



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I Edgar Tista García

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 9

No de Práctica(s): 3

Integrante(s): Díaz Hernández Marcos Bryan

*No. de Equipo de
cómputo empleado:* Equipo personal

No. de Lista o Brigada: 9

Semestre: 2021-1

Fecha de entrega: 18 de octubre de 2020

Observaciones:

CALIFICACIÓN: _____

Objetivo de la practica

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

Introducción

En el siguiente reporte se analizaran los problemas realizados la practica con la intención de que sea explicada la forma en que estos funcionan y como se resolvieron los algoritmos para cada caso partículas que se trabajó, donde se implementa tanto en C como en Java.

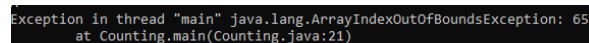
Ejercicios de la practica:

- Ejercicio 1:

El primer ejercicio, consistió en la elaboración algoritmo Counting-Sort, para esto fue necesario el implementar arreglos en java y utilizar el lenguaje Java, primero se crea un arreglo de 15 elementos, posteriormente se crea un segundo arreglo donde se va a almacenar el conteo de cada elemento del primer arreglo y al final de acuerdo a la suma de los valores de este arreglo se determinan las posiciones de los valores, la última operación corresponde a devolver los valores a su arreglo final de acuerdo a la posición que se indica en el segundo arreglo.

- Dificultades en el código

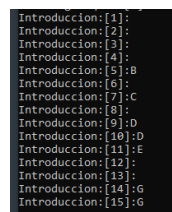
El primer problema que me encontré fue que al ingresar los valores de regreso, entraba en posiciones que no estaban definidas, entonces lo que hice fue tomar lápiz y papel, en ese análisis me di cuenta de que el arreglo contador contempla posiciones en un rango de $[0,15]$, lo que en los arreglos de longitud 15 es de $[0,14]$, entonces era necesario el colocar una posición menos para que pudiera acceder a las posiciones que correspondieran (Imagen 1).



```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 65
at Counting.main(Counting.java:21)
```

Imagen 1

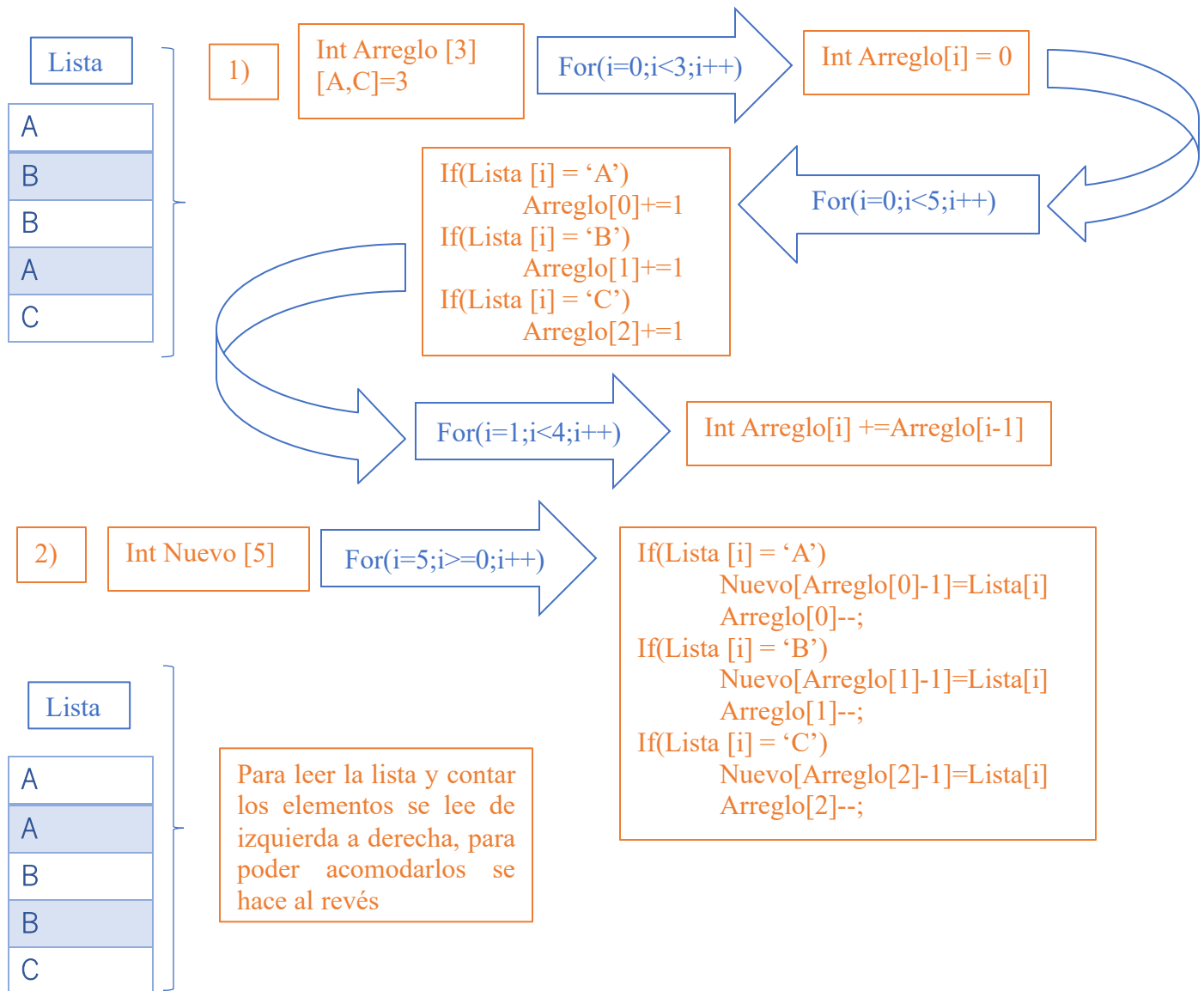
Este no fue un error pero si esta raro, porque es la impresión de cada vez que se regresa un valor, de esta forma se tiene que los valores, que se van ordenando dentro del arreglo original pero no los muestra, no entiendo porque, ya que si la impresión la coloco fuera del ciclo que retorna los valores si los imprime completos, de cualquier modo esto no afecta en la operación del algoritmo (Imagen 2).



```
Introduccion:[1]:
Introduccion:[2]:
Introduccion:[3]:
Introduccion:[4]:
Introduccion:[5]:B
Introduccion:[6]:
Introduccion:[7]:C
Introduccion:[8]:
Introduccion:[9]:D
Introduccion:[10]:D
Introduccion:[11]:E
Introduccion:[12]:
Introduccion:[13]:
Introduccion:[14]:G
Introduccion:[15]:G
```

