Práctica 1

Christian Néstor Barriga Marcapura
Weimar Ccapatinta Huamani

Agosto 20, 2022

Contents

1 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 [1].
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.

2 Algoritmos

Se han evaluado los siguientes algoritmos:

2.1 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.

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
```

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 $T(1) = T(0) = b$

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

$$T(N) = T(2^k) = 2 * T(2^k - 1) + a * 2^k \tag{1}$$

$$T(N) = 2 * (2 * T(2^k - 2) + a * 2^k - 1) + a * 2^k \tag{2}$$

$$= 2^k * T(1) + k * a * 2^k \tag{3}$$

$$= b * 2^k + k * a * 2^k \tag{4}$$

$k = log2N$

En consecuencia

$$T(N) \equiv b * N + a * N * Log2N \tag{5}$$

2.2 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

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)

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	$\mathbf{O(}n^2\mathbf{)}$	O(n log n)
intermedio de los elementos	$\mathbf{O(}n^2\mathbf{)}$	O(n log n)
pseudo-mediana	O(n log n)	O(n log n)

Podemos imaginar un comportamiento parecido al Merge sort

$T(N) \equiv N * log2N$

2.3 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

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

Costo Computacional

$$c1 + c2 + c3 + ...c(n-1) = c(1 + 2 + 3 + ... + (n-1)) \tag{6}$$

$$C(n-1+1)((n-1)/2) = cn^2/2 - cn/2 \tag{7}$$

Al utilizar una notación grande, podemos descartar $cn/2$

El peor caso se da cuando la tabla se encuentra inicialmente ordenada en orden decreciente

$$T(n) = \theta(n^2) \tag{8}$$

Cuando la tabla esta ordenada su tiempo de ejecución es:

$$T(n) = \theta(n) \tag{9}$$

2.4 Radix -sort

Es una generalización del método de ordenación por cajas El método se puede aplicar siempre que los valores a ordenar sean secuencias de dígitos (o letras)

- 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

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
```

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)

3 Implementación

La implementación se realizó en se envía el siguiente enlace en Github:

https://github.com/weicap/MCC_practica_1.git

4 Resultados

Luego de realizar la implementación de los algoritmos en diversos tipos de lenguaje, se obtuvieron los siguientes resultados.

4.1 Tabla Comparativa con el promediode tiempo de procesamiento y SD

El cuadro comparativo se muestra a continuación:

