Comparativa de Algoritmos de Ordenamiento Estructura de Datos y Algoritmos

Jesus Alpaca Rendon

Agosto 2023

1 Introduccion

El presente informe muestra la comparativa de tiempos de ejecucion realizada entre 5 algoritmos de ordenamiento como son: Binary Insertion Sort, Bubble sort, Quick sort, Selection sort y Merge sort. Implementados en 3 diferentes lenguajes de programacion: Python, Golang y C++. Los algoritmos fueron sometidos a pruebas para obtener un tiempo promedio de ejecucion y ver los resultados en tablas comparativas y graficos con los resultados para determinar el performance de los algoritmos y lenguajes.

2 Algoritmos

Los algoritmos seleccionados para el presente trabajo de investigacion son los siguientes:
* Binary Insertion Sort
* Bubble sort
* Quick sort
* Selection sort
* Merge Sort

1. Binary Insertion Sort

   Este algoritmo es una combinacion de 2 algoritmos existentes: Binary Search y Insertion Sort, Esta combinacion nos permite ordenar una lista de n valores, el cual se constituye en un inicio de un bucle que

nos permitira recorrer todo el array seleccionando el segundo elemento, ya que se considera el primero como ordenado, una vez seleccionado, aplicaremos la busqueda binaria, Cabe resaltar que la busqueda binaria funciona correctamente si la lista esta ordenada. En este caso la busqueda binaria solo se aplicara al lado izquierdo del array ya que es la parte considerada ordenada. Comparamos el valor siguiente i+1 con el punto medio de la lista ordenada, y se verifica si es mayor o menor, en caso de ser mayor se coloca en al derecha y caso contrario a la izquierda. En el mejor de los casos, el algoritmo de Insertion Sort puede tener una sola comparacion, pero en el peor de los casos puede tener n comparaciones, y en el caso de Binary Insertion sort, reducimos el maximo de casos, ya que se reduce a log n comparaciones como maximo .Estos pasos se repiten por cada elemento dejandonos en un inicio una complejidad de O(n), la cual con la complejidad de Binary Search de nos da una complejidad de O(n log n), sin embargo, no se esta considerando el movimiento que se da a los items en caso tengamos numeros peque;os al final, para esto, tendria que recorrer n posiciones hasta colocarlo en su lugar, lo cual incrementaria la complejidad, quedando lo siguiente: Binary Search + movimiento de items = O(log n) + O(n). Si ahora colocamos la complejidad de Insert Sort, nos quedaria de esta manera: O(n) x (O(log n) + O(n)) = O(n log n) + O(n2). Como complejidad final del algoritmo Binary Insertion Sort tendriamos O(n2).

```
procedure binary insertion sort(a_1, a_2, ..., a_n:
   real numbers with n ≥ 2)
for j := 2 to n
  {binary search for insertion location i}
  left := 1
  right := j − 1
  while left < right
    middle := ⌊(left + right)/2⌋
    if a_j > a_middle then left := middle + 1
    else right := middle
  if a_j < a_left then i := left else i := left + 1
  {insert a_j in location i by moving a_i through a_{j−1}
      toward back of list}
  m := a_j
  for k := 0 to j − i − 1
    a_{j−k} := a_{j−k−1}
  a_i := m
{a_1, a_2, ..., a_n are sorted}
```

Figure 1: Binary Insertion Sort Algorithm

2. Bubble Sort

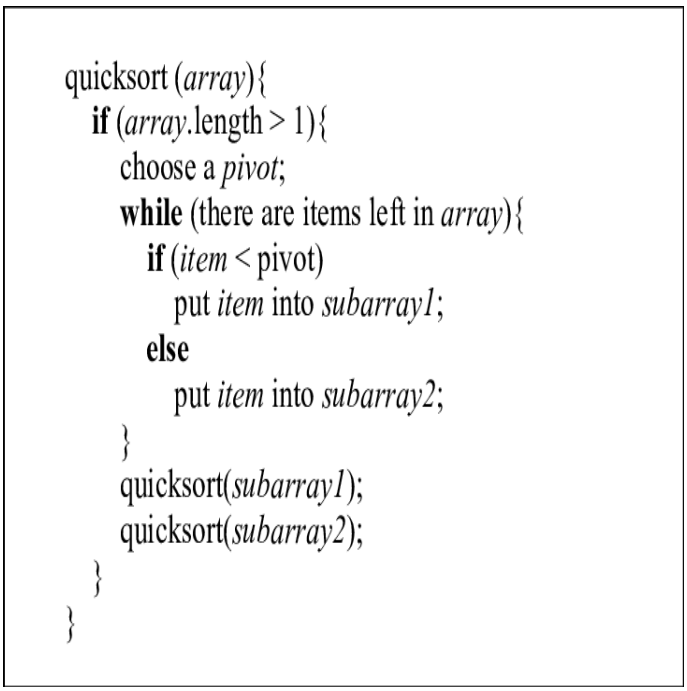
   Este algoritmo realiza el ordenamiento o reordenamiento de una lista a de n valores, en este caso de n términos numerados del 0 al n-1; consta de dos bucles anidados, uno con el índice i, que da un tamaño menor al recorrido de la burbuja en sentido inverso de 2 a n, y un segundo bucle con el índice j, con un recorrido desde 0 hasta n-i, para cada iteración del primer bucle, que indica el lugar de la burbuja. La burbuja son dos términos de la lista seguidos, j y j+1, que se comparan: si el primero es mayor que el segundo sus valores se intercambian.

Data: a_1,a_2,a_3,a_4...$a_{(n-1)}$
Result: ordered list
initialization;
for i **to** *n-1* **do**
  for j **to** *n-i-1* **do**
    if $a_j > a_{j+1}$ **then**
      go to next section;
      $aux \leftarrow a_j$;
      $a_j \leftarrow a_{j+1}$;
      $a_{j+1} \leftarrow aux$;
    end
  end
end

Algorithm 1: Bubble Algorithm

3. Quick Sort

   Es uno de los algoritmos mas rapidos. Tiene un elemento muy importante para su ejecucion la cual es la seleccion del pivote. El pivote es el elemento desde el cual nosotros dividiremos la lista. Quicksort es un algoritmo de divide y venceras y el pivote es muy importante, incluso su seleccion influye en su complejidad. Principalmente el algoritmo toma el pivote, divide la lista y empieza a analizar: Si encuentra un numero menor que el pivote, lo colocara a su izquierda, caso contrario a la derecha, adicionalmente se debe tener punteros al inicio y al final. Por cada array que se forma se debe escoger un pivote,al final tendremos una lista ordenada sin necesidad de aplicar una mezcla como lo usa el merge sort. La complejidad del algoritmo es de O(nlogn) el cual es comparado con merge sort, sin embargo hay que tener en cuenta que el peor de los casos de este algoritmo es cuando tenemos una lista ordenada y se selecciona como pivote uno de los extremos, dandonos una complejidad en este caso de O(n2)



```
quicksort (array){
  if (array.length > 1){
    choose a pivot;
    while (there are items left in array){
      if (item < pivot)
        put item into subarray1;
      else
        put item into subarray2;
    }
    quicksort(subarray1);
    quicksort(subarray2);
  }
}
```

Figure 2: Quick Sort Algorithm

4. Selection Sort

   Este algoritmo tiene una complejidad de O(n2), por lo que no es uno de los mas rapidos, el algoritmo empieza determinando 2 elementos claves los cuales son: el minimo actual y el item actual, para ello inicia en la posicion inicial. Durante cada iteracion, va seleccionando el item minimo de la lista desordenada y lo mueve a la posicion ordenada, cuando encuentra el minimo, lo intercambia con la posicion del item seleccionado. Es por esta razon que tiene una complejidad alta ya que al intercambiarlo, en caso que sea el item minimo consecuente, tendra que recorrer en el peor de los casos n veces para hacer el intercambio

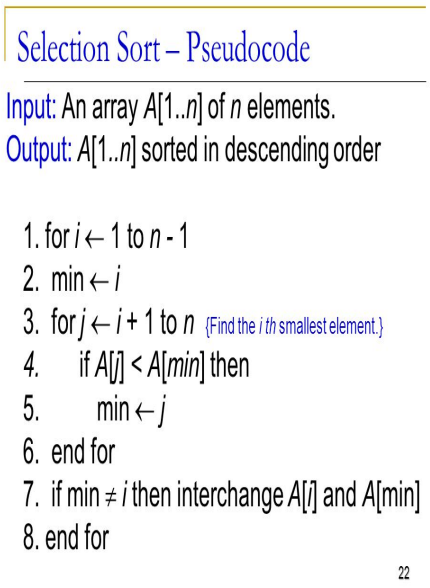


Figure 3: Selection Sort Algorithm

5. Merge Sort

   El algoritmo Merge Sort se puede representar en 2 partes: la funcion MergeSort que realiza la division de una lista de numeros en 2 partes, siendo el punto medio la parte donde se divide, y la funcion mezclar Que es la que se encargara de realizar la logica donde se ordenara 2 listas que recibe como parametro y hacer una mezcla entre ellas. El algoritmo Merge Sort es uno de los algoritmos mas rapidos de ordenamiento, ya que nos permite reducir la lista en partes iguales y realizar mezclas sucesivas. Cuando la longitud de un array es 1, se considera ordenado y retorna, en caso de lista de n elementos, la comparacion se hace en parejas es decir, empieza a comparar uno a uno los elementos iniciales de cada lista, creando una lista temporal donde tendremos la nueva lista mezclada y ordenada. Si durante el proceso de comparacion, el indice supera la longitud de la lista, se toma los elementos restantes de la otra y se colocan en la nueva lista. La complejidad de este algoritmo es de O(n log n), siendo la parte de la comparacion la que tome mas protagonismo ya que al momento de comparar las 2 listas, estas se recorren con sus respectivos indices iterando de uno en uno los elementos y comparandolos segun la posicion que esten.

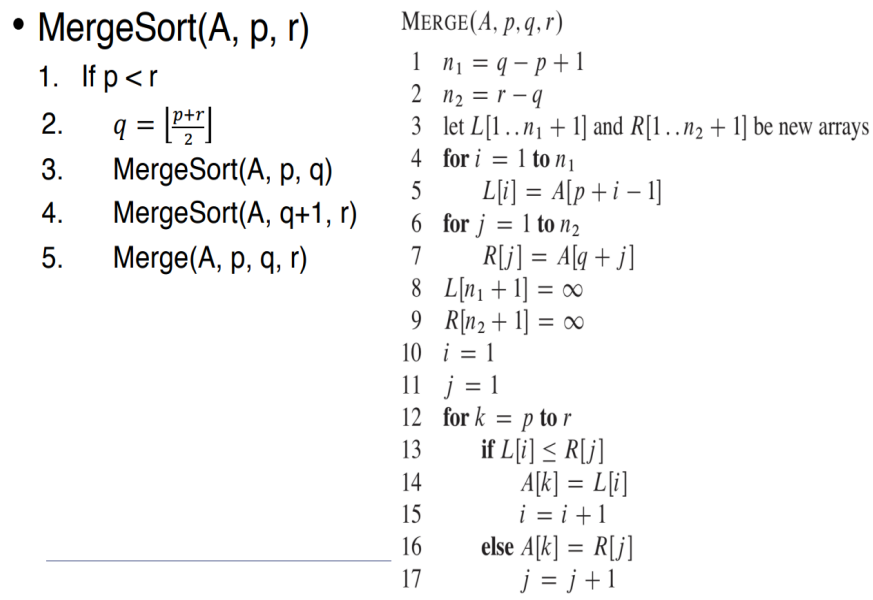


Figure 4: Merge Sort Algorithm

3 Implementacion

Para la implementacion se trabajo en secuencia de la siguiente manera:

- Se selecciono los algoritmos para trabajar y se busco su implementacion en los 3 lenguajes
- Se realizaron modificaciones para automatizar la ejecucion de las iteraciones con archivos txt que contenian listas de numeros de diferentes tamaños y que generara archivos con los resultados.
- Se creo scripts para generar los calculos del promedio y desviacion estandar por lenguaje y tamaño de la lista.
- Se genero un script para la generacion de los graficos comparativos por algoritmo y longitud de lista.
- Se genero una tabla comparativa general y otras por algoritmo para incluirse en el informe.
- Se desarrollo el presente informe en latex.

La implementacion del ejercicio se encuentra en el siguiente enlace de Github: https://github.com/Alpha004/AyEDJAAR2023

4 Resultados

Los resultados del informe se muestran en 5 tablas divididas por lenguajes donde cada una muestra las graficas correspondientes a las pruebas realizadas con los diferentes lenguajes A continuacion se muestran las tablas comparativas y los graficos correspondientes por algoritmo:

BUBBLE SORT

DATOS	Python		C++		Golang	
Cantidad	Promedio	Desv. Est.	Promedio	Desv. Est.	Promedio	Desv. Est.
100	0.0018	0.0004	0.00054	0.000062	0.00034	0.000277
1000	0.1342	0.002401	0.0042	0.003119	0.001477	0.000428
2000	0.5764	0.043482	0.00923	0.000409	0.003846	0.000178
3000	1.232601	0.004318	0.020784	0.001754	0.007909	0.000213
4000	2.230502	0.042581	0.03684	0.00091	0.013814	0.000629
5000	3.5132	0.055782	0.058489	0.000285	0.021313	0.000226
6000	5.058199	0.163511	0.08836	0.001533	0.031634	0.0011
7000	6.958199	0.194601	0.119168	0.001122	0.046711	0.005821
8000	9.105999	0.21782	0.158906	0.001598	0.057926	0.000417
9000	11.510303	0.144547	0.209791	0.00918	0.075737	0.000391
10000	14.1332	0.269819	0.254562	0.00789	0.095036	0.000408
20000	56.44461	0.485198	1.088809	0.008167	0.442731	0.013076
30000	127.527399	0.335833	2.574614	0.057651	1.039052	0.015263
40000	226.8496	0.944975	4.670932	0.089209	1.89731	0.035354
50000	366.973335	15.659894	7.311926	0.038472	2.999633	0.039834

Table 1: Tabla de Promedios y Desviacion Estandar de los 3 lenguajes con Bubble Sort

