Tarea N°3:

Algoritmos de Ordenamiento y Funciones de Hash

Joaquín Ignacio Ormazábal Acevedo
Departamento de Ingeniería Eléctrica, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile
20 de Octubre, 2024

***Abstract*—Este informe presenta el desarrollo e implementacion de algoritmos de ordenamiento y funciones hash como parte de la tarea asignada en el curso EL4203: Programación Avanzada. Se abordaron tres problemas relacionados con algoritmos de ordenamiento: (1) optimización de *QuickSort* mediante la estrategia de Mediana de Tres, (2) diseño de un *Radix Sort* para ordenar datos alfanuméricos, y (3) implementación del algoritmo *CombSort* para comparar su desempeño con *Bubble Sort*. Además, se implementaron tres soluciones basadas en funciones de hash: (1) una tabla hash con resolución de colisiones mediante doble hashing, (2) una función de hash para identificar duplicados en archivos de texto, y (3) un programa para identificar elementos frecuentes en grandes volúmenes de datos. Los resultados incluyen análisis de eficiencia, comparaciones experimentals y visualización de los tiempos de ejecución, destacando los beneficios de las técnicas implementadas en distintos contextos. Finalmente, se presentan conclusiones generales sobre el impacto de las soluciones propuestas y posibles mejoras futuras.**

I. Introduction

En el contexto del curso EL4203: Programación Avanzada, esta tarea busca fortalecer las habilidades de diseño e implementación de algoritmos avanzados en Python, centrándose en dos áreas clave: algoritmos de ordenamiento y funciones de hash. Estos conceptos son fundamentales en informática debido a su amplia aplicación en el manejo eficiente de datos.

El informe se estructura en torno a dos preguntas principales. La primera aborda la optimización y comparación de algoritmos de ordenamiento, un tema relevante para mejorar el rendimiento en sistemas que procesan grandes volúmenes de datos. En particular, se implementaron y analizaron *QuickSort* con Mediana de Tres, *Radix Sort* para datos alfanuméricos, y *CombSort*, evaluando sus tiempos de ejecución y casos de uso.

La segunda pregunta se enfoca en las funciones de hash, herramientas críticas para la organización y búsqueda eficiente en estructuras de datos. Se desarrollaron tres aplicaciones prácticas: una tabla hash con doble hashing para resolver colisiones, una función para detectar duplicados en archivos de texto, y un programa para identificar elementos frecuentes en grandes volúmenes de datos, comparando su eficiencia con bibliotecas estándar.

Este informe documenta brevemente la metodología empleada, los resultados experimentales obtenidos y un análisis detallado del rendimiento y aplicabilidad de las soluciones, ofreciendo una base para futuras mejoras y extensiones. Para mayor profundidad, revisar el archivo ipynb adjunto a esta tarea. En este último archivo se documenta el funcionamiento de cada función implementada en este tarea.

II. Metodología

A. Pregunta 1: Algoritmos de Ordenamiento

*1) **QuickSort con Mediana de Tres**:* La implementación del algoritmo QuickSort con Mediana de Tres introduce una mejora significativa en la selección del pivote, lo que permite optimizar su rendimiento en comparación con la versión clásica de QuickSort. Esta variante utiliza la mediana entre el primer elemento, el elemento central y el último elemento del subarreglo actual como pivote. Esta estrategia reduce el impacto de los peores casos, como cuando el arreglo ya está ordenado o casi ordenado, y mejora la eficiencia promedio del algoritmo.

En primer lugar, se seleccionan tres elementos clave del subarreglo: el primero, el central y el último. Estos elementos se ordenan, y el valor que se encuentra en el medio de los tres se elige como el pivote. Posteriormente, este pivote se intercambia con el último elemento del subarreglo para mantener la estructura esperada por la función de partición. En la partición, los elementos menores o iguales al pivote se colocan a la izquierda, mientras que los mayores se ubican a la derecha, ubicando finalmente el pivot en su posición final.

Finalmente, el algoritmo se aplica de forma recursiva a los subarreglos izquierdo y derecho utilizando la misma estrategia de selección del pivote. Para evaluar el rendimiento del algoritmo, se midió el tiempo de ejecución al ordenar arreglos de diferentes tamaños (1,000, 10,000 y 100,000 elementos) generados de forma aleatoria. El resultado de esta prueba se puede apreciar en la Figura 1. Los tiempos fueron registrados utilizando la biblioteca time de python para garantizar mediciones precisas.

