**UNIVERSIDAD AMERICANA (UAM)**

****

Facultad de Ingeniería y Arquitectura

**Eficiencia Computacional de QuickSort y Tablas Hash con Encadenamiento Separado ante el Aumento del Tamaños de Datos**

Franco Xavier Aguilera Ortez

David Joel Sánchez Acevedo

Alicia Massiel Estrada Acevedo

Sara Alejandra Zambrana Taylor

Andrea Johanna Duarte Guerrero

Docente: Silvia Ticay

*Managua, Julio 2025*

**Índice**

[**Antecedentes 5**](#_x37ylprxgsqa)

[**Planteamiento del problema 9**](#_agczff7sgwsj)

[Problema e Interrogantes científicas 10](#_9bhr5mergvw7)

[Pregunta General: 10](#_4whkcul2wm4k)

[Preguntas específicas: 10](#_x6mzkwja5ecd)

[Objetivos 11](#_2auhnb75wm42)

[Objetivo General 11](#_lw8xkbqoseub)

[Objetivos Específicos 11](#_u5z30p7xwz6s)

[**Hipótesis General 14**](#_jyrvh38fs75z)

[**Marco Teórico 15**](#_qedltrq9c5ob)

[Conceptos Fundamentales 15](#_6pn6s7x4ca8e)

[Algoritmo 15](#_7m58y6swxefw)

[Ordenamiento 15](#_nq5zyyun4zk9)

[Estructura de datos 16](#_mg5l9yw9yccp)

[Complejidad algorítmica 16](#_eqoqi1ntqk4)

[Notación Big O 16](#_xyiraa4wk3o8)

[Mejor caso, caso promedio y peor caso 17](#_58lu0kjfx9i5)

[Eficiencia computacional 17](#_n4qknuuuh0m)

[Tiempo de ejecución 17](#_fkp4dl2209i1)

[Uso de memoria 18](#_hsh6ybub6a3w)

[Escalabilidad 18](#_1comvyajqau9)

[Análisis A Priori y A Posteriori en la Investigación de Algoritmos 18](#_3pnvbsq3yiz0)

[Análisis a Priori 19](#_aaw8tfqr0p6z)

[Análisis a Posteriori 20](#_3ydgoyg641w4)

[Importancia combinada 21](#_ikph54s482h4)

[QuickSort 21](#_5q2rw8mqebwq)

[Complejidad de QuickSort 22](#_7gpgat71sapv)

[Análisis del Rendimiento: Tiempo de Ejecución y Uso de Memoria 22](#_623xokg3xnz0)

[Tablas Hash 23](#_kmjlezso1564)

[Estrategias de Resolución de Colisiones: Encadenamiento Separado 23](#_41sczukf7r4b)

[Análisis del Rendimiento: Tiempo, Memoria y Factor de Carga 24](#_c8d7q3nop7op)

[**Marco Metodológico 26**](#_bl2hh437n9s5)

[Diseño de la Investigación 26](#_la7dqot8ujww)

[Enfoque de la Investigación 27](#_dsald1eo6k0m)

[Alcance de la Investigación 27](#_wgj4vipq9r8x)

[Unidades de Análisis y Conjunto de Datos 28](#_fhgzp3hsbrsp)

[Técnicas e Instrumentos de Recolección de Datos 28](#_szu63q81ikjr)

[Procedimiento de la Investigación 29](#_j7ep2cbr12qf)

[Operacionalización de Variables 31](#_x96vehvd9thk)

[Técnicas de Análisis de Datos 32](#_3qmowqmnxuo2)

[**Análisis de los Resultados 32**](#_eypbnexq3y91)

[QuickSort 32](#_5pt3oikfrtj6)

[Documentación del Código 32](#_a1ce9suikcp)

[Análisis y Discusión de Resultados 34](#_b9u2zoxyve8p)

[Tiempo de Ejecución 34](#_myzbj6rhlebl)

[Uso de Memoria 37](#_vtjhrw2gxscs)

[Conclusiones sobre la Eficiencia de QuickSort 40](#_2hj8in65j3yy)

[Hashing con Encadenamiento Separado 41](#_1vkuhb1tx4te)

[Variables, métricas y parámetros experimentales 41](#_8wvtqghs41oe)

[Perfil de uso de memoria bajo distintas cargas 45](#_52lbcikkb2fd)

[Resultados agregados de múltiples ejecuciones (resultados\_hash.csv) 47](#_xck32v1h6cj2)

[Rendimiento de tiempo 47](#_cptaked9yp4b)

[Tiempo de inserción 48](#_6bkim1mwp8wq)

[Tiempos de búsqueda e eliminación 49](#_hk5l2nqoco7x)

[Comportamiento del uso de memoria 50](#_x7jsl1dzd618)

[Interpretación de resultados 53](#_oufitiggh66o)

[Eficiencia temporal de las operaciones 53](#_3jq1zgl0j9mo)

[Uso de memoria bajo distintas cargas 54](#_c1eh9zigo3dx)

[Validación vs. Complejidad teórica 55](#_z2jcg8gkpw8g)

[Conclusiones finales 56](#_9kz6r3yf9hb6)

[**Referencias 60**](#_t9xar0cng0a3)

[**Anexos 61**](#_jo42mf6gfdx8)

Introducción

El crecimiento exponencial de los volúmenes de datos en aplicaciones informáticas modernas plantea nuevos desafíos en cuanto al procesamiento eficiente de información. En este contexto, resulta fundamental evaluar el rendimiento práctico de algoritmos y estructuras de datos ampliamente utilizados. Esta investigación se enfoca en analizar la eficiencia computacional de QuickSort y de las tablas hash con encadenamiento separado, dos herramientas esenciales en el campo de la informática, considerando cómo se comportan ante el aumento progresivo del tamaño de los datos.

# 

# Antecedentes

La eficiencia computacional es una preocupación constante en el diseño de algoritmos y estructuras de datos, especialmente en un contexto donde el volumen de información a procesar crece de forma exponencial. En este marco, los algoritmos de ordenamiento y las tablas hash han sido objeto de numerosos estudios, tanto teóricos como empíricos, que han buscado comprender y optimizar su rendimiento en distintos escenarios. La presente investigación se inscribe en esta línea, enfocándose específicamente en el análisis de QuickSort y de las tablas hash con encadenamiento separado, frente al aumento progresivo del tamaño de los datos. A continuación, se presenta una revisión de los principales aportes científicos relacionados con estos temas, destacando vacíos, limitaciones y oportunidades de aplicación a la investigación propuesta.

En cuanto al algoritmo QuickSort, su desarrollo se remonta a 1961, cuando fue introducido por Tony Hoare como un método eficiente de ordenamiento basado en la estrategia de "divide y vencerás". Desde entonces, ha sido objeto de numerosas mejoras y variantes. Aftab et al. (2021) realizaron una revisión exhaustiva sobre la evolución de QuickSort y sus distintas variantes, incluyendo enfoques con múltiples pivotes y versiones híbridas que lo combinan con otros algoritmos como MergeSort y ShellSort. Este estudio destaca que, aunque la complejidad promedio de QuickSort es de O(n log n), en el peor de los casos puede alcanzar O(n²), especialmente cuando la selección del pivote es inadecuada. Por ello, diversas estrategias han sido desarrolladas para mejorar su desempeño, tales como la selección del pivote mediante la mediana de tres, el uso de pivotes aleatorios o la incorporación de métodos híbridos para reducir la profundidad de recursión.

Xiang (2011) aportó una base teórica sólida al analizar formalmente la complejidad temporal de QuickSort bajo diferentes condiciones. Su trabajo resalta que, si bien QuickSort se comporta de forma eficiente en la mayoría de los casos, su rendimiento puede verse afectado de manera significativa por la distribución de los datos y la elección del pivote. Este enfoque resalta la necesidad de evaluar no solo el comportamiento promedio del algoritmo, sino también su respuesta ante situaciones adversas, lo cual es relevante para el propósito de esta investigación al probar el algoritmo frente a datos crecientes.

En la misma línea, Hulín (2017), en una tesis de licenciatura, llevó a cabo un análisis empírico riguroso del rendimiento de varios algoritmos de ordenamiento, incluyendo QuickSort, bajo diferentes configuraciones de hardware y estructuras de datos. Utilizando un entorno de pruebas controlado y una metodología cuidadosamente diseñada, Hulín implementó y comparó múltiples algoritmos, destacando la eficiencia de QuickSort con pivote de mediana de tres. Además, su estudio subraya la importancia de factores como el tipo de datos, el uso de memoria y el impacto del stack en el rendimiento final, elementos clave que también serán abordados en la presente investigación. Lo más relevante es que Hulín utilizó explícitamente el tiempo de ejecución y el uso de memoria como las principales métricas de evaluación de eficiencia, lo cual respalda directamente el enfoque cuantitativo adoptado en este estudio.

Por otro lado, Marcellino, Pratama, Suntiarko y Margi (2021) realizaron una comparación experimental entre QuickSort y otros algoritmos avanzados como HeapSort, IntroSort, MergeSort y RadixSort, utilizando un conjunto real de 11,000 títulos de libros. Esta investigación, implementada en Python y ejecutada en un entorno moderno, reveló que QuickSort y IntroSort fueron los algoritmos más rápidos, aunque con un uso de memoria relativamente elevado. Su enfoque práctico permite validar los límites del análisis teórico, resaltando que los valores esperados de complejidad no siempre se reflejan directamente en contextos reales. El uso sistemático de tiempo de ejecución y consumo de memoria como variables medibles refuerza la importancia de incorporar estas métricas en pruebas empíricas, tal como se propone en esta investigación.

En cuanto a las estructuras de datos, específicamente las tablas hash, su estudio es igualmente amplio y relevante. Las tablas hash son una de las estructuras más utilizadas para búsqueda y almacenamiento debido a su tiempo promedio de acceso constante O(1). Sin embargo, su rendimiento práctico está condicionado por factores como la calidad de la función hash, el manejo de colisiones y la carga de datos. García Hernández (2021) realizó una evaluación empírica del desempeño de las tablas hash en varios lenguajes de programación, destacando cómo la técnica de resolución de colisiones afecta la eficiencia. Lo más relevante de su estudio es que evaluó explícitamente las operaciones de inserción, búsqueda y eliminación, utilizando como métrica central el tiempo de ejecución promedio de cada operación. Este enfoque permite medir el comportamiento real de la tabla en condiciones variables de carga, lo que resulta directamente aplicable a este estudio.

A su vez, Liu et al. (2014) abordaron el rendimiento de las tablas hash desde un enfoque experimental, comparando métodos de direccionamiento abierto y cerrado, siendo este último el que implementa el encadenamiento separado, foco del presente estudio. Sus resultados mostraron que, si bien el direccionamiento abierto puede ser más eficiente en el uso de memoria, el encadenamiento separado ofrece mayor estabilidad en grandes volúmenes de datos. Además, observaron que el rendimiento no mejora indefinidamente al aumentar el tamaño de la tabla, lo cual indica que existe un límite práctico a la eficiencia de las tablas hash. En su análisis, también se midieron explícitamente las operaciones de inserción, búsqueda y eliminación mediante el número de accesos a memoria, una métrica concreta que sirve como proxy del tiempo de ejecución, reforzando la validez de las mediciones utilizadas en esta investigación.

A nivel metodológico, la mayoría de estos estudios comparten el enfoque de implementar algoritmos en lenguajes como C++, Java o Python y medir su rendimiento mediante pruebas repetidas, utilizando diferentes tamaños de entrada. Esta técnica, que será replicada en la presente investigación, permite construir una visión más clara sobre el comportamiento de los algoritmos bajo condiciones reales, más allá de las predicciones teóricas.

En síntesis, la literatura revisada evidencia que, si bien QuickSort y las tablas hash con encadenamiento separado gozan de un sólido respaldo teórico, sus comportamientos reales pueden variar significativamente. Las investigaciones existentes aportan modelos experimentales y hallazgos importantes, pero también dejan abierta la necesidad de validar estos comportamientos en lenguajes y plataformas modernas, como Python, y en contextos con tamaños crecientes de datos. Además, el uso explícito de métricas como el tiempo de ejecución, consumo de memoria y tiempo promedio de operaciones específicas (inserción, búsqueda y eliminación) en investigaciones previas, no solo respalda metodológicamente este estudio, sino que garantiza que sus resultados puedan ser comparables, relevantes y transferibles al campo aplicado del desarrollo de software.

# 

# Planteamiento del problema

El crecimiento acelerado de los volúmenes de datos en aplicaciones informáticas genera la necesidad de utilizar algoritmos y estructuras de datos eficientes para procesarlos y almacenarlos adecuadamente. QuickSort y las tablas hash con encadenamiento separado son técnicas ampliamente reconocidas por su eficiencia teórica en ordenamiento y gestión de datos. Sin embargo, su desempeño práctico puede variar significativamente debido a factores como la implementación específica, el lenguaje de programación utilizado, las características del hardware y el tamaño creciente de los datos. Esta discrepancia entre el comportamiento teórico y real dificulta la selección óptima de algoritmos en entornos reales de desarrollo.

