Tarea N°3:

Algoritmos de ordenamiento y Hashing

Tomás Maximiliano Barros Barrios
<Departamento de Ingeniería Eléctrica>, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile
17, Octubre, 2024

I. Introducción

El siguiente trabajo será dividido en dos secciones. La primera sección tiene como propósito probar, modificar y optimizar distintos métodos de ordenamiento vistos en clases. La segunda sección tiene como propósito utilizar funciones de Hash para resolver distintos problemas.

II. Algoritmos de ordenamiento

El objetivo de la siguiente sección es probar distintos métodos de ordenamiento y comparar su desempeño. La parte uno de esta sección optimizará QuickSort utilizando la elección del pivote mediante el método de la mediana de tres. La segunda parte modificará el algoritmo de RadixSort para que ahora sea capaz de trabajar con strings. La tercera y ultima parte consistirá en implementar el algoritmo CombSort, el cual es una mejora de BubbleSort visto en clases, para finalmente comparar el desempeño de ambos.

A. Parte 1: Optimización QuickSort con mediana de 3

En esa sección, se comparará el algoritmo QuickSort regular obtenido en [1] (ya que es un algoritmo altamente estudiado, no se perderá tiempo programandolo desde cero), con el algoritmo QuickSort optimizado usando el método de la mediana de tres (programado aquí).

El algoritmo de ordenamiento Quicksort consiste en seleccionar a un elemento de un arreglo como 'pivote' (típicamente se selecciona al ultimo elemento del arreglo como pivote). Luego divide el arreglo en dos subconjuntos, uno con elementos menores que el pivote y otro con elementos mayores. La idea es aplicar este procedimiento de forma recursiva cada vez que se crean los subconjuntos. Por otro lado, [2] y [3] dicen que el método de la mediana de tres se utiliza para seleccionar un elemento pivote para el algoritmo Quicksort. Mediana de 3 considera tres elementos: el primero, el de en medio y el último elemento de la lista. Al elegir la mediana de estos tres valores, el algoritmo puede minimizar las posibilidades de encontrarse con los peores escenarios posibles, que ocurren cuando el pivote se elige de manera incorrecta. El algoritmo de mediana de tres consiste el ordenar el primer, el de en medio y el último elemento de la lista de menor a mayor, para luego intercambiar el elemento de en medio por el ultimo. Luego, se seleccionara como pivote el ultimo elemento. En resumen, a diferencia del algoritmo típico donde se selecciona directamente el ultimo elemento como pivote, en este nuevo algoritmo primero se ordenaran los 3 elementos mencionados previamente, luego se intercambiara el elemento de en medio por el ultimo, y finalmente se seleccionara como pivote el ultimo elemento, que corresponderá a la mediana. El código para el quicksort con mediana de tres es el siguiente:

```
1  def Md3(L,low,high):
2      mid = (low+high) // 2 # Se obtiene punto medio
3      # Ordenar los tres valores de menor a mayor
4      if L[low] > L[mid]:
5          L[low], L[mid] = L[mid], L[low]
6      if L[low] > L[high]:
7          L[low], L[high] = L[high], L[low]
8      if L[mid] > L[high]:
9          L[mid], L[high] = L[high], L[mid]
10
11     # Colocar la mediana como pivote
12     L[mid], L[high] = L[high], L[mid]
13     return L[high]
14
15 def partition2(L, low, high): #Misma funcion de
       particion que antes solo cambiando el pivote
16     pivot = Md3(L,low,high)
17     i = low - 1
18     for j in range(low, high):
19         if L[j] <= pivot:
20             i = i + 1
21             (L[i], L[j]) = (L[j], L[i])
22     (L[i + 1], L[high]) = (L[high], L[i + 1])
23     return i + 1
24
25 def quicksort(L, low, high):
26     if low < high:
27         pivote = partition2(L, low, high)
28         quicksort(L, low, pivote - 1)
29         quicksort(L, pivote + 1, high)
30
31
32 def quicksort_con_Md3(L):
33     quicksort(L, 0, len(L)-1)
```

Código 1: Código de QuickSort con Mediana de 3

Lo relevante del código 1 viene en la función 'Md3(L,low,high)' definida desde la linea 1 hasta la 13. Lo que hace esta función es el procedimiento mencionado anteriormente para ordenar el primer elemento, el elemento de en medio y el ultimo elemento del arreglo de menor a mayor, para luego intercambiar estos últimos de posición. De esta forma, obtenemos el pivote nuevo que será utilizado para quicksort. El resto de funciones son las mismas que se utiliza en un quicksort regular, siendo la unica diferencia la elección del pivote. A continuación se presentan los códigos y los resultados obtenidos al comparar el desempeño de quicksort normal con quicksort utilizando mediana de 3:

```
1  def tiempoFuncion(function):
2      def wrapper(*args, **kwargs):
3          import time
```

```
        inicio = time.time() # Tiempo inicial de la funcion
        resultado = function(*args, **kwargs) # Se ejecuta la funcion
        total = time.time() - inicio # Tiempo al finalizar la funcion
        return resultado,total
    return wrapper
```

Código 2: Código para medir tiempo de la función (reutilizada de la tarea 1)

```
def arreglo(a, b, n):
    if a > b:
        print("'a' debe ser menor o igual a 'b'.")
    if n <= 0:
        print("El tama o del arreglo debe ser mayor que cero.")
    return [random.randint(a, b) for _ in range(n)]
```