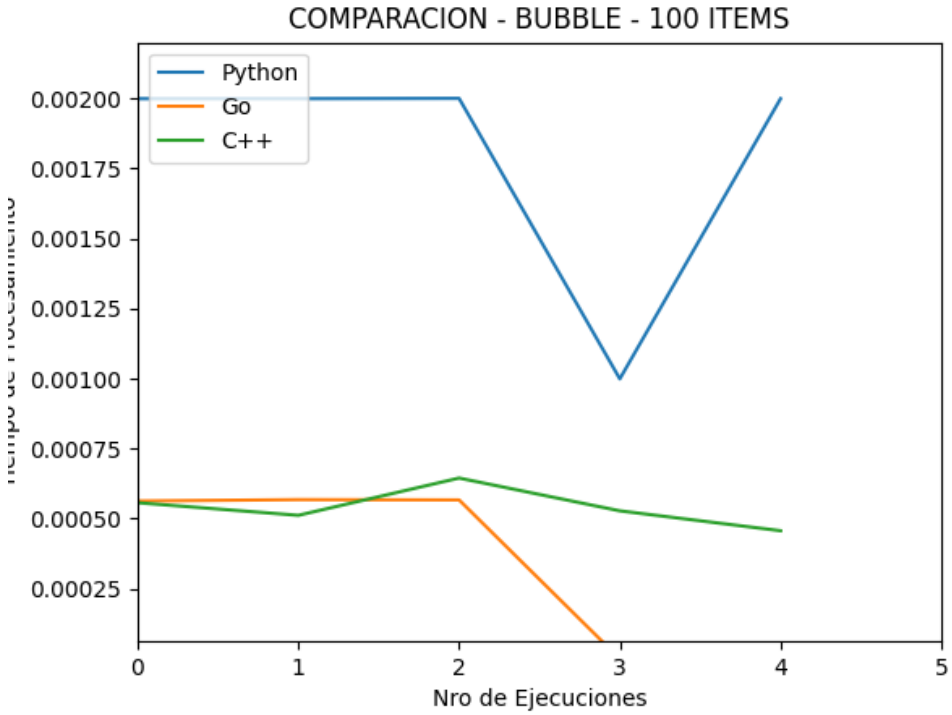


Figure 5: Bubble Sort - 100 items

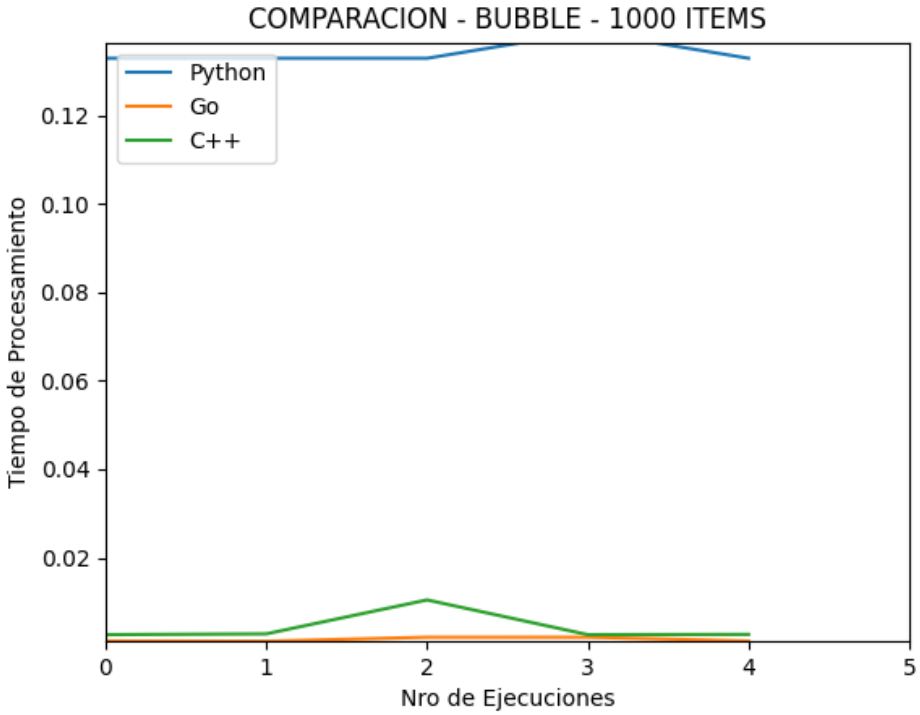


Figure 6: Bubble Sort - 1000 items

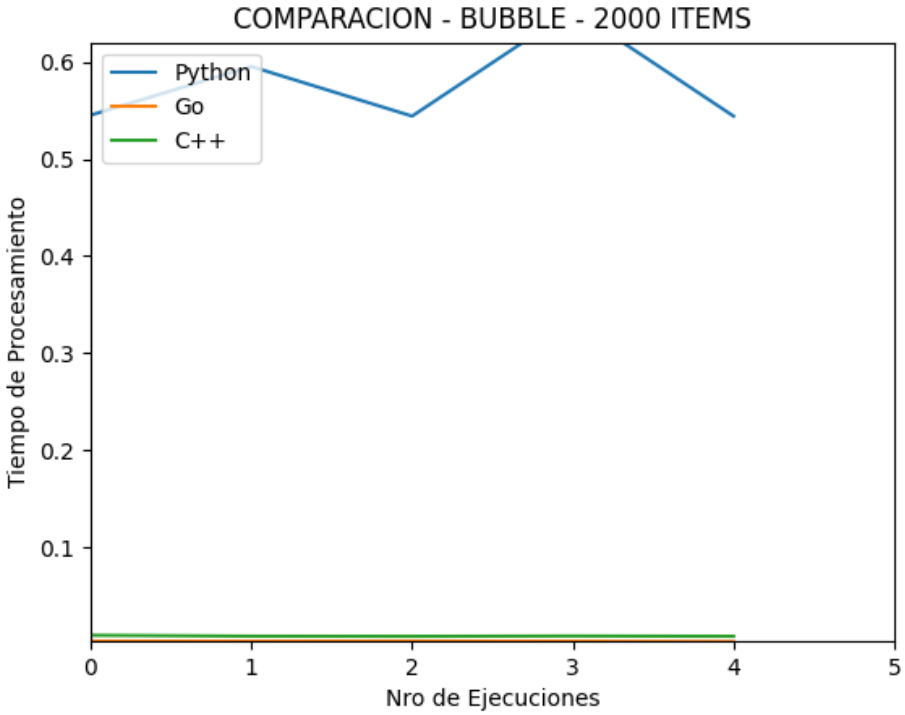


Figure 7: Bubble Sort - 2000 items

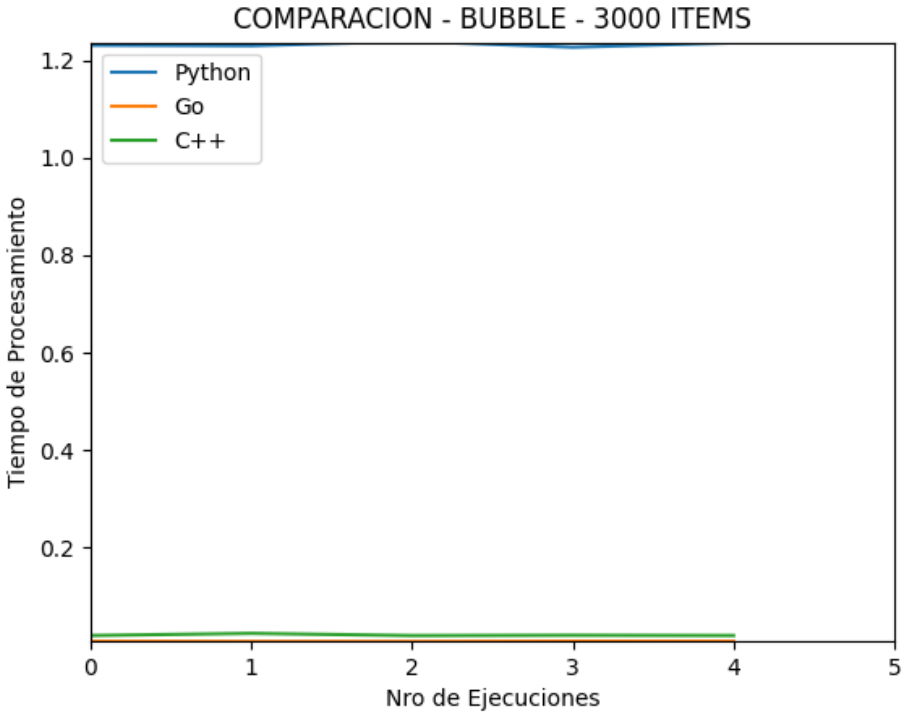


Figure 8: Bubble Sort - 3000 items

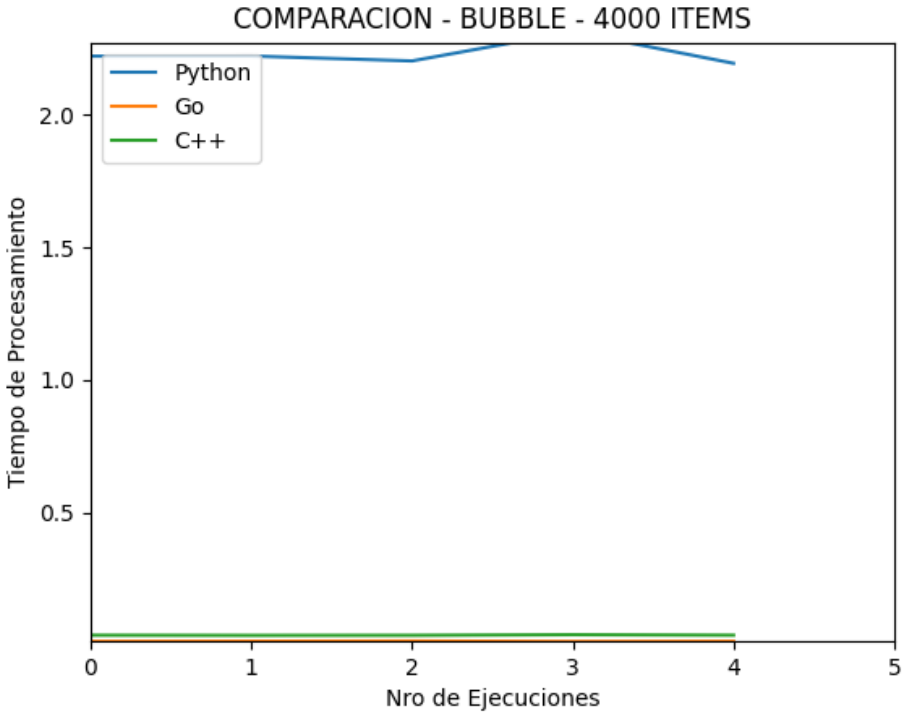


Figure 9: Bubble Sort - 4000 items

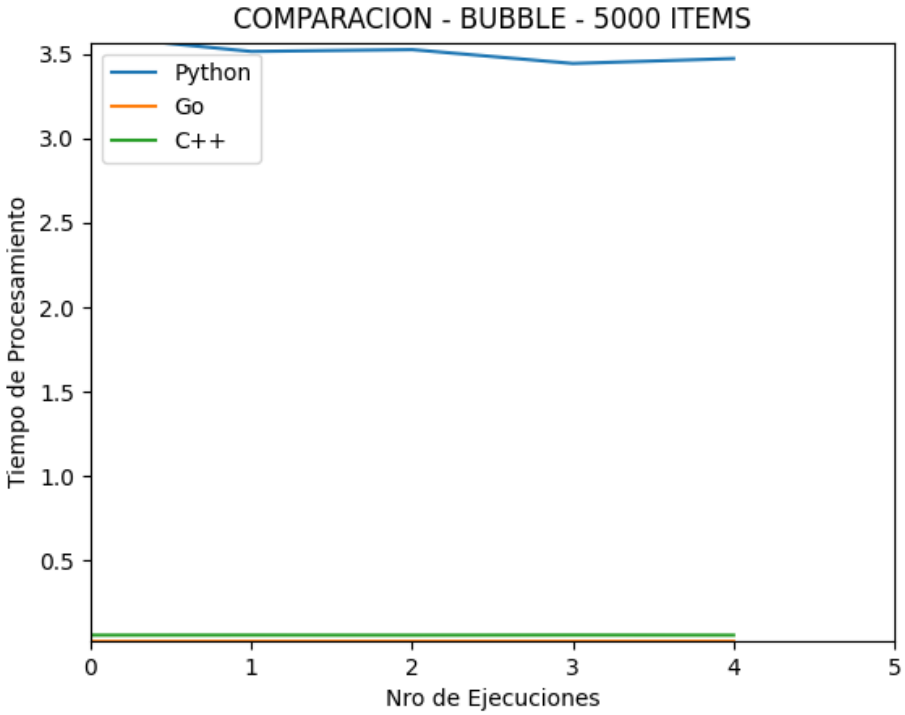


Figure 10: Bubble Sort - 5000 items

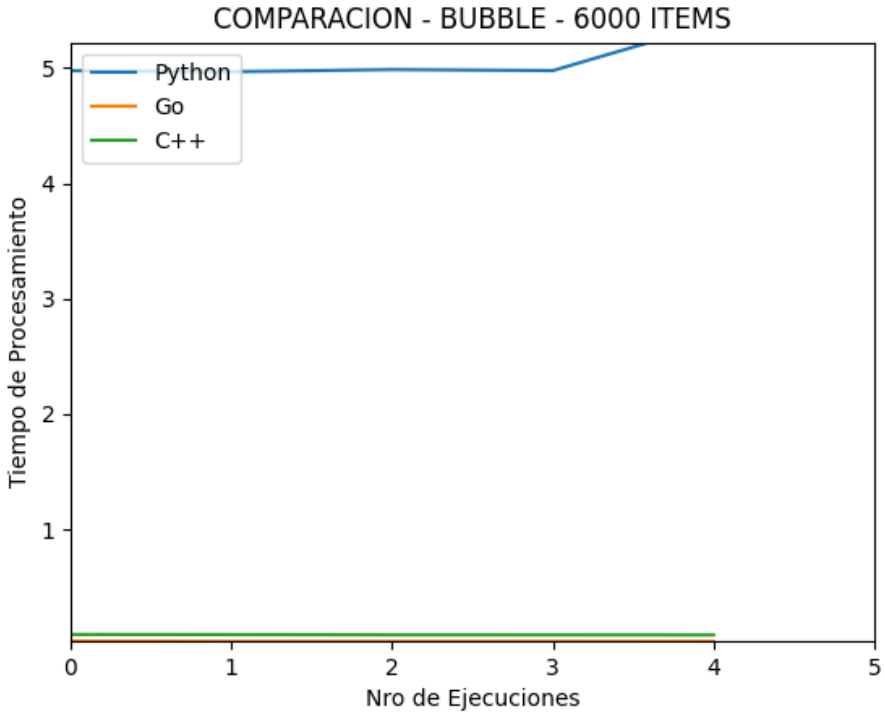


Figure 11: Bubble Sort - 6000 items

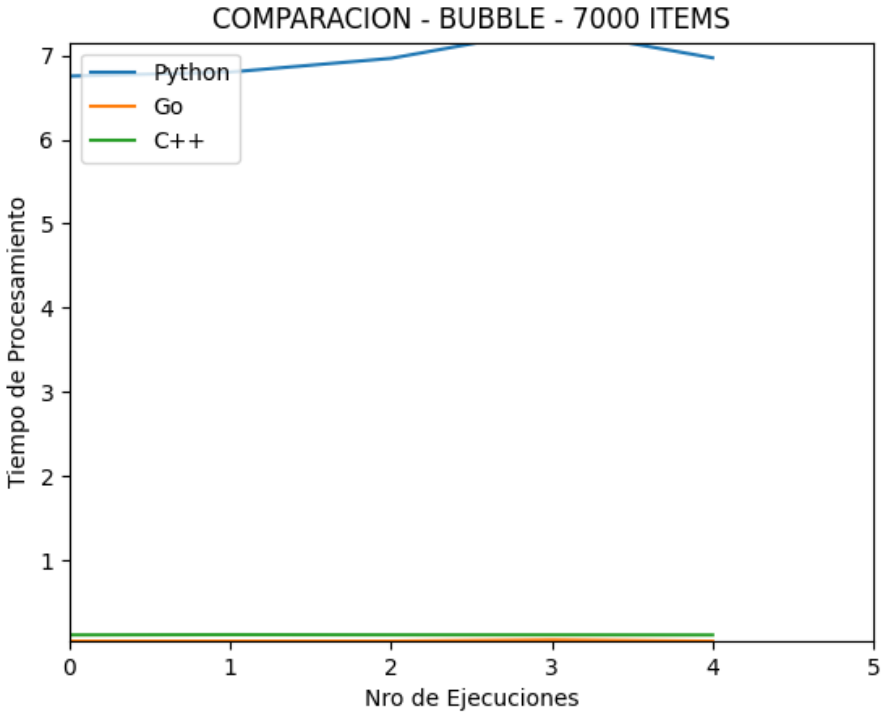


Figure 12: Bubble Sort - 7000 items

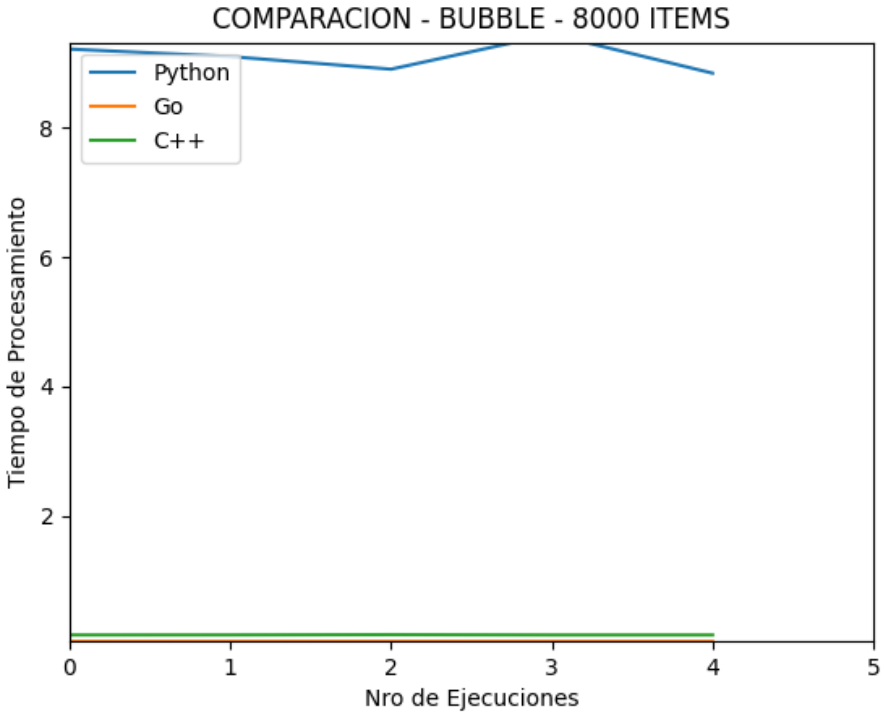


Figure 13: Bubble Sort - 8000 items

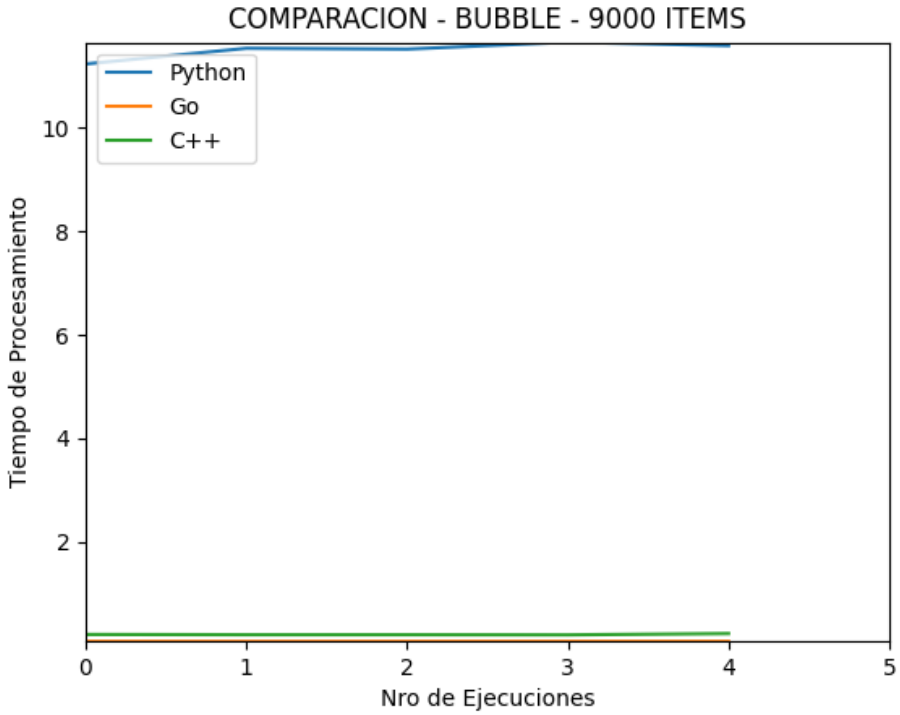


Figure 14: Bubble Sort - 9000 items

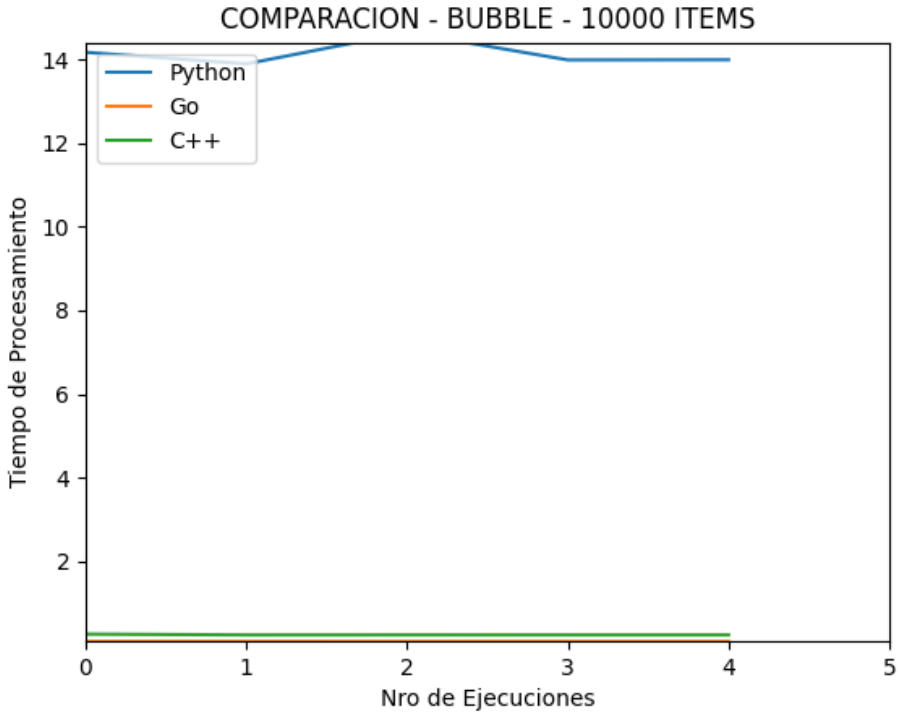


Figure 15: Bubble Sort - 10000 items

Figure 16: Bubble Sort - 20000 items

Figure 17: Bubble Sort - 30000 items

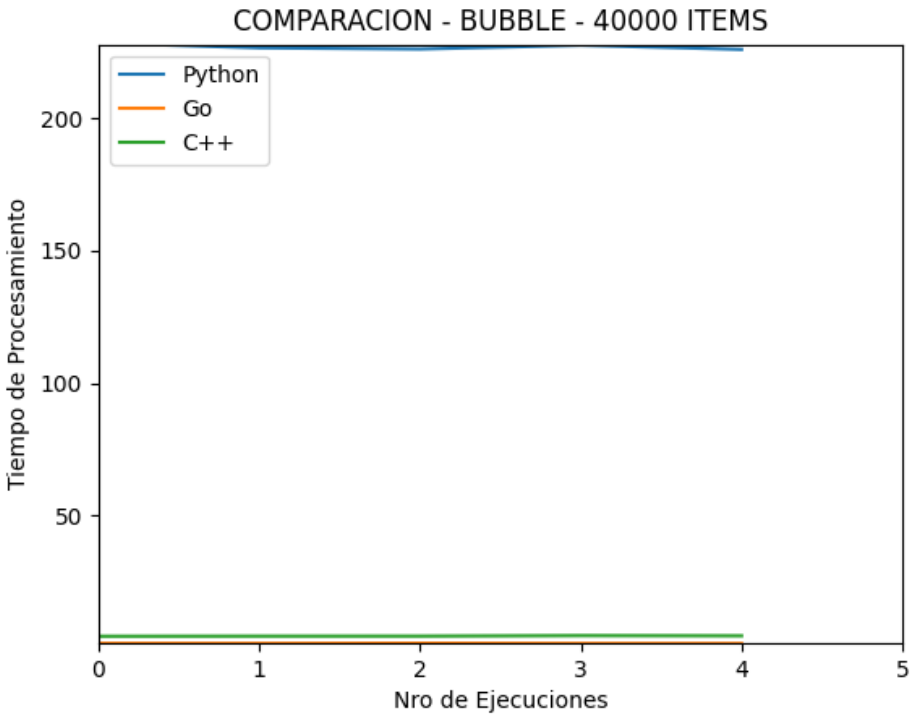


Figure 18: Bubble Sort - 40000 items

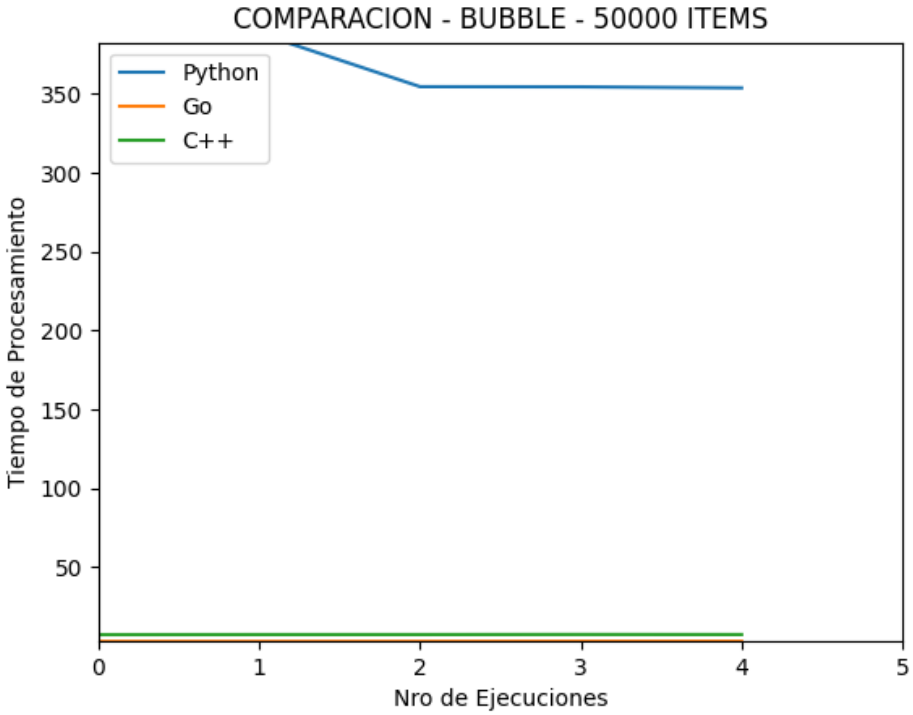


Figure 19: Bubble Sort - 50000 items

BINARY INSERT SORT

DATOS	Python		C++		Golang	
Cantidad	Promedio	Desv. Est.	Promedio	Desv. Est.	Promedio	Desv. Est.
100	0.001179	0.000412	0.000567	0.000165	0.000227	0.000278
1000	0.011801	0.0004	0.001388	0.000204	0.001113	0.001147
2000	0.036404	0.000486	0.00277	0.0002	0.00188	0.000297
3000	0.077004	0.000543	0.004932	0.000198	0.003172	0.000225
4000	0.146665	0.007182	0.00784	0.000115	0.004691	0.000264
5000	0.230609	0.00242	0.011518	0.000306	0.006384	0.00026
6000	0.34093	0.005628	0.017372	0.002291	0.008773	0.000949
7000	0.467704	0.004512	0.022372	0.002704	0.01045	0.000267
8000	0.676307	0.045625	0.027731	0.000381	0.013188	0.000331
9000	0.839014	0.013345	0.03324	0.000159	0.016326	0.000955
10000	1.095616	0.065637	0.044264	0.005229	0.019185	0.00045
20000	4.26201	0.03094	0.155018	0.004108	0.062971	0.000338
30000	9.677808	0.236474	0.345256	0.004813	0.134754	0.002914
40000	17.100235	0.29154	0.599238	0.006217	0.237646	0.009486
50000	27.796896	0.641884	0.945106	0.010825	0.357538	0.012083

Table 2: Tabla de Promedios y Desviacion Estandar de los 3 lenguajes con Binary Insertion Sort