*2) **Radix Sort para Datos Alfanuméricos**:* La implementación del algoritmo Radix Sort se adaptó para ordenar listas de strings alfanumericos de longitud variable, trabajando carácter por carácter desde el menos significativo hasta el más significativo. Para lograrlo, se utilizo el algoritmo Counting Sort como subrutina para ordenar los strings de forma estable según un carácter específico. Este enfoque garantiza que strings con caracteres idénticos en las posiciones actuales mantengan su orden relativo de entrada, un requisito clave para el correcto funcionamiento de Radix Sort. A continuación, algunos aspectos fundamentales del algoritmo creado:

- Counting Sort para strings: El algoritmo Counting Sort fue modificado para manejar caracteres ASCII. Durante

cada iteración, se cuentan las ocurrencias de cada carácter (incluyendo un valor predeterminado para índices fuera de rango), y luego se ordenan los strings de acuerdo al carácter correspondiente en la posición especificada.
- Radix Sort por carácter: El algoritmo Radix Sort itera sobre los caracteres desde el último (menos significativo) hacia el primero (más significativo) en cada string, aplicando Counting Sort en cada paso. Esto asegura que lo string queden completamente ordenados al final de las iteraciones.
- Generación de datos de prueba: Para probar la implementación, se generó una lista de 10,000 strings aleatorios, cada uno compuesto por letras y dígitos, con longitudes variables entre 1 y 20 caracteres.
- Medición del rendimiento: Se midió el tiempo de ejecución total del algoritmo al ordenar la lista completa, y se verificó si el resultado estaba correctamente ordenado mediante comparaciones consecsutivas entre los elementos.

3) ***CombSort****:* El objetivo de este inciso fue implementar y analizar el rendimiento de CombSort en comparación con el tradicional Bubble Sort. CombSort se diseñó como una mejora del Bubble Sort, introduciendo un factor de reducción del ”gap” entre elementos comparados en cada iteración, lo que acelera el proceso de ordenamiento al reducir el número de intercambios necesarios. A continuación, algunos aspectos fundamentales del algoritmo creado:

- Implementación de CombSort: CombSort utiliza un ”gap” inicial equivalente al tamaño del arreglo y lo reduce en cada iteración por un factor constante (1.3 en este caso). Durante cada iteración, se comparan elementos separados por esta distancia. Si se encuentran desordenados, se intercambian, y se establece una bandera para verificar si el arreglo ya está ordenado. Este enfoque disminuye el tiempo necesario para mover elementos mayores hacia sus posiciones finales, una de las principales limitaciones del Bubble Sort.
- Implementación de Bubble Sort: Bubble Sort sigue un enfoque más básico, comparando y moviendo elementos adyacentes en múltiples pasadas a través del arreglo hasta que no se realicen más intercambios. Aunque este algoritmo es relativamete fácil de entender e implementar, es notoriamente ineficiente para listas grandes, con una complejidad temporal promedio y en el peor caso de $O(n^2)$.
- Medición de tiempo de ejecución: Se midió el tiempo de ejecución de ambos algoritmos en arreglos de diferentes tamaños (100, 500, 1,000, 5,000 y 10,000 elementos) generados aleatoriamente. Cada algoritmo se probó con copias idénticas de los datos para garantizar una comparación justa. En la Figura 2 se puede apreciar los resultados de esta comparación.
- Validación de resultados: Los arreglos ordenados por ambos algoritmos se compararon con los resultados de la función incorporada sorted de Python para verificar que el ordenamiento fuese correcto.