Imagen 2

- Diagrama de funcionamiento:



- Relación con la teoría / Análisis :

Con respecto a la teoría encontré bastantes topes porque sabía lo que hacía el algoritmo y porque, pero no sabía llevarlo a la forma general, o encontrar un proceso que hiciera todo, sin la necesidad de comprobar valor, por valor lo que me llevo a sentirme mal con respecto a la habilidad de poder hacer eficiente un algoritmo ya que la implementación que hice no resuelve todos los posibles casos, únicamente un determinado, y creo que eso cumple con el objetivo de la práctica, pero me hubiese gustado el implementar algo general.

La operación que creo es la más importante o la modular del ejercicio es la que se muestra en la imagen y esto porque permite contabilizar los valores del arreglo y al final poder determinar la posición final de cada elemento, dentro de lo que se me ocurrió debería tener un if, para cada caso posible, y de esta forma poder contar todos los valores que aparecieran (Imagen 3).

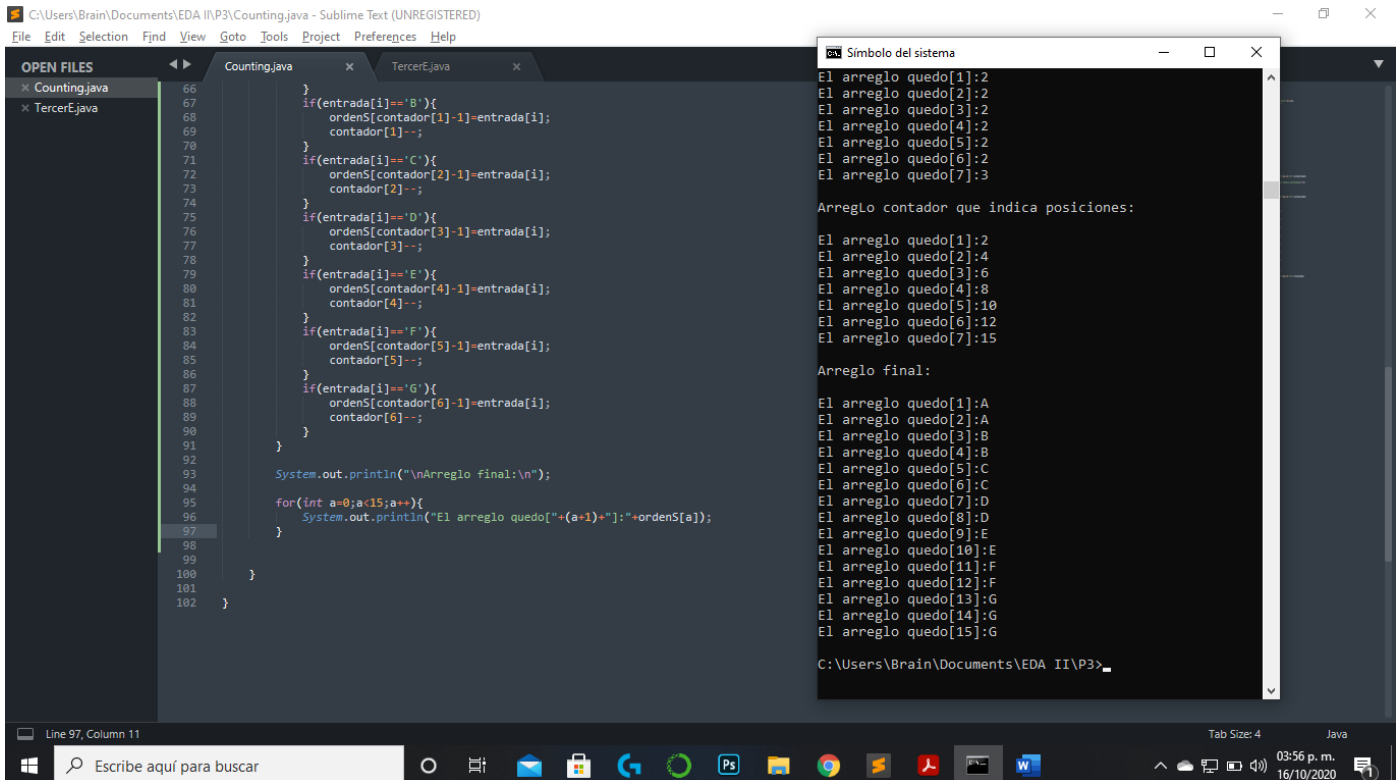
```

for(int i=0;i<15;i++){
    if(entrada[i]=='A'){
        contador[0]+=1;
    }
    if(entrada[i]=='B'){
        contador[1]+=1;
    }
    if(entrada[i]=='C'){
        contador[2]+=1;
    }
}