Código 3:]Código para obtener arreglos de tamaño 'n' con números enteros dentro del intervalo [a,b]

```
#Funciones con decorador de tiempo
@tiempoFuncion
def quicksort_normal_t(L):
    quicksort_normal(L)

@tiempoFuncion
def quicksort_mediana_t(L):
    quicksort_con_Md3(L)

#Funcion para comparar rendimiento y graficar
def comparar_rendimiento():
    # Tama os de los arreglos
    tamanos = [10**3, 10**4, 10**5]
    tiempos_normal = []
    tiempos_md3 = []

    for n in tamanos:
        # Generar un arreglo aleatorio y copiarlo
        arr = arreglo(-10000000, 10000000, n)
        arr_copia_normal = arr
        arr_copia_md3 = arr
        # Medir QuickSort normal
        _, tiempo_normal = quicksort_normal_t(arr_copia_normal)
        tiempos_normal.append(tiempo_normal)
        # Medir QuickSort con Mediana de Tres
        _, tiempo_md3 = quicksort_mediana_t(arr_copia_md3)
        tiempos_md3.append(tiempo_md3)

    # Graficar
    plt.plot(tamanos, tiempos_normal, marker='o', label='QuickSort Normal')
    plt.plot(tamanos, tiempos_md3, marker='s', label='QuickSort Mediana de Tres')
    plt.xlabel('Tama o del arreglo (n)')
    plt.ylabel('Tiempo de ejecuci n (segundos)')
    plt.title('QuickSort vs QuickSort con Mediana de Tres')
    plt.legend()
    plt.grid(True)
    plt.show()

# Llamar a la funci n para comparar
comparar_rendimiento()
```

Código 4: Código que compara y grafica el rendimiento de ambos QuickSort

El código 2 muestra la función para medir el tiempo de ejecución de los QuickSort (esta función fue reutilizada de la tarea 1). El código 3 muestra como se genera un arreglo de tamaño 'n', con números aleatorios enteros entre [a,b]. El código 4 muestra como se utilizó el decorador de medición del tiempo para ambas funciones QuickSort, para luego definir la función 'comparar_rendimiento()'. Esta función toma tamaños de arreglo 'n' predefinidos en la función, donde $n = [10^3, 10^4, 10^5]$. Luego genera arreglos de tiempo para ir guardando cuanto demora cada QuickSort para distintos tamaños de los arreglos. Finalmente ejecuta las funciones QuickSort, guardando los tiempos de ejecución y las gráfica.

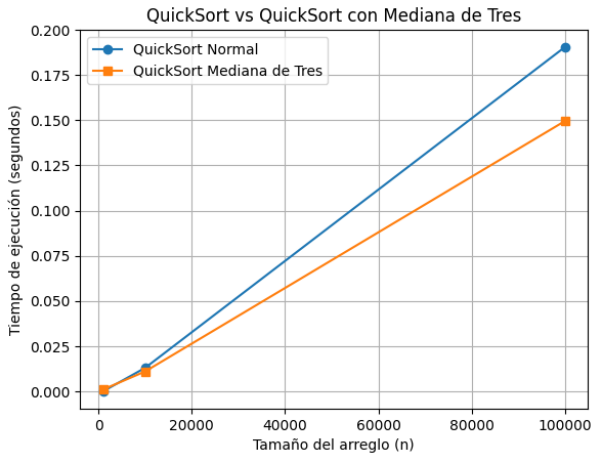

Fig. 1: Resultado de tiempo de ejecución entre QuickSort normal v/s QuickSort con Mediana de 3

La figura 1 muestra como el algoritmo para QuickSort utilizando Mediana de tres tiene menor tiempo de ejecución que el algoritmo regular. Esto era lo esperado, ya que escoger el pivote con Mediana de tres minimiza las probabilidades de encontrar el peor escenario posible en QuickSort normal, haciendo que este algoritmo sea en promedio más eficiente que el QuickSort normal.

B. Parte 2: RadixSort para datos alfanuméricos

La siguiente sección tiene como objetivo ordenar datos alfanuméricos utilizando el algoritmo de RadixSort. Por un lado, los datos alfanuméricos son una combinación entre las letras del alfabeto y números ([4]). Un ejemplo de esto puede ser 'To3ma2si92to', que corresponde a una combinación de letras (mayúsculas o minúsculas) y números.

Por otro lado, el algoritmo de ordenación RadixSort se basa en utilizar CountingSort pero dígito por dígito ([5], [6] y [7]). Es decir, que primero aplica CountingSort para las unidades, luego para las decenas, luego centenas y así. El problema radica en que RadixSort es solo para dígitos, y por tanto el desafío viene en modificarlo para que ahora sea capaz de ordenar lexicográficamente strings. El orden lexicográfico a considerar será el de python, donde se cumple lo siguiente: 'número' ¡ 'letra mayúscula' ¡ 'letra minúscula'. Este orden fue encontrado en [8], donde se ve que los números tienen el menor valor (48 a 57), las mayúsculas tienen valores intermedios (65 a 90), y las minúsculas tienen los valores mas grandes (97 a 122).

