input("Seleccione una opción: "))
    except ValueError:
        print("Entrada inválida.")
        continue

    # Opción 1: Buscar producto por código usando Hash y Binaria
    if opcion == 1:
        try:
            codigo = int(input("Ingrese el código del producto a buscar: "))
        except ValueError:
            print("Código inválido.")
            continue

        # Búsqueda por hash
        resultado_hash, tiempo_hash = tabla_hash.buscar(codigo)

        # Búsqueda binaria (sobre inventario ordenado)
        resultado_bin, tiempo_bin = busqueda_binaria(inventario, codigo, clave=lambda x: x['codigo'])
```

Continuamos con el menú, aquí nos permite buscar los productos, visualizar inventario mediante archivo .txt, actualizar producto y salir del programa. Cada opción está validada para no permitir números menores a 1 o mayores que 4. Asimismo nos muestra el resultado de la búsqueda tanto su tiempo de duración, en este caso lo compararemos con el algoritmo de búsqueda binaria.

```
        # Resultados comparativos
        print("\n--- Resultado de Búsqueda ---")
        print(f"Hashing:  {resultado_hash}")
        print(f"Tiempo (hash):    {tiempo_hash:.6f} segundos")
        print(f"Binaria:  {resultado_bin}")
        print(f"Tiempo (binaria): {tiempo_bin:.6f} segundos")

    # Opción 2: Visualizar el inventario desde el archivo .txt
    # También actualiza el archivo sobrescribiéndolo
    elif opcion == 2:
        guardar_inventarios_txt(inventario, merge_sort(inventario.copy(), clave=lambda x: x['codigo']), "inventario.txt")
        print("\n--- Visualización desde 'inventario.txt' ---")
        mostrar_desde_txt("inventario.txt")

    # Opción 3: Actualizar la cantidad de un producto existente
    elif opcion == 3:
        try:
            codigo = int(input("Ingrese el código del producto a actualizar: "))
        except ValueError:
            print("Código inválido.")
            continue

        encontrado = False
        # Recorre el inventario para buscar y actualizar
        for prod in inventario:
            if prod['codigo'] == codigo:
                try:
                    nueva_cantidad = int(input(f"Ingrese nueva cantidad para '{prod['nombre']}': "))
                except ValueError:
                    print("Cantidad inválida.")
                    break
                prod['cantidad'] = nueva_cantidad  # Actualiza la cantidad
                tabla_hash.insertar(prod['codigo'], prod)  # Actualiza también en hash
                encontrado = True
                break
```

Muestra los resultados de la búsqueda y su tiempo de duración según cada algoritmo.

Opción 2: visualización de inventario mediante archivo .txt; Opción 3: Ingresamos el código del producto que queremos actualizar, si el código no existe manda un mensaje de que el código es inválido, si existe se actualiza la cantidad, de igual forma se actualiza la tabla hash.

```
        if encontrado:
            print("Producto actualizado.")
            # El archivo .txt se actualiza con los nuevos datos
            guardar_inventarios_txt(inventario, merge_sort(inventario.copy(), clave=lambda x: x['codigo']), "inventario.txt")
        else:
            print("Producto no encontrado.")

    elif opcion == 4:
        print("Saliendo del programa.")
        break

    else:
        print("Opción no válida.")
```

Se actualiza el inventario en el archivo .txt; Opción 4: Cierra el programa, si no se ingresa cualquiera de esas opciones salta un mensaje de error.

Módulos del programa

hash_busqueda.py

```python
import time

class HashBusqueda:
    def __init__(self):
        self.tabla_hash = {}    # Crea una tabla hash vacía (diccionario)

    def insertar(self, clave, valor):
        if clave not in self.tabla_hash:    # Si la clave no existe aún
            self.tabla_hash[clave] = valor  # Inserta el valor con la clave dada
            return True     # Retorna True si se insertó
        return False    # Retorna False si ya existía la clave

    def buscar(self, clave):
        inicio = time.time()    # Marca el tiempo inicial
        valor = self.tabla_hash.get(clave, None)      # Busca el valor con la clave, o None si no está
        fin = time.time()
        tiempo = fin - inicio   # Calcula el tiempo que tardó en buscar
        return valor, tiempo

    def eliminar(self, clave):
        if clave in self.tabla_hash:    # Verifica que la clave existe en la tabla
            del self.tabla_hash[clave]   # Elimina el par clave-valor
            return True
        return False

    def mostrar_todo(self):
        if not self.tabla_hash:      # Si la tabla está vacía
            print("La tabla está vacía.")              # Muestra mensaje de tabla vacía
        else:
            print("Contenido de la tabla hash:")       # Título antes de listar los elementos
            for clave, valor in self.tabla_hash.items():  # Recorre cada par clave-valor
                print(f"Clave: {clave}, Valor: {valor}")  # Muestra la clave y su valor
```

Simula una tabla personalizada, permite insertar, buscar, eliminar y mostrar los productos mediante una clave. Además, mide el tiempo que tarda ejecutarse