```

Imagen 3

- Evidencia de implementación



- Ejercicio 2:

En este ejercicio se pedía implementar el algoritmo Radix-Sort, con condiciones determinadas para poder resolver un caso particular, este caso estaba determinado por valores que contenían 4 dígitos, en un rango de [0,3], un arreglo de 10 elementos, y la creación de colas para cada dígito del rango, y poder implementar las operaciones características del algoritmo.

- Dificultades en el código

La mayor dificultad que pude tener fue el querer implementar el algoritmo en Java, esto porque para poder realizar el ejercicio se tienen que usar colas, y resulta que no se hacen colas en Java, no hay mucha diferencia, pero no comencé definiendo las colas en Java, sino que las implemente como en C, y como métodos de la clase, lo que llevo a que en un determinado momento de la programación, no se pudiera hacer la segunda iteración del algoritmo, lo que me llevo a pensar en regresar a C y poder resolverlo desde ahí, ya que tenía las operaciones fundamentales del algoritmo dentro de Java, solo era cuestión de adaptarlas a C y complementar lo que había realizado (Imagen 1).

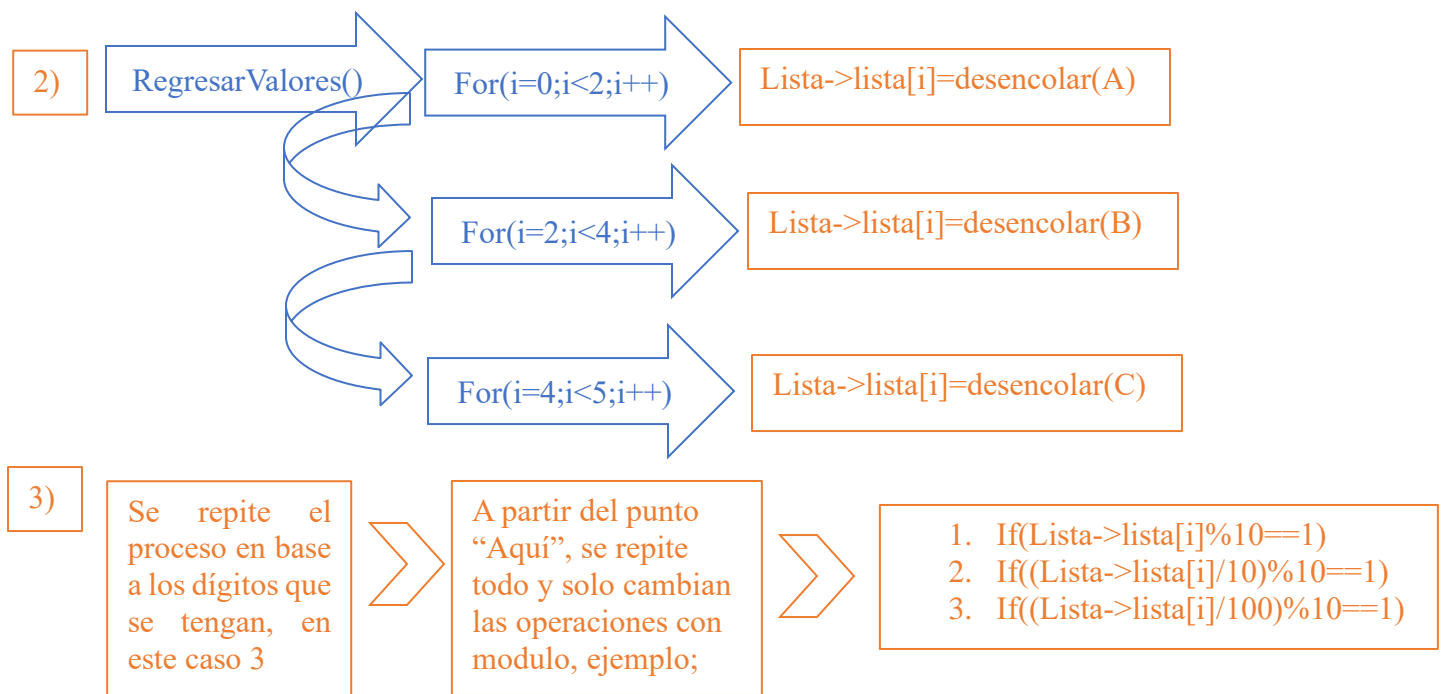
Imagen 1

```
int Cantidad(Cola *a){ //C
    int i,z=0;
    for (i=0;i<10;i++){
        if(a->lista[i]!=0){
            z+=1;
        }
    }
    return z;
}
```