Por otro lado, teniendo en cuenta que los datos alfanuméricos

no pueden ser ordenados como se hace típicamente con los números (pues un dato alfanumérico no tiene un valor), la forma de ordenarlos será la siguiente:

- Se ordenará lexicográficamente el primer valor de cada dato siguiendo lo descrito anteriormente ('número' < 'letra mayúscula' < 'letra minúscula').
- En caso de haber caracteres repetidos, se ordenarán a partir del siguiente carácter.
- En caso de haber dos datos iguales, pero uno con más caracteres que otro (por ejemplo, 'bA2a' y 'bA2'), se considerará como valor ASCII = 0 al carácter siguiente a comparar (es decir, el orden será ['bA2', 'bA2a'], pues se está comparando el carácter 'a' con un carácter vacío).

Luego, el código modificado para CountingSort y RadixSort, y el código para generar arreglos de datos alfanuméricos es el siguiente:

```python
def countingSort(L, indx, max_len):
    n = len(L)
    out = [""] * n
    count = [0] * 128

    for i in range(n):
        # Si el string no tiene car cter en la posicion 'indx', usar valor ASCII 0
        if max_len - indx - 1 < len(L[i]):
            char = ord(L[i][max_len - indx - 1])
        else:
            char = 0
        count[char] += 1
    for i in range(1, 128):
        count[i] += count[i - 1]

    for i in range(n - 1, -1, -1):
        if max_len - indx - 1 < len(L[i]):
            char = ord(L[i][max_len - indx - 1])
        else:
            char = 0
        out[count[char] - 1] = L[i]
        count[char] -= 1
    for i in range(n):
        L[i] = out[i]
```

Código 5: Código CountingSort

```python
def radixSort(L):
    max_len = max(len(s) for s in L)
    for indx in range(max_len):
        countingSort(L, indx, max_len)
    return L
```

Código 6: Código RadixSort

```python
#Definimos las cadenas de string a juntar
num=["0","1","2","3","4","5","6","7","8","9"]
alf = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z",
       "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"]
T = num + alf

def datos_alfnum(n, min, max):
    # 'n' ser  el largo del array, min y max indican el largo minimo y m ximo de cada dato alfanum rico que compone al array
    L = []
    for i in range(n):
        k = random.randint(min, max)
        palabra = []
        for j in range(k):
            palabra.append(random.choice(T))
        L.append("".join(palabra))
    return L
```

Código 7: Código datos Alfanuméricos

El código 7 muestra como se generarán los arreglos de tamaño 'n' con datos alfanuméricos. El código 5 muestra la implementación del algoritmo CountingSort modificado. La linea 3 genera un arreglo donde se almacenará el resultado ordenado de la iteración. La linea 4 genera un array para contar cuantas veces aparece cada carácter. El 'for' que va desde la linea 6 a la 12 es para recorrer la lista 'L' completa y contar la aparición de cada carácter en la posición 'indx'. Luego, si el string no tiene caracteres en esa posición, entonces se le asigna valor char = 0 (por lo comentado en como se ordenará anteriormente). El 'for' que va desde la linea 13 a la 14 convierte el arreglo de conteo 'count' en un arreglo acumulado (va sumando los 'count' como el algoritmo original). El bucle 'for' de la linea 16 a la 22 recorre la lista 'L' desde su ultimo elemento hasta el primer elemento, luego procede a colocar los elementos de 'L' en su posición final en la salida, utilizando el arreglo acumulado. Finalmente el 'for' de la linea 23 a 24 actualiza la lista 'L' con los resultados ordenados. Por otro lado, el código 6 calcula la longitud máxima que puede llegar a tener un dato alfanumérico en la lista 'L' para determinar el número de iteraciones a realizar (determina la cantidad máxima de índices a recorrer en cada dato). Luego, se itera desde el carácter menos significativo (posición 0) hasta el más significativo (posición max_len -1), retornando finalmente la lista 'L' ordenada.

A continuación se presentan los resultados del tiempo de ejecución del algoritmo para arreglos de entre 10 000 y 20 0000 datos alfanuméricos.

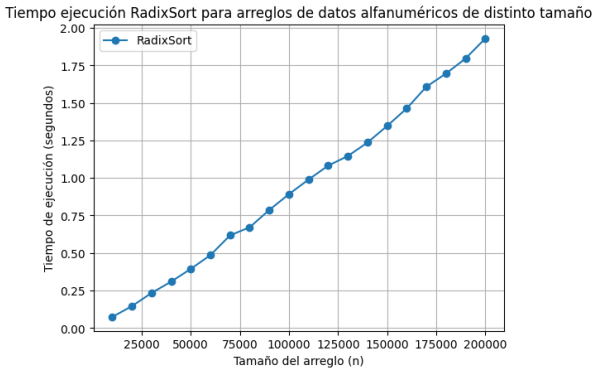

Fig. 2: Resultado de tiempo de ejecución para RadixSort con datos alfanuméricos

La figura 2 muestra el desempeño del algoritmo frente a un gran número de datos. El tamaño de los arreglos varía entre los 10 000 y 200 000, con variaciones de 10 000 en cada ejecución (es decir, primero 10 000, luego 20 000, luego 30 000 y así hasta 200 000). El gráfico muestra un comportamiento lineal y creciente para el tiempo de ejecución, con un tiempo aproximado de 0.125 segundos para

10 000 datos y cerca de 2 segundos para 200 000 datos. Por un lado sabemos que la complejidad teórica de RadixSort es 0(n*k) (presenta una complejidad lineal), con 'n' el tamaño del arreglo y 'k' el largo máximo de cada dato. En este caso, 'n' va desde los 10 000 a los 200 000 y 'k' vale 20. Por otro lado, RadixSort es eficiente cuando 'k' es pequeño en comparación a 'n' (lo cual se cumple en este caso), y además el comportamiento lineal de la figura muestra que efectivamente se presenta una complejidad de la forma lineal esperada teóricamente.

