Práctica 1

Algoritmos de Ordenamiento

J. P. Abarca[1] C. T. Apari[1] C. A. Suca[1] A. Vargas[1]

[1]Universidad Nacionan del San Agustin. Facultad de Producción y Servicios.
Escuela Profesional de Ciencias de la Computación
Maestría en Ciencias de la Computación
Docente: Mg. Vicente Machaca

26 de Junio del 2021

Contenido

Contenido

Resumen

- Se realizó la implementación de algoritmos de ordenamiento y la medición de su respectivo tiempo de ejecución graficados.
- La implementación de los algoritmos se realizó en dos lenguajes de programación: C++ y Python, para realizar la medición se crearon archivos con elementos desordenados, cada algoritmo genera un archivo *.txt resultante con la cantidad de datos y el tiempo de ejecución, finalmente estos resultados se graficaron para mostrar las diferencias entre lenguaje de programación y algoritmo.

Contenido

Preparación de Datos

- Se generaron 21 archivos con números generados aleatoriamente.
- Cada archivo contiene 100, 500, 1000, 2000, 3000, ..., 10000, 20000, 30000, ..., 1000000 números con la denominación ejemplo_<número>

```
100%|██████████| 100/100 [00:00<00:00, 25765.12it/s]
100%|██████████| 500/500 [00:00<00:00, 49433.15it/s]
100%|██████████| 1000/1000 [00:00<00:00, 63405.96it/s]
100%|██████████| 2000/2000 [00:00<00:00, 70244.58it/s]
100%|██████████| 3000/3000 [00:00<00:00, 73384.73it/s]
100%|██████████| 4000/4000 [00:00<00:00, 48259.20it/s]
100%|██████████| 5000/5000 [00:00<00:00, 72717.79it/s]
100%|██████████| 6000/6000 [00:00<00:00, 62599.84it/s]
100%|██████████| 7000/7000 [00:00<00:00, 72032.25it/s]
100%|██████████| 8000/8000 [00:00<00:00, 46595.94it/s]
100%|██████████| 9000/9000 [00:00<00:00, 53528.21it/s]
100%|██████████| 10000/10000 [00:00<00:00, 67285.64it/s]
100%|██████████| 20000/20000 [00:00<00:00, 49611.78it/s]
100%|██████████| 30000/30000 [00:00<00:00, 61301.00it/s]
100%|██████████| 40000/40000 [00:00<00:00, 45272.71it/s]
100%|██████████| 50000/50000 [00:01<00:00, 47290.92it/s]
100%|██████████| 60000/60000 [00:01<00:00, 54710.12it/s]
100%|██████████| 70000/70000 [00:01<00:00, 46331.58it/s]
100%|██████████| 80000/80000 [00:01<00:00, 69166.27it/s]
100%|██████████| 90000/90000 [00:01<00:00, 56055.46it/s]
100%|██████████| 100000/100000 [00:01<00:00, 55143.17it/s]
```

Figura: Archivos generados

Preparación de Datos - Codigo (parte 1)

```
1 import numpy as np
2 import os
3 from tqdm import tqdm
4 save_path_generatedFiles = "generatedTestData"
5 if (os.path.isfile(save_path_generatedFiles)==False):
6   os.mkdir(save_path_generatedFiles)
7
8 nameFiles_idx = np.concatenate((np.arange(100,501,400),np.
    arange(1000,10000,1000),np.arange(10000,100001,10000)),
    axis=None)
```

Listing 1: Generación de Números(parte 1)

Preparación de Datos - Codigo (parte 2)

```
1
2  dicPaths = {}
3  for namefile in nameFiles_idx:
4    dicPaths.update({namefile:'generatedTestData/example_'+str
       (namefile)+".txt"})
5    #create specific file
6
7    f = open(dicPaths[namefile], "w")
8      #generar un nro aleatorio de 1 y 100'000
9    for i in tqdm(range(1,namefile+1)):
10     f.write(str(np.random.randint(1,100000,1)[0])+"\n")
11   f.close()
```

Listing 2: Generación de Números(parte 2)

Contenido

Desarrollo

- Para la implementación de cada algoritmo se tomaron los archivos generados en el primer punto.
- Cada algoritmo implementado ordenara los números generados por cada archivo.
- Cada vez que calcule a cada archivo este generará un nuevo archivo *.txt mostrando la cantidad de números revisado con el tiempo de ejecución.

BubbleSort - Definición

- El procedimiento de ejecución comienza con un arreglo de números enteros distribuidos de forma aleatoria, los cuales deben ser ordenados con una búsqueda secuencial sobre los elementos ya procesados, esto consiste en recorrer una y otra vez los elementos del arreglo que ya fueron ordenados, encontrar una posicion especifica e intercambiar con el elemento seleccinoado.

Algorithm 1 BUBBLE-SORT(A)

```
1: for i ← 1 to N do
2:     for j ← i + 1 to N do
3:         if A[j] < A[i] then
4:             temp ← A[i]
5:             A[i] ← A[j];
6:             A[j] ← temp;
7:         end if
8:     end for
9: end for
```

BubbleSort - Análisis

- La complejidad computacional del algoritmo es $O(n^2)$, en el peor y mejor caso. Siendo n la cantidad de elementos a ordenar.
- La comparacion del tiempo de ejecucion del algoritmo Bubble Sort en los lenguajes C++ y Python, teniendo como mas optimo al C++.