Imagen 2

-
- ```
graph TD
 Lista[Lista: 123, 231, 312, 122, 131]
 Step1[1) Cola A=crearCola()
Cola B=crearCola()
Cola C=crearCola()]
 Step2[For(i=0;i<10;i++)
If(Lista->lista[i]%10==1)
 Encolar(A, Lista->lista[i])
If(Lista->lista[i]%10==2)
 Encolar(B, Lista->lista[i])
If(Lista->lista[i]%10==3)
 Encolar(C, Lista->lista[i])]
 Step3[Cola A={231,131}=2
Cola B={312,122}=2
Cola C={123}=1]
 Step4[RegresarValores()]
 Step5[A.lista[i]=0
B.lista[i]=0
C.lista[i]=0]
 Step6[For(i=0;i<10;i++)]
 Step7[Se cuentan los términos de cada lista y se envían al regresar valores]

 Lista --> Step1
 Step1 --> Step2
 Step2 --> Step3
 Step3 --> Step4
 Step4 --> Step5
 Step5 --> Step6
 Step6 --> Step7
```



- Relación con la teoría / Análisis:

En cuanto a lo que tenía que saber para poder realizar el algoritmo, conocía como funcionaba pero no encontraba la forma en que se tendría que determinar el dígito, esto fue porque en clase estaba muy sencillo verlo y decidir cual dígito se tendría que mandar, pero decirle a algo que no te puede escuchar el cómo lo tiene que seleccionar y dónde mandarlo fue lo complejo, no lo coloqué en las dificultades porque al final no genero errores, sino que tuve que pensar en cómo decirle al programa que tendría que escoger, de aquí probé con las divisiones y el módulo para la primera iteración, lo cual funcionó porque el módulo permitía evaluar el valor que resultase como residuo, lo que me llevó a probar las divisiones entre potencias de 10 para reducir el tamaño de los valores, y debido a que son enteros, las decimales no se consideraban en la división y el módulo permitía evaluar el dígito, esto no hubiese funcionado si se implementara otro tipo de dato.

Por lo anterior la operación más importante es la de determinar el valor del dígito y posteriormente enviar el valor a la lista correspondiente, es un ejercicio bastante curioso por el hecho de que se tiene que considerar el tipo de dato, además que los algoritmos responden a la cantidad de datos que se quieren ordenar y al tipo de datos (Imagen 1).

```

void EnviarColas(Cola *a, Cola *b, Cola *c, Cola *d, Cola *orden){
 int i;
 for(i=0;i<10;i++){
 if((orden->lista[i])%10==0){ //Estas son las operaciones básicas
 encolar(a,orden->lista[i]); //se puede deducir el dígito, y
 } //La potencia sea el número de l
 if((orden->lista[i])%10==1){ //al final se hace el módulo par
 encolar(b,orden->lista[i]);
 }
 if((orden->lista[i])%10==2){
 encolar(c,orden->lista[i]);
 }
 if((orden->lista[i])%10==3){
 encolar(d,orden->lista[i]);
 }
 }
}

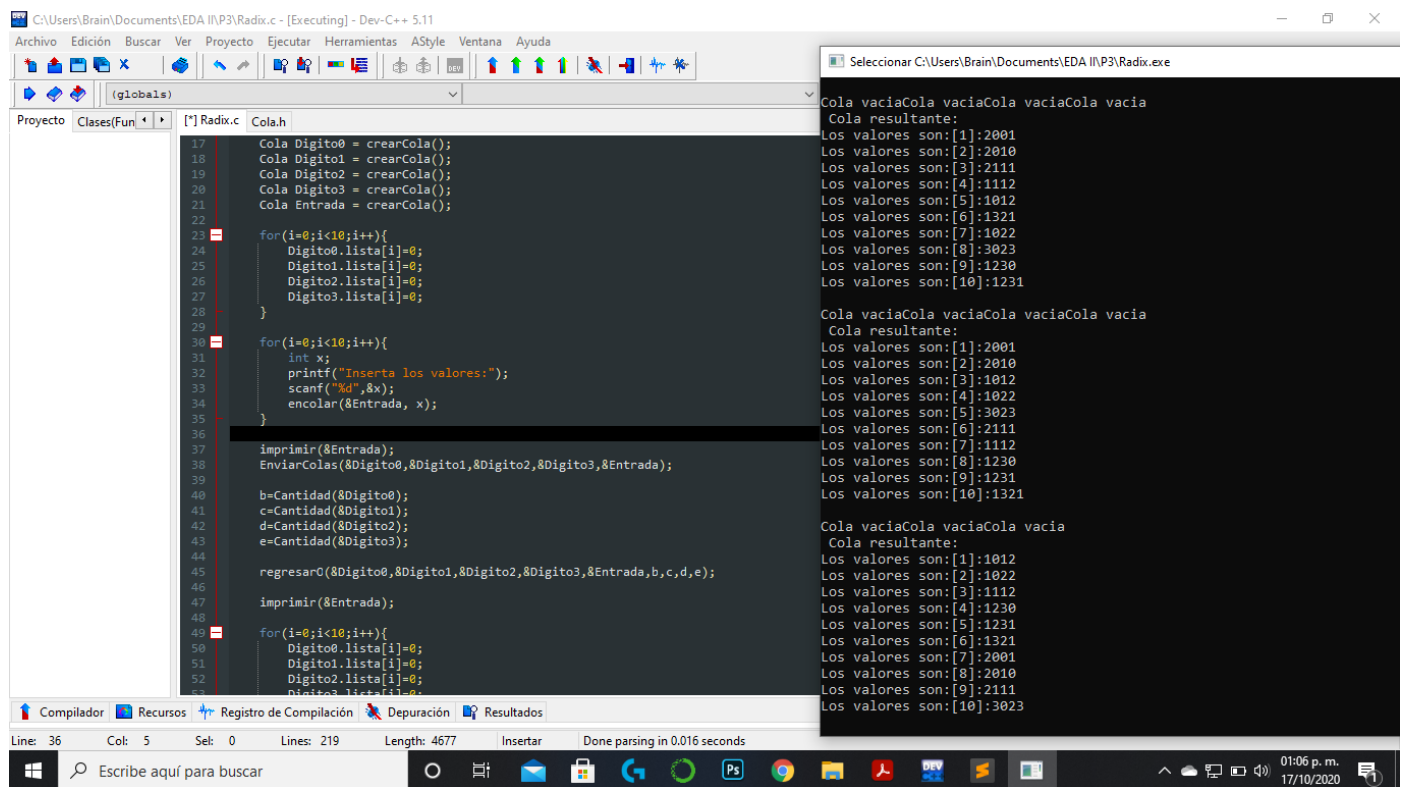
```