C. Parte 3: CombSort

El propósito de esta sección es investigar e implementar en python el algoritmo CombSort, el cual corresponde a una mejora del algoritmo BubbleSort visto en clases. Por un lado el algoritmo BubbleSort compara elementos adyacentes de una lista, y los intercambia en caso de que estén desordenados, repitiendo este proceso hasta ordenar la lista completa. Esto significa que la complejidad de este algoritmo es del orden de $0(n^2)$.
Según [9], el problema principal que presenta BubbleSort son las denominadas 'tortugas', que corresponden a elementos pequeños que están ubicados en posiciones alejadas del inicio del arreglo. Esto lleva a que las denominadas 'tortugas' necesiten muchas iteraciones para llegar a su posición ordenada (que será al inicio del array), pues BubbleSort solo intercambia los elementos de uno en uno. La idea de CombSort es permitir que los intercambios entre elementos del array no solo ocurran de manera adyacente, de manera que ahora los elementos 'tortuga' más alejados de su posición ordenada, puedan ir desde el final del array hasta el principio, ahorrando entonces tiempo de ejecución. Para permitir que esto suceda, se introduce al algoritmo un elemento llamado 'gap'. Este 'gap' permitirá que las 'tortugas' recorran una distancia más larga en cada iteración. Por ejemplo, si el 'gap' es 10, entonces ocurrirán intercambios de elementos espaciados por 10 elementos. La idea es tener un 'gap' grande al comienzo del algoritmo, y luego ir decreciendo su valor a medida que pasa la ejecución (hasta llegar a 1), lo que permitirá a los elementos 'tortuga' llegar a su posición más rápidamente. Para calcular el 'gap', se utiliza la formula $gap = \frac{n}{factor}$, donde 'n' corresponde al tamaño del arreglo, y factor = 1.3 generalmente ([10]). Para calcular los siguientes 'gap' (ya que irán reduciendoce hasta llegar a 1), la formula es análoga, pero en vez de 'n' es el 'gap' anterior, es decir: $gap_{nuevo} = \frac{gap_{anterior}}{factor}$. Respecto a la eficiencia de cada algoritmo, CombSort presenta una complejidad de $0(n * log(n))$, mientras que BubbleSort presenta una complejidad de $0(n^2)$ ([10]). Esto significa que CombSort es más eficiente que BubbleSort en general, y esa eficiencia evidencia sobre todo para arreglos de tamaño muy grande. Otra de las ventajas que presenta CombSort con respecto a BubbleSort es que es más estable (es decir que elementos iguales del arreglo van a mantener su orden después del ordenamiento con más probabilidad que con BubbleSort), además CombSort tiende a beneficiarse más de la memoria cache del computador, lo cual puede mejorar el rendimiento ([10]).

```python
#Primero hay que definir el gap
def next_gap(gap):
    gap = (gap*10)//13 # gap = gap / 1.3
    if gap < 1:
        return 1
    return gap

#CombSort
def CombSort(L):
    n = len(L)
    gap = n # gap inicial
    swap = True # True para iniciar while
    while gap != 1 or swap == True:
        gap = next_gap(gap) # Calcula gap de la
    iteracion
        swap = False # Setea swap en False (Si
    despues de la iteracion swap sige siendo falso,
    entonces termina, pues est  ordenado)
        for i in range (0, n-gap): #Se hace el
    intercambio de los elementos del array,
    espaciados por el gap
            if L[i] > L[i+gap]:
                L[i],L[i+gap] = L[i+gap],L[i]
                swap = True # Si se realiza
    intercambio, entonces swap es True y el while
    continua
```

Código 8: Código CombSort

El código 8 muestra la implementación de CombSort en python. Se comienza definiendo la función para calcular el 'gap' en cada iteración, siguiendo la formula $gap_{nuevo} = \frac{gap_{anterior}}{factor}$, y retornando 1 en caso de que el resultado sea menor que 1. Con respecto a la función CombSort, esta consta de un ciclo 'while', el cual iterará siempre que 'swap' sea 'True' (Es decir, ocurrió un cambio en el arreglo, y por tanto es necesario recorrerlo de nuevo), o 'gap' sea distinto de 1 (si 'gap' es 1, entonces ya ocurrió el ultimo recorrido con gap = 1 dentro del while, y por tanto no es necesario que ocurra de nuevo). Luego, dentro del 'while' simplemente se calcula el 'gap' a utilizar en el ciclo, se setea 'swap' como falso (pues al comienzo del ciclo no se ha realizado ningún 'swap'), y se recorre el arreglo. Si hay elementos a ordenar dentro del arreglo, entonces se pone 'swap' como 'True' para continuar con el ciclo 'while' luego de ordenar los elementos.

La figura 3 muestra como efectivamente para arreglos grandes, CombSort es más rápido que BubbleSort, mientras que para arreglos pequeños la diferencia es mínima. Esto comprueba el comportamiento discutido anteriormente sobre las complejidades de ambos, donde se concluyo que CombSort será más eficiente para arreglos de gran tamaño.

