



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: TISTA GARCÍA EDGAR

Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS II

Grupo: 5

Número de Práctica(s): Guía práctica de estudio 3: Algoritmos de Ordenamiento de Datos III

Integrante(s): MURRIETA VILLEGAS ALFONSO

*Núm. De Equipo de
cómputo empleado :* 32

Semestre : 2019 - 1

Fecha de entrega: 28 DE AGOSTO DE 2018

Observaciones:

CALIFICACIÓN: _____

ALGORITMOS DE ORDENAMIENTO DE DATOS. PARTE III

INTRODUCCIÓN

Se le conoce como **algoritmo** a un conjunto finito de instrucciones libre de ambigüedades que sirven para realizar una tarea o acción específica. Por otro lado, entendemos el concepto de “ordenar” como la acción de reagrupar o reorganizar un conjunto de datos u objetos en una secuencia específica, la cual nos permite manejar información de manera más eficiente.

Dicho lo anterior, podemos definir que un algoritmo de ordenamiento es aquel que cumple con factores como la cantidad de memoria o el entorno de software para reacomodar colecciones de elementos de tal forma que todos sus elementos cumplan con una lógica respecto al criterio global ya sea ascendente o descendente. Por último, las condiciones que deben cumplirse para que se pueda considerar un algoritmo de ordenamiento de datos son las siguientes:

- 1) **Verificación:** El algoritmo siempre devuelve los elementos en orden correcto.
- 2) **Tiempo de ejecución:** Para calcular el tiempo es importante contabilizar comparaciones, intercambios, o acceso al arreglo – lista.
- 3) **Memoria:** Se debe considerar la cantidad de memoria extra necesaria para realizar operaciones.

Clasificación de algoritmos de ordenamientos

Pese existen muchos criterios para clasificar los algoritmos de ordenamiento de datos, estos se pueden clasificar en 4 grupos:

1) Algoritmos de inserción:

Se consideran un elemento a la vez y cada elemento insertado en la posición apropiada con respecto a los demás elementos que ya han sido ordenados.

Ejemplos: Shell sort, inserción binaria y hashing.

2) Algoritmos de intercambio:

En estos algoritmos se trabaja con parejas de elementos que se van comparando e intercambiando si no están en el orden adecuado. El proceso se realiza hasta que se han revisado todos los elementos del conjunto a ordenar.

Ejemplos: Bubble Sort y Quicksort.

3) Algoritmos de selección:

En estos algoritmos se selecciona el elemento mínimo o el máximo de todo el conjunto a ordenar y se coloca en la posición apropiada. Esta selección se realiza con todos los elementos restantes del conjunto.

4) Algoritmos de enumeración:

Se compara cada elemento con todos los demás y se determina cuántos son menores que él. La información del conteo para cada elemento indicará su posición de ordenamiento. Otras formas de clasificar este tipo de algoritmos es a través de la memoria usada como es el caso de: Ordenamiento interno y Ordenamiento externo. A su vez también se pueden clasificar por su estructura: Directo (Sobre el mismo elemento) e Indirecto (A través de otros elementos).

En el caso del ordenamiento interno podemos destacar que se caracteriza por el rápido acceso aleatorio a la memoria, mientras que en el ordenamiento externo se necesita recursos de la memoria secundaria.

OBJETIVOS DE LA PRÁCTICA

- El estudiante identificará la estructura de algoritmos de ordenamiento *Merge sort*, *Counting sort* y *Radix sort*

DESARROLLO

Actividad 1. Análisis del manual de prácticas

Es esta práctica los 3 métodos que se lleva se llevarán a cabo serán Merge sort, Counting sort y Radix sort.

1.1 Merge Sort

Generalización

Merge Sort es un algoritmo de ordenamiento basado en el paradigma divide y vencerás o también conocido como “Divide and conquer”.

NOTA: Divide y vencerás es una estrategia de diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño.

Consta de 3 pasos principales, dividir el problema, resolver cada sub-problema y combinar las soluciones obtenidas para la solución al problema original. A continuación, se explican con mayor detalle cada una de las fases:

Divide y vencerás:

Se divide una secuencia A de n elementos a ordenar en dos subsecuencias (arreglos o listas) de n/2 elementos. Si la secuencia se representa por un arreglo lineal, para dividirlo en 2 se encuentra un número q entre p y r que son los extremos del arreglo o lista

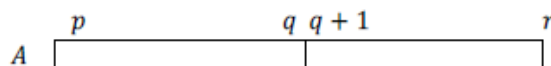


Imagen 1: Representación de cómo quedaría la lista o arreglo mediante esta estrategia.

Posteriormente se ordena las sub-listas de forma **recursiva**, donde el caso base es cuando la lista o arreglo solamente tiene un elemento (Donde por obvias razones ya está ordenado). Por último, se mezclan o conjuntan todas las sub-listas para poder obtener como resultado final la lista ya ordenada.