Imagen 1

Otro punto relevante del algoritmo es el conteo de los valores de cada elemento en las sublistas, esto significa que se tiene que evaluar el numero de elementos por lista para saber cuantos se van a agregar al arreglo o regresar al arreglo original. Esto se puede realizar por medio del CountingSort, y en específico con la operación que determina la cantidad de valores, pero no me permitía realizarla así que hice un método similar que me permitiera hacer lo mismo, tal y como lo puse en el punto de dificultades.

Como prueba del intento se envía el algoritmo realizado en Java, aunque este no funcione contiene las operaciones que se implementaron dentro del algoritmo en C.

- Evidencia de implementación



The screenshot shows a C++ IDE with a project named 'Radix.c'. The code in the editor implements a radix sort algorithm. It defines a 'Cola' structure with a 'lista' array and a 'cantidad' array. The main function initializes four 'Cola' objects (Digito0, Digito1, Digito2, Digito3) and an input 'Entrada'. It then processes the input by distributing its elements into the digit-based lists, counting the number of elements in each list, and shifting the digits to the right. The final output is printed as a sequence of numbers.

```
Cola Digito0 = crearCola();
Cola Digito1 = crearCola();
Cola Digito2 = crearCola();
Cola Digito3 = crearCola();
Cola Entrada = crearCola();

for(i=0;i<10;i++){
 Digito0.lista[i]=0;
 Digito1.lista[i]=0;
 Digito2.lista[i]=0;
 Digito3.lista[i]=0;
}

for(i=0;i<10;i++){
 int x;
 printf("Inserta los valores:");
 scanf("%d",&x);
 encolar(&Entrada, x);
}

imprimir(&Entrada);
EnviarColas(&Digito0,&Digito1,&Digito2,&Digito3,&Entrada);

b=Cantidad(&Digito0);
c=Cantidad(&Digito1);
d=Cantidad(&Digito2);
e=Cantidad(&Digito3);

regresarC(&Digito0,&Digito1,&Digito2,&Digito3,&Entrada,b,c,d,e);

imprimir(&Entrada);

for(i=0;i<10;i++){
 Digito0.lista[i]=0;
 Digito1.lista[i]=0;
 Digito2.lista[i]=0;
 Digito3.lista[i]=0;
}
```

The output window shows the execution results for three different input sequences:

```
Cola vaciaCola vaciaCola vaciaCola vacia
Cola resultante:
Los valores son:[1]:2001
Los valores son:[2]:2010
Los valores son:[3]:2111
Los valores son:[4]:1112
Los valores son:[5]:1012
Los valores son:[6]:1321
Los valores son:[7]:1022
Los valores son:[8]:3023
Los valores son:[9]:1230
Los valores son:[10]:1231

Cola vaciaCola vaciaCola vaciaCola vacia
Cola resultante:
Los valores son:[1]:2001
Los valores son:[2]:2010
Los valores son:[3]:1012
Los valores son:[4]:1022
Los valores son:[5]:3023
Los valores son:[6]:2111
Los valores son:[7]:1112
Los valores son:[8]:1230
Los valores son:[9]:1231
Los valores son:[10]:1321

Cola vaciaCola vaciaCola vacia
Cola resultante:
Los valores son:[1]:1012
Los valores son:[2]:1022
Los valores son:[3]:1112
Los valores son:[4]:1230
Los valores son:[5]:1231
Los valores son:[6]:1321
Los valores son:[7]:2001
Los valores son:[8]:2010
Los valores son:[9]:2111
Los valores son:[10]:3023
```

- Ejercicio 3:

En este ejercicio se requería el análisis del programa, y la implementación de este para un arreglo determinado, ya que los métodos estaban en otra clase era necesario crear una instancia e invocar cada método para ordenar e imprimir los arreglos, de esta manera se tiene el análisis, además de añadir las impresiones necesarias que demostrasen la división del arreglo original en dos arreglos, y así sucesivamente.