La presente investigación se centra en evaluar la eficiencia computacional de QuickSort y las tablas hash con encadenamiento separado, ambos implementados en Python. Para ello, se medirán el tiempo de ejecución y el uso de memoria al procesar distintos tamaños de datos. El análisis se llevará a cabo en dos entornos controlados con hardware específico: el primero con Windows 11 Home, procesador Intel Core i7-1165G7 de 11ª generación y 20 GB de RAM; el segundo, con macOS Sequoia 15.1, chip M1 y 8 GB de RAM. Este estudio se limita a aplicaciones locales, excluyendo entornos distribuidos, otros lenguajes de programación y métricas adicionales, como el consumo energético o el rendimiento en tareas paralelas.

## 

## Problema e Interrogantes científicas

### Pregunta General:

¿Cuál es la eficiencia computacional del algoritmo de ordenamiento QuickSort y de las tablas hash con encadenamiento separado ante el aumento del tamaño de datos?

### Preguntas específicas:

1. ¿Cómo se comporta el tiempo de ejecución y el uso de memoria del algoritmo QuickSort al aumentar el tamaño de los datos?
2. ¿Qué relación existe entre el tamaño de los datos y el tiempo promedio de las operaciones de inserción, búsqueda y eliminación en tablas hash con encadenamiento separado?
3. ¿Cómo varía el uso de memoria de las tablas hash con encadenamiento separado al incrementar el tamaño de los datos?
4. ¿En qué medida los resultados experimentales de QuickSort y las tablas hash reflejan el comportamiento esperado según sus complejidades teóricas?

## 

## Objetivos

### Objetivo General

Analizar la eficiencia computacional del algoritmo de ordenamiento QuickSort y de las tablas hash con encadenamiento separado ante el aumento del tamaño de datos.

### Objetivos Específicos

1. Implementar el algoritmo QuickSort en Python para medir su tiempo de ejecución y uso de memoria ante tamaños crecientes de datos.
2. Evaluar el rendimiento de una tabla hash con encadenamiento separado, midiendo el tiempo de las operaciones de inserción, búsqueda y eliminación con diferentes tamaños de datos.
3. Cuantificar el uso de memoria en las tablas hash con encadenamiento separado bajo distintas cargas de datos.
4. Identificar patrones en el comportamiento práctico de QuickSort y las tablas hash, comparando los resultados con sus complejidades teóricas para validar su eficiencia computacional.

**Justificación**

Esta investigación aporta al conocimiento teórico en ciencias de la computación al validar empíricamente la eficiencia de dos algoritmos fundamentales: QuickSort y tablas hash con encadenamiento separado. Mediante la implementación en Python y la medición del tiempo de ejecución y uso de memoria, se confrontan las complejidades teóricas clásicas con comportamientos prácticos en escenarios con tamaños de datos crecientes. Esto fortalece la comprensión del desempeño real de estas estructuras y métodos en contextos modernos, aportando evidencia que puede ser utilizada para mejorar futuras implementaciones y estudios en el área.

Los resultados tienen relevancia práctica para desarrolladores y profesionales del sector tecnológico que requieren optimizar el procesamiento y almacenamiento de grandes volúmenes de datos. Al comprender cómo varían el tiempo y la memoria con diferentes tamaños de datos, es posible tomar decisiones informadas para seleccionar y ajustar algoritmos en sistemas reales, mejorando la eficiencia y rendimiento. Esto contribuye a la productividad y al uso racional de recursos computacionales, aspectos clave en el diseño y operación de aplicaciones de alto desempeño y bases de datos.

Entre las limitaciones de esta investigación se encuentran la implementación exclusiva en Python y la evaluación restringida a dos entornos de hardware específicos: uno con procesador Intel Core i7-1165G7 de 11ª generación y 20 GB de RAM, y otro con chip M1 y 8 GB de RAM. Esta restricción puede limitar la generalización de los resultados a otros lenguajes de programación o plataformas. Además, el estudio se enfoca únicamente en las métricas de tiempo de ejecución y uso de memoria, sin explorar otros aspectos como el consumo energético o el rendimiento en sistemas distribuidos. A pesar de estas restricciones, el estudio proporciona una base sólida para comprender el comportamiento básico de QuickSort y las tablas hash bajo condiciones prácticas controladas satisfaciendo la necesidad actual de adaptar conocimientos teóricos a contextos tecnológicos contemporáneos y orientando futuras investigaciones.

# 

# Hipótesis General

El tiempo de ejecución y el uso de memoria de QuickSort y las tablas hash con encadenamiento separado reflejan un comportamiento consistente con sus complejidades teóricas bajo diferentes tamaños de datos.

# 

# Marco Teórico

Este marco teórico establece las bases para analizar la eficiencia computacional de algoritmos, con énfasis en QuickSort y las tablas hash con encadenamiento separado. Se abordarán conceptos clave como algoritmos, complejidad algorítmica y notación Big O, así como los análisis a priori y a posteriori del rendimiento. Además, se detallarán las características y desempeño de QuickSort y las tablas hash, preparando el contexto para su evaluación empírica en términos de tiempo, memoria y escalabilidad.

## Conceptos Fundamentales

Para comprender adecuadamente los análisis teóricos y empíricos de los algoritmos QuickSort y las tablas hash con encadenamiento separado, es necesario definir una serie de conceptos fundamentales que sustentan esta investigación.

### Algoritmo

Un algoritmo es una secuencia finita de pasos bien definidos que resuelve un problema específico o realiza una tarea computacional (Cormen et al., 2009). Su propósito es transformar entradas en salidas deseadas a través de un proceso estructurado. Los algoritmos pueden clasificarse según su función (búsqueda, ordenamiento, cifrado), estructura (iterativo, recursivo) y eficiencia computacional.

En este estudio se analizan algoritmos para ordenamiento (QuickSort) y estructuras algorítmicas para almacenamiento y acceso eficiente (tablas hash).

### Ordenamiento

El ordenamiento es un proceso algorítmico que organiza elementos según un criterio específico, como orden numérico o alfabético. Es fundamental en programación, ya que mejora la eficiencia de otras operaciones como búsqueda binaria, eliminación de duplicados y análisis estadístico. QuickSort es un algoritmo de ordenamiento eficiente basado en la estrategia "divide y vencerás".

### Estructura de datos

Una estructura de datos es una forma de organizar, almacenar y manipular información en un programa. Las tablas hash son estructuras que permiten acceder a datos de forma eficiente mediante funciones hash, asociando claves a valores y resolviendo posibles colisiones a través del encadenamiento.

### Complejidad algorítmica

La complejidad algorítmica se refiere al análisis de los recursos necesarios para ejecutar un algoritmo. Se expresa principalmente en:

* **Complejidad temporal**: número de operaciones respecto al tamaño de la entrada (n).
* **Complejidad espacial**: cantidad de memoria requerida durante la ejecución.

Se representa habitualmente con la notación Big O (por ejemplo, O(n), O(log n), O(n log n), O(n²)).

### Notación Big O

Es una notación matemática utilizada para describir el comportamiento asintótico de un algoritmo en función del tamaño de los datos. Permite comparar la escalabilidad de algoritmos de forma independiente del hardware o lenguaje de programación. Por ejemplo:

* QuickSort: O(n log n) en promedio.
* Tabla hash: O(1) en operaciones básicas, bajo condiciones óptimas.

### Mejor caso, caso promedio y peor caso

Estos escenarios permiten describir el rendimiento de un algoritmo bajo diferentes condiciones de entrada:

* **Mejor caso**: cuando el algoritmo se ejecuta en condiciones ideales.
* **Caso promedio**: rendimiento general esperado con entradas aleatorias.
* **Peor caso**: cuando el algoritmo enfrenta las condiciones más desfavorables.

Estos casos serán evaluados empíricamente en esta investigación para ambos algoritmos, comparando su desempeño real con las estimaciones teóricas.

## Eficiencia computacional

La eficiencia computacional es la capacidad de un algoritmo para utilizar de forma óptima los recursos computacionales (tiempo y memoria) al resolver un problema. En esta investigación, este concepto se contextualiza como la capacidad de QuickSort y de las tablas hash con encadenamiento separado para mantener un rendimiento estable y eficaz ante el incremento del tamaño de los datos procesados.

### Tiempo de ejecución

En este estudio, el tiempo de ejecución se refiere al período que tarda un algoritmo en completar una tarea específica. Esta métrica se mide en segundos o milisegundos usando herramientas como *timeit* o *perf\_counter* en Python. En QuickSort, se mide el tiempo total para ordenar listas de diferentes tamaños. En tablas hash, se mide el tiempo promedio de operaciones individuales (inserción, búsqueda, eliminación).

### 

### Uso de memoria

El uso de memoria representa la cantidad de espacio en RAM que consume un algoritmo durante su ejecución. Este valor depende de las variables, estructuras auxiliares y profundidad de la recursión. Se medirá con herramientas como *memory\_profiler*. En QuickSort, se observará el impacto de la recursión en la memoria. En las tablas hash, se medirá el crecimiento del consumo a medida que aumentan las colisiones.

### Escalabilidad

Es la capacidad de un algoritmo para mantener su eficiencia a medida que aumentan los datos. Una solución escalable mantiene su rendimiento estable o crece de forma controlada en consumo de recursos. Este criterio es esencial para validar el uso de QuickSort y tablas hash en contextos reales.

## Análisis A Priori y A Posteriori en la Investigación de Algoritmos

En el estudio de algoritmos y estructuras de datos, existen dos enfoques fundamentales para evaluar su comportamiento y eficiencia: el análisis a priori y el análisis a posteriori. Ambos enfoques son complementarios y necesarios para obtener una visión integral del rendimiento de un algoritmo en la teoría y en la práctica. A continuación, se presentan las definiciones y aplicaciones de cada uno en el contexto de la presente investigación sobre la eficiencia computacional del algoritmo QuickSort y de las tablas hash con encadenamiento separado.

### Análisis a Priori

El análisis a priori se refiere a la evaluación teórica del comportamiento de un algoritmo antes de ser implementado o ejecutado. Este tipo de análisis permite anticipar el rendimiento de un algoritmo en función del tamaño de los datos de entrada, utilizando herramientas matemáticas y lógicas. Es independiente del lenguaje de programación, del hardware y del sistema operativo.

En el contexto de esta investigación, el análisis a priori se realiza para:

* Determinar la **eficiencia temporal** teórica, mediante la estimación del número de operaciones que realiza el algoritmo en función de la entrada (usualmente expresado con la notación Big O).
* Evaluar la **eficiencia espacial**, que se refiere a la cantidad de memoria que requiere el algoritmo, considerando variables, estructuras auxiliares y recursividad.

Ejemplos:

* QuickSort tiene una complejidad temporal promedio de **O(n log n)** y una complejidad espacial de **O(log n)** en su versión recursiva.
* Las operaciones de las tablas hash (inserción, búsqueda y eliminación) tienen una complejidad promedio de **O(1)** si se mantiene un buen factor de carga.

Este análisis permite comparar algoritmos sin necesidad de ejecutarlos, lo cual es útil en la etapa de selección de técnicas para resolver un problema.

### 

### Análisis a Posteriori

El análisis a posteriori es una evaluación empírica del comportamiento del algoritmo tras su implementación y ejecución en un entorno real. Este tipo de análisis considera el hardware, el lenguaje de programación, las condiciones del sistema y los datos reales o simulados que se procesan.

En esta investigación, el análisis a posteriori se aplicará para:

* Medir el **tiempo de ejecución real** del algoritmo en diferentes tamaños de entrada, utilizando herramientas como *timeit* en Python.
* Evaluar el **uso real de memoria**, monitoreando el consumo de recursos con bibliotecas como *memory\_profiler.*
* Comparar el comportamiento en el **mejor caso**, **caso promedio** y **peor caso**, según las condiciones del conjunto de datos y las características del algoritmo.

Ejemplos:

* Se medirá cuánto tiempo tarda QuickSort en ordenar listas de 1,000 a 1,000,000 elementos.
* Se evaluará cuál es el tiempo promedio que tarda una tabla hash en insertar, buscar o eliminar elementos bajo distintas cargas.

Este análisis permite validar o refutar las predicciones teóricas del análisis a priori y ofrece una perspectiva más cercana a la realidad de uso del algoritmo en entornos de producción o experimentación.

### 

### Importancia combinada

Utilizar ambos enfoques garantiza una comprensión completa del rendimiento algorítmico:

* El análisis a priori permite **predecir el comportamiento esperado** bajo modelos matemáticos.
* El análisis a posteriori permite **medir el comportamiento real** y descubrir factores no considerados en la teoría, como la gestión de memoria del lenguaje o las optimizaciones del compilador.

En conjunto, ambos enfoques permiten tomar decisiones fundamentadas en la selección, ajuste y evaluación de algoritmos en contextos académicos, profesionales y experimentales.

## QuickSort

QuickSort es un algoritmo de ordenamiento basado en la estrategia "divide y vencerás", desarrollado por C. A. R. Hoare en 1961. Su funcionamiento consiste en seleccionar un elemento denominado pivote y reorganizar el arreglo de modo que los elementos menores al pivote queden a la izquierda y los mayores a la derecha. Este proceso se aplica de forma recursiva a cada sublista resultante, hasta que todas estén ordenadas (Cormen et al., 2009).

Es uno de los algoritmos más eficientes en la práctica para conjuntos de datos grandes, ya que combina velocidad con una implementación relativamente sencilla. Además, tiene un buen comportamiento en la mayoría de los casos, aunque su eficiencia puede verse afectada por una mala elección del pivote.