Modelo	Muestra	Python		Go		C++	
		Mean	STD	Mean	STD	Mean	STD
Insertion_sort	**100**	0.2582	0.022073	1.1676	0.046554	1.8964	0.039298
	1000	0.2846	0.003362	1.1552	0.019537	1.8376	0.013831
	2000	0.4222	0.02915	1.361	0.039592	1.8756	0.017271
	3000	0.7336	0.064162	1.355	0.022946	1.9192	0.044144
	4000	0.9352	0.024345	1.3458	0.019575	1.8956	0.016288
	5000	1.2228	0.036622	1.4232	0.055468	1.9534	0.051306
	6000	1.6332	0.020055	1.4696	0.044287	1.948	0.019837
	7000	2.1264	0.031548	1.4638	0.050584	2.01	0.056445
	8000	2.6892	0.024056	1.468	0.013285	2.0056	0.044948
	9000	3.5332	0.157986	1.5288	0.02827	2.112	0.109117
	10000	4.1098	0.049605	1.6232	0.088106	2.0164	0.023234
	20000	16.307	1.155462	2.1204	0.040097	2.3698	0.121401
	30000	36.6776	0.822062	3.2256	0.057121	2.5398	0.030947
	40000	65.5718	0.978431	4.4314	0.033239	3.0338	0.033432
	50000	104.2486	1.771072	6.2008	0.084188	3.637	0.055109
Merge_sort	**100**	0.265	0.020724	1.1264	0.044003	1.8858	0.031878
	1000	0.2616	0.018147	1.157	0.037195	1.9184	0.046312
	2000	0.2742	0.022621	1.2536	0.075913	1.9546	0.061906
	3000	0.2854	0.023628	1.2242	0.057072	2.042	0.072612
	4000	0.267	0.006403	1.2038	0.051655	1.9768	0.027968
	5000	0.268	0.004301	1.2808	0.052599	2.0426	0.094841
	6000	0.2726	0.004506	1.2304	0.039068	1.9896	0.034428
	7000	0.2796	0.011567	1.269	0.034706	2.0532	0.080388
	8000	0.2872	0.004868	1.3858	0.053471	2.023	0.09639
	9000	0.2892	0.009039	1.3096	0.031286	2.0296	0.033642
	10000	0.292	0.006819	1.3286	0.038279	2.1186	0.168274
	20000	0.3424	0.008019	1.5504	0.027537	2.1232	0.052813
	30000	0.3942	0.007887	1.8946	0.060331	2.073	0.018466
	40000	0.4612	0.025956	2.0662	0.122383	2.214	0.303309
	50000	0.5106	0.004506	2.0088	0.188159	2.1226	0.152218
Quick_sort	**100**	0.2802	0.027087	1.1646	0.033754	1.998	0.048806
	1000	0.25	0.018262	1.1878	0.034186	1.9858	0.015418
	2000	0.2692	0.037963	1.1432	0.029457	2.052	0.085141
	3000	0.2632	0.01436	1.1816	0.02988	2.0358	0.075145
	4000	0.294	0.036959	1.1726	0.043644	2.028	0.040515
	5000	0.2748	0.019791	1.1784	0.052643	2.0276	0.039475
	6000	0.2876	0.030632	1.1742	0.036024	2.0836	0.044484
	7000	0.2798	0.015975	1.2142	0.08764	2.0584	0.030908
	8000	0.2844	0.018929	1.158	0.039516	2.1194	0.07614
	9000	0.278	0.007176	1.1744	0.031053	2.1172	0.10512
	10000	0.2864	0.007266	1.3216	0.106667	2.1456	0.11927
	20000	0.3438	0.035682	1.2868	0.098047	2.1128	0.077283
	30000	0.3452	0.007791	1.1836	0.026576	2.096	0.013546
	40000	0.3948	0.016991	1.2696	0.064392	2.2456	0.086921
	50000	0.416	0.008396	1.2358	0.069132	2.201	0.101656

Modelo	Muestra	Python		Go		C++	
		Mean	STD	Mean	STD	Mean	STD
Radix_sort	**100**	0.2634	0.020959	1.1764	0.030435	1.8822	0.015418
	1000	0.2618	0.008379	1.2864	0.083602	1.9426	0.051432
	2000	0.2516	0.004278	1.2592	0.059621	1.9308	0.040776
	3000	0.2662	0.014202	1.3096	0.053841	1.9736	0.066943
	4000	0.2692	0.014822	1.3178	0.029987	1.9406	0.038201
	5000	0.263	0.003937	1.329	0.015182	1.9338	0.026584
	6000	0.2754	0.018202	1.4128	0.05396	2.0138	0.078776
	7000	0.272	0.005339	1.4406	0.051013	1.9438	0.02336
	8000	0.2684	0.002793	1.4512	0.068189	1.9428	0.028735
	9000	0.298	0.029589	1.4546	0.038798	1.9104	0.011696
	10000	0.2768	0.006058	1.5274	0.04104	1.997	0.065639
	20000	0.3482	0.032935	1.7324	0.042864	2.0226	0.073231
	30000	0.3434	0.012954	1.802	0.053315	1.9934	0.021161
	40000	0.372	0.007314	1.8158	0.025173	2.045	0.067915
	50000	0.4054	0.021732	1.861	0.035377	2.0952	0.085125

4.2 Gráficos

Con la tabla anterior se realizaron tres tipos de analisis

- Tiempo de Procesamiento por algoritmo.
- Tiempo de procesamiento por lenguaje de programación.
- Desviación Estándar.

