Evaluación asíncrona 1 de Estrategias Algorítmicas.

Fecha de entrega: lunes, 21 de septiembre de 2021 (23:55).

Integrantes:

- Valle Garcia Miguel Angel - is726507
- Lobato Martinez Lilia Loabto - ie706937
- Santana Hernandez Claudia Alejandra - is726396

Estudia el siguiente método de ordenamiento llamado **CocktailSort**, el cual representa una pequeña mejora del popular algoritmo BubbleSort.

```
void cocktailSort (int[] array) {
     int left = 1, right = array.length - 1;
     int last = right;
     while(left <= right) {
          for(int r = right; r >= left; r --) {
               if(array[r - 1] > array[r]) {
                    swap(array, r - 1, r);
                    last = r;
               }
          }
          left = last + 1;
          for(int l = left; l <= right; l ++) {
               if(array[l - 1] > array[l]) {
                    swap(array, l - 1, l);
                    last = l;
               }
          }
          right = last - 1;
     }
}

void swap(int[] array, int i1, int i2) {
     int tmp = array[i1];
     array[i1] = array[i2];
     array[i2] = tmp;
}
```

- c1: costo de una inicialización de variables, o de una comparación, movimiento o cualquier operación aritmética entre índices y conteos.
- c2: costo de un movimiento entre datos.
- c3: costo de una comparación entre datos.

Instruction	**Costo**	**Mejor caso**	**Peor caso**
int left = 1	C1	1	1
right = array.length - 1	C1	1	1
int last = right	C1	1	1
while	--	--	--
For	--	--	--
array[r - 1] > array[r]	C3	N-1	Realiza una sumatoria n-1+(n-3)+….+5+3+1
swap(array, r - 1, r)	C2	0	(n-1+(n-3)+….+5+3+1)(3)
last = r	C1	0	n-1+(n-3)+….+5+3+1
left = last + 1	C1	N-1	N
For	--	--	--
array[l - 1] > array[l]	C3	0	Realiza una sumatoria n-2+(n-4)+….+4+2
swap(array, l - 1, l)	C2	0	3(n-2+(n-4)+….+4+2)
last = l	C1	0	n-2+(n-4)+….+4+2
right = last - 1	C1	N	N

n-1+n-2+…+5+4+3+2+1
$=\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$

$\left(\frac{n(n-1)}{2}\right)(3)$

1) Describe de forma completa, concisa y precisa cómo funciona este algoritmo, es decir, cómo logra ordenar un arreglo. Aquí es importante la correcta redacción, ortografía y puntuación.

CocktailSort es una mejora al BubbleSort. En el BubbleSort se recorre el arreglo de izquierda a derecha de forma que se mueve el elemento más grande a la posición final (hasta el último elemento de la izquierda). Hace este mismo recorrido hasta tener ordenado el arreglo.

En el CocktailSort recorremos el arreglo en ambas direcciones de manera alternada, es decir, de izquierda a derecha luego de derecha a izquierda y así hasta ordenar el arreglo.

- Cuando recorremos de izquierda a derecha, vamos comparando pares de datos y moviendo el mayor una posición a la izquierda de forma que al terminar, el valor de la izquierda será el mayor.
- Cuando recorremos de derecha a izquierda, vamos comparando pares de datos y moviendo el menor una posición a la derecha de forma que al terminar, el valor de la derecha será el menor.

Ejemplo: Usando el algoritmo CocktailSort, vamos a ordenar el arreglo [5,8,1,4,7,9,2]

Nota: Los números en el arreglo marcados con verde corresponden al pivote, los rojos son contra el que se compara.

```
[5, 8, 1, 4, 7, 2, 9] last = right = 6 (no swap)
[5, 8, 1, 4, 2, 7, 9] last = right-1 = 5
[5, 8, 1, 2, 4, 7, 9] last = right-2 = 4
[5, 8, 1, 2, 4, 7, 9] last = right-3 = 3
[5, 1, 8, 2, 4, 7, 9] last = right-4 = 2
[1, 5, 8, 2, 4, 7, 9] last = right-5 = 1
```

Se va recorriendo el array de derecha a izquierda comparando los pares y si el valor de la izquierda es mayor, se cambia de lugar.

