Práctica 01

DOCENTE	CARRERA	CURSO
Vicente Machaca Arceda	Maestría en Ciencia de la Computación	Estructura de Datos y Algoritmos

PRÁCTICA	TEMA	DURACIÓN
01	Algoritmos de Ordenamiento	3 horas

1. Datos de los estudiantes

- Grupo: 9
- Integrantes:
  - Abarca Murillo, Jhonatan Piero
  - Apari Pinto, Christian Timoteo
  - Suca Velando, Christian Anthony
  - Vargas Zuni, Arturo
- Repositorio: `https://github.com/jabarcamu/EDA_Practica1`

2. Ejercicios

2.1. Preparación de Datos

La generación de datos se realizó de la siguiente manera:

- Por medio las librerías de generación de números aleatorios y ademas de la libreria de escritura para archivos .txt de inicio con la generación de los archivos
- Cada Archivo tiene el nombre de la cantidad de números generados aleatoriamente.
- Cada archivo contiene 100, 500, 1000, 2000, 3000, ..., 10000, 20000, 30000, ..., 1000000.
- El resultado final se utilizará para ordenar a cada uno de los números generados en cada archivo creado, estos archivos estarán en la carpeta del repositorio donde se alojarán y serán utilizados para la segunda parte de la práctica.

Los archivos generados se guardarán con la denominación ejemplo_<número_iteraciones> en la misma ruta del archivo donde se ejecuta el codigo en la carpeta “generatedTestData”. El resultado se muestra en la figura 1.

```
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 1: Archivos generados

Para la creacion de los archivos se requeria iterar sobre un mismo archivo el cual se realiza mediante el uso de la libreria numpy tomando los rangos pedidos en la práctica. Sobre la lista generada se declara la ruta de generacion del archivo resultante y sobre la misma iterar la lista de numeros generados anteriormente.
Ya estando dentro de cada iteracion mediante el uso de random de la libreria numpy se realiza la generacion de numeros segun sea el turno de iteracion (desde 100 hasta 100000), tal cual como se puede observa en el código mostrado.

```python
import numpy as np
import os
from tqdm import tqdm
save_path_generatedFiles = "generatedTestData"
if (os.path.isfile(save_path_generatedFiles)==False):
  os.mkdir(save_path_generatedFiles)

nameFiles_idx = np.concatenate((np.arange(100,501,400),np.arange(1000,10000,1000),np.
    arange(10000,100001,10000)),axis=None)

dicPaths = {}
for namefile in nameFiles_idx:
  dicPaths.update({namefile:'generatedTestData/example_'+str(namefile)+".txt"})
  #create specific file

  f = open(dicPaths[namefile], "w")
    #generar un nro aleatorio de 1 y 100'000
  for i in tqdm(range(1,namefile+1)):
    f.write(str(np.random.randint(1,100000,1)[0])+"\n")
  f.close()
```

Listing 1: generatedFiles.py

2.2. Implementación de algoritmos

La implementación de todos los algoritmos que presentaremos a continuación estan implementados en los lenguajes C++ y Python.

2.2.1. Bubble sort

Según la definición del libro [1] el algoritmo Bubble Sort tiene la siguiente estructura, concepto y pseudocodigo:

Input: Un arreglo de n números enteros positivos $< a_1, a_2, ..., a_n >$, un arreglo de n números enteros positivos inicializados en 0 para almacenar el resultado $< 0_1, 0_2, ..., 0_n >$ y el máximo valor del arreglo n
Output: Un arreglo ordenado $< a'_1 \leq a'_2 \leq ... \leq a'_n >$

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, para este procedimiento utilizaremos una variable temporal que realizará el intercambio de valores, a continuación se muestra un pseudocódigo de la ejecución del algoritmo.

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
```

1. **Análisis**
   La complejidad computacional del algoritmo es $O(n^2)$, en el peor y mejor caso. Siendo n la cantidad de elementos a ordenar.

2. **Implementación C++**

```cpp
int arr[n];
int i, j, temp;

for(i = 0; i<n; i++) {
   for(j = i+1; j<n; j++){
      if(arr[j] < arr[i]) {
         temp = arr[i];
         arr[i] = arr[j];
         arr[j] = temp;
      }
   }
}
```

Listing 2: BubbleSort.cpp

3. **Implementación Python**

```python
arr = []
i = 0
j = 0
for i in range(tam):
    for j in range(0, tam - i - 1):
        if arr[j] > arr[j + 1] :
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Listing 3: BubbleSort.py

2.2.2. Counting Sort

Input: Un arreglo de n números enteros positivos $< a_1, a_2, ..., a_n >$, un arreglo de n números enteros positivos inicializados en 0 para almacenar el resultado $< 0_1, 0_2, ..., 0_n >$ y el máximo valor del arreglo n
Output: Un arreglo ordenado $< a'_1 \leq a'_2 \leq ... \leq a'_n >$

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
```

El algoritmo countingsort fue propuesto por Harold H. Seward en 1954. Permite ordenar números sin necesidad de realizar comparaciones, usando un contador para almacenar los valores repetidos del arreglo. Los pasos del algoritmo son:

- Obtener el valor máximo del arreglo.
- Crear un arreglo temporal del mismo tamaño del valor máximo, e inicializarlo en 0.
- Recorrer el arreglo principal e incrementar el contador del arreglo temporal por cada elemento del arreglo hallado.
- Recorrer el arreglo principal, y conjuntamente con el arreglo temporal, ubicar los elementos en su posición correcta en base a la cantidad de valores del contador.

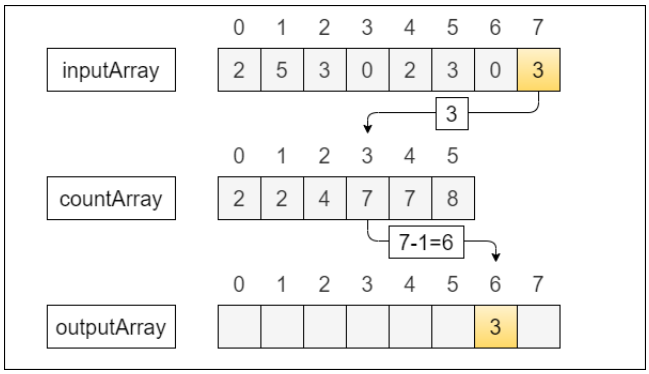


Figura 2: Counting Sort en funcionamiento

1. **Análisis**
   La complejidad computacional del algoritmo es $O(n + k)$, en el peor y mejor caso. Siendo n la cantidad de elementos a ordenar y k el tamaño de arreglo auxiliar(memoria adicional).

- Se puede adaptar tanto a números decimales, así como a caracteres, pero con un costo adicional.
- La mayor limitación del algoritmo lo provoca la cantidad de memoria extra necesaria para realizar el algoritmo, dicho proceso se da cuando el rango entre el mayor y menor valor del arreglo son muy grandes.

2. **Implementación C++**

```cpp
 // retorna el maximo valor de un arreglo de numeros enteros
int maxValueInArray(int list[], size_t sizeList) {
  int max = 0;
  for (int i = 0; i < sizeList; i++) {
    max = (max >= list[i]) ? max : list[i];
  }

  return max;
}

// A = lista no ordenada inicial.
// B = lista ordenada para retornar.
// k = valor maximo de la lista.
// sizeList = tamanho de la lista A
int *counting_sort(int A[], int B[], int k, size_t sizeList){

    // temporal para almacenar los conteos, inicialmente a 0
    int C[k + 1] = {0};

    // contando los valores repetidos
    for (int i = 0; i < sizeList; i++) {
        C[A[i]] = C[A[i]] + 1;
    }

    // acumulando los repetidos para las posiciones
    for (int i = 1; i < (k + 1); i++)
    {
        /* code */
        C[i] = C[i] + C[i - 1];
    }

    // ubicando los valores en sus respectivos indices
    for (int i = (sizeList - 1); i >= 0; i--)
    {
        // asignamos el valor en la posicion ordenada
        B[C[A[i]] - 1] = A[i];
        // descontamos en 1 al valor acumulado(posicion correcta)
        C[A[i]] = C[A[i]] - 1;
    }
    return B;
}
```

Listing 4: countingSort.cpp

3. **Implementación Python**

```python
# retorna el maximo valor de un arreglo de numeros
def maxValueInArray(lista, sizeList):

  max = 0
  # inicializando los contadores a 0
  for x in range(sizeList):
    max = max if max >= lista[x] else lista[x]
  return max