```python
def busqueda_binaria(lista, objetivo, clave=lambda x: x):
    inicio = time.time()
    izquierda = 0
    derecha = len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        valor_medio = clave(lista[medio])
        if valor_medio == objetivo:
            fin = time.time()
            return lista[medio], fin - inicio
        elif valor_medio < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    fin = time.time()
    return None, fin - inicio
```

Esta función realiza una búsqueda binaria en una lista ordenada. Recibe tres parámetros: la lista, el objetivo a buscar y una clave opcional para acceder al valor que se debe comparar (por defecto, el mismo elemento). La función divide la lista en mitades hasta encontrar el valor o descartar, y además mide el tiempo que tarda en ejecutar la búsqueda. Si encuentra el objetivo, devuelve el elemento y el tiempo; si no, devuelve None y el tiempo total consumido.

Merge_sort.py

```python
def merge_sort(lista, clave=lambda x: x):
    # Si la lista tiene 0 o 1 elementos, ya está ordenada
    if len(lista) <= 1:
        return lista

    medio = len(lista) // 2     # Se calcula el punto medio de la lista para dividirla

    izquierda = merge_sort(lista[:medio], clave)    # Se aplica recursivamente merge_sort a la mitad izquierda

    derecha = merge_sort(lista[medio:], clave)    # Se aplica recursivamente merge_sort a la mitad derecha

    return fusionar(izquierda, derecha, clave)    # Se combinan las mitades ordenadas en una lista final

# Función que fusiona dos listas ordenadas en una sola lista ordenada
def fusionar(izquierda, derecha, clave):
    resultado = []  # Lista que almacenará el resultado final ordenado
    i = j = 0        # Índices para recorrer las listas izquierda y derecha

    # Se recorre mientras queden elementos en ambas listas
    while i < len(izquierda) and j < len(derecha):
        # Se compara el valor actual de cada lista usando la clave
        if clave(izquierda[i]) <= clave(derecha[j]):
            resultado.append(izquierda[i])  # Agrega el menor al resultado
            i += 1                          # Avanza en la lista izquierda
        else:
            resultado.append(derecha[j])   # Agrega el menor al resultado
            j += 1                          # Avanza en la lista derecha