### 

### Complejidad de QuickSort

* **Mejor caso**: O(n log n), ocurre cuando el pivote divide uniformemente el arreglo.
* **Caso promedio**: O(n log n), se da con entradas aleatorias o pivote aleatorio.
* **Peor caso**: O(n²), ocurre cuando el pivote es el mínimo o máximo valor y las divisiones son desbalanceadas.

### Análisis del Rendimiento: Tiempo de Ejecución y Uso de Memoria

En el presente estudio, QuickSort será evaluado mediante pruebas experimentales en Python, midiendo:

* **Tiempo de ejecución**: cuánto tarda el algoritmo en ordenar listas de 1,000, 10,000, 100,000 elementos.
* **Uso de memoria**: cuánta memoria se consume durante la recursión y manipulación de listas.

Estas mediciones se realizarán con herramientas como *timeit* y *memory\_profiler*, comparando versiones básicas y optimizadas del algoritmo. Se evaluará también su escalabilidad, observando cómo se comporta frente a conjuntos de datos crecientes, en condiciones controladas.

Esta sección permite contextualizar la relevancia del algoritmo QuickSort dentro del análisis algorítmico, ofreciendo una base para comparar sus resultados con los obtenidos por las tablas hash con encadenamiento separado.

## 

## Tablas Hash

Las tablas hash son estructuras de datos diseñadas para el almacenamiento y la recuperación eficiente de información, asociando claves únicas con valores. Su principal ventaja teórica es la capacidad de ejecutar operaciones de inserción, búsqueda y eliminación en un tiempo promedio constante, representado por una complejidad de O(1) (García Hernández, 2021). Este rendimiento ideal, sin embargo, depende de la eficacia de sus componentes fundamentales: la función hash y la estrategia para manejar colisiones.

La función hash es el mecanismo que convierte una clave en un índice dentro de un arreglo o tabla. Una función robusta es esencial para distribuir las claves de manera uniforme, lo cual no solo minimiza la probabilidad de colisiones, sino que también es un factor relevante para evitar vulnerabilidades de seguridad (García Hernández, 2021). Cuando dos o más claves generan el mismo índice, se produce una colisión. La estrategia empleada para resolver estas colisiones es un factor determinante en el rendimiento práctico de la tabla hash (García Hernández, 2021). La presente investigación se enfoca en el método de

encadenamiento separado, una técnica ampliamente utilizada para este fin (Liu et al., 2014).

### Estrategias de Resolución de Colisiones: Encadenamiento Separado

El encadenamiento separado resuelve las colisiones almacenando todos los elementos que corresponden a un mismo índice en una estructura de datos secundaria, comúnmente una lista enlazada. Esta aproximación ha demostrado tener una estabilidad superior en aplicaciones que manejan grandes volúmenes de datos en comparación con otras técnicas como el direccionamiento abierto (Liu et al., 2014).

Estudios empíricos han validado la efectividad de este enfoque. Por ejemplo, la implementación *Unordered\_map* de C++, que utiliza encadenamiento, demostró el mejor rendimiento promedio y el uso de memoria más eficiente en una comparativa entre lenguajes (García Hernández, 2021). En contraste, técnicas de direccionamiento abierto, como el sondeo lineal, han sido consideradas imprácticas debido a su bajo rendimiento en escenarios con grandes volúmenes de datos (Liu et al., 2014). De igual forma, la implementación en Python, que utiliza direccionamiento abierto, puede presentar una complejidad menos predecible ante colisiones (García Hernández, 2021).

### Análisis del Rendimiento: Tiempo, Memoria y Factor de Carga

Si bien el encadenamiento separado es robusto, su rendimiento no es inmune a las condiciones de operación. Un concepto clave es el **factor de carga** (α), que se define como la relación entre el número de elementos almacenados y el tamaño de la tabla. A medida que este factor aumenta, también lo hace la probabilidad de colisiones, lo que puede llevar a una degradación del rendimiento si las listas enlazadas se vuelven excesivamente largas. Este efecto fue observado en la implementación de PHP, que, a pesar de usar encadenamiento, mostró una caída considerable en su rendimiento en situaciones de alta colisión (García Hernández, 2021).

El uso de memoria es otra dimensión crítica. En el encadenamiento separado, el consumo de memoria es directamente proporcional al número de elementos y al tamaño inicial de la tabla. No obstante, se ha señalado que esta técnica puede incurrir en un desperdicio significativo de espacio debido a los punteros o cabeceras que requiere cada nodo de las listas enlazadas (Liu et al., 2014). Esto presenta una compensación fundamental: el encadenamiento tiende a ser más rápido y estable bajo altas cargas, mientras que el direccionamiento abierto puede ser más eficiente en el uso de la memoria (García Hernández, 2021).

Finalmente, los estudios advierten que, aunque el tiempo de acceso amortizado de las tablas hash es consistente, los peores casos de rendimiento tienden a agravarse a medida que aumenta el tamaño de la tabla (Liu et al., 2014). Este hallazgo subraya la importancia de realizar análisis empíricos que, como el presente, validen el comportamiento teórico en escenarios prácticos y con conjuntos de datos crecientes.

# 

# Marco Metodológico

En este capítulo se describe la metodología que guiará el desarrollo de la investigación para cumplir con los objetivos propuestos. Se detallan el diseño, enfoque y alcance del estudio, así como las técnicas y procedimientos para la recolección y análisis de los datos. El propósito es establecer un plan de trabajo sistemático y riguroso que permita evaluar la eficiencia computacional de QuickSort y las tablas hash con encadenamiento separado de una manera objetiva y replicable.

## Diseño de la Investigación

El diseño de esta investigación es experimental. Se llevará a cabo una manipulación activa de variables independientes, como el tamaño de los conjuntos de datos y la carga de la tabla hash, en un entorno controlado para observar y medir sus efectos sobre las variables dependientes: el tiempo de ejecución y el uso de memoria. Este diseño permite establecer relaciones de causa y efecto entre el aumento de los datos y el rendimiento de los algoritmos, validando su comportamiento práctico más allá del análisis teórico.

## 

## Enfoque de la Investigación

El estudio se enmarca en un enfoque **cuantitativo**. Se recolectarán y analizarán datos numéricos y medibles para describir y explicar el fenómeno estudiado. Las métricas centrales, como el tiempo de ejecución (medido en segundos o milisegundos) y el uso de memoria (medido en megabytes), proporcionarán evidencia empírica para evaluar la eficiencia computacional. Este enfoque permite comparar el rendimiento de los algoritmos de forma objetiva y contrastar los resultados experimentales con las complejidades teóricas establecidas, como O(n log n) para QuickSort y O(1) para las tablas hash.

## Alcance de la Investigación

El alcance de la investigación es **descriptivo - explicativo**.

* **Descriptivo**: Se detallarán y caracterizarán los patrones de comportamiento de QuickSort y las tablas hash con encadenamiento separado. Se documentará cómo varían el tiempo de ejecución y el uso de memoria a medida que aumenta progresivamente el tamaño de los datos de entrada.
* **Explicativo**: Se buscará explicar la relación entre los resultados empíricos y las bases teóricas de cada algoritmo. Se analizará en qué medida el rendimiento observado en dos entornos de hardware y software específicos (Python sobre un procesador Intel Core i7 con 20 GB de RAM, Python sobre chip M1 y 8 GB de Ram) refleja las predicciones de la notación Big O, ofreciendo una explicación sobre las posibles discrepancias.

## 

## Unidades de Análisis y Conjunto de Datos

* **Unidades de Análisis**: Los sujetos de estudio son las implementaciones en Python del algoritmo de ordenamiento **QuickSort** y la estructura de datos **tabla hash con encadenamiento separado**.
* **Generación de Datos**: Para las pruebas se generarán conjuntos de datos sintéticos.
  + Para **QuickSort**, se crearán listas de números enteros generados de forma aleatoria con tamaños crecientes, por ejemplo: 1,000, 10,000, 100,000 y 1,000,000 de elementos, para evaluar su escalabilidad.
  + Para las **tablas hash**, se generarán claves y valores para simular operaciones de inserción, búsqueda y eliminación bajo diferentes factores de carga.

## Técnicas e Instrumentos de Recolección de Datos

Para la recolección de datos se utilizará un entorno de pruebas controlado y herramientas de software especializadas.

* **Entorno de Ejecución**: Las pruebas se realizaran en dos equipos con hardwares específicos para garantizar la consistencia de los resultados: un procesador **Intel Core i7-1165G7 de 11ª generación y 20 GB de memoria RAM**, **Chip Apple M1 y 8 GB de Ram**
* **Software y Lenguaje**: Los algoritmos serán implementados y probados utilizando el lenguaje de programación **Python**, debido a su relevancia en el desarrollo de software moderno.
* **Instrumentos de Medición**:
  + **Tiempo de Ejecución**: Se utilizarán las bibliotecas *timeit* o *perf\_counter* de Python para medir con precisión el tiempo transcurrido durante la ejecución de las operaciones de ordenamiento (para QuickSort) y de inserción, búsqueda y eliminación (para las tablas hash).
  + **Uso de Memoria**: Se empleará la biblioteca *memory\_profiler* de Python para monitorear el consumo de memoria RAM durante la ejecución de los algoritmos, registrando el uso máximo alcanzado para cada tamaño de datos.

## 

## Procedimiento de la Investigación

El proceso experimental se dividirá en las siguientes fases:

1. **Fase de Preparación**: Configuración de los entornos de desarrollo en los hardware especificados e instalación de las bibliotecas de Python necesarias (*memory\_profiler*).
2. **Fase de Implementación**: Desarrollo en Python de una implementación funcional del algoritmo QuickSort y de una clase para la tabla hash con encadenamiento separado.
3. **Fase de Pruebas de QuickSort**:
   * Generar los conjuntos de datos aleatorios con los tamaños definidos
   * Registrar el tiempo de ejecución promedio y el uso máximo de memoria para cada prueba.
4. **Fase de Pruebas de Tablas Hash**:
   * Para un tamaño de tabla fijo, realizar un número creciente de operaciones de inserción para simular diferentes factores de carga.
   * En cada nivel de carga, medir el tiempo promedio de las operaciones de inserción, búsqueda (de elementos existentes y no existentes) y eliminación.
   * Registrar el uso total de memoria de la estructura bajo cada carga.
5. **Fase de Análisis de Datos**: Consolidar los datos recolectados en tablas y proceder a su análisis estadístico y gráfico para identificar tendencias y patrones de comportamiento.

## Operacionalización de Variables

**Tabla 1**

*Matriz de operacionalización de variables*

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Definición Conceptual** | **Dimensiones** | **Indicadores** |
| **Eficiencia Computacional de QuickSort** | Capacidad del algoritmo para ordenar un conjunto de datos utilizando de forma óptima los recursos de tiempo y memoria, ante un incremento en el tamaño de la entrada. | Tiempo de Ejecución | Tiempo total para ordenar una lista (Segundos).  . |
| Uso de Memoria | Cantidad máxima de memoria RAM consumida (en MiB) |
| **Eficiencia Computacional de Tablas Hash con Encadenamiento Separado** | Capacidad de la estructura para gestionar operaciones de inserción, búsqueda y eliminación manteniendo un rendimiento estable y un uso de memoria controlado ante distintas cargas de datos. | Tiempo de Operaciones | Tiempo promedio por operación de inserción, búsqueda y eliminación (milisegundos). |
| Factor de Carga | Relación entre elementos almacenados y tamaño de la tabla (α). |
| Uso de Memoria | - Cantidad total de memoria RAM consumida (en Megabytes). |

## Técnicas de Análisis de Datos

Una vez recolectados los datos, se aplicarán las siguientes técnicas de análisis:

* **Análisis Gráfico**: Se elaborarán gráficos de líneas para visualizar la relación entre el tamaño de los datos (eje X) y el tiempo de ejecución y uso de memoria (eje Y). Esto permitirá identificar visualmente los patrones de crecimiento y la escalabilidad de cada algoritmo.
* **Análisis Comparativo**: Los resultados empíricos obtenidos se compararán directamente con la complejidad teórica (notación Big O) de cada algoritmo. El objetivo es validar si el comportamiento práctico se ajusta a las predicciones teóricas (ej. si el crecimiento del tiempo de QuickSort es log-lineal y si el de las operaciones de la tabla hash es constante).

# Análisis de los Resultados

## QuickSort

### Documentación del Código

El proyecto está modularizado en varios archivos Python para organizar la lógica y las funcionalidades. A continuación, se detalla cada archivo, sus variables y la lógica de sus funciones.

**quicksort\_algo.py:** Este archivo contiene la implementación fundamental del algoritmo de ordenamiento QuickSort.

*Variables:*

* arr (list): La lista de elementos a ordenar.
* pivot (any): El elemento seleccionado como pivote. En esta implementación, se elige el elemento central del arreglo.
* left (list): Sub-arreglo que contiene todos los elementos de 'arr' menores que 'pivot'.
* middle (list): Sub-arreglo que contiene todos los elementos de 'arr' iguales a 'pivot'.
* right (list): Sub-arreglo que contiene todos los elementos de 'arr' mayores que 'pivot'.

**data\_generator.py:** Este archivo contiene funciones utilitarias para generar diferentes tipos de conjuntos de datos de entrada para las pruebas. Genera una lista de números enteros aleatorios.

*Variables:*

* size (int): El número de elementos que contendrá la lista.
* max\_value (int, optional): El valor máximo posible para los números aleatorios generados. Si es None, se establece en *size \* 10* por defecto.