La parte principal de este algoritmo es sin duda la integración de las listas de forma ordenada en la fase de combinación además a través de la recursividad es como realmente se puede llevar a cabo toda la lógica.

Ejemplificación

A continuación, se muestra el pseudocódigo de Merge sort:

```
Merge(A,p,q,r)
Inicio
  Formar sub-arreglo Izq[0,..., q-p] y sub-arreglo Der[0,...,r-q]
  Copiar contenido de A[p...q] a Izq[0,..., q-p] y A[q+1...r] a Der[0,...,r-q-1]
  i=0
  j=0
  Para k=p hasta r
    Si (j >= r-q) ó (i < q-p+1 y Izq[i] < Der[j]) entonces
      A[k]=Izq[i]
      i=i+1
    En otro caso
      A[k]=Der[j]
      j=j+1
    Fin Si
  Fin Para
Fin
```

Imagen 2: Algoritmo de Merge Sort

Análisis de complejidad

En el caso de Merge Sort tenemos un algoritmo de ordenamiento de datos donde en todos los casos siempre tenemos un grado de complejidad es $O(n \log n)$

Actividad (Python)

Para esta actividad se proporcionó parte del código de Merge Sort, lo primero que se solicitó fue la salida del programa, además de agregar en el lugar adecuado la línea que imprima las sub-listas obtenidas en la recursión.

```
In [27]: runfile('C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/
Prácticas_LAB/3_practice/CODES_MANUAL/MergeSort.py', wdir='C:/Users/Murrieta/OneDrive/Documentos/
Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL')
[34, 45, 12, 98, 67]
[34]
[78]
[34, 78, 45, 12, 98, 67]
[34, 78]
[45]
[34, 45, 78, 12, 98, 67]
[12]
[98]
[34, 45, 78, 12, 98, 67]
[12, 98]
[67]
[34, 45, 78, 12, 67, 98]
[34, 45, 78]
[12, 67, 98]
[12, 34, 45, 67, 78, 98]
[12, 34, 45, 67, 78, 98]
```

Imagen 3: Salida de la lista ordenada mediante el algoritmo Merge Sort.

1.1 Counting Sort

Generalización

Es un algoritmo que no se basa en comparaciones a través del conteo de los elementos de cada clase en un rango de 0 a k para posteriormente ordenarlos determinando para cada elemento de entrada el número de elementos menores a este.

De esta forma, la lista o arreglo a ordenar solo puede utilizar elementos que sean contables(Enteros). Otro aspecto relevante es que para llevar a cabo este algoritmo son necesarios 3 arreglos:

- 1) El arreglo principal con los elementos desordenados
- 2) Un arreglo final de n elementos (N es la cantidad de elementos de la lista principal) para guardar la lista ya ordenada (Salida)
- 3) Un arreglo final con los elementos k con el propósito de ir guardando temporalmente.

A continuación, se muestran paso a paso lo que se necesita para llevar a cabo counting sort:

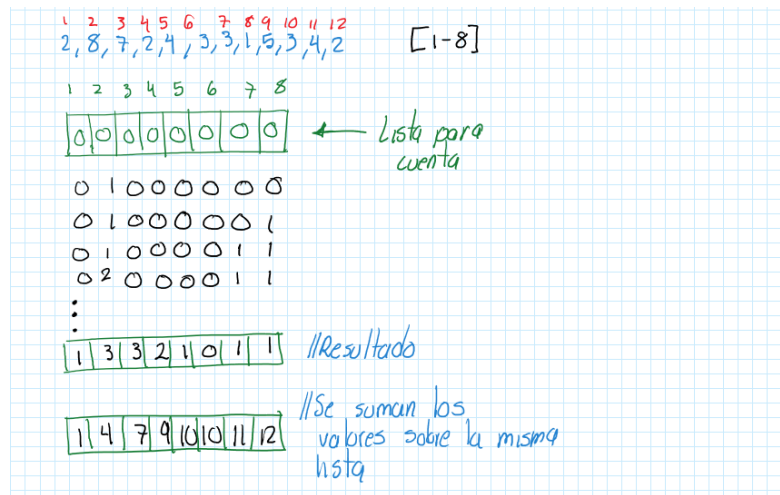


Imagen 4: De azul la lista original, de rojo las posiciones de cada elemento de la lista original, de color verde la lista auxiliar que sirve para contar la cantidad de veces que se repiten cada uno de los elementos de la lista azul.