```

```python
# a = arreglo no ordenado, k = valor maximo del arreglo
def counting_sort(a, k):
  # temporal para acumular valores repetidos
  c = []

  # arreglo ordenado para retornar, mismo tamanho del arreglo principal.
  b = [0] * len(a)

  # inicializando los contadores a 0
  for x in range(k + 1):
    c.append(0)

  # contando los valores repetidos
  for x in range(len(a)):
    c[a[x]] = c[a[x]] + 1

  # acumulando los repetidos para las posiciones(indices del arreglo)
  for x in range(1, len(c)):
    c[x] = c[x] + c[x-1]

  # ubicando los valores en sus respectivos indices
  for x in range(len(a) - 1, -1,-1):
    # asignamos el valor en la posicion ordenada
    b[c[a[x]] - 1] = a[x]
    # descontamos en 1 al valor acumulado(posicion correcta)
    c[a[x]] = c[a[x]] - 1
  return b
```

Listing 5: countingSort.py

2.2.3. Heap Sort

Input: Un arreglo de n elementos $< a_1, a_2, ..., a_n >$ que pueden ser ordenado por su clave (carácter, número)
Output: Un arreglo ordenado $< a'_1 \leq a'_2 \leq ... \leq a'_n >$

El algoritmo de heapsort fue creado por los autores J. Williams y refinado por R.W Floyd en el año 1964, se basa en un almacenamiento de datos conocido como Heap, su representación de la raíz es dado por $A[1]$ y dado cualquier índice i de nodo se puede obtener:

- padre $Parent(i) \leftarrow \lfloor i/2 \rfloor$
- hijo izquierdo $Left(i) \leftarrow 2i$
- hijo derecho $Right(i) \leftarrow 2i + 1$

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 3.

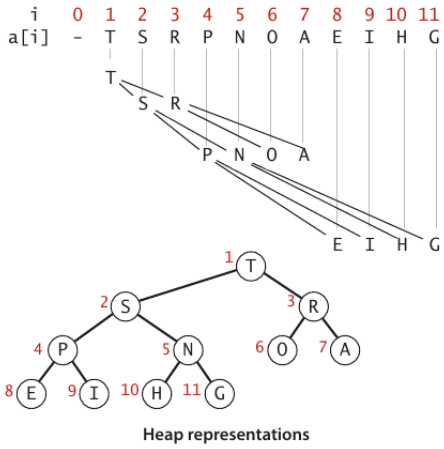


Figura 3: Representación de un Heap como arbol Binario [2]

Además se usan operaciones definidas en una cola de prioridad para elaborar un ordenamiento conocido como heapsort, tales operaciones como Swim y Sink son fundamentales para mantener las propiedades general o invariante. La operación Swim se encarga de preguntar por los hijos si son mayores a sus padres, intercambiando siempre aquellos nodos que no cumplan esta condición y recursivamente ser llamada hasta llegar a la raíz, con respecto a Sink es lo inverso, pregunta por los padres si son menores a sus hijos, si es así, va intercambiando los nodos para que se cumpla la invariante, progresivamente hacia abajo hasta llegar a un nodo hoja, esta función es la que se usa en el heapsort.

Para el ordenamiento Heapsort se requieren dos principales tareas, el ordenamiento por heap Build-Max-Heap y el ordenamiento hacia abajo SortDown aplicando Max-Heapify.

El Build-Max-Heap, el principal objetivo es hacer cumplir la invariante, el cual es que todo nodo padre sea mayor a sus nodos hijo, la construcción del max-heap hará que los elementos mas grandes estén al inicio del array y los elementos menos grandes muy cerca del primer elemento, esto requiere aplicar comparaciones del nodo actual con sus respectivos hijos y saber si el nodo siempre será mayor a sus hijos, en caso de no serlo se producirá el intercambio y recursivamente se ayudará de la función Max-Heapify haciendo la propagación hacia abajo. Además no requiere que se cubra todo el array ya que los nodos hoja al ser un Heap de tamaño uno no requiere realizar un proceso de Max-Heapify. Es así que a lo mucho el costo de realizar el Build-Max-Heap es un coste lineal O(n).

El seudocódigo para el primer proceso Build-Max-Heap es el siguiente:

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**

De acuerdo al seudocódigo 3 el tamaño de heap comenzará con todo los items del arreglo A y a partir de la mitad de los elementos del array hacia el primer elemento son considerados nodos intermedios del árbol y la propia raíz, sin considerar los nodos hoja o terminales ya que estos cumplen la propiedad de invarianza, conforme se va iterando se aplica la función Max-Heapify, (propagación hacia abajo) pero que siempre irá cambiando el nodo evaluado, desde los nodos intermedios hasta llegar a la raíz.

Max-Heapify se aplica al nodo actual que se esta evaluando y recursivamente evalúa los nodos hacia abajo (Sink). En otra palabras se hace cumplir la propiedad por subHeap.

El segundo proceso, es el SortDown aplicando Max-Heapify quien se encarga principalmente de hacer cumplir la invariante, cuando se evalúa Max-Heapify puede darse el caso de no cumplir la

invariante y viole la propiedad max-heap por ese motivo el valor actual $A[i]$ navegará hacia abajo (Sink), así el subárbol con raíz i se convertirá en un max-heap o sub-montículo.

Algorithm 4 Max-Heapify(A,i)

```
1: l ← LEFT(i)
2: r ← RIGHT(i)
3: if l ≤ heap − size[A]andA[l] > A[i] then
4:    largest ← l
5: else
6:    largest ← i
7: end if
8: if r ≤ heap − size[A]andA[r] > A[largest] then
9:    largest ← r
10: end if
11: if largest ≠ i then
12:    exchangeA[i] ↔ A[largest]
13:    Max − Heapify(A, largest)
14: end if
```

Es así que el algoritmo de heapsort usa primero el Build-Max-Heap para hacer cumplir la invariante en todos los nodos $A[1..n]$ donde $n = length[A]$, y hacer que el primero elemento del array sea mayor a todos sus descendientes, dando la posibilidad de intercambiar el elemento final con la raíz haciendo que este elemento ya se encuentre ordenado por ser el mayor de los elementos $A[1]$. Este elemento queda ya descartado del heap $heap - size[A] - 1$.

Sin embargo este elemento intercambiado se tendrá que evaluar su propiedad de Max-Heap por lo cual se aplicará la función Max-Heapify sobre el primer elemento $Max - Heapify(A, 1)$ y propagar hacia abajo, hasta que se cumpla la invariante.

Esto se hará iterando desde el último elemento $length[A]$ hasta llegar al segundo item dejando espacio para el intercambio con el primer elemento $A[1]$. El completo algoritmo de heapsort es dado en el seudocódigo 5

Algorithm 5 Heapsort(A)

```
1: BUILD − MAX − HEAP(A)
2: for i ← length[A] downto 2 do
3:    exchange A[1] ↔ A[i]
4:    heap − size[A] ← heapsize[A] − 1
5:    MAX − HEAPIFY(A, 1)
6: end for
```

En la siguiente figura se puede ver el proceso del Heapsort 4 tanto para el proceso de construcción o BUILD-MAX-HEAP y para el proceso de SortDown con la aplicación de Max-Heapify por cada nodo intermedio.

		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
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

Figura 4: Aplicación del algoritmo Heapsort: Construcción y SortDown [2]

1. **Análisis**
   El procedimiento de Heapsort toma un tiempo de O(nlgn). El Build-Max-Heap toma un tiempo lineal O(n) y es absorbido por el coste del SortDown el cual es mayor, el coste de Build-Max-Heap es lineal ya que es una cota ajustada por el hecho de que la llamada a Max-Heapify interna, va relacionada con la altura de los montículos y 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.