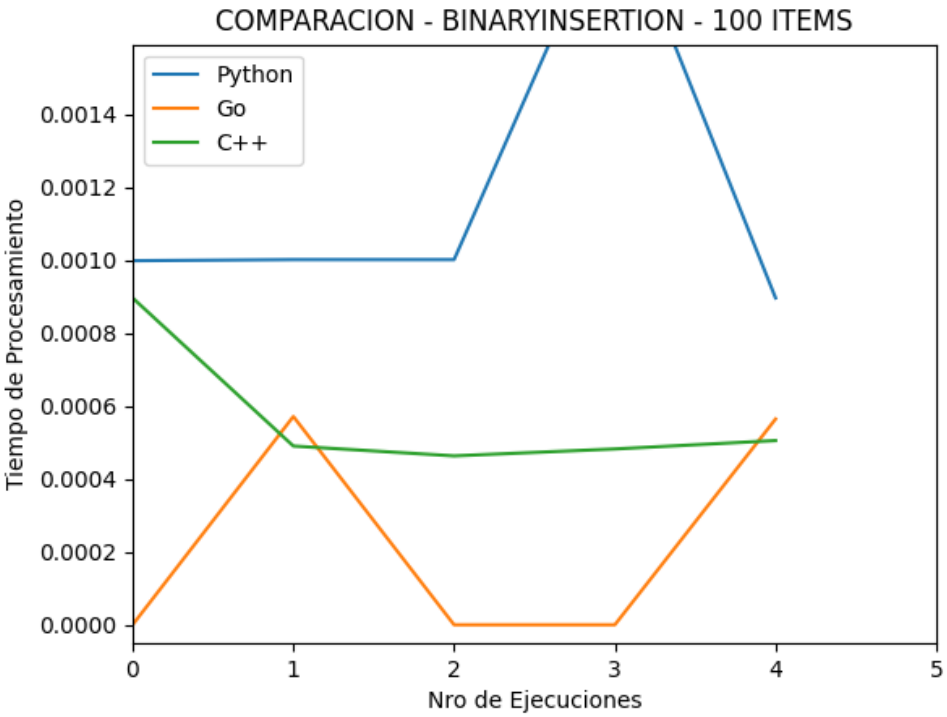


Figure 20: Binary Insert Sort - 100 items

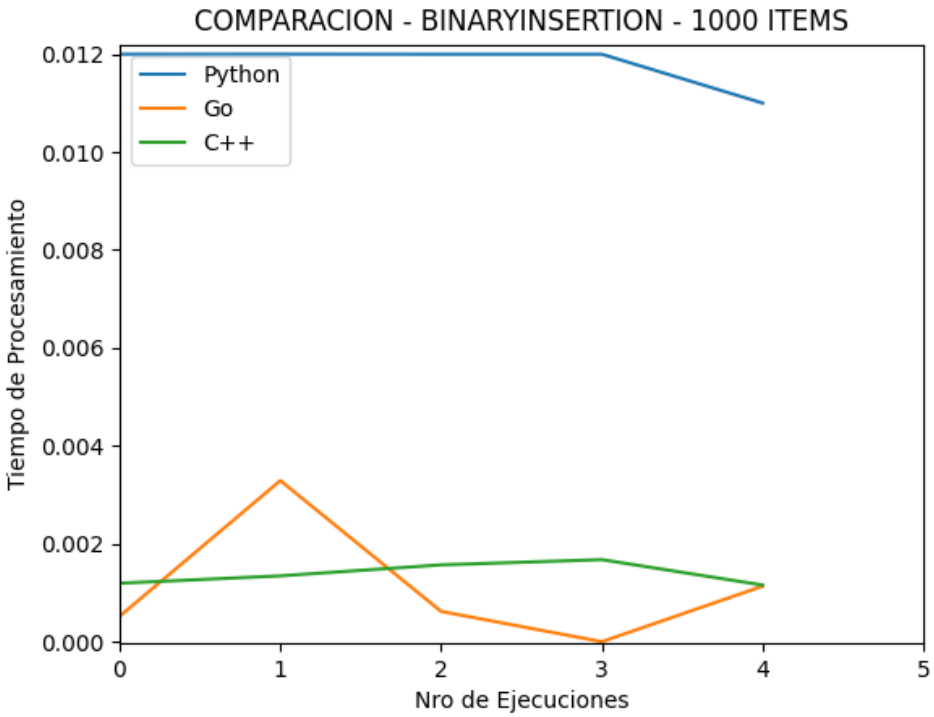


Figure 21: Binary Insert Sort - 1000 items

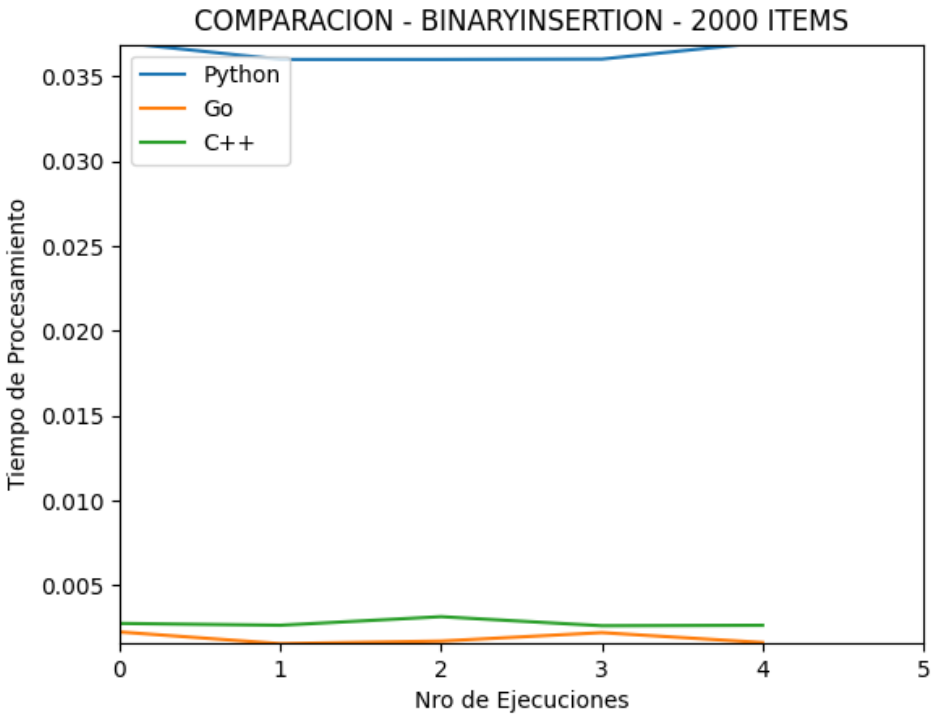


Figure 22: Binary Insert Sort - 2000 items

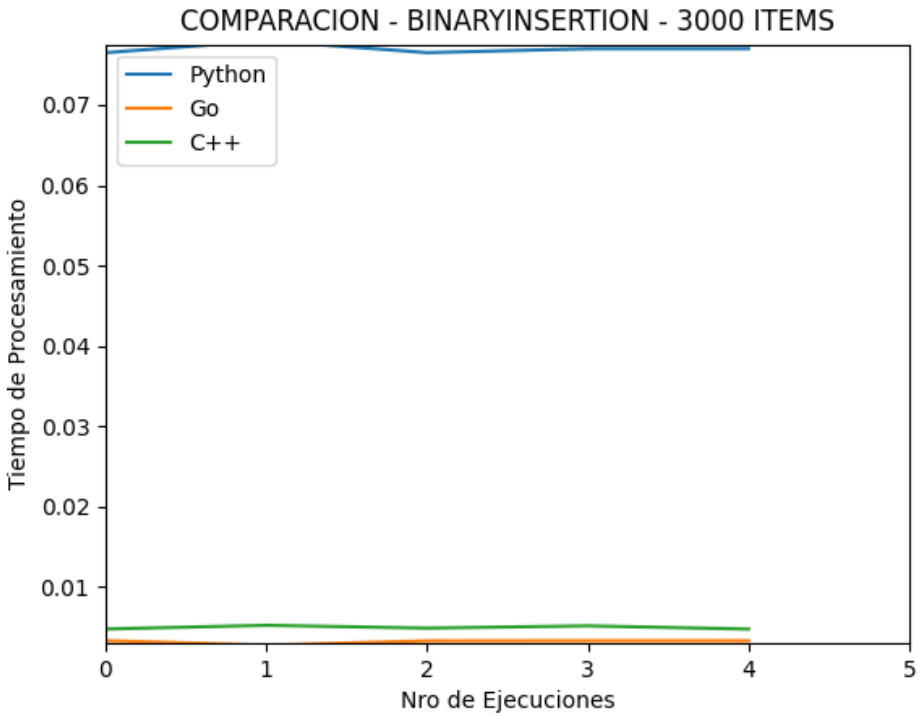


Figure 23: Binary Insert Sort - 3000 items

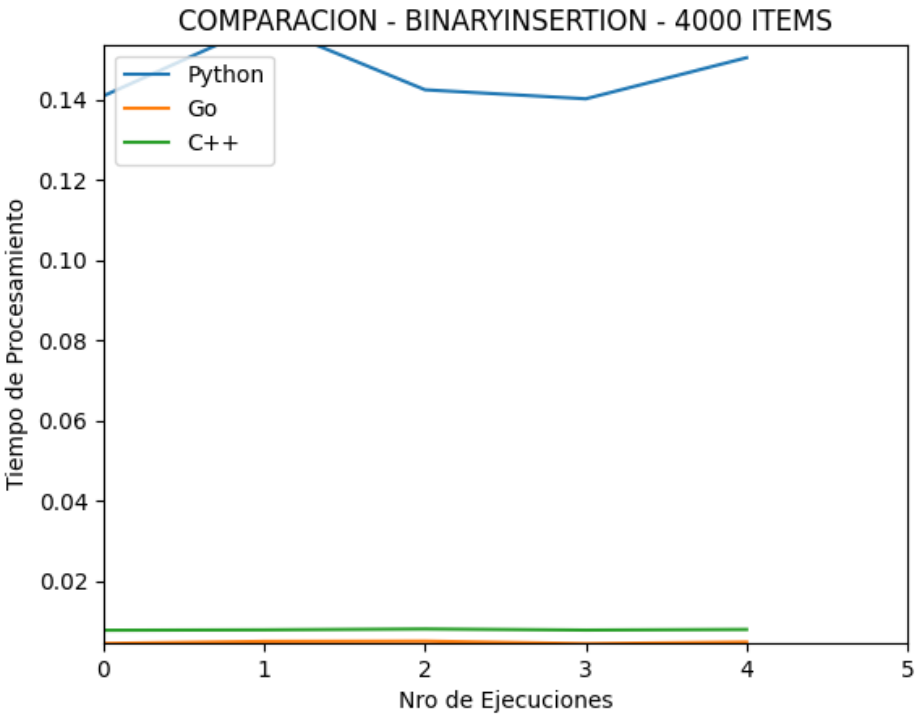


Figure 24: Binary Insert Sort - 4000 items

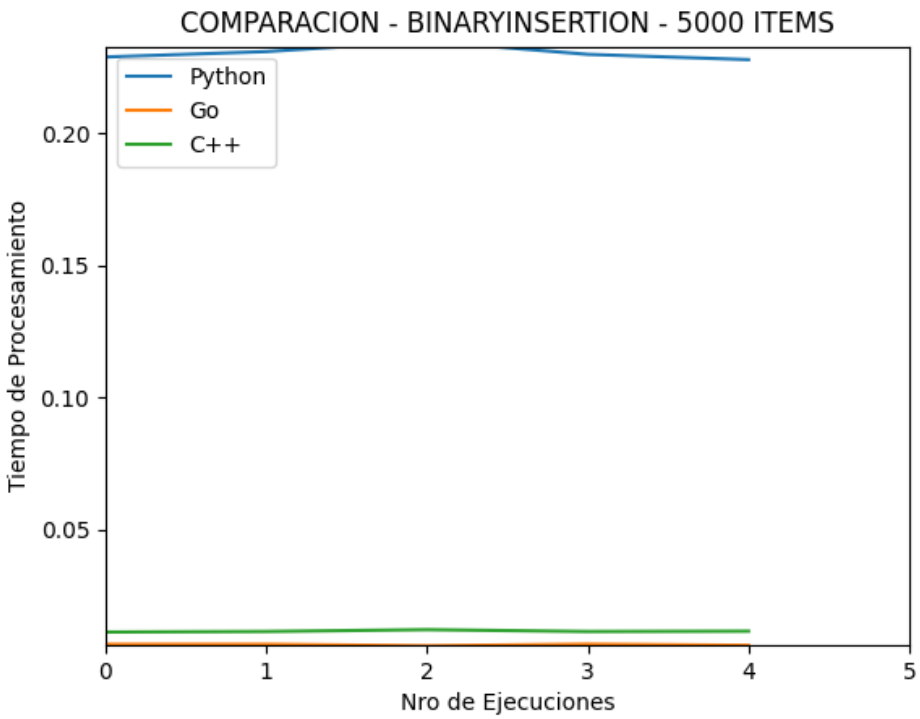


Figure 25: Binary Insert Sort - 5000 items

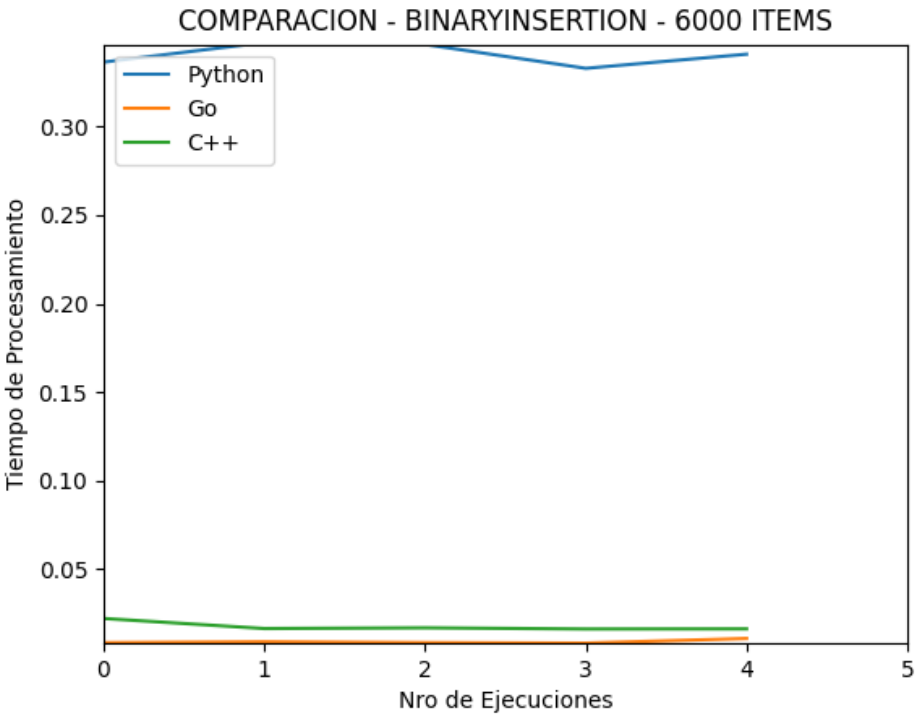


Figure 26: Binary Insert Sort - 6000 items

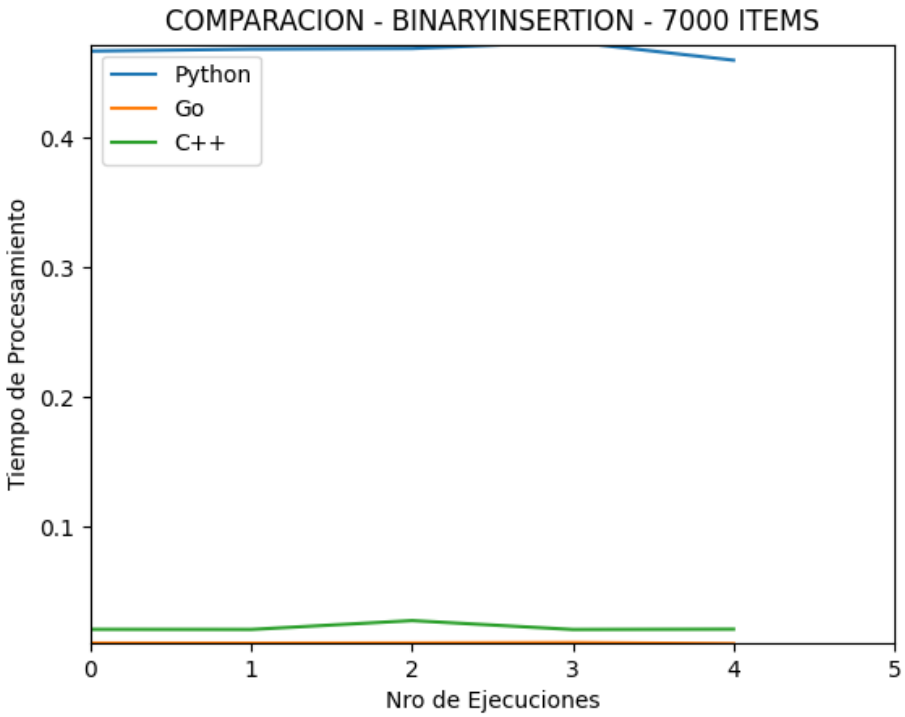


Figure 27: Binary Insert Sort - 7000 items

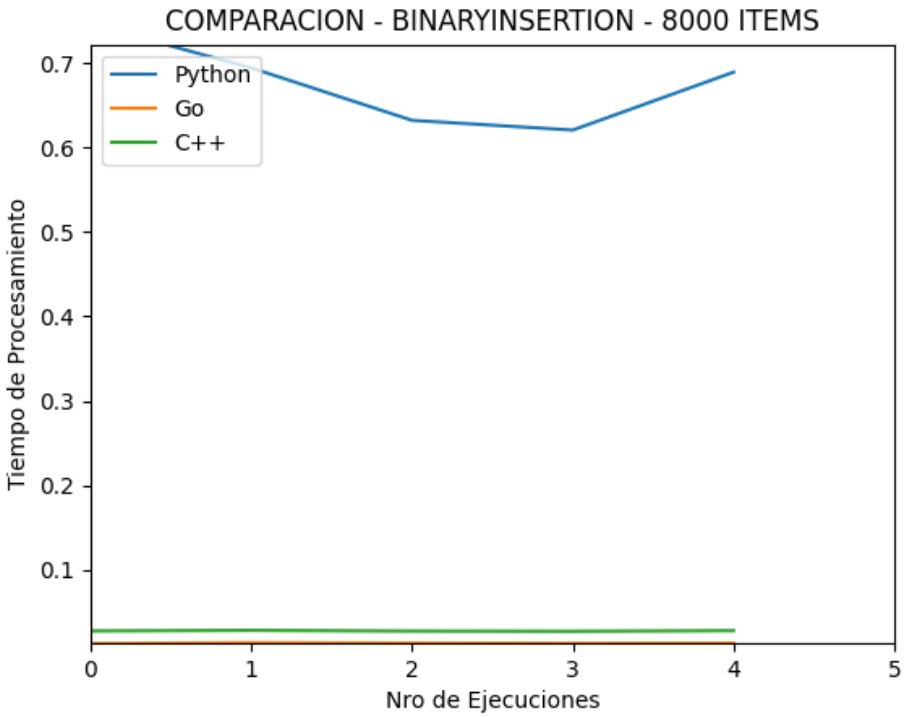


Figure 28: Binary Insert Sort - 8000 items

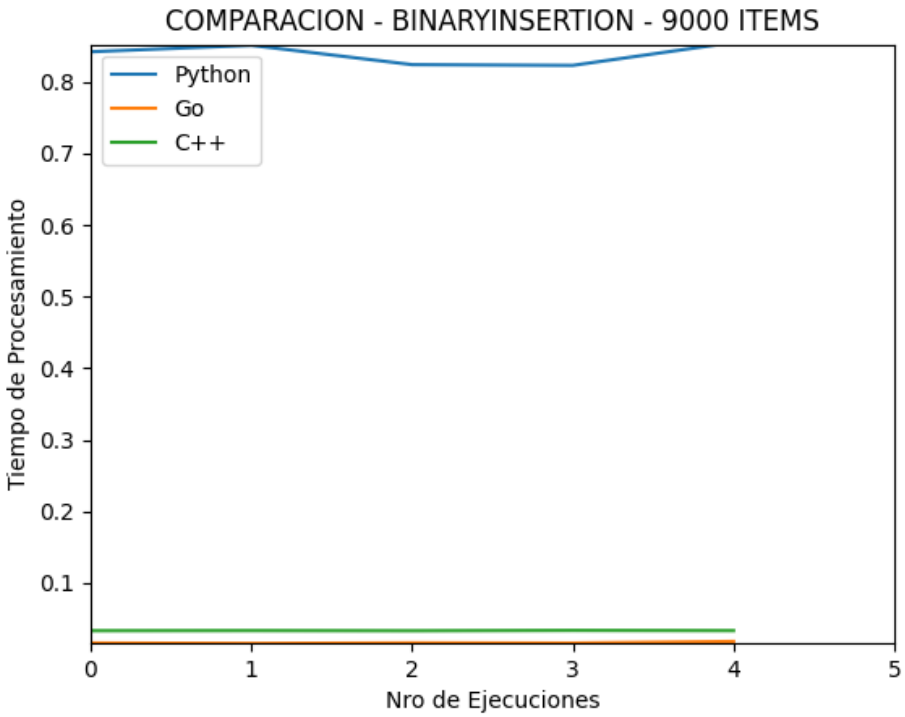


Figure 29: Binary Insert Sort - 9000 items

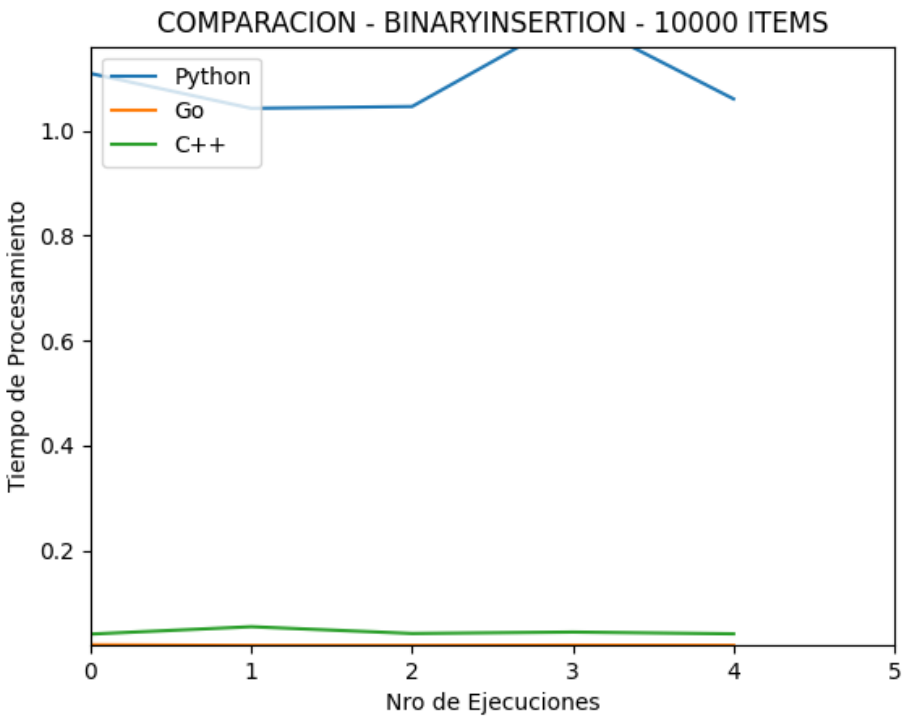


Figure 30: Binary Insert Sort - 10000 items

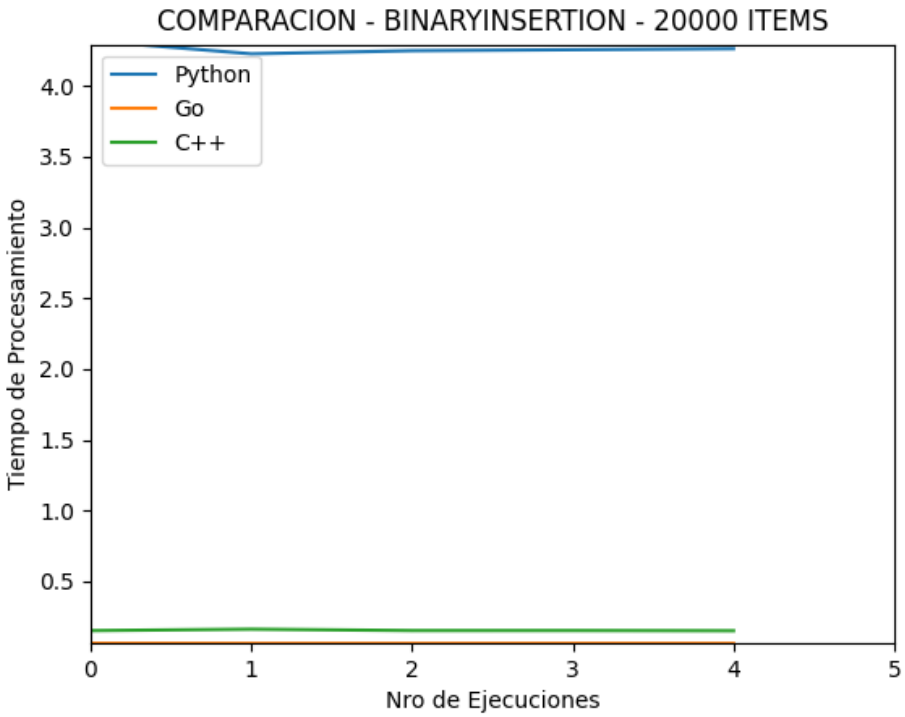


Figure 31: Binary Insert Sort - 20000 items

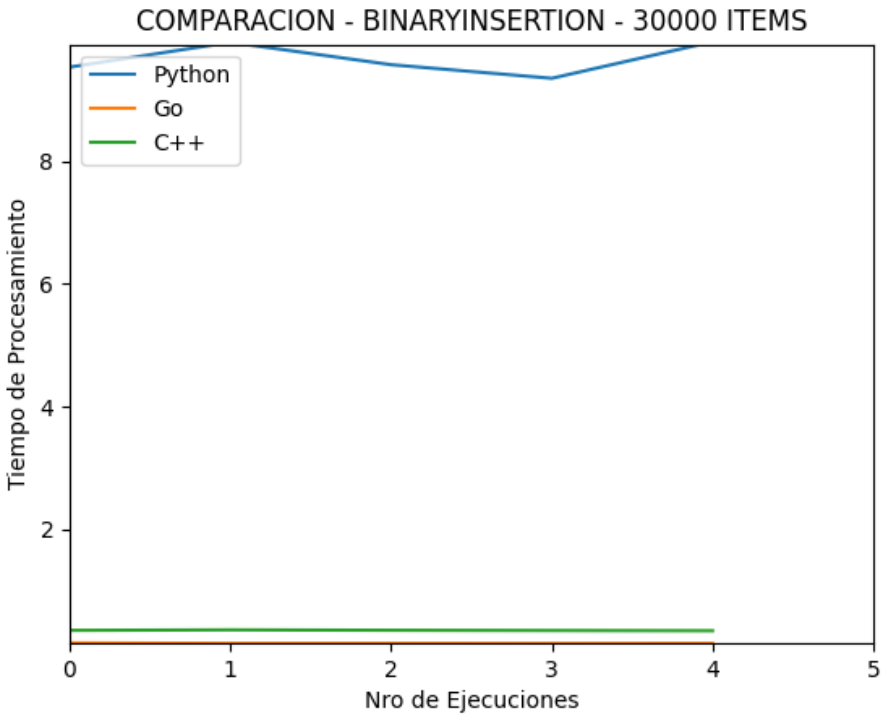


Figure 32: Binary Insert Sort - 30000 items

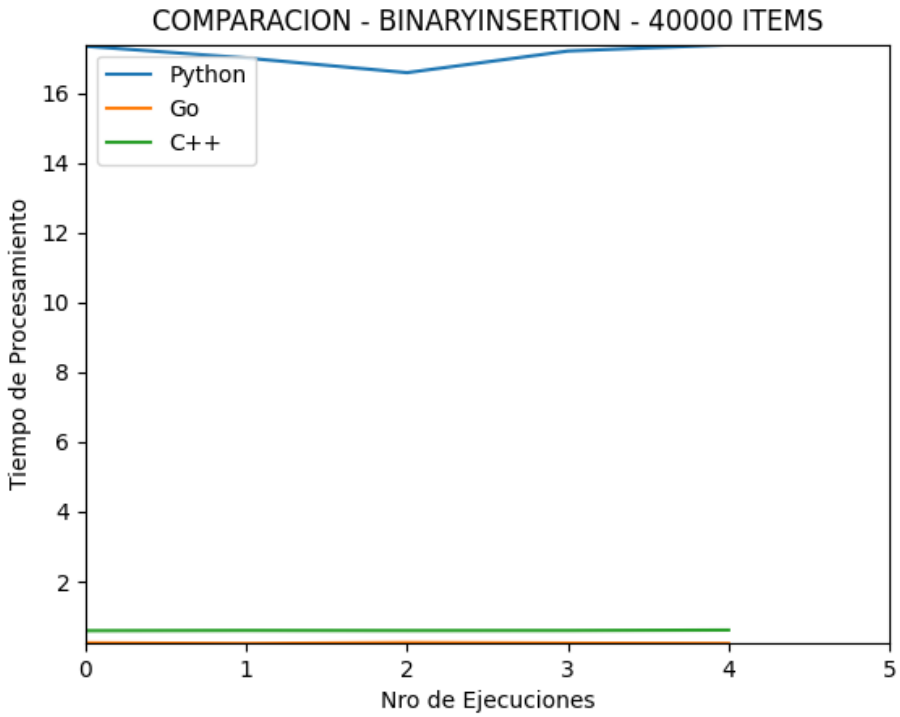


Figure 33: Binary Insert Sort - 40000 items

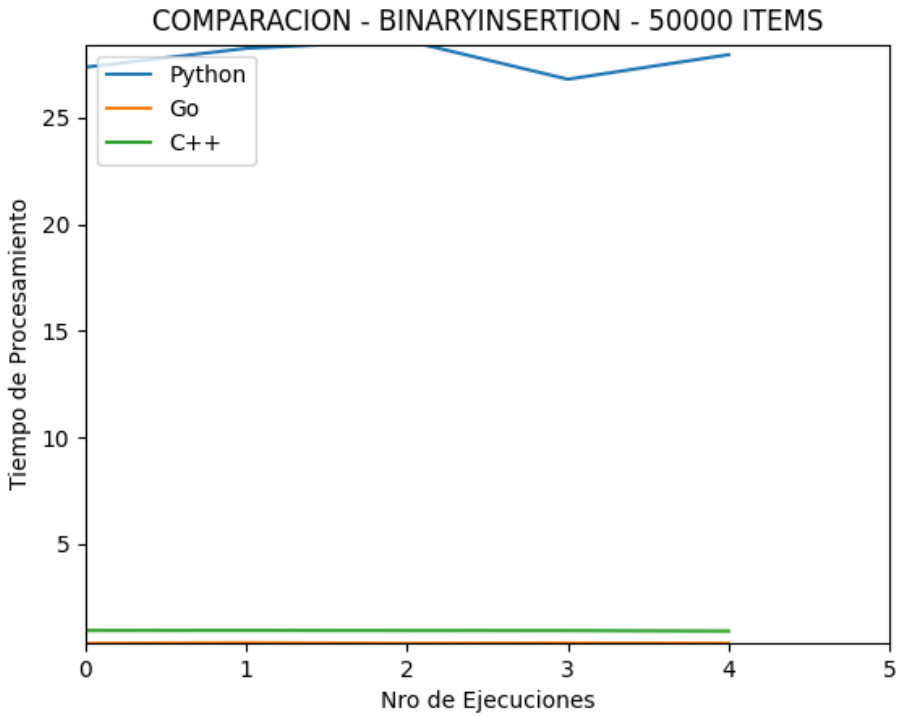


Figure 34: Binary Insert Sort - 50000 items

MERGE SORT

DATOS	Python		C++		Golang	
Cantidad	Promedio	Desv. Est.	Promedio	Desv. Est.	Promedio	Desv. Est.
100	0.0006	0.00049	0.000533	0.000034	0.000429	0.000367
1000	0.003	0.000001	0.000923	0.000036	0.000971	0.000206
2000	0.0074	0.000491	0.001465	0.00009	0.00162	0.000278
3000	0.0118	0.0004	0.001893	0.00005	0.002489	0.000452
4000	0.0162	0.001469	0.002358	0.000061	0.003235	0.000381
5000	0.0214	0.001358	0.00309	0.000373	0.004025	0.000274
6000	0.053601	0.027376	0.003449	0.000175	0.005202	0.000436
7000	0.0302	0.0004	0.00626	0.004867	0.005494	0.000316
8000	0.038199	0.004167	0.004378	0.000259	0.006255	0.000319
9000	0.043999	0.007043	0.005047	0.000189	0.008111	0.00061
10000	0.045199	0.0004	0.005335	0.000297	0.008815	0.000114
20000	0.0988	0.005601	0.010452	0.000529	0.017305	0.002439
30000	0.153799	0.007222	0.015751	0.000578	0.023033	0.000283
40000	0.2198	0.018808	0.020355	0.000142	0.030863	0.000457
50000	0.263199	0.000748	0.025606	0.000265	0.038622	0.001223

Table 3: Tabla de Promedios y Desviacion Estandar de los 3 lenguajes con Merge Sort