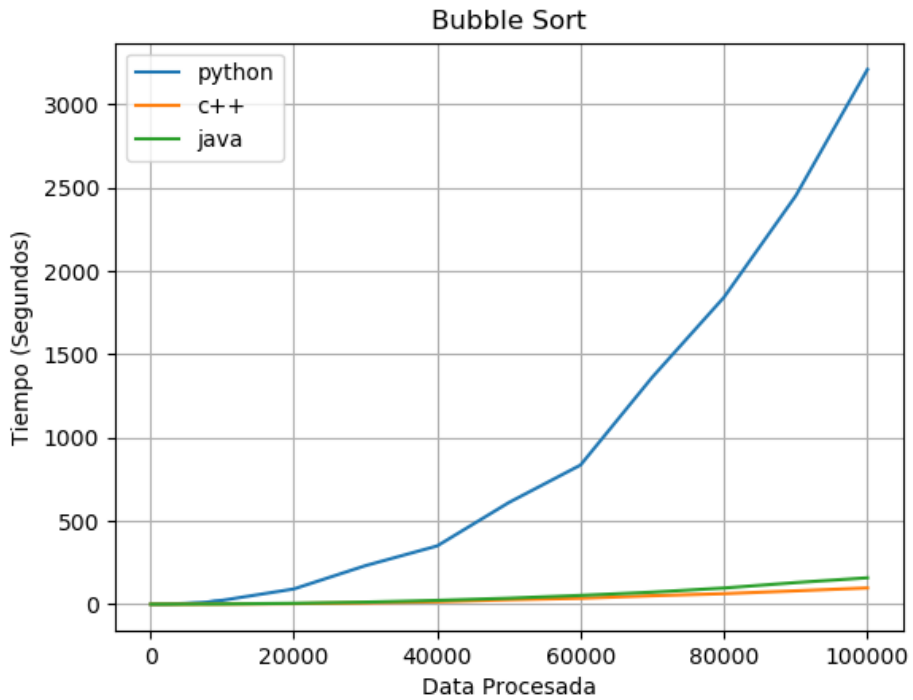


Figura: Bubble Sort - Tiempo de Ejecución

CountingSort - Definición

- Permite ordenar números sin necesidad de realizar comparaciones, usando un contador para almacenar los valores repetidos del arreglo.

Algorithm 2 COUNTING-SORT(A,B,k)

```
1:  for i ← 0 to K do
2:      C[i] ← 0
3:  end for
4:  for j ← 1 to length[A] do
5:      C[A[j]] ← C[A[j]] + 1
6:  end for
7:  for i ← 1 to k do
8:      C[i] ← C[i] + C[i − 1]
9:  end for
10: for j ← length[A] to 1 do
11:     B[C[A[j]]] ← A[j]
12:     C[A[j]] ← C[A[j]] − 1
13: end for
```