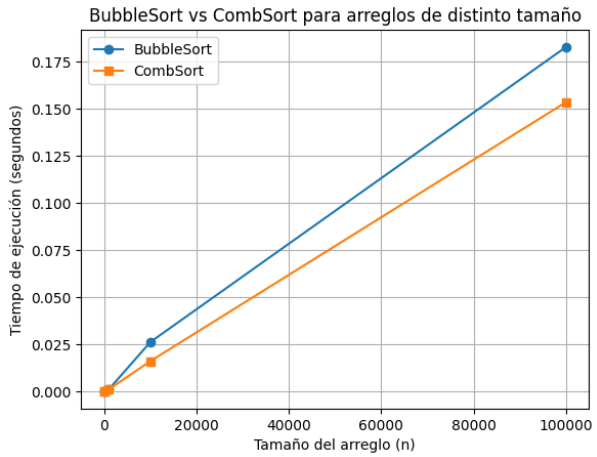

Fig. 3: Resultado de tiempo de ejecución BubbleSort v/s CombSort

III. Funciones de Hash

La siguiente sección tiene como objetivo resolver problemas utilizando funciones de Hashing. La primera parte consiste en comparar los métodos de Doble Hashing y Hashing por encadenamiento, donde el criterio de comparación será el número de colisiones que presentan en su implementación.

A. Parte 1: Implementación de una Tabla Hash con Resolución de Colisiones por Doble Hashing

Hashing es una técnica utilizada en el contexto de las estructuras de datos para almacenar y recuperar datos de manera eficiente. Implica el uso de una función de Hashing que asigna datos a una matriz de tamaño fijo denominada tabla Hash. Uno de los problemas de esta técnica viene por las denominadas 'colisiones'. Las colisiones en el algoritmo Hash ocurren cuando dos claves diferentes se asignan al mismo valor Hash, es decir, dadas dos claves 'x' e 'y', una colisión ocurre cuando h(x) = h(y), con h() la función de Hashing. Para solucionar esto, se propone el método de Hashing Doble. [11] explica que en hashing doble la idea es generar la secuencia de funciones de hashing usando dos funciones de hashing independientes, denominadas h(x) y s(x) (s(x) se denomina 'step'). Entonces, si el casillero h(x) está ocupado, entonces se intenta ocupar el casillero h(x)+s(x). Si ese casillero también esta ocupado, se sigue con h(x)+2*s(x), y así sucesivamente. De manera más precisa, para Hashing doble se utiliza:

$$h_0(x) = h(x) \tag{1}$$

$$h_{i+1}(x) = (h_i(x) + s(x)) \bmod m \tag{2}$$

Para este caso, utilizaremos como primaria la función de Hashing por método de la división, y como secundaria la función de Hashing por método de la multiplicación, ambas vistas en clases. Para el correcto funcionamiento del método por división, es necesario escoger el divisor como un número primo 'm' lo más cercano al tamaño de la tabla (idealmente que el tamaño de la tabla sea un número primo), mientras que para el método de la multiplicación, el valor que debe tener 0<A<1 será 0,618033 (obtenido en [12], donde se menciona que determinados valores de A funcionan mejor que otros). Dicho esto, el código 9 muestra como fue implementado Doble Hashing en python.

```python
class DobleHashing:
    def __init__(self,size):
        self.size = int(size)
        self.tabla = [None]*int(size) # Tabla vacia
        self.colisiones = 0 # Contador de colisiones

    def hash1(self,llave): # Metodo de la divison
        return llave % self.size
    def hash2(self,llave): # Metodo de la
    multiplicacion
        A = 0.618033
        b = (llave * A) % 1  # k*A mod 1
        return int(b * self.size)

    #Funciones
    def insert(self,llave):
        h = self.hash1(llave)
        s = self.hash2(llave)

        for i in range(self.size):
            hh = (h + i*s) % self.size
            if self.tabla[hh] is None:
                self.tabla[hh] = llave
                return

        self.colisiones += 1 # No se pudo insertar
        print("No se pudo insertar la llave: ",
    llave)
        return

    def search(self,llave):
        h = self.hash1(llave)
        s = self.hash2(llave)

        for i in range(self.size):
            hh = (h + i*s) % self.size
            if self.tabla[hh] == llave:
                return True #Si se encuentra retorna
     True
        return False #Si no se encuentra retorna
    False

    def delete(self,llave):
        h = self.hash1(llave)
        s = self.hash2(llave)

        for i in range(self.size):
            hh = (h + i*s) % self.size
            if self.tabla[hh] == llave:
                self.tabla[hh] = None
                return
        print("No existe la llave ",llave," dentro
    de la tabla")
        return
```

Código 9: Código de clase Doble Hashing

La clase descrita en el código 9 consta de tres instancias, y solo tiene un input para crearse. El input corresponde al tamaño de la tabla Hashing que se usara. Por otro lado las instancias de la clase corresponden al tamaño de la tabla (que utiliza el input y lo pasará a numero entero), la tabla (que se inicializa con 'None'), y finalmente esta un contador de colisiones que inicializa en 0. El objetivo de esta ultima instancia es contar la cantidad de colisiones que ocurren al insertar llaves a la tabla, y de esta manera tener una forma de comparar Doble Hashing con el Hashing por encadenamiento pedido en enunciado. Luego se definen las 2 funciones Hash mencionadas anteri-

ormente (método de división y multiplicación), y finalmente se definen las 3 funciones para insertar elementos, buscarlos y eliminarlos.