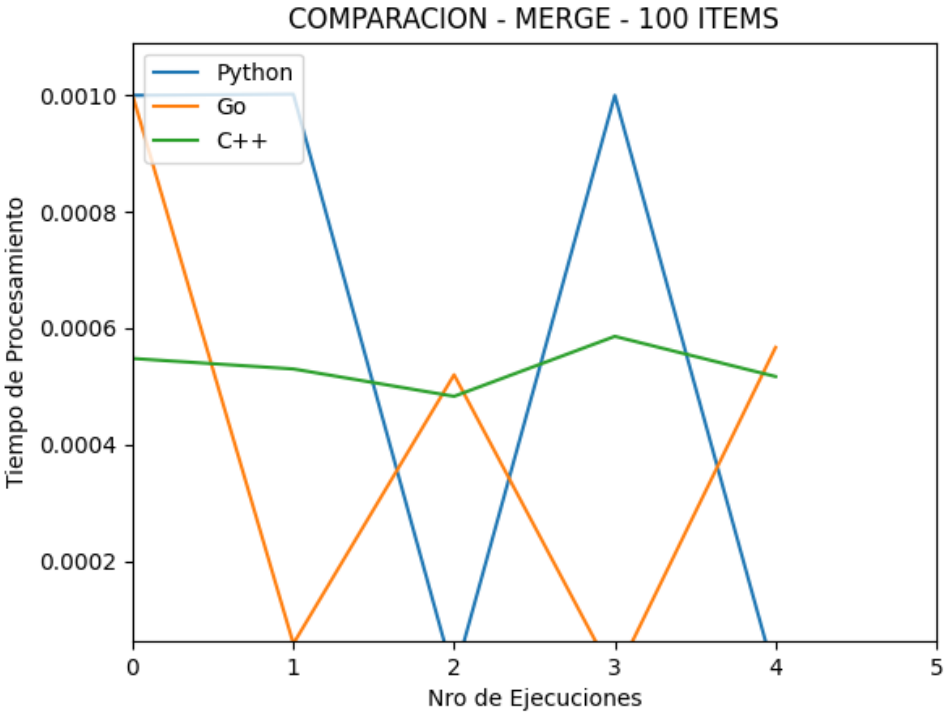


Figure 35: Merge Sort - 100 items

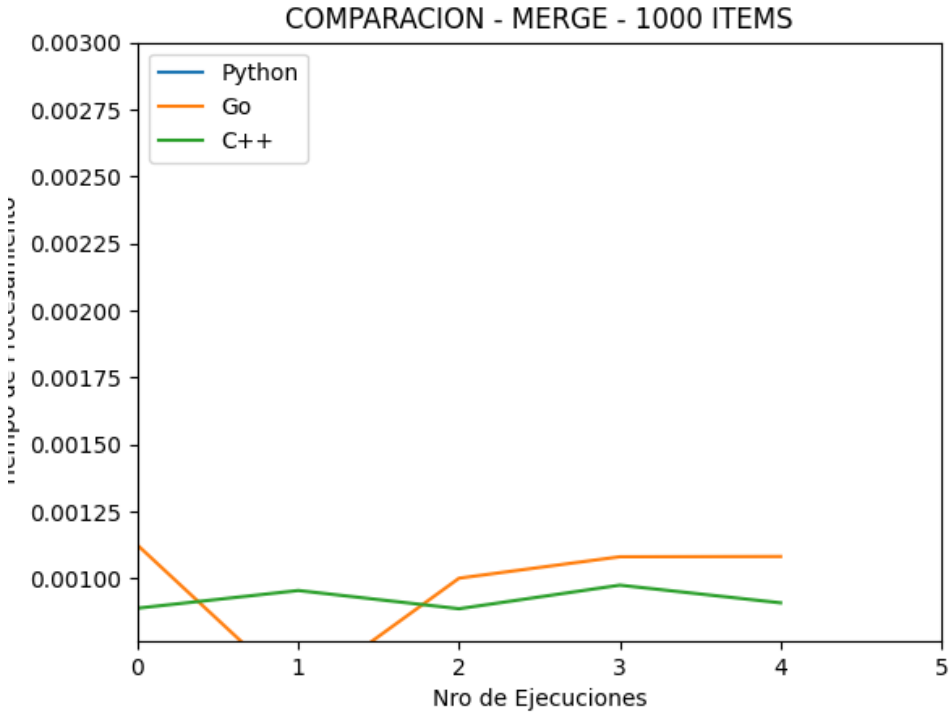


Figure 36: Merge Sort - 1000 items

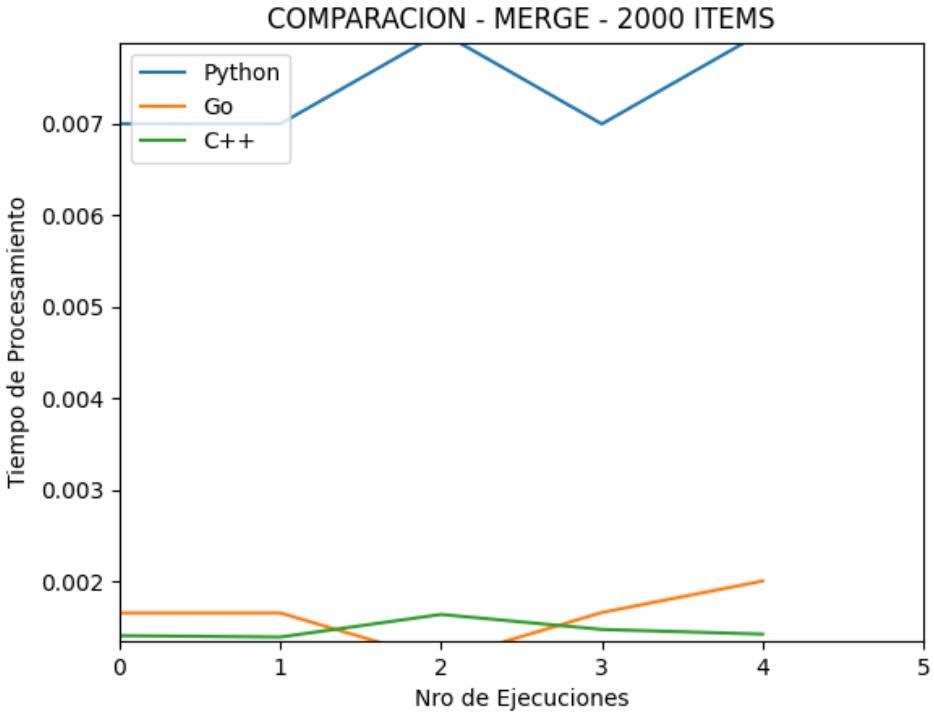


Figure 37: Merge Sort - 2000 items

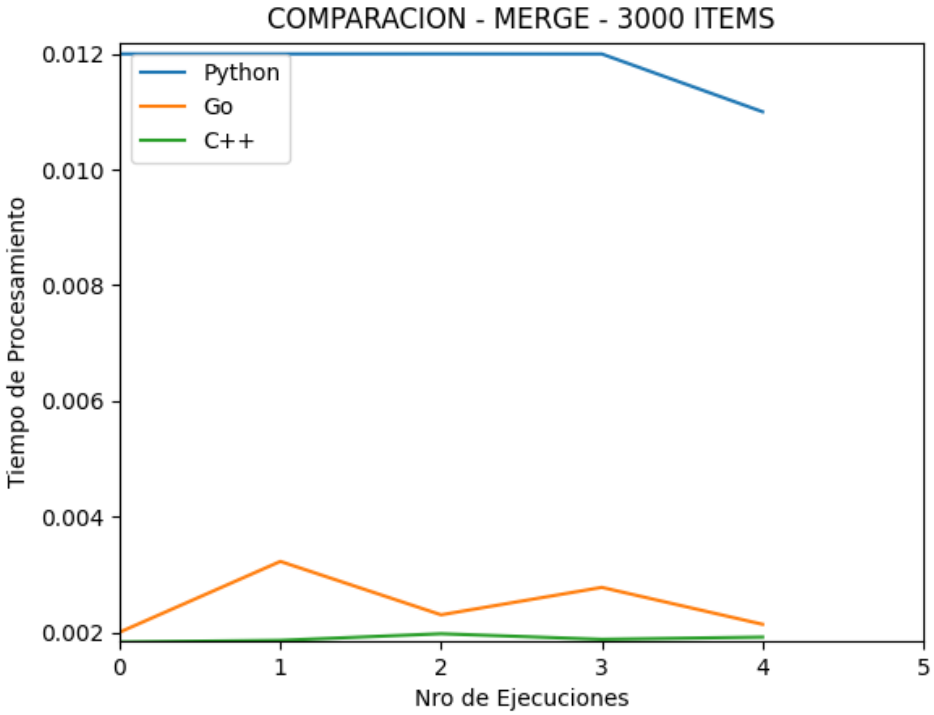


Figure 38: Merge Sort - 3000 items

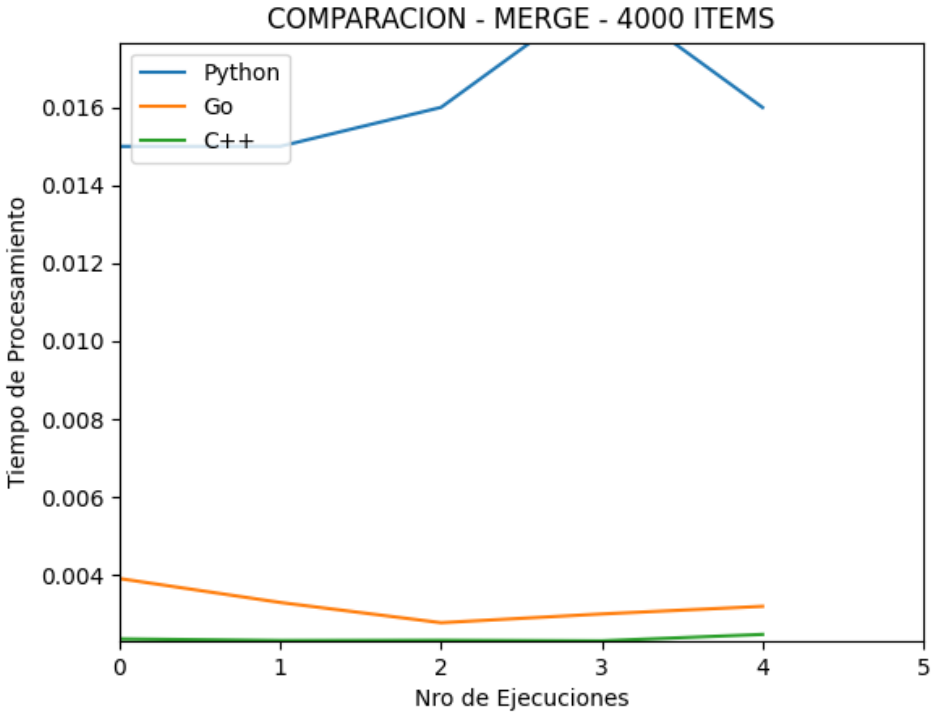


Figure 39: Merge Sort - 4000 items

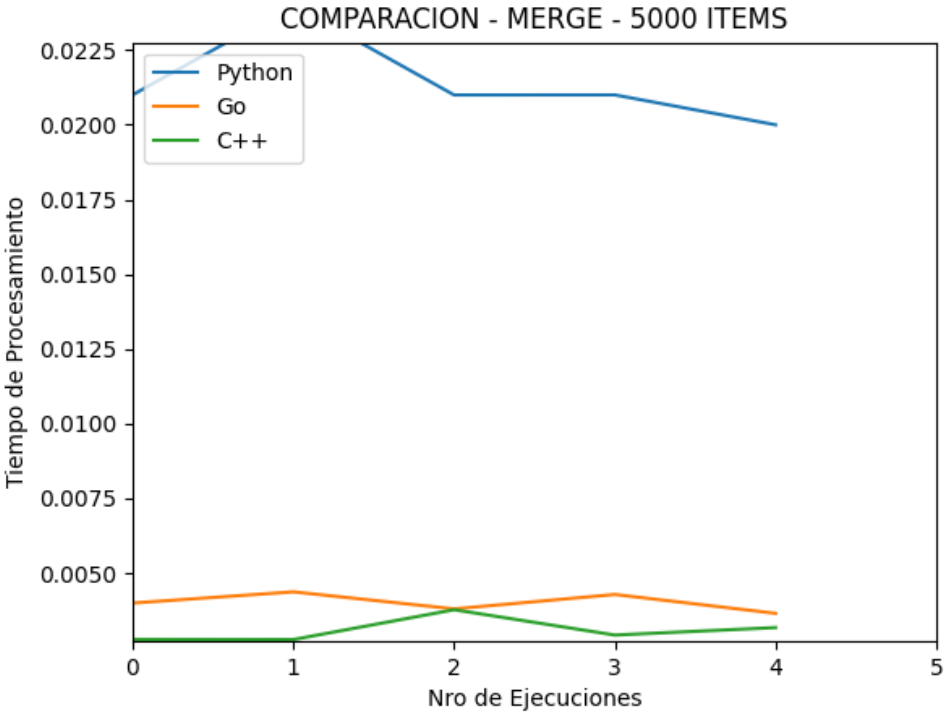


Figure 40: Merge Sort - 5000 items

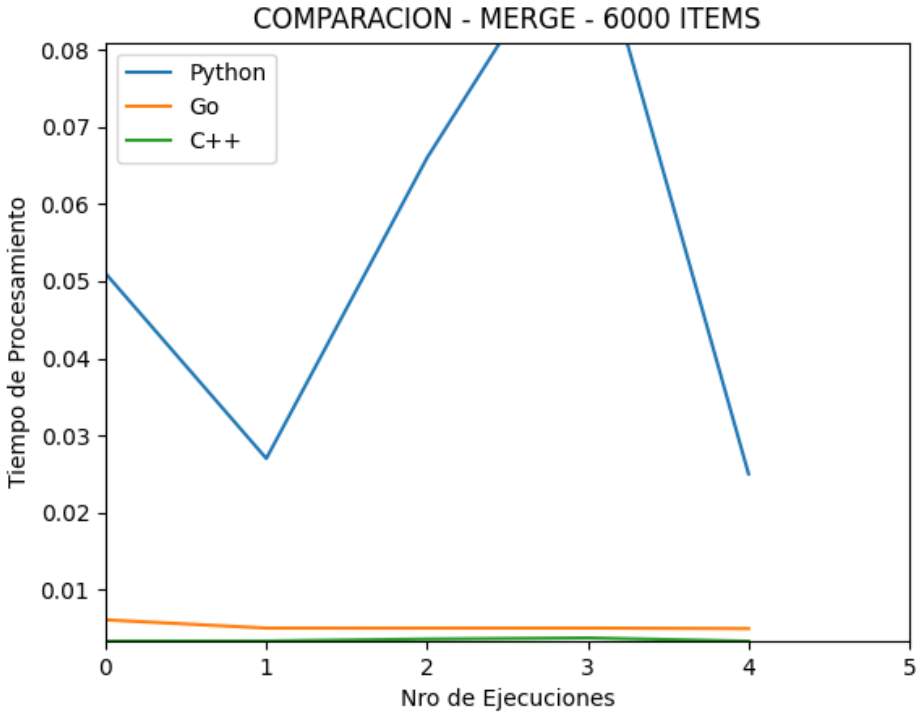


Figure 41: Merge Sort - 6000 items

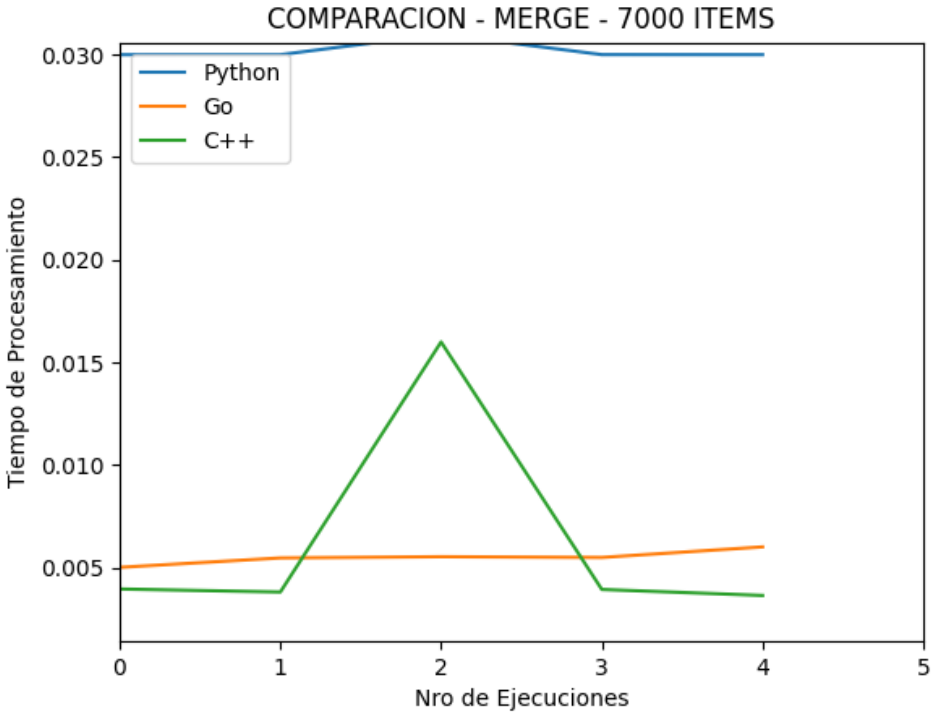


Figure 42: Merge Sort - 7000 items

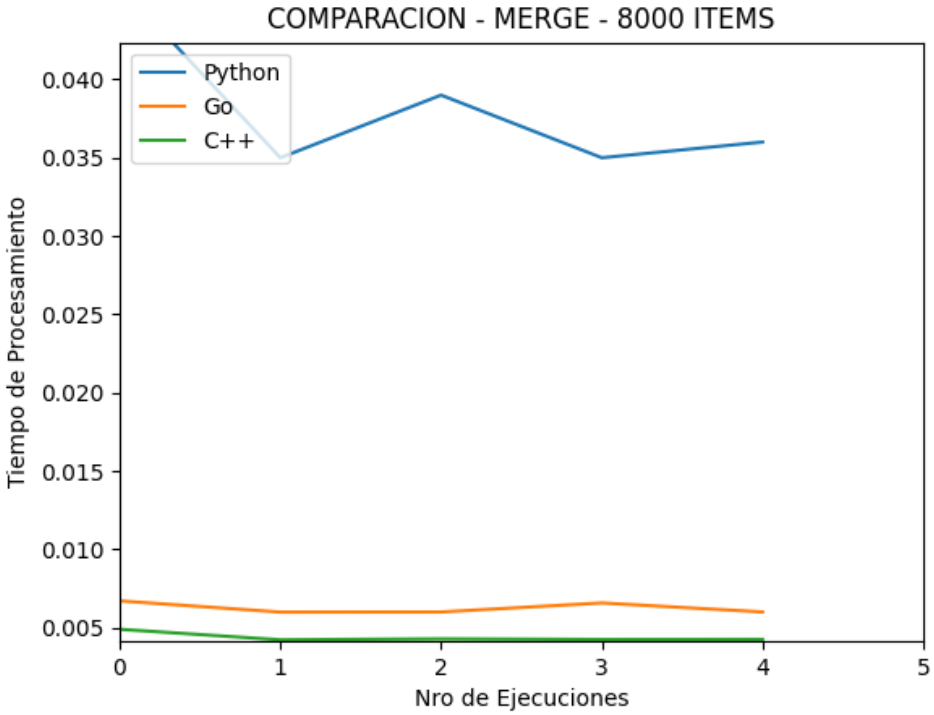


Figure 43: Merge Sort - 8000 items

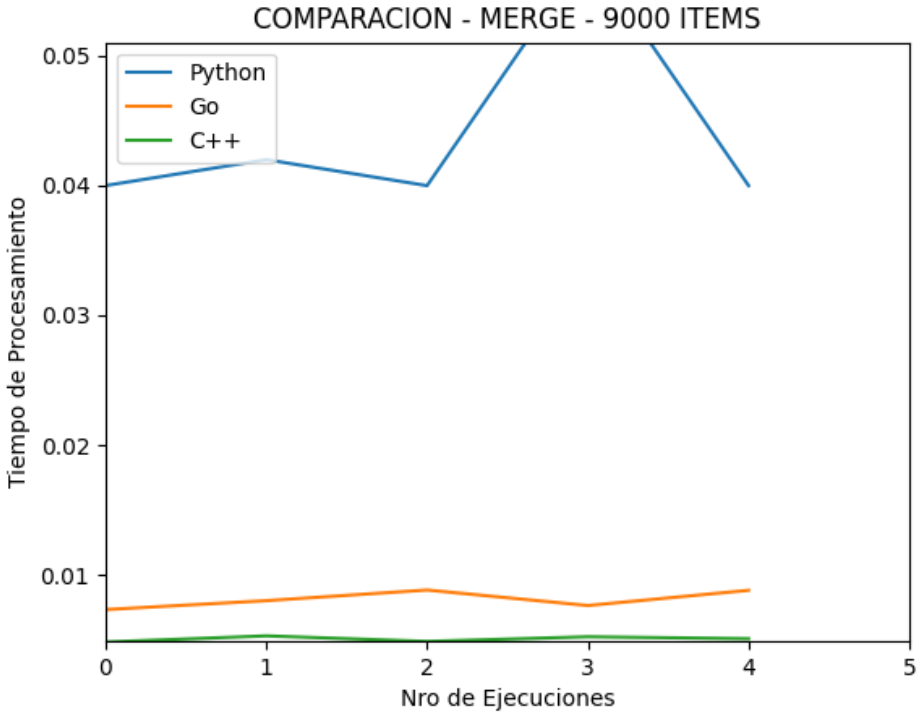


Figure 44: Merge Sort - 9000 items

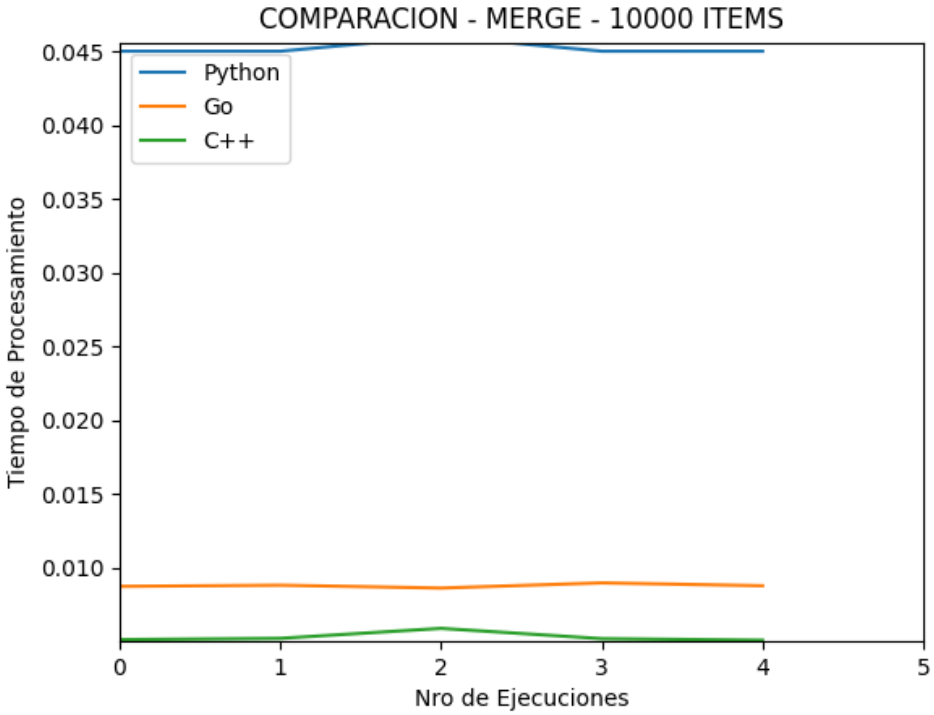


Figure 45: Merge Sort - 10000 items

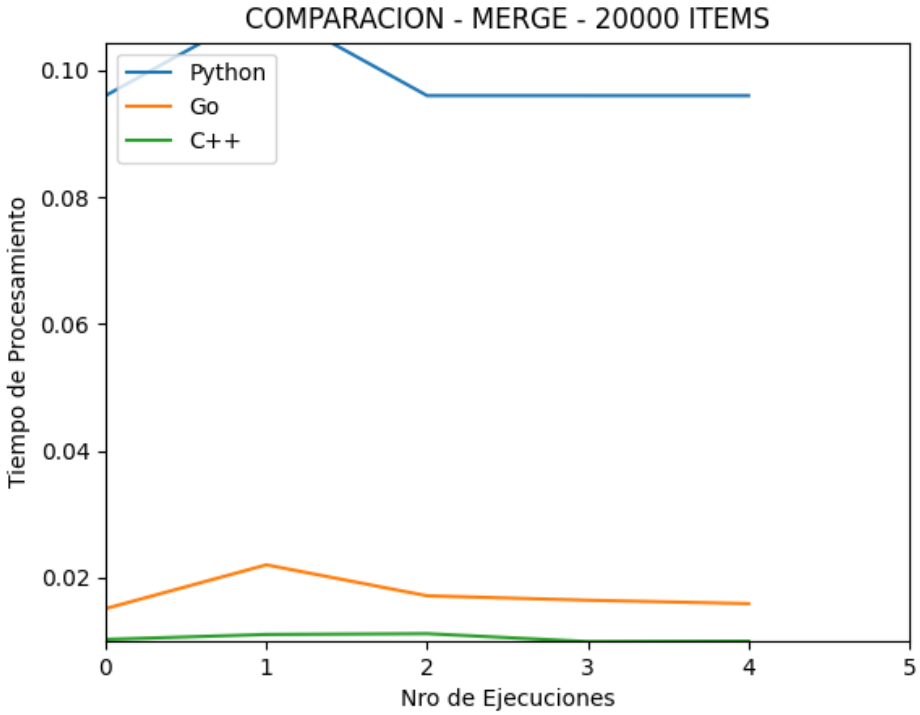


Figure 46: Merge Sort - 20000 items

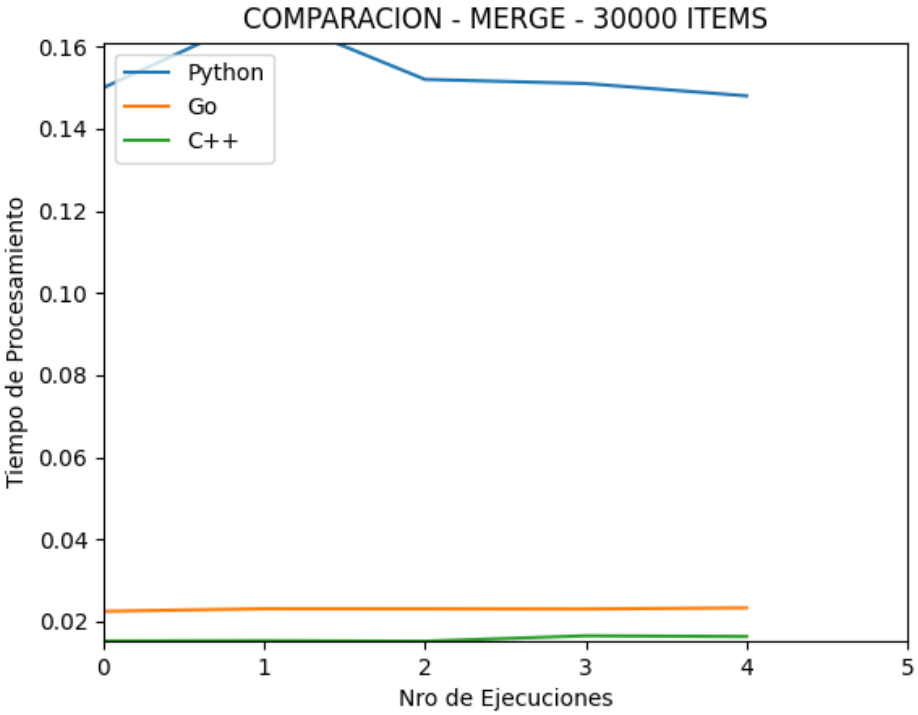


Figure 47: Merge Sort - 30000 items

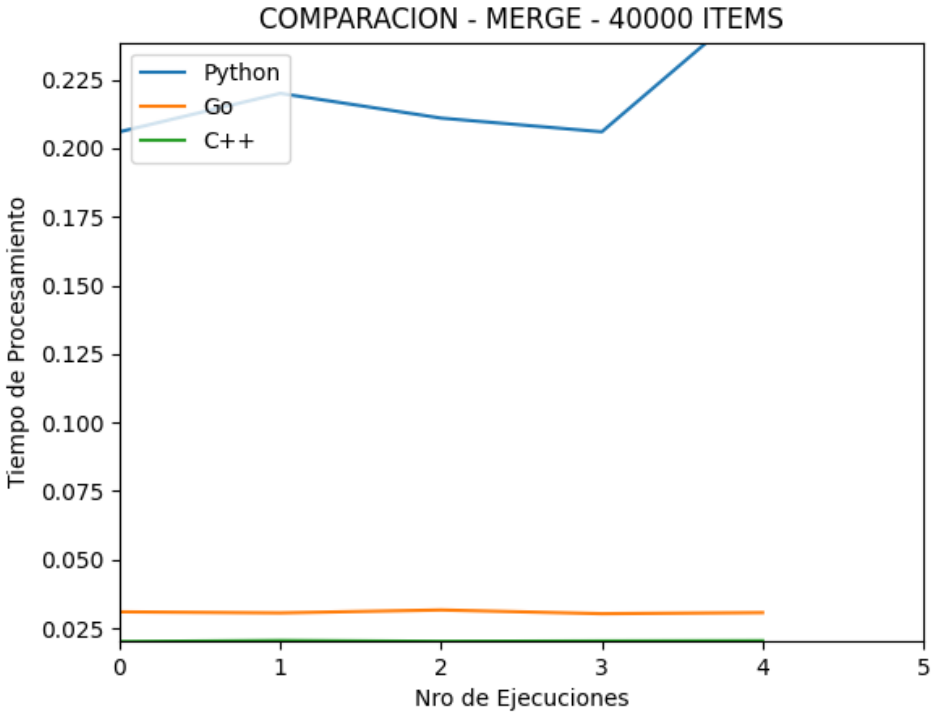


Figure 48: Merge Sort - 40000 items