Se recorre desde right = 6 a left = 1

```
[1, 5, 8, 2, 4, 7, 9] last = 1 (no swap)
[1, 5, 2, 8, 4, 7, 9] last = left + 1 = 3
[1, 5, 2, 4, 8, 7, 9] last = left + 2 = 4
[1, 5, 2, 4, 7, 8, 9] last = left + 3 = 5
[1, 5, 2, 4, 7, 8, 9] last = left + 4 = 5
```

Se recorre ahora de izquierda a derecha y si el valor de la izquierda es mayor, se cambia de lugar.

Se recorre desde left = last+1 = 2 a right = 6

```
[1, 5, 2, 4, 7, 8, 9] last = 5 (no swap)
[1, 5, 2, 4, 7, 8, 9] last = 5 (no swap)
[1, 5, 2, 4, 7, 8, 9] last = 5 (no swap)
[1, 2, 5, 4, 7, 8, 9] last = right-2 = 2
```

Se va recorriendo el array de derecha a izquierda.

Se recorre desde right = last-1 = 4 a left = 2

```
[1, 2, 4, 5, 7, 8, 9] last = left + 0 =3
[1, 2, 4, 5, 7, 8, 9] last = 3 (no swap)
[1, 2, 4, 5, 7, 8, 9] last = 3 (no swap)
```

Se recorre de izquierda a derecha.

Se recorre desde left = last+1=3 a right = 4

```
[1, 2, 4, 5, 7, 8, 9] last = 2(no swap)
[1, 2, 4, 5, 7, 8, 9] last = 2(no swap)
```

Termina el algoritmo porque left es mayor, ya recorrimos todo el arreglo.

right = 2 y left = 3

2) Identifica el mejor caso del algoritmo y, mediante un análisis **a priori**, calcule el tiempo de ejecución de ***CocktailSort*** para este caso considerando movimientos y comparaciones.

En la respuesta redacta una narrativa (clara, completa y concisa) de cómo se llegó a las ecuaciones finales siguiendo el comportamiento del algoritmo tomando en cuenta que estamos en el mejor caso.

El mejor caso sería cunado entregan un arreglo que ya está ordenado. Se toman en cuenta solo los costos de C3 definidos previamente en la tabla.

1.Se asignan los valores a las variables

```
int left = 1, right = array.length - 1;

int last = right;
```

Por lo que no s quedaria
Left=1
Rigth=n-1
Last=n-1

2.Como el while `left <= right` sera verdadero entrara

3.En el if del primer for se iran comprando pares de datos para verificar si el dato de la derecha es menor que el de la izquierda, recorriendo el arreglo de derecha a izquierda, **se comparan n-1 pares**.

4.Como no habrá ningún caso donde se tenga que el número de la derecha sea menor nunca entrara dentro del if por lo que last seguirá siendo n-1 como cuando lo asignamos al principio y **no se realizaría ningún movimiento** entre datos

5.Se asigna a left el valor que hay en last más 1 por lo que quedaría
Left=n

6. El siguiente for se tiene `for(int l = left; l <= right; l ++)`
Como a l se le asigna el valor de left l=n y como n<n-1 desde el principio se cumplira la condicion de paro por lo que no se ejecutaría lo que venga dentro del for

7. Al hacer la comparación entre variables para el while `left <= right` como left(n) no es menor ni igual a Righ(n-1) se cumpliría su condición de paro y acabará la ejecución del algoritmo

$$T_{Best,Cmp}(N) = n - 1 \rightarrow \theta(\text{n})$$

$$T_{Best,Mov}(N) = 0 \rightarrow \theta\ (1)$$

3) Identifica el peor caso del algoritmo y, mediante un análisis **a priori**, calcule el tiempo de ejecución de ***CocktailSort*** para este caso considerando movimientos y comparaciones:

En la respuesta redacta una narrativa (clara, completa y concisa) de cómo se llegó a las ecuaciones finales siguiendo el comportamiento del algoritmo tomando en cuenta que estamos en el peor caso.

