



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Tista García Edgar Ing.

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 5

No de Práctica(s): 3

Integrante(s): Téllez González Jorge Luis

*No. de Equipo de
cómputo empleado:* 37

No. de Lista o Brigada: X

Semestre: 2020-2

Fecha de entrega: 23/02/2020

Observaciones:

CALIFICACIÓN: _____



Índice

1. Introducción	2
1.1. Algoritmos de distribución: Counting-Sort y Radix-Sort	2
1.2. Algoritmos de intercalación: Merge-Sort	4
2. Objetivos	6
3. Implementaciones en Java	6
3.1. Propuesta inicial	6
3.2. Radix-Sort	6
3.3. Counting-Sort	8
4. Merge-Sort en Java	10
5. Conclusiones	14

1. Introducción

1.1. Algoritmos de distribución: Counting-Sort y Radix-Sort

Los algoritmos de ordenamiento poseen diferentes enfoques de acuerdo a los métodos que utilicen con el fin de lograr su objetivo. A diferencia de los enfoques analizados previamente, los algoritmos *de distribución* tienen como característica principal el uso de **estructuras externas** al bloque de información a ordenar con el fin de lograr su objetivo.

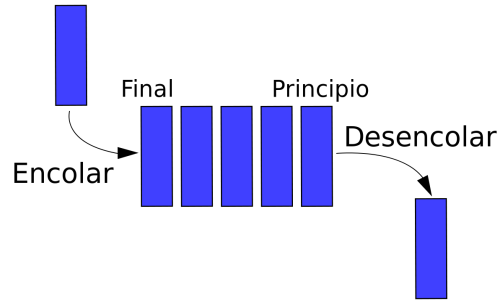


Figura 1: Entre las posibles estructuras externas a utilizar se encuentran las colas.

Entre los algoritmos que utilizan este enfoque se encuentran *Counting-Sort* y *Radix-Sort*. El primer algoritmo, con base en un rango predefinido de valores, utiliza 2 estructuras secundarias con el fin de ordenar el conjunto original de información. La primer estructura se encarga de contar la frecuencia de aparición de cada dato en el conjunto a ordenar, posteriormente, se utiliza la segunda estructura para realizar un conteo de *distribución acumulada*. Finalmente, y con base en la suma acumulada presente en la segunda estructura, se ordenan los elementos del conjunto.

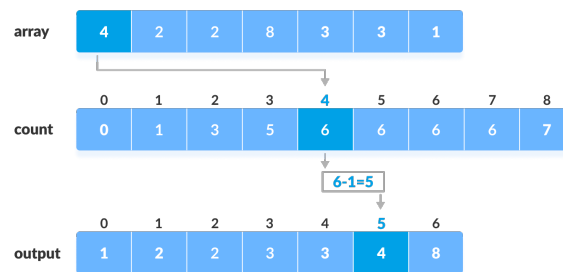


Figura 2: Funcionamiento de Counting-Sort. En este caso únicamente se presenta la segunda estructura.

Como puede observarse, cuando un elemento dentro del rango predefinido no se encuentra en el conjunto a ordenar, no se realiza ninguna adición y el valor de la suma acumulada permanece constante. Para asignar la ubicación de un elemento en el arreglo se considera el valor de la suma acumulada en su índice correspondiente. En el ejemplo anterior, el índice 2 posee una suma acumulada de 3, por lo tanto, le

corresponderá la posición $3 - 1 = 2$ en el arreglo ordenado. Si el elemento se encuentra repetido, se procede a colocar estos elementos a la izquierda del primero.

Otra propuesta se encuentra representada por *Radix-Sort*, el cual opta por el uso de estructuras de datos del tipo *First-In/First Out*. Su funcionamiento se basa en los valores absolutos de los dígitos presentes en los números a ordenar. Debido al uso de estructuras de datos, su requerimiento de memoria es superior a comparación de otros algoritmos de ordenamiento.

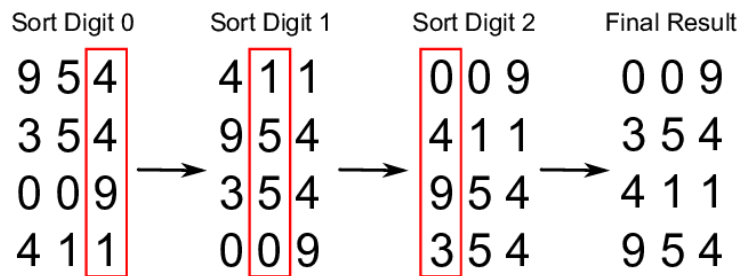


Figura 3: Se analizan y ordenan de forma consecutiva los elementos con base en la unidades, decenas, centenas, etc. presentes en los números.