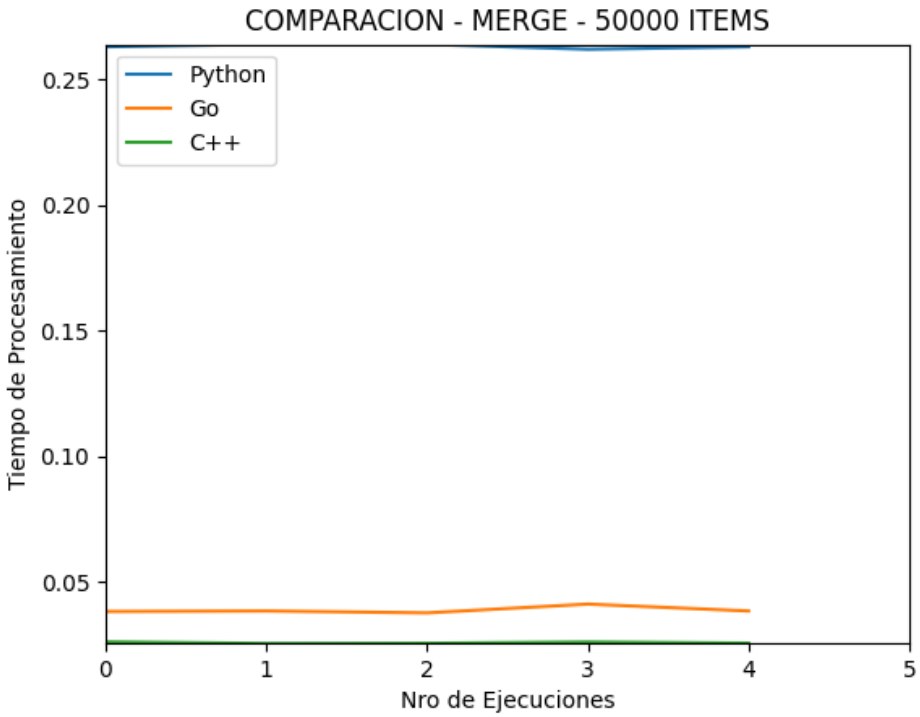


Figure 49: Merge Sort - 50000 items

QUICK SORT

DATOS	Python		C++		Golang	
Cantidad	Promedio	Desv. Est.	Promedio	Desv. Est.	Promedio	Desv. Est.
100	0.000993	0.000015	0.000505	0.000049	0.000233	0.000262
1000	0.0092	0.0064	0.00085	0.000052	0.000694	0.000202
2000	0.0176	0.00933	0.001574	0.000481	0.002188	0.001663
3000	0.020801	0.000748	0.001696	0.000126	0.001742	0.000207
4000	0.0284	0.0048	0.003398	0.002469	0.002316	0.000223
5000	0.04	0.012538	0.002429	0.000055	0.003177	0.00019
6000	0.041799	0.001166	0.002925	0.000124	0.003416	0.000159
7000	0.052201	0.003543	0.003394	0.000218	0.004177	0.000253
8000	0.056599	0.001356	0.003988	0.000487	0.004597	0.000297
9000	0.065	0.000895	0.00412	0.000181	0.005236	0.000265
10000	0.074599	0.0012	0.0047	0.00029	0.005903	0.000202
20000	0.1538	0.000749	0.009062	0.000274	0.011642	0.00019
30000	0.257399	0.024021	0.01318	0.000115	0.018196	0.000386
40000	0.341	0.022135	0.01858	0.001031	0.023219	0.000234
50000	0.431598	0.001743	0.022693	0.001548	0.030996	0.003827

Table 4: Tabla de Promedios y Desviacion Estandar de los 3 lenguajes con Quick Sort