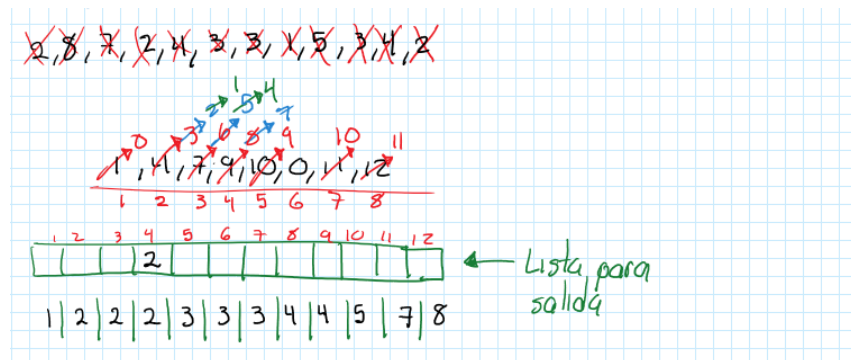


Imagen 5: Empleando la lista de color verde (La auxiliar) podemos llevar a cabo la parte de ordenamiento mediante la interacción de esta respecto a la lista original (Es importante considerar la lista final para el guardado de los elementos.)

NOTA: Counting sort es un algoritmo estable, es decir si el ordenamiento se hace con base en una relación de orden y en esa relación 2 elementos son equivalentes, entonces se preserva el orden original entre los elementos equivalentes

Análisis de complejidad

En el caso de Counting Sort tenemos un algoritmo de ordenamiento de datos donde se emplea una variable extra en los casos de complejidad, es de esta forma que para este algoritmo el grado de complejidad es $O(n+k)$ dado que para el primer y tercer ciclo un tiempo de $O(k)$ y para el segundo y último un tiempo de $O(n)$

Actividad (Python)

Para esta actividad se proporcionó parte del código de Counting Sort, lo primero que se solicitó fue la salida del programa.

```
In [46]: runfile('C:/Users/Murrieta/OneDrive/Documentos/Universidad/3º Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL/CountingSort.py', wdir='C:/Users/Murrieta/OneDrive/Documentos/Universidad/3º Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL')
Lista Original:

[0, 21, 4, 40, 10, 35, 19]
Valor más grande:
40
Cantidad de datos:
40
Indice de los datos:
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0]
Lista Ordenada:
[0, 0, 0, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 6, 6]
```

Imagen 5: Salida de la lista ya ordenada, además de que se incluyen todos los datos pedidos y requeridos para poder llevar a cabo counting sort.

Por otro lado, se pidió que a través del análisis del código se pudiera llevar a cabo, pero en su versión inversa o con la lista de forma descendente. Para ello lo primero que se realizó fue buscar la función o el ciclo for encargado de la posición de los datos, para ello directamente lo único que se hizo fue cambiar los parámetros que se le pasaba al ciclo for y las condiciones dentro de este. A continuación, se muestra la parte del código que fue editada.

```
#FUNCIÓN QUE SE CAMBIO para salida inversa
for i in range(k-1,0,-1):#Ciclo que determina la posición de los datos
    C[i]=C[i]+C[i+1]
print("Índice de los datos: ")
print(C)
```

Imagen 6: Parte editada del código para obtener la salida en forma descendente o inversa.

Por último, se muestra la salida en forma descendente del algoritmo:

```
In [48]: runfile('C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL/CountingSort_Inversa.py', wdir='C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL')
Lista Original:
```

```
[0, 21, 4, 40, 10, 35, 19]
```

```
Valor más grande:
```

```
40
```

```
Cantidad de datos:
```

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0]
```

```
Indice de los datos:
```

```
[0, 6, 6, 6, 6, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 0]
```

```
Lista Ordenada:
```

```
[0, 40, 35, 21, 19, 10, 4]
```

Imagen 7: Salida de la lista ya ordenada, además de que se incluyen todos los datos pedidos y requeridos para poder llevar a cabo counting sort de forma descendiente.

1.1 Radix Sort

Generalización

El ordenamiento de datos Radix Sort o también conocido como ordenamiento por residuos puede utilizarse cuando los valores a ordenar están compuestos por secuencias de letras o dígitos que admiten un orden lexicográfico.

NOTA: El algoritmo ordena utilizando un algoritmo de ordenamiento estable, está basado en los valores absolutos de los dígitos de los números que son ordenados.

A continuación, se muestra a través de una representación visual hecha en OneNote, cómo es que se lleva a cabo el algoritmo de ordenamiento de Radix sort:

Initial List: 1023, 1122, 2121, 0011, 3012, 2222

Colas (Centenas)

Q ₀ [0]	0011, 3012, 1023
Q ₁ [1]	1122, 2121
Q ₂ [2]	2222
Q ₃ [3]	

∴ 0011, 3012, 1023, 1122, 2121, 2222

Colas (Sigüiente Dígito)

Q ₀ [0]	0011, 3012
Q ₁ [1]	1023, 1122, 2121, 2222
Q ₂ [2]	
Q ₃ [3]	

∴ 0011, 1023, 1122, 2121, 2222, 3012

Imagen 9: Del lado izquierdo se encuentra la en color azul la lista inicial a ordenar, posteriormente en la primera iteración o ciclo lo que se realiza es un “ordenamiento” a través de la primera cifra de los dígitos que es la ubicada en las unidades, posteriormente tenemos como resultado una lista que sigue estando desordenada. Posteriormente del lado derecho, se realiza el mismo proceso hasta llegar a la última posición de los dígitos que en ese caso fue el de los millares, como resultado final tenemos a la lista ya ordenada.

Análisis de complejidad

En el caso de Radix Sort tenemos un algoritmo de ordenamiento de datos donde se emplea una variable extra en los casos de complejidad, es de esta forma que para este algoritmo el grado de complejidad es $O(nk)$ en todos los casos.

Actividad (Python)

Para esta actividad se proporcionó parte del código de Counting Sort, lo primero que se solicitó fue la salida del programa.