1. En el peor caso lo que está dentro del while se ejecutara N cantidad veces
2. El if del primer for siempre será verdadero por lo que se correrá n-1+(n-3) +…+5+3+1 esto equivale a hacer
3. Como el if siempre será verdadero entrará y se ejecutará el código al interior
4. El costo de swap es de 3 movimientos y se ejecutara n-1+(n-3) +…+5+3+1 cantidad de veces
5. Como el if del primer for siempre se cumple left se le asigna el valor del último valor del for anterior más uno recorriendo el arreglo hacia la derecha bloqueando los valores que queden a la izquierda por lo que el if se ejecutara n-2+(n-4) +…+4+2 si juntamos esta suma con la suma del anterior if quedaría n-1+n-2+n-3+n-4…+5+4+3+2+1 por lo que **entre los dos if tendrán (N(N-1)) /2 comparaciones**
6. Como el if siempre será verdadero entrará y se ejecutará el código al interior
7. El costo de swap es de 3 movimientos y se ejecutara n-1+(n-3) +…+5+3+1 cantidad de veces puede juntarse de igual manera con la sumatoria del anterior swap teniendo la suma de n-1+n-2+n-3+n-4….+5+4+3+2+1 por los tres movimientos que se realizan por lo que **entre los dos swaps tendrán (3)(N(N-1))/2 movimientos**

$$T_{Worst,Cmp}(N) = \frac{n^2-n}{2} \rightarrow \theta(n^2)$$

$$T_{Worst,Mov}(N) = \left(\frac{N(N-1)}{2}\right)(3) \rightarrow \theta(n^2)$$

4) De acuerdo a las 4 ecuaciones obtenidas en los dos ejercicios anteriores, utiliza notación asintótica para caracterizar (si es posible, generalizar) el tiempo de ejecución de *CocktailSort* para cada uno de los casos siguientes:

$$T_{Worst}(N) = \Theta(n^2)$$

$$T_{Best}(N) = \Omega(1), O(n)$$

$$T_{Cmp}(N) = \Omega(n), O(n^2)$$

$$T_{Mov}(N) = \Omega(1), O(n^2)$$

$$T(N) = \Omega(1), O(n^2)$$

5) Conteste las siguientes preguntas con respecto al algoritmo ***CocktailSort***:
Justifique su respuesta. Puede hacer referencias a líneas de código o con un *contraejemplo*.

a. Al finalizar una iteración de *while,* ¿el subarreglo $A[0 \ldots left\text{-}1]$ siempre está ordenado?
Sí, en la iteración while el primer ciclo for va comparando los datos adyacentes y poniendo el de menor valor a la izquierda guardando la posición donde se hizo el cambio en last, al terminar el ciclo for el arreglo estará ordenado de la posición 0 a donde se realizó el ultimo cambio por lo cual el subarreglo [0….last] estará ordenado, al salir del for se asigna a left el valor de last+1, y vimos anteriormente que last marcaba hasta que posición el arreglo estaba ordenado y como left-1=last, left-1 siempre estará ordenado

b. Al finalizar una iteración de *while,* ¿el subarreglo $A[right+1 \ldots N]$ siempre está ordenado?
Sí, el segundo ciclo for tiene una funcionalidad similar al primero que vimos en la anterior pregunta solo que invertido por lo que va comparando los datos adyacentes desde donde quedo el left hacia la derecha, pero ahora verifica si el valor de la izquierda es mayor y de ser el caso intercambia, guardando la posición donde se hizo el cambio en last, al terminar el ciclo for el arreglo estará ordenado de la posición N a donde se realizó el ultimo cambio por lo cual el subarreglo [last…N] estará ordenado, al salir del for se asigna a rigth el valor de last-1, y vimos anteriormente que last marcaba hasta que posición el arreglo estaba ordenado y como rigth+1=last, rigth+1 siempre estará ordenado

c. Al finalizar una iteración de *while,* ¿el subarreglo $A[left \ldots right]$ siempre está ordenado?
No, como vimos en el inciso a y b se garantiza que el arreglo estará siempre ordenado de la posición 0 a left-1 y de la posición rigth+1 a la posición N dejando de la posición de left a la posición right los números que faltan de ser ordenados

d. Al finalizar una iteración de *while,* ¿el subarreglo $A[0 \ldots left\text{-}1]$ tiene los mismos elementos que el arreglo original (*loop-invariant*)?
No, un elemento en la posición n en el primer while podría terminar en la posición 0, esto se cumpliría si el elemento de la posición n es igual al elemento más pequeño del arreglo, por lo tanto, en ese caso y otros no cumpliría que será loop-invariant.
Al encontrar el mínimo mientras se hace el barrido de derecha a izquierda, el elemento se arrastra hasta la posición inicial correspondiente; esto hace que el arreglo cambie de elementos. Lo mismo sucede con el barrido de izquierda a derecha.

e. ¿El algoritmo es *estable*?
Para que sea estable el algoritmo debe de mantener el orden de los elementos que tienen el mismo valor entre sí.
Si es estable porque en las comparaciones, solamente compara contra elementos adyacentes y siempre revisamos que sea mayor, es decir, si encontramos elementos equivalentes, estos no se van a mover.
Estas secciones del código son las que aseguran que sea estable if(array[r - 1] > array[r]) y if(array[l - 1] > array[l]), ya que de esta manera se compara que sea mayor, no mayor e igual que, por lo tanto, dos elementos iguales no se intercambiaran.