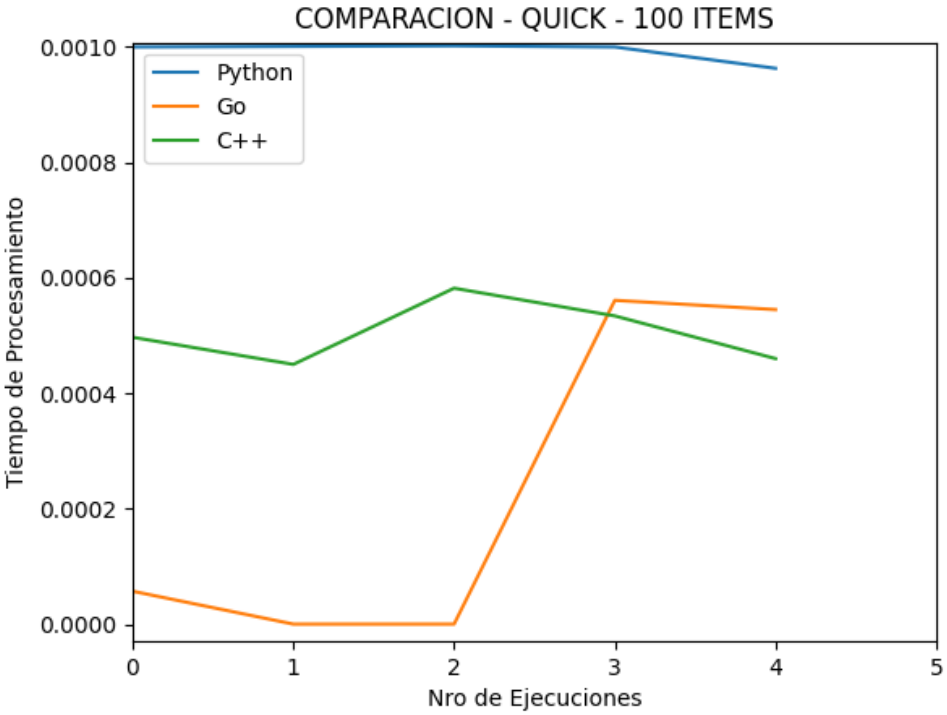


Figure 50: Quick Sort - 100 items

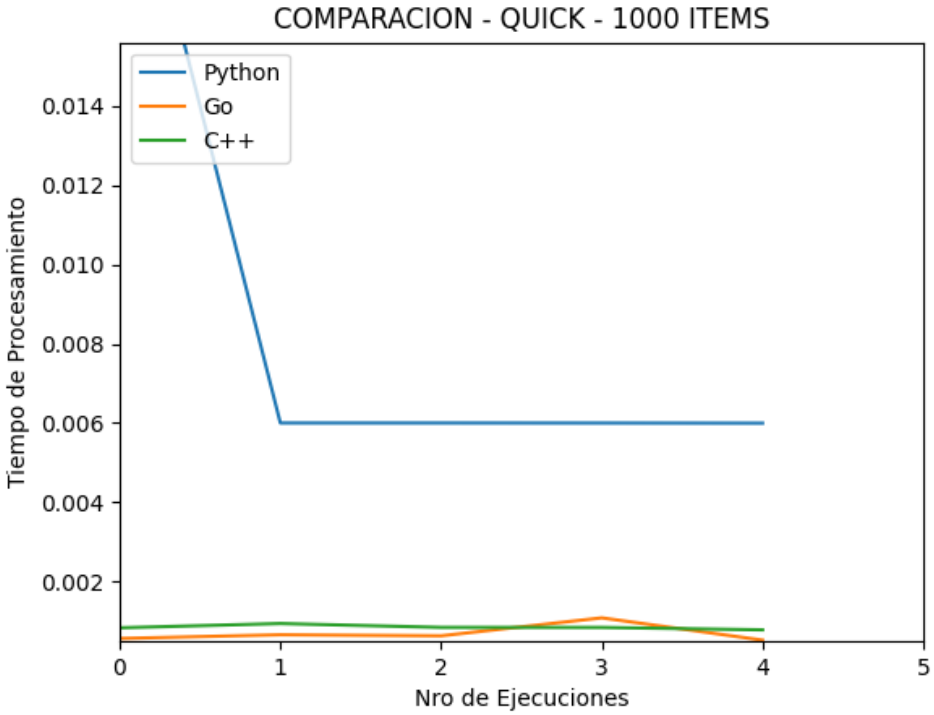


Figure 51: Quick Sort - 1000 items

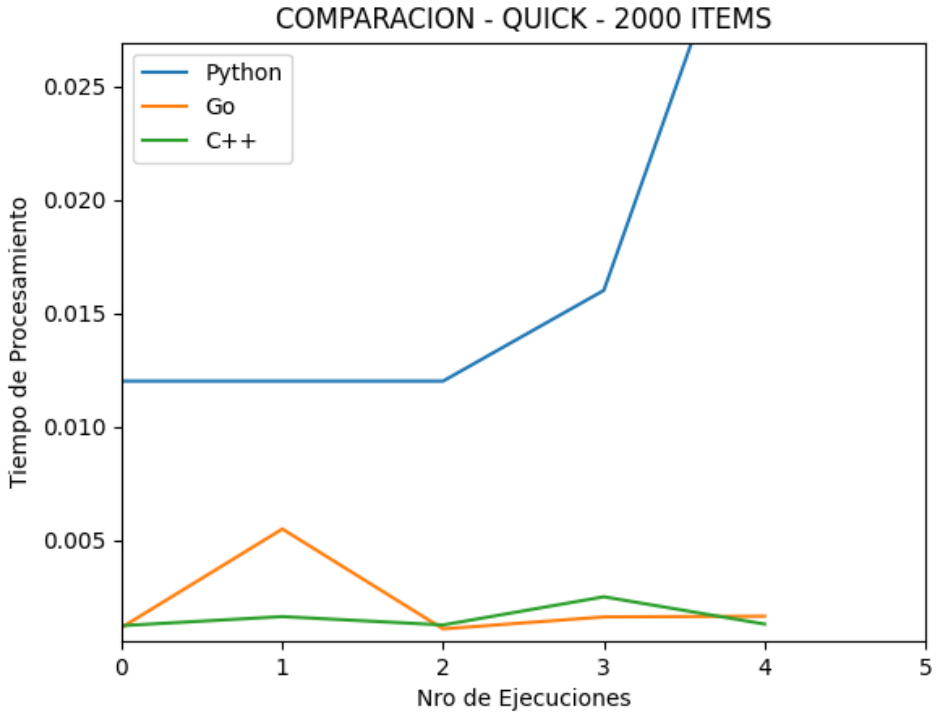


Figure 52: Quick Sort - 2000 items

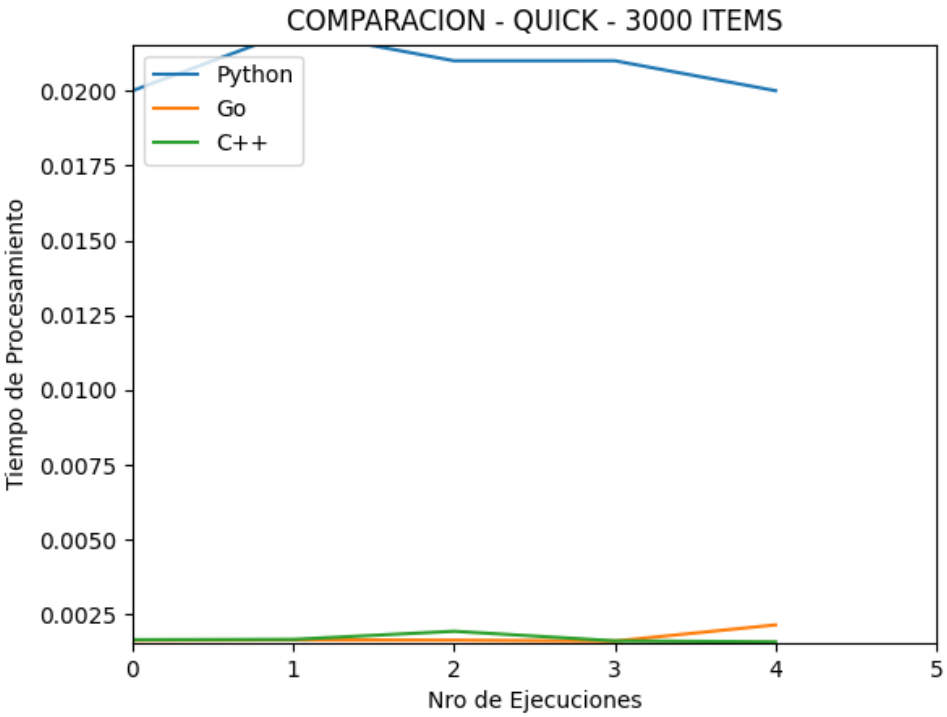


Figure 53: Quick Sort - 3000 items

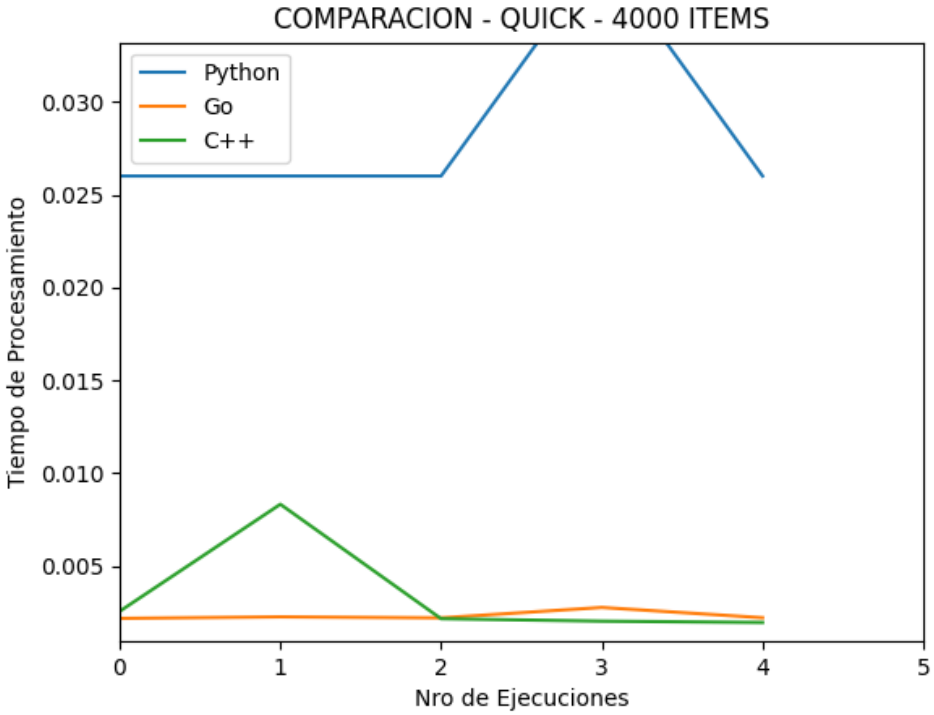


Figure 54: Quick Sort - 4000 items

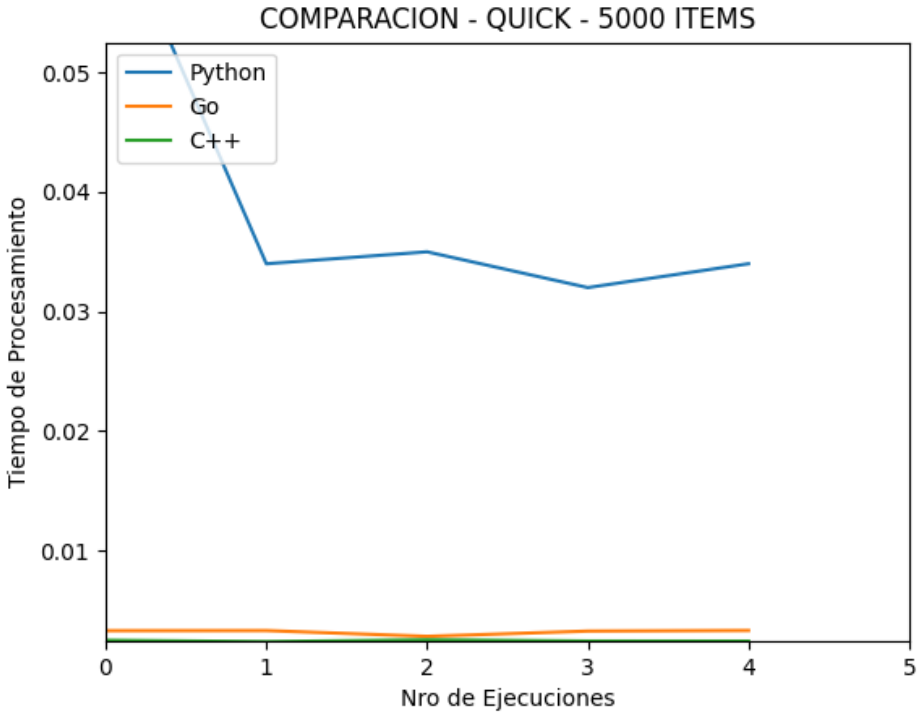


Figure 55: Quick Sort - 5000 items

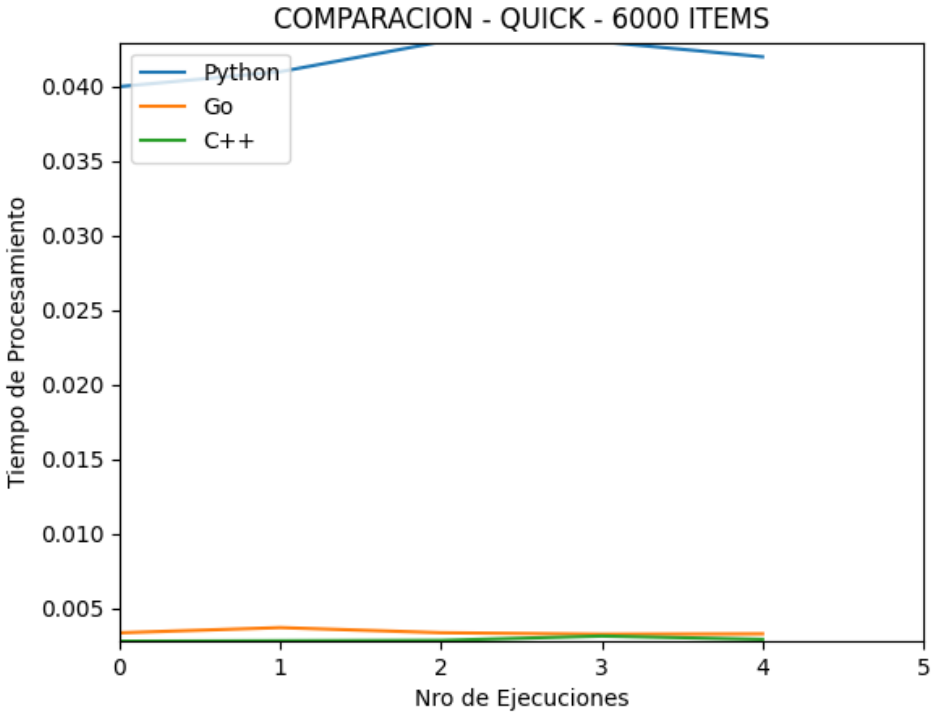


Figure 56: Quick Sort - 6000 items

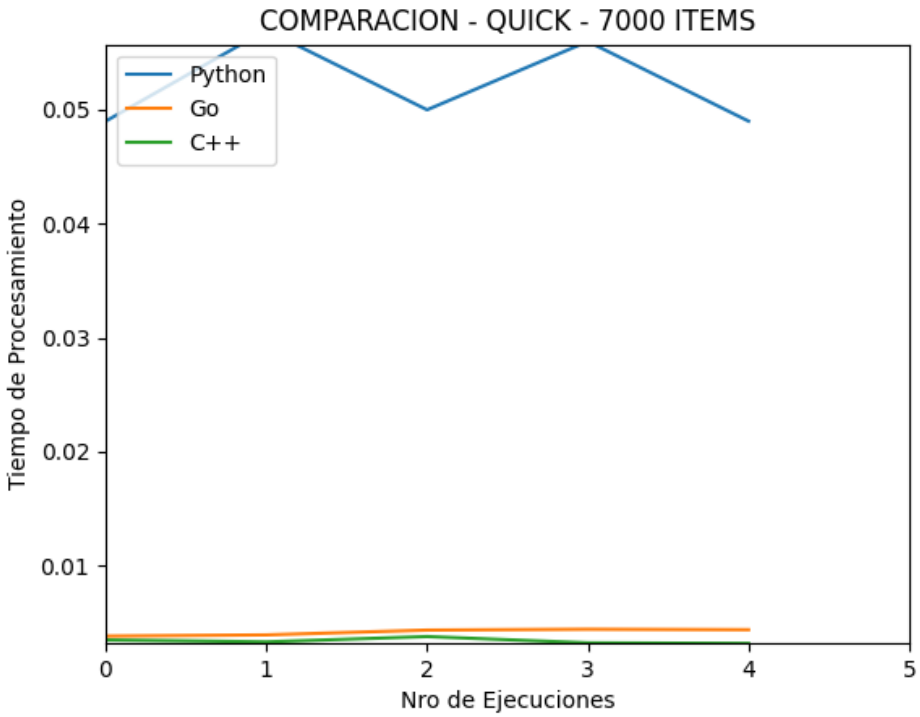


Figure 57: Quick Sort - 7000 items

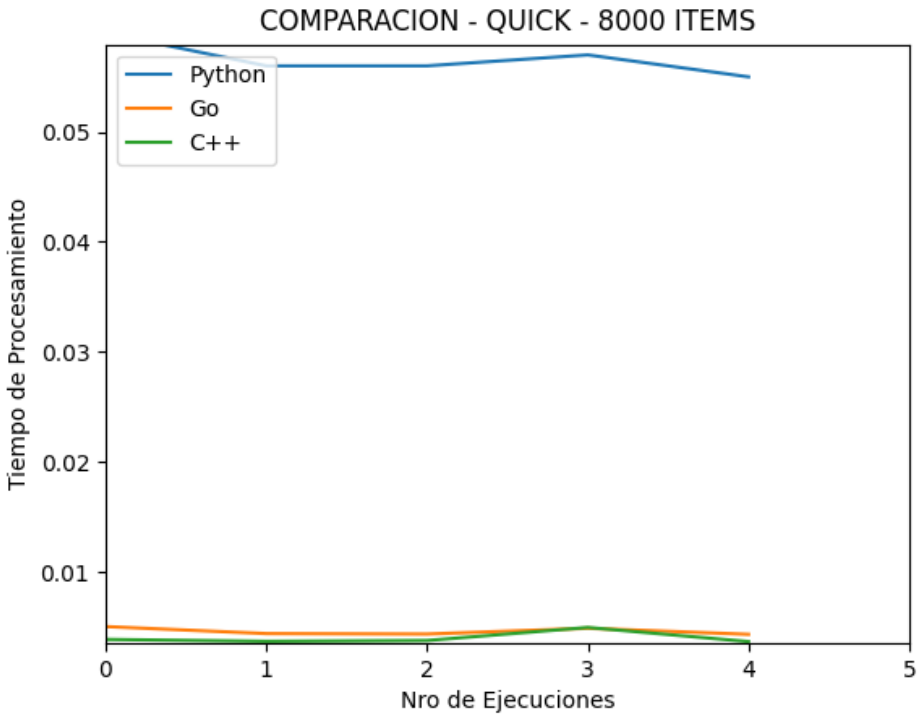


Figure 58: Quick Sort - 8000 items

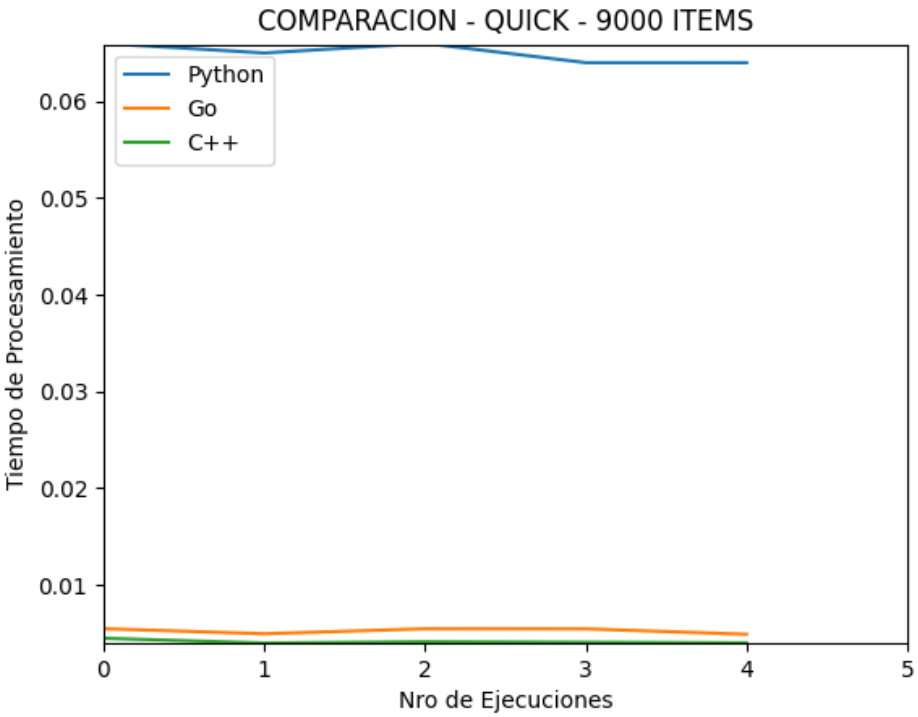


Figure 59: Quick Sort - 9000 items

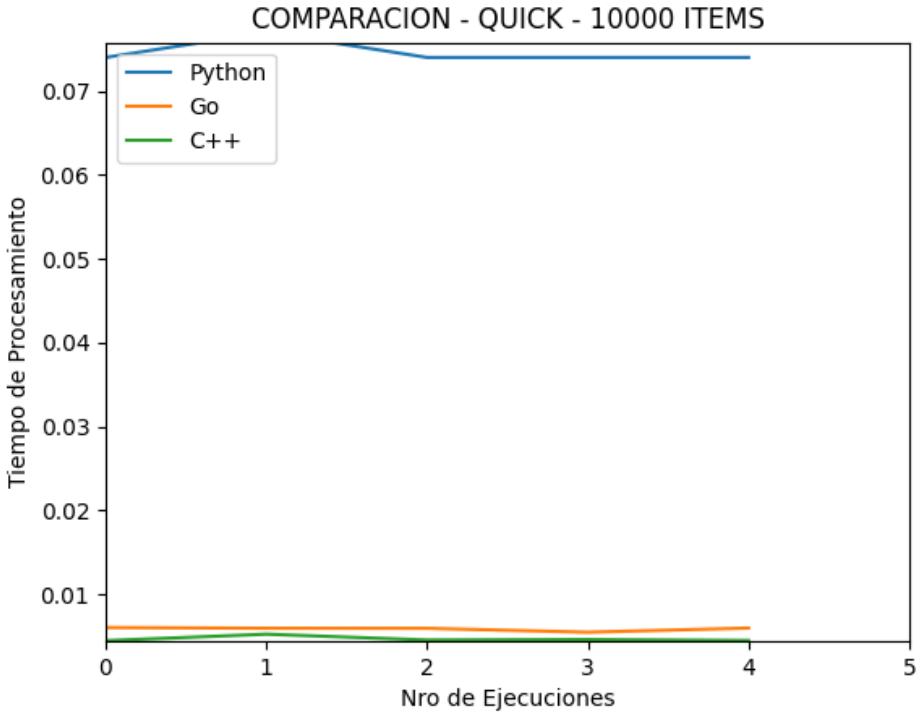


Figure 60: Quick Sort - 10000 items

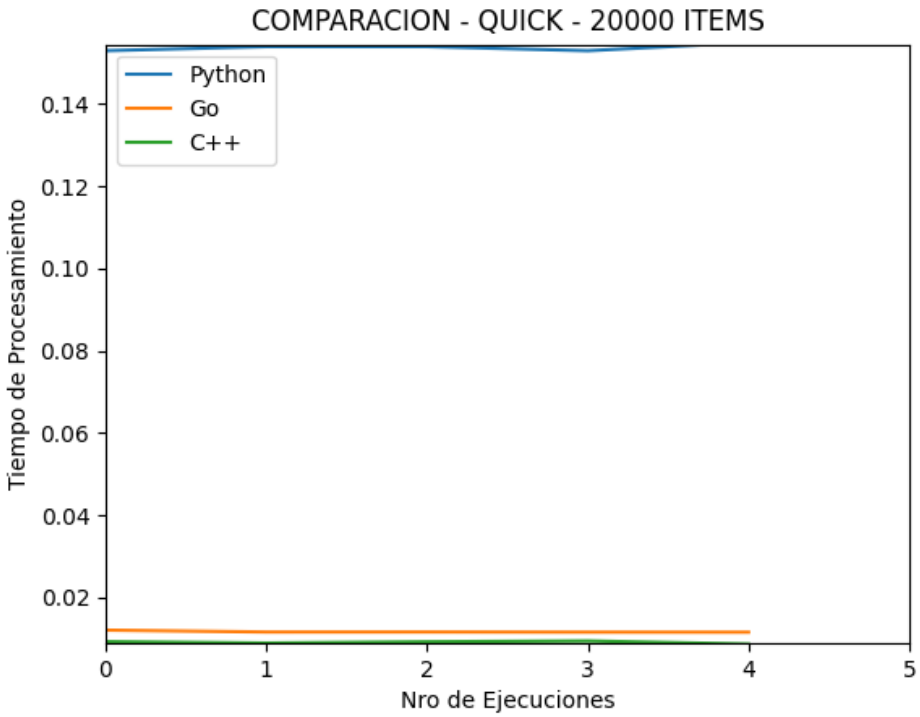


Figure 61: Quick Sort - 20000 items

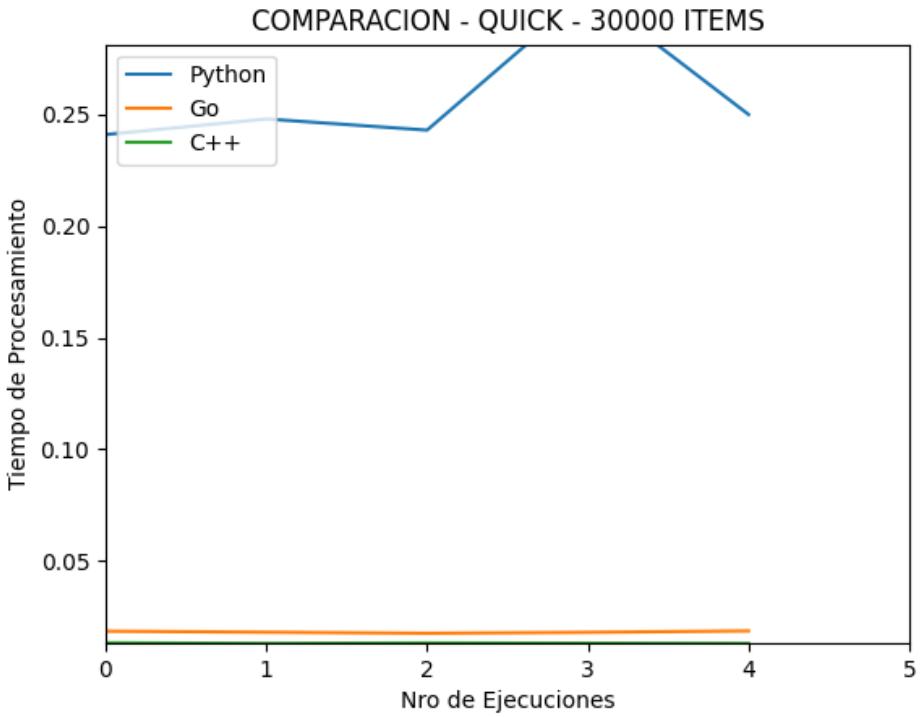


Figure 62: Quick Sort - 30000 items

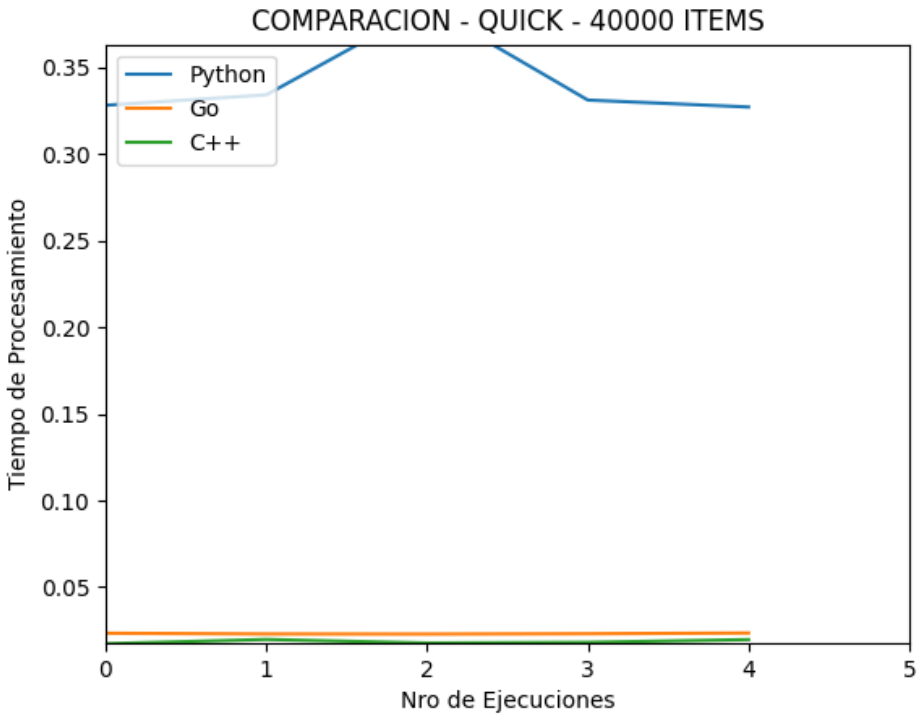


Figure 63: Quick Sort - 40000 items

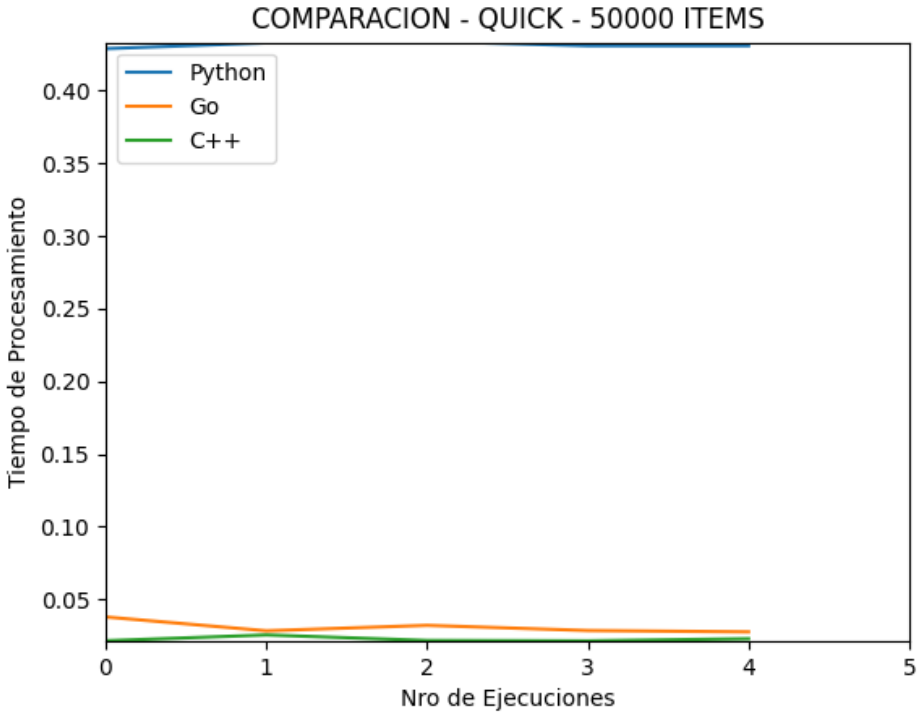