**performance\_metrics.py:** Este archivo contiene funciones para medir el rendimiento de los algoritmos, específicamente el tiempo de ejecución y preparando la medición de memoria. También, se encarga de procesar los resultados de las pruebas y presentarlos en un formato legible, utilizando tablas.

**plotter.py:** Este archivo contiene funciones para generar gráficos de rendimiento visuales utilizando la librería matplotlib

**main.py:** Este es el script principal que orquesta la ejecución de todas las pruebas, la recolección de métricas, el análisis y la visualización.

### Análisis y Discusión de Resultados

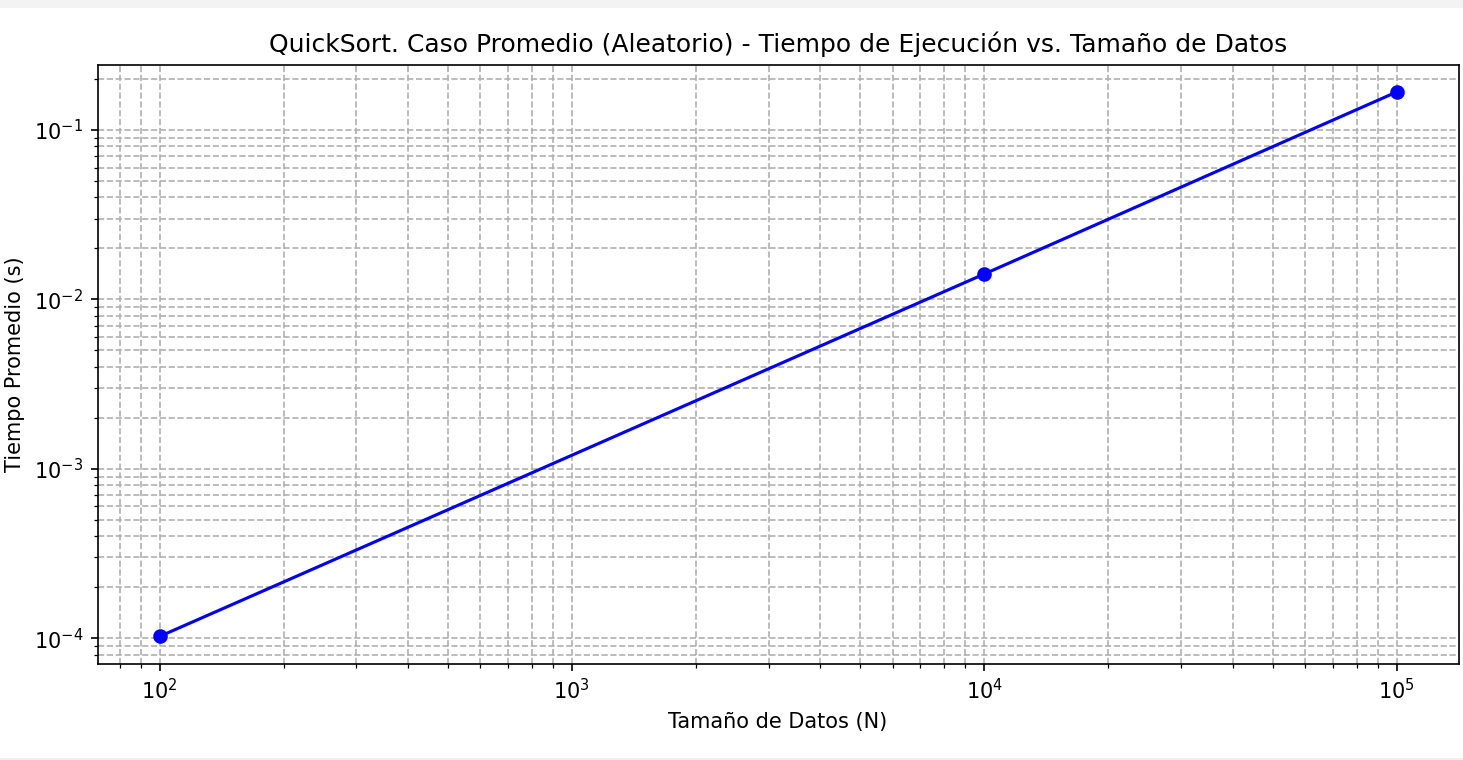
El presente capítulo se dedica al análisis empírico de la eficiencia computacional del algoritmo QuickSort, tal como se implementó en Python. Se examinarán los resultados obtenidos en términos de tiempo de ejecución y uso de memoria para diferentes tamaños de datos y casos de entrada, contrastándolos con las predicciones de la complejidad teórica. Las mediciones de tiempo se expresan en segundos (s) y el uso de memoria en megabytes binarios (MiB), siendo esta última unidad el estándar de reporte de la librería *memory\_profiler* utilizada y una aproximación aceptable a los megabytes (MB).

#### Tiempo de Ejecución

El tiempo de ejecución de QuickSort fue medido para conjuntos de datos crecientes, abarcando el caso promedio (datos aleatorios) y los casos que teóricamente podrían degenerar al peor escenario.

* **Caso Promedio (Datos Aleatorios)**

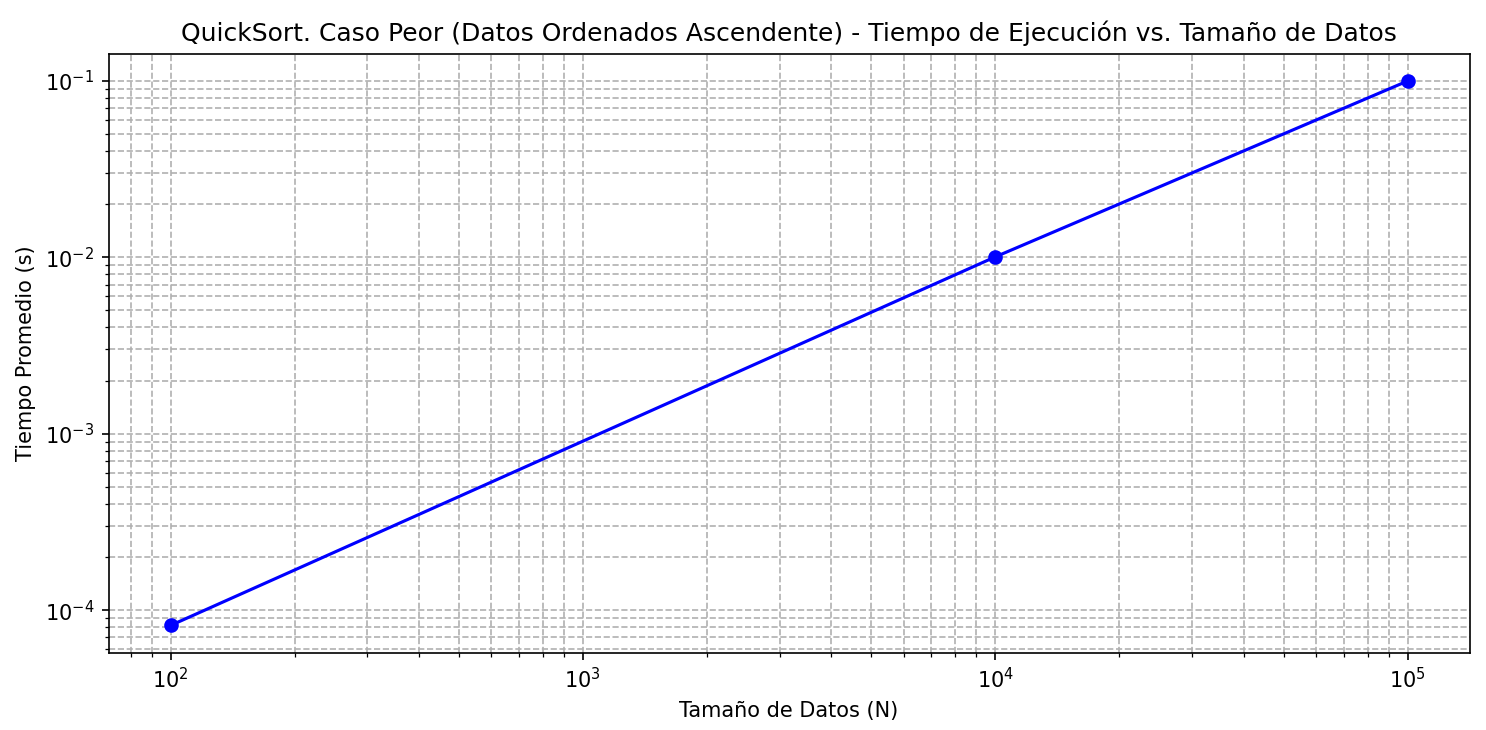
Para el caso promedio, donde los datos de entrada son aleatorios, la complejidad teórica de QuickSort es O(NlogN). Los resultados empíricos para el tiempo de ejecuciónse visualizarán en el Gráfico 1: QuickSort. Caso Promedio (Aleatorio) - Tiempo de Ejecución vs. Tamaño de Datos

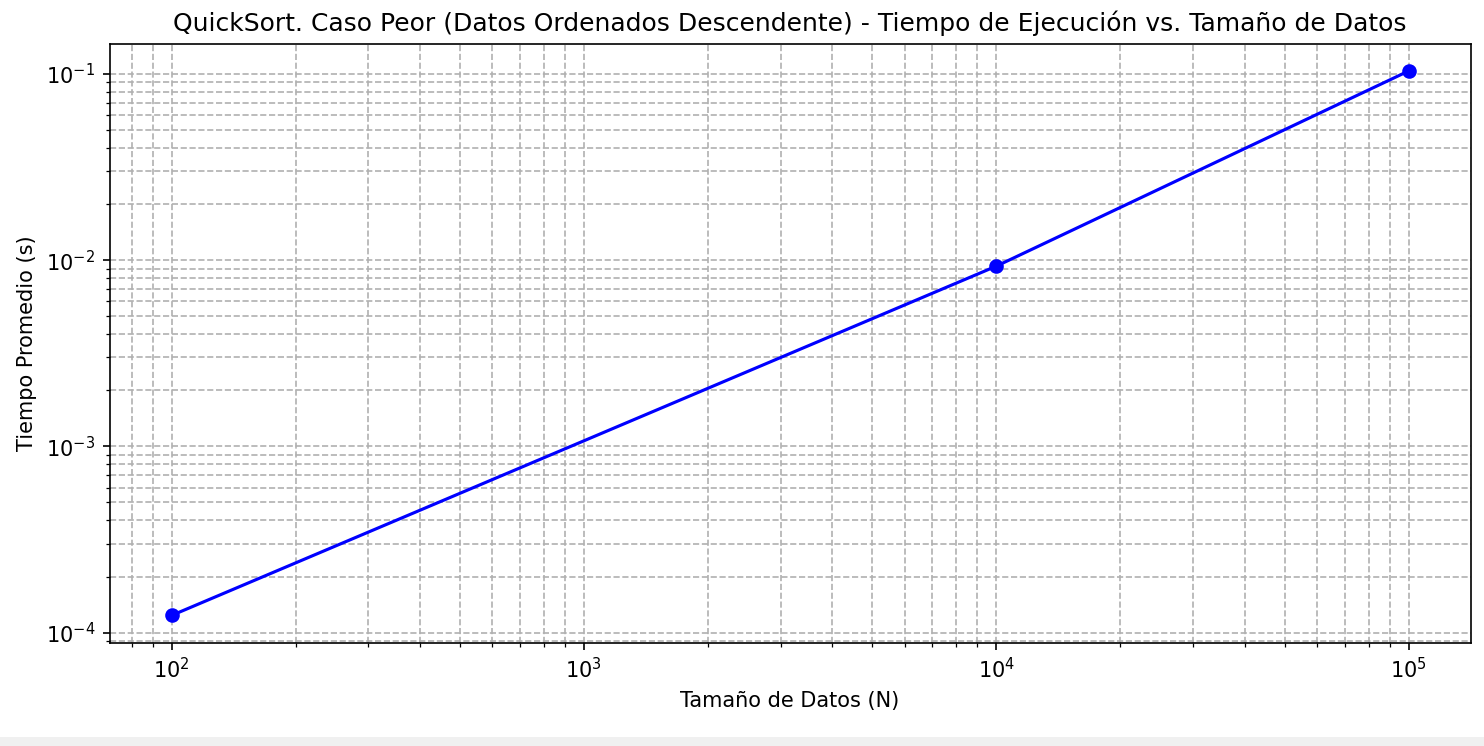


Como se observa en el gráfico, la relación entre el tamaño de los datos (N) y el tiempo de ejecución promedio en escala logarítmica para ambos ejes muestra una **línea recta con una pendiente positiva, cercana a 1**. Este comportamiento es consistente con la complejidad teórica de O(NlogN), ya que una función log-lineal en un gráfico log-log tiende a manifestarse con una pendiente ligeramente superior a la unidad, indicando un crecimiento eficiente y escalable a medida que el tamaño de los datos aumenta. Esto valida la predicción del análisis a priori sobre la eficiencia temporal de QuickSort en escenarios típicos.