    # Se agregan los elementos restantes
    resultado.extend(izquierda[i:])

    # Se agregan los elementos restantes
    resultado.extend(derecha[j:])

    # Retorna la lista completamente ordenada
    return resultado
```

Mediante este módulo se ordena el inventario, primeramente se divide entre dos mitades. Se aplica recursividad a la mitad izquierda y a la derecha. Una vez hecho esto se fusionan ambas mitades ordenadas en una lista final.

Método fusionar, crea una lista vacía y recorre las dos mitades. Compara los elementos usando la clave, si la clave izquierda es menor o igual que clave derecha agrega el menor al resultado y avanza la lista a la izquierda, de lo contrario agrega al de la derecha y avanza a la derecha. Para finalizar se agregan los elementos restantes a cualquier mitad y retorna la lista completamente ordenada.

V. Análisis a priori

Análisis a Priori Merge Sort

1) *Eficiencia espacial*

Merge Sort requiere espacio adicional para llevar a cabo el proceso de combinación (merge), ya que durante la ejecución se crean arreglos temporales para almacenar las mitades divididas y luego fusionarlas. A diferencia de otros algoritmos como Quicksort, que pueden operar en el mismo arreglo, Merge Sort no es un algoritmo in-place. Su complejidad espacial es de O(n), donde *n* representa el número total de elementos en el arreglo. Esto significa que para ordenar una lista de tamaño considerable, el algoritmo necesitará una cantidad equivalente de memoria auxiliar, lo que puede ser un factor limitante en sistemas con recursos reducidos o donde se procesan múltiples listas simultáneamente. Sin embargo, esta necesidad de memoria extra garantiza una mayor estabilidad y seguridad durante el proceso de ordenamiento.

2) *Eficiencia temporal*

Merge Sort se caracteriza por tener una complejidad temporal constante de O(n log n) en todos los casos posibles:

- Mejor caso, cuando el arreglo ya está ordenado,
- Peor caso, cuando está completamente desordenado,
- Caso promedio, cuando los elementos están en orden aleatorio.

Esta eficiencia se logra gracias a la estrategia divide y vencerás, en la cual:

- El arreglo se divide recursivamente en mitades hasta llegar a subarreglos de un solo elemento ($\log_2(n)$ divisiones),

- Luego, esos subarreglos se fusionan en orden, comparando elementos de manera secuencial (n comparaciones por nivel).

Esto da lugar a una estructura recursiva con una profundidad logarítmica y un costo lineal por nivel, generando así un comportamiento predecible y eficiente, independientemente del tipo de entrada. Esta regularidad convierte a Merge Sort en una excelente opción cuando se necesita estabilidad en el tiempo de ejecución, especialmente en sistemas donde los datos pueden llegar en cualquier orden.

3) Análisis del orden

Merge Sort es un algoritmo estable, lo cual significa que mantiene el orden relativo de los elementos con valores iguales. Esta característica es especialmente útil cuando se ordenan objetos con múltiples atributos y se desea conservar un criterio de ordenación secundaria.

Su rendimiento no se ve afectado significativamente por el orden inicial de los datos, lo que lo distingue de otros algoritmos como Quick Sort o Bubble Sort, cuyos tiempos de ejecución pueden variar drásticamente entre los mejores y peores casos.

Además, Merge Sort es altamente adecuado para estructuras como listas enlazadas y para procesamiento de archivos externos, ya que se puede implementar en forma adaptativa y multifase, dividiendo y fusionando datos en partes sin necesidad de cargarlos completamente en memoria.

Análisis a Priori Hashing

4) Eficiencia espacial

El análisis de la eficiencia espacial en un algoritmo de hashing evalúa cuánta memoria se necesita para almacenar y gestionar los datos, considerando las estructuras de datos auxiliares, las variables temporales y las técnicas empleadas para resolver colisiones. En su forma más básica, una tabla hash es la estructura clave de estos algoritmos y su tamaño está determinado por el número de elementos que se desean almacenar. Si almacenamos nnn elementos, la memoria utilizada por la tabla hash será proporcional a O(n⋅c)O(n \cdot c)O(n⋅c), donde ccc es el tamaño promedio de las claves y los valores asociados. Esto implica que, en el caso de claves y valores de tamaño constante, la memoria total se incrementa linealmente con el número de elementos.

Sin embargo, la eficiencia espacial también depende de cómo se manejan las colisiones. Cuando dos o más claves generan el mismo valor hash (lo que ocurre comúnmente debido a la limitación de los tamaños de las tablas), el algoritmo debe manejar estas colisiones, lo que puede requerir memoria adicional. Una técnica común de manejo de colisiones es el encadenamiento, donde cada celda de la tabla hash contiene una lista enlazada (o estructura similar) que guarda todas las claves que comparten el mismo valor hash. En este caso, el uso de memoria aumenta proporcionalmente con la cantidad de colisiones, ya que por cada celda de la tabla puede haber varias claves almacenadas en una lista. Si el número de colisiones es bajo, el espacio adicional es relativamente pequeño, pero si es alto, el costo en memoria puede crecer considerablemente.

Otra técnica común para manejar colisiones es la dirección abierta, en la que, en lugar de usar listas enlazadas, se buscan celdas vacías dentro de la propia tabla para almacenar las claves

en conflicto. Este enfoque puede reducir el uso de memoria adicional para las colisiones, pero su eficiencia depende de factores como el tamaño de la tabla y el factor de carga, es decir, la proporción de celdas ocupadas en la tabla. A medida que la tabla se llena, la eficiencia espacial y la velocidad de acceso pueden disminuir, y la tabla puede necesitar ser redimensionada para mantener un buen rendimiento.

5) Eficiencia temporal

```
Seleccione una opción (1-5): 2
Ingrese el código del producto a buscar: 1902
Producto encontrado: Azúcar #1902 - 75 unidades
Tiempo de búsqueda: 0.0000071526 segundos