2. **Implementación C++**

```
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i+1;
    int r = 2*i+2;
    if (l < n)
      if (l < n && arr[largest] < arr[l])
          largest = l;
    if (r < n)
      if (r < n && arr[largest] < arr[r])
          largest = r;

    if (largest != i){
        swap(arr[i], arr[largest]);
        heapify(arr,n,largest);
    }
}

void heapsort(int arr[], int n){

    for (int i=n/2-1; i>=0; i--)
        heapify(arr,n,i);
```

```
23      for (int i=n-1; i>0; i--){
24          swap(arr[0], arr[i]);
25          heapify(arr,i,0);
26      }
27  }
```

Listing 6: heapsort.cpp

3. **Implementación Python**

```
1   def heapsort(arr):
2
3       n = len(arr)
4       #construir un maxheap
5       #a partir de la mitad hacia atras y siempre con el mismo n(limite heapify)
6       #se "ordenan los valores" haciendo que los mayores vayan al inicio del arreglo
7       for i in range(n//2 -1, -1, -1):
8           heapify(arr,n,i)
9
10      #extraccion de elementos
11      #ahora intercambiamos de acuerdo al pivote "i" que siempre ira reduciendose
12      #hasta el segundo elemento
13
14      #aplicamos heapify un limite heapify(n) progresivamente mas pequenho
15      #ya que cada elemento intercambiado es extraido como el mayor elemento(raiz)
16      #y van al final del array
17
18      #asi quedarian ordenados de manera ascendente
19      for i in range(n-1,0,-1):
20          arr[i], arr[0] = arr[0], arr[i]
21          heapify(arr,i,0)
```

Listing 7: heapsort.py

2.2.4. Insertion Sort

Tomando lo que define [1] como parte de su primer algoritmo de ordenamiento:

Input: Una secuencia de n números $< a_1, a_2, ..., a_n >$
Output: Una permutación(reordenar) $< a'_1, a'_2, ..., a'_n >$ de la secuencia de entrada tal que $< a'_1 \leq a'_2 \leq ... \leq a'_n >$

Inicia con una lista vacia de una cantidad de elementos. Algo similar a como se ordena una mano de cartas o naipes. Inicia con una mano izquierda vacia y cartas boca abajo sobre la mesa. Se remueve una carta de la mesa al mismo tiempo que se inserta una en la mano y se inserta en la posicion correcta de la mano izquierda. Para encontrar una correcta posicion de la carta, comparemoslo con cada una de las cartas ya en mano, de izquierda a derecha. En todas las veces, las cartas que se mantuvieron en la mano izquierda estan ordenadas ya que estas cartas, fueron originalmente, las que estuvieron encima de la mesa.
El siguiente pseudocódigo 6, que toma como parametro una lista $A[n]$, con n como el tamaño de la lista a ser ordenada. El índice i indica la "carta actual"siendo insertada en la mano. Al principio de cada iteracion del bucle for, el cual esta indexado con i, el subarray a ordenar corresponde a $A[1...i-1]$, la mano actualmente ordenada, y el array sobrante $A[i+1..n]$ corresponde a la pila de cartas aun en la mesa. El hecho es que los elementos $A[1...i-1]$ son los elementos originalmente en la posicion 1 a traves de $i-1$, pero ahora ordenado, conocido como bucle invariante, al inicio de cada iteracion del bucle for se ordena por llegada [1].

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
```

1. **Análisis**
   En el mejor caso de los n items estan en sus respectivos lugares ordenados, entonces Insertion Sort tomará un tiempo lineal o $O(n)$. Esto podría parecer un punto trivial, ya que frecuentemente no se encuentra una lista totalmente ordenda, pero si es importante saberlo ya que es un algoritmo al cual se le puede analizar su mejor caso.
   En el mundo real, Insertion Sort para datos parcialmente ordenados, podria ser un algoritmo efectivo de usar. La eficiencia de Insertion sort se incrementa cuando duplicamos los items.
   Desafortunadamente tomando el peor caso en que todos items se transpongan en su posición a la inversa, Insertion Sort requiere $O(n^2)$ (tiempo cuadrático).

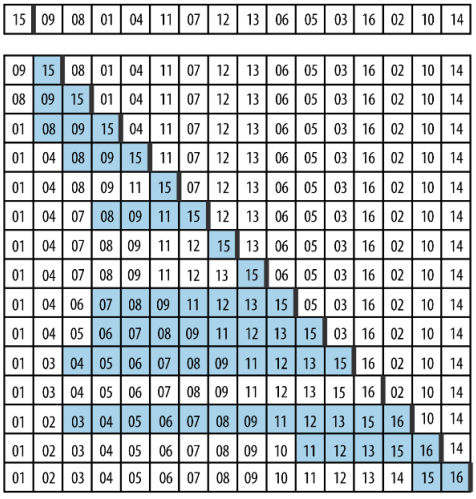


Figura 5: Insertion Sort en ejecución sobre una lista pequeña [3]

Insertion Sort opera ineficientemente por valor de la data debido a la cantidad de memoria que debe intercambiarse cuando toma espacio para un nuevo valor.

2. **Implementación C++**

```
/* Funcion para ordenar un array usando InsertionSort */
void insertionSort(int arr[], int n)
{
    int i, key, j;
    //Partir desde 1 hasta len(arr),tamanho
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
```

```
10
11          /*
12              Mover los elementos de arr[0..i-1] que son
13              mayores que el valor de key, a una posicion
14              adelante de su posicion actual
15          */
16          while (j >= 0 && arr[j] > key)
17          {
18              arr[j + 1] = arr[j];
19              j = j - 1;
20          }
21          arr[j + 1] = key;
22      }
23  }
```

Listing 8: insertionSort.cpp

3. **Implementación Python**

```
1  # Funcion para ordenar un array usando InsertionSort
2  def insertionSort(arr):
3
4      # Partir desde 1 hasta len(arr),tamanho
5      for i in range(1, len(arr)):
6
7          key = arr[i]
8
9          # Mover los elementos de arr[0..i-1] que son
10         # mayores que el valor de key, a una posicion
11         # adelante de su posicion actual
12         j = i-1
13         while j >= 0 and key < arr[j] :
14                 arr[j + 1] = arr[j]
15                 j -= 1
16         arr[j + 1] = key
```

Listing 9: insertionSort.py

2.2.5. Merge Sort

Input: Un arreglo de n números enteros positivos $< a_1, a_2, ..., a_n >$, un valor inferior p y un valor superior r
Output: Un arreglo ordenado $< a'_1 \leq a'_2 \leq ... \leq a'_n >$

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

1: **if** $r > p$ **then**
2: $m \leftarrow p + (r - p)/2$
3: CALL mergeSort(A,p,m)
4: CALL mergeSort(A,m+1,r)
5: CALL merge(A,p,m,r)
6: **end if**

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

```
1: n1 ← q − p + 1
2: n2 ← r − q
3: create arrays L[1..n1+1] and R[1..n2+1]
4: for i ← 1 to n1 do
5:    L[i] ← A[p + i − 1]
6: end for
7: for j ← 1 to n2 do
8:    R[j] ← A[q + j]
9: end for
10: i ← 1
11: j ← 1
12: k ← l
13: while i < n1 and j < n2 do
14:    if L[i] <= R[j] then
15:       A[k] ← L[i]
16:       i ← i + 1
17:    else
18:       A[k] ← R[j]
19:       j ← j + 1
20:    end if
21:    k = k + 1;
22: end while
23: while i < n1 do
24:    A[k] = L[i]
25:    i = i + 1;
26:    k = k + 1;
27: end while
28: while j < n2 do
29:    A[k] = R[j]
30:    j = j + 1;
31:    k = k + 1;
32: end while
```

Fue desarrollado en 1945, por Jhon Von Neumann. Este algoritmo recursivo se basa en la técnica divide y vencerás. Los pasos del algoritmo son:

- Hallar el punto medio del arreglo.
- Dividir el arreglo no ordenado en dos subarreglos cuyo tamaño son aproximadamente la mitad del arreglo principal.
- Llamar recursivamente a cada subarreglo(izquierdo y derecho) hasta encontrar el arreglo contiene un solo elemento, el cual explícitamente indica que el arreglo ya esta ordenado.
- Iniciar el proceso de mezcla de subarreglos ordenados hasta completar la unión del arreglo inicial.

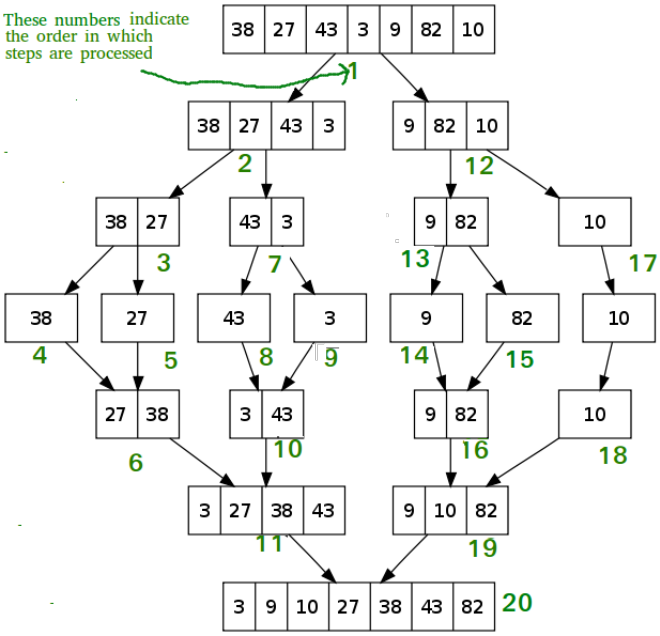


Figura 6: Merge Sort en funcionamiento

1. **Análisis**
   La complejidad computacional del algoritmo en el mejor caso, en el caso promedio y en el peor de los caso es siempre $O(nlogn)$.

   - El ordenamiento no trabaja in situ, es decir que no realiza un cambio en el mismo arreglo directamente, sino que hace una copia antes de actualizar el arreglo.
   - Para mezclar los dos subarreglos, ambos deberán estar ordenados.
   - Mergesort requiere un espacio auxiliar (n), en comparación con Heapsort que requiere un espacio de (1)

2. **Implementación C++**

```cpp
// arr, arreglo de numeros
// l, indice del limite inferior del arreglo
// m, indice medio entre el limite inferior l y superior r, del arreglo
// r, indice del limite superior del arreglo
void merge(int arr[], int l, int m, int r)
{
  int n1 = m - l + 1; // cantidad de elementos del arreglo izquierdo.
  int n2 = r - m; // cantidad de elementos del arreglo derecho.