* **Casos de Prueba con Datos Ordenados (Ascendente y Descendente)**

Teóricamente, el peor caso de QuickSort es O(N2), lo cual ocurre cuando la selección del pivote resulta consistentemente en divisiones desbalanceadas, como sucede con pivotes fijos (primer o último elemento) en listas ya ordenadas (Cormen et al., 2009). Sin embargo, la implementación utilizada en esta investigación selecciona el pivote central (*arr[len(arr) // 2]*). Los resultados para datos ordenados ascendentemente y descendentemente se ilustrarán en los Gráficos 2 y 3: QuickSort. Caso Peor (Datos Ordenados Ascendente) - Tiempo de Ejecución vs. Tamaño de Datos y QuickSort. Caso Peor (Datos Ordenados Descendente) - Tiempo de Ejecución vs. Tamaño de Datos respectivamente.





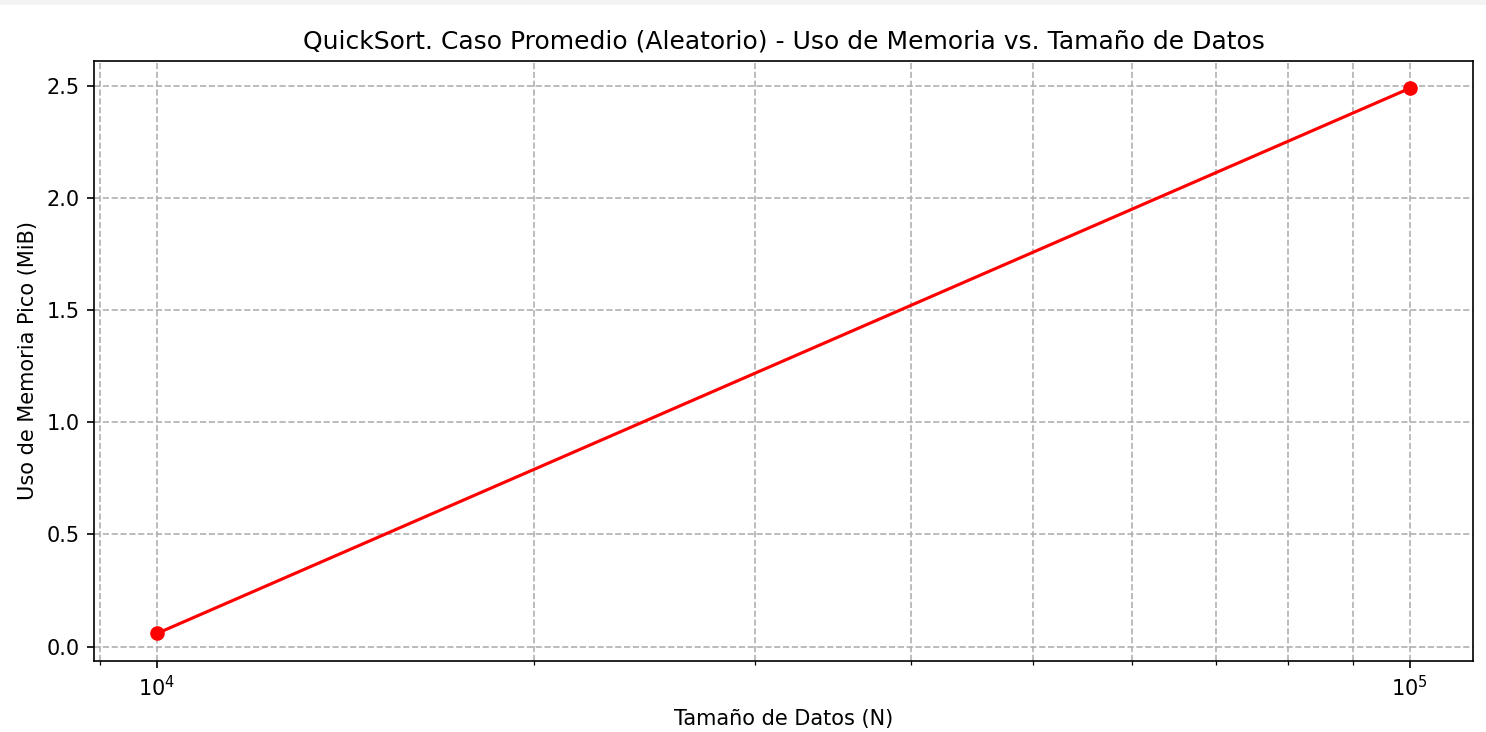
De manera notable, ambos gráficos, correspondientes a los datos ordenados ascendentemente y descendentemente, también exhiben una **línea recta con una pendiente cercana a 1 en la escala log-log**. Este hallazgo indica que, a pesar de ser teóricamente condiciones que podrían llevar al peor caso O(N2) para otras estrategias de pivote, la implementación con pivote central de QuickSort mantiene un rendimiento similar al caso promedio, es decir, cercano a O(NlogN). Esto se debe a que el pivote central en una lista ordenada tiende a dividirla de forma equilibrada, evitando la degeneración del rendimiento. Este comportamiento demuestra la robustez de esta estrategia de pivote ante este tipo de entradas.

#### Uso de Memoria

El uso de memoria es una métrica crucial para evaluar la eficiencia espacial de un algoritmo. Para QuickSort, la complejidad espacial teórica promedio es O(logN) debido a la profundidad de la pila de recursión (Hulín, 2017).

* **Caso Promedio (Datos Aleatorios)**

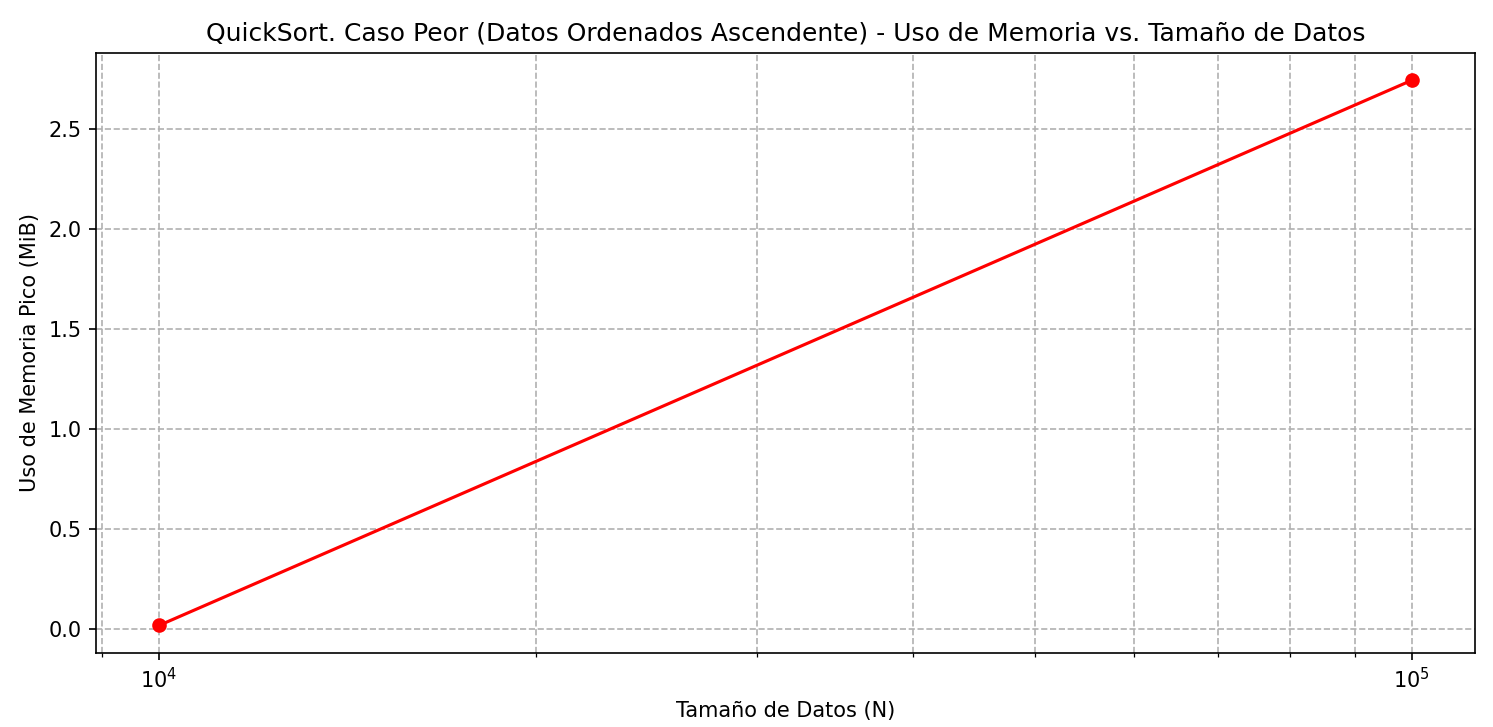
El uso de memoria para el caso promedio se visualizará en el Gráfico 4: QuickSort. Caso Promedio (Aleatorio) - Uso de Memoria vs. Tamaño de Datos]. Solo se utiliza MiB por cuestiones de la librería *memory\_profiler.*

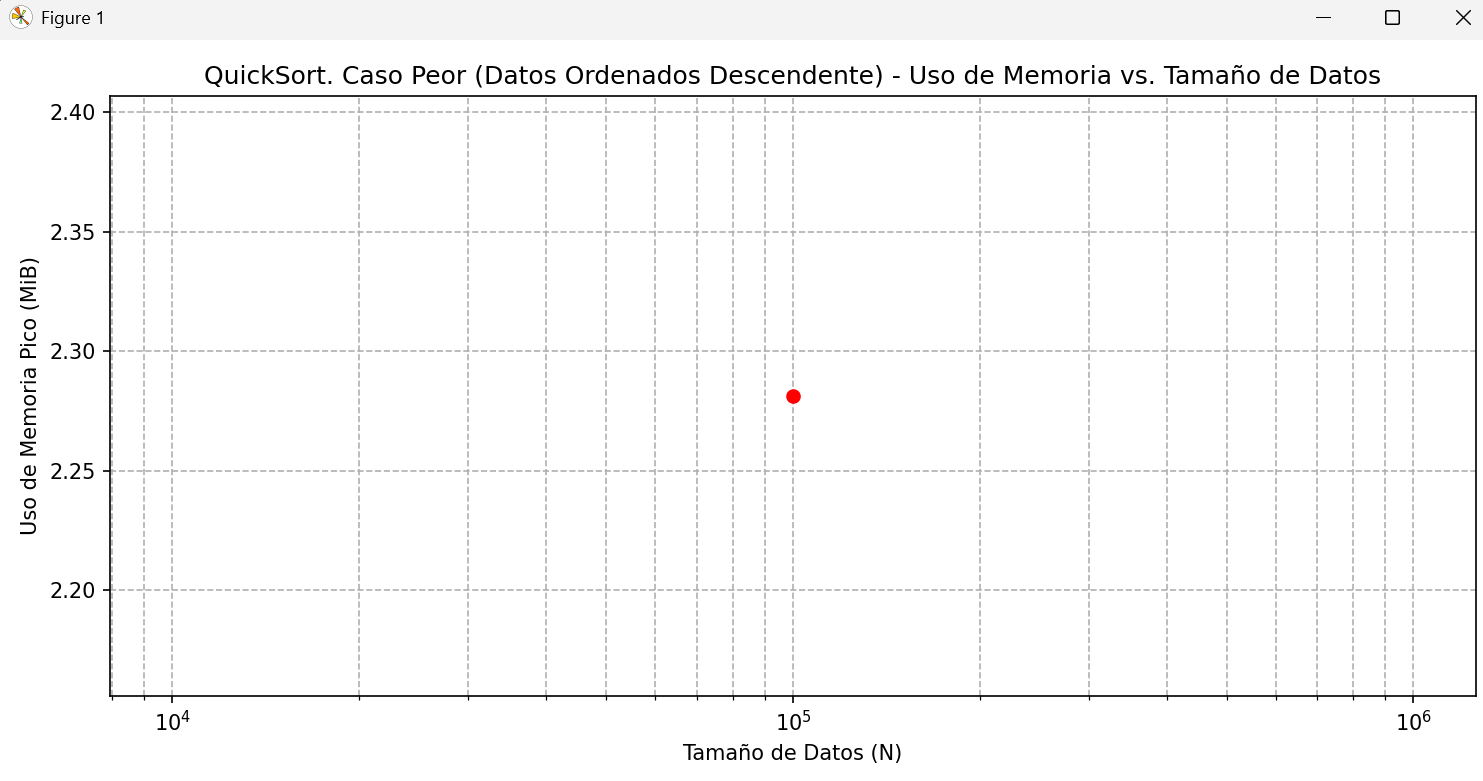


El gráfico muestra una línea recta con una pendiente positiva en la escala logarítmica, lo que sugiere un crecimiento aproximadamente lineal con N para el uso de memoria. Esta observación difiere de la complejidad espacial teórica de O(logN). La razón de esta discrepancia radica en la implementación específica de QuickSort en Python, que utiliza listas por comprensión para crear las sublistas *left, middle, y right* en cada paso recursivo. La creación de múltiples objetos de lista intermedios y las copias de datos asociadas, junto con el overhead propio del intérprete de Python en la gestión de memoria, contribuyen a un consumo de memoria que puede escalar más cerca de O(N) en la práctica para los rangos de datos probados, dominando el O(logN) de la pila de recursión (Marcellino et al., 2021).