Figure 64: Quick Sort - 50000 items

SELECTION SORT

DATOS	Python		C++		Golang	
Cantidad	Promedio	Desv. Est.	Promedio	Desv. Est.	Promedio	Desv. Est.
100	0.001999	0.000001	0.000531	0.000029	0.000422	0.000212
1000	0.201399	0.066018	0.006025	0.008607	0.001102	0.000024
2000	0.6088	0.014274	0.008384	0.005724	0.004712	0.002605
3000	1.3838	0.049297	0.017627	0.013963	0.008327	0.000347
4000	2.465399	0.056237	0.02121	0.006782	0.015216	0.000272
5000	3.838001	0.071756	0.027197	0.000474	0.023363	0.000289
6000	5.5782	0.167367	0.039137	0.00143	0.039792	0.006679
7000	7.695804	0.399161	0.051664	0.000391	0.05184	0.000344
8000	10.266799	0.444722	0.066706	0.000557	0.074552	0.003197
9000	12.484799	0.234154	0.083624	0.000264	0.099216	0.012354
10000	15.568	0.241852	0.10276	0.000199	0.114273	0.000478
20000	62.645672	1.58434	0.403848	0.00289	0.516882	0.005456
30000	143.626945	5.320718	0.902256	0.003313	1.176029	0.01479
40000	259.757819	13.970079	1.652836	0.04999	2.139661	0.048178
50000	385.468399	3.933536	2.551786	0.093821	3.354922	0.037221

Table 5: Tabla de Promedios y Desviacion Estandar de los 3 lenguajes con Selection Sort

