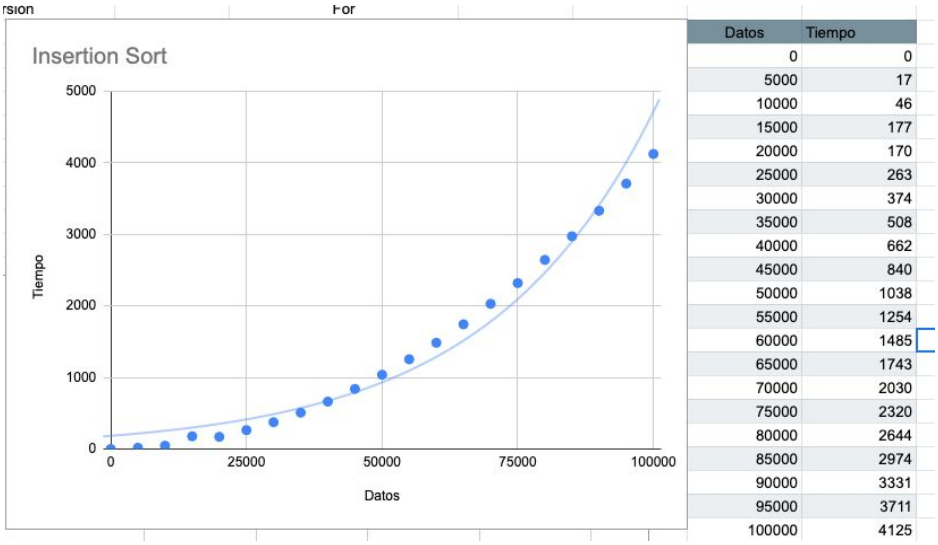


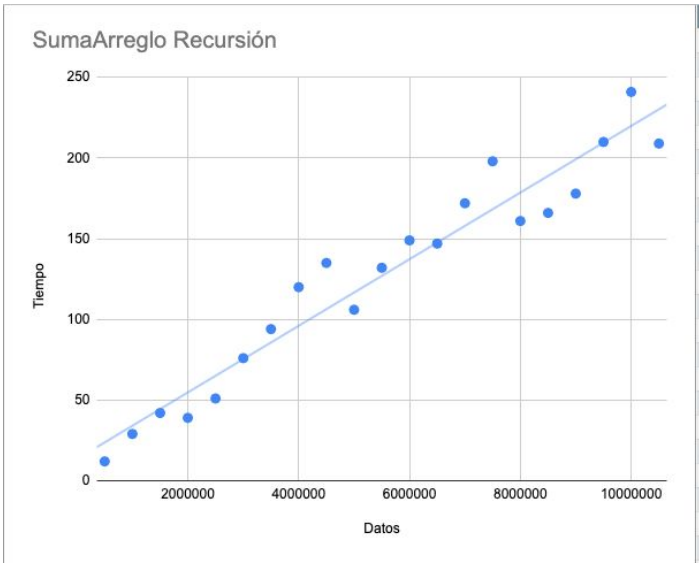
Datos	Tiempo
0	0
5000	17
10000	46
15000	177
20000	170
25000	263
30000	374
35000	508
40000	662
45000	840
50000	1038
55000	1254
60000	1485
65000	1743
70000	2030
75000	2320
80000	2644
85000	2974
90000	3331
95000	3711
100000	4125

```
15      public static int[] insertionSort(int[] arr) {
16          for (int i = 0; i < arr.length - 1; i++) {
17              int key = arr[i];
18              for (int j = i + 1; j > 0; j--) {
19                  if (arr[j] < arr[j - 1]) {
20                      int temp = arr[j];
21                      arr[j] = arr[j - 1];
22                      arr[j - 1] = temp;
23                  }
24              }
25
26          }
27          return arr;
28      }
29  }
```

n-1 * c2
c3
(n*n) * c4
c5
c6
c7
c8
F(n) = (n-1*c2) + c3+((n*n)*c4)+c5+c6+c7+c8
F(n) = (n-1) + (n*n)
F(n)=(n)+n^2
F(n) = n^2
O(n^2)

El algoritmo Insertion Sort no es apropiado para ordenar grandes volúmenes ya que tiene una complejidad cuadrática y si le metemos un millón de datos o mil millones, el tiempo se extenderá bastante.

