TALLER #5

Carlos Gustavo Vélez Manco y John Esteban Castro Ramírez

1. Insertion Sort

- **Complejidad asintótica para el peor de los casos:**

T(n)=C1*n+n*C2+C3*n(n+1)/2+C4*n(n+1)/2+C5*n(n+1)/2+C6*n(n+1)/2+C7*n(n+1)/2
T(n)=(n(n+1)/2*)(C3+C4+C5+C6)+C' (Regla de las sumas y producto)
T(n)=(n^2+n)*1/2*C' (Se quita la suma interna)
T(n)=n^2+n (Se quitan los productos y la suma interna)
T(n)=O(n^2)

- **Gráficas:**

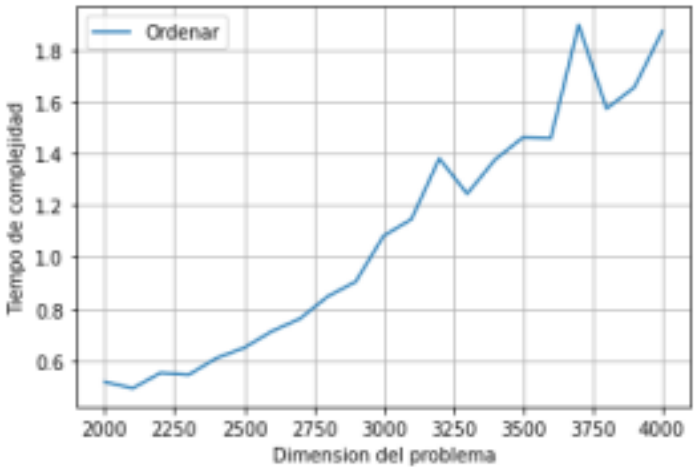

Figura 1. Gráfica (experimental) del algoritmo 'Insertion Sort' con 20 tamaños de problema diferentes en Python

Tamaño del problema(n) [En este caso la longitud del	Tiempo de complejidad T(n)
2000	0.517613173
2100	0.49319768
2200	0.552578688
2300	0.544637442
2400	0.609321833
2500	0.649451733
2600	0.713601351
2700	0.761494875
2800	0.847904444
2900	0.904559374
3000	1.081242561
3100	1.145543814
3200	1.380022287
3300	1.243818283
3400	1.375633955
3500	1.46132493
3600	1.458250523
3700	1.895799398
3800	1.572344542
3900	1.65513587
4000	1.871318579

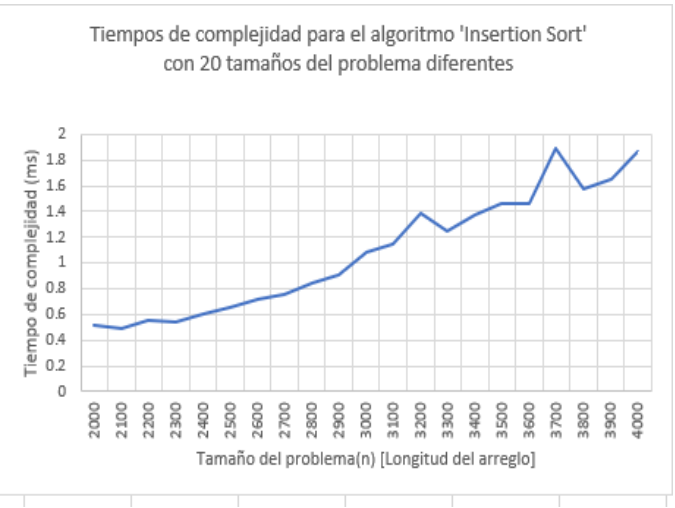

Figura 2. Gráfica (experimental) del algoritmo 'Insertion Sort' con sus longitudes (n) y tiempo de complejidad para 20 tamaños del problema diferentes en Excel.

- **Análisis:** Aunque en la gráfica generada no se ve del todo claro el comportamiento cuadrático O(n^2) esto se debe a que la dimensión del problema, que en este caso es la longitud del arreglo, no representa el peor de los casos; pero podemos ver que, a partir de cierto punto, la gráfica de n^2 está por encima de la graficada. Hay cierta similitud, y se puede ver que a medida que aumenta la dimensión del problema, aumentará el tiempo de complejidad para el algoritmo.

- **¿Este algoritmo es apropiado para ordenar grandes volúmenes de datos?:** No, este algoritmo no es apropiado para ordenar grandes volúmenes de datos ya que como crece con una tendencia cuadrática, al aumentar la dimensión del problema este tomará tiempos muy grandes para ejecutarse.