```
In [49]: runfile('C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL/
RadixSort.py', wdir='C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL')
Lista original:
['000000', 'X17FS6', 'PL4ZQ2', 'JI8FR9', 'XL8F16', 'PY2ZR5', 'KV7WS9', 'JL2ZV3', 'KI4WR2']
['000000', 'PL4ZQ2', 'KI4WR2', 'JL2ZV3', 'PY2ZR5', 'X17FS6', 'XL8F16', 'JI8FR9', 'KV7WS9']
['000000', 'XL8F16', 'PL4ZQ2', 'KI4WR2', 'PY2ZR5', 'JI8FR9', 'X17FS6', 'KV7WS9', 'JL2ZV3']
['000000', 'XL8F16', 'JI8FR9', 'X17FS6', 'KI4WR2', 'KV7WS9', 'PL4ZQ2', 'PY2ZR5', 'JL2ZV3']
['000000', 'PY2ZR5', 'JL2ZV3', 'KI4WR2', 'PL4ZQ2', 'X17FS6', 'KV7WS9', 'XL8F16', 'JI8FR9']
['000000', 'X17FS6', 'KI4WR2', 'JI8FR9', 'JL2ZV3', 'PL4ZQ2', 'XL8F16', 'KV7WS9', 'PY2ZR5']
['000000', 'JI8FR9', 'JL2ZV3', 'KI4WR2', 'KV7WS9', 'PL4ZQ2', 'PY2ZR5', 'X17FS6', 'XL8F16']
Lista ordenada:
['000000', 'JI8FR9', 'JL2ZV3', 'KI4WR2', 'KV7WS9', 'PL4ZQ2', 'PY2ZR5', 'X17FS6', 'XL8F16']
```

Imagen 10: Salida de la lista ya ordenada, además de que se incluyen todos los datos pedidos y requeridos para poder llevar a cabo Radix sort.

Por otro lado, se pidió que a través del análisis del código se pudiera llevar a cabo, pero en su versión inversa o con la lista de forma descendente. Para ello lo primero que se realizó fue buscar la función o el ciclo for encargado de la posición de los datos, para ello directamente lo único que se hizo fue cambiar los parámetros que se le pasaba al ciclo for y las condiciones dentro de este. A continuación, se muestra la parte del código que fue editada.

```
for i in range(k-1,0,-1):#Parte del código que fue editada
    C[i]=C[i]+C[i+1]
```

Imagen 11: Parte editada del código para obtener la salida en forma descendente o inversa.

Por último, se muestra la salida en forma descendente del algoritmo:

