

Universidad Nacional de San Agustín



Estructura de Datos Avanzados **TEMA: Algoritmo y costo computacional**

Laboratorio Grupo B

Docente:

FRANKLIN LUIS ANTONIO CRUZ GAMERO

Realizado por:

Edwar Jhoel Vargas Herhuay
Fabian Vladimir Florez Aguilar
Luis Angel Bustamente Torres

Arequipa – Perú

2021

Universidad Nacional de San Agustín	1
1. CONCEPTOS BÁSICOS	3
1.1 LENGUAJE DE PROGRAMACIÓN	3
1.2 COMPLEJIDAD COMPUTACIONAL	3
2. ALGORITMOS DE ORDENAMIENTO	4
2.1 BUBBLESORT	4
2.2 HEAPSORT	7
3.3 INSERTIONSORT	11
3.4 SELECTIONSORT	14
3.5 SHELLSORT	17
3.6 MERGESORT	21
3.7 QUICKSORT	25
3. CONCLUSIONES	30
3.1 Rendimiento en Python	31
Rendimiento en C++	31
Rendimiento en Java	31
BIBLIOGRAFÍA	32

1. CONCEPTOS BÁSICOS

1.1 LENGUAJE DE PROGRAMACIÓN

Un lenguaje de programación es un lenguaje de computadora que los programadores utilizan para comunicarse y para desarrollar programas de software, aplicaciones, páginas webs, scripts u otros conjuntos de instrucciones para que sean ejecutadas por los ordenadores.

- python
- Java
- C++

1.2 COMPLEJIDAD COMPUTACIONAL

La teoría de la complejidad computacional o teoría de la complejidad informática es una rama de la teoría de la computación que se centra en la clasificación de los problemas computacionales de acuerdo con su dificultad inherente, y en la relación entre dichas clases de complejidad.

Un problema se cataloga como "inherentemente difícil" si su solución requiere de una cantidad significativa de recursos computacionales, sin importar el algoritmo utilizado. La teoría de la complejidad computacional formaliza dicha aseveración, introduciendo modelos de computación matemáticos para el estudio de estos problemas y la cuantificación de la cantidad de recursos necesarios para resolverlos, como tiempo y memoria.

Una de las metas de la teoría de la complejidad computacional es determinar los límites prácticos de qué es lo que se puede hacer en una computadora y qué no. Otros campos relacionados con la teoría de la complejidad computacional son el análisis de algoritmos y la teoría de la computabilidad. Una diferencia significativa entre el análisis de algoritmos y la teoría de la complejidad computacional, es que el primero se dedica a determinar la cantidad de recursos requeridos por un algoritmo en particular para resolver un problema, mientras que la segunda, analiza todos los posibles algoritmos que pudieran ser usados para resolver el mismo problema.

- **Clase L (D Log Space):** Son aquellos problemas que pueden ser resueltos por una MT determinista utilizando una cantidad de espacio o memoria logarítmicamente proporcional al tamaño de la entrada: $\log(n)$, donde n es el tamaño de la entrada; en estos problemas si existe una solución es única.
- **Clase NL (Nondeterministic Logarithmic space):** Son los problemas de decisión que pueden ser resueltos en espacio $\log(n)$, por una MT no determinista tal que la solución si existe es única.

- **Clase P (Polynomial Time):** Son todos aquellos problemas de decisión que pueden ser resueltos por una MT determinista en un período de tiempo polinómico. Estos problemas son tratables, es decir se pueden resolver en tiempos razonable; buena parte de los problemas de ordenamiento, priorización y búsqueda caben dentro de esta clase.
- **Clase NP (Non Deterministic Polynomial Time):** Formalmente, es el conjunto de problemas que pueden ser resueltos en tiempo polinómico por una MT no determinista. Una definición más intuitiva es la siguiente: Es el conjunto de los problemas de decisión para los cuales las instancias donde la respuesta es “sí” tienen demostraciones verificables por cálculos determinísticos en tiempo polinómico.