--- MENÚ DE INVENTARIO ---
1. Agregar producto
2. Buscar producto
3. Eliminar producto
4. Mostrar inventario completo
5. Salir
Seleccione una opción (1-5): 2
Ingrese el código del producto a buscar: 9999
Producto encontrado: Carne #9999 - 32 unidades
Tiempo de búsqueda: 0.0000092983 segundos

--- MENÚ DE INVENTARIO ---
1. Agregar producto
2. Buscar producto
3. Eliminar producto
4. Mostrar inventario completo
5. Salir
Seleccione una opción (1-5): 2
Ingrese el código del producto a buscar: 5400
Producto encontrado: Carne #5400 - 76 unidades
Tiempo de búsqueda: 0.0000066757 segundos
```

Los tiempos de búsqueda observados son extremadamente bajos y muy similares entre sí. Las búsquedas se realizan en menos de un microsegundo (alrededor de 0.7 a 0.9 microsegundos), lo que indica un acceso casi instantáneo.

Estos resultados son consistentes con la teoría, ya que la búsqueda en una tabla hash tiene un tiempo constante O(1). Cuando se utiliza una buena función hash y se mantienen bajas las colisiones, el tiempo promedio de búsqueda es constante e independiente del tamaño de la tabla.

No se aprecia una diferencia significativa en los tiempos al buscar claves bajas, medias o altas. Por ejemplo, el tiempo para buscar productos con códigos bajos (1902), medios (5400) o cercanos al límite (9999) es muy parecido. Esto indica que la función hash distribuye bien las claves y que la estructura no presenta colisiones significativas en esas posiciones.

En cuanto a la escalabilidad, aunque el programa cuenta con 10,000 productos, la búsqueda mediante hashing debería mantener tiempos muy similares incluso si se incrementa considerablemente el número de elementos, siempre y cuando se mantenga una tabla hash de buen tamaño y el factor de carga sea controlado.

Es importante considerar que los tiempos tan bajos también dependen del entorno de ejecución, la máquina utilizada y el lenguaje Python. Sin embargo, para un programa simple que utiliza hashing en memoria, estos valores son normales y esperados.

En conclusión, el algoritmo de búsqueda basado en hashing implementado es muy eficiente para búsquedas en tablas grandes, con un tiempo promedio cercano a O(1). El tiempo medido confirma que la búsqueda es prácticamente instantánea para 10,000 elementos, y la estructura actual es adecuada para aplicaciones que requieran búsquedas rápidas y frecuentes.

Se recomienda mantener la tabla hash con un buen tamaño para evitar que el factor de carga aumente y degrade el rendimiento con el tiempo.

6) Análisis del orden

El algoritmo de hashing según la plataforma educativa Runestone (s.f): en general tiene una complejidad de O(1) en el mejor caso, lo que significa que la operación (ya sea una inserción, búsqueda o eliminación) se realiza en un tiempo constante, independientemente del tamaño de los datos. Esto es posible porque el proceso de hashing utiliza una función hash que mapea una clave a un índice en una tabla (también conocida como tabla hash). Cuando se accede a un dato, el algoritmo no necesita recorrer toda la estructura, sino que simplemente calcula el índice y accede directamente a esa posición.

Sin embargo, en el peor caso, si hay muchas colisiones (es decir, múltiples claves mapeadas al mismo índice), la complejidad puede llegar a O(n), donde n es el número de elementos almacenados, ya que las colisiones suelen resolverse mediante técnicas como encadenamiento o direccionamiento abierto. En estos casos, la operación podría requerir recorrer una lista de elementos en ese índice específico. Por eso, aunque el caso promedio y el mejor caso son O(1), el peor caso puede ser mucho peor dependiendo de la calidad de la función hash y el manejo de las colisiones.

VI. Análisis posteriori

Análisis Posteriori Merge Sort

1) Análisis del mejor caso

```
Tiempo de ordenamiento (Merge Sort): 5.539877 segundos
Inventario generado y guardado en 'inventario.txt'.