```
In [50]: runfile('C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/CODES_MANUAL/
RadixSort_Inversa.py', wdir='C:/Users/Murrieta/OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/3_practice/
CODES_MANUAL')
Lista original:
['000000', 'X17FS6', 'PL4ZQ2', 'JI8FR9', 'XL8F16', 'PY2ZR5', 'KV7WS9', 'JL2ZV3', 'KI4WR2']
['000000', 'JI8FR9', 'KV7WS9', 'X17FS6', 'XL8F16', 'PY2ZR5', 'JL2ZV3', 'PL4ZQ2', 'KI4WR2']
['000000', 'JL2ZV3', 'KV7WS9', 'X17FS6', 'JI8FR9', 'PY2ZR5', 'KI4WR2', 'PL4ZQ2', 'XL8F16']
['000000', 'JL2ZV3', 'PY2ZR5', 'PL4ZQ2', 'KV7WS9', 'KI4WR2', 'X17FS6', 'JI8FR9', 'XL8F16']
['000000', 'JI8FR9', 'XL8F16', 'KV7WS9', 'X17FS6', 'PL4ZQ2', 'KI4WR2', 'JL2ZV3', 'PY2ZR5']
['000000', 'PY2ZR5', 'KV7WS9', 'XL8F16', 'PL4ZQ2', 'JL2ZV3', 'JI8FR9', 'KI4WR2', 'X17FS6']
['000000', 'XL8F16', 'X17FS6', 'PY2ZR5', 'PL4ZQ2', 'KV7WS9', 'KI4WR2', 'JL2ZV3', 'JI8FR9']
Lista ordenada:
['000000', 'XL8F16', 'X17FS6', 'PY2ZR5', 'PL4ZQ2', 'KV7WS9', 'KI4WR2', 'JL2ZV3', 'JI8FR9']
```

Imagen 12: Salida de la lista ya ordenada, además de que se incluyen todos los datos pedidos y requeridos para poder llevar a cabo Radix sort de forma descendiente.

Actividad 2. Ejercicios de laboratorio

2.1 Merge Sort

Para este problema se proporcionaron previamente una biblioteca con el algoritmo de ordenamiento Merge Sort. A continuación, se explica función a función como es que se puede llevar a cabo la salida del programa:

1] Análisis del programa de prueba

En este programa solamente existen 2 funciones, una que es la función “printArray” y otra que es la función “main”.

En el caso de la función **printArray** es una función que imprime el arreglo que se le pasa a través de un simple ciclo for.

Por otro lado, la función **main** es aquella que contiene tanto el arreglo como el orden en que se irán llamando a las demás funciones necesarias para ordenar a la lista inicial.

2] Análisis de la biblioteca merge.h

En esta biblioteca solamente existen 2 funciones, “mergeSort” y “merge” a continuación se explican brevemente cada una de las funciones:

La función “mergeSort” es aquella encargada que a través de los 3 parámetros que se le pasan (el arreglo, el punto donde inicia o valor inicial y el tamaño del arreglo) es como este puede llevar a cabo el ordenamiento de todos los datos mediante la recursividad.

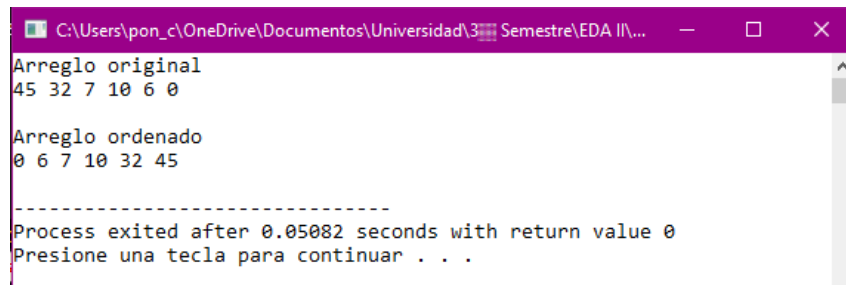
Cabe destacar que cada vez que se llama a esta función parte a la mitad el arreglo que se le pasa. NOTA: Cabe destacar que el caso donde termina la recursividad es cuando la dimensión de los arreglos es de 1.

Por otro lado, tenemos la función merge la cual se le pasan el arreglo, el valor inicial (El que empieza en el arreglo), la dimensión del arreglo y por último la posición donde se ubica la mitad del arreglo.

A través de los datos mencionados previamente, es como directamente la función merge se encarga de ir verificando la posición y la unión de cada una de las listas para de esta forma poder llevar a cabo el ordenamiento de los elementos del arreglo inicial.

En el caso de los ciclos while, estos sirven para llevar a cabo el ordenamiento de los datos, esto es posible debido a que dentro de sus condiciones se va verificando que los elementos del lado de la posición 0 o inicial del arreglo sean menores al elemento de la mitad del arreglo además también a través de la dimensión del arreglo es como se lleva a cabo la otra parte del ordenamiento que es aquella que todos los elementos mayores al de la mitad del arreglo se encuentren del lado “derecho”.

Por último, se muestra la salida del algoritmo Merge Sort:



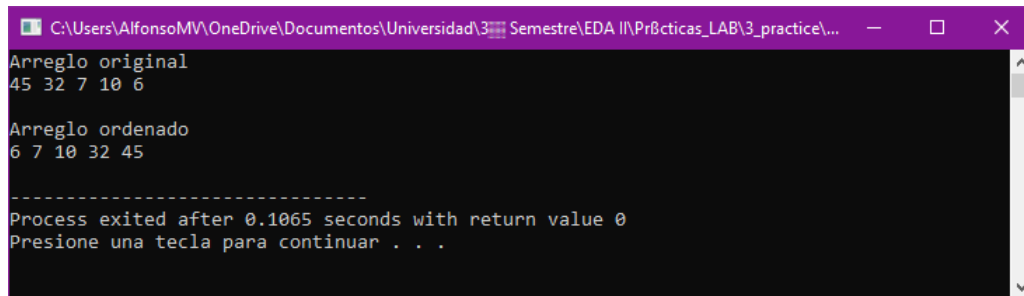
```
C:\Users\pon_c\OneDrive\Documentos\Universidad\3 Semestre\EDA II\...
Arreglo original
45 32 7 10 6 0

Arreglo ordenado
0 6 7 10 32 45

-----
Process exited after 0.05082 seconds with return value 0
Presione una tecla para continuar . . .
```

Imagen 13: Salida de la lista ya ordenada a través de Merge Sort

NOTA: Respecto a la actividad A del ejercicio 1, cabe destacar que en Windows al llevar a cabo la ejecución del algoritmo siempre se tomaba probablemente como pivote un 0 el cual ocupaba un espacio en el arreglo, es por ello que al momento de imprimir nos recortaba el arreglo ordenado, para arreglar este pequeño detalle simplemente podemos partir de que empiece en una posición antes o al momento de imprimir que agregue una iteración más.