2. ALGORITMOS DE ORDENAMIENTO

2.1 BUBBLESORT

La ordenación por burbujas es el algoritmo de ordenación más simple que funciona intercambiando repetidamente los elementos adyacentes si están en un orden incorrecto.

Complejidad de tiempo en el peor caso y en el promedio: $O(n^2)$. El peor caso ocurre cuando el array está ordenado de forma inversa.

Complejidad temporal en el mejor de los casos: $O(n)$. El mejor caso ocurre cuando el array ya está ordenado.

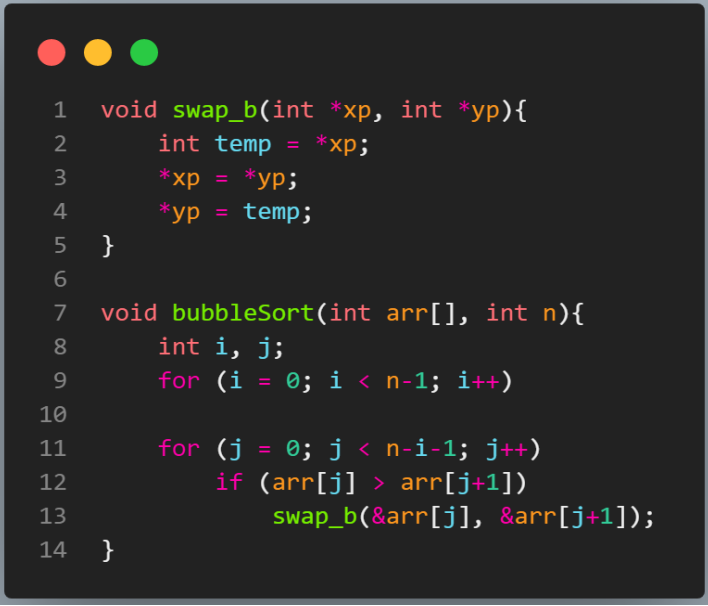
Espacio auxiliar: $O(1)$

Casos límite: La ordenación en burbuja toma un tiempo mínimo (Orden de n) cuando los elementos ya están ordenados.

Ordenación en el lugar: Sí

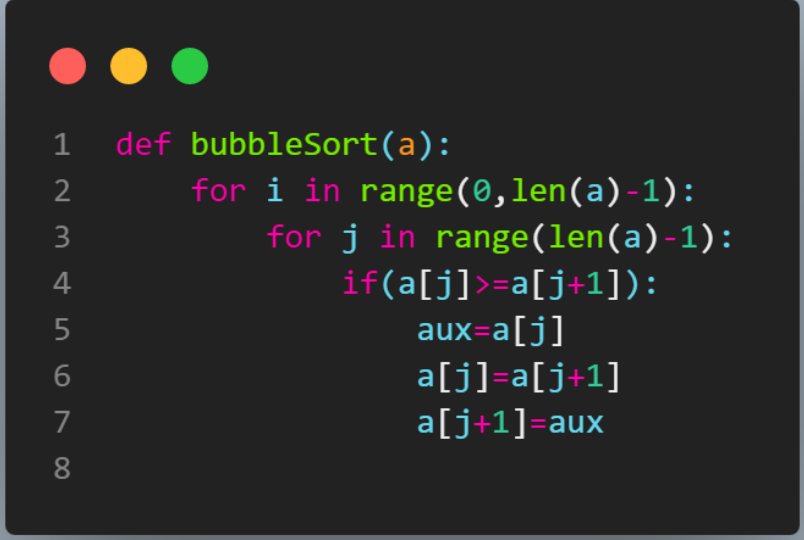
IMPLEMENTACIÓN

C++



```
1 void swap_b(int *xp, int *yp){
2     int temp = *xp;
3     *xp = *yp;
4     *yp = temp;
5 }
6
7 void bubbleSort(int arr[], int n){
8     int i, j;
9     for (i = 0; i < n-1; i++)
10
11     for (j = 0; j < n-i-1; j++)
12         if (arr[j] > arr[j+1])
13             swap_b(&arr[j], &arr[j+1]);
14 }
```