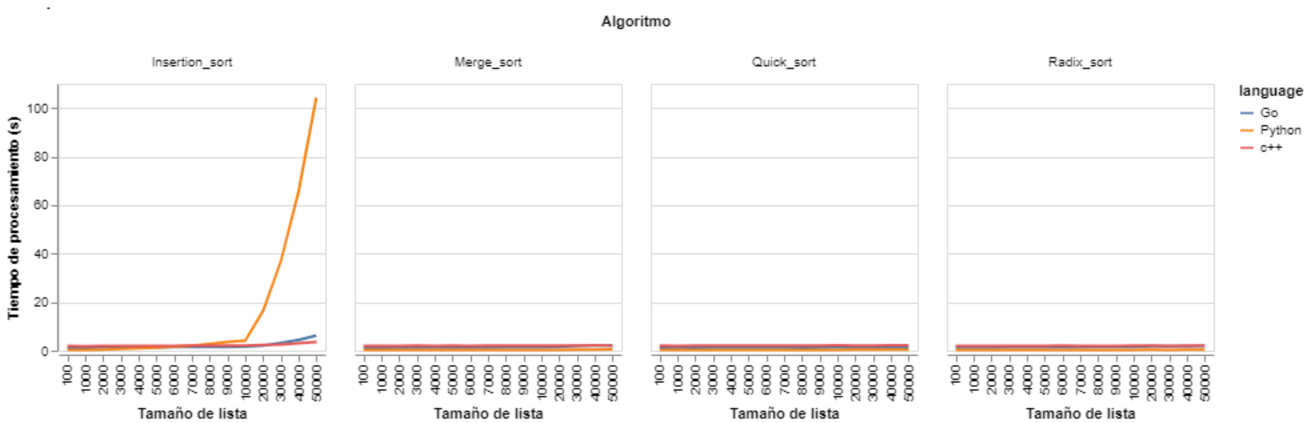

Figura 1 Tiempo de procesamiento vs Tamaño de Lista - Elaboración propia

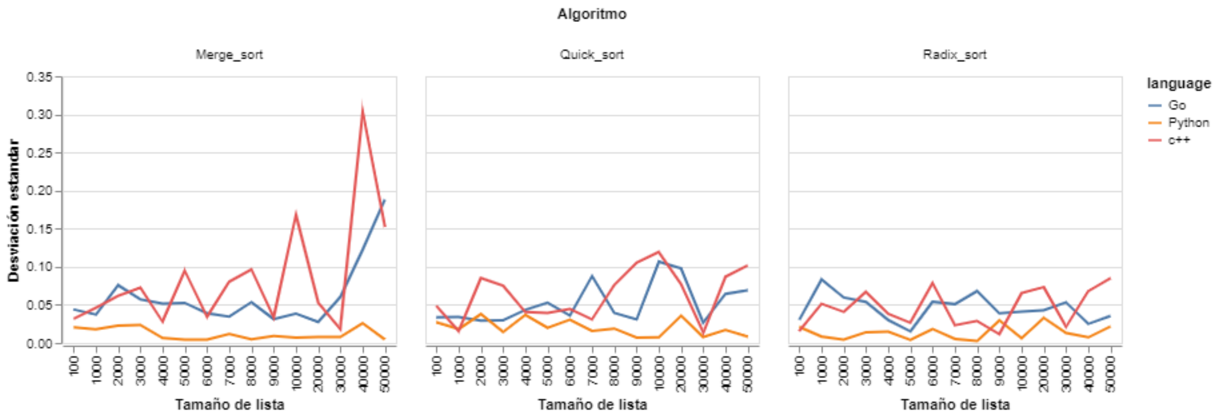

Figura 2 Desviación Standard vs Tamaño de Lista - Elaboración propia

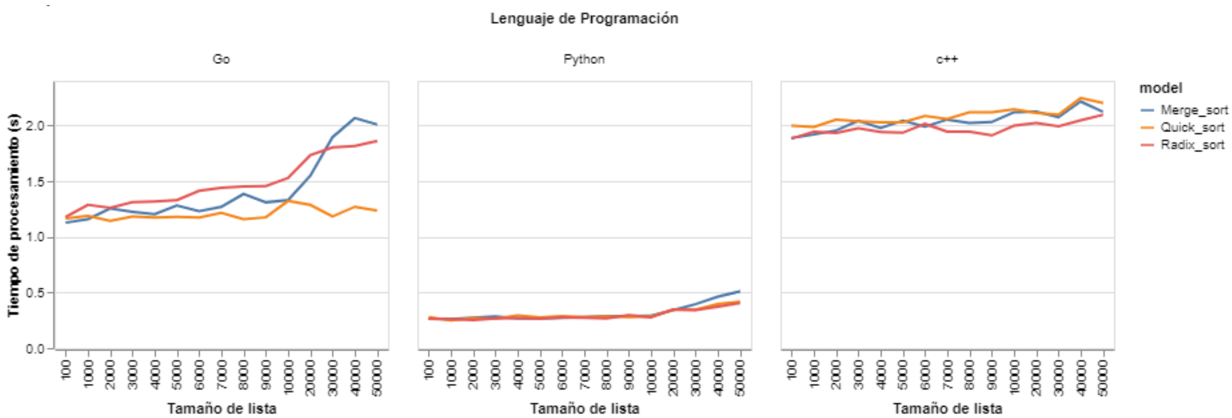

Figura 3 Tiempo de procesamiento vs Tamaño de Lista - Lenguaje de Programación - Elaboración propia