CountingSort - Funcionamiento

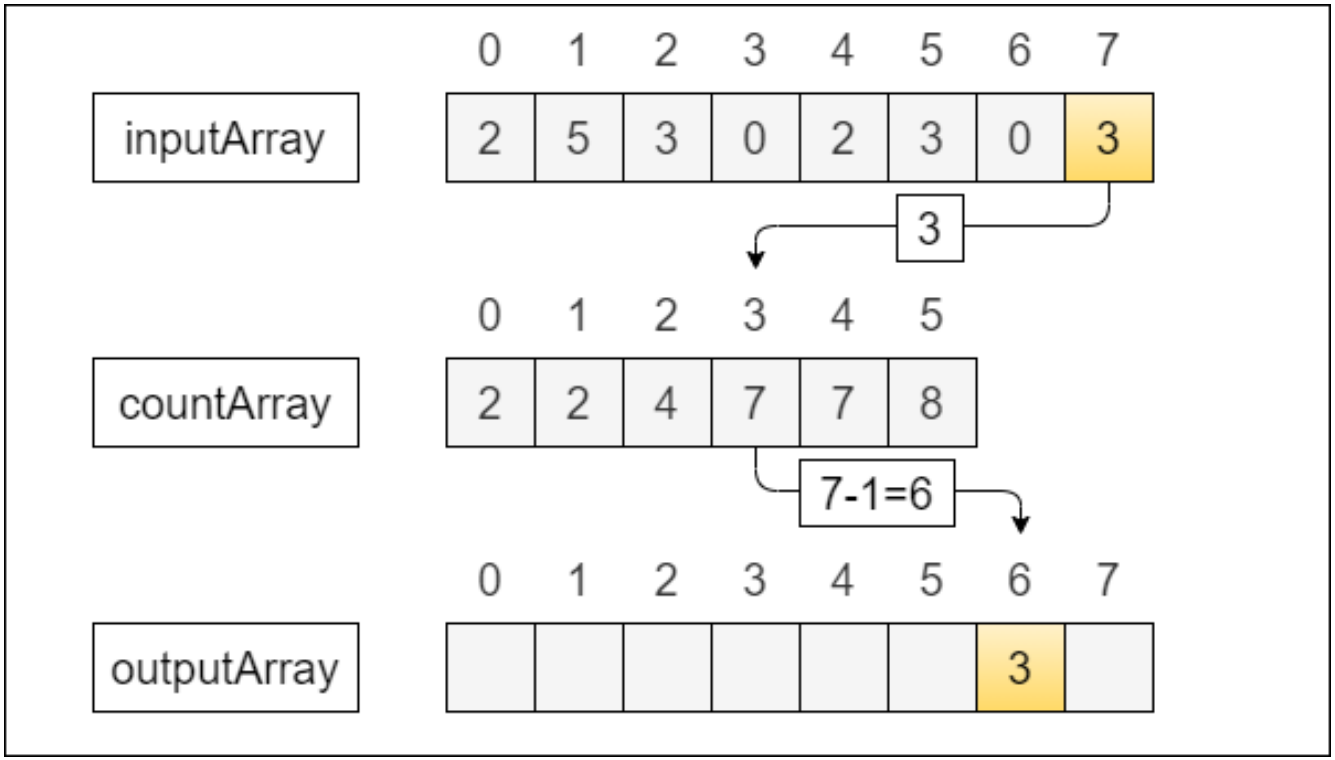


Figura: Counting Sort

CountingSort - Análisis

- La complejidad computacional del algoritmo es $O(n+k)$, en el peor y mejor caso.

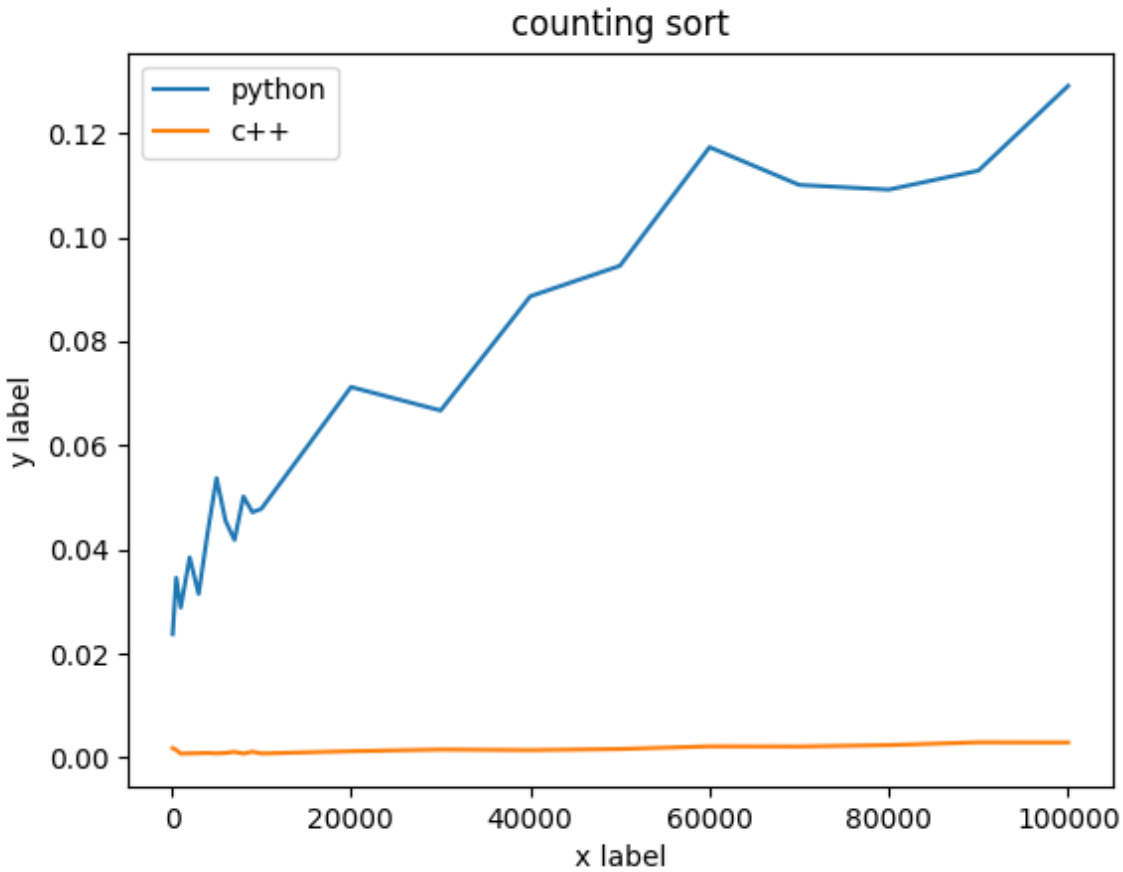


Figura: CountingSort - Tiempo de Ejecución

HeapSort - Definición

- Esta estructura basada en un árbol binario mantiene una propiedad o invariante de que en cada nivel cualquier nodo sea siempre mayor a sus hijos $A[Parent(i)] \geq A[i]$, como se observa en la figura 5.

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X

HeapSort - Definición

- El primer proceso es Build-Max-Heap, el principal objetivo es hacer cumplir la invariante, el cual es que todo nodo padre sea mayor a sus nodos hijo, en caso de no serlo se producirá el intercambio y recursivamente se ayudará de la función Max-Heapify haciendo la propagación hacia abajo.

Algorithm 3 Build-Max-Heap(A)

1: $heap - size[A] \leftarrow length[A]$
2: **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ to 1 **do**
3: $\quad Max - Heapify(A, i)$
4: **end for**

HeapSort - Definición

- El segundo proceso, es el SortDown aplicando Max-Heapify quien se encarga de hacer cumplir la invariante.