Por otro lado, [11] explica que la idea de Hashing por Encadenamiento es que los elementos colisionados en la misma posición de la tabla se enlazan en una lista secuencial, con cada casillero apuntando al inicio de su lista respectiva. Para implementar este método, resulta útil la implementación de colas, ya que son listas 'FIFO' (first input, first output), como indica [13]. El método FIFO permite mantener el orden de llegada de los elementos, lo que es coherente con la definición de encadenamiento donde los elementos colisionados se enlazan de manera secuencia en una lista.

```
class Cola:
    def __init__(self):
        self.q=[]
    def enq(self,x):
        self.q.insert(0,x)
    def deq(self):
        assert len(self.q)>0
        return self.q.pop()
    def is_empty(self):
        return len(self.q)==0


class HashingEnc:
    def __init__(self,size):
        self.size = int(size)
        self.tabla = [Cola() for i in range(self.
    size)] # Tabla de colas vacias
        self.colisiones = 0 # Contador de colisiones

    def hash(self,llave): # Metodo de la division
        return llave % self.size

    #Funciones
    def insert(self,llave):
        h = self.hash(llave)
        if self.tabla[h].is_empty() == False:
            self.colisiones += 1
        self.tabla[h].enq(llave)
        return

    def search(self,llave):
        h = self.hash(llave)
        for i,j in enumerate(self.tabla[h].q):
            if j == llave:
                return True
        return False

    def delete(self,llave):
        h = self.hash(llave)
        if self.search(llave) is not False:
            nueva_cola = Cola()
            eliminado = False # Para eliminar llave
    una sola vez en caso de repeticion
            while not self.tabla[h].is_empty():
                elemento = self.tabla[h].deq()
                if elemento == llave and eliminado
    == False:
                    eliminado = True # No guarda la
    llave, y setea eliminado en True (para guardar
    todo una vez eliminado)
                else:
                    nueva_cola.enq(elemento)
            self.tabla[h] = nueva_cola
        else:
            print("No existe la llave ",llave,"
    dentro de la tabla")
            return

    def display(self):
        for i, cola in enumerate(self.tabla):
            print(f" ndice  {i}: {[elemento for
    elemento in cola.q]}")
```

Código 10: Código de clase Hashin con encadenamiento

El código 10 muestra la implementación de Hashing por encadenamiento utilizando una clase de colas obtenida en [13]. Esta clase tiene una función adicional llamada display(), pues al intentar llamar al objeto usando '.tabla', se generaban problemas y no entregaba la tabla, a diferencia de en Doble Hashing. Esta clase tiene 3 instancias y 1 input al igual que la anterior, la diferencia radica en que ahora la definición de la tabla de Hash se hace iniciando una cola vacía.

Pasando a las pruebas de rendimiento, queremos comparar como trabajan estos métodos frente a las colisiones. Para esto, se generará un arreglo de números aleatorios entre con valores entre $[0,10^7]$, con un tamaño que variará entre n = $[10^3, 50 * 10^3]$. Luego, el tamaño 'm' a utilizar en las tablas de Hash debe ser un número primo, por lo visto en [11]. Como primo se utilizaran distintos valores: m = [4999,15013,24989] (obtenidos en [14]).

Los resultados son los siguientes:

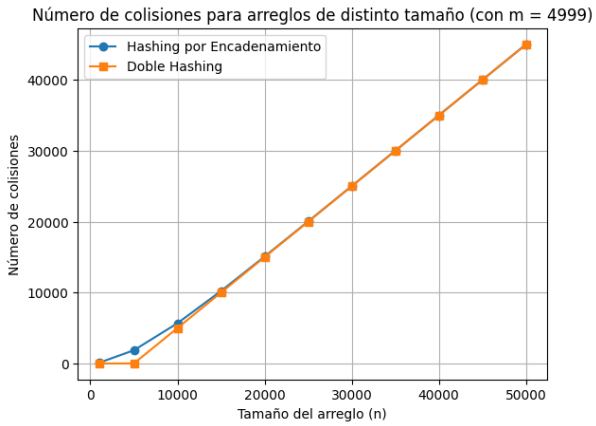

Fig. 4: Resultado de cantidad de colisiones Doble Hashing vs Hashing por encadenamiento (tamaño de tabla m = 4999)

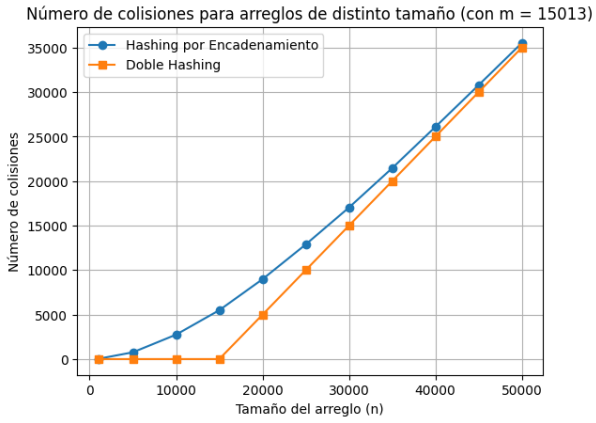

Fig. 5: Resultado de cantidad de colisiones Doble Hashing vs Hashing por encadenamiento (tamaño de tabla m = 15013)