  // creacion de los arreglos temporales izquierdo y derecho respectivamente.
  int L[n1], R[n2];

  // copia de los valores en el arreglo izquierdo(arreglo esta ordenado)
  for (int i = 0; i < n1; i++)
    L[i] = arr[l + i];
  // copia de los valores en el arreglo derecho(arreglo esta ordenado)
  for (int j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];

  // indice para el arreglo izquierdo
  int i = 0;

```

```
23    // indice para el arreglo derecho
24    int j = 0;
25
26    // indice para el arreglo ordenado final
27    int k = l;
28
29    // copiar elementos en el arreglo principal, con elementos del arreglo izquierdo
         y/o derecho.
30    while (i < n1 && j < n2) {
31      if (L[i] <= R[j]) {
32        arr[k] = L[i];
33        i++;
34      }
35      else {
36        arr[k] = R[j];
37        j++;
38      }
39      k++;
40    }
41
42    // copiar elementos en el arreglo principal, con elementos del arreglo izquierdo
        (restantes)
43    while (i < n1) {
44      arr[k] = L[i];
45      i++;
46      k++;
47    }
48
49    // copiar elementos en el arreglo principal, con elementos del arreglo derecho(
        restantes)
50    while (j < n2) {
51      arr[k] = R[j];
52      j++;
53      k++;
54    }
55  }
56
57
58  // arr, arreglo no ordenado de numeros.
59  // l, indice inferior del arreglo
60  // r, indice superior del arreglo
61  void mergeSort(int arr[],int l,int r){
62    if(l>=r){
63      return; // retorno de la recursividad(paso base)
64    }
65    int m =l+ (r-l)/2; // calculando el indice del punto medio del arreglo
66    mergeSort(arr,l,m); // llamada recursiva del arreglo izquierdo
67    mergeSort(arr,m+1,r); // llamada recursiva del arreglo derecho
68    merge(arr,l,m,r); // combinar arreglo izquierdo y derecho
69  }
```

Listing 10: mergeSort.cpp

3. Implementación Python

```
1  # arr,  arreglo no ordenado de numeros
2  def mergeSort(arr):
3    if len(arr) > 1:
4
5      # Encontrando el indice del punto medio del arreglo
6      mid = len(arr)//2
7
8      # arreglo izquierdo
9      L = arr[:mid]
```

```
10
11      # arreglo derecho
12      R = arr[mid:]
13
14      # llamada recursiva del arreglo izquierdo
15      mergeSort(L)
16
17      # llamada recursiva del arreglo derecho
18      mergeSort(R)
19
20      i = j = k = 0
21
22      # copiar elementos en el arreglo principal, con elementos del arreglo
        izquierdo y/o derecho.
23      while i < len(L) and j < len(R):
24        if L[i] < R[j]:
25          arr[k] = L[i]
26          i += 1
27        else:
28          arr[k] = R[j]
29          j += 1
30        k += 1
31
32      # copiar elementos en el arreglo principal, con elementos del arreglo
        izquierdo(restantes)
33      while i < len(L):
34        arr[k] = L[i]
35        i += 1
36        k += 1
37      # copiar elementos en el arreglo principal, con elementos del arreglo derecho(
        restantes)
38      while j < len(R):
39        arr[k] = R[j]
40        j += 1
41        k += 1
```

Listing 11: mergeSort.py

2.2.6. Quick Sort

Input: Un arreglo de n elementos $< a_1, a_2, ..., a_n >$, posición inicial p, posición final $r \leftarrow length[A]$
Output: Un arreglo ordenado $< a'_1 \leq a'_2 \leq ... \leq a'_n >$

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. La pequeña desventaja es que es bastante sensible al balance de los datos, si no estuvieran muy des-balanceados, es decir diversas tamaños del proceso de partición, podría ocasionar un tiempo considerable de orden $O(n^2)$

QuickSort usa como base el concepto de divide y conquista, en este caso dado un array $A[p..r]$.

- Dividir: divide el arreglo en dos sub-arreglos de tamaño $A[p..q-1]$ y $A[q+1..r]$ donde $A[q]$ es mayor al primer sub-array y menor e igual al segundo sub-array. Computar q como parte de la partición
- Conquistar: Ordenar los subarrays por llamadas recursivas a quicksort
- Combinar: Desde que el ordenamiento es en el lugar no requiere la combinación. El array $A[p..r]$ estaría ya ordenado.

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 ← Partition(A, p, r)
3:     QUICKSORT(A, p, q − 1)
4:     QUICKSORT(A, q + 1, r)
5: end if
```