Un aspecto destacable de este algoritmo reside que, por ejemplo, si se tiene un rango de [0-3] en los dígitos de los números a ordenar, serán necesarias 4 estructuras **adicionales** para cada dígito (0, 1, 2 y 3). Si se utilizaran todos los dígitos, se requerirán hasta 10 estructuras para cada uno de ellos.

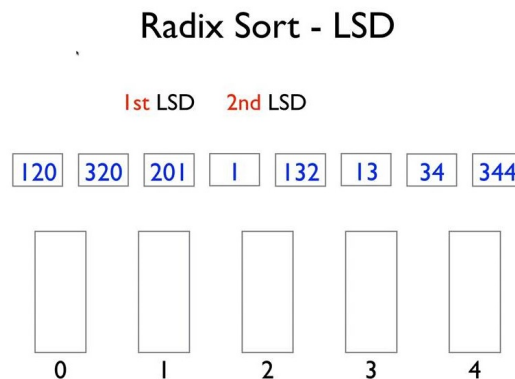


Figura 4: Estructuras individuales para cada dígito presente.

La complejidad promedio de *Counting-Sort* es de $O(n + k)$, donde k representa el factor representativo de la memoria utilizada por las estructuras adicionales. En cambio, *Radix-Sort* posee una complejidad lineal de $O(nk)$ para todos sus posibles escenarios.

1.2. Algoritmos de intercalación: Merge-Sort

Uno de los algoritmos de ordenamiento interno más célebres fue creado por el matemático húngaro-estadounidense John von Neumann en 1945. Un aspecto notable de este algoritmo reside en su equilibrio con respecto a otros algoritmos de ordenamiento y su admirable complejidad de $O(n * \log(n))$ en cada uno de sus posibles escenarios.

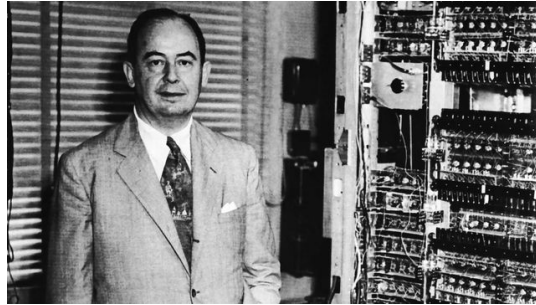


Figura 5: La computación moderna tien entre sus mayores aportadores a Neumann.

Merge-Sort utiliza la estrategia *Divide y Vencerás* de forma notable: su idea básica consiste en combinar dos listas ordenadas previamente para obtener finalmente una lista ordenada más grande. Los pasos básicos de este algoritmo son los siguientes:

1. Dividir la lista de elementos en 2 sub-listas de $(n/2)$ elementos cada una.
2. Ordenar las dos sub-listas con Merge-Sort.
3. Intercalar las dos sub-listas ordenadas para, finalmente, obtener la lista final completamente ordenada.

El único aspecto negativo de este algoritmo con respecto de otras propuestas reside en su uso de *memoria adicional* para la creación recursiva de sub-listas.

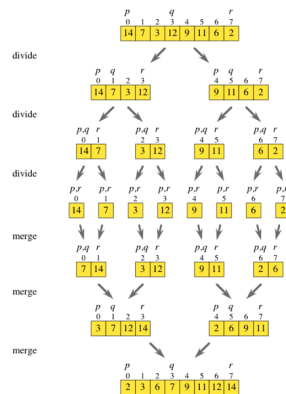


Figura 6: El algoritmo divide la lista hasta llegar a un caso trivial de ordenamiento.

Merge-Sort hace uso extensivo del concepto de recursividad durante el proceso de división de cada una de las listas con el fin de optimizar el tiempo de procesamiento de cada una de ellas. Un análisis extenso del proceso de división y combinación finalmente lleva a su fórmula de recurrencia de $T(n) = 2T(n/2) + \Theta(n)$.