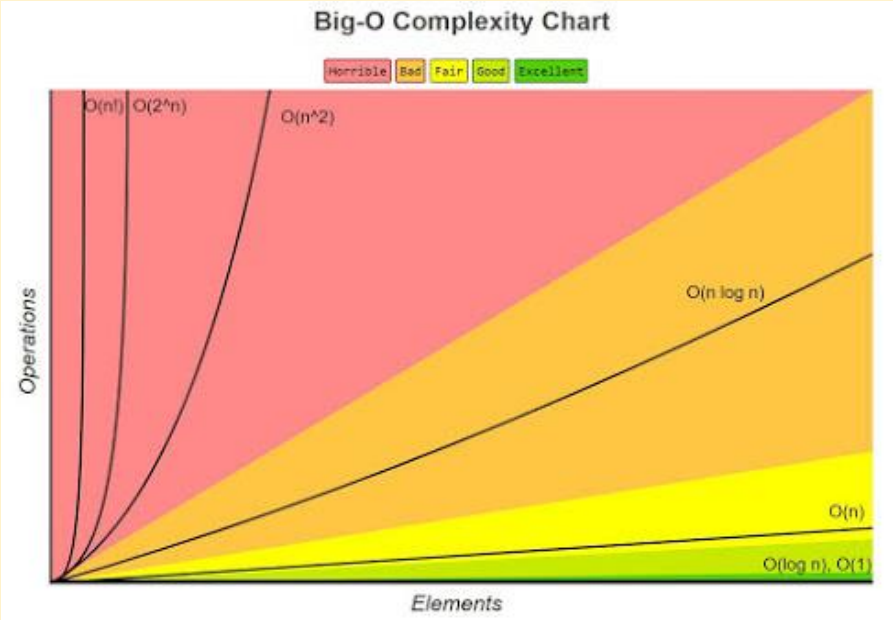

Figura 3. Notación BigO

Podemos ver en la gráfica anterior que O(n^2) se encuentra en el rango de horrible, por lo que no funcionará bien para grandes volúmenes de datos.

2. ArraySum

- **Complejidad asintótica para el peor de los casos:**
  T(n)=C1+C2*n+C3*n+C4
  T(n)=n*(C2+C3)+C' –(Regla de las sumas)
  T(n)=n*C' –(Se quitan los productos)
  T(n)=O(n)

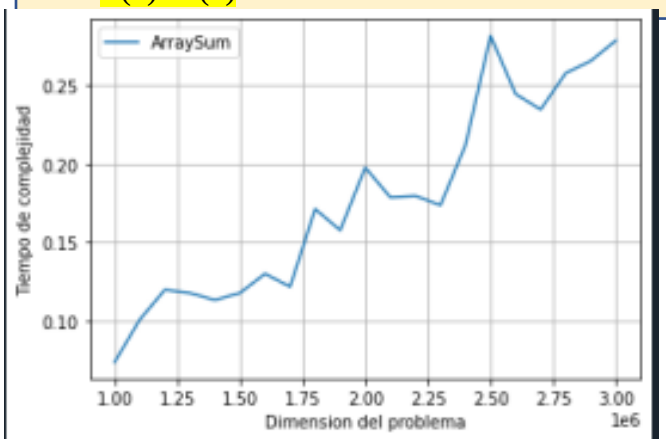

Figura 4. Gráfica (experimental) del algoritmo 'ArraySum' con 20 tamaños de problema diferentes en Python.

Tamaño del problema (n) [En este caso la longitud del arreglo]	Tiempo de complejidad(T(n))
1000000	0.083278418
1100000	0.110213995
1200000	0.094318151
1300000	0.116163015
1400000	0.111736536
1500000	0.154599667
1600000	0.133598089
1700000	0.148109913
1800000	0.17852211
1900000	0.152636528
2000000	0.201558352
2100000	0.180508137
2200000	0.188529491
2300000	0.206016302
2400000	0.221915722
2500000	0.269281149
2600000	0.231835127
2700000	0.283311605
2800000	0.249519348
2900000	0.281291723
3000000	0.254285097

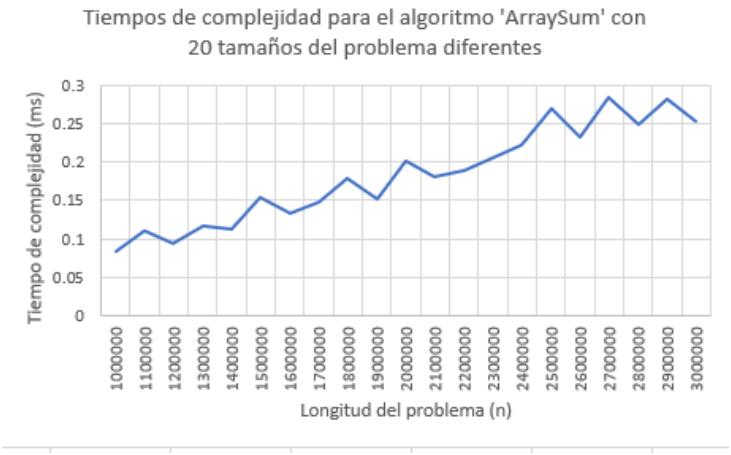

Figura 5. Gráfica (experimental) del algoritmo 'ArraySum' con sus longitudes (n) y tiempo de complejidad para 20 tamaños del problema diferentes en Excel.