```
C:\Users\AlfonsoMV\OneDrive\Documentos\Universidad\3 Semestre\EDA II\Prácticas_LAB\3_practice\...
Arreglo original
45 32 7 10 6

Arreglo ordenado
6 7 10 32 45

-----
Process exited after 0.1065 seconds with return value 0
Presione una tecla para continuar . . .
```

Imagen 14: Salida de la lista ya ordenada a través de Merge Sort cambiando la línea donde se le pasaban los datos a la función mergeSort(arr, 0, arrS-1);

La razón de porque este mínimo cambio funciona es porque al final el arreglo contiene todos los elementos, realmente no es como que se pierda un elemento o algo parecido.

2.2 Counting Sort

Análisis y consideraciones

Para poder realizar un buen código lo primero que se realizó fue un análisis y revisión de los requisitos que se pedía para llevar el código a cabo. A continuación, se presentan las consideraciones y requerimientos necesarios para el código de Counting Sort:

- 1) Utilizar alguna estructura (arreglo, lista, etc) que tenga una dimensión de 20 elementos y sea de caracteres.
 - a. Los elementos los ingresará el usuario
 - b. El rango de búsqueda debe estar acotado entre las letras A y H
- 2) Debido a la forma en que se lleva el algoritmo de ordenamiento, se debe crear una segunda

estructura donde cada posición será asociada a una de las letras que serán ingresadas por el usuario.

- 3) En la primera pasada el algoritmo deberá contar las apariciones de cada una de las letras ingresadas por el usuario. (Estructura auxiliar)
- 4) Realizar la cuenta de la estructura; en cada índice se considerará la cantidad de elementos actuales y anteriores (La lista de como irá descendiendo la cantidad de elementos tenidos en la estructura de conteo)
 - a. Se da la opción de poder llevar a cabo otro arreglo o lista en caso de problemas.
- 5) Crear una estructura (La lista/arreglo final) donde se ingresarán los elementos ya guardados

Programación y ejecución

La verdad este creo es uno de los códigos menos óptimos que he hecho sobre todo por la manera tan burda en que llevé a cabo tanto las comparaciones como la forma en que abordé el problema. Lo primero que realicé fue emplear 2 clases al igual que en la práctica anterior (El código de ejemplo del profesor) a continuación se explica brevemente que hace cada clase y sus respectivos métodos:

Clase “Counting Sort”

Solamente contiene un método que es el método main, en ese método se hace la declaración y creación de las variables y objetos necesarios del algoritmo, además contiene un ciclo for encargado de ingresar los datos dentro de un arreglo.

```
public class Counting_Sort {  
    public static void main(String[] args) {
```

Imagen 15: Clase counting sort junto a su único método.

Clase “Counting Sort”

Al igual que la otra clase, este método solamente contiene un método llamado “countingSort” el cual es el encargado de llevar todo el ordenamiento de los elementos del arreglo.

Para ello fue necesario crear todos los casos de comparación de las letras, esto a través de la inicialización y asimilación de índices (Variables llamadas index).

Por otro lado, a través de 2 ciclos for es como se lleva el ordenamiento, en el primer ciclo for lo que se hace es el conteo y guardado de la cantidad de cada letra, mientras que el segundo ciclo for es el encargado de llevar a cabo el ordenamiento de todos los elementos del arreglo.

Ejecución:

A continuación, se muestra la salida del programa con los requisitos solicitados:

```
Output
radix_Sort_Java (run) x Counting_Sort (run) x
ENTRA: D D tiene: : 4
ENTRA: A [null, null, null, null, null, A, null, null, null, null, null, null, null, null, null, null, null]
ENTRA: D [null, null, null, null, null, A, null, null, null, null, null, null, null, D, null, null, null]
ENTRA: C [null, null, null, null, null, A, null, null, null, null, null, C, null, null, null, D, null, null]
ENTRA: E [null, null, null, null, null, A, null, null, null, null, null, C, null, null, null, D, null, null]
ENTRA: A [null, null, null, null, A, A, null, null, null, null, null, C, null, null, null, D, null, null]
ENTRA: B [null, null, null, null, A, A, null, B, null, null, null, C, null, null, null, D, null, null]
ENTRA: E [null, null, null, null, A, A, null, B, null, null, null, C, null, null, null, D, null, null]
ENTRA: D [null, null, null, null, A, A, null, B, null, null, null, C, null, null, D, D, null, null]
ENTRA: C [null, null, null, null, A, A, null, B, null, null, C, C, null, null, D, D, null, null]
ENTRA: A [null, null, null, A, A, A, null, B, null, null, C, C, null, null, D, D, null, null]
ENTRA: E [null, null, null, A, A, A, null, B, null, null, C, C, null, null, D, D, null, null]
ENTRA: A [null, null, A, A, A, A, null, B, null, null, C, C, null, null, D, D, null, null]
ENTRA: C [null, null, A, A, A, A, null, B, null, C, C, C, null, null, D, D, null, null]
ENTRA: C [null, null, A, A, A, A, null, B, C, C, C, C, null, null, D, D, null, null]
ENTRA: B [null, null, A, A, A, A, B, B, C, C, C, C, null, null, D, D, null, null]
ENTRA: A [null, A, A, A, A, A, B, B, C, C, C, C, null, null, D, D, null, null]
ENTRA: D [null, A, A, A, A, A, B, B, C, C, C, C, null, D, D, D, null, null]
ENTRA: A [A, A, A, A, A, A, B, B, C, C, C, C, null, D, D, D, null, null]
ENTRA: E [A, A, A, A, A, A, B, B, C, C, C, C, null, D, D, D, E, E, E]
ENTRA: D [A, A, A, A, A, A, B, B, C, C, C, C, D, D, D, D, E, E, E]
La lista ordenada es:
[A, A, A, A, A, A, B, B, C, C, C, C, D, D, D, D, E, E, E, E]
BUILD SUCCESSFUL (total time: 19 seconds)
```

Imagen 16: Ordenamiento de datos (caracteres) a través de counting sort, hasta arriba el conteo de elementos, hasta abajo la lista ya ordenada.

2.3 Radix Sort

Análisis y consideraciones

Para poder realizar un buen código lo primero que se realizó fue un análisis y revisión de los requisitos que se pedía para llevar el código a cabo. A continuación, se presentan las consideraciones y requerimientos necesarios para el código de Radix Sort:

- 1) Los datos ingresados serán números de 4 dígitos, de los cuales estarán acotados del 0 a 3.
- 2) Los datos serán ingresados por el usuario y serán guardados en cualquier tipo de estructura, lista o arreglo.
- 3) Se deberá implementar una cola para cada uno de los dígitos anteriormente mencionados (0,1,2,3)
 - a) En esta opción se *recomienda* emplear C o Java.
- 4) Debido a la forma en que se lleva a cabo el algoritmo es necesario pasar por los elementos considerando que se deben guardar en sus respectivas colas, considerando en la posición decimal en la que se encuentra.
 - a. NOTA: Debe mostrarse en cada iteración
- 5) Por último, se debe mostrar la lista o arreglo ordenado.

Programación y ejecución

Para este programa definitivamente descarté utilizar Python debido a que pese pudo haberse llevado a cabo la codificación de una manera amigable el problema era que no sabía de que forma llevar a cabo los acotamientos y sobre todo utilizar colas dentro de este.

Por otro lado, C lo tuve que descartar debido a la falta de tiempo durante la semana y es que realmente hacerlo en C me hubiera llevado mucho más tiempo.

Es de esta forma que al final decidí utilizar Java, además de que recientemente en la clase de POO ví una estructura de datos u objetos que se le conoce como ArrayList el cual tiene la misma mecánica que una cola como son las funciones básicas de agregar (o encolar), borrar (o desencolar), entre otras.

A continuación, se muestran y explican todos los métodos que se utilizaron para poder llevar a cabo el ordenamiento de datos a través del algoritmo radix Sort

Método Main

Es el encargado de llamar todos los demás métodos, es del tipo estático y no regresa nada. Algo destacable es que en este método fue donde decidí meter una de las condiciones impuestas en el problema que es aquella de validar que los números ingresados por el usuario sean solamente de 4 dígitos, esto fue realmente fácil debido que simplemente se debe comparar el tamaño lo cual en Java es un método ya del mismo. (`_.length`)

```
public static void main(String[] args) {  
  
    ArrayList<Integer> Main_list= new ArrayList<>();  
    Scanner sc= new Scanner(System.in);  
  
    int cant;//Cantidad de elementos  
    int i= 0;//contador
```

Imagen 17: Parte del código que muestra el método main

Por último, en este método es donde se declaran las variables y objetos de todo el programa un ArrayList para todos los elementos ingresados por el usuario y todas las variables que se emplearán en el algoritmo.

Método sort

Es el encargado de toda la lógica del programa, es por ello que lo que necesita es crear 4 colas para poder cumplir con las condiciones que se plantearon anteriormente, por otro lado, esto es posible a través de un ciclo for el cual tiene varias funciones

1) La primera función es dar inicio a un ciclo do while el cual es el encargado de ir agregando los elementos de la lista en sus correspondientes colas, esto respecto a la iteración en la que se encuentra el ciclo for.

NOTA: Cada vez que se evalúa en el ciclo do while se hace un llamado a la función `printQueue`

2) La segunda función es llamar tanto a la función `printQueue` indirectamente como a la función `clearQueue`

3) `clearQueue`

Esta es el método encargada de ir limpiando las colas conforme se da una iteración en el ciclo for del método sort, este método solamente se llama 4 veces en cada iteración debido a que se llama una vez por cada cola