Algorithm 4 Max-Heapify(A,i)

1: $l \leftarrow LEFT(i)$
2: $r \leftarrow RIGHT(i)$
3: **if** $l \leq heap - size[A]$**and**$A[l] > A[i]$ **then**
4: $largest \leftarrow l$
5: **else**
6: $largest \leftarrow i$
7: **end if**
8: **if** $r \leq heap - size[A] and A[r] > A[largest]$ **then**
9: $largest \leftarrow r$
10: **end if**
11: **if** $largest \neq i$ **then**
12: $exchange A[i] \leftrightarrow A[largest]$
13: $Max - Heapify(A, largest)$

HeapSort - Definición

- El completo algoritmo de heapsort es dado en el seudocódigo 5

Algorithm 5 Heapsort(A)

1: $BUILD - MAX - HEAP(A)$
2: **for** $i \leftarrow length[A]$ downto 2 **do**
3: exchange $A[1] \leftrightarrow A[i]$
4: $heap - size[A] \leftarrow heapsize[A] - 1$
5: $MAX - HEAPIFY(A, 1)$
6: **end for**

HeapSort - Análisis

- El procedimiento de Heapsort toma un tiempo de O(nlgn).
- El Build-Max-Heap toma un tiempo lineal O(n), la altura de los montículos, multiplicado por el número de nodos dados en un nivel particular, esto sumado por cada nivel de la construcción en el árbol, da un orden lineal O(n).
- En cuanto al proceso de SortDown se realizar n-1 llamadas de Max-Heapify interno el cual por si solo tiene un coste de O(lgn), en este caso Max-Heapify toma un coste de acuerdo a la altura del montículo. Es así que el coste total de el proceso de SortDown es O(nlgn) y es el coste del proceso de Heapsort.

HeapSort - Análisis

- En la siguiente gráfica 6 se muestra la comparación de los tiempo tomados para el heapsort en 3 lenguajes Python, Java, y C++, donde se observa el crecimiento casi lineal del tiempo tomado por Python conforme la carga de datos crece.

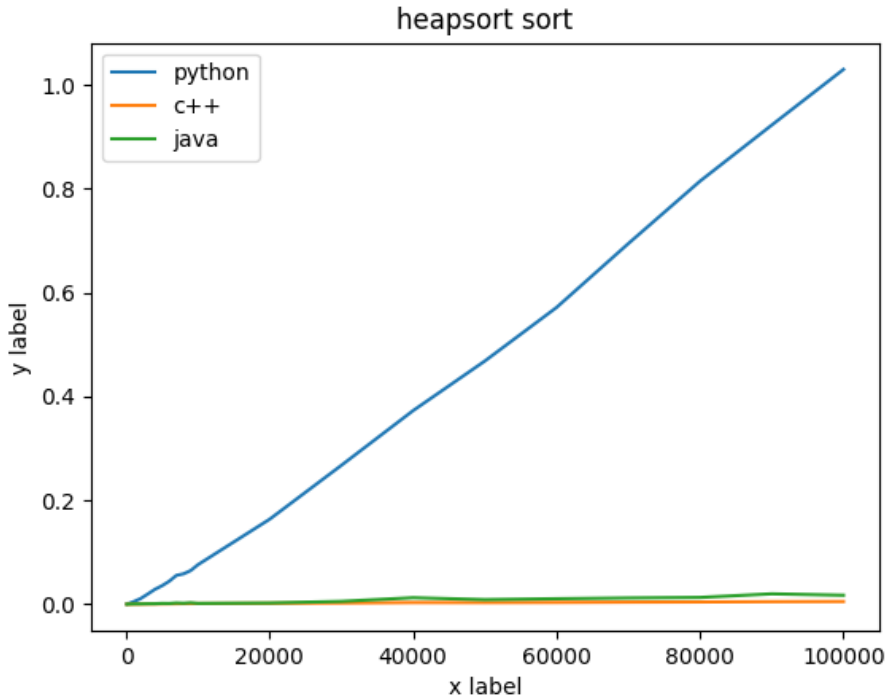


Figura: Heapsort - Tiempo de Ejecución

Insertion Sort - Definición

Algorithm 6 INSERTION-SORT(A)

```
1: for i ← 1 to N do
2:     key ← A[i]
3:     j ← i − 1
4:     while j >= 0 and A[j] > key do
5:         A[j + 1] ← A[j]
6:         j ← j − 1
7:     end while
8:     A[j + 1] ← key
9: end for
```

Insertion Sort - Análisis

- Similar a la manera de ordenar una baraja de un juego de naipes en las manos. El array es virtualmente dividido en una parte ordenada y otra desordenada, los valores de la parte desordenada es elegida y ubicada en la posición correcta en la parte ordenada.
- El algoritmo tiene un costo de $O(n^2)$ [2]

