Índice

Índice

Introducción

El análisis de algoritmos puede entenderse como la estimación del consumo de recursos que un algoritmo requiere, proporcionando herramientas para poder estimar si una solución propuesta satisface las restricciones de recursos de un problema sin necesidad de implementarlo.

En la práctica 1 vamos a realizar un análisis cuatro algoritmos de ordenamiento con tres tipos de programación, obteniendo de esta manera un cuadro comparativo, del cual a partir de ello evaluaremos que tipo de lenguaje es mucho mas funcional respecto al algoritmo probado, para tal motivo se esta utilizando el mismo ordenador, asi como editor de texto para poder mantener las mismas condiciones para los diversos tipos de lenguaje.

Índice

Merge sort

Algoritmo basado en la técnica DyV

- Divide el vector en dos partes iguales.
- Ordena por separado cada una de las partes (llamando recursivamente a ordenaPorFusión).
- mezcla ambas partes manteniendo la ordenación.

Merge sort

Algoritmo Merge Sort

dividir cada elemento en particiones de tamaño 1
fusionar recursivamente particiones adyacentes
for i = leftPartIdx to rightPartIdx
if leftPartHeadValue ¡= rightPartHeadValue
copy leftPartHeadValue
else: copy rightPartHeadValue; Increase InvIdx
copiar elementos de nuevo a la matriz original

Merge sort

Costo Computacional

La longitud de la Lista es N

Dos listas N/2

El tiempo a*N

Suposición $N = 2^k$

Cuando la lista es pequeña $T(1) = T(0) = b$

$T(N) = 2 * T(N/2) + a * N$

$k = log2N$

En consecuencia

$$T(N) \equiv b * N + a * N * Log2N \quad (1)$$

Quick sort

Es un algoritmo DyV muy parecido al de la selección (búsqueda del k-ésimo menor elemento):

- se reorganiza la tabla en dos subtablas respecto a un pivote: elementos mayores o iguales a un lado y menores al otro, después de la reorganización, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada
- se repite el proceso de forma recursiva para cada subtabla

Quick sort

Algoritmo Quick Sort

para cada partición (sin ordenar)
establecer el primer elemento como pivote
storeIndex = pivotIndex+1
for i = pivotIndex+1 to rightmostIndex
if ((a[i] ¡ a[pivot]) o (igual pero 50p/ciento afortunado))
swap(i, índicetienda); ++storeIndex
intercambio (pivote, storeIndex-1)

Quick sort

Costo Computacional

Su tiempo es menor que e de todos los algoritmos de ordenación de complejiad O(n log n)

Pivote	Peor caso	Caso Promedio
primer elemento	$\mathbf{O(n^2)}$	O(n log n)
intermedio de los elementos	$\mathbf{O(n^2)}$	O(n log n)
pseudo-mediana	O(n log n)	O(n log n)

Podemos imaginar un comportamiento parecido al Merge sort

$T(N) \equiv N * log2N$

Insertion sort

Este algoritmo divide la tabla en una parte ordenada y otra no

- la parte ordenada comienza estando formada por un único elemento (el que ocupa la primera posición de la tabla)
- los elementos son insertados uno a uno desde la parte no ordenada a la ordenada
- finalmente la parte ordenada acaba abarcando toda la tabla

Insertion sort

Algoritmo Insertion Sort

marcar el primer elemento como ordenado
para cada elemento sin clasificar X
'extraer' el elemento X
for j = lastSortedIndex hasta 0
if elelemento actual j ¿ X
mover elemento ordenado a la derecha por 1
romper bucle e insertar X aquí

Insertion sort

Costo Computacional

$$C(n-1+1)((n-1)/2) = cn^2/2 - cn/2 \quad (2)$$

Al utilizar una notación grande, podemos descartar $cn/2$
El peor caso se da cuando la tabla se encuentra inicialmente ordenada en orden decreciente

$$T(n) = \theta(n^2) \quad (3)$$

Cuando la tabla esta ordenada su tiempo de ejecución es:

$$T(n) = \theta(n) \quad (4)$$

Radix sort

Es una generalización del método de ordenación por cajas

- Se crea una cola para cada dígito
- Se encola cada elemento en la cola correspondiente a su dígito menos significativo
- Se vuelcan los contenidos de las colas en el array
- Se vuelven a encolar, ahora en base a su segundo dígito menos significativo y así sucesivamente