PYTHON

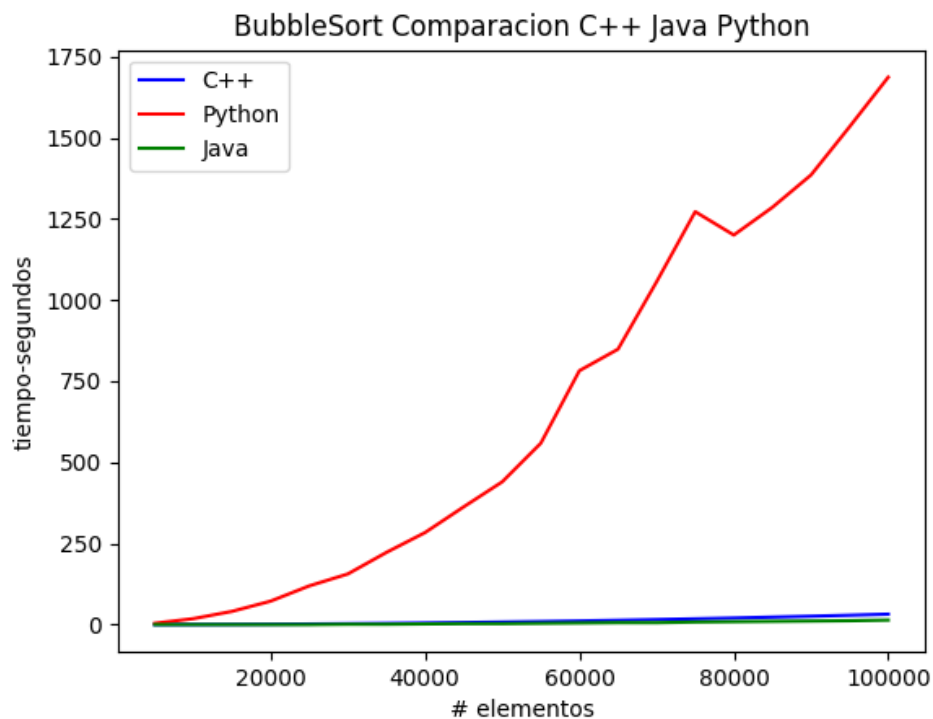


```
1 def bubbleSort(a):
2     for i in range(0, len(a)-1):
3         for j in range(len(a)-1):
4             if(a[j]>=a[j+1]):
5                 aux=a[j]
6                 a[j]=a[j+1]
7                 a[j+1]=aux
8
```

JAVA

```
1 public class BubbleSort
2 {
3     public void bubbleSort(int arr[])
4     {
5         int n = arr.length;
6         for (int i = 0; i < n-1; i++)
7             for (int j = 0; j < n-i-1; j++)
8                 if (arr[j] > arr[j+1])
9                     {
10                        // swap arr[j+1] and arr[j]
11                        int temp = arr[j];
12                        arr[j] = arr[j+1];
13                        arr[j+1] = temp;
14                    }
15     }
16 }
17
```

GRÁFICA EN LOS 3 LENGUAJES



2.2 HEAPSORT

La ordenación de montón es una técnica de ordenación basada en la comparación y en la estructura de datos del montón binario. Es similar a la ordenación por selección, en la que primero encontramos el elemento mínimo y lo colocamos al principio. Repetimos el mismo proceso para el resto de los elementos.

¿Qué es un montón binario?

Definamos primero un Árbol Binario Completo. Un árbol binario completo es un árbol binario en el que cada nivel, excepto posiblemente el último, está completamente lleno, y todos los nodos están lo más a la izquierda posible (Fuente Wikipedia)

Un montón binario es un árbol binario completo en el que los elementos se almacenan en un orden especial, de manera que el valor de un nodo padre es mayor (o menor) que los valores de sus dos nodos hijos. El primero se denomina max heap y el segundo min heap. El montón puede representarse mediante un árbol binario o un array.