B. Pregunta 2: Funciones de Hash

1) ***Tabla Hash con Resolución de Colisiones por Doble Hashing****:* Se implementaron y evaluaron dos métodos de resolución de colisiones para tablas hash: Doble Hashing y Encadenamiento. Ambos enfoques buscan garantizar la eficiencia en operaciones de inserción, búsqueda y eliminación, pero utilizan estrategias diferentes para manejar las colisiones. A continuación, algunos aspectos fundamentales del algoritmo creado:

- Tabla Hash con Doble Hashing: Este método utiliza dos funciones hash: una primaria para calcular el índice inicial y una secundaria para definir un paso en caso de colisión. El doble hashing reduce el riesgo de agrupamiento de claves en un segmento específico de la tabla al dispersarlas de manera más uniforme. Se inicializa la tabla con un tamaño fijo y se utilizan marcadores especiales para diferenciar elementos eliminados. Durante una colisión, se aplica el desplazamiento definido por la función secundaria hasta encontrar un espacio disponible.
- Tabla Hash con Encadenamiento: Este enfoque asocia una lista enlazada a cada índice de la tabla. En caso de colisión, los pares (clave, valor) se almacenan en la lista correspondiente al índice. Esto permite manejar eficientemente múltiples elementos con el mismo hash primario. Cada índice contiene una lista de pares (clave, valor), que se recorre linealmente durante las operaciones de búsqueda, inserción y eliminación. Las colisiones se manejan directamente al agregar nuevos elementos a la lista enlazada correspondiente.
- Generación de datos de prueba: Se generaron 1,000 claves aleatorias, cada una compuesta de caracteres alfanuméricos, y se definió un conjunto de operaciones optimizadas:
  - Inserciones: Agregar claves con valores aleatorios entre 0 y 1,000.
  - Búsquedas: Recuperar los valores asociados a cada clave.
  - Eliminaciones: Eliminar las claves de la tabla.
- Evaluación del rendimiento: Se midió el tiempo total para completar todas las operaciones (inserción, búsqueda y eliminación) en tablas hash de tamaño 3,000 para ambos métodos. La evaluación se realizó con la función time para garantizar mediciones precisas.

2) ***Función de Hash para Comparación de Textos****:* Para detectar líneas duplicadas en un archivo de texto, se implementaron dos enfoques basados en funciones hash: Rolling Hash y Hash Simple. Ambos métodos buscan identificar duplicados de manera eficiente, pero difieren en su diseño y características.

El Rolling Hash utiliza una base (256) y un número primo (101) para calcular un hash único para cada línea. Este método opera acumulativamente sobre los caracteres de la línea, multiplicando el hash actual por la base y sumando el valor ASCII del carácter correspondiente. Finalmente, el valor se reduce mediante el módulo del número primo para minimizar las colisiones. Este enfoque es especialmente útil en contextos donde se requiere precisión, ya que el uso del

número primo limita significativamente la probabilidad de colisiones entre diferentes cadenas.

Por otro lado, el Hash Simple se basa en la función nativa hash() de Python. Esta función, aunque rápida y fácil de implementar, no ofrece un control explícito sobre las colisiones, lo que puede afectar la precisión cuando se trabaja con grandes volúmenes de datos. En este enfoque, el hash de cada línea se calcula directamente y se compara con un conjunto de hashes previamente almacenados. Si el hash ya existe en el conjunto, la línea se considera duplicada.

Para probar ambos métodos, se generó un archivo con 100,000 líneas de texto. La mitad de estas líneas eran únicas, generadas aleatoriamente con una longitud de 50 caracteres alfanuméricos, mientras que la otra mitad consistía en duplicados tomados de las líneas únicas. Este diseño permitió evaluar la capacidad de detección de duplicados en un entorno controlado. Finalmente, se midió el tiempo de ejecución de cada método y se comparó el número de duplicados detectados para evaluar su rendimiento.

3) ***Hash de Conteo de Elementos con Alta Frecuencia****:* El propósito de esta implementación fue comparar dos métodos para contar la frecuencia de elementos en una lista grande: una tabla hash personalizada y la clase Counter de la biblioteca estándar de Python. Para esto, se generó una lista con 100,000 elementos, donde cada valor se seleccionó aleatoriamente de un conjunto de 1,000 valores únicos. Esto permitió evaluar ambos métodos en un contexto de alto volumen con repeticiones significativas, simulando un caso práctico típico en análisis de data.

El método basado en una tabla hash personalizada utiliza un diccionario para almacenar cada elemento como clave y su frecuencia como valor. A medida que se recorre la lista, se verifica si el elemento ya está presente en el diccionario. Si está presente, su contador se incrementa; en caso contrario, se inicializa con una frecuencia de uno. Este enfoque es intuitivo y flexible, pero depende de la eficiencia del manejo interno de los diccionarios de Python.

