Taller 5: Complejidad de los algoritmos propuestos.

Integrantes: Juan Andrés Arroyave y Gustavo López

1. Sort Insertion:

1.1. Código:

```
public static int [] SortInsertion (int [] n) {
    for(int i=0;i<n.length;i++) {
       int elMenor=i;
          for(int j=elMenor+1;j<n.length;j++) {
             try {
               TimeUnit.SECONDS.sleep(1);
             }
            catch(Exception e) {
            }
              if(n[j]<n[elMenor]) {
                elMenor=n[j];
              }
         }
         int aux=n[i];
         n[i]=n[elMenor];
         n[elMenor]=aux;
      }
      return n;
  }
```

1.2. Tamaño del problema:

N^2: Representa la cantidad de iteraciones que realiza el ciclo.

1.3. Cuanto demora cada Línea:

```
public static int [] SortInsertion (int [] n) {
    for(int i=0;i<n.length;i++) {
       int elMenor=i;
          for(int j=elMenor+1;j<n.length;j++) {
             try {
               TimeUnit.SECONDS.sleep(1);
             }
```

```
            catch(Exception e) {
            }
              if(n[j]<n[elMenor]) {
                elMenor=n[j];
              }
          }
          int aux=n[i];
          n[i]=n[elMenor];
          n[elMenor]=aux;
      }
      return n;
  }
```

1.4. Complejidad del algoritmo en Notación O:

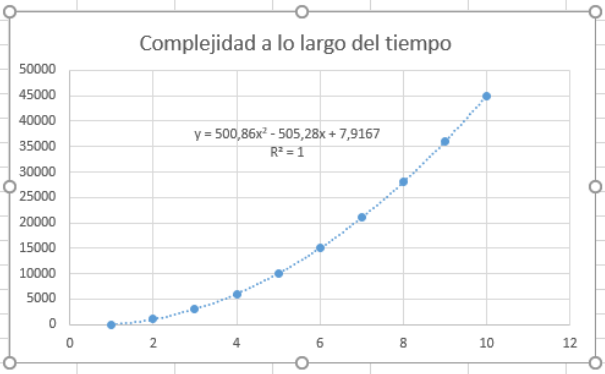
O(N^2)

1.5. Descripción de la complejidad

Para ordenar n elementos nuestro algoritmo requeriría de N^2 pasos. Por lo que su complejidad es de O(N^2).

1.6. Grafica:

1	0
2	1002
3	3002
4	6004
5	10004
6	15006
7	21012
8	28017
9	36027
10	45046



1.7. Conclusiones:

El crecimiento en el tiempo de ejecución de este programa es muy grande, para ordenar datos muy grandes no sería recomendable usar algoritmos

como estos, recomendaríamos usar algoritmos como QuickSort o Buckert Sort.

2. ArraySum:

2.1. Código del programa:

```
public static int suma(int[] a){
        int count = 0; // c_1
        for(int i = 0; i < a.length; i++) { // c2 + sum c3,
i=0 to n
            try {
                TimeUnit.SECONDS.sleep(90/100);
            } catch (Exception e) {
            }
            count += a[i];  //sum c4, i=0 to n-1
        }
        return count; //c5
    } //La complejidad de este algorítmo es O(n)
```

2.2. Tamaño del problema

El tamaño de este algoritmo es n (Suma n elementos)