¿Por qué una representación basada en un array para un montón binario?

Dado que un montón binario es un árbol binario completo, puede representarse fácilmente como una matriz y la representación basada en una matriz es eficiente en términos de espacio. Si el nodo padre se almacena en el índice I , el hijo izquierdo puede ser calculado por $2 * I + 1$ y el hijo derecho por $2 * I + 2$ (asumiendo que la indexación comienza en 0).

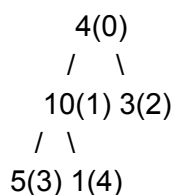
Algoritmo de ordenación del montón para ordenar en orden creciente:

1. Construir un montón máximo a partir de los datos de entrada.
2. En este punto, el elemento más grande se almacena en la raíz del montón. Sustitúyalo por el último elemento del montón y luego reduzca el tamaño del montón en 1. Por último, haga un montón de la raíz del árbol.
3. Repita el paso 2 mientras el tamaño del montón sea mayor que 1.

¿Cómo se construye el montón?

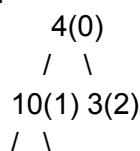
El procedimiento de heapificación sólo puede aplicarse a un nodo si sus nodos hijos están heapificados. Por tanto, la heapificación debe realizarse en orden ascendente.

Datos de entrada: 4, 10, 3, 5, 1



Los números entre paréntesis representan los índices en la matriz representación de los datos.

Aplicando el procedimiento heapify al índice 1:



5(3) 1(4)

Aplicando el procedimiento de heapify al índice 0:

```
    10(0)
   /\
  5(1) 3(2)
 /\
4(3) 1(4)
```

El procedimiento heapify se llama a sí mismo recursivamente para construir el montón de manera descendente.

Complejidad:

Heap : $O(\log n)$

BuildHeap : $O(n)$

Heap Sort : $O(n \log n)$

Espacio Auxiliar: $O(n \log n)$


Estable: No

IMPLEMENTACIÓN:

C++

```
1  void heapify(int arr[], int n, int i)
2  {
3      int largest = i;
4      int l = 2 * i + 1;
5      int r = 2 * i + 2;
6      if (l < n && arr[l] > arr[largest]) largest = l;
7      if (r < n && arr[r] > arr[largest]) largest = r;
8      if (largest != i) {
9          swap(arr[i], arr[largest]);
10         heapify(arr, n, largest);
11     }
12 }
13 void heapSort(int arr[], int n)
14 {
15     for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
16     for (int i = n - 1; i > 0; i--) {
17         swap(arr[0], arr[i]);
18         heapify(arr, i, 0);
19     }
20 }
```