Por otro lado, el método que utiliza la clase Counter emplea una implementación optimizada basada en tablas hash internas. Este enfoque integra el proceso de conteo en un único paso, y además proporciona métodos adicionales como most common, que permite extraer directamente los elementos más frecuentes. La simplicidad y eficiencia de Counter lo hacen adecuado para tareas de conteo estándar.

Para evaluar el rendimiento de ambos métodos, se midió el tiempo de ejecución usando la función time. Una vez que los resultados de las frecuencias fueron obtenidos, se identificaron los 10 elementos más frecuentes en cada caso. Para la tabla hash personalizada, esto se logró ordenando los pares (elemento, frecuencia) por la frecuencia de forma descendente. En el caso de Counter, se utilizó el método most common directamente.

III. Análisis y Resultados

A. Resultados de la Pregunta 1: Algoritmos de Ordenamiento

La Figura 1 demuestra que la selección del pivote mediante la mediana de tres es una estrategia efectiva para mejorar la estabilidad del rendimiento del algoritmo QuickSort. En particular, esta técnica reduce las particiones desequilibradas que pueden presentarse en el QuickSort clásico, especialmente en casos donde el arreglo inicial está parcialmente ordenado. Este comportamiento valida la complejidad promedio del algoritmo $O(nlog(n))$, ya que el tiempo de ejecución crece de manera proporcional al tamaño del arreglo. Esta característica confirma la eficiencia del enfoque implementado. Además, la implementación es particularmente útil en escenarios donde los datos presentan cierto grado de orden, ya que minimiza la probabilidad de particiones desfavoranles.

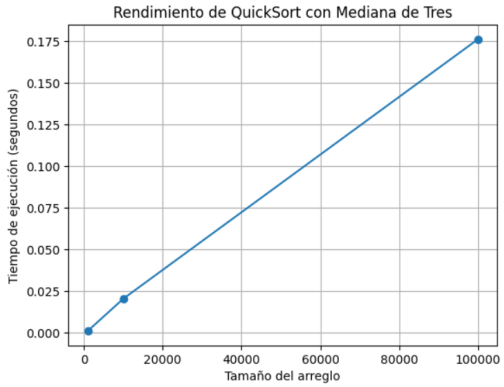

Fig. 1: Rendimiento de *QuickSort* para distintos tamaños de arreglos

Respecto a la implementación del algoritmo *Radix Sort*, el algoritmo completó la ordenación de la lista de 10,000 strings en 0.0455 segundos, demostrando un rendimiento eficiente para datos de tamaño significativo. Además, se verificó que la lista quedó correctamente ordenada de acuerdo al orden lexicográfico, sin errores en el orden. El tiempo de ejecución observado valida la eficiencia del algoritmo, ya que Radix Sort tiene una complejidad temporal de $O(n \cdot k)$, donde n es el número de elementos y k es la longitud del string más largo. Para los datos probados, k fue relativamente pequeño (máximo 20), lo que contribuyó al buen desempeño. Además, la estabilidad de Counting Sort permitió mantener la consistencia en el orden durante cada iteración. Por lo tanto, el algoritmo demostró ser ideal para aplicaciones que requieren ordenar grandes volúmenes de datos alfanuméricos de longitud variable. Sin embargo, su eficiencia depende de que k sea relativamente pequeño, ya que un incremento significativo en la longitud máxima de los strings podría impactar el tiempo de ejecución.

Respecto a la comparación entre CombSort y Bubble Sort, la Figura 2 revela las ventajas de optimizar el ordenamiento básico mediante la reducción del ”gap”. CombSort se beneficia de una mejor distribución inicial de los elementos y realiza menos intercambios innecesarios, lo que reduce significativamente el tiempo de ejecución para arreglos grandes. Además, su complejidad temporal promedio de $O(nlog(n)$ lo hace más adecuado para aplicaciones prácticas donde se requiere un rendimiento aceptable sin recurrir a algoritmos más complejos como QuickSort o MergeSort. Por otro lado, Bubble Sort demostró ser poco eficiente, con un rendimiento que empeora

rápidamente a medida que aumenta el tamaño del arreglo. Este algoritmo es útil únicamente para listas pequeñas o cuando la simplicidad de implementación es una prioridad.