Menú:
1. Buscar producto
2. Visualizar inventario
3. Actualizar producto
4. Salir
Seleccione una opción: 1
Ingrese el código del producto a buscar: 500000

--- Resultado de Búsqueda ---
Hashing:  {'codigo': 500000, 'nombre': 'Servilletas', 'cantidad': 80}
Tiempo (hash):    0.000019 segundos
Binaria:  {'codigo': 500000, 'nombre': 'Servilletas', 'cantidad': 80}
Tiempo (binaria): 0.000262 segundos
```

Se buscó el producto con código 500000 usando dos métodos: hash y búsqueda binaria.

Ambos encontraron el producto correctamente.

La búsqueda hash fue más rápida (0.000019 s), ya que accede directo por clave. La búsqueda binaria tardó un poco más (0.000262 s), aunque sigue siendo rápida.

Sin embargo, la búsqueda binaria requiere que la lista esté ordenada previamente, y eso tomó 5.53 segundos con Merge Sort.

En resumen: hash es más rápido y no necesita ordenamiento. Binaria es útil, pero depende de que la lista ya esté ordenada.

2) Análisis del caso promedio

```
Tiempo de ordenamiento (Merge Sort): 12.695718 segundos
Inventario generado y guardado en 'inventario.txt'.


Menú:
1. Buscar producto
2. Visualizar inventario
3. Actualizar producto
4. Salir
Seleccione una opción: 1
Ingrese el código del producto a buscar: 750000

--- Resultado de Búsqueda ---
Hashing:  {'codigo': 750000, 'nombre': 'Pollo', 'cantidad': 11}
Tiempo (hash):    0.000005 segundos
Binaria:  {'codigo': 750000, 'nombre': 'Pollo', 'cantidad': 11}
Tiempo (binaria): 0.000014 segundos
```

Se buscó el producto con código 750000 usando búsqueda hash y búsqueda binaria. Ambos métodos funcionaron correctamente y devolvieron el mismo producto: 'Pollo' con cantidad 11.

La búsqueda hash fue más rápida (0.000005 segundos), mientras que la binaria tardó un poco más (0.000014 segundos). Hash sigue siendo la opción más eficiente.

El ordenamiento previo con Merge Sort tomó 12.69 segundos.

3) Análisis del peor caso

```
Tiempo de ordenamiento (Merge Sort): 5.064435 segundos
Inventario generado y guardado en 'inventario.txt'.


Menú:
1. Buscar producto
2. Visualizar inventario
3. Actualizar producto
4. Salir
Seleccione una opción: 1
Ingrese el código del producto a buscar: 250444