PYTHON

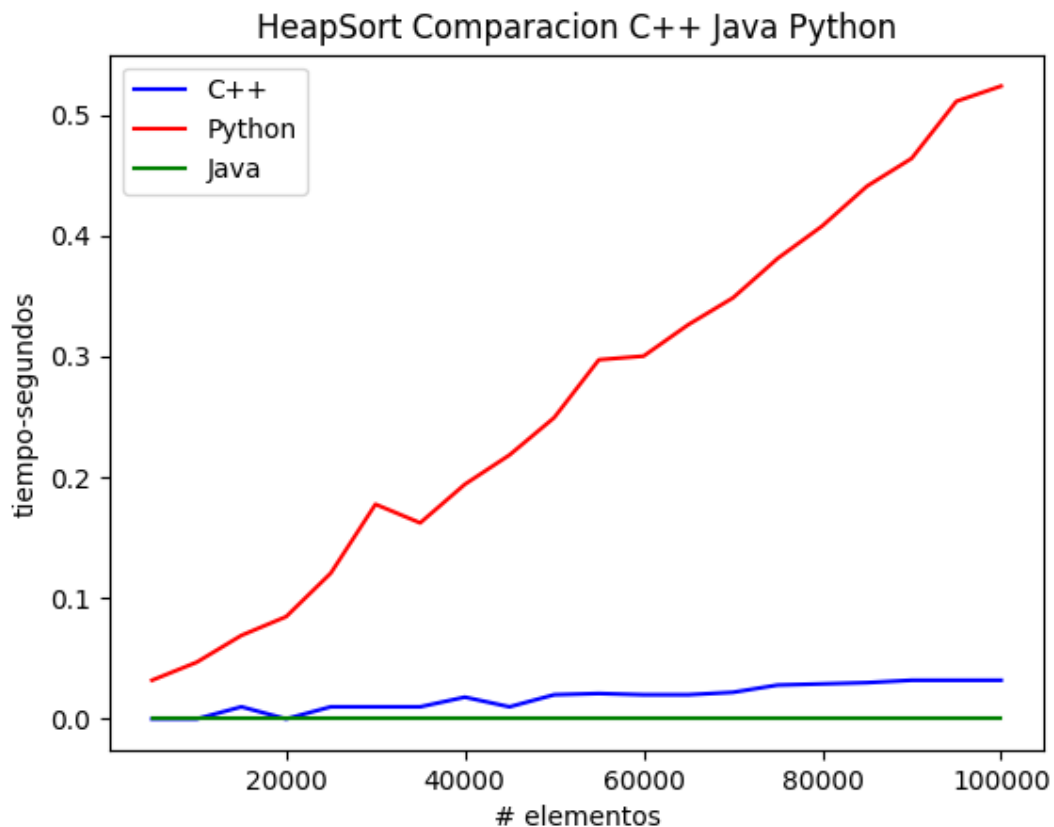


```
1  def heapify(a, n, i):
2      largest = i
3      l = 2 * i + 1
4      r = 2 * i + 2
5      if l < n and a[i] < a[l]:
6          largest = l
7      if r < n and a[largest] < a[r]:
8          largest = r
9      if largest != i:
10         a[i],a[largest] = a[largest],a[i]
11         heapify(a, n, largest)
12 def heapSort(a):
13     n = len(a)
14     for i in range(n, -1, -1):
15         heapify(a, n, i)
16     for i in range(n-1, 0, -1):
17         a[i], a[0] = a[0], a[i]
18         heapify(a, i, 0)
```

JAVA

```
1  public class HeapSort {
2      public void sort(int arr[])
3      {
4          int n = arr.length;
5          for (int i = n / 2 - 1; i >= 0; i--)
6              heapify(arr, n, i);
7          for (int i = n - 1; i > 0; i--) {
8              int temp = arr[0];
9              arr[0] = arr[i];
10             arr[i] = temp;
11             heapify(arr, i, 0);
12         }
13     }
14     void heapify(int arr[], int n, int i)
15     {
16         int largest = i;
17         int l = 2 * i + 1;
18         int r = 2 * i + 2;
19         if (l < n && arr[l] > arr[largest])
20             largest = l;
21         if (r < n && arr[r] > arr[largest])
22             largest = r;
23         if (largest != i) {
24             int swap = arr[i];
25             arr[i] = arr[largest];
26             arr[largest] = swap;
27             heapify(arr, n, largest);
28         }
29     }
30 }
```