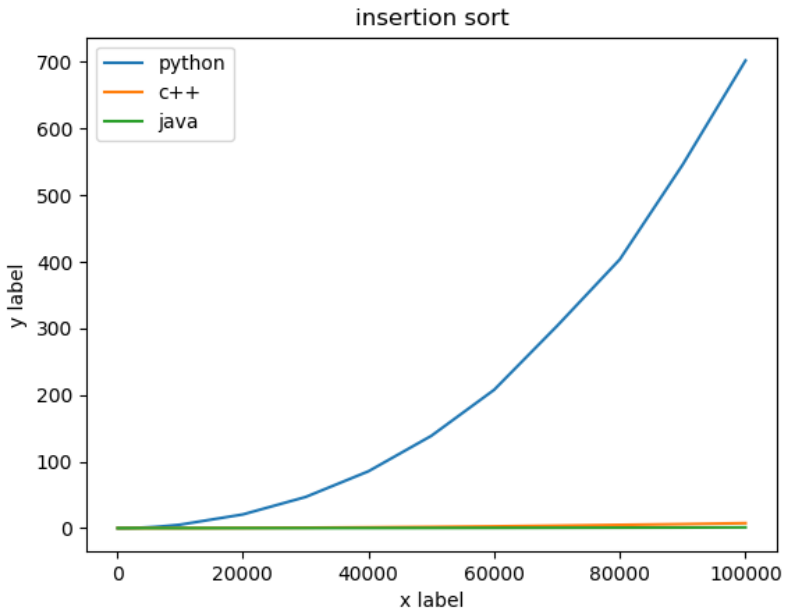


Figura: Tiempo de Ejecución Insertion Sort

MergeSort - Definición

- Se basa en la técnica divide y vencerás.
- Dividir el arreglo no ordenado en dos subarreglos(la mitad del arreglo principal), mezcla las sublistas hasta completar el arreglo principal.

Algorithm 7 MERGE-SORT(A,p,r)

```
1: if r > p then
2:     m ← p + (r − p)/2
3:     CALL mergeSort(A,p,m)
4:     CALL mergeSort(A,m+1,r)
5:     CALL merge(A,p,m,r)
6: end if
```

MergeSort - Definición

Algorithm 8 MERGE(A,p,q,r)

1: $n1 \leftarrow q - p + 1$
2: $n2 \leftarrow r - q$
3: create arrays L[1..n1+1] and R[1..n2+1]
4: **for** $i \leftarrow 1$ to $n1$ **do**
5: $\quad L[i] \leftarrow A[p + i - 1]$
6: **end for**
7: **for** $j \leftarrow 1$ to $n2$ **do**
8: $\quad R[j] \leftarrow A[q + j]$
9: **end for**
10: $i \leftarrow 1$
11: $j \leftarrow 1$
12: $k \leftarrow l$
13: **while** $i < n1$ and $j < n2$ **do**
14: $\quad$ **if** $L[i] <= R[j]$ **then**
15: $\quad\quad A[k] \leftarrow L[i]$

MergeSort - Funcionamiento

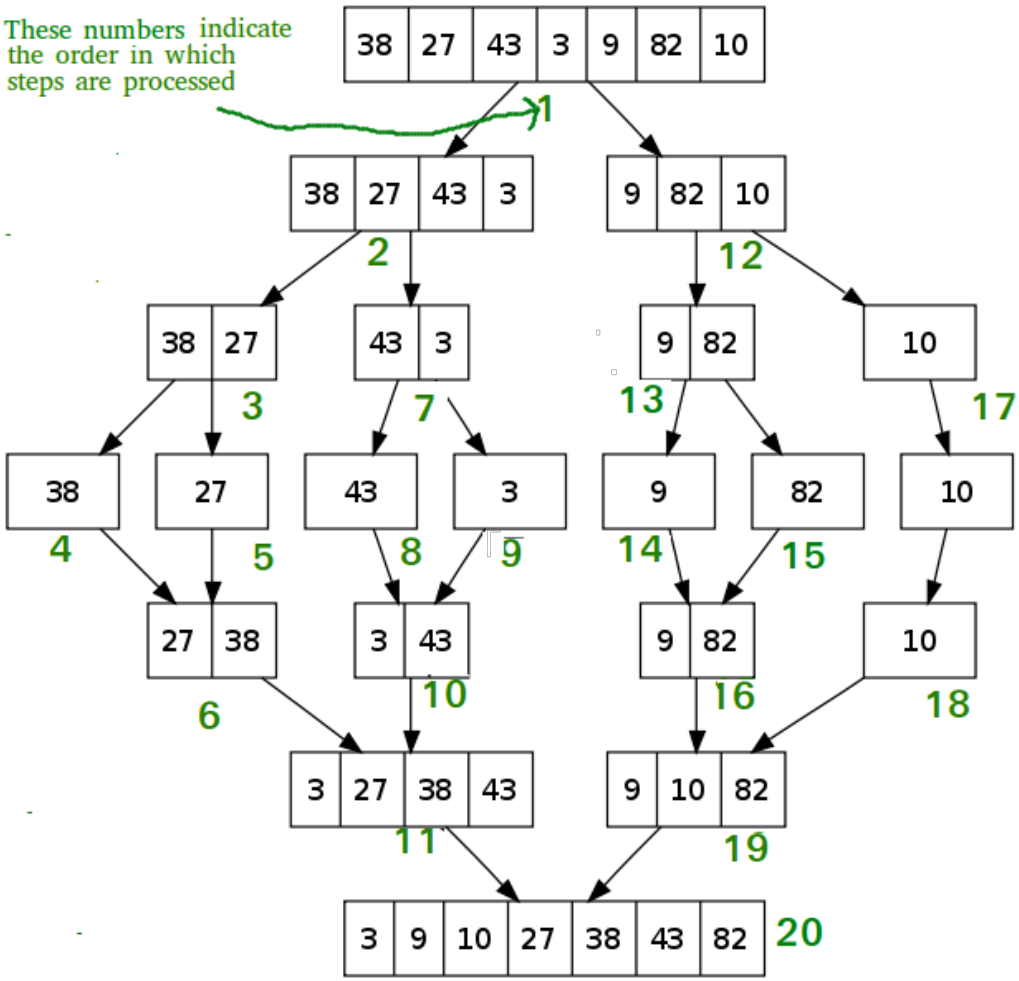
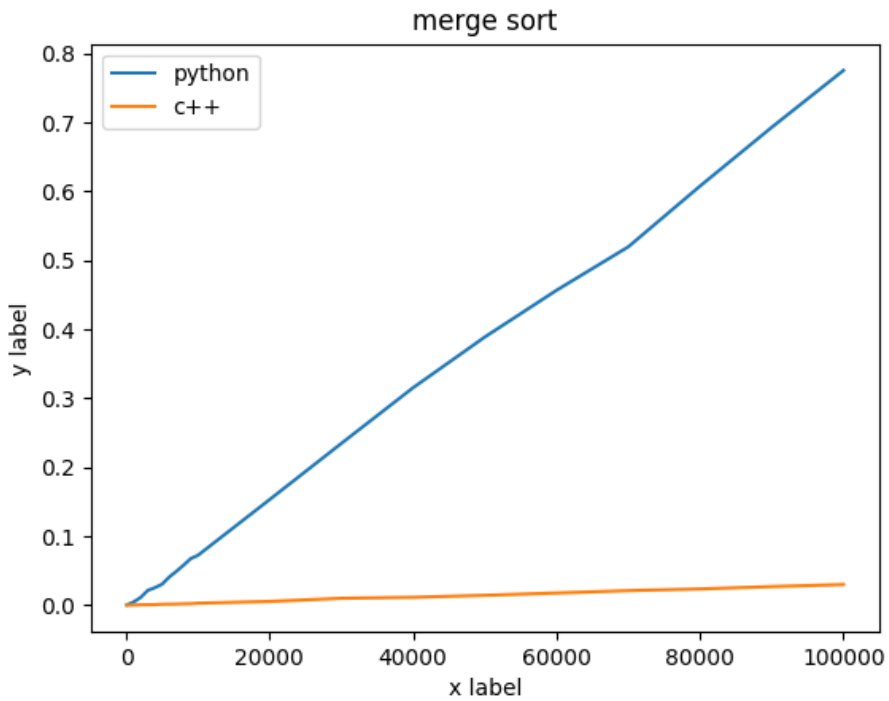