- **Análisis:** Se puede ver que, aunque la gráfica tiene ciertos picos, debido al "ruido", sigue un patrón lineal O(n) que es precisamente lo que nos dio la complejidad teórica; aunque al igual que en el punto anterior no se está tomando el peor de los casos por lo que no se ve del todo claro. Esta complejidad es medianamente buena para todo tipo de datos ya que no toma mucho tiempo de ejecución.

- **¿Existe una diferencia significativa, en el tiempo de ejecución, entre la implementación con recursión y la implementación con ciclos? ¿Por qué?** En la carpeta de Github se encuentra este algoritmo con ambas opciones (ciclos y recursión), al realizar el calculo de complejidad con recursión para el peor de los casos da como resultado $T(n)=C+T(n-1)$ que al resolverla en WolframAlpha da como resultado $T(n)=Cn+C1$ que es lo mismo que $O(Cn+C1)$ y al aplicarle regla de la suma queda $O(Cn)$ y luego con regla del producto queda $O(n)$ donde n es el número de elementos del arreglo. Por lo que no hay una diferencia significativa entre ambas implementaciones.

3. Ejercicio Opcional

- **Complejidad asintótica para el peor de los casos.**
  T(n)=C1*n+C2*n^2+C3*n^2+C4*n^2
  T(n)=n^2*(C1+C3+C4)+C' (Regla de las sumas y producto)
  T(n)=n^2*C' (Se quitan los productos y se simplifica la suma)
  T(n)=O(n^2)

- **Gráfica.**

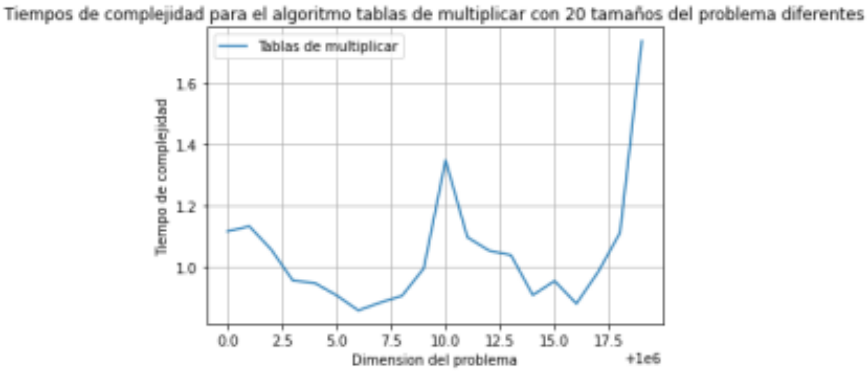

Figura 6. Gráfica (experimental) del algoritmo 'Tablas de multiplicar' con 20 tamaños de problema diferentes en Python.

Tamaño del problema (n) [En este caso hasta donde va a llegar la tabla de multiplicar]	Tiempo de complejidad(T(n))
1000000	1.117292166
1000001	1.133828878
1000002	1.058720589
1000003	0.957368612
1000004	0.949926615
1000005	0.909908056
1000006	0.860890865
1000007	0.886265755
1000008	0.908121824
1000009	0.997231722
1000010	1.348102331
1000011	1.09817481
1000012	1.054695368
1000013	1.040132046
1000014	0.910562277
1000015	0.956426859
1000016	0.882814407
1000017	0.986088037
1000018	1.112864971
1000019	1.735034466

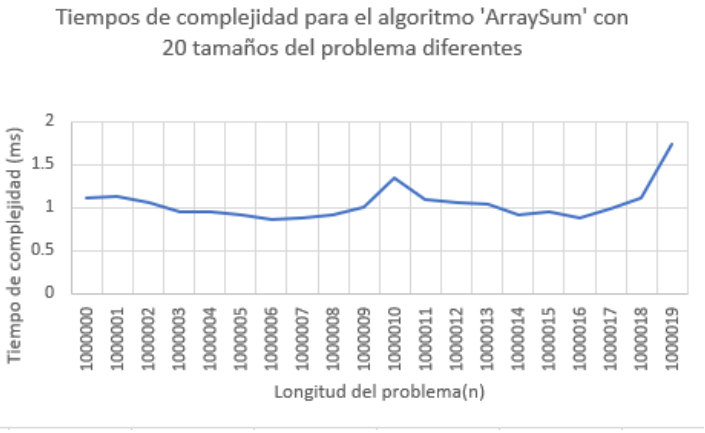

Figura 7. Gráfica (experimental) del algoritmo 'Tablas de multiplicar' con sus longitudes (n) y tiempo de complejidad para 20 tamaños del problema diferentes en Excel.

- **Análisis:** Se puede ver que al igual que en los puntos anteriores, la gráfica se asemeja a la de una función cuadrática pero no es del todo precisa debido a que no se está tomando el peor de los casos; también se puede ver que no funcionará muy bien para grandes dimensiones del problema ya que se tomará mucho tiempo.

- **¿La teoría (notación asintótica) corresponde a lo encontrado de forma experimental al tomar los tiempos de la implementación?** Si, corresponde a lo encontrado, pero no de forma exacta, aunque si aproximada; debido a la justificación del punto anterior.