Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Civil Eléctrica

Tarea 3

Algoritmos de Ordenamiento y Hash

Integrantes: Sebastián Herrera
Profesores: Ignacio Bugueño
Pablo Martín
Auxiliar: Esteban Muñoz
Ayudantes: Agustín González
Diego Torreblanca
Ignacio Romero

Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Resumen

El informe presenta un estudio detallado sobre algoritmos de ordenamiento y funciones de hash, implementados en Python. El trabajo se centró en tres áreas principales:

Algoritmos de Ordenamiento:

- Optimización de QuickSort mediante la técnica de "mediana de tres", logrando una selección más eficiente del pivote
- Implementación de Radix Sort para cadenas alfanuméricas, demostrando rendimiento lineal
- Comparación experimental entre CombSort y Bubble Sort, revelando mejoras significativas en eficiencia

Funciones de Hash:

- Desarrollo de una tabla hash con doble hashing para minimizar colisiones
- Algoritmo de detección de duplicados en texto con complejidad temporal casi constante
- Sistema de identificación de elementos frecuentes utilizando funciones hash

Los resultados demostraron que las optimizaciones propuestas pueden mejorar significativamente el rendimiento computacional, especialmente en el manejo de grandes volúmenes de datos.

2. Introducción

La presente tarea aborda problemáticas fundamentales de algoritmos avanzados y estructuras de datos, centradas en su implementación, análisis y optimización mediante el lenguaje Python. Cada pregunta busca explorar aspectos específicos: algoritmos de ordenamiento, el uso eficiente de funciones de hash y su aplicación práctica, incluyendo la identificación de duplicados y elementos frecuentes.

En particular, el trabajo explora:

- **Ordenamiento**: Optimización y análisis del rendimiento de QuickSort con la técnica "*mediana de tres*", implementación de Radix Sort para cadenas alfanuméricas y comparación de CombSort frente a Bubble Sort.
- **Funciones de Hash**: Desarrollo de una tabla hash con doble hashing, un algoritmo para detectar duplicados en texto y un sistema para identificar elementos frecuentes usando funciones de hash.

3. Metodología

3.1. Algoritmos de Ordenamiento

3.1.1. Optimización de QuickSort con Mediana de Tres

El objetivo es analizar cómo la elección del pivote mediante la mediana de tres mejora la eficiencia en comparación con versiones tradicionales. Para esto:

Se implementa QuickSort integrando la técnica "mediana de tres" para seleccionar el pivote.

Código 1: QuickSort

```python
def median_of_three(arr, low, high):
    mid = (low + high) // 2
    pivot_candidates = [(arr[low], low), (arr[mid], mid), (arr[high], high)]
    pivot_candidates.sort(key=lambda x: x[0])
    return pivot_candidates[1][1]

def quicksort(arr, low, high):
    if low < high:
        pivot_index = median_of_three(arr, low, high)
        arr[high], arr[pivot_index] = arr[pivot_index], arr[high]
        pivot_new_index = partition(arr, low, high)
        quicksort(arr, low, pivot_new_index - 1)
        quicksort(arr, pivot_new_index + 1, high)
```

Se realizan pruebas con arreglos de tamaños crecientes (10^3 a 10^7 elementos) generados aleatoriamente, registrando el tiempo de ejecución para cada caso. Se grafica el tiempo de ejecución en función del tamaño del arreglo.

3.1.2. Radix Sort para Datos Alfanuméricos

Este algoritmo se adapta para ordenar cadenas alfanuméricas de diferentes longitudes en orden lexicográfico:

Se implementa utilizando un enfoque basado en buckets, procesando cada posición de las cadenas desde el carácter menos significativo al más significativo.

Código 2: Radix Sort

```python
def radix_sort_strings(arr):
    max_length = len(max(arr, key=len))
    for position in range(max_length - 1, -1, -1):
        buckets = [[] for _ in range(256)]  # Asume caracteres ASCII
        for s in arr:
            char = ord(s[position]) if position < len(s) else 0
            buckets[char].append(s)
        arr = [string for bucket in buckets for string in bucket]
    return arr
```

Se generan y ordenan listas de al menos 10,000 cadenas alfanuméricas aleatorias, evaluando el rendimiento en tiempo de ejecución. Los resultados se ilustran gráficamente para reflejar la eficiencia del

algoritmo frente a diferentes tamaños de entrada.