* **Casos de Prueba con Datos Ordenados (Ascendente y Descendente)**

El uso de memoria para los datos ordenados se ilustrará en los Gráficos 5 y 6: QuickSort. Caso Peor (Datos Ordenados Ascendente) - Uso de Memoria vs. Tamaño de Datos y QuickSort. Caso Peor (Datos Ordenados Descendente) - Uso de Memoria vs. Tamaño de Datos respectivamente. Solo se utiliza MiB por cuestiones de la librería *memory\_profiler.*



****

Para el caso de datos ordenados ascendentemente, el gráfico de uso de memoria también presenta una **línea recta con pendiente positiva**, similar al caso promedio. Esto refuerza la idea de que la creación de objetos intermedios en Python es un factor dominante en el consumo de memoria para esta implementación, incluso cuando el algoritmo no experimenta su peor caso de tiempo de ejecución.

Para el caso de datos ordenados descendentemente, el gráfico solo muestra un **único punto de datos** significativo para el tamaño de 105 elementos. Esta representación con un solo punto dificulta la inferencia de una tendencia de crecimiento. No obstante, el valor de memoria pico registrado para este tamaño de datos (aproximadamente 2.28 MiB) es comparable al de otros casos, sugiriendo un comportamiento de memoria similar para ese punto específico. La ausencia de puntos para otros tamaños podría deberse a que el uso de memoria fue muy bajo o constante y no varió significativamente para el rango de datos más pequeños, o a una limitación en la recolección de datos para este caso particular.

#### Conclusiones sobre la Eficiencia de QuickSort

Los resultados empíricos del análisis de QuickSort en Python demuestran que:

* **Eficiencia Temporal:** La implementación con selección de pivote central exhibe un rendimiento temporal altamente eficiente, cercano a O(NlogN), no solo en el caso promedio con datos aleatorios, sino también en escenarios con datos ordenados (ascendente y descendente). Esto sugiere que esta estrategia de pivote es robusta y evita la degeneración al peor caso O(N2) en estas condiciones, lo que es una ventaja práctica para el algoritmo.
* **Uso de Memoria:** Si bien la complejidad espacial teórica de QuickSort es O(logN), la implementación en Python, debido a la creación de nuevas listas en cada paso recursivo, tiende a mostrar un consumo de memoria que se comporta más cerca de O(N). Esta es una consideración importante para el desarrollo de software en entornos con recursos limitados, donde la optimización de memoria podría requerir enfoques alternativos (ej., partición *in-place* para evitar la creación de nuevas listas).

Estos hallazgos validan la hipótesis general respecto al tiempo de ejecución (el crecimiento del tiempo de QuickSort es log-lineal), pero revelan una diferencia en el comportamiento del uso de memoria que puede atribuirse a las características del lenguaje de programación y la implementación específica, en lugar de una desviación del comportamiento teórico del algoritmo en sí mismo.

## Hashing con Encadenamiento Separado

### Variables, métricas y parámetros experimentales

En las pruebas realizadas se registraron diversas variables e indicadores para evaluar el rendimiento del algoritmo de hashing con encadenamiento separado. A continuación se documenta cada uno, indicando su propósito, cálculo y relevancia:

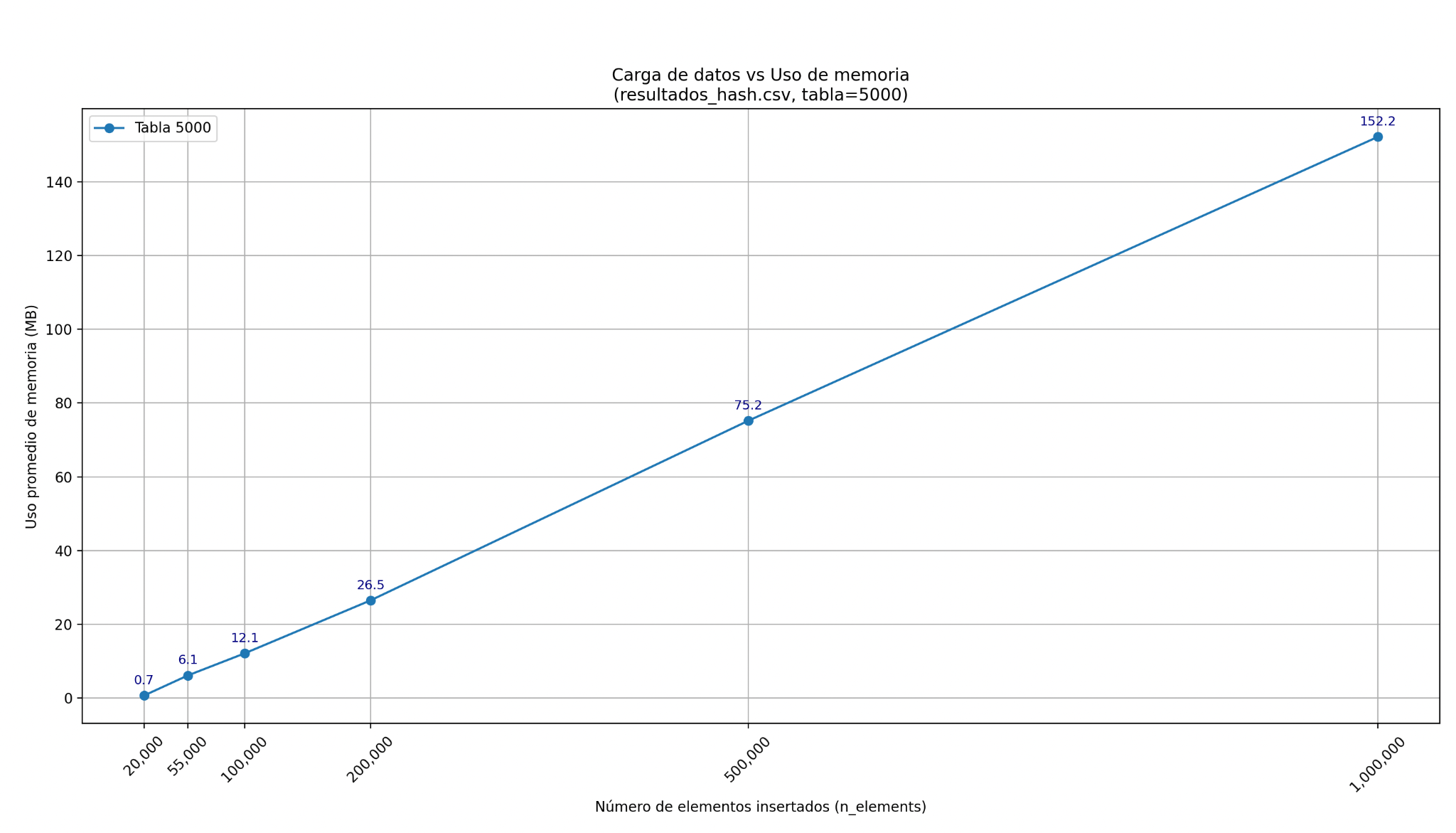
* **n\_elements (número de elementos):** Cantidad de claves-valores insertados en la tabla hash durante la prueba. Define la carga de datos introducida en la estructura. Un número mayor de elementos implica más operaciones y potencialmente más colisiones, afectando los tiempos de inserción, búsqueda y eliminación. Es fundamental para analizar la escalabilidad temporal y espacial del algoritmo conforme crece el tamaño de entrada.
* **table\_size (tamaño de la tabla):** Número de slots o cubetas disponibles en la tabla hash. Equivale al tamaño del array subyacente donde se almacenan las listas enlazadas de colisiones. Este parámetro determina la capacidad base de la tabla; un valor más alto reduce el factor de carga (ver abajo) para un mismo número de elementos, potencialmente disminuyendo colisiones. Es relevante para estudiar el impacto de la dimensión de la tabla en el rendimiento (p.ej., comparar tablas más espaciadas vs. más saturadas).
* **factor\_carga (α, factor de carga):** Es la razón número de elementos / tamaño de tabla (α = n\_elements / table\_size). Indica el nivel de llenado o saturación de la tabla hash. Un factor de carga bajo (α << 1) significa muchos slots libres en relación a elementos (pocas colisiones en promedio), mientras que un factor alto (α >> 1) implica largas listas de encadenamiento. Este factor es crítico en el análisis de rendimiento, ya que influye directamente en la complejidad promedio de las operaciones: en encadenamiento separado, el tiempo promedio de operación es O(1 + α); a mayor α, más nodos se esperan recorrer en una operación.
* **insert\_time, search\_time, delete\_time (tiempo de inserción, búsqueda, eliminación):** Tiempo total empleado en ejecutar cada tipo de operación fundamental sobre la tabla hash. Se miden en segundos usando time.perf\_counter() antes y después de realizar las operaciones en bloque. En las pruebas:
  + Tiempo de inserción corresponde al tiempo para insertar n\_elements elementos (en la prueba individual) o el promedio de los tiempos de inserción en múltiples repeticiones (en el experimento completo). Es relevante para evaluar la eficiencia de la operación insertar bajo distintas cargas (cómo crece con n y α).
  + Tiempo de búsqueda corresponde al tiempo para realizar n\_search búsquedas (se usó n\_search = 1000 en experimentos agregados) incluyendo búsquedas exitosas e infructuosas. Mide la eficiencia de lookup: búsquedas más lentas pueden indicar largas listas de colisión (sobre todo en búsquedas no existentes, que recorren la lista completa).
  + Tiempo de eliminación es el tiempo para n\_delete eliminaciones (también 1000 en los experimentos agregados), con casos existentes y no existentes. Indica la eficiencia de remover elementos y, similar a búsqueda, depende de la longitud de las cadenas en cada slot.
* **memory\_usage (uso de memoria):** Consumo máximo de memoria RAM durante la fase de inserción de elementos, medido en megabytes (MB). Se obtiene utilizando la biblioteca memory\_profiler, capturando la diferencia entre el pico de memoria y la memoria base antes de iniciar inserciones. Este valor refleja principalmente la memoria adicional utilizada para almacenar las estructuras de datos internas (nodos de las listas enlazadas, arreglos, etc.) al insertar n\_elements. Es relevante para analizar la eficiencia espacial del algoritmo, el overhead por elemento almacenado y cómo incide el factor de carga en el requerimiento de memoria.
* **total\_exec\_time (tiempo total de ejecución):** Tiempo total en segundos que duró la prueba completa, incluyendo todas las operaciones (inserciones, búsquedas, eliminaciones) y las mediciones de memoria cuando aplican. Sirve para tener una idea del costo global de cada experimento. En la prueba personalizada incluye también el overhead de instrumentación (por ejemplo, el perfilador de memoria agrega demoras significativas), por lo que no representa únicamente las operaciones algorítmicas sino el tiempo de la prueba en conjunto. En las pruebas repetitivas este dato se usó para verificar la duración total de todas las repeticiones.
* **Promedios y desviaciones estándar:** En el experimento completo con repeticiones, para cada métrica de tiempo y memoria se calcularon el promedio (media aritmética) y la desviación estándar a partir de 10 ejecuciones repetidas por configuración. El promedio sintetiza el rendimiento típico esperado, mientras que la desviación estándar cuantifica la variabilidad o consistencia de los resultados. Una desviación estándar baja relativa al promedio indica resultados muy consistentes entre repeticiones (alta confiabilidad), mientras que una desviación alta sugiere fluctuaciones notables entre ejecuciones (posibles efectos aleatorios, de caché, garbage collector, etc.). Estas medidas estadísticas permiten evaluar la confiabilidad de los datos experimentales y dar rigor al análisis comparativo.

Adicionalmente, en la configuración de las pruebas de búsqueda y eliminación se consideró que el 50% de las consultas fueran sobre claves inexistentes (parámetro percent\_non\_existing = 0.5 en las funciones de prueba). Esto fue deliberado para simular casos promedio realistas: en búsqueda/eliminación, aproximadamente la mitad de las operaciones recorren la lista completa (cuando la clave no está) y la otra mitad la terminan antes (cuando la clave sí existe). Este detalle influye en los tiempos medios de búsqueda y borrado, ya que una búsqueda infructuosa cuesta proporcional al largo de la lista en esa cubeta, mientras que una exitosa típicamente es más rápida. Incluimos este factor en la interpretación de resultados de búsqueda/eliminación.

## 

### Perfil de uso de memoria bajo distintas cargas

Para examinar cómo varía el consumo de memoria con la cantidad de datos almacenados, se fijó un tamaño de tabla y se midió el uso de memoria a diferentes niveles de carga. En particular, la Figura 1 ilustra la relación entre el número de elementos insertados y la memoria utilizada, manteniendo la tabla con un tamaño fijo de 5000 slots (m=5000):

Figura 1. Relación entre el número de elementos insertados (n\_elements) y el uso promedio de memoria (MB) en la tabla hash de tamaño fijo m = 5000 (encadenamiento separado). Cada punto corresponde a una configuración experimental (promedio de 10 repeticiones); se indican algunos valores de n\_elements destacados en la escala horizontal.

Como se observa en la figura, el uso de memoria crece de forma aproximadamente lineal con el número de elementos almacenados. Por ejemplo, insertar 50,000 elementos (α ≈ 10 en m=5000) consumió en promedio unos ~6 MB de memoria, subir a 200,000 elementos (α = 40) elevó el uso a ~26.5 MB, y llegar a 1,000,000 de elementos (α = 200) requirió alrededor de 152 MB de memoria. Esto sugiere que cada elemento adicional aporta un consumo casi constante de memoria. De hecho, estimando la pendiente, cada elemento ocupa en promedio del orden de ~150 bytes (incluyendo su nodo, referencias y datos), valor coherente con la estructura de almacenamiento utilizada.