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
```

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, la primer región es la que contendrá los elementos menores al pivote, la segunda región contendrá los elementos mayores e igual al pivote, la 3era región es aquella que contendrá los elementos que serán comparados con el pivote y la última región es el que contiene el pivote.

Por ello se satisface las siguientes propiedades aplicados a las líneas del 3 al 6 en el seudocódigo 10 y como se observa en la figura 7

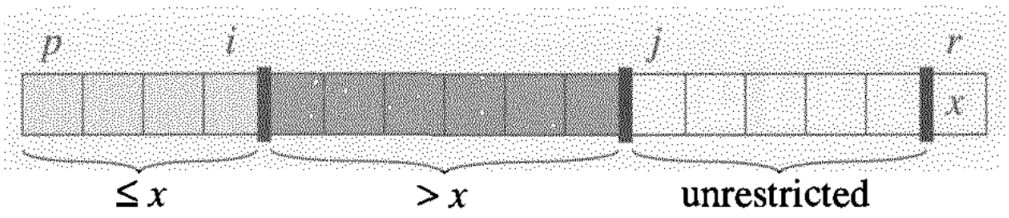


Figura 7: Regiones en el Partition

- Todos los elementos k que este entre p y el índice i son menores al pivote x.

  IF $p \leq k \leq i$ THEN $A[k] \leq x$

- Todos los elementos k que estén entre $i + 1$ y $j - 1$ son mayores al pivote x

  IF $i + 1 \leq k \leq j - 1$ THEN $A[k] > x$

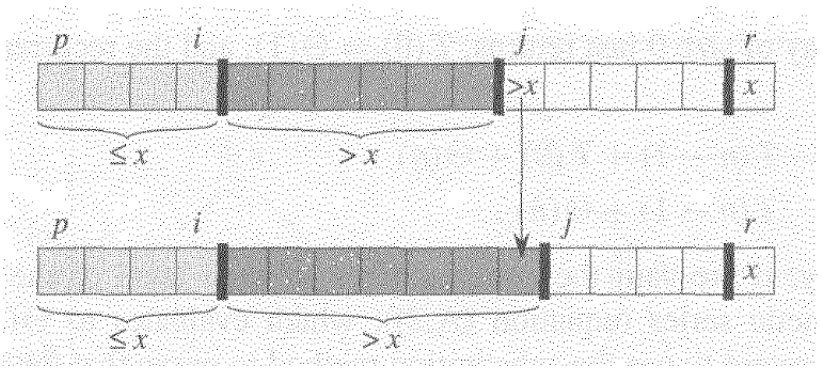
- Todos los elementos k entre $j + 1$ y $r - 1$ son elementos que deben ser comparados con el pivote x

  IF $j + 1 \leq k \leq r - 1$ THEN $compare(A[k], x)$

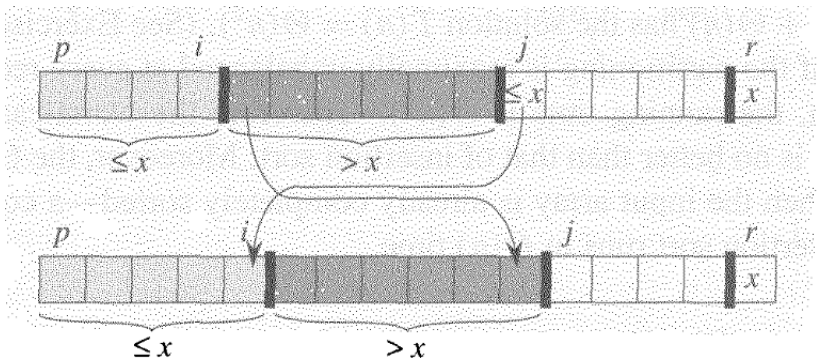
- El elemento k ubicado en r, es el pivote actual x

  IF $k = r$, then $A[k] = x$

Una manera de ejemplificar los casos dados por la partición, dado en la siguiente figura 8, el primero es cuando $A[j] > x$, lo cual incrementa el j añadiendolo al intervalo de los mas grandes que x, el segundo es cuando $A[j] <= x$, aquí el primer indice i es incrementado con el objetivo de intercambiar uno de los items del intervalo grande por el elemento contenido en j (el menor al pivote) así el intervalo de los más pequeños habrá ganado un item adicional. Posteriormente el j es incrementado para la siguiente item a comparar con el pivote.



(a) Caso para intervalo grande

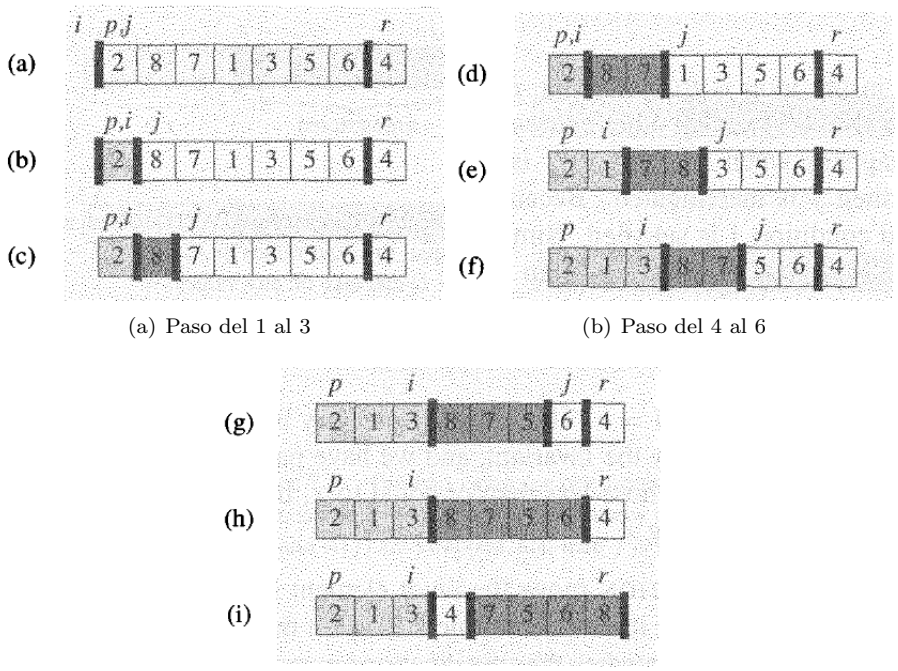


(b) Caso para Intervalo pequeño

Figura 8: Casos dados en Partition [2]

Finalmente, el procedimiento de ordenamiento de Quicksort, donde se observa paso a paso la aplicación de las particiones, dado en la figura 9, como se observa en la 9 en la figura a se da la configuración inicial donde tanto los indices como posiciones de fin e inicio se incializan. en la figura b se da el primer avance del límite j haciendo que este produzca un intercambio entre el mismo elemento que en este caso es el número 2, incrementado el indice $i$una unidad. En la figura c y d el indice j avanza dos posiciones por ser elementos mayores al pivote y respetar la invarianza, en las figuras e y f se dan intercambios al ser elementos menores al pivote, que corresponden a los números 1 y 3. En la figura g y h se repite el proceso de avance de la indice j por tener elementos que cumplen la condición y finalmente en la figura i se realiza el proceso de avance del indice i para ser intercambiado con el último elemento o pivote, retornando la posición actual de j.

1. **Análisis**



(a) Paso del 1 al 3

(b) Paso del 4 al 6

(c) Paso del 7 al 9

Figura 9: Pasos del proceso de ordenamiento usando QuickSort

Para el algoritmo de QuickSort el tiempo de ejecución dependerá mucho de la forma en como los datos se encuentren, en este caso cuan balanceados los datos se presentan al inicio del ordenamiento.

Por ello lo separaremos por mejor, peor y caso promedio:

Tiempo en el Peor Caso:

Este caso ocurre cuando se da una partición que genera n-1 elementos en un aubarray y 0 elementos en el otro subarray, además esto ocurre en las demás llamadas recursivas, siendo un coste de partición de $\Theta(n)$, y la llamada del subarray de cero elementos sería $T(0) = \Theta(1)$, siendo el tiempo de la llamada de recurrencia:

$$T(n) = T(n-1) + T(0) + \Theta(n) T(n) = T(n-1) + \Theta(n)$$

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 $O(n)$.

Tiempo en el 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 $T(n) = O(nlgn)$.

Tiempo en el caso Promedio:

La clave para entender el coste promedio cercano al mejor caso es saber como se refleja el balanceo en la recurrencia.

Si se da una partición en relación de 9 a 1 partes se obtiene la siguiente recurrencia $T(n) \leq T(9n/10) + T(n/10) + cn$ este cuenta con el término oculto del coste de partición Θn , como se observa en la figura, cada nivel al barre los n elementos tendría un coste de cn hasta que sea alcanzado el primer nodo hoja dado en la profundidad $log_10(n) = \Theta(lg(n))$ a partir de aquí cada nivel tendría a lo mucho un coste de $\leq c(n)$. La recursión termina en profundidad $log_(10/9)(n) = \Theta(lg(n))$ por ese motivo cualquiera sea el nivel siempre el coste será de $\Theta(lg(n))$ multiplicado por el coste del nivel cn siendo asi un coste total de $\Theta(nlg(n))$. siempre y cuando exista almenos una división y sea constante durante todos los niveles del árbol.

2. **Implementación C++**

```cpp
int partition(int arr[], int low, int high)
{
  //pivote
  int pivot  = arr[high];

  //indice del elemento mas pequenio e indica la correcta posicion del pivote
    encontrado
  int i  = (low - 1);

  for (int j = low; j <= high -1; j++)
  {
    //si el elemento es mas pequenio que el pivote
    if (arr[j] < pivot)
    {
      i++; //incrementamos el indice del elemento mas pequenio
      swap(arr[i], arr[j]);
    }
  }
  swap(arr[i+1], arr[high]);
  return (i+1);
}

void quicksort(int arr[], int low, int high)
{
  if (low < high)
  {
    int pi = partition(arr, low, high);

    //ordenar los elementos antes y despues del pivote
    quicksort(arr, low, pi - 1);
    quicksort(arr, pi+1, high);
  }
}
```

Listing 12: quicksort.cpp

3. **Implementación Python**

```python
#Puntos de inicio y final y el ultimo elemento del array
def partition(start, end, array):
    #Inicializando indice del pivote inicial
    pivot_index = start
    pivot = array[pivot_index]

    #iterar hasta que el ptro de inicio cruce con el ptro final,
    #intercambiamos el pivote con el elemento sobre el ptro final
    while start < end:

        #incrementamos el ptro de inicio hasta que encuentre
        #un elemento mas grande que el pivote
        while start < len(array) and array[start] <= pivot:
```

```
            start += 1

        #decrementamos el puntero final hasta que que encuentre
        #un elemento menos que el pivote
        while array[end] > pivot:
            end -= 1

        #Si el start y end no se han cruzado, intercambiamos los numeros
        #del ptro start y end
        if (start < end):
            array[start], array[end] = array[end], array[start]

    #Cambiamos el pivote con el elemento sobre el ptro final
    #Esto colocal el pivote en el correcto lugar de orden
    array[end], array[pivot_index] = array[pivot_index], array[end]

    #retorna el ptro final para dividir el array en 2
    return end

def quicksort(start, end, array):
    if (start < end):
        #p es el indice de particion, array[p] es el correcto lugar
        p = partition(start, end, array)