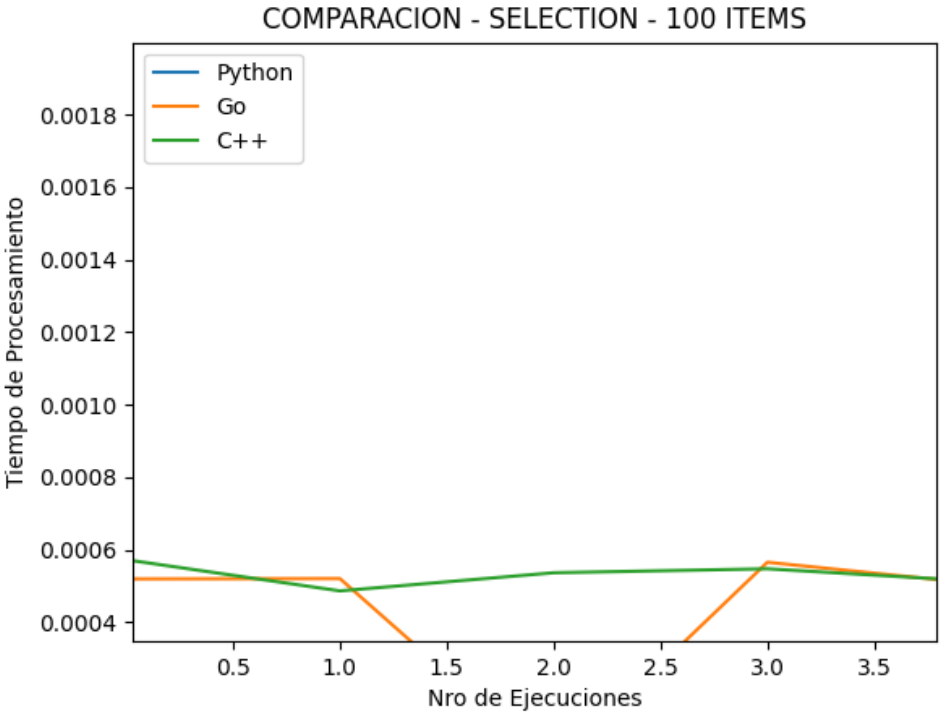


Figure 65: Selection Sort - 100 items

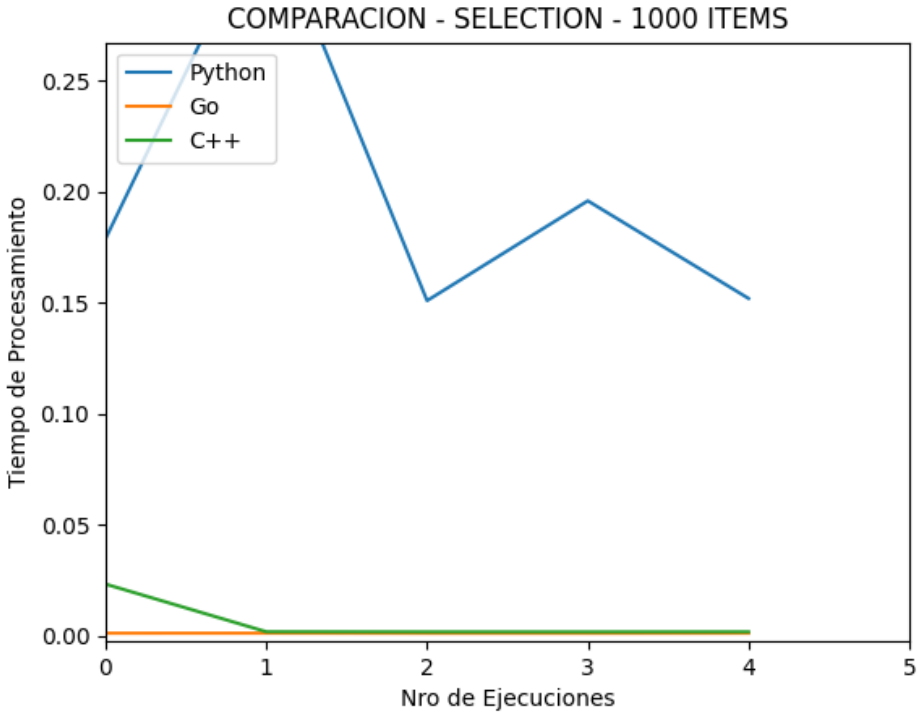


Figure 66: Selection Sort - 1000 items

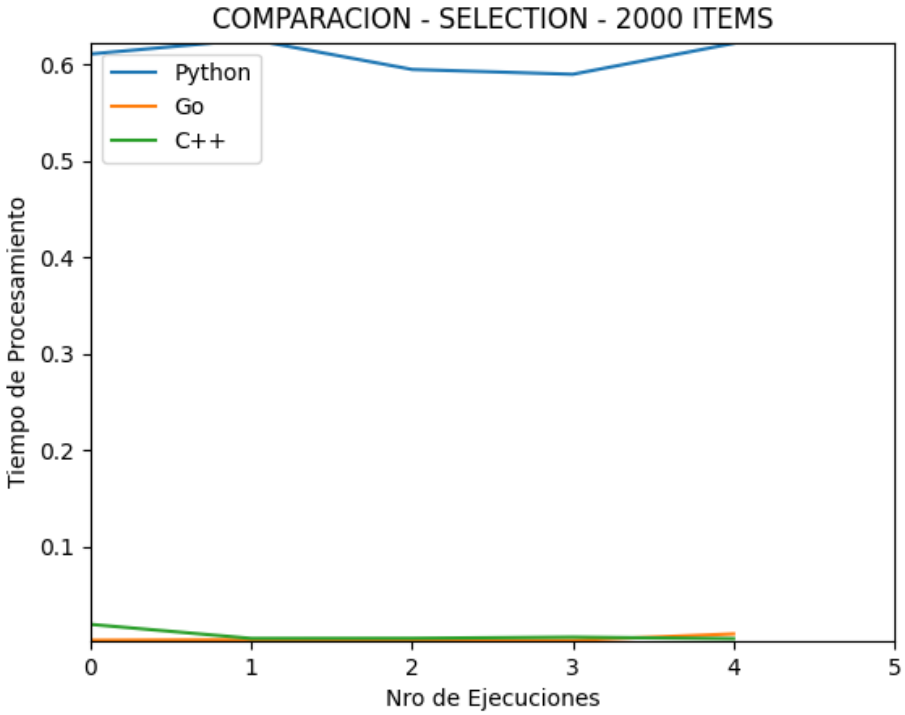


Figure 67: Selection Sort - 2000 items

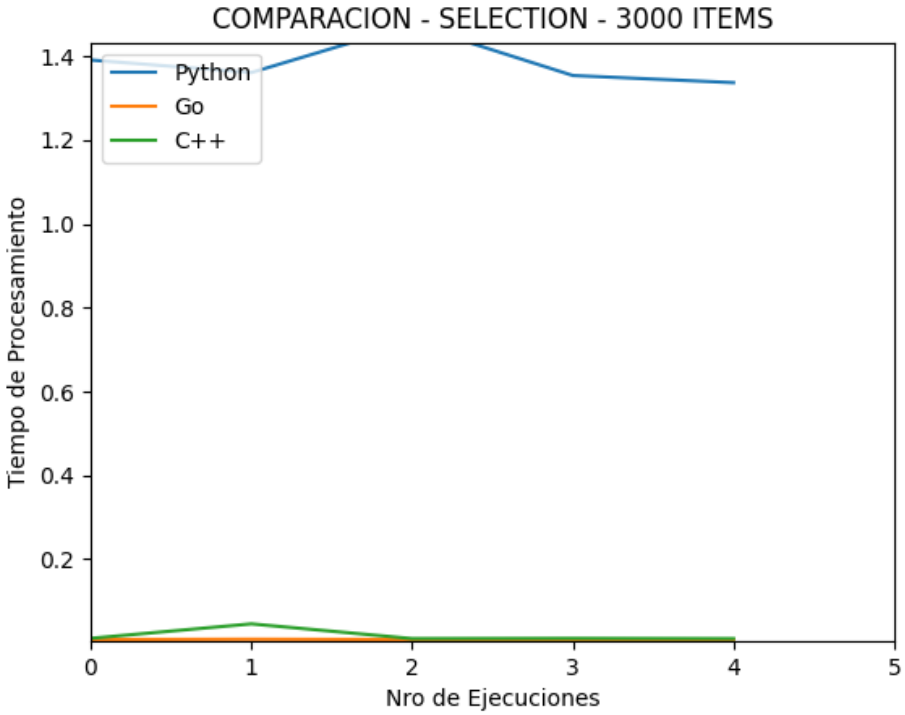


Figure 68: Selection Sort - 3000 items

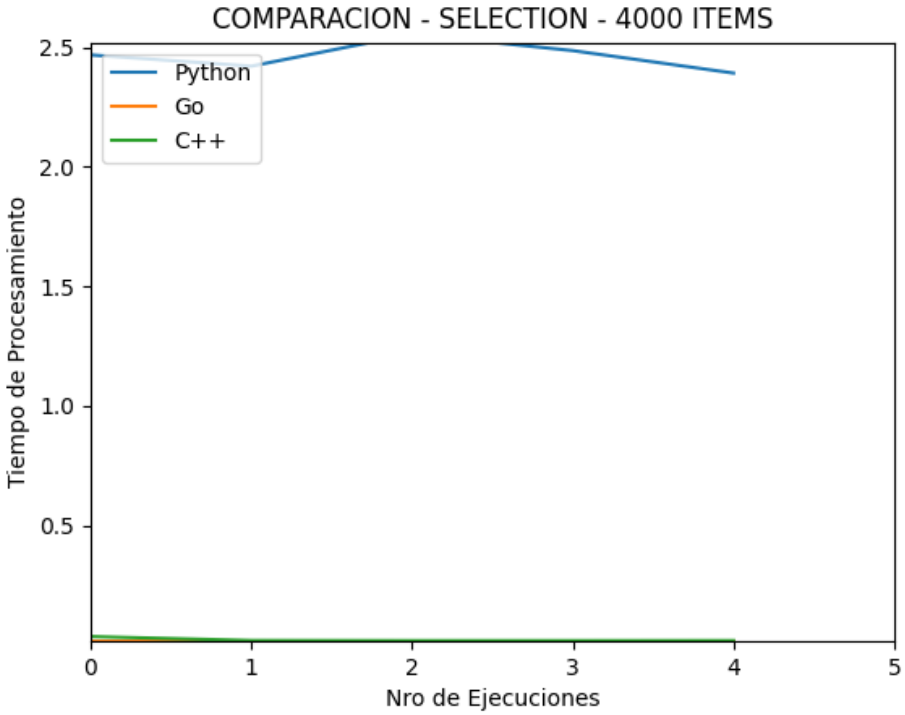


Figure 69: Selection Sort - 4000 items

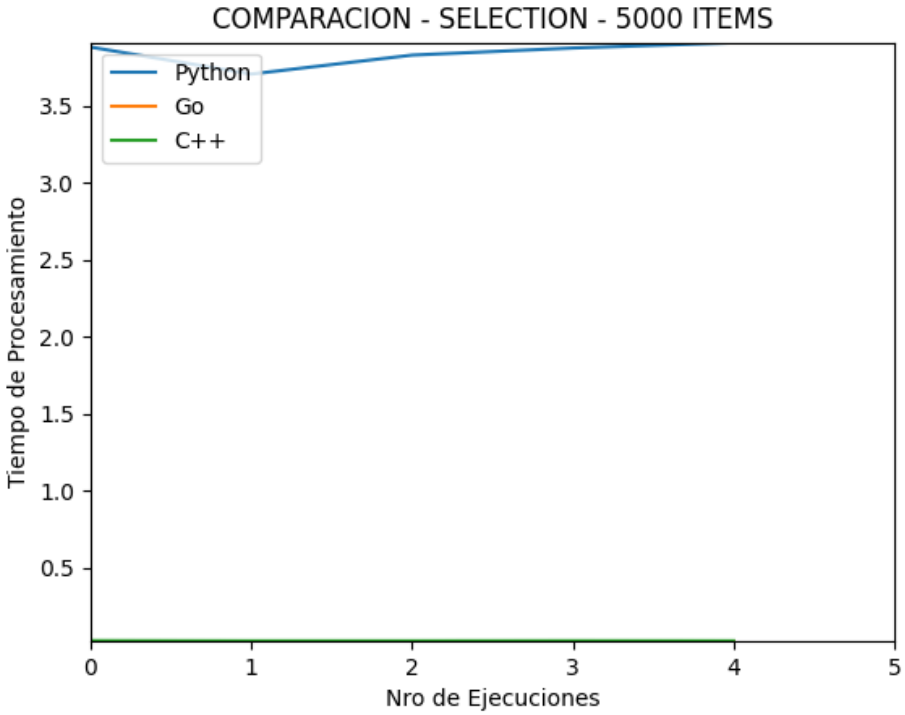


Figure 70: Selection Sort - 5000 items

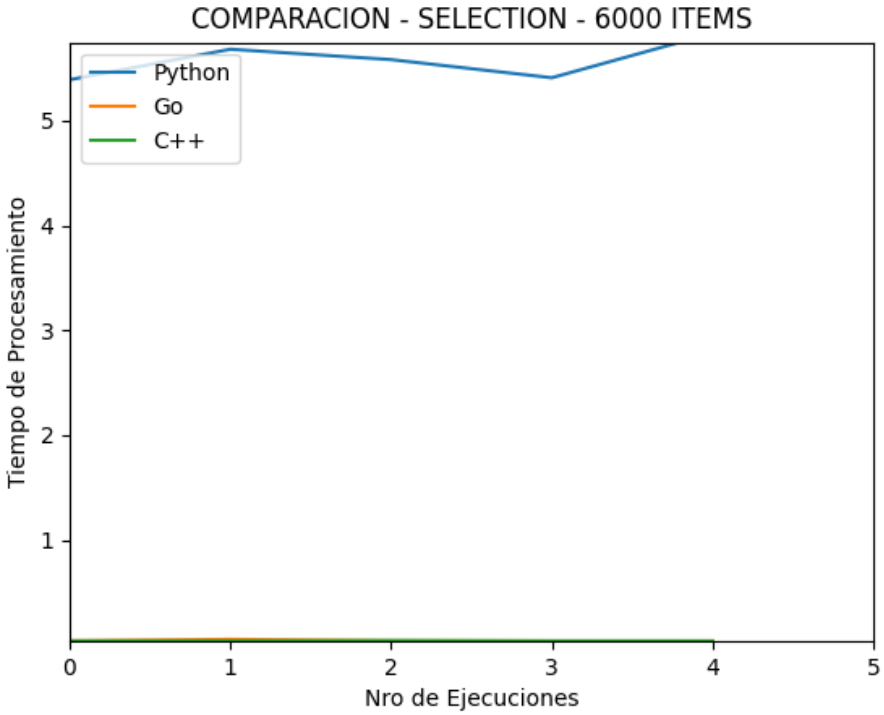


Figure 71: Selection Sort - 6000 items

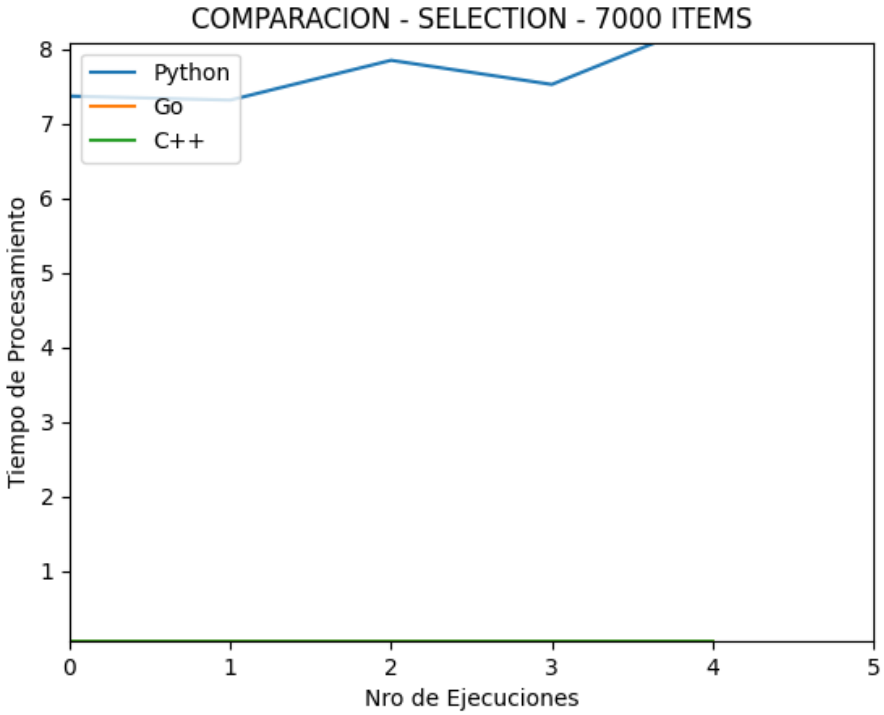


Figure 72: Selection Sort - 7000 items

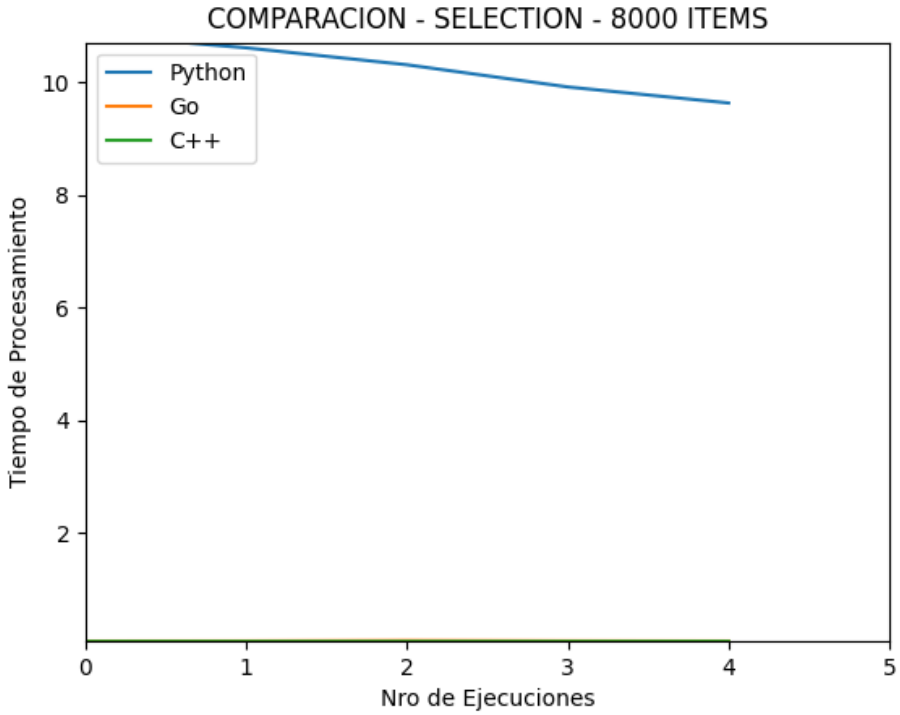


Figure 73: Selection Sort - 8000 items

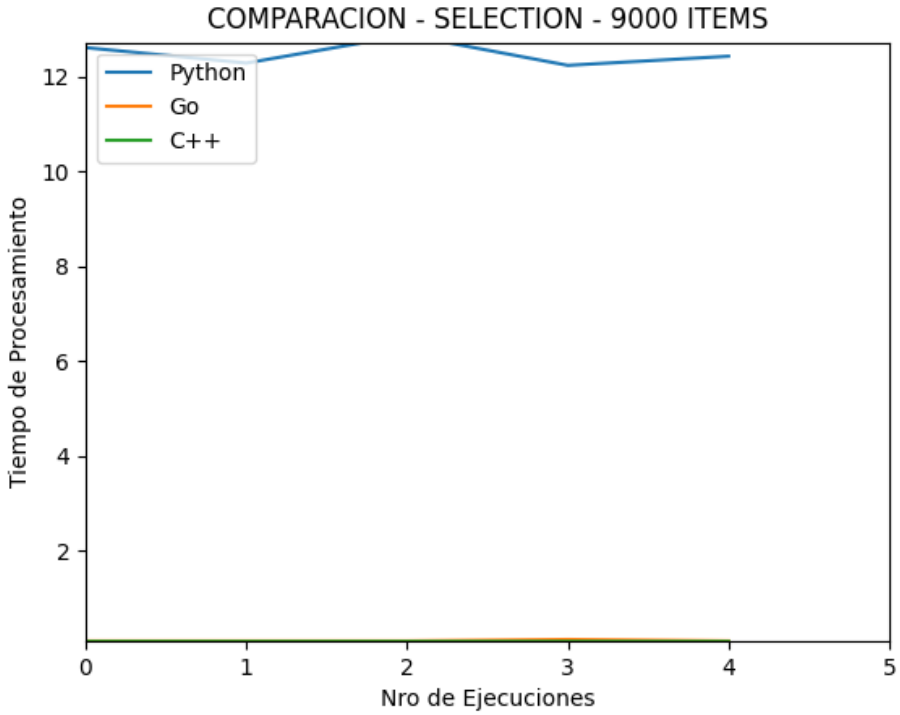


Figure 74: Selection Sort - 9000 items

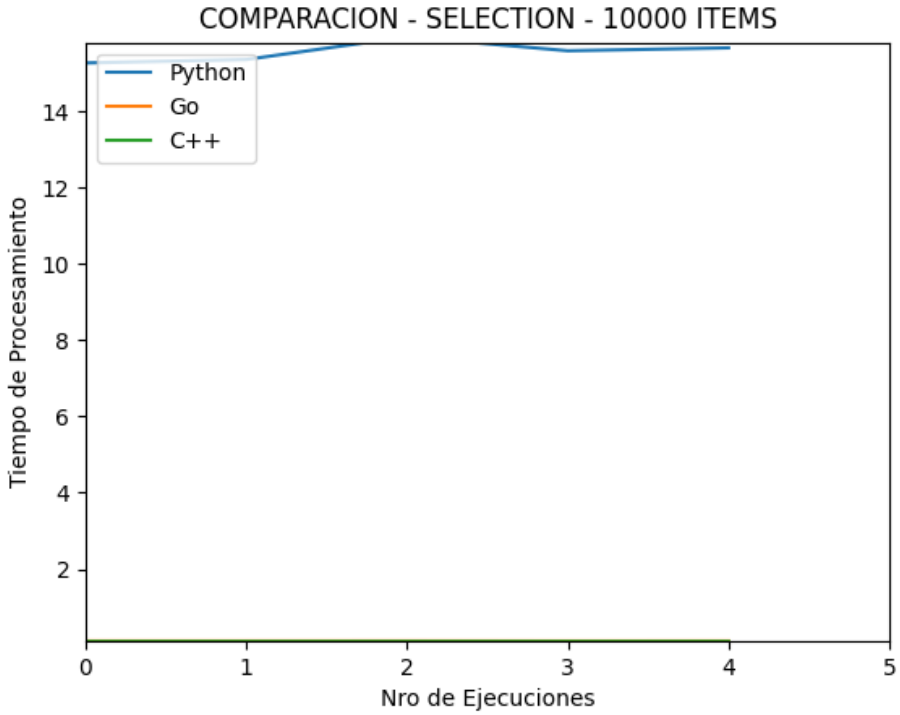


Figure 75: Selection Sort - 10000 items

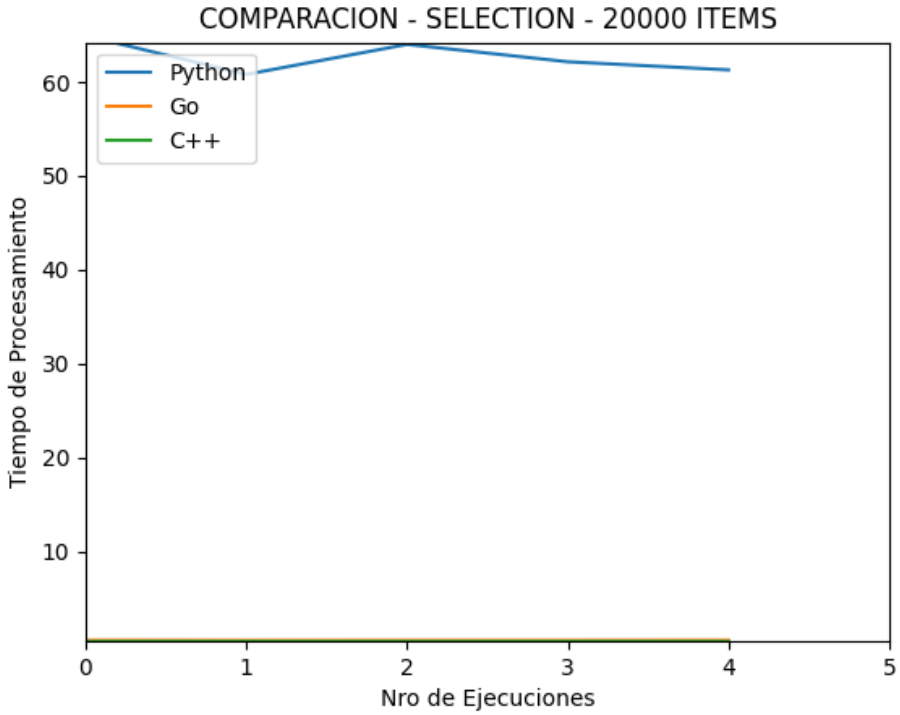


Figure 76: Selection Sort - 20000 items

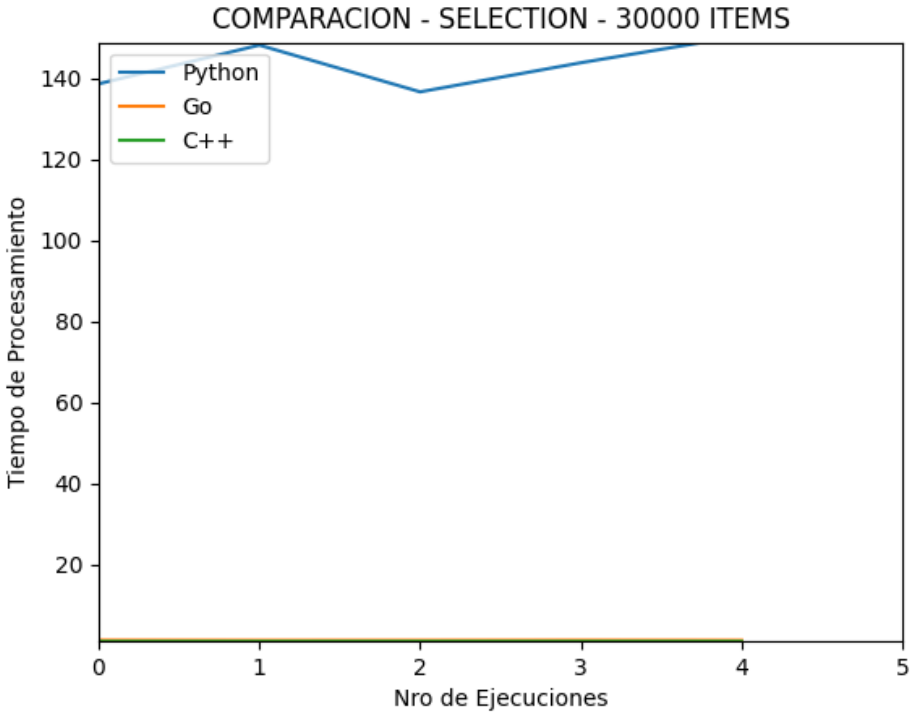


Figure 77: Selection Sort - 30000 items

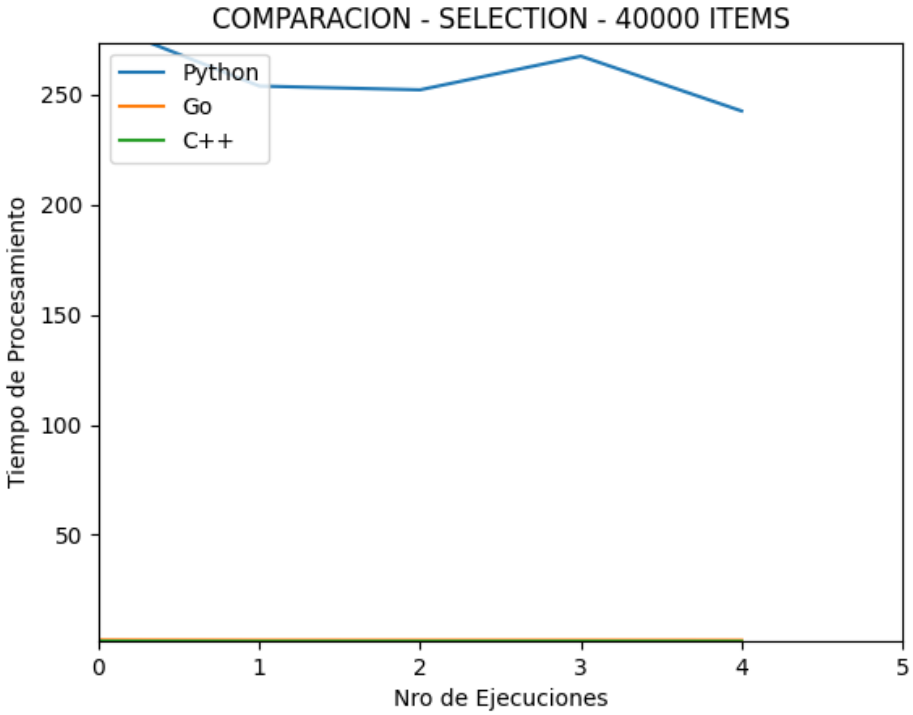


Figure 78: Selection Sort - 40000 items

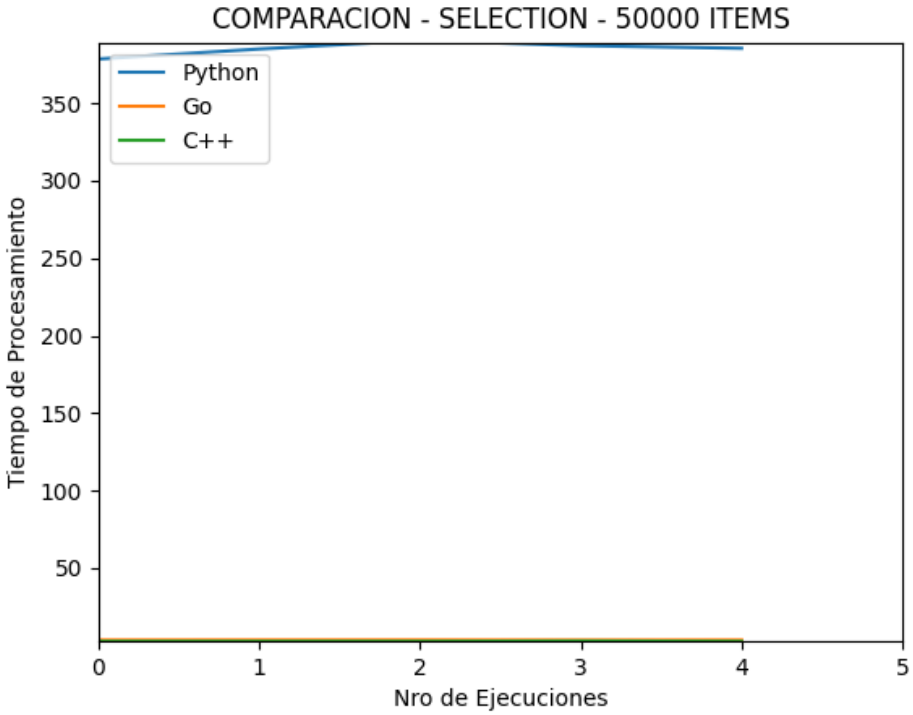


Figure 79: Selection Sort - 50000 items

Como se puede ver en las tablas comparativas, Python llego a necesitar mas de 300 seg para la ejecucion de 50000 datos, sobre todo con algoritmos como BubbleSort y SelectionSort, siendo los de mas complejidad, Tambien podemos observar que Go y C++ tienen muy similares los tiempos en promedio, siendo los lenguajes mas rapidos incluso en los 2 algoritmos mencionados anteriormente.

5 Conclusiones

Como conclusiones del informe podemos indicar lo siguiente:

- En la comparativa de los 5 algoritmos seleccionados, se concluye que quicksort y mergesort son los algoritmos mas rapidos, binary insertion sort seria el que les sigue y bubblesort y selectionsort serian los mas lentos segun el tamaño de las listas comparadas
- En la comparativa de los 3 lenguajes de programacion: Python, C++ y Golang, Python seria el lenguaje mas lento, mostrando altos tiempos de ejecucion en listas de mayor tamaño, mientras que C++ y Golang serian mas rapidos, manteniendo un margen no mayor a 3 seg incluso en listas de gran tamaño