Cabe notar que el factor de carga en sí no incrementa el uso de memoria por elemento de manera significativa. En encadenamiento separado, todos los elementos almacenados siempre ocupan espacio para al menos un nodo, independientemente de cómo estén distribuidos en las cubetas. Una tabla más saturada (α alto) no requiere memoria extra por elemento en comparación con una más holgada con igual número de elementos; la única diferencia de memoria proviene del tamaño del array base (table\_size), cuyo impacto es mínimo. Por ejemplo, ampliar la tabla de 2000 a 5000 slots implica apenas unos ~24 KB adicionales (asumiendo punteros de 64 bits, cada slot extra ~8 bytes), que es despreciable frente a los varios MB ocupados por cientos de miles de nodos. En nuestras mediciones, las pequeñas discrepancias • • 2 iniciales (ejemplos: el punto de 20k elementos en la figura aparece ligeramente por debajo de la línea esperada) se atribuyen a la gestión interna de memoria de Python: para cantidades muy bajas, el intérprete pudo reutilizar memoria ya reservada, retrasando la solicitud de nuevos bloques al sistema. Sin embargo, en rangos mayores la relación se estabiliza claramente lineal. En resumen, la eficiencia espacial del algoritmo es O(n), creciendo proporcionalmente con el número de elementos almacenados, tal como se anticipa teóricamente.

### Resultados agregados de múltiples ejecuciones (resultados\_hash.csv)

Se realizaron experimentos sistemáticos variando la cantidad de datos y el tamaño de la tabla, con múltiples repeticiones para obtener promedios fiables. En total se evaluaron 18 configuraciones combinando diferentes valores de n\_elements y table\_size. En particular, se probaron tamaños de datos desde 20,000 hasta 1,000,000 de elementos, frente a tablas de 1000, 2000 y 5000 slots. Estos rangos abarcan factores de carga desde moderados (α ≈ 4 cuando 20k elementos en tabla de 5000) hasta extremadamente altos (α = 1000 con 1e6 elementos en tabla de 1000). Cada configuración fue repetida 10 veces y se registraron los tiempos promedio de inserción, búsqueda y eliminación, así como el uso promedio de memoria, junto con sus desviaciones estándar. A continuación presentamos el análisis separado de los resultados de tiempo y memoria.

#### Rendimiento de tiempo

Los tiempos de ejecución medidos para las operaciones de inserción, búsqueda y eliminación confirmaron las expectativas teóricas de eficiencia. En todos los casos, las operaciones se realizaron en tiempos muy reducidos, del orden de milisegundos o microsegundos, incluso al escalar la carga de datos a cientos de miles de elementos. Además, la variabilidad entre repeticiones fue baja (desviaciones estándar pequeñas), lo que indica un comportamiento consistente del algoritmo.

##### Tiempo de inserción

La operación de inserción mostró un crecimiento casi lineal del tiempo total conforme aumentó n\_elements, sin evidenciar cuellos de botella inesperados. Por ejemplo, insertar 100 mil elementos en la tabla más grande probada (m=5000, α≈20) tomó en promedio on the order of 0.7–0.8 segundos, mientras que la misma inserción en la tabla más pequeña (m=1000, α≈100) tomó alrededor de 1–2 segundos. Al duplicar o triplicar el número de elementos, el tiempo de inserción total tendió a multiplicarse en proporción similar. Esto concuerda con la complejidad esperada O(n) para realizar n inserciones (cada inserción es O(1) amortizado, y repetirla n veces da O(n)).

Sin embargo, cuando el factor de carga se incrementa notablemente, se observa un ligero sobrecosto adicional por elemento debido a las colisiones. Por ejemplo, con tabla de 1000 slots, al pasar de 20k a 100k elementos (α aumentando de 20 a 100, un factor 5 en n), el tiempo total de inserción creció más que 5×, indicando que el costo unitario por inserción aumentó bajo α más elevados. En efecto, cada inserción nueva debe recorrer en promedio la mitad de la lista en la cubeta correspondiente para verificar duplicados antes de encadenar el nuevo nodo. Conforme α crece, esas listas son más largas: en el caso extremo de α ≈ 1000 (1e6 elementos en tabla de 1000), cada inserción recorre unas ~500 comparaciones en promedio. Esto añade un componente lineal en α al tiempo de inserción, coherente con la formulación O(1 + α). Aun así, es importante resaltar que incluso con factores de carga muy altos, las inserciones siguieron siendo rápidas en valor absoluto. Por ejemplo, insertar 300k elementos con α=150 tomó apenas ~2.3 s en total, lo 3 que implica un tiempo promedio por inserción de ~7.6 μs; incluso con α extremadamente grande, el algoritmo pudo manejar on the order of 10^5 operaciones por segundo. No se observó ninguna degradación catastrófica: el impacto de las colisiones se manifestó de forma gradual y lineal, no exponencial, manteniendo las inserciones en escalas de tiempo manejables (micro o milisegundos por operación).

##### Tiempos de búsqueda e eliminación

Las operaciones de búsqueda y borrado presentaron patrones similares de eficiencia. Para cada configuración se realizaron 1000 búsquedas y 1000 eliminaciones mixtas (éxitos y fracasos). En escenarios de baja carga (e.g. n=20k, m=5000, α≈4), 1000 búsquedas se completaron en ~0.001–0.003 s (unos pocos microsegundos por búsqueda en promedio). En el peor de los casos medido (α muy alto, e.g. n=1e6, m=1000, α=1000), 1000 búsquedas tomaron del orden de 0.01–0.02 s – es decir, ~10– 20 μs por búsqueda en promedio. Esto significa que incluso bajo cargas extremas cada búsqueda individual siguió tomando decenas de microsegundos, lo cual es extremadamente rápido. La tendencia con respecto a α fue análoga a la de inserción: a mayor α, las búsquedas y eliminaciones se hicieron ligeramente más lentas, aproximadamente de forma lineal.

Esto se debe a que una búsqueda infructuosa recorre la lista completa de la cubeta (longitud ~α), y una exitosa recorre en promedio la mitad. Por tanto, el tiempo promedio por búsqueda tiende a crecer proporcional a α (en nuestros datos, duplicar α aproximadamente duplicó el tiempo promedio de búsqueda). De manera ilustrativa, una configuración muy cargada (por ejemplo, α≈10) produjo tiempos por búsqueda ~2 veces mayores que otra casi vacía (α≈0.2), coherente con que cada elemento extra por cubeta añade una comparación promedio. Para las eliminaciones se observaron comportamientos similares: eliminar elementos costó un poco más que buscarlos bajo cargas altas, lo cual es esperable ya que la operación de borrado en nuestra implementación requiere reencadenar los punteros de la lista enlazada además de localizar el nodo.

En todos los casos, no se detectó ninguna caída abrupta de rendimiento al aumentar la carga; únicamente un incremento progresivo y controlado del tiempo de las operaciones, acorde con la teoría. Cabe destacar que con factores de carga moderados (α hasta ~10), el desempeño temporal fue excelente – los tiempos por operación permanecieron en el orden de microsegundos, prácticamente constantes – y aun con factores de carga extremos (α de tres dígitos), el algoritmo siguió siendo operativo con sobrecostos lineales. Esto demuestra una sólida escalabilidad temporal: procesar conjuntos de datos crecientes (de miles a cientos de miles o un millón de elementos) sigue siendo factible en segundos o fracciones de segundo, cumpliendo con la complejidad esperada O(n) para n operaciones totales.

### Comportamiento del uso de memoria

Los resultados de uso de memoria corroboran que el consumo de espacio es lineal en la cantidad de elementos almacenados. Tal como se analizó en el perfil de memoria, la memoria adicional ocupada por la tabla hash creció esencialmente en proporción directa a n\_elements, sin dependencias fuertes en el factor de carga más allá de lo que implica aumentar n. En las pruebas agregadas, el uso máximo de memoria reportado abarcó desde ~0.7 MB (en las configuraciones más pequeñas, ~20k elementos) hasta alrededor de 150 MB (para ~1e6 elementos). Es importante resaltar que el tamaño de la tabla (table\_size) tuvo un efecto marginal en la memoria total consumida. Para un mismo número de elementos, usar una tabla más grande o más pequeña no cambia significativamente la memoria ocupada por elemento; casi todo el consumo proviene de los nodos de datos en las listas enlazadas, mientras que la diferencia en tamaño del array de cubetas es muy pequeña. Por ejemplo, nuestros datos muestran que almacenar ~50k elementos en m=1000 (α=50) o en 4 m=5000 (α=10) resultó en consumos de memoria prácticamente iguales (la variación estuvo dentro del margen de error de las mediciones).

La única diferencia teórica sería el espacio extra del array: pasando de 1000 a 5000 slots hay ~4000 punteros adicionales, que equivalen a ~0.032 MB, una fracción ínfima comparada con la memoria ocupada por 50k nodos (del orden de varias megabytes). Esto explica por qué en ocasiones una configuración con α más bajo puede reportar un uso de memoria similar o incluso ligeramente mayor que otra con α más alto pero menos slots: la distribución de los mismos n elementos en más cubetas no reduce el espacio por elemento, solo añade un pequeño overhead fijo. En suma, el factor de carga no impacta de forma apreciable la eficiencia espacial por sí mismo; el uso de memoria depende casi exclusivamente de n.

Otra observación de las pruebas repetidas es que, cuando el incremento absoluto de memoria es muy pequeño (unos pocos KB, típicamente en configuraciones con n relativamente bajo), la variabilidad relativa entre repeticiones puede ser alta e incluso algunas mediciones podrían indicar 0.0 MB de aumento neto. Esto se debe a la resolución limitada de la herramienta de perfilado y a la gestión interna de Python (que puede asignar memoria en bloques discretos y reutilizar espacios ya reservados).

Sin embargo, para valores grandes de n (decenas de miles en adelante), estas fluctuaciones se estabilizan y los resultados de memoria fueron consistentes. De hecho, para los mayores tamaños probados (≥100k elementos) no se hallaron discrepancias significativas entre repeticiones: la desviación estándar del uso de memoria fue baja en términos porcentuales, reforzando la confiabilidad de las tendencias observadas. En conclusión, el comportamiento espacial del algoritmo de hash con encadenamiento separado se alinea con la complejidad teórica esperada O(n): duplicar el número de elementos aproximadamente duplicó la memoria utilizada, sin indicios de sobrecostos exponenciales ni problemas de fragmentación.

Si bien el consumo absoluto de memoria en Python puede ser relevante (por ejemplo, almacenar del orden de 10^5 a 10^6 elementos ocupó decenas de MB de RAM, debido al overhead por nodo y objetos), este crecimiento fue proporcional al número de datos almacenados y, por lo tanto, predecible y manejable. Mantener un factor de carga bajo no se orienta tanto a ahorrar memoria (ya que tener la misma n en más cubetas usa prácticamente la misma memoria total), sino a mantener la eficiencia temporal de las operaciones. Solo en aplicaciones con restricciones de memoria muy estrictas podría considerarse limitar n independientemente de α; en la mayoría de escenarios, el enfoque es asegurar α razonable para rendimiento, dado que el espacio por elemento es fijo.

### Interpretación de resultados

A continuación, interpretamos los hallazgos clave en función de los objetivos específicos planteados para este experimento, contrastando los resultados obtenidos con el comportamiento teórico esperado del algoritmo de hashing con encadenamiento separado

#### Eficiencia temporal de las operaciones

Los resultados experimentales confirman una alta eficiencia computacional temporal del algoritmo. En promedio, las operaciones fundamentales mostraron tiempos de ejecución extremadamente bajos, cumpliendo la complejidad amortizada O(1) por operación. Incluso al aumentar drásticamente el número de elementos, no se observó una degradación abrupta: el tiempo por operación creció gradualmente con α, de forma cercana a lineal. Para cargas moderadas (hasta ~10k elementos, α ≤ 10), los tiempos por operación se mantuvieron en el orden de microsegundos, prácticamente constantes. Para cargas muy altas (hasta 300k–1e6 elementos, con α de decenas a centenares), los • 5 tiempos por operación escalaron a unos pocos o algunas decenas de microsegundos – un aumento perceptible pero modesto en términos absolutos.

Esto demuestra que la tabla hash con encadenamiento separado es escalable en tiempo: procesar cientos de miles de elementos sigue siendo factible en cuestión de segundos, y el costo temporal crece acorde a la teoría (aproximadamente proporcional a n y lineal en α). La búsqueda y la eliminación resultaron ligeramente más rápidas que la inserción en promedio por operación en nuestros experimentos (debido a que siempre se realizaron un número fijo de 1000 búsquedas/eliminaciones, independientemente de n, y esas operaciones no procesan todos los elementos), pero su dependencia con α fue similar. En suma, la eficiencia temporal observada es excelente: el algoritmo mantiene un rendimiento cercano a O(1) por operación en la práctica, con un aumento lineal moderado bajo factores de carga elevados pero manejables.