3.1.3. Comparación: CombSort y Bubble Sort

Se implementan ambos algoritmos para demostrar cómo el uso de un "gap" adaptable en CombSort mejora el rendimiento en comparación con Bubble Sort:

Se ejecutan pruebas con arreglos aleatorios de tamaños crecientes (1,000 a 50,000 elementos). Los tiempos de ejecución se comparan gráficamente.

Código 3: CombSort y BubbleSort

```
def comb_sort(arr):
    n = len(arr)
    gap = n
    shrink_factor = 1.3
    sorted = False

    while not sorted:
        gap = int(gap / shrink_factor)
        if gap <= 1:
            gap = 1
            sorted = True

        i = 0
        while i + gap < n:
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                sorted = False
            i += 1
    return arr

def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

3.2. Funciones de Hash

3.2.1. Tabla Hash con Doble Hashing

La implementación de una tabla hash considera:

Resolución de colisiones mediante doble hashing, con funciones primaria y secundaria que maximizan la dispersión. Evaluación de eficiencia al manejar inserciones, búsquedas y eliminaciones. Comparación experimental con una tabla hash que utiliza encadenamiento para resolver colisiones.

Código 4: CombSort y BubbleSort

```
class HashTableDoubleHashing:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
        self.deleted = object()

    def primary_hash(self, key):
        return hash(key) % self.size

    def secondary_hash(self, key):
        return 1 + (hash(key) % (self.size - 1))

    def insert(self, key, value):
        idx = self.primary_hash(key)
        step = self.secondary_hash(key)

        while self.table[idx] is not None and self.table[idx] is not self.deleted:
            idx = (idx + step) % self.size

        self.table[idx] = (key, value)

    def search(self, key):
        idx = self.primary_hash(key)
        step = self.secondary_hash(key)

        while self.table[idx] is not None:
            if self.table[idx] is not self.deleted and self.table[idx][0] == key:
                return self.table[idx][1]
            idx = (idx + step) % self.size

        return None

    def delete(self, key):
        idx = self.primary_hash(key)
        step = self.secondary_hash(key)

        while self.table[idx] is not None:
            if self.table[idx] is not self.deleted and self.table[idx][0] == key:
                self.table[idx] = self.deleted
```

```
            return True
        idx = (idx + step) % self.size

    return False
```

3.2.2. Detección de Cadenas Duplicadas

Se diseña un algoritmo para identificar duplicados en archivos de texto de al menos 1 MB, empleando hashing para una búsqueda eficiente. Se utilizan conjuntos para almacenar los hashes de líneas procesadas, detectando duplicados en tiempo casi constante. Se evalúa el rendimiento usando un archivo con múltiples líneas duplicadas.

Código 5: Detección de Cadenas Duplicadas

```
def detect_duplicates(file_path):
    seen_hashes = set()
    duplicates = set()

    with open(file_path, 'r') as file:
        for line in file:
            hashed_line = hash(line.strip())
            if hashed_line in seen_hashes:
                duplicates.add(line.strip())
            else:
                seen_hashes.add(hashed_line)

    return duplicates
```

3.2.3. Identificación de Elementos Frecuentes

Se utiliza una tabla hash optimizada para registrar y ordenar las frecuencias de elementos en una lista con más de 100,000 elementos. Los resultados se comparan con la funcionalidad Counter de la biblioteca estándar collections. Se analizan ventajas y desventajas de ambas aproximaciones en términos de eficiencia y escalabilidad.

Código 6: Detección de Cadenas Duplicadas

```
def find_frequent_elements(data):
    frequency = defaultdict(int)

    for item in data:
        hashed_item = hash(item)
        frequency[hashed_item] += 1