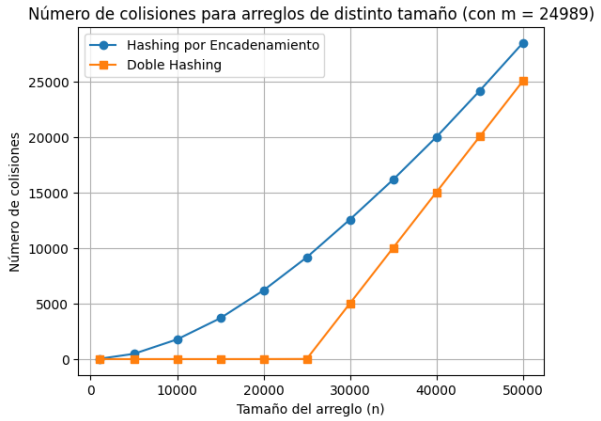

Fig. 6: Resultado de cantidad de colisiones Doble Hashing vs Hashing por encadenamiento (tamaño de tabla m = 24989)

La tendencia mostrada en las figuras 4, 5 y 6 es que a medida que aumenta el tamaño de la tabla de Hash, el algoritmo de Doble Hashing siempre es el que tarda más en presentar colisiones. Esto significa que la tabla de Hash con el método de Doble Hashing se llena de manera más eficiente que la tabla de Hash utilizando el método de Hashing por encadenamiento, lo cual es lo esperado debido a como fue diseñado Doble Hashing. La figura 4 muestra que cuando el tamaño de los datos es muy grande con respecto al tamaño de las tablas de Hash, las colisiones en ambos métodos tienden a ser iguales. Por tanto se concluye que el algoritmo de Doble Hashing es más eficiente que el método de Hashing por encadenamiento, siempre y cuando se escoja adecuadamente el tamaño de la tabla de Hash para una cantidad de datos dada.

B. Parte 2: Función de Hash para Comparación de Textos

C. Parte 3: Hash de Conteo de Elementos con Alta Frecuencia

El objetivo de esta sección es identificar los elementos que aparecen con mayor frecuencia en una lista de elementos de manera eficiente (al menos 100 000 elementos). Para esto, se reutilizará el código de Hashing por encadenamiento. Esto es debido a que, si bien hashing por encadenamiento no es igual de eficiente en la utilización de la tabla de Hash al compararlo con Doble Hashing (por lo concluido al comparar ambos métodos), sabemos que todos los elementos que sean iguales estarán en la misma posición en la tabla Hash. Esto permitirá contar a los elementos de manera más rápida y eficiente que al usar Doble Hash (pues en este ultimo los elementos pueden estar en cualquier lugar). Además, al estar los elementos agrupados en Colas, serán fáciles de trabajar y manipular una vez encontrada la posición donde están.
Dicho lo anterior, la forma de identificar los elementos será mediante una clase (código 11) y funciones que utilicen esta clase (código 12). La clase tendrá como propósito transformar una lista de elementos 'L' a un objeto Hashing con encadenamiento con tabla de Hash de tamaño 'm'. Esta clase tendrá funciones que permitirán insertar todos los elementos de 'L' en la tabla de Hash, y una función que permitirá contar la frecuencia de los elementos y guardarla en una tupla (elemento, frecuencia). Luego se crearan funciones aparte cuyo objetivo será utilizar la clase para entregar de manera automática todo el proceso que lleve contar y entregar los resultados.

```python
class Hash_frec:
    def __init__(self,size):
        self.size = int(size)
        self.Tabla = HashingEnc(self.size)

    def insert_arr(self,L):
        for i in L:
            self.Tabla.insert(i)

    def contar_frec(self):
        frec = {}
        for i, cola in enumerate(self.Tabla.tabla):
            for j in cola.q:
                if j in frec:
                    frec[j] += 1
                else:
                    frec[j] = 1
        return frec

    def Display(self):
        self.Tabla.display()
```

Código 11: Código de clase Hashin con encadenamiento para contar frecuencia de elementos en un array

```python
def frecs(L,m):
    H = Hash_frec(m)
    H.insert_arr(L)
    H.contar_frec()
    Arr = []
    for i,f in frec.items():
        Arr.append([f"{i}:",f]) #Dise ada para entregar algo similar a Counter
    return Arr

def frec_max(L,m):
    H = Hash_frec(m)
    H.insert_arr(L)
    frec = H.contar_frec()
    max_f = -1
    dato_asociado = []
    for i,f in frec.items():
        if f > max_f: #Si la frec vista es mayor que la registrada, se reemplazan valores
            max_f = f
            dato_asociado = [i]
        elif f == max_f: # Si elemento repite frec maxima, se agrupa con elementos con misma frec maxima
            dato_asociado.append(i)
    return dato_asociado,max_f
```

Código 12: Código de funciones que aplican clase Hash$_{frec}$

Para realizar la comparación de rendimiento, se utilizará el tiempo de ejecución como criterio. Los resultados son los mostrados en la figura 7.