GRÁFICA EN LOS 3 LENGUAJES



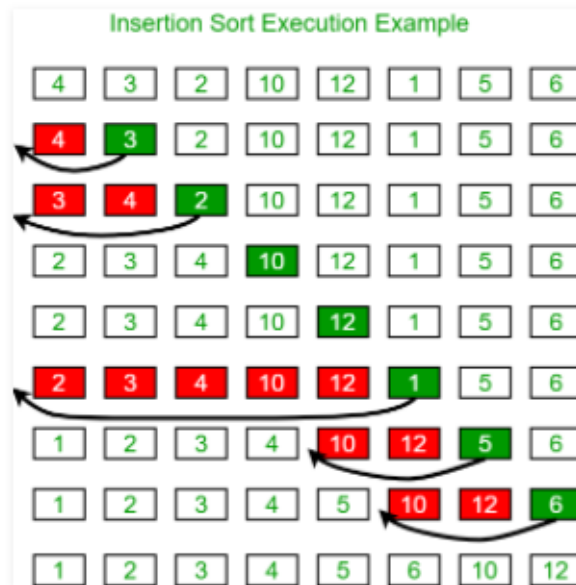
3.3 INSERTIONSORT

La ordenación por inserción es un sencillo algoritmo de ordenación que funciona de forma similar a la forma en que se ordenan las cartas en las manos. La matriz se divide virtualmente en una parte ordenada y otra sin ordenar. Los valores de la parte no ordenada se eligen y se colocan en la posición correcta en la parte ordenada.

Algoritmo

Para ordenar un array de tamaño n en orden ascendente

- 1: Iterar desde $arr[1]$ hasta $arr[n]$ sobre el array.
- 2: Comparar el elemento actual (clave) con su predecesor.
- 3: Si el elemento clave es menor que su predecesor, compárelo con los elementos anteriores. Mueve los elementos mayores una posición hacia arriba para hacer espacio para el elemento intercambiado.



Complejidad temporal: $O(n^2)$

Espacio auxiliar: $O(1)$

Casos límite: La ordenación por inserción tarda el máximo tiempo en ordenar si los elementos están ordenados en orden inverso. Y toma un tiempo mínimo (Orden de n) cuando los elementos ya están ordenados.

Estable: Sí

Usos: La ordenación por inserción se utiliza cuando el número de elementos es pequeño. También puede ser útil cuando el array de entrada está casi ordenado, sólo unos pocos elementos están mal colocados en un array grande completo.


IMPLEMENTACIÓN

C++

```


1  void insertionSort(int arr[], int n){
2      int i, key, j;
3      for (i = 1; i < n; i++){
4          key = arr[i];
5          j = i - 1;
6          while (j >= 0 && arr[j] > key){
7              arr[j + 1] = arr[j];
8              j = j - 1;
9          }
10         arr[j + 1] = key;
11     }
12 }
```

PYTHON



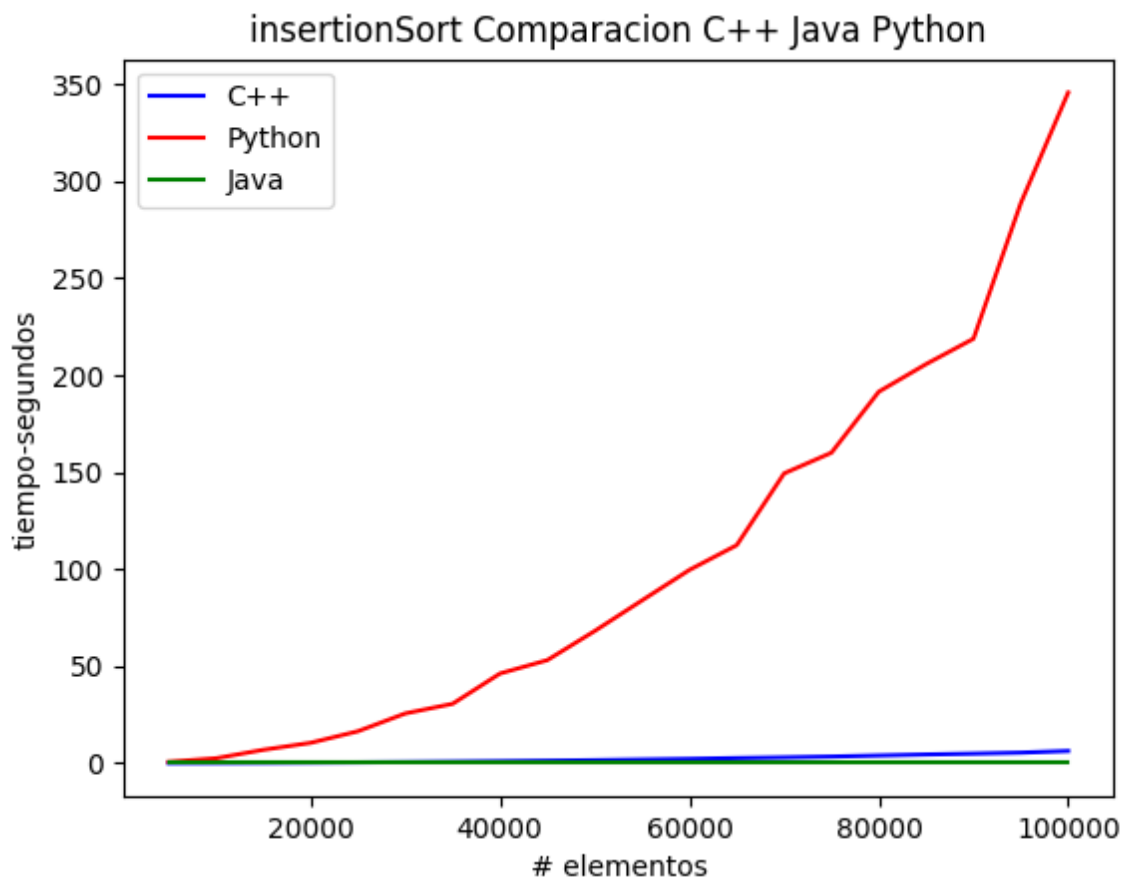
```
1  def insertionSort(a):
2
3      for i in range(1, len(a)):
4          key = a[i]
5          j = i-1
6          while j >= 0 and key < a[j]:
7              a[j + 1] = a[j]
8              j -= 1
9          a[j + 1] = key
```