Figura: Merge Sort

MergeSort - Análisis

- La complejidad del algoritmo es $O(nlogn)$.
- No trabaja in situ, es decir no cambia directamente el valor en el arreglo.
- Para mezclar dos subarreglos, deben estar ordenados.
- Requiere un espacio auxiliar $O(n)$, en comparación con Heapsort que requiere un espacio de $O(1)$



QuickSort Definición

- El algoritmo de Quicksort fue inventado por el autor C.A.R. Hoare en 1960, este algoritmo es bastante eficiente y es garantizado el ordenamiento, en el peor caso con un orden de $O(n^2)$ y en el caso promedio $O(n \log(n))$, el ordenamiento es en lugar, solo requiere una pequeña pila adicional.
  En el siguiente seudocódigo 9 se da la implementación del algoritmo quicksort.

Algorithm 9 QUICKSORT(A,p,r)

1: **if** $p < r$ **then**
2: $q \leftarrow Partition(A, p, r)$
3: $QUICKSORT(A, p, q-1)$
4: $QUICKSORT(A, q+1, r)$
5: **end if**

QuickSort Definición

- El factor mas importante en el algoritmo de Quicksort es el proceso de partición , dado por el siguiente seudocódigo 10

Algorithm 10 PARTITION(A,p,r)

```
1:  x ← A[r]
2:  i ← p − 1
3:  for j ← p to r − 1 do
4:      if A[j] ≤ x then
5:          i ← i + 1
6:          A[i] ↔ A[j]
7:      end if
8:  end for
9:  exchange A[i + 1] ↔ A[r]
10: return i+1
```

QuickSort Definición

- El proceso de partición implica tomar el último elemento del array como pivote $x = A[r]$ el cual será el elemento que divida y genere los dos subarrays, en la inicialización de los iteradores i y j se generan 4 regiones, como se observa en la figura 10