        #ordenar los elemento antes y despues del pivote
        quicksort(start, p-1, array)
        quicksort(p+1, end, array)
```

Listing 13: quicksort.py

2.2.7. Selection Sort

Una estrategia comun de ordenamiento es seleccionar el valor mas grande desde un array $A[0, n]$ e intercambiar su ubicación con el elemento que esta mas a la derecha $A[n-1]$. Este proceso se repite, subsecientemente, asi avanzar en cada pequeño rango sucesivo $A[0, n-1]$ hasta que A este ordenado, este es un caso del enfoque Greedy [3].

Algorithm 11 SELECTION-SORT(A)

for $i \leftarrow 0$ to $N-1$ **do**
  $minIdx \leftarrow i$
  for $j \leftarrow i+1$ to N **do**
    if $A[j] < A[minIdx]$ **then**
      $minIdx \leftarrow j$
    end if
  end for
  $A[j+1] \leftarrow key$
end for

1. **Análisis**
   Es el algoritmo de ordenamiento mas lento segun [3] en los algoritmos que nombra, requiere un tiempo $O(n^2)$ aun en el mejor caso, la lista totalmente ordenada, siempre seleccionando al elemento de mayor tamaño, tomando $n-1$ comparaciones, muchas de estas comparaciones son por desgaste, porque si un elemento es mas pequeño al segundo, es imposible ser el elemento de mayor tamaño, teniendo un desarrollo pobre al momento de continuar con las comparaciones.

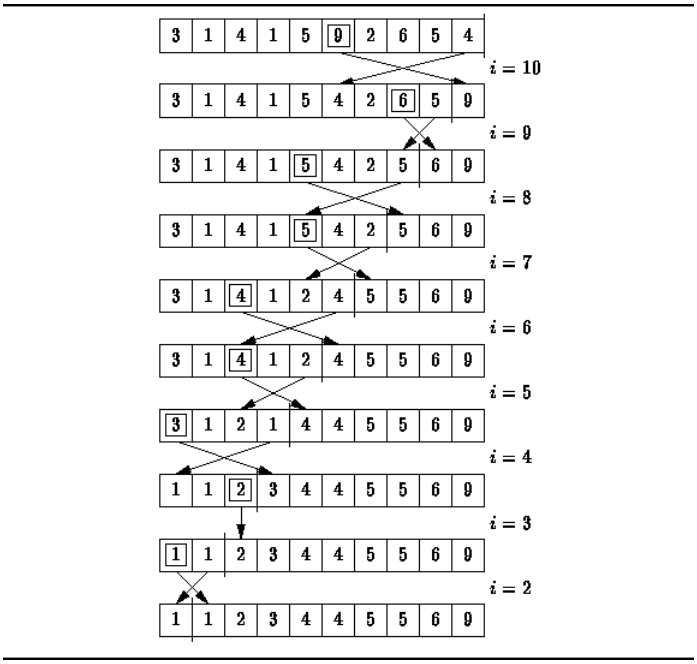


Figura 10: Selection Sort en ejecución sobre una lista pequeña

2. Implementación C++

```cpp
// funcion para intercambiar posicion de clave(key)
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // Mover uno por uno entre el limite del subarray desordenado
    for (i = 0; i < n-1; i++)
    {
        // Encontrar el minimo elemento en el array desordenado
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Intercambiar el elemento minimo encontrado con el primer elemento
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Listing 14: selectionSort.cpp

3. Implementación Python

```python
def selectionSort(A):
  # Recorrer todo el tamanho del array
```

```
  for i in range(len(A)):

    # Encontrar el minimo elemento en el array desordenado
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Intercambiar el elemento minimo encontrado con el primer elemento
    A[i], A[min_idx] = A[min_idx], A[i]
```

Listing 15: selectionSort.py

2.3. Comparación de tiempo de procesamiento

2.3.1. Comparación de dos lenguajes de programación por cada algoritmo

Para todos los casos presentados en cada implementación de los algoritmos, se ejecutaron utilizando los archivos generados anteriormente, iterando sobre todos ellos, luego en cada ejecucion de iteracion se realizo el calculo del tiempo en cada lenguaje de programacion dando como resultado un archivo con la cantidad de elementos y el tiempo de ejecución, en todos los casos se utilizó la libreria *matplotlib.pyplot* de Python.

1. **Bubble Sort:** Para calcular el tiempo del Bubble Sort en lenguaje C++ se utilizó la librería <chrono> que puede obtener un punto de tiempo que se representa en el tiempo ejecución, con el metodo now() podemos obtener los milisegundos exactos del inicio y fin del tiempo transcurrido

```

#include <chrono>

  auto t1 = std::chrono::high_resolution_clock::now();
  //... (Bubble Sort)
  auto t2 = std::chrono::high_resolution_clock::now();

  //calcular, convertir e imprimir tiempo
  std::chrono::duration<double> elapsed = t2 - t1;
  float miliseg = (float)(elapsed.count());
  cout<<archivos[indx]<<","<<setprecision(4)<<miliseg<<endl;
```

Listing 16: Ejecución C++ del Bubble Sort

En el caso del cálculo de tiempo en el lenguaje Python se utilizó la librería 'time' que gracias al metodo time.time() podemos obtener el tiempo de la ejecución en milisegundos

```

import time

  tiempo_ini = time.time()
  # ... (Bubble Sort)
  tiempo_fin = time.time()

  # calcular, convertir e imprimir tiempo
  print(str(archivo) + "," + str(round((float)(tiempo_fin - tiempo_ini), 5)))
```

Listing 17: Ejecución Python del Bubble Sort

La comparacion del tiempo de ejecucion del algoritmo Bubble Sort en los lenguajes C++ y Python, tiene como resultado una gran diferencia de tiempos con respecto a la cantidad de datos a ordenar, esto se debe a que el compilador de C++ es del propio sistema operativo y permite una ejecución directa de funciones, sin embargo Python tiene un compilador adaptado a una sintaxis resumida de funciones básicas del lenguaje maquina

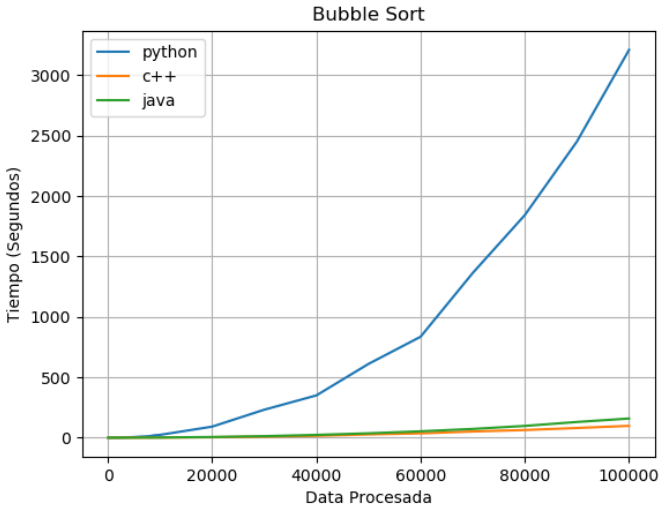


Figura 11: Bubble Sort - Tiempo de Ejecución

2. **Counting Sort:** Los tiempos de ejecución para las distintas dimensiones del arreglo se realizaron en el orden de tiempo que demanda su coste O(n + k). Cabe resaltar que la ejecución del algoritmo en Python tiene tiempos más elevados que los ejecutados en C++.

```
        // declaracion del tiempo de ejecucion del algoritmo
        clock_t start, end;
        start = clock();
        counting_sort(arr,B,maxValue,generate[k]);
        end = clock();

        // tiempo de ejecucion obtenido
        double time_taken = double(end - start) / double(CLOCKS_PER_SEC);

        cout << "Se ordeno el array de tamanho " + to_string(n) + " :";
        cout << "\n Tiempo tomado por el programa es: " << fixed
             << time_taken << setprecision(8);
        cout << " segundos " << endl;
```

Listing 18: Ejecucion C++ de CountingSort

```

# declaracion del tiempo de ejecucion del algoritmo
start = time.process_time()
counting_sort(arr, maxValueInArray(arr, len(arr)))
result = time.process_time() - start

print("Se ordeno el array de tamanho " + str(len(arr)) + ": Tiempo tomado por el
   programa es... " + str(time.process_time() - start) + " segundos")
```

Listing 19: Ejecucion Python de CountingSort

En la siguiente gráfica 12 se muestra la comparación de los tiempos tomados para el algoritmo countingsort en 2 lenguajes Python y C++, donde se observa un comportamiento lineal cuando se ejecuto con C++ y una curva cercana a la lineal cuando se ejecutó con Python.