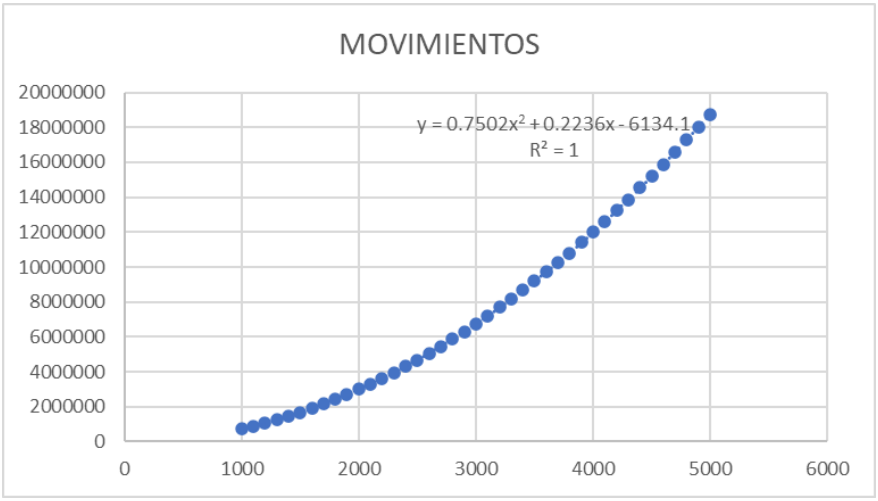
6) Mediante un análisis **a posteriori**, calcule el tiempo de ejecución de ***CocktailSort*** en el caso promedio de acuerdo con el número de movimientos y comparaciones. Siga la metodología vista en clase: $N \rightarrow$ 1000 a 5000, $N/100$ corridas por cada N.

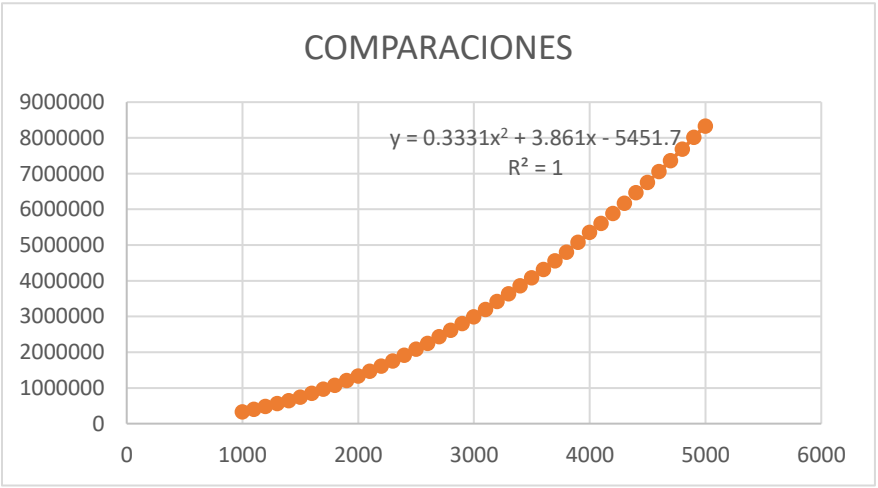
$$T_{Mov}(N) = \theta(n^2)$$

$$T_{Cmp}(N) = \theta(n^2)$$

N	COMPARACIONES	MOVIMIENTOS
1000	332854	744316
1100	403506	904600
1200	481329	1080513
1300	562854	1263636
1400	644647	1444776
1500	745933	1672915
1600	854770	1921158
1700	965977	2165993
1800	1080747	2424727
1900	1208829	2711729
2000	1339612	3004455
2100	1473160	3307492
2200	1618009	3632880
2300	1757244	3940966
2400	1918964	4317388
2500	2085855	4681438
2600	2253342	5062946
2700	2434371	5464779
2800	2620725	5886014
2900	2806394	6308429
3000	2994249	6735219
3100	3201678	7187732
3200	3426794	7700785
3300	3636392	8169150
3400	3863941	8671926
3500	4090465	9184302
3600	4322531	9709977
3700	4565129	10252715
3800	4804284	10790984
3900	5081906	11419739
4000	5354165	12024952
4100	5605581	12605328
4200	5891176	13237129
4300	6169584	13852363
4400	6466788	14530844
4500	6755672	15185190
4600	7056890	15876854
4700	7364519	16549031
4800	7687752	17285368
4900	8022322	18034167
5000	8331442	18729027