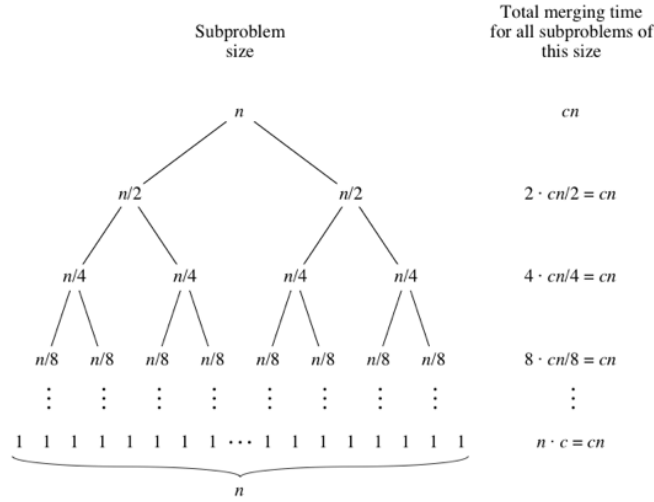


Figura 7: Análisis gráfico del proceso de división.

Merge-Sort se llama a sí mismo en 2 ocasiones diferentes. realizando una operación de división recursiva en la lista de entrada. Finalmente, se realiza una operación de combinación entre las sub-listas correspondientes.

El **Teorema del método maestro**, introducido por primera vez en el libro *Introducción a los algoritmos* en 1990, permite hallar formalmente la complejidad de *Merge-Sort* al ubicarse en el caso de recurrencia $T(n) = \Theta(n^{\log_b(a)})$.

Dados $a \geq 1$ y $b > 1$ y la recurrencia

$$T(n) = aT(n/b) + f(n),$$

entonces

$$T(n) \in \begin{cases} \Theta(n^{\lg_b a}) & \text{si } \exists \epsilon > 0. f(n) \in O(n^{\lg_b a - \epsilon}) \\ \Theta(n^{\lg_b a} \lg n) & \text{si } f(n) \in \Theta(n^{\lg_b a}) \\ \Theta(f(n)) & \text{si } \exists \epsilon > 0. f(n) = \Omega(n^{\lg_b a + \epsilon}) \\ & \text{y } \exists c < 1, N \in \mathbb{N}. \forall n > N. \\ & af(n/b) \leq cf(n) \end{cases}$$

Figura 8: El Teorema Maestro proporciona una herramienta formal para demostrar la complejidad computacional de algoritmos recursivos como *Merge-Sort* con base en su ecuación de recurrencia.

2. Objetivos

- El estudiante identificará la estructura de los algoritmos de ordenamiento *Merge-Sort*, *Counting-Sort* y *Radix-Sort*.
- Realizar una implementación en el lenguaje Java de los algoritmos de ordenamiento *Counting-Sort* y *Radix-Sort* con los requerimientos solicitados.
- Analizar a profundidad el funcionamiento de *Merge-Sort*.

3. Implementaciones en Java

3.1. Propuesta inicial

El entorno de trabajo del lenguaje orientado a objetos **Java** proporciona herramientas útiles para el desarrollo e implementación de *Radix-Sort* y *Counting-Sort*. Con el fin de implementar adecuadamente ambos algoritmos, se creó un proyecto denominado *Pr3* con un paquete que incluyese 4 clases: *Pr3*, *CountingSort*, *RadixSort* y *Utilerías*. En la clase principal del proyecto fue implementado un menú similar al presentado en las prácticas anteriores.

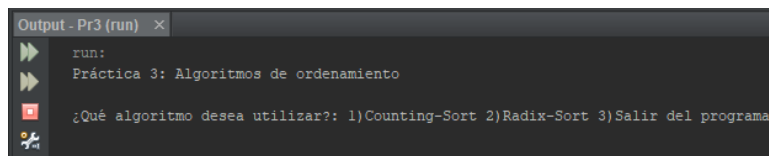


Figura 9: Menú de selección del programa creado.

La clase *Utilerías* posee en su interior 2 métodos estáticos dedicados a imprimir arreglos de caracteres y enteros, respectivamente.

A continuación se detallarán las implementaciones propuestas para cada algoritmo.

3.2. Radix-Sort

Java permite la creación de objetos de tipo *ArrayList*, los cuales funcionan como un arreglo redimensionable que aumenta su tamaño conforme crece la colección de elementos introducidos en el mismo. Esta característica fue utilizada para establecer las estructuras requeridas por *Radix-Sort* para cada dígito presente.