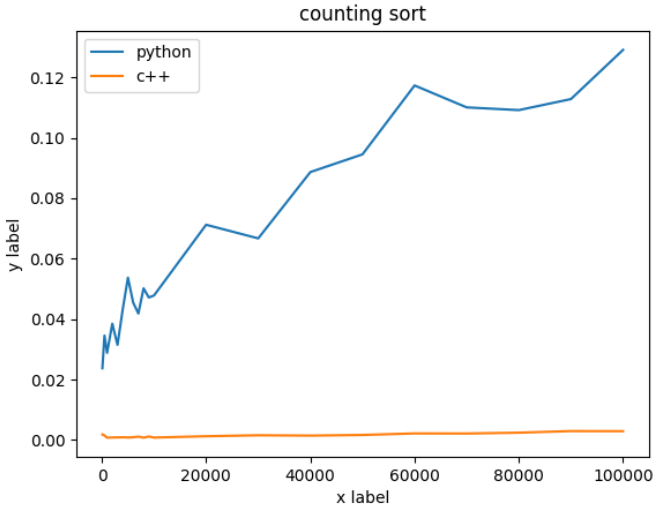


Figura 12: CountingSort - Tiempo de Ejecución

3. **Heap Sort :** Los tiempos de ejecución para los distintos tamanios de arreglo se realizaron en el orden de tiempo que demanda su coste O(nlgn), además su costo es muy similar al del ordenamiento por quicksort.

```
        // declaracion del tiempo de ejecucion del algoritmo
        clock_t start, end;

        start = clock();
        heapsort(arr, n);
        end = clock();

        // tiempo de ejecucion obtenido
        double time_taken = double(end - start) / double(CLOCKS_PER_SEC);

        cout << "Se ordeno el array de tamanho " + to_string(n) + " :";
        cout << "\n Tiempo tomado por el programa es: " << fixed
             << time_taken << setprecision(8);
        cout << " segundos " << endl;
```

Listing 20: Ejecucion C++ de Heapsort

A pesar de contar con un coste bajo para C++, heapsort en python tomaría un gran tiempo si la carga de datos es considerablemente enorme.

```
# declaracion del tiempo de ejecucion del algoritmo
start = time.process_time()
heapsort(arr)
result = time.process_time() - start

print("Se ordeno el array de tamanho " + str(len(arr)) + ": Tiempo tomado por el
   programa es... " + str(time.process_time() - start) + " segundos")
```

Listing 21: Ejecucion Python de Heapsort

En la siguiente gráfica 13 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, sin conocer ello y el orden de coste es importante pero también el lenguaje usado en la implementación, Python al ser un lenguaje de alto nivel

requiere el casting de sus funciones a funciones de librería base como C y poder recién realizar la ejecución.

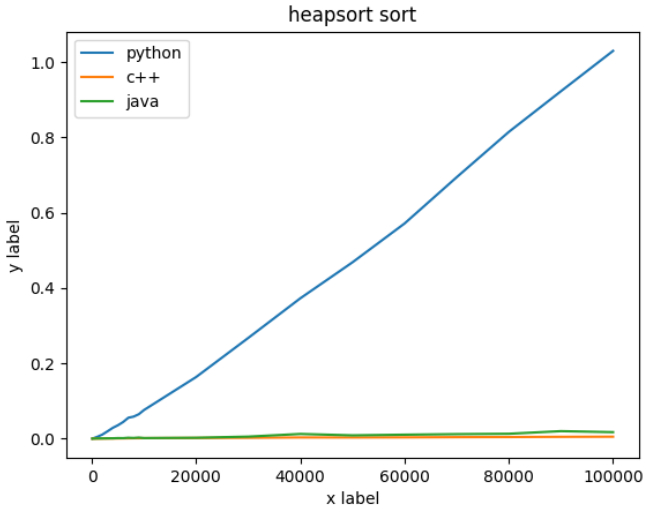


Figura 13: Heapsort - Tiempo de Ejecución

4. **Insertion Sort:** En el caso de la ejecución del Insertion Sort se utilizó para C++ la variable clock_t que toma las pulsaciones de reloj del procesador luego pasado a segundos

```
        // solo para tener el numero de array, igualmente el valor ya es conocido
        int n = sizeof(arr) / sizeof(arr[0]);

        // declaracion del tiempo de ejecucion del algoritmo
        clock_t start, end;

        start = clock();
        insertionSort(arr, n);
        end = clock();

        // tiempo de ejecucion obtenido
        double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
```

Listing 22: Ejecucion C++ de Insertion Sort

Igualmente en el caso de Python se toma la libreria *time* para obtener la cantidad de tiempo de ejecución del algoritmo

```
  # declaracion del tiempo de ejecucion del algoritmo
  start = time.process_time()
  insertionSort(arr)
  result = time.process_time() - start
```

Listing 23: Ejecucion Python de Insertion Sort

Como resultado final se muestra la figura 14 la representación por cada lenguaje de programación ejecutado y medido, se incluyó Java solo con fines de comparacion ya que la finalidad solo era entre C++ y Python, demostrado ya en las pruebas anteriores en la generacion de los archivos se nota que el algoritmo hecho en C++ es mucho más rápido que en Python, ocurre porque la ejecución de C++ pasa directamente compilado para lenguaje máquina a diferencia de Python que es compilado en un lenguaje intermedio para luego recien pasar a lenguaje máquina.

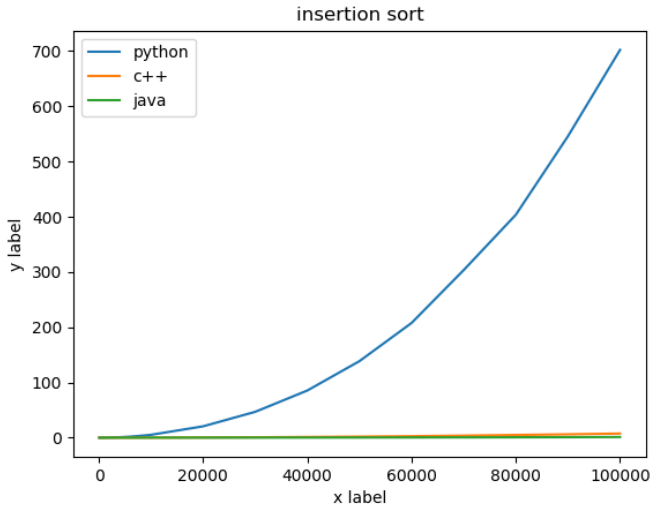


Figura 14: Insertion Sort - Tiempo de ejecución

5. **Merge Sort:** Los tiempos de ejecución para las distintas dimensiones del arreglo se realizaron en el orden de tiempo que demanda su coste O(nlogn). Cabe resaltar que la ejecución del algoritmo en Python tiene tiempos más elevados que los ejecutados en C++.

```
1
2       // calculando el tiempo de ejecucion del algoritmo.
3       clock_t start, end;
4       start = clock();
5       mergeSort(arr, 0, n - 1);
6       end = clock();
7
8       // tiempo de ejecucion obtenido
9       double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
10
11      cout << "Se ordeno el array de tamanho " + to_string(n) + " :";
12      cout << "\n Tiempo tomado por el programa es: " << fixed
13           << time_taken << setprecision(8);
14      cout << " segundos " << endl;
```

Listing 24: Ejecucion C++ de MergeSort

```
1
2  # declaracion del tiempo de ejecucion del algoritmo
3  start = time.process_time()
4  mergeSort(arr)
5  result = time.process_time() - start
6
7  print("Se ordeno el array de tamanho " + str(len(arr)) + ": Tiempo tomado por el
      programa es... " + str(time.process_time() - start) + " segundos")
```

Listing 25: Ejecucion Python de MergeSort

En la siguiente gráfica 15 se muestra la comparación de los tiempos tomados para el algoritmo mergesort en 2 lenguajes Python y C++, donde se observa un comportamiento lineal cuando se ejecuto con Python y una curva logarítmica cuando se ejecutó con C++.