--- Resultado de Búsqueda ---
Hashing:  {'codigo': 250444, 'nombre': 'Agua', 'cantidad': 68}
Tiempo (hash):    0.000005 segundos
Binaria:  {'codigo': 250444, 'nombre': 'Agua', 'cantidad': 68}
Tiempo (binaria): 0.000027 segundos
```

Se buscó el producto con código 250444 usando búsqueda hash y búsqueda binaria. Ambos métodos funcionaron correctamente y devolvieron el mismo producto: 'Agua' con la cantidad 68.

La búsqueda hash fue más rápida (0.000005 segundos), mientras que la binaria tardó un poco más (0.000027 segundos). Hash sigue siendo la opción más eficiente. El ordenamiento previo con Merge Sort tomó 5.06 segundos.

Análisis Posteriori Hashing

Función hash (transformación de clave)

Operaciones clave: aplicar la función hash a la clave.

Ejemplo:

hash_index = hash (clave) % tamaño_tabla

Conteo de operaciones:

1) Acceder a los caracteres de la clave (si es una cadena): **O(m)**, donde *m* es la longitud de la clave.
2) Calcular valor hash (suma, multiplicación, etc.): depende del algoritmo, pero generalmente es **O(m)**.
3) Módulo con el tamaño de la tabla **O(1)**.

Total: **O(m)**

4) Análisis del Mejor Caso

La inserción sin colisiones es el caso ideal al implementar Hashing ya que no tenemos que buscar otros datos que coincidan con la clave hash.

Aplicar función hash: **O(m)**

Insertar en posición calculada: **O(1)**

Total: O(m + 1) ≈ **O(m)**

5) Análisis del Caso Promedio

Si la tabla no está muy llena (es decir, tiene una baja carga de factores y no hay muchas colisiones), el número de intentos que se requieren para encontrar una posición libre o una coincidencia es más reducido. El promedio de intentos de búsqueda sería mucho menor que nnn, y el costo de comparación sigue siendo O(m)O(m)O(m).

Total en el caso promedio:

O(m)O(m)O(m)

(donde O(m)O(m)O(m) es el costo de la comparación en cada intento, y en promedio se hacen pocas comparaciones).

Si las colisiones están distribuidas uniformemente, el tamaño de la lista enlazada en cada posición será pequeño, aproximadamente O(1)O(1)O(1) en promedio por posición. El costo de la función de hash sigue siendo O(m)O(m)O(m), y el costo de recorrer la lista es O(k)O(k)O(k), donde kkk es el número promedio de colisiones en cada posición. Si la tabla tiene una carga adecuada, el número de colisiones por posición será bajo, y la operación de búsqueda, inserción o eliminación tomará un tiempo constante por operación.

Total en el caso promedio:

O(m+k)(donde k es pequen˜o, por lo tanto O(m+1))O(m + k) \quad \text{(donde \(k \) es pequeño, por lo tanto \(O(m + 1) \))}O(m+k)(donde k es pequen˜o, por lo tanto O(m+1))

6) ***Análisis del Peor Caso***

En el caso de sondeo lineal, si la tabla está muy llena o si hay muchas colisiones, la búsqueda de una posición libre o una coincidencia puede requerir explorar varias posiciones de la tabla. En el peor de los casos, podríamos tener que recorrer toda la tabla, lo que resulta en un número de intentos de hasta O(n)O(n)O(n) para encontrar una posición libre o un elemento coincidente.

Además, cada intento de comparación de claves tiene un costo de O(m)O(m)O(m), donde mmm es el tamaño promedio de la clave (o el tiempo requerido para comparar dos claves).

Total en el peor caso:

O(n)×O(m)=O(n×m)O(n) \times O(m) = O(n \times m)O(n)×O(m)=O(n×m)

En el caso de encadenamiento, si muchas claves colisionan en el mismo índice, la lista enlazada en esa posición puede crecer considerablemente. En el peor de los casos, si todas las claves se mapean al mismo índice (lo cual sería un escenario muy desfavorable), la lista enlazada podría tener hasta O(n)O(n)O(n) elementos, y la búsqueda de una clave en esa lista tomaría un tiempo proporcional al número de elementos en ella.

La función de hashing toma O(m)O(m)O(m) por cada comparación de claves, y si tenemos una lista de kkk elementos en una posición, el tiempo total para buscar o insertar un elemento sería O(k)O(k)O(k) para recorrer la lista.

Total en el peor caso:

O(m+k)(donde k puede ser hasta n)O(m + k) \quad \text{(donde \(k \) puede ser hasta \(n \))}O(m+k)(donde k puede ser hasta n)

Si todas las claves caen en el mismo índice, esto sería equivalente a O(m+n)O(m + n)O(m+n).

VII. Resultados

En conclusión, el método de búsqueda mediante hashing demostró ser más eficiente a la búsqueda binaria en todos los escenarios evaluados (mejor caso, caso promedio, peor caso), mostrando consistentemente tiempos de ejecución menores. Además, su principal ventaja es que no necesita de un ordenamiento previo de los datos, ya que accede directamente a los

elementos mediante claves. Lo que lo convierte en una opción óptima para su implementación en inventarios grandes donde el rendimiento es un factor importante.

Por otro lado, el tiempo de ejecución del algoritmo de ordenamiento tuvo resultados similares en todos los escenarios:

En el mejor de los casos, el ordenamiento de un inventario con 1,000,000 de productos previamente ordenados tomó 5.53 segundos, un resultado bastante eficiente considerando el volumen de datos.

En el caso promedio, donde los 1,000,000 de productos fueron generados aleatoriamente, el algoritmo tardó 12.69 segundos. Aunque es un tiempo aceptable, se nota un incremento debido al mayor trabajo de comparación y mezcla.