Una vez declaradas las referencias correspondientes, se establece un método de inicialización para cada una de las listas a utilizar.

Posteriormente, el método **RadixSort** se encarga de realizar el ordenamiento respectivo. Para ello, se utiliza dos ciclos for anidados, dónde el primer ciclo toma en cuenta el número de dígitos presentes y

el segundo ciclo recorre la lista de entrada utilizando un *for-each*. Cada uno de los elementos es leído e introducido en su respectiva lista de acuerdo al dígito presente en la posición $j - 1$.

```
//Declaración de listas para cada dígito
private ArrayList<String> lista3;
private ArrayList<String> lista4;
private ArrayList<String> lista5;
private ArrayList<String> lista6;
```

Figura 10: Listas adicionales utilizadas para cada dígito.

```
public ArrayList<String> RadixSort(ArrayList<String> lista) {
    int k=0;
    for (int j = 4; j >= 1; j--) {
        for (String i : lista) {
            switch (i.charAt(j - 1)) {
                case '3':
                    this.lista3.add(i);
                    break;
                case '4':
                    this.lista4.add(i);
                    break;
                case '5':
                    this.lista5.add(i);
                    break;
                case '6':
                    this.lista6.add(i);
                    break;
            }
        }
    }
}
```

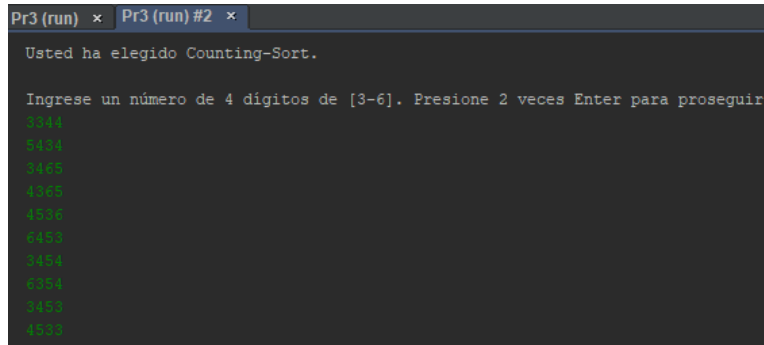
Figura 11: Estructura general del método **RadixSort**.

Finalmente, se limpia la lista original y se vacía progresivamente el contenido de las listas secundarias en la lista original. Posteriormente, las listas secundarias se limpian para volver a repetir el proceso hasta, finalmente, ordenar la colección de dígitos.

```
lista.clear();
lista.addAll(lista3);
lista.addAll(lista4);
lista.addAll(lista5);
lista.addAll(lista6);
lista3.clear();
```

Figura 12: Proceso de vaciado y limpieza.

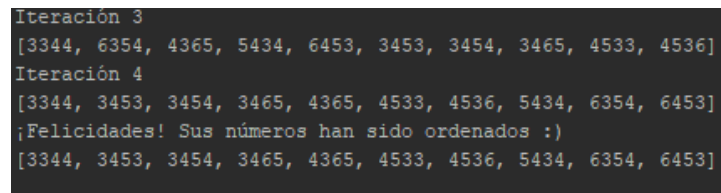
La implementación realizada permite introducir desde 1 solo número hasta 10 o más. Se especifica y se asume que el usuario introducirá correctamente los números de acuerdo a lo solicitado. Una vez introducidos los números, se imprimen las iteraciones consecutivas de ordenamiento hasta la impresión final de la lista de números ordenada.



```
Pr3 (run) x Pr3 (run) #2 x
Usted ha elegido Counting-Sort.

Ingrese un número de 4 dígitos de [3-6]. Presione 2 veces Enter para proseguir.
3344
5434
3465
4365
4536
6453
3454
6354
3453
4533
```

Figura 13: Introducción de 10 dígitos al programa.



```
Iteración 3
[3344, 6354, 4365, 5434, 6453, 3453, 3454, 3465, 4533, 4536]
Iteración 4
[3344, 3453, 3454, 3465, 4365, 4533, 4536, 5434, 6354, 6453]
¡Felicidades! Sus números han sido ordenados :)
[3344, 3453, 3454, 3465, 4365, 4533, 4536, 5434, 6354, 6453]
```

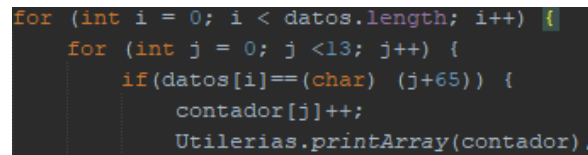
Figura 14: Lista de números ordenada.