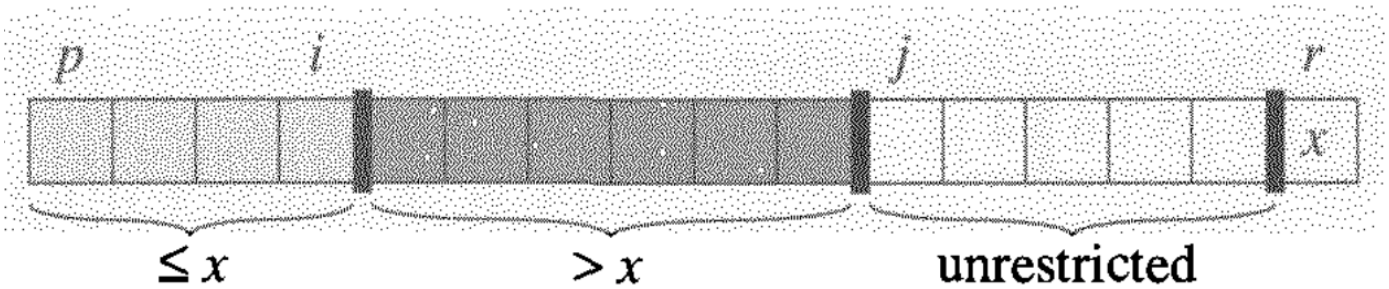


Figura: Regiones en el Partition

Quickort Análisis

- **Pero Caso** Resolviendo al recurrencia el tiempo tomado para el peor caso se produce un tiempo de $T(n) = O(n^2)$, es así que es muy similar al caso del insertion sort, y esto ocurre cuando los elmentos de ingreso ya están completamente ordenados, en este caso insertion sort gana con un orden de $O(n)$.
- **Mejor Caso** Si se diera el caso de que los subarrays sean de tamaño no más de $n/2$, la recurrencia para el tiempo de ejecución es: $T(n) = 2T(n/2) + \Theta(n)$ esta recurrencia tiene la solución de $T(n) = O(nlgn)$.
- **Caso Promedio** Tendrá un coste total de $\Theta(nlg(n))$. siempre y cuando exista almenos una división y sea constante durante todos los niveles del árbol.

Quickort Análisis

- Finalmente se muestra la figura 11 con la comparación en 3 lenguajes Python, Java, y C++ en los tiempos de ejecución para una carga de datos des 100 hasta 100'000 datos a ordenar, para el caso de Python el crecimiento es exponencial a pesar de trabajar con algoritmos de ordenamiento bastante buenos en coste asintótico.

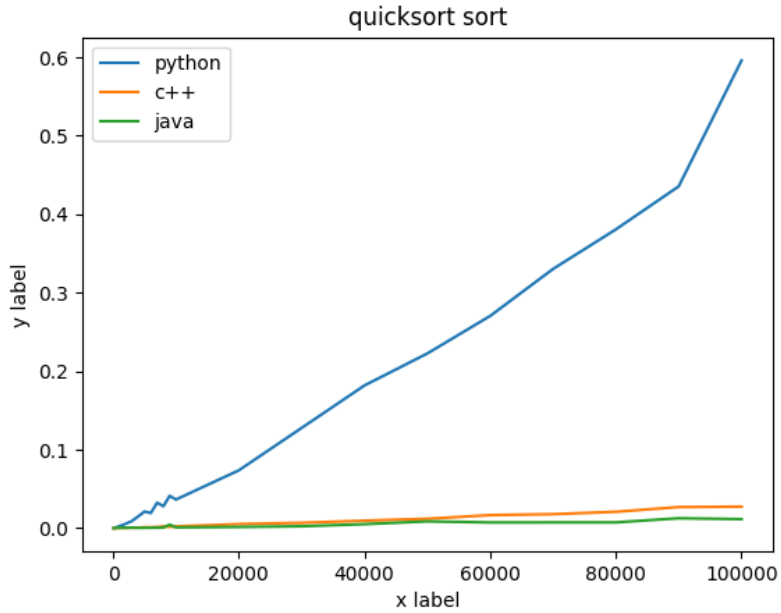


Figura: QuickSort - Tiempo de ejecución

Selection Sort - Definición

Algorithm 11 SELECTION-SORT(A)

```
1: for i ← 0 to N − 1 do
2:     minIdx ← i
3:     for j ← i + 1 to N do
4:         if A[j] < A[minIdx] then
5:             minIdx ← j
6:         end if
7:     end for
8:     A[j + 1] ← key
9: end for
```

Selection Sort - Análisis

- Ordena un Array por encontrar repetidamente el mínimo elemento, en orden ascendente, desde la parte desordenada y poniéndolo al inicio. El algoritmo mantiene un array ya ordenado y otro desordenado.
- El algoritmo tiene un costo de $O(n^2)$ [3]