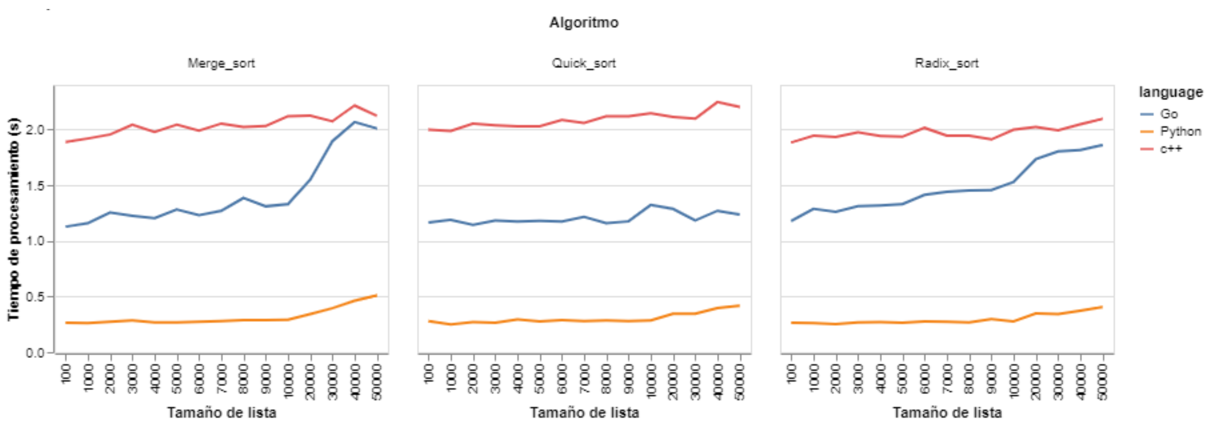

Figura 4 Tiempo de procesamiento vs Tamaño de Lista - Elaboración propia

5 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.

References

[1] Villegas Jaramillo Eduardo et all, *Análisis y diseño de algoritmos - Un enfoque práctico.* Editorial Universidad Nacional de Colombia, Manizales 2016.