3.3. Counting-Sort

La clase *CountingSort* consta de 3 métodos principales: **conteo**, **intermedio** y **ordenamiento**.

- El primer método recibe un arreglo de enteros y un arreglo secundario de conteo; el cual corresponde al primer arreglo utilizado para contar la frecuencia de aparición de cada elemento. Para ello, se implementó un ciclo for anidado que considera el tamaño del arreglo y el rango de letras a considerar [A-M: 13].

Posteriormente, se verifica si el arreglo introducido contiene una de las letras pertenecientes al rango haciendo uso indirecto del código ASCII de las letras [A-M], que tienen un respectivo valor de [65-77]. Con el fin de realizar el procedimiento adecuadamente, se agrega un *casting* para realizar adecuadamente la comparación con los caracteres presentes en el arreglo.



```
for (int i = 0; i < datos.length; i++) {
    for (int j = 0; j < 13; j++) {
        if(datos[i]==(char) (j+65)) {
            contador[j]++;
            Utilerias.printArray(contador);
        }
    }
}
```

Figura 15: Proceso de comparación y conteo.

- El segundo método recibe como parámetro el arreglo de conteo utilizado para observar la frecuencia de aparición de cada elemento. A continuación, se crea el segundo arreglo de suma acumulada y

se comienzan a sumar los elementos del arreglo de conteo para introducirlos consecutivamente al segundo arreglo.

```
for (int i = 0; i < arreglo.length; i++) {  
  
    conteo=arreglo[i]+conteo;  
    arregloAux[i]=conteo;  
}
```

Figura 16: *Proceso de creación del segundo arreglo requerido por Counting-Sort.*

- El último método recibe como primer parámetro el arreglo de suma acumulada y como segundo el arreglo de frecuencia. A través de un ciclo for anidado que toma en cuenta el tamaño del arreglo final y el valor de la suma acumulada en los índices, se procede a ubicar en su respectivo lugar cada uno de los elementos; recordando que el arreglo de suma acumulada es el que delimita la posición dónde será ubicado cada carácter. La variable *elemento* es utilizada para considerar los caracteres repetidos dentro del arreglo original.

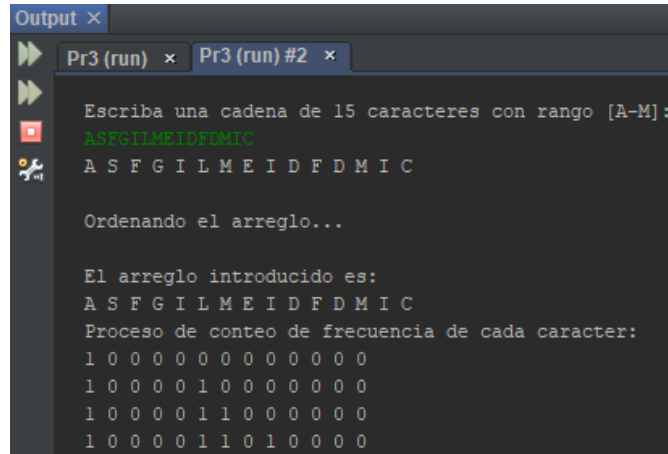
```
for (int j = arregloAux[i]-1; j > elemento; j--) {  
  
    arregloOrdenado[j]=(char) (i+65);  
    Utilerias.printArrayC(arregloOrdenado);  
}  
elemento=arregloAux[i]-1;
```

Figura 17: *Creación final del arreglo ordenado.*

La solución encontrada para la lectura de los caracteres consistió en solicitar una cadena de 15 caracteres dentro del rango especificado. A continuación, la cadena es transformada a un arreglo de caracteres haciendo uso de uno de los métodos de la clase *String*.

```
System.out.println("Escriba una cadena de 15 caracteres con rango [A-M]: ");  
String letra = entrada.nextLine();  
if (letra.isEmpty()) {  
    break;  
}  
arreglo = letra.toCharArray();  
Utilerias.printArrayC(arreglo);
```