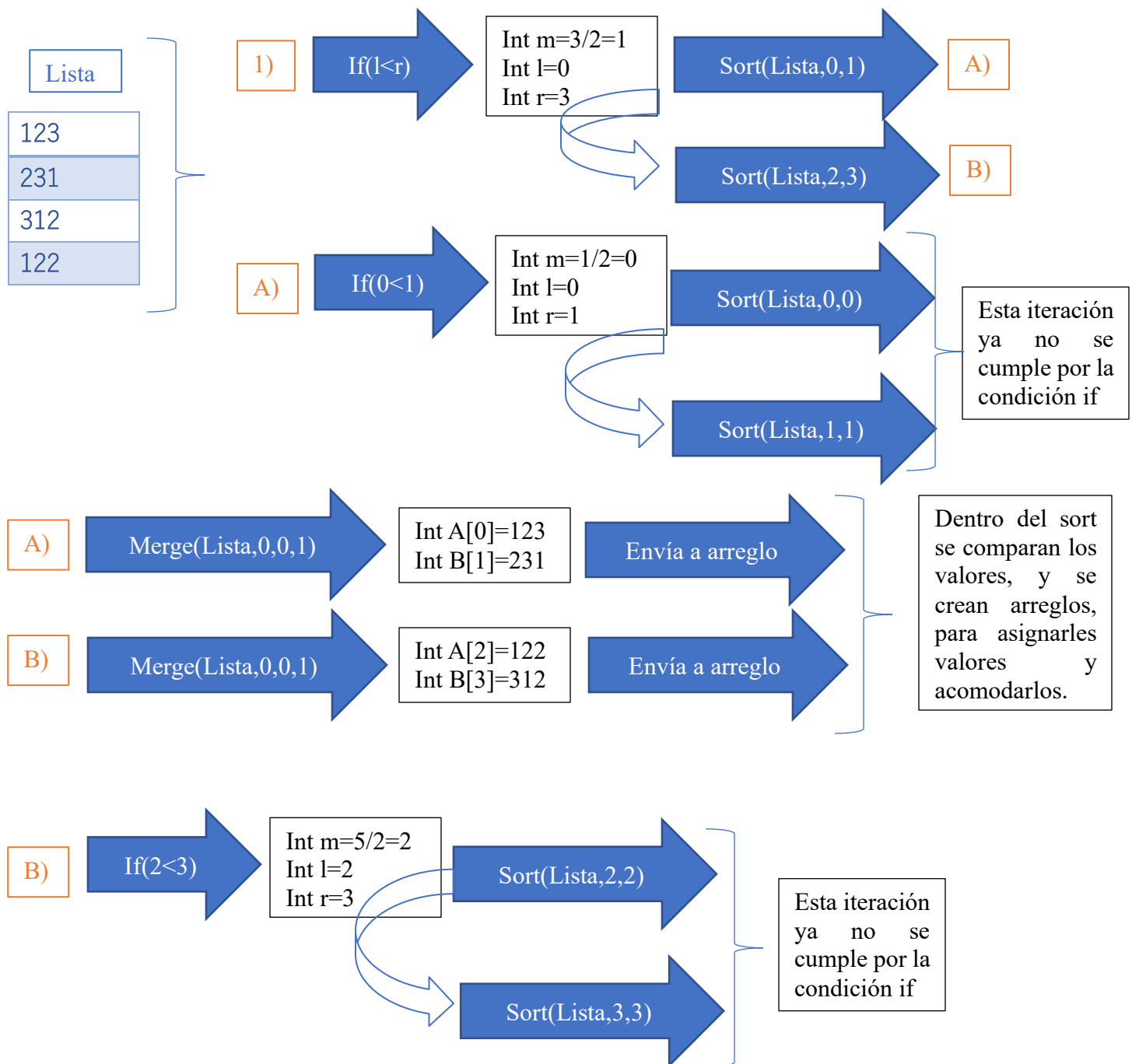
- Dificultades en el código

En este momento llego la felicidad ya que no tuve dificultades en el código, lo que si tuve que hacer fue el analizar el algoritmo y una vez hecho esto, el resultado fue la posición de las impresiones y cómo funcionaba el algoritmo (Imagen 1).

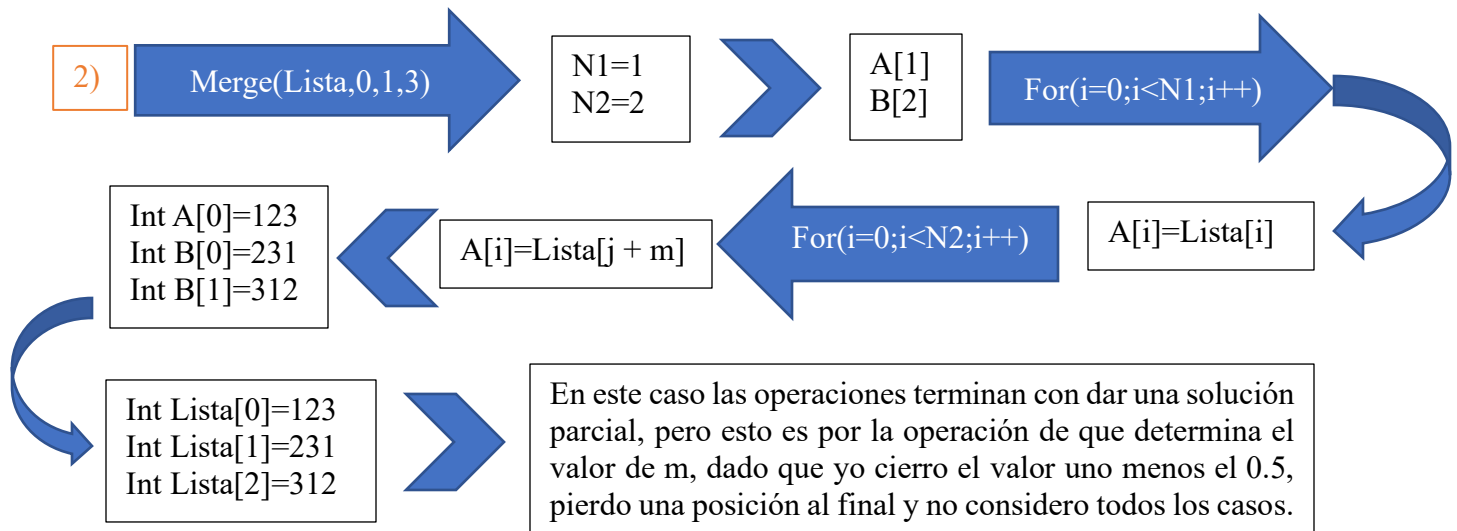
```
System.out.println("Division");
printArray(L, 0);
System.out.println("Division");
printArray(R, 0);
```