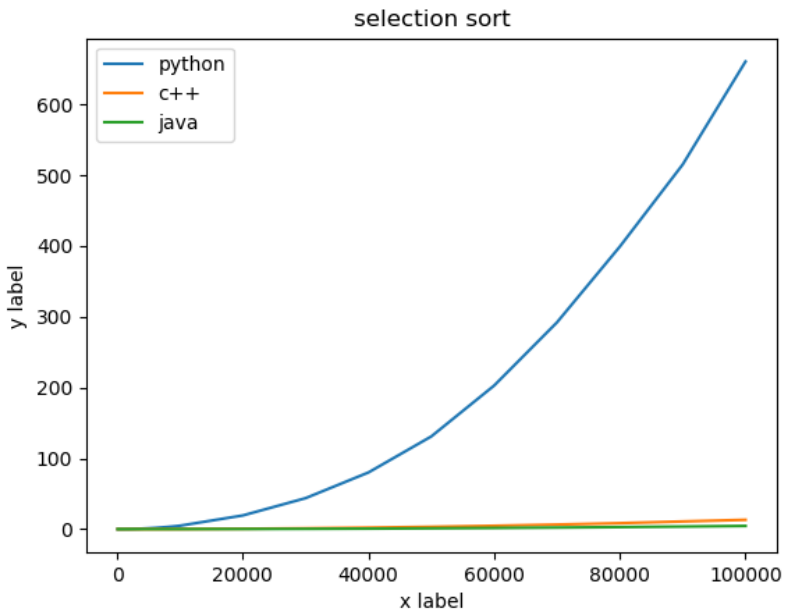


Figura: Tiempo de Ejecución Selection Sort

Contenido

Resultado en C++

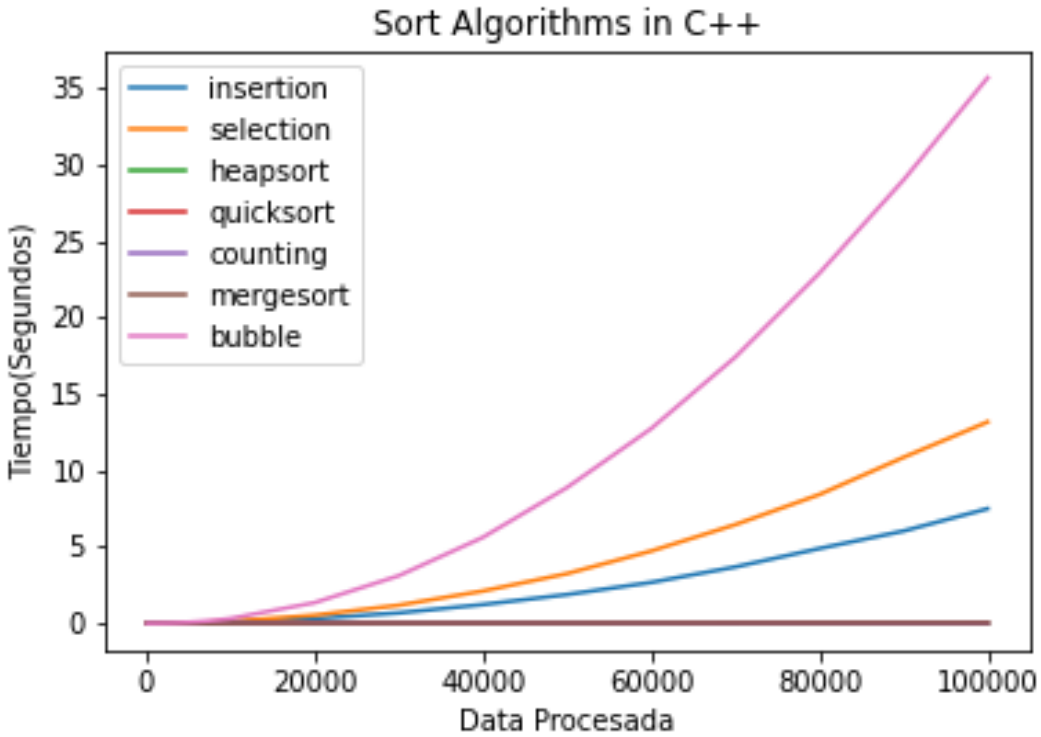


Figura: Algoritmos de Ordenamiento en C++

Resultado en Python

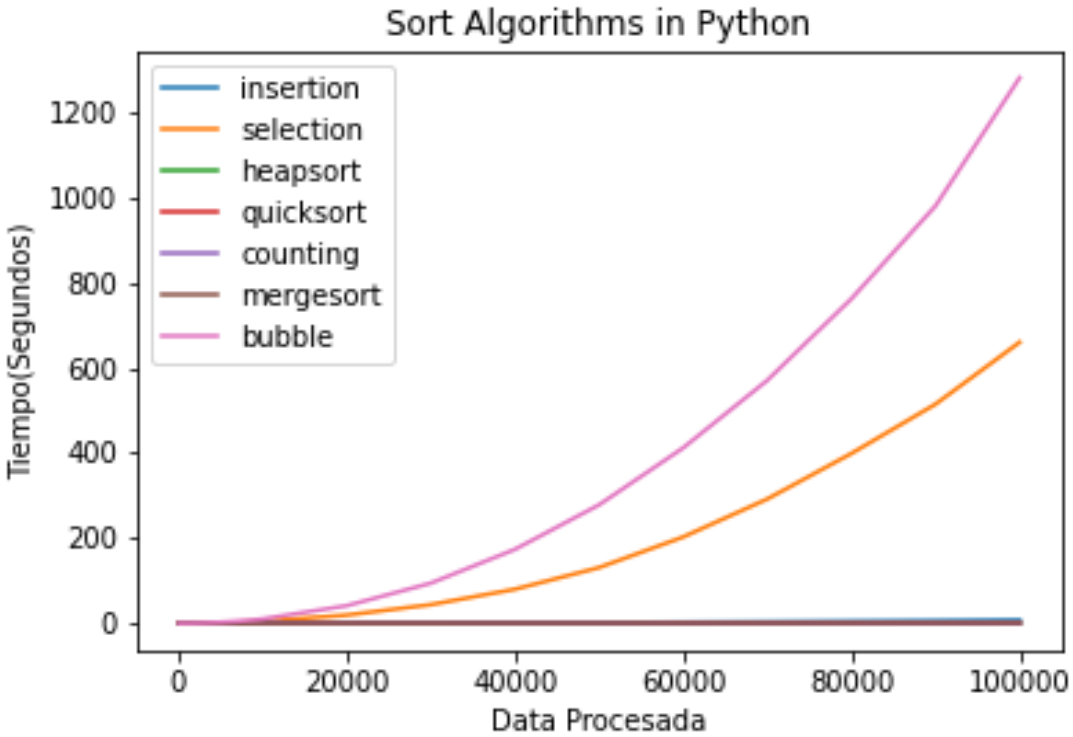


Figura: Algoritmos de Ordenamiento en Python

GRACIAS

Repositorio: `https://github.com/jabarcamu/EDA_Practica1`

Contenido

References I

[1] K. W. Robert Sedgewick, *Algorithms: 4th Edition*.
Addison-Wesley Professional, 2011.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*.
The MIT Press, 3rd ed., 2009.

[3] G. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell*.
O'Reilly Media, Inc., 2008.