Figura 18: *Lectura y conversión de una cadena a un arreglo de caracteres.*



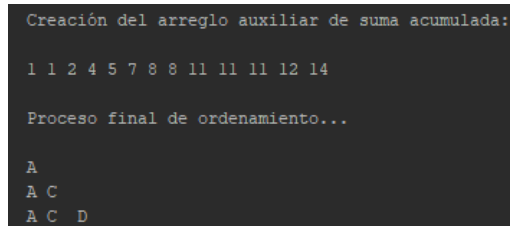
```

Output x
Pr3 (run) x Pr3 (run) #2 x
Escriba una cadena de 15 caracteres con rango [A-M]:
ASFGILMEIDFDMIC
A S F G I L M E I D F D M I C

Ordenando el arreglo...

El arreglo introducido es:
A S F G I L M E I D F D M I C
Proceso de conteo de frecuencia de cada caracter:
1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 1 1 0 0 0 0 0 0
1 0 0 0 0 1 1 0 1 0 0 0 0

```

Figura 19: *Proceso inicial de ordenamiento.*


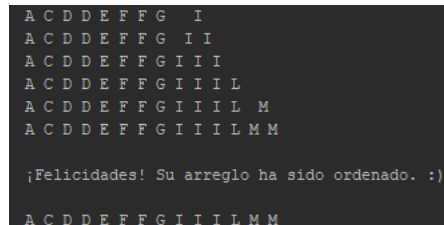
```

Creación del arreglo auxiliar de suma acumulada:
1 1 2 4 5 7 8 8 11 11 11 12 14

Proceso final de ordenamiento...

A
A C
A C D

```

Figura 20: *Proceso intermedio de Counting-Sort.*


```

A C D D E F F G I I I L M M
A C D D E F F G I I I L M M
A C D D E F F G I I I L M M
A C D D E F F G I I I L M M
A C D D E F F G I I I L M M
A C D D E F F G I I I L M M

¡Felicidades! Su arreglo ha sido ordenado. :)

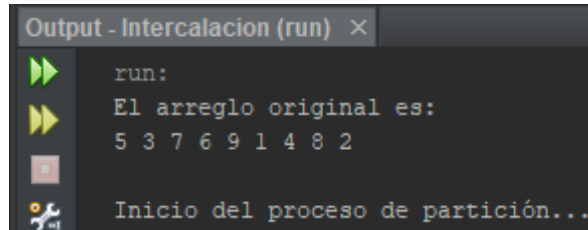
A C D D E F F G I I I L M M

```

Figura 21: *Impresión del arreglo de caracteres ordenado.*

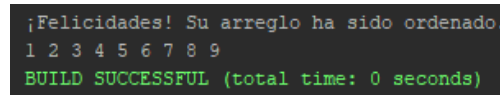
4. Merge-Sort en Java

Con el fin de realizar un análisis del algoritmo *Merge-Sort*, se ha recibido una implementación escrita en Java. Para poner a prueba de forma básica el algoritmo, se ha creado un arreglo pre-definido y se ha instanciado a la clase *MergeSort* con el fin de probar el funcionamiento adecuado de la implementación.



```
Output - Intercalacion (run) X
run:
El arreglo original es:
5 3 7 6 9 1 4 8 2
Inicio del proceso de partición...
```

Figura 22: Arreglo de enteros predefinido como entrada.



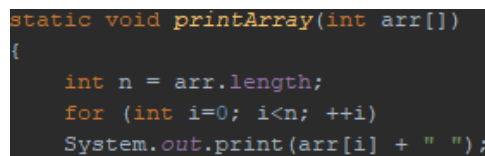
```
¡Felicidades! Su arreglo ha sido ordenado.
1 2 3 4 5 6 7 8 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 23: Salida del arreglo ordenado.

La clase *MergeSort* contiene tres métodos en su interior:

- El método **printArray** se encarga únicamente de recibir un arreglo de enteros e imprimirlo en pantalla por medio de un ciclo *for*. Como se ha visto anteriormente, el arreglo posee como atributo su tamaño, por lo que se elimina la necesidad de tener un segundo parámetro en el método.

Por otro lado, el identificador *static* que posee elimina la necesidad de instanciar a *MergeSort* para utilizar el método desde otra clase perteneciente al mismo paquete (considerado que se utiliza el identificador de acceso *friendly* de forma implícita).