Imagen 1

- Diagrama de funcionamiento:







- Relación con la teoría / Análisis:

En cuanto al funcionamiento del algoritmo se tiene que los arreglos que entran el algoritmo de sort, se comienzan a dividir en dos y después cada uno de esos arreglos de vuelve a dividir en dos, de esta manera se tienen que los valores son individuales, pero se tienen la referencia del siguiente, no como en las listas, sino que se sabe la posición del anterior y se pueden comparar, ya que se usan arreglos.

```

int k = 1;
while (i < n1 && j < n2)
{
 if (L[i] <= R[j])
 {
 arr[k] = L[i];
 i++;
 }
}

```

Imagen 1

De forma tal que los valores se ordenan de dos en dos, y posteriormente se vuelven a unir, haciendo el proceso inverso, esto significa que los valores que se cambiaron de posición mandan los valores al arreglo original que va acomodando los valores de las sublistas ya que estas se utilizan como parámetro de orden y si es necesario el hacer el cambio este se ve reflejado en el arreglo original. Con esta forma se tiene que los valores se van ordenando de acuerdo con los sub-arreglos, y al final todo se regresa el arreglo inicial.

```

while (i < n1) { //
 arr[k] = L[i]; //
 i++;
 k++;
}

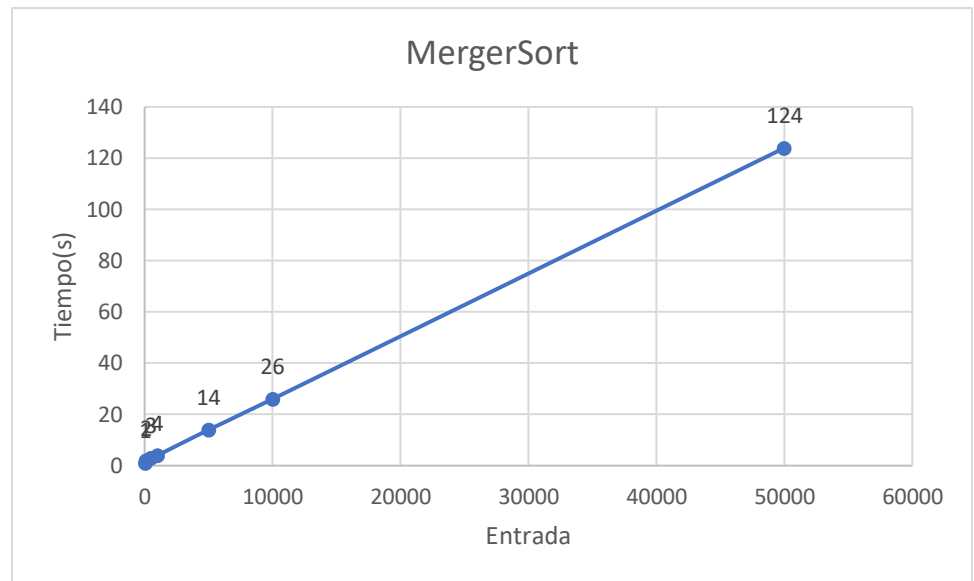
while (j < n2) {
 arr[k] = R[j];
 j++;
 k++;
}

```

Imagen 2

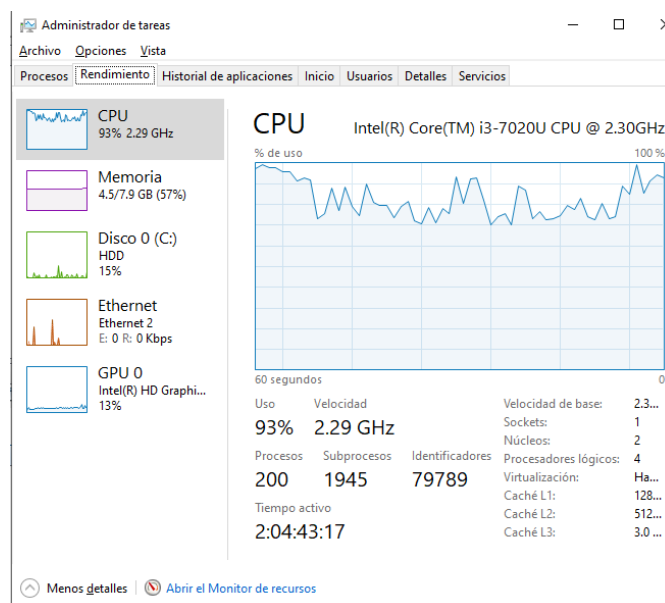
- Análisis de complejidad

| Entrada | Tiempo |
|---------|--------|
| 50      | 1      |
| 100     | 2      |
| 500     | 3      |
| 1000    | 4      |
| 5000    | 14     |
| 10000   | 26     |
| 50000   | 124    |