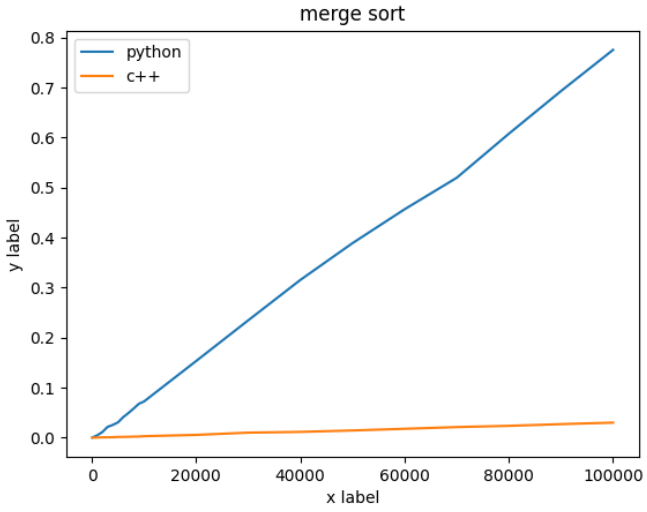


Figura 15: MergeSort - Tiempo de Ejecución

6. **Quick Sort:** El tiempo tomado para el quicksort fue el tomado de manera similar, para ello en el mejor caso podría comportarse muy similar al mergesort o heapsort $O(nlgn)$ pero en el peor caso podría comportarse como el selection sort (n^2). El siguiente bloque de código muestra la captura de tiempo en C++.

```
        // declaracion del tiempo de ejecucion del algoritmo
        clock_t start, end;
        start = clock();

        quicksort(arr,0, n-1);

        end = clock();

        // tiempo de ejecucion obtenido
        double time_taken = double(end - start) / double(CLOCKS_PER_SEC);

        cout << "Se ordeno el array de tamanho " + to_string(n) + " :";
        cout << "\n Tiempo tomado por el programa es: " << fixed
            << time_taken << setprecision(8);
        cout << " segundos " << endl;
```

Listing 26: Ejecucion C++ de Quicksort Sort

Para el caso de la implementación en Python el crecimiento de datos es lineal pero con los últimos datos de tamanio mayor el tiempo se acentúa más, por ello se debe tener cuidado cuando se trabaje con una gran cantidad de datos en temas de ordenamiento, el lenguaje escogido puede ser muy sensitivo a la carga de datos como es el caso de Python. El siguiente bloque de código muestra la captura de tiempo de la llamada a quicksort en Python.

```
# declaracion del tiempo de ejecucion del algoritmo
start = time.process_time()
quicksort(0,len(arr)-1,arr)
result = time.process_time() - start

print("Se ordeno el array de tamanho " + str(len(arr)) + ": Tiempo tomado por el
   programa es... " + str(time.process_time() - start) + " segundos")
```

Listing 27: Ejecucion Python de Quicksort Sort

Finalmente se muestra la figura 16 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.

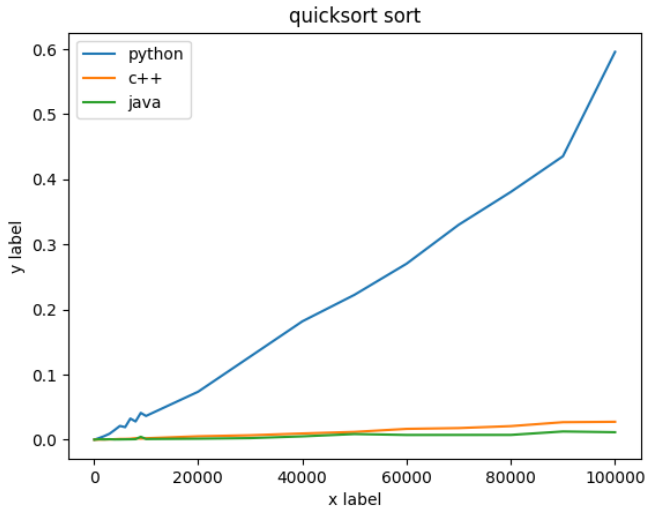


Figura 16: Selection Sort - Tiempo de ejecución

7. **Selection Sort:** En el caso de la ejecución del Selection Sort se utilizó para C++ la variable clock_t que toma las pulsaciones de reloj del procesador luego pasado a segundos

```
        // solo para tener el numero de array, igualmente el valor ya es conocido
        int n = sizeof(arr)/sizeof(arr[0]);

        // declaracion del tiempo de ejecucion del algoritmo
        clock_t start, end;
        start = clock();

        selectionSort(arr, n);

        end = clock();
```

Listing 28: Ejecucion C++ de Selection Sort

Igualmente en el caso de Python se toma la libreria *time* para obtener la cantidad de tiempo de ejecución del algoritmo

```
  # declaracion del tiempo de ejecucion del algoritmo
  start = time.process_time()
  selectionSort(A)
  result = time.process_time() - start
```

Listing 29: Ejecucion Python de Selection Sort

Como resultado final se muestra la figura 17 la representación por cada lenguaje de programación ejecutado y medido, se incluyó Java solo con fines de comparación ya que la finalidad solo era entre C++ y Python, demostrado ya en las pruebas anteriores en la generacion de los archivos se nota que el algoritmo hecho en C++ es mucho más rápido que en Python, ocurre porque la ejecución de C++ pasa directamente compilado para lenguaje máquina a diferencia de Python que es compilado en un lenguaje intermedio para luego recien pasar a lenguaje máquina.

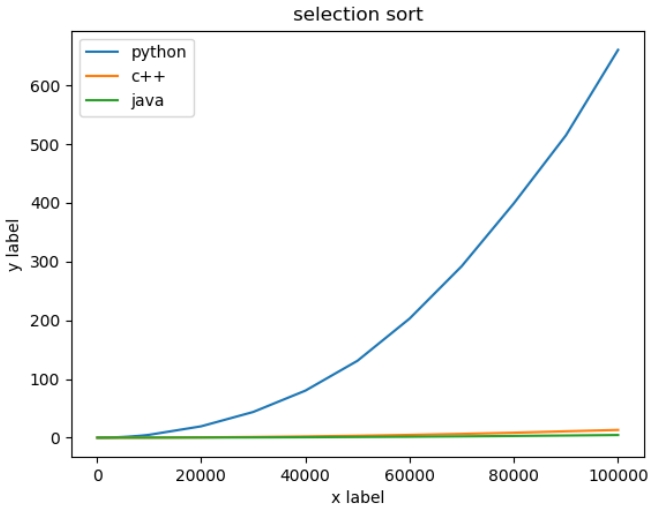


Figura 17: Selection Sort - Tiempo de ejecución

2.3.2. Comparación de tiempo de procesamiento de todos los algoritmos implementados en cada lenguaje de programación

Finalizando la revisión de los algoritmos directamente en sus implementaciones en C++ y Python, se observa que desde el análisis en cada uno de los algoritmos presentados se puede observar como el algoritmo mas ineficiente a Bubble Sort y los mas eficientes a los que están con tiempo logarítmico (HeapSort, MergeSort, QuickSort) incluso tomar a Coounting Sort en estos casos pero su tendencia solo aplica a este tipo de ordenamiento con números mayores a 0 y enteros además de la dependencia del consumo de memoria.

Respectivamente en la Figura 18 de C++ se observa la gráfica menos notoria en algunos algoritmos pesados como Insertion Sort, por el resultado en bajo tiempo respecto a la Figura 19 que muestra mayor notoriedad en los algoritmos ya que sus resultados fueron mas extensos respecto al lenguaje C++.

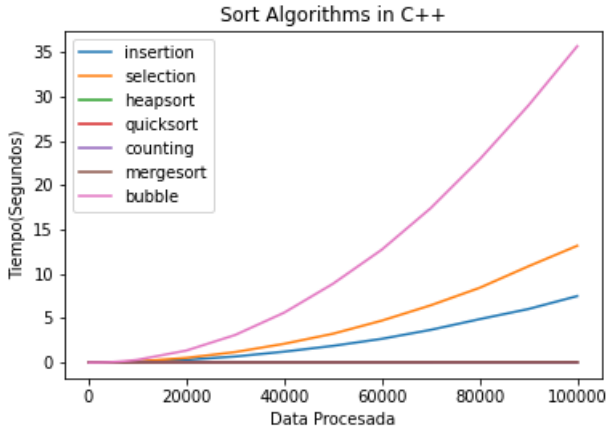


Figura 18: Algoritmos de Ordenamiento implementados en C++

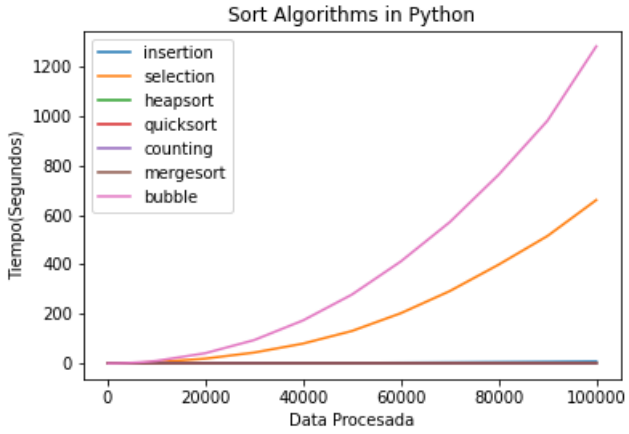


Figura 19: Algoritmos de Ordenamiento implementados en Python

En conclusión cada uno de los algoritmos presentados cumplen un mismo objetivo en ordenar los elementos de un arreglo con diferentes enfoques de resolución y ejecución, mostrando tanto sus beneficios como dificultades, siempre realizando la comparación con el peor caso y la revisión en tiempos de ejecución nos mostro que tan factible es elegir un algoritmo respecto a otro.

Referencias

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

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

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

Listings