#### Uso de memoria bajo distintas cargas

Los experimentos demostraron que el uso de memoria crece linealmente con el número de elementos, validando la complejidad espacial O(n) esperada. Cada elemento adicional introduce un incremento aproximadamente constante en la memoria ocupada, correspondiente al espacio para su nodo y datos asociados. No se evidenció ninguna dependencia super-lineal; por el contrario, duplicar la cantidad de elementos produjo aproximadamente el doble de consumo de memoria. El factor de carga α, por sí mismo, no incrementa el consumo de memoria por elemento: las pruebas con tablas más saturadas no mostraron un gasto de memoria mayor por dato almacenado en comparación con tablas más holgadas con igual n (más allá de diferencias mínimas por el tamaño del arreglo de cubetas, que resulta insignificante).

Esto significa que, desde el punto de vista espacial, una tabla hash con encadenamiento separado puede llenarse mucho (α≫1) sin penalización en la eficiencia de uso de memoria más allá de almacenar los propios datos. En nuestras mediciones, almacenar 300k elementos ocupó en torno a 50 MB y 1 millón de elementos alrededor de 150 MB, cantidades considerables pero plenamente justificadas por la cantidad de objetos almacenados. Por lo tanto, la eficiencia espacial del algoritmo quedó corroborada: el uso de memoria es proporcional a la cantidad de elementos, y el overhead por elemento se mantuvo aproximadamente constante en diversos escenarios de carga.

## 

#### Validación vs. Complejidad teórica

Los patrones empíricos observados se alinean estrechamente con las predicciones teóricas para tablas hash con encadenamiento separado. Teóricamente, estas estructuras ofrecen tiempo promedio O(1) por operación de búsqueda/insert/eliminación, degradándose linealmente con α debido a las colisiones, y un uso de espacio O(n). Nuestros datos confirman precisamente eso: tiempos promedio prácticamente constantes para α pequeños, aumentando proporcionalmente a α (es decir, O(1 + α)) cuando la tabla se vuelve muy cargada, y consumo de memoria creciendo linealmente con n. No se encontraron desviaciones significativas que contradigan la teoría; por el contrario, cada tendencia medida respaldó el modelo teórico.

Por ejemplo, el incremento lineal de los tiempos de inserción con α (duplicar α casi duplicó el tiempo promedio por inserción), o el hecho de que el uso de memoria por elemento se mantuvo constante (~0.15 KB por elemento en promedio), son comportamientos previstos por el análisis teórico de esta estructura. Esto valida que la implementación y el algoritmo funcionan con la eficiencia computacional esperada en la práctica. Cabe mencionar que factores extrínsecos (propios del entorno de ejecución, e.g. recolección de basura de Python, asignación de memoria en bloques) introdujeron solo variaciones menores y no • • 6 alteraron las tendencias generales. En conjunto, los resultados experimentales sustentan la confiabilidad y la eficiencia del algoritmo de hash con encadenamiento separado, confirmando sus ventajas teóricas en un escenario real.

## 

### Conclusiones finales

En base a los objetivos específicos planteados y los resultados obtenidos, resumimos las conclusiones sobre el rendimiento del algoritmo de tabla hash con encadenamiento separado:

* **Eficiencia computacional (tiempo):** El algoritmo demostró una eficiencia muy alta. Las operaciones de inserción, búsqueda y eliminación se realizaron en tiempos promedio extremadamente bajos, cumpliendo con la expectativa de complejidad O(1) amortizada. Incluso al aumentar considerablemente el volumen de datos, la degradación del tiempo por operación fue gradual y acorde con un comportamiento lineal. Para tamaños moderados (unos pocos miles de elementos), los tiempos por operación permanecieron en el orden de microsegundos, y para cargas muy altas (cientos de miles a un millón de elementos, con factores de carga de hasta 1000) se mantuvieron en el orden de decenas de microsegundos. Esto demuestra que el algoritmo escala bien en términos de tiempo: incluso procesar 10^5–10^6 elementos es factible en unos segundos, sin que surjan cuellos de botella inesperados.
* **Impacto del factor de carga (α):** El factor de carga resultó ser la variable clave para el rendimiento. Se comprobó que a mayor α:
  + Aumentan linealmente las colisiones y por tanto el tiempo promedio de las operaciones (demostrado por el incremento de ~2× en tiempo al pasar de α≈1 a α≈10 en nuestros datos). Esto valida la importancia de mantener α en rangos bajos para garantizar operaciones prácticamente O(1).
  + No afecta significativamente el uso de memoria por elemento (la memoria total crece con n, no con α directamente). Sin embargo, un α alto implica que se está guardando una cantidad *n* muy grande en una tabla fija, lo que en la práctica quiere decir muchos elementos -> mucha memoria total. Es decir, α es indicativo de cuán saturada está la tabla; saturaciones altas no consumen memoria extra aparte de los propios datos, pero sí pueden señalar que se está llegando al límite de capacidad deseable por motivos de tiempo. En una implementación típica, al sobrepasar cierto α máximo se **redimensionaría** la tabla para volver a bajar α, logrando así un trade-off entre tiempo y espacio. Nuestra implementación no re-dimensionó, permitiéndonos estudiar qué ocurre con α altos: vimos rendimiento degradado pero lineal, sin colapso. Esto sugiere que no hay un punto crítico rígido desde el punto de vista de corrección, pero sí **factores prácticos** (α entre 1 y 3 suelen recomendarse como óptimos) donde el desempeño es óptimo. Más allá, el rendimiento decae y eventualmente puede hacerse lineal en n si α crece proporcional a n.
* **Variabilidad y confiabilidad de resultados:** : Las mediciones fueron consistentes a lo largo de las repeticiones, especialmente en los escenarios de mayor carga donde los tiempos y memorias involucrados eran sustanciales. Esto brinda confianza en la validez de las tendencias observadas. Pequeñas anomalías en configuraciones de carga muy baja (p. ej., un caso donde se reportó 0 MB de aumento de memoria) se atribuyen a limitaciones de medición y no afectan las conclusiones generales. En general, los patrones empíricos identificados concuerdan con las propiedades teóricas de las tablas hash con encadenamiento separado: tiempos promedio constantes O(1) (creciendo linealmente con α) y uso de memoria lineal O(n).
* **Escalabilidad y comportamiento bajo carga creciente:** Bajo cargas crecientes, la escalabilidad de la tabla hash con encadenamiento separado fue excelente. A medida que incrementamos el tamaño de los datos de forma masiva (en órdenes de magnitud), el tiempo total de las operaciones aumentó de forma aproximadamente lineal, en línea con la complejidad esperada O(n) para procesar n operaciones. No se observaron saltos inesperados ni estrangulamientos de rendimiento. De igual modo, la escala espacial se comportó linealmente: duplicar el número de elementos aproximadamente duplicó la memoria utilizada, sin indicios de consumos super-lineales. El overhead absoluto por elemento (del orden de pocas centenas de bytes en Python) hace que almacenar millones de elementos requiera decenas de MB de RAM, pero este crecimiento fue proporcional y predecible. No se detectaron problemas de fragmentación u otros efectos que impidan a la estructura manejar grandes volúmenes de datos.

En conclusión, el algoritmo de tabla hash con encadenamiento separado mostró excelente eficiencia computacional en las pruebas, cumpliendo las predicciones teóricas: tiempos promedio constantes y uso de memoria lineal. Un factor de carga elevado degrada gradualmente el rendimiento temporal (operaciones más lentas proporcionalmente a α), por lo que en aplicaciones reales se justifica mantener α controlado (ya sea eligiendo un tamaño de tabla suficientemente grande de inicio o expandiéndose dinámicamente). No obstante, dentro de los rangos examinados, incluso α de dos dígitos fue manejable. Los resultados experimentales fueron consistentes, reforzando la confiabilidad de la evaluación. En suma, las tablas hash con encadenamiento separado se confirman como una estructura muy eficaz, escalable y estable bajo distintas cargas de trabajo, ofreciendo tiempos de acceso prácticamente constantes y aprovechando bien la memoria para brindar rápidas operaciones de inserción, búsqueda y borrado. Los patrones empíricos identificados se alinean con la complejidad promedio O(1) (y O(α) en casos degenerados) de estas operaciones, validando su eficiencia computacional en la práctica.

# 

# 

# Referencias

Aftab, A., Ali, M. A., Ghaffar, A., Shah, A. U. R., Ishfaq, H. M., & Shujaat, M. (2021). Review on performance of quick sort algorithm.

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

García Hernández, P. (2021).*Evaluación empírica de tablas hash*.

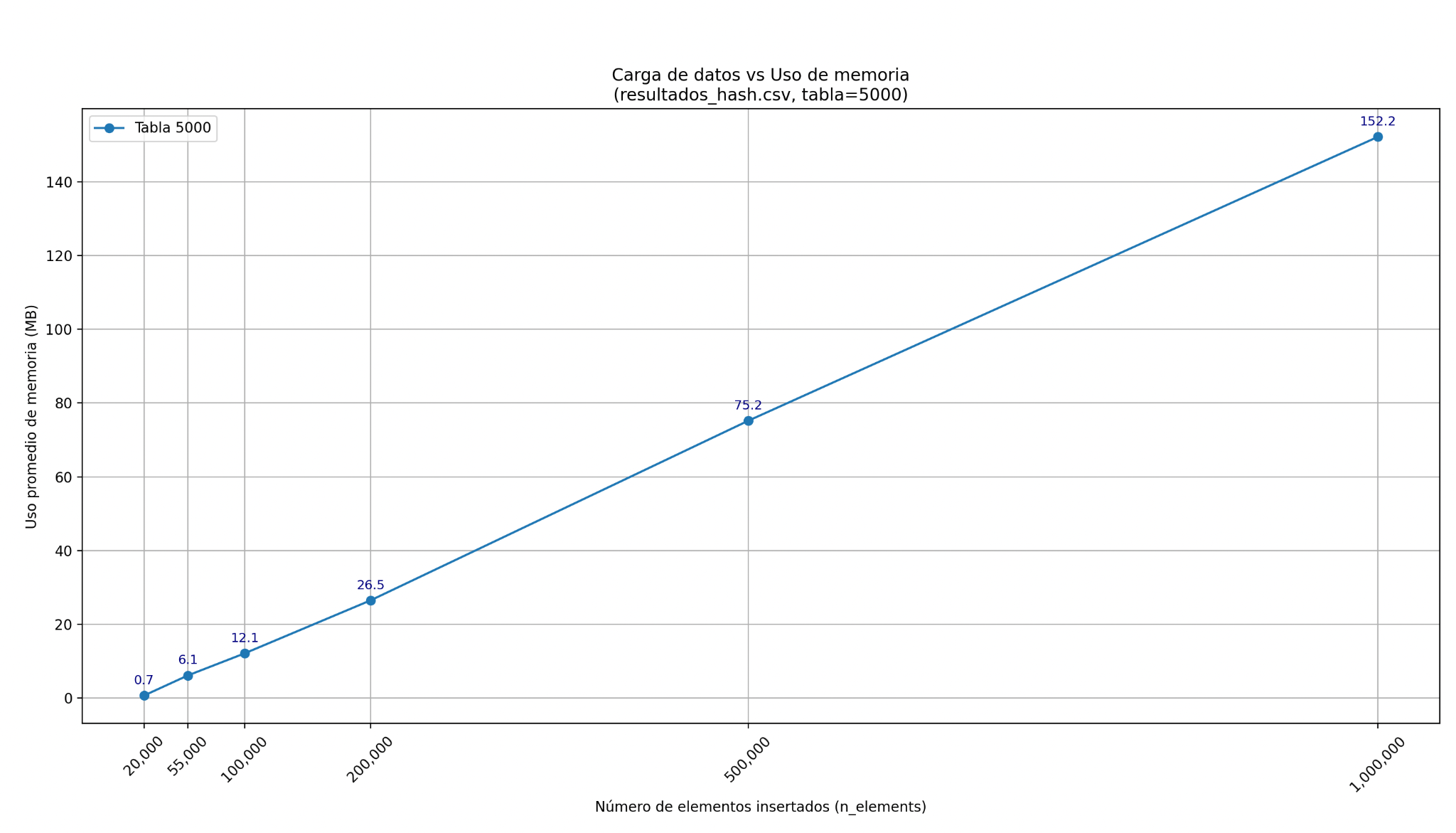
Hulín, M. (2017). *Performance analysis of Sorting Algorithms* (Doctoral dissertation, Thesis, Masaryk University Faculty of Informatics, Pole-Ponava, Czechia).

Liu, D., Cui, Z., Xu, S., & Liu, H. (2014, 4-6 de junio). An empirical study on the performance of hash table. En *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)* (pp. 477-484). IEEE.

Marcellino, M., Pratama, D. W., Suntiarko, S. S., & Margi, K. (2021, 27-28 de octubre). Comparative of advanced sorting algorithms (quick sort, heap sort, merge sort, intro sort, radix sort) based on time and memory usage. En *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)* (Vol. 1, pp. 154-160). IEEE.

Xiang, W. (2011, 18-20 de noviembre). Analysis of the time complexity of quick sort algorithm. En *2011 International Conference on Information Management, Innovation Management and Industrial Engineering* (Vol. 1, pp. 408-410). IEEE.

# Anexos

**Figura 1.** Documento de instrucción de instalación. Instrucciones de instalación (PruebaHash)

**Documento de instrucción de instalación.** [Instrucciones de instalación (PruebaHash)](https://docs.google.com/document/d/1j-Vqqf0R8PYgVQuEFCuHxQg37v7xOMNhnhuVnfMnXSA/edit?usp=sharing)

**Repositorio de GitHub.**<https://github.com/AliciaEA/InvAlgEstDatos_Qck_Hsh.git>