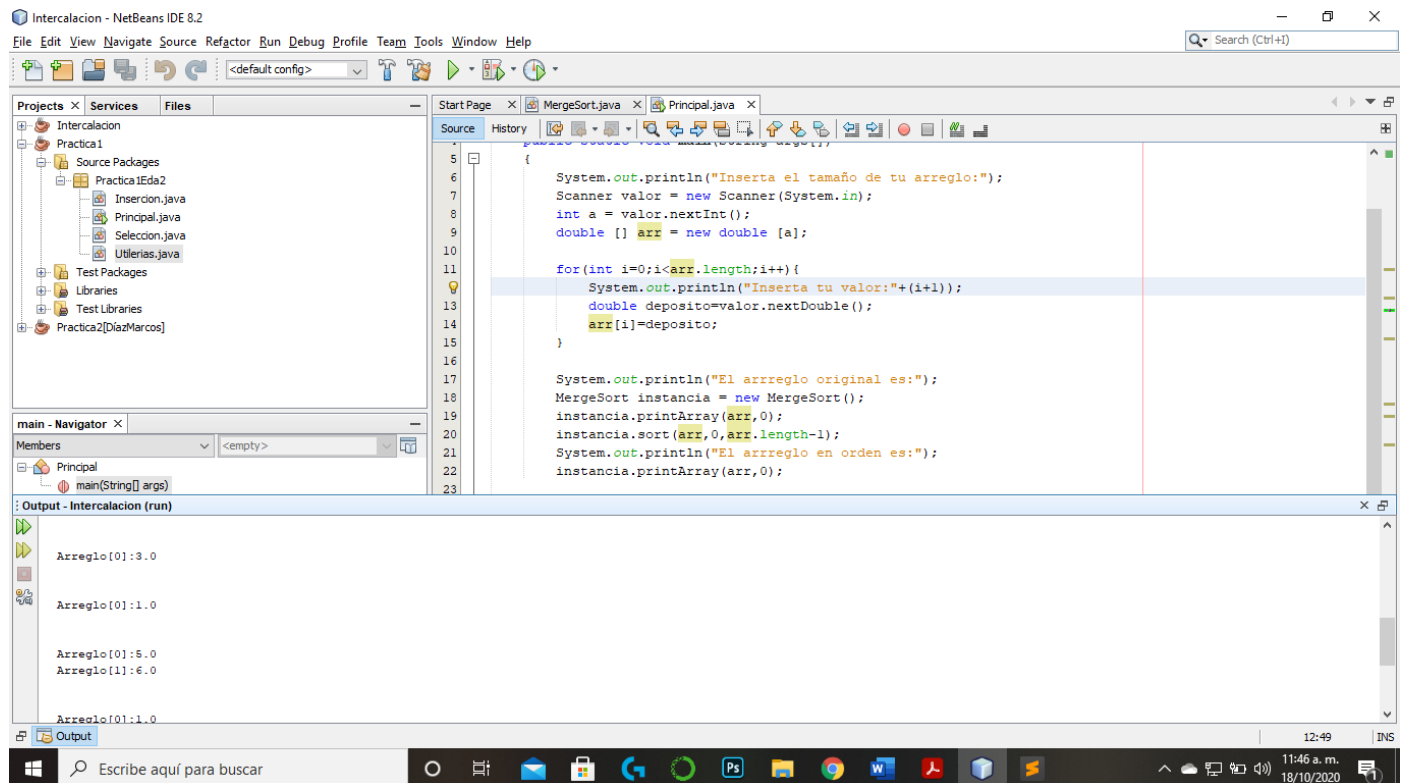


Como es visible, mientras mas grande sea la entrada, mas tiempo toma en realizar las operaciones, de tal forma que el crecimiento en las operaciones no es tan proporcional al tiempo que utilizan, esto es porque las operaciones mantienen una independencia de los valores, además que como el algoritmo divide en dos el arreglo se tiene que los valores se vuelven mas sencillos de trabajar, pero el consumo en memoria es creciente, ya que con las 50000 entradas el procesador se alentó un poco, y tuvo picos del 93% de uso el CPU, además que la memoria de 56% llego al 59%, no es mucho pero consume memoria.

Por lo que la complejidad estaría entre  $O(n \log n)$ , o con una tendencia lineal, eso igual depende de las condiciones del equipo, ya que después de un tiempo de uso el procesador decae un poco e igual las aplicaciones abiertas pueden demandar.



- Evidencia de implementación



## Conclusiones

En este reporte se puede dar como conclusión, que los algoritmos vistos, Counting, Radix y Merge, son muy eficientes con respecto al tipo de dato que manejan, además de poder trabajar con grandes cantidades en su entrada, sin embargo tienen un coste en la memoria que lo convierte en un arma de doble filo, ya que para optimizar un elemento como memoria o tiempo, se tiene que sacrificar uno de los dos, y beneficiar al otro, de igual manera el conocer la estructura general y las operaciones fundamentales de cada algoritmo es algo primordial si se quiere hacer la implementación en x lenguaje, porque de esta forma es posible el saber lo que tiene que hacer el algoritmo y como.

Por último el análisis de cada algoritmo demuestra su complejidad y como funcionan las funciones de cada algoritmo entre sí, esto significa que es posible diferenciarlos y descubrir cuando es conveniente usarlos.