En resumen, CombSort es una mejora directa y eficiente sobre Bubble Sort, adecuada para casos donde se busca simplicidad con un mejor rendimiento. Sin embargo, para listas extremadamente grandes o aplicaciones críticas, se recomienda considerar alternativas más avanzadas.

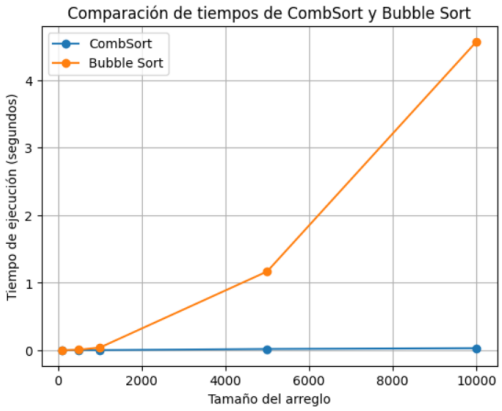

Fig. 2: Comparación de rendimiento entre *Combsort* y *Bubble Sort*

B. Resultados de la Pregunta 2: Funciones de Hash

1) ***Tabla Hash con Resolución de Colisiones por Doble Hashing****:* Los tiempos de ejecución para ambos métodos fueron similares, con Doble Hashing completando las operaciones en 0.002995 segundos y Encadenamiento en 0.003088 segundos. Estas diferencias marginales reflejan la eficiencia comparable de ambos enfoques en términos de tiempo promedio de operación. Sin embargo, cada método presenta ventajas y desventajas dependiendo del contexto.

El Doble Hashing mostró un buen rendimiento gracias a su capacidad para dispersar uniformemente los elementos en la tabla. Esto minimizó la cantidad de colisiones y redujo el tiempo necesario para buscar un espacio disponible o para localizar un elemento existente. Sin embargo, este método puede ser menos eficiente cuando la tabla se encuentra cerca de su capacidad máxima, ya que los desplazamientos adicionales aumentan en frecuencia.

En contraste, el Encadenamiento es más flexible, ya que las colisiones se manejan directamente en listas enlazadas sin necesidad de desplazamientos. Esto hace que el método sea menos sensible al tamaño de la tabla. Sin embargo, en casos con muchas colisiones, las listas asociadas pueden volverse largas, degradando el rendimiento a medida que aumenta el tiempo necesario para recorrerlas.

2) ***Función de Hash para Comparación de Textos****:* El archivo de prueba contenía 100,000 líneas, de las cuales la mitad eran únicas y la otra mitad eran duplicados generados aleatoriamente. El método Rolling Hash calculó hashes utilizando una base (256) y un módulo primo (101) para minimizar colisiones, mientras que el Hash Simple empleó la función hash() de Python para calcular los valores hash de cada línea.

Los resultados mostraron diferencias significativas tanto en el tiempo de ejecución como en la cantidad de duplicados detectados. El Rolling Hash detectó 49,950 duplicados en 0.308 segundos, mientras que el Hash Simple identificó solo 25,000 duplicados en un tiempo mucho menor de 0.030 segundos. Estos resultados evidencian que el Rolling Hash fue más preciso al minimizar las colisiones, lo que permitió detectar correctamente casi todos los duplicados del archivo. Sin embargo, esta precisión tuvo un costo computacional mayor debido al cálculo incremental y moludar del hash para cada carácter.

El Hash Simple, aunque más rápido, fue menos preciso debido a una mayor probabilidad de colisiones al no emplear un módulo primo para reducirlas. Esto resultó en la detección incorrecta de duplicados, ya que varias líneas diferentes compartieron el mismo valor hash. A pesar de esto, su velocidad lo hace adecuado para aplicaciones donde el tiempo de procesamiento es crítico y la precisión no es prioritaria.

En conclusión, el Rolling Hash es más adecuado para aplicaciones que requieren alta precisión en la detección de duplicados, como sistemas de deduplicación de datos críticos. Por otro lado, el Hash Simple es ideal para escenarios donde la velocidad es más importante que la precisión, como en prototipos o análisis exploratorios. La elección del método dependerá de los requisitos específicos de cada caso.