La figura 7 muestra una diferencia abismal en la optimización de ambas funciones programadas. Counter no solo es mejor en rendimiento, sino que entrega de forma ordenada a los elementos con mayor frecuencia en un arreglo, mientras que la función propuesta entrega las frecuencias en el orden en que fueron encontradas mediante el 'for' (función 'frecs'). Por otro lado, 'max_frecs' entrega directamente al elemento (o los elementos) con mayor frecuencia, sin embargo esto no tiene mucho valor por lo dicho anteriormente de la salida ordenada de Counter. Como ventaja de las funciones creadas, se puede

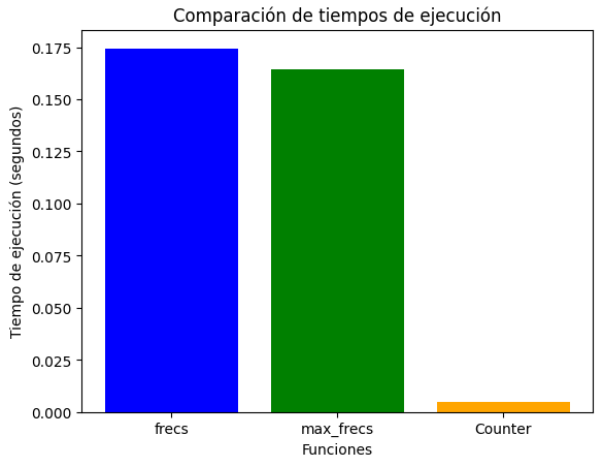

Fig. 7: Resultado de tiempo de ejecución BubbleSort v/s CombSort

destacar que están hechas a partir de una clase, y por tanto puede llegar a optimizarse. En este caso se decidió por utilizar Hashing por encadenamiento (ya que se consideró como lo más fácil e intuitivo), pero hay margen para utilizar métodos mejores que optimicen a las funciones. En conclusión, si bien no se llego a un resultado óptimo en comparación a Counter, la programación e implementación de estas funciones es intuitiva y fácil de entender, lo que les puede llegar a dar al menos una aplicación académica, más no práctica.

IV. Referencias

Referencias

[1] GeeksforGeeks, "Python program for quicksort - geeksforgeeks," https://www.geeksforgeeks.org/python-program-for-quicksort/, 2024, implementación de QuickSort en Python con explicaciones detalladas.

[2] Fiveable, "Median of three - fiveable," https://fiveable-me.translate.goog/key-terms/introduction-algorithms/median-of-three?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq&_x_tr_hist=true, 2023, explicación sobre el método Mediana de 3.

[3] M. E. Califf, "Median of three explained - quicksort optimization," https://www.youtube.com/watch?v=1Vl2TB7DoAM, 2024, video explicativo sobre la optimización Mediana de Tres en QuickSort.

[4] B. F. School, "3ns4l4d4 d3 l3tr4s: Los caracteres alfanuméricos," https://www.byjusfutureschool.com/blog/3ns4l4d4-d3-l3tr4s-los-caracteres-alfanumericos/, 2024, explicación de los caracteres alfanuméricos.

[5] A. Cloud, "¿qué es el algoritmo radix sort?" https://asimov.cloud/blog/programacion-5/que-es-el-algoritmo-radix-sort-272#:~:text=Eficiente%20para%20datos%20con%20un,sino%20solo%20los%20dgitos%20individuales., 2024, explicación y características del algoritmo Radix Sort.

[6] C. Code, "Counting sort algorithm," https://www.youtube.com/watch?v=W_euZjKoHkM, 2024, video explicativo sobre el algoritmo Counting Sort.

[7] ——, "Radix sort algorithm," https://www.youtube.com/watch?v=x-Yo8zAy9Uw, 2024, video explicativo sobre el algoritmo Radix Sort.

[8] E. C. ASCII, "El código ascii," https://elcodigoascii.com.ar/#google_vignette, 2024, referencia completa de códigos ASCII.

[9] C. Cassowary, "Comb sort algorithm," https://www.youtube.com/watch?v=D7wUVtrUHO4, 2024, video explicativo sobre el algoritmo Comb Sort.

[10] GeeksforGeeks, "Is comb sort better than bubble sort?" https://www.geeksforgeeks.org/is-comb-sort-better-than-bubble-sort/, 2024, comparativa entre Comb Sort y Bubble Sort.

[11] I. Sipirán, "Diccionarios - apuntes de aed," https://colab.research.google.com/github/ivansipiran/AED-Apuntes/blob/main/06_Diccionarios.ipynb#scrollTo=FhBOcHO4GRhB, 2024, ejemplo práctico sobre el uso de diccionarios en estructuras de datos.

[12] TutorialsPoint, "Hash functions and hash tables," https://www-tutorialspoint-com.translate.goog/hash-functions-and-hash-tables?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq#:~:text=The%20hash%20function%20used%20for,to%20get%20the%20hash%20value., 2024, explicación sobre las funciones hash y tablas hash.

[13] Sipirán, Iván. (2024) Pilas, Colas y Colas de Prioridad. Último acceso: 23 de noviembre de 2024. [Online]. Available: https://colab.research.google.com/github/ivansipiran/AED-Apuntes/blob/main/05_Pilas_Colas_y_Colas_de_Prioridad.ipynb

[14] "Números primos de 1 a 200000," 2024, consultado el 23 de noviembre de 2024. [Online]. Available: https://numerosprimos.org/numeros-primos-de-1-a-200000/