<<Gráfica de dispersión con la ecuación de tendencia para *movimiento* y *comparaciones*>>





7) El siguiente método elimina los valores negativos de una lista de números reales.

```
public static void removeNegatives(ArrayList<Double> list) {
  List<Double> tempList = new ArrayList<>();
  while(!list.isEmpty()) {
     double d = list.remove(0);
     if(d >= 0)
     tempList.add(d);
  }
  for(int i = 0; i < tempList.size(); i++) {
     double d = tempList.get(i);
     list.add(d);
  }
}
```

Instruction	**Costo**	**Mejor caso**	**Peor caso**
List<Double> tempList = new ArrayList<>()		1	1
while(!list.isEmpty())		--	--
double d = list.remove(0)	C2	(N)N	(N)N
d >= 0	C1	N	N
tempList.add(d)	C2	0	N
For(i < tempList.size())		--	--
double d = tempList.get(i)	C2	0	N
list.add(d)	C2	0	N

a) ¿Cuál es el tiempo de ejecución del algoritmo en términos del tamaño de la lista *N*? Incluye el desarrollo de la solución (tiempo de ejecución por línea de código) y notación asintótica.

$$T_{Best}(N) = c1n + c2(\text{n})\text{n} = \Theta(n^2)$$

$$T_{Worst}(N) = c1n + c2(\text{n})\text{n} + \text{c2}(4\text{n}) = \theta(n^2)$$

$$T(N) = \theta(n^2)$$

Incluye en la respuesta los tiempos de ejecución parciales: por línea de código y ciclo *for*.

b) Si ejecutas el método con una lista de más de 100 mil elementos, podrás notar que tarda varios segundos. Implementa una versión más rápida del método sin modificar el tipo del parámetro de entrada. Puedes seguir usando una lista temporal. Elige bien el tipo de lista que convenga utilizar, así como los métodos (y argumentos) que convengan invocar en cada momento para reducir la complejidad al mínimo a cada paso.

```
Implementación del algoritmo
public static void fastRemoveNegatives(ArrayList<Double> list) {
	List<Double> tempList = new ArrayList<>();
	for (Double e: list) {
		if(e>=0) {
			tempList.add(e);
		}
	}
	list.clear();
	for(int i = 0; i < tempList.size(); i++) {
		double d = tempList.get(i);
		list.add(d);
	}
}
```

c) Calcula el tiempo de ejecución de esta nueva versión. Incluye desarrollo de la solución y la notación asintótica más adecuada.

$$T_{Best}(N) = c1(n) + c2(1)$$
$$T_{Worst}(N) = c1(n) + c2(4n)$$

$$T(N) = \theta(n)$$

Instruction	**Costo**	**Mejor caso**	**Peor caso**
List<Double> tempList = new ArrayList<>()		--	--
for (Double e: list)	--	--	--
e>=0	C1	n	n
tempList.add(e)	C2	0	n
list.clear()	C2	1	1
for(i < tempList.size())		--	--
double d = tempList.get(i)	C2	0	n
list.add(d)	C2	0	n

Criterio de evaluación	Puntos
Ejercicio 1. ¿Cómo ordena ***CocktailSort***?	10
Ejercicio 2. Análisis a priori mejor caso: razonamiento y ecuación final.	15
Ejercicio 3. Análisis a priori peor caso: razonamiento y ecuación final.	15
Ejercicio 4. Notación asintótica.	10
Ejercicio 5. Cinco preguntas con respecto a ***CocktailSort***.	25
Ejercicio 6. Análisis a posteriori. Extracto de la salida de la ejecución del programa, 2 gráficas (pueden estar superpuestas) y 2 ecuaciones.	15
Ejercicio 7a. Tiempo de ejecución total y por línea de código.	10
Ejercicio 7b. Versión más rápida que funcione como se espera.	20
Ejercicio 7c. Tiempo de ejecución total y por línea de código.	10
TOTAL	**130**