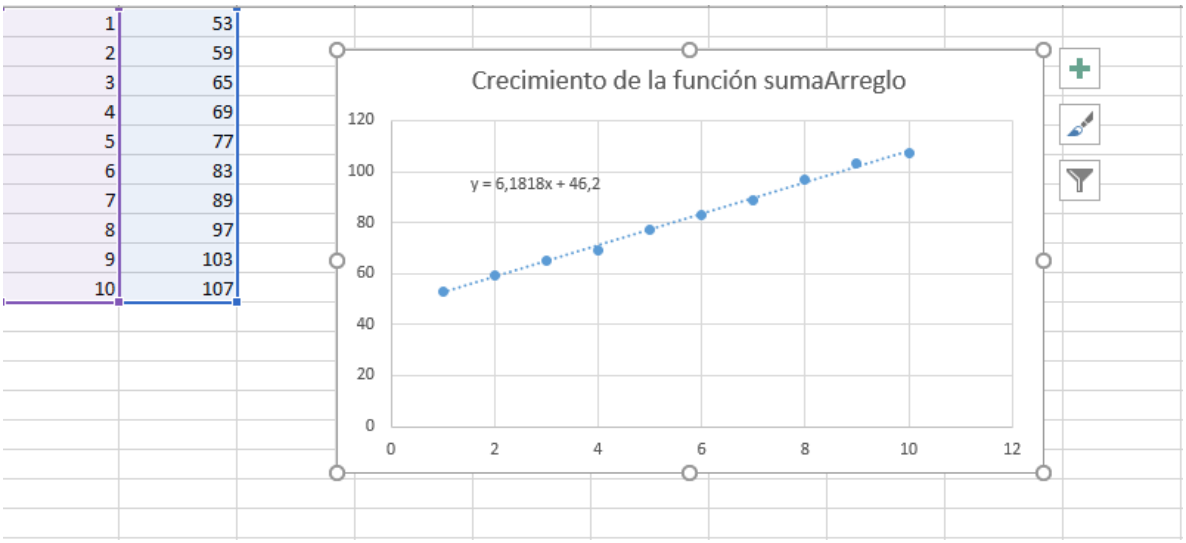
2.3. Complejidad del algoritmo:

La complejidad de este algoritmo es O(n)

2.4. Descripción de la complejidad del problema

Para sumar n elementos el algoritmo toma n pasos, el tiempo de ejecución es constante.

2.5. Gráfica del crecimiento del algoritmo



2.6. Conclusiones:

Este código es eficiente para el manejo de grande cantidad de datos, ya que el crecimiento del tiempo de ejecución es un crecimiento lineal.

3. Tablas Demultiplicar

3.1. Código:

```
private static void TablasDeMultiplicar (int n) {
    for(int i=1;i<=n;i++) { //N+1 iteraciones +n incrementos + constante
        for(int j=1;j<=n;j++) {// (N+1 iteraciones+ n incrementos)*n
            try {
              TimeUnit.MILLISECONDS.sleep(35);
            }
            catch(Exception e) {}
                    System.out.println( i + " X " + j +  " = " + (i*j));
        }
    }
```

}

3.2. Tamaño del Problema

El tamaño de este algoritmo es N^2(Cantidad de iteraciones realizadas por los ciclos).

3.3. Complejidad del Algoritmo

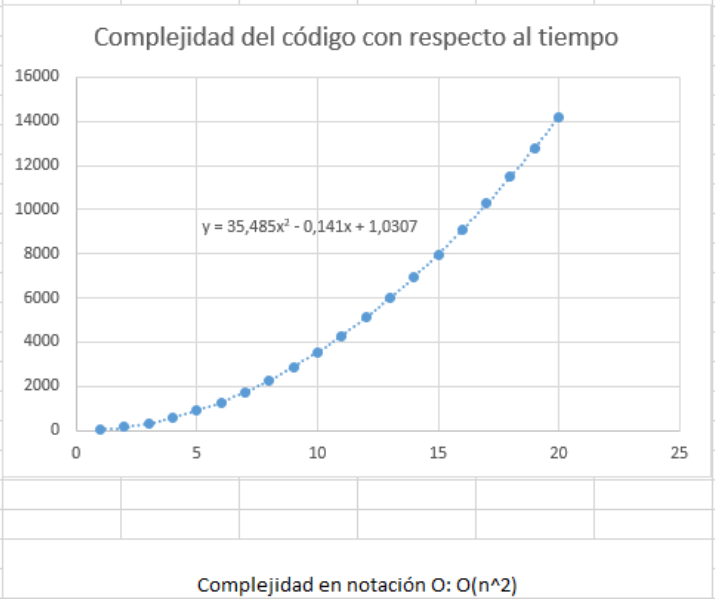
La complejidad del algoritmo es O(N^2).

3.4. Descripción de la complejidad del algoritmo

Este algoritmo requiere de N^2 pasos para poder realizar las tablas de multiplicar de una forma óptima.

3.5. Grafica del crecimiento del algoritmo

1	36
2	149
3	319
4	568
5	887
6	1273
7	1743
8	2268
9	2875
10	3538
11	4289
12	5117
13	5985
14	6954
15	7974
16	9116
17	10278
18	11481
19	12789
20	14194



3.6. Conclusiones:

Este algoritmo no es muy eficiente al momento de realizar gran cantidad de tablas de multiplicar ya que este posee un crecimiento cuadrático.