3) ***Hash de Conteo de Elementos con Alta Frecuencia****:* Ambos métodos fueron precisos al identificar los elementos más frecuentes en la lista, mostrando resultados idénticos en cuanto a los 10 valores principales y sus respectivas frecuencias. Sin embargo, hubo una diferencia significativa en los tiempos de ejecución. La tabla hash personalizada completó el conteo en 0.008926 segundos, mientras que el método basado en Counter fue considerablemente más rápido, con un tiempo de 0.003601 segundos, lo que representa una mejora de aproximadamente 2.5 veces en velocidad.

El rendimiento superior de Counter se debe a las optimizaciones internas de esta clase, que está diseñada específicamente para realizar operaciones de conteo de manera eficiente. Además, proporciona funcionalidades adicionales como el método most common, que simplifica la obtención de los elementos más frecuentes sin necesidad de realizar ordenamientos manuales. Por otro lado, la tabla hash personalizada, aunque funcional, requiere iteraciones explícitas y cálculos manuales para lograr el mismo resultado, lo que incrementa ligeramente el tiempo de ejecución.

Ambos métodos tienen una complejidad promedio de $O(n)$ con nn es el tamaño de la lista. Sin embargo, en la práctica, las implementaciones de Counter optimizan las operaciones subyacentes mediante el uso de estructuras de hash internas altamente eficientes, mientras que la tabla hash personalizada depende de la implementación estándar del diccionario en Python, que, aunque robusta, no está específicamente optimizada para tareas de conteo.

En conclusión, mientras que la tabla hash personalizada puede ser útil en escenarios donde se necesite personalizar la lógica de conteo o manipular estructuras de datos no estándar, el método basado en Counter es claramente la opción preferida para aplicaciones generales. Su combinación de velocidad, pre-

cisión y facilidad de uso lo convierte en una herramienta ideal para manejar tareas de conteo de frecuencias en listas grandes. Por lo tanto, a menos que existan requisitos específicos que justifiquen el uso de una tabla hash personalizada, Counter es la solución más práctica y eficiente.

IV. Conclusiones Generales

El informe presentó el análisis y comparación de diversas técnicas de ordenamiento y funciones de hash, evaluando su rendimiento y aplicabilidad en distintos escenarios. Los algoritmos de ordenamiento, como QuickSort con Mediana de Tres, Radix Sort y CombSort, destacaron por su eficiencia en manejar conjuntos de datos de gran tamaño, cada uno mostrando fortalezas específicas dependiendo de las características de los datos procesados. QuickSort con Mediana de Tres mostró estabilidad en su rendimiento al reducir los casos extremos; Radix Sort demostró ser especialmente eficiente al trabajar con cadenas de caracteres, y CombSort ofreció un balance interesante entre simplicidad y velocidad al optimizar el desempeño de Bubble Sort.

En el caso de las funciones de hash, las implementaciones evaluadas se enfocaron en tareas como la detección de duplicados, la resolución de colisiones y el conteo de elementos frecuentes. Rolling Hash resaltó por su precisión al minimizar colisiones mediante el uso de un módulo primo, mientras que Counter de Python demostró una velocidad sobresaliente gracias a sus optimizaciones internas, haciéndolo ideal para tareas estándar de conteo de elementos. Por otro lado, las soluciones personalizadas, como las tablas hash con doble hashing y encadenamiento, exhibieron un desempeño robusto en la resolución de colisiones, subrayando la flexibilidad de estas implementaciones para adaptarse a necesidades específicas.

En términos generales, los resultados muestran que la elección del método depende de las características y requerimientos del problema. Métodos como Counter o Radix Sort son altamente recomendables para escenarios donde el rendimiento es prioritario y los datos presentan estructuras conocidas, mientras que soluciones personalizadas, como tablas hash y algoritmos basados en Rolling Hash, son más adecuadas para aplicaciones que demandan mayor control o especificidad en su implementación.

Este análisis pone en evidencia no solo las capacidades de las técnicas evaluadas, sino también la importancia de comprender las limitaciones inherentes a cada enfoque. El uso adecuado de estas herramientas puede marcar una diferencia significativa en el rendimiento y la efectividad de los sistemas, ofreciendo una base sólida para abordar problemas computacionales de diversa índole y escalabilidad. Assimismo, los resultados abren la puerta a futuras optimizaciones y estudios más profundos sobre cómo estas técnicas pueden combinarse o adaptarse para maximizar su impacto en aplicaciones prácticas.