```

public static void clearQueue(ArrayList <Integer> elementos, ArrayList <Integer> queue){
    if(!queue.isEmpty()){
        while(!queue.isEmpty()){
            elementos.add(queue.remove(0));
        }
    }
}

```

Imagen 18: Parte del código que muestra el método clearQueue

printQueue

Por último, tenemos el método printQueue que es el encargado de llevar las impresiones de las 4 colas cada vez que se hace una iteración en el ciclo for, esto sobre todo fue utilizado para poder ver como progresaba el ordenamiento mientras se hacía el código, sin embargo, también se dejó debido a que es un requisito en el ejercicio.

```

//Método encargado de imprimir las colas
public static void printQueue(ArrayList <Integer> queue_0, ArrayList <Integer> queue_1, ArrayList <Integer> queue_2, ArrayList <Integer> queue_3){
    System.out.println("Los elementos están en las colas");
    System.out.println("[0]: "+queue_0);
    System.out.println("[1]: "+queue_1);
    System.out.println("[2]: "+queue_2);
    System.out.println("[3]: "+queue_3);
}

```

Imagen 19: Parte del código que muestra el método printQueue

Ejecución:

A continuación, se muestra la salida del programa con los requisitos solicitados:

```

Output - radix_Sort_Java (run)
[2]: []
[3]: []
Los elementos están en las colas
[0]: []
[1]: [1010]
[2]: [2011]
[3]: []
Los elementos están en las colas
[0]: []
[1]: [1010]
[2]: [2011]
[3]: [3121]
Los elementos están en las colas
[0]: []
[1]: [1010, 1212]
[2]: [2011]
[3]: [3121]

Lista Ordenada: [1010, 1212, 2011, 3121]
BUILD SUCCESSFUL (total time: 16 seconds)

```

Imagen 19: Salida de la lista en orden a través del algoritmo “Radix Sort”

NOTAS IMPORTANTE:

El programa no cumple solamente con una condición de las previas mencionadas que es aquella que solamente tenga números de 4 dígitos pero solamente con números entre el 0 y 3.

Probable Solución

Tal vez si utilizará alguna forma de poder ver los enteros como cadenas para comprobar los elementos particulares podría llevarse a cabo la condición faltante del programa.

Conclusiones

Durante esta práctica se conocieron 3 algoritmos de ordenamiento, fue el caso de Merge sort, Counting sort y Radix sort.

Algo que debo mencionar es que realmente he notado que me gusta el suficiente tiempo para poder desarrollar cada uno de los apartados de la práctica esto sobre todo porque en la primera versión solamente pude terminar con totalidad el ejercicio 1 pero obviamente bien hecho, creo que realmente prefiero hacer menos, pero con mejor calidad a hacer muchas cosas a medias o incluso mal hechas.

Por otro lado, como se lo mencioné en el laboratorio, la importancia de saber cómo se llevan o realizan muchos algoritmos importantes como son los de ordenamiento de datos o los de búsqueda son sobre todo para poder comprender como es que realmente se llevan a cabo los procesos internos de muchas bibliotecas, métodos (Hablando tal vez de algunos lenguajes orientado a objetos) o incluso de códigos previamente hechos y además que en dado caso de que se tenga un problema en el cual no se pueda recurrir directamente a alguna herramienta previamente programa o que se tengan limitaciones ya sean de memoria o tal vez de tiempo uno como buen programador sepa como poder abordar y resolver el problema de la forma óptima y correcta.

Por último, sinceramente el llevar a cabo ambos algoritmos de ordenamiento fue bastante complicado sobre todo porque a pesar de que la gran mayoría de los programas que he hecho han sido en Python esta vez tuve muchas limitaciones debido a que en otros lenguajes como es C o Java ya tenía mayor idea de cómo resolver o al menos abordar los ejercicios. Por un lado, el algoritmo de counting sort en verdad me pareció que lo hice de una forma muy burda sobre todo porque debe existir una forma más apropiada de llevar a cabo ordenamiento de caracteres no sé tal vez por algún código o implementando el ASCII, por otro lado, realmente me gustó muchísimo el haber programado Radix ya que realmente fue un poco como un reto debido a que recientemente en POO vimos algunas estructuras como son ArrayList, Hashtable, etc y la verdad esa fue la razón principal de por qué decidí hacerlo en java y no en Python.

COMENTARIO: Espero que conforme pase el tiempo realmente pueda implementar otros lenguajes obvio con el permiso del profesor.

REFERENCIAS

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). Problem Solving with Algorithms and Data Structures using Python. Segunda Edición.

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.

Elba Karen Saenz García. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos II*. UNAM, Facultad de Ingeniería.