```
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
}
```

Figura 24: Estructura de *printArray*.

- El método **sort** es la llamada inicial para el algoritmo de ordenamiento. El caso base de Merge-Sort sucede cuando $l \geq r$, es decir, se tiene un sub arreglo que contiene menos de 2 elementos. Por tanto, la condición de división y combinación es $l < r$. Posteriormente, se calcula el punto medio que será utilizado para dividir el arreglo en 2 partes. Finalmente, esos arreglos son unidos recursivamente.

```
if (l < r) {  
    int m = (l + r) / 2;  
    /*A continuación se divide el arreglo  
    en dos subarreglos y se divide recursivamente  
    cada uno de ellos*/  
    sort(arr, l, m);  
    sort(arr, m + 1, r);  
    /*Finalmente, se une el arreglo  
    dividido en dos*/  
    merge(arr, l, m, r);  
}
```

Figura 25: Estructura de *sort*.

- El objetivo de **merge** es realizar la mezcla progresiva de los arreglos que recibe conforme se ejecuta el programa. Previo al proceso, se calcula el tamaño de los sub arreglos temporales que serán creados para almacenar temporalmente el contenido presente en el arreglo original.

```
int n1 = m - l + 1;  
int n2 = r - m;  
  
//Creación de los arreglos temporales  
int L[] = new int[n1];  
int R[] = new int[n2];  
  
/*Los dos ciclos for a continuación  
copian los elementos del arreglo  
original en los dos subarreglos  
temporales*/  
for (int i = 0; i < n1; ++i) {  
    L[i] = arr[l + i];  
}  
  
for (int j = 0; j < n2; ++j) {  
    R[j] = arr[m + 1 + j];  
}
```

Figura 26: Llenado de los arreglos temporales.

Una vez realizado lo anterior, se inicia el proceso de mezcla. En primer lugar, se establecen los índices i, j iterativos para los sub arreglos. Posteriormente, se establece la variable iterativa k para indicar el primer índice del arreglo mezclado. Finalmente, se comienzan a comparar los elementos presentes en los arreglos temporales; el menor será acomodado en la posición k .

```
while (i < n1 && j < n2) {  
    if (L[i] <= R[j]) {  
        arr[k] = L[i];  
        i++;  
    } else {  
        arr[k] = R[j];  
        j++;  
    }  
    k++;  
}
```

Figura 27: Proceso de comparación y ordenamiento del arreglo.

El proceso finaliza copiando los elementos restantes de los arreglos secundarios, si es que quedan elementos por introducir al arreglo:

```
while (i < n1) {  
    arr[k] = L[i];  
    i++;  
    k++;  
}  
  
while (j < n2) {  
    arr[k] = R[j];  
    j++;  
    k++;  
}
```

Figura 28: *Verificación final de elementos.*

Con el fin de satisfacer los requerimientos solicitados, se implementaron líneas de código dedicadas a la impresión de los sub-arreglos creados por *Merge-Sort* y su respectiva mezcla:

```
System.out.println("Arreglo izquierdo:");  
MergeSort.printArray(L);  
System.out.println("Arreglo derecho");  
MergeSort.printArray(R);  
  
System.out.println("Mezcla de sub-arreglos: ");  
MergeSort.printArray(arr, l, r+1);
```

Figura 29: *Líneas agregadas al código.*