Radix sort

Algoritmo Radix Sort

crear 10 cubos (colas) para cada dígito (0 a 9)
por cada digito colocado
para cada elemento en la lista
mover el elemento al cubo respectivo
para cada cubo, a partir del dígito más pequeño
mientras el cubo no está vacío
restaurar elemento a la lista

Radix sort

Costo Computacional

Análisis de eficiencia

- El lazo externo se realiza k veces
- El primer lazo se realiza n veces
- Los lazos anidados para volcar las cajas en el array se realizan en el peor caso b+n-1 veces (O(n) cuando b<<n)
- Luego su eficiencia es O(k*n)

Índice

Resultados

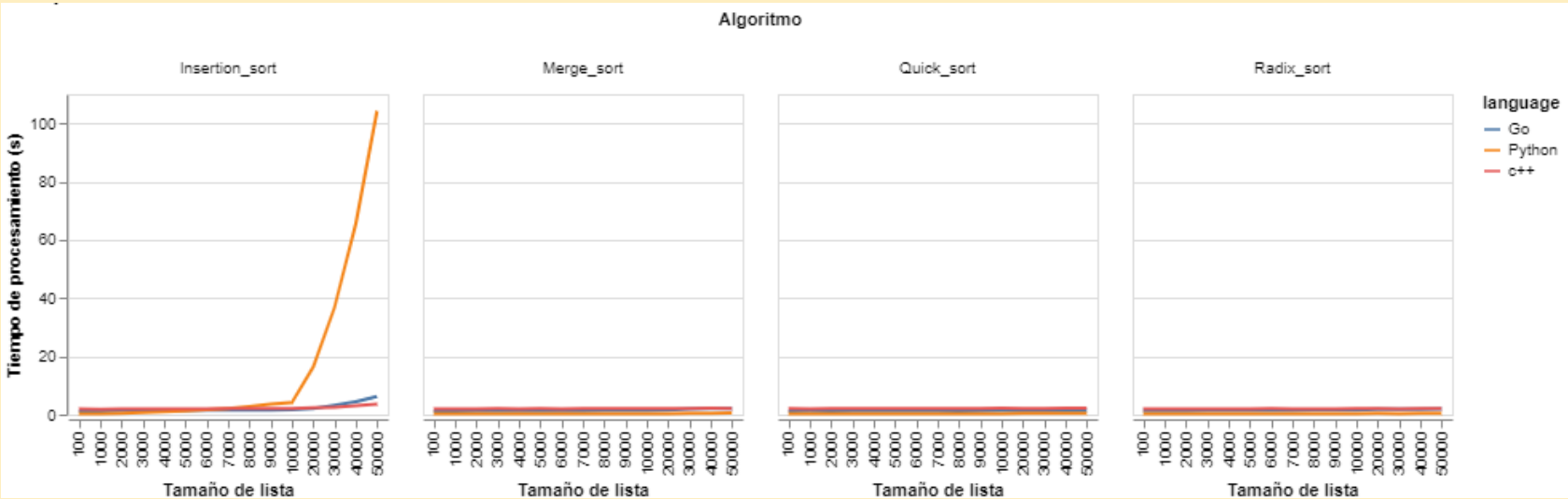

Figura 1 Tiempo de procesamiento vs Tamaño de Lista - Elaboración propia

Resultados

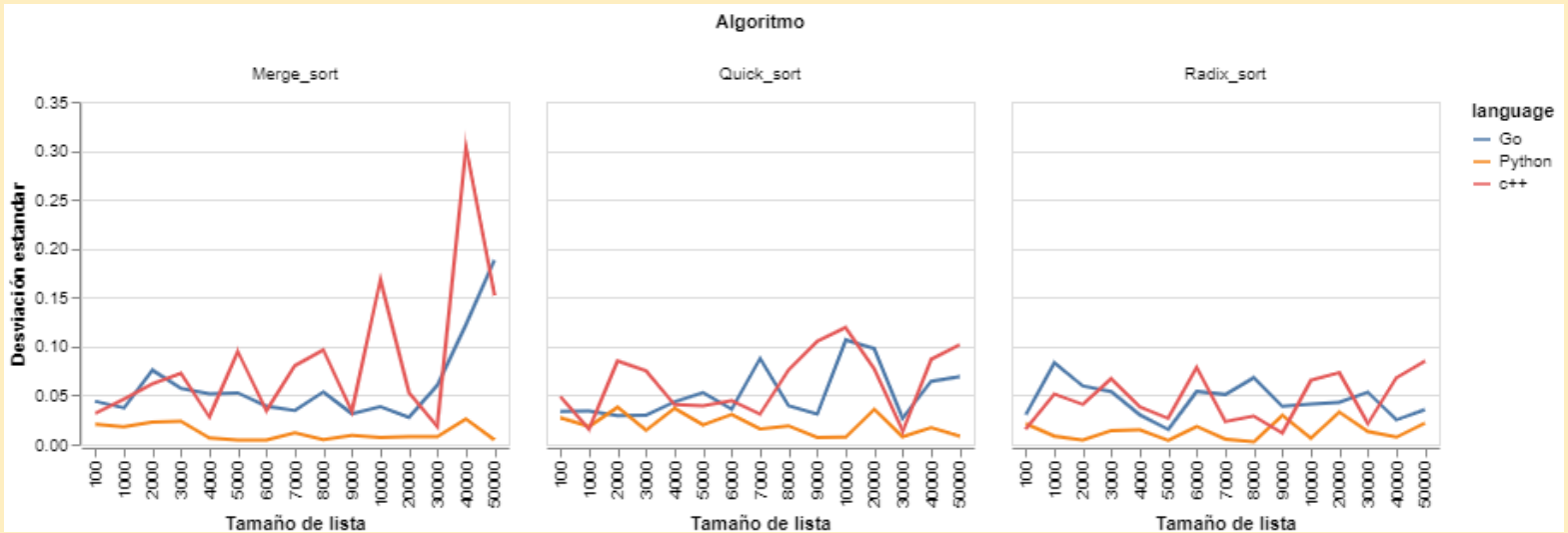

Figura 2 Desviación Standard vs Tamaño de Lista - Elaboración propia

Resultados

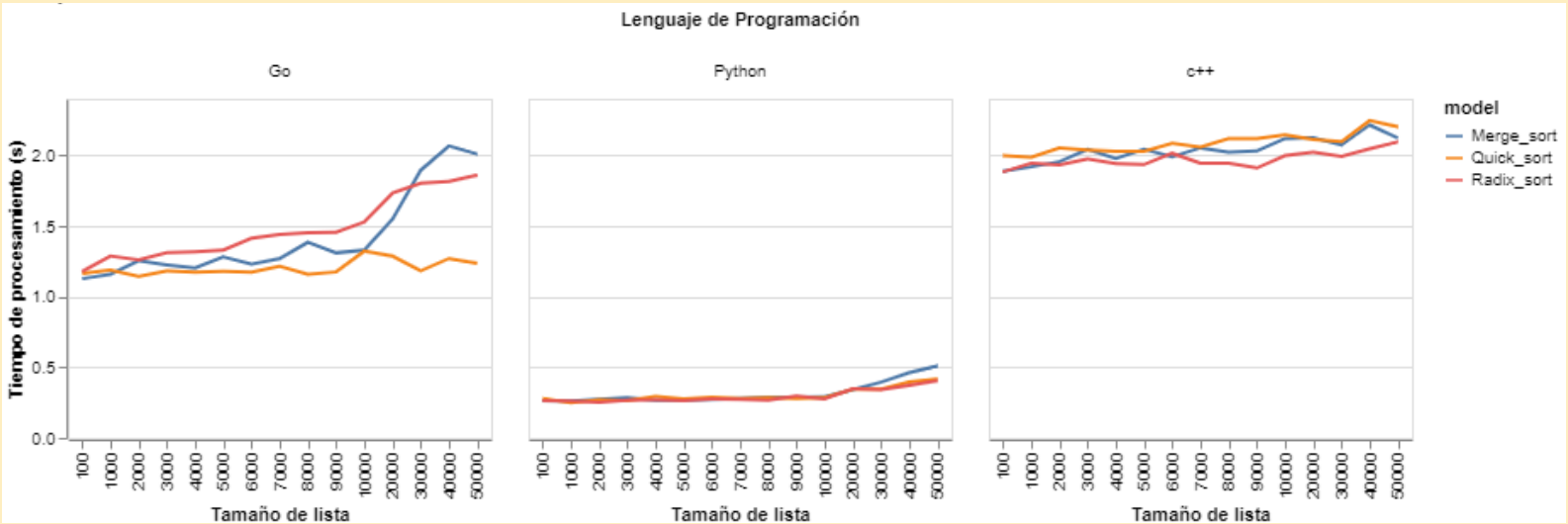

Figura 3 Tiempo de procesamiento vs Tamaño de Lista - Lenguaje de Programación - Elaboración propia

Resultados

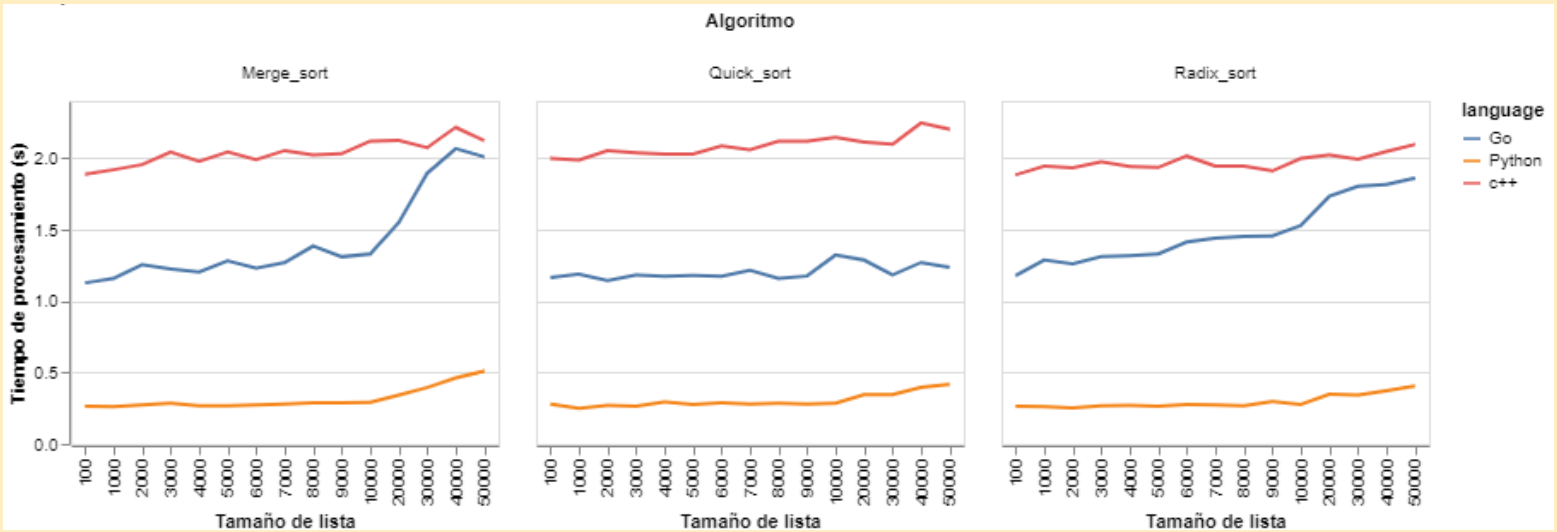

Figura 4 Tiempo de procesamiento vs Tamaño de Lista - Elaboración propia

Índice

Conclusiones

- El algoritmo que tuvo mejor resultado fue el Radix sort y en el lenguaje de ería el python.
- En las pruebas realizadas se puede considerar como mejor algoritmo el radix siendo casi similar a los demas algoritmos a excepción de Insert que tiene un costo de n^2
- El lenguaje a parte de su sencillez para programar tiene los tiempos mas bajos a excepción del modelo Insert, ademas tiene poca variabilidad en sus tiempos.