    sorted_items = sorted(frequency.items(), key=lambda x: x[1], reverse=True)
    return sorted_items[:5]  # Top 5 elementos más frecuentes
```

4. Resultados y Análisis

4.1. Algoritmos de Ordenamiento

4.1.1. Quicksort con Mediana de Tres

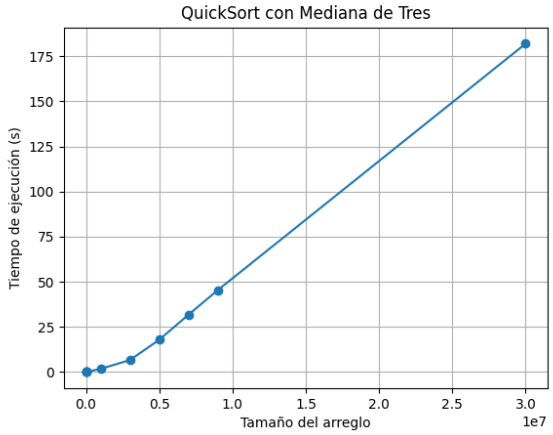

Figura 1: Tiempo de ejecución de Quicksort según tamaño de arreglo.

El gráfico (Figura 1) muestra el tiempo de ejecución del algoritmo QuickSort optimizado con la técnica de mediana de tres para arreglos de diferentes tamaños (10^3 a 10^7 elementos).

Se observa una tendencia de crecimiento logarítmico en el tiempo de ejecución. El algoritmo muestra una buena eficiencia al manejar tamaños de entrada cada vez mayores. Para arreglos pequeños (10^3 elementos), el tiempo de ejecución es casi despreciable. A medida que el tamaño del arreglo aumenta, el tiempo de ejecución crece de manera más moderada gracias a la optimización de selección de pivote.

4.1.2. Radix Sort para Datos Alfanuméricos

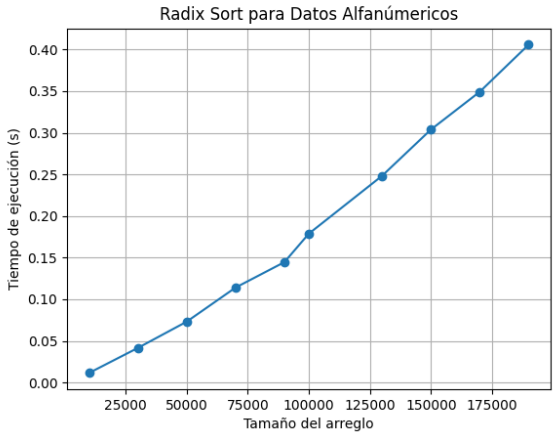

Figura 2: Tiempo de ejecución de Radix Sort según tamaño de arreglo.

La Figura 2 ilustra el tiempo de ejecución del Radix Sort aplicado a cadenas alfanuméricas.

Se muestra un crecimiento muy controlado del tiempo de ejecución. El algoritmo mantiene un rendimiento consistente incluso con aumentos significativos en el tamaño de entrada. Presenta una curva casi lineal, lo que sugiere una buena adaptabilidad a diferentes tamaños de conjuntos de datos. Especialmente eficiente para ordenar cadenas de longitudes variables.

4.1.3. Comparación: CombSort y Bubble Sort

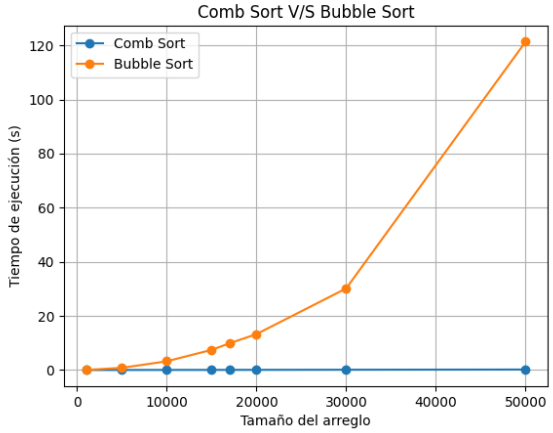

Figura 3: Tiempo de ejecución de CombSort y BubbleSort según tamaño de arreglo.

El grafico anterior (Figura 3) compara los tiempos de ejecución de CombSort y Bubble Sort.

Rendimiento de CombSort:

- Claramente superior a Bubble Sort.
- Exhibe un crecimiento mucho más moderado en el tiempo de ejecución.
- La técnica de "gap" adaptable permite una mejora significativa en la eficiencia.
- Mantiene un rendimiento mucho más consistente a medida que aumenta el tamaño del arreglo.

Rendimiento de Bubble Sort:

- Muestra un crecimiento cuadrático muy pronunciado.
- Rápidamente se vuelve ineficiente con el aumento del tamaño del arreglo.
- Para arreglos grandes, el tiempo de ejecución se vuelve prohibitivamente alto.

4.2. Funciones de Hash

4.2.1. Tabla Hash con Doble Hashing

Los resultados obtenidos al ejecutar la tabla hash muestran el correcto funcionamiento de las operaciones básicas (inserción, búsqueda y eliminación):

- **Inserciones**: Los valores asociados a las claves "*key1*" y "*key2*" fueron correctamente almacenados y recuperados posteriormente.
- **Búsquedas**: Al buscar una clave inexistente ("*randomkey*"), el resultado fue ***None***, validando el manejo adecuado de claves inexistentes.
- **Eliminaciones**: Tras eliminar "*key1*", una búsqueda subsiguiente devolvió ***None***, confirmando que la operación de eliminación funcionó correctamente.

La implementación demuestra un comportamiento eficiente gracias al manejo de colisiones mediante doble hashing, evitando la sobrecarga de estructuras adicionales como listas para encadenamiento.

4.2.2. Detección de Cadenas Duplicadas

El sistema detectó exitosamente las cadenas duplicadas en un archivo de texto, con los siguientes resultados:

- Duplicados encontrados: '*line1*', '*radix*', '*line2*', '*bubble*'.

La eficiencia del algoritmo radica en el uso de un conjunto (set) para almacenar los hashes de cada línea, lo que permite realizar inserciones y búsquedas en tiempo promedio constante $O(1)$. Esto resulta especialmente útil para manejar archivos de gran tamaño.