```
Inicio del proceso de partición...  
Arreglo izquierdo:  
5  
Arreglo derecho  
3  
Mezcla de sub-arreglos:  
3 5  
Arreglo izquierdo:  
3 5  
Arreglo derecho  
7  
Mezcla de sub-arreglos:  
3 5 7
```

Figura 30: *Resultado de las modificaciones hechas.*

5. Conclusiones

Cada uno de los algoritmos de ordenamiento analizados poseen sus propias ventajas y desventajas. Así mismo, poseen escenarios dónde su rendimiento puede aventajar a otros algoritmos o, al contrario, presentar deficiencias.

A continuación se mostrará una recapitulación de cada algoritmo analizado en las 3 prácticas:

- *Insertion-Sort*, a pesar de poseer una cota polinomial de $O(n^2)$, posee un buen rendimiento en situaciones que no involucren una gran cantidad de datos.

Lo anterior también es aplicable a *Selection-Sort*, debido a que su comportamiento frente a los incrementos de la entrada es muy similar al primero. Por otra parte, en términos formales, *Insertion-Sort* reduce su complejidad a $O(n)$ en su mejor caso; quedando ligeramente por encima de *Selection-Sort* que permanece con la misma complejidad de $O(n^2)$ para todos sus casos.

- Uno de los competidores más destacados fue *Quick-Sort*. Haciendo gala de su nombre, este algoritmo demostró una velocidad impresionante para ordenar arreglos de una cantidad de elementos muy elevada. Tal es su velocidad que puede ponerse a la par de algoritmos tan eficaces como *Merge-Sort*, a pesar de tener un peor caso donde su complejidad se eleva a $O(n^2)$.
- *Heap-Sort* demostró tener un rendimiento relativamente equilibrado frente a otros algoritmos de ordenamiento. En comparación con otros algoritmos de complejidad $O(n * \log(n))$, *Heap-Sort* fue uno de los más lentos. A pesar de tal detalle, su velocidad supera a otros algoritmos de complejidad polinomial.
- Sin duda, el competidor con peor rendimiento fue *Bubble-Sort*. Este algoritmo demostró ser eficiente únicamente con entradas pequeñas; conforme se aumenta la entrada, su rendimiento empeora notablemente. Por tal motivo, una hipotética aplicación de *Bubble-Sort* en situaciones reales no es una opción viable.
- *Counting-Sort* y *Radix-Sort* representan opciones de complejidad lineal viables y de buen rendimiento. Sin embargo, un detalle importante a considerar reside en su uso de memoria adicional conforme se aumentan las entradas: si bien para colecciones pequeñas de elementos este factor es irrelevante, puede resultar en un aspecto importante a considerar para colecciones de datos de gran tamaño.
- El algoritmo que es capaz de rivalizar a *Quick-Sort* posee la complejidad computacional mas baja de todos los algoritmos analizados: *Merge-Sort*. Su rendimiento es excepcional en todos los casos y lleva el proceso de división y recursividad al extremo; resultando en un algoritmo muy eficiente y equilibrado en cuanto a gestión de recursos.

Las primeras tres prácticas realizadas en el curso han resultado ser una experiencia muy retadora y satisfactoria, debido a que me ha obligado a esforzarme y cambiar mi flujo de trabajo con respecto a previos

cursos. En general, considero que esta dinámica me ha servido mucho para aprender a trabajar a presión y entregar resultados.

Pasé por notables dificultades iniciales debido a que mis bases de programación práctica en C se encontraban un tanto *oxidadas*. Sin embargo, me esforcé en obtener rápidamente las bases necesarias para realizar el trabajo solicitado: acción que dió los frutos esperados. Además, este esfuerzo resultó ser muy positivo ya que la introducción al trabajo en Java fue mucho más fácil, e incluso pude aplicar lo aprendido en la presente práctica realizado las implementaciones de *Counting-Sort* y *Radix-Sort* directamente en este lenguaje orientado a objetos.

El análisis de complejidad fue una de las partes más interesantes de las prácticas, ya que se pudo comprobar experimentalmente el rendimiento de los algoritmos con base en el número de operaciones realizadas. Esta actividad, combinada con el análisis de recurrencia y complejidad que estudié en EDA I, me ha hecho comprender de forma más clara algoritmos como *Merge-Sort*, el cuál analicé por primera vez en mi curso anterior.

Finalmente, considero que la experiencia obtenida a lo largo de estas tres prácticas será de enorme utilidad a lo largo de todo el curso. Los tres puntos principales a seguir mejorando son lo siguientes:

1. Trabajar rápidamente y bajo constante presión del tiempo.
2. Entregar resultados funcionales e implementados de la mejor forma posible.
3. Adaptarme a entornos de trabajo diferentes a los que me encuentro acostumbrado.

Sin duda alguna, el trabajo que he realizado tendrá un impacto futuro en mi futuro académico y profesional como Ingeniero en Computación. Por el momento, lo que sigue es continuar trabajando y entregando los mejores resultados posibles: el mundo necesita gente capaz de hacer las cosas adecuadamente para cambiar el rumbo de nuestra sociedad actual.

Referencias

- [1] Merge Sort in Java. Recuperado de: <https://www.baeldung.com/java-merge-sort>. Fecha de consulta: 22/02/2020.
- [2] Recurrencias. Recuperado de: <https://www.slideshare.net/RoadManuelGarciaRami/recurrencias>. Fecha de consulta: 22/02/2020.
- [3] Vista general del ordenamiento por mezcla. Recuperado de: <https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort>. Fecha de consulta: 22/02/2020.



Los créditos de las fotografías pertenecen a sus respectivos autores. ©