Curiosamente, en el peor de los casos con los datos completamente ordenados inversamente, el tiempo fue el más bajo de los tres: 5.06 segundos. Este resultado puede atribuirse a la naturaleza estable y balanceada del algoritmo Merge Sort, cuya complejidad O(n log n) se mantiene constante sin importar el escenario.

VIII. Conclusiones y recomendaciones

La presente investigación logró cumplir con éxito su propósito de analizar, implementar y evaluar dos algoritmos fundamentales en la informática: Hashing para búsqueda y Merge Sort para ordenamiento, aplicados dentro de un sistema real de inventario desarrollado en Python. A lo largo del estudio, se demostró que la correcta selección e integración de estos algoritmos puede generar mejoras significativas en la eficiencia, rapidez y organización de los datos, elementos esenciales para sistemas modernos que procesan grandes volúmenes de información.

Desde el análisis a priori, se evidenció que ambos algoritmos presentan ventajas teóricas sólidas: Hashing permite búsquedas con complejidad constante O(1) en condiciones óptimas, mientras que Merge Sort mantiene un rendimiento estable de O(n log n), sin importar el orden inicial de los datos. Estas características fueron confirmadas en el análisis a posteriori mediante pruebas prácticas, donde se registraron tiempos de ejecución bajos y un comportamiento altamente eficiente tanto en búsquedas como en ordenamientos.

El desarrollo del sistema de inventario permitió materializar estos conceptos, ofreciendo una solución funcional y realista para la gestión de productos. Las métricas obtenidas respaldan la efectividad de los algoritmos implementados, y su aplicabilidad demuestra que la teoría algorítmica, cuando se traslada correctamente a la práctica, tiene un impacto directo en el rendimiento y calidad de los sistemas informáticos.

En conclusión, este proyecto no sólo valida la importancia de algoritmos bien estructurados para resolver problemas comunes de búsqueda y ordenamiento, sino que también refuerza la necesidad de un enfoque experimental, técnico y aplicado dentro de la ingeniería y el desarrollo de software. El sistema creado representa una base sólida sobre la cual se pueden construir soluciones más complejas y escalables, optimizando procesos en diversos entornos computacionales.

Recomendaciones

1. Utilizar Merge Sort en escenarios donde se requiera ordenamiento estable y el tamaño de los datos pueda ser considerable. Es ideal para sistemas donde la precisión y consistencia en la presentación de información son claves, como reportes o visualización ordenada de registros.
2. Aplicar Hashing cuando se necesiten búsquedas rápidas y frecuentes, especialmente en sistemas que gestionan grandes cantidades de datos con claves únicas, como códigos de productos en inventarios o usuarios en plataformas.
3. Mantener un buen diseño de la función hash y controlar el factor de carga para evitar colisiones que puedan afectar el rendimiento del sistema. Se recomienda usar una tabla hash de tamaño adecuado según el volumen de datos esperado.
4. Aunque no se hizo una comparación directa con otros métodos, se sugiere a futuro explorar alternativas híbridas o complementarias, como Quicksort o árboles balanceados, para determinar si ofrecen beneficios adicionales en contextos específicos.
5. Finalmente, documentar bien cada módulo del sistema y realizar pruebas periódicas con datos reales, lo que ayudará a mantener un rendimiento óptimo y a facilitar futuras mejoras o adaptaciones.

IX. Referencias bibliográficas

Brassard, G., & Bratley, P. (1996). *Fundamentals of Algorithmics*. Prentice Hall.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Equipo de Enciclopedia Significados. (2024, November 12). *Investigación Experimental: qué es, características, tipos y ejemplos*. Enciclopedia Significados. Retrieved July 6, 2025, from https://www.significados.com/investigacion-experimental/

FreeCodeCamp (2023). *Algoritmos de ordenación explicados con ejemplos en JavaScript, Python, Java y C++*. freeCodeCamp.org. https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/

Frias, S (2021). *Significado del algoritmo divide y vencerás: Explicado con ejemplos*. freeCodeCamp.org. https://www.freecodecamp.org/espanol/news/significado-del-algoritmo-divide-y-venceras/

Investigadores. (2022, febrero 06). *Definición del alcance de la investigación científica*. Investigación Cientifica.org. https://investigacioncientifica.org/definicion-del-alcance-de-la-investigacion-a-realizar-exploratorio-descriptiva-correlacional-o-explicativa/

Levitin, A. (2012). *Introduction to the Design and Analysis of Algorithms* (3rd ed.). Pearson.

Rivera, J. (2021). *Introducción a la complejidad temporal de los algoritmos*. freeCodeCamp.org. https://www.freecodecamp.org/espanol/news/introduccion-a-la-complejidad-temporal-de-los-algoritmos/

Rodriguez, S. (s. f). *Metodología Cuantitativa: Ejemplos Prácticos y Aplicaciones Relevantes*. LAB-ES Blog de economía. Retrieved 2025, from https://labes-unizar.es/metodologia-cuantitativa-ejemplos-practicos-y-aplicaciones-relevantes/?expand_article=1

Runestone (s.f). *Problem Solving with Algorithms and Data Structures using C++*. https://runestone.academy/ns/books/published/cppds/AlgorithmAnalysis/HashTableAnalysis.html

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.