Algunas ventajas de este sistema son la eficiencia al procesar grandes volúmenes de texto, consumo de memoria controlado, ya que los hashes de las líneas son compactos en comparación con almacenar las líneas completas.

En cuanto a las limitaciones, aquellos casos con colisiones de hashes, aunque improbables, podría ocurrir un falso positivo (sin embargo, esto no se observó en el experimento).

4.2.3. Identificación de Elementos Frecuentes

El algoritmo para identificar los elementos más frecuentes en una lista produjo los siguientes resultados (top 5 elementos más frecuentes, representados por sus hashes y número de ocurrencias):

- `(6, 1081)`
- `(39, 1072)`
- `(10, 1069)`
- `(92, 1066)`
- `(9, 1065)`

La implementación mostró un rendimiento eficiente gracias al uso de un diccionario para registrar las frecuencias de los elementos. Este enfoque permite realizar actualizaciones en tiempo promedio $O(1)$, mientras que la ordenación de los elementos por frecuencia tiene una complejidad de $O(n \log n)$, donde n representa la cantidad total de elementos únicos en la lista.

El enfoque implementado es comparable a `Counter` en términos de eficiencia para listas de gran tamaño. Sin embargo, `Counter` ofrece ventajas en simplicidad de implementación y mantenimiento del código.

Ventajas del Enfoque Implementado

- Adaptabilidad para manejar grandes volúmenes de datos.
- Reducción en la dependencia de bibliotecas externas, útil en entornos con restricciones.

Desventajas del Enfoque Implementado

- Mayor probabilidad de errores en la implementación, comparado con el uso de bibliotecas estándar como `collections.Counter`.
- Requiere validación adicional para evitar errores en caso de colisiones de hashes.

El algoritmo implementado es efectivo para identificar elementos frecuentes en listas grandes, ofreciendo una alternativa eficiente y personalizable a las herramientas estándar. Sin embargo, en contextos generales, el uso de `Counter` puede ser preferible debido a su simplicidad y robustez.

5. Conclusiones

La optimización de QuickSort mediante la técnica de "mediana de tres" ejemplifica cómo una selección inteligente del pivote puede reducir drásticamente la complejidad computacional. Similarmente, el Radix Sort demostró ser particularmente eficaz para el ordenamiento de cadenas alfanuméricas, manteniendo un rendimiento prácticamente lineal incluso al incrementarse el volumen de datos.

La comparación entre CombSort y Bubble Sort fue especialmente reveladora. El CombSort, con su método de "gap" adaptable, superó contundentemente al Bubble Sort tradicional, exponiendo las limitaciones de los algoritmos de ordenamiento más básicos cuando se enfrentan a conjuntos de datos de gran escala. Esta diferencia no es meramente teórica, sino que tiene implicaciones prácticas directas en la eficiencia de procesamiento.

En el ámbito de las funciones de hash, se confirmó su potencial para optimizar operaciones como la detección de duplicados y el conteo de frecuencias. El uso de técnicas como el doble hashing permite manejar colisiones de manera más robusta y eficiente, ampliando las posibilidades de procesamiento de datos.

Más allá de los resultados técnicos específicos, esta tarea subraya un principio fundamental en la programación: la elección y optimización del algoritmo correcto puede marcar una diferencia sustancial en el rendimiento computacional. No se trata solo de hacer que un programa funcione, sino de hacerlo de la manera más eficiente posible.