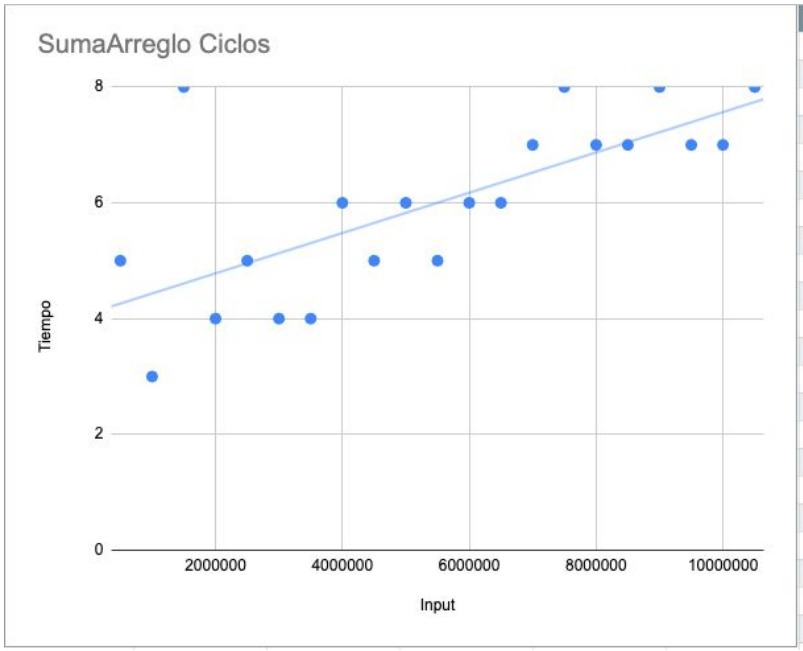
Punto 2



Recursión	
Datos	Tiempo
500000	12
1000000	29
1500000	42
2000000	39
2500000	51
3000000	76
3500000	94
4000000	120
4500000	135
5000000	106
5500000	132
6000000	149
6500000	147
7000000	172
7500000	198
8000000	161
8500000	166
9000000	178
9500000	210
10000000	241
10500000	209

```java
public static int cunetaRecuersiva(int[] nums,int index){
        if(index == nums.length-1){ // 4C
            return nums[index]; // 2C
        }
        return nums[index] + cunetaRecuersiva(nums,index+1); // 4C + F(n-1)
    }
        // F(n) = 4C + 2C + 4C + F(n-1)
        // F(n) = 4C + 2C + 4C + C_2*n + c_1
        // F(n) = C_2*n
        // F(n) = n
        O(n)
```

En términos del tiempo el algoritmo revulsivo que suma todos los elementos de un arreglo se comporta bien cuando el n es muy grande. Pero el algoritmo si necesita de demasiada memoria para poder hacer correr cuando el n es muy grande.



For	
Datos	Tiempo
500000	5
1000000	3
1500000	8
2000000	4
2500000	5
3000000	4
3500000	4
4000000	6
4500000	5
5000000	6
5500000	5
6000000	6
6500000	6
7000000	7
7500000	8
8000000	7
8500000	7
9000000	8
9500000	7
10000000	7
10500000	8

```
public static int cuentaTodo(int[] nums){

        int contador = 0; // C

        for(int tt = 0;tt< nums.length;tt++){ // C + 3C*(n-1)
            contador = contador + nums[tt]; // (n-1)(3C)
        }

        return contador; // C
    }

        F(n) = C + C + 3C*(n-1) + (n-1)(3C)
        F(n) = C + C + 3C*n - 3C + 3C*n - 3C
        F(n) = 3C*n + 3C*n
        F(n) = n + n
        F(n) = 2n
        F(n) = n

        O(n)
```

El algoritmo con loops que suma todos los elementos de un arreglo es bastante eficiente aun en el peor de los casos. Incluso cuando no es 100 millones el algoritmo se demora poco tiempo y no consume mucha memoria. En conclusión si tengo loops y recursión y debe contar la suma de los elementos de un arreglo, escogería los loops siempre.

Conclucion:
En la clase de lenguajes de programación estamos programando en Assembler. Gracias a esto entendí, que hacer un llamado recursivo necesita muchas más líneas en el lenguaje assembler para poder representar un evento repetitivo que lo que necesita un loop. Esto significa que las diferencias en tiempos de ejecución, entre la recursión y los loops, vienen de la forma en que funcionan los lenguajes de programación.

Pero en cuestión de memoria recuerdas sí demostró ser un dolor de cabeza. Entonces me quedo con loops.