JAVA



```
1  public class InsertionSort
2  {
3      public void insertionSort(int arr[])
4      {
5          int n = arr.length;
6          for (int i = 1; i < n; ++i) {
7              int key = arr[i];
8              int j = i - 1;
9              while (j >= 0 && arr[j] > key) {
10                 arr[j + 1] = arr[j];
11                 j = j - 1;
12             }
13             arr[j + 1] = key;
14         }
15     }
16 }
```

GRÁFICA EN LOS 3 LENGUAJES



3.4 SELECTIONSORT

El algoritmo de ordenación por selección ordena un array encontrando repetidamente el elemento mínimo (considerando el orden ascendente) de la parte no ordenada y poniéndolo al principio. El algoritmo mantiene dos submatrices en un array dado.

- 1) La submatriz que ya está ordenada.
- 2) La submatriz restante que no está ordenada.

En cada iteración de la ordenación por selección, el elemento mínimo (considerando el orden ascendente) de la submatriz no ordenada se elige y se mueve a la submatriz ordenada.

```
arr[] = 64 25 12 22 11
```

```
// Encuentra el elemento mínimo en arr[0...4]
// y colócalo al principio
11 25 12 22 64
```

```
// Encuentra el elemento mínimo en arr[1...4]
```

```
// y colócalo al principio de arr[1...4]
11 12 25 22 64

// Encuentra el elemento mínimo en arr[2...4]
// y colócalo al principio de arr[2...4]
11 12 22 25 64

// Encuentra el elemento mínimo en arr[3...4]
// y colócalo al principio de arr[3...4]
11 12 22 25 64
```


Complejidad temporal: $O(n^2)$.

Espacio auxiliar: $O(1)$

Lo bueno de la ordenación por selección es que nunca hace más de $O(n)$ intercambios y puede ser útil cuando la escritura en memoria es una operación costosa.


IMPLEMENTACIÓN

C++




```
1 void swap_s(int *xp, int *yp){
2     int temp = *xp;
3     *xp = *yp;
4     *yp = temp;
5 }
6 void selectionSort(int arr[], int n){
7     int i, j, min_idx;
8     for (i = 0; i < n-1; i++){
9         min_idx = i;
10        for (j = i+1; j < n; j++)
11            if (arr[j] < arr[min_idx])
12                min_idx = j;
13        swap_s(&arr[min_idx], &arr[i]);
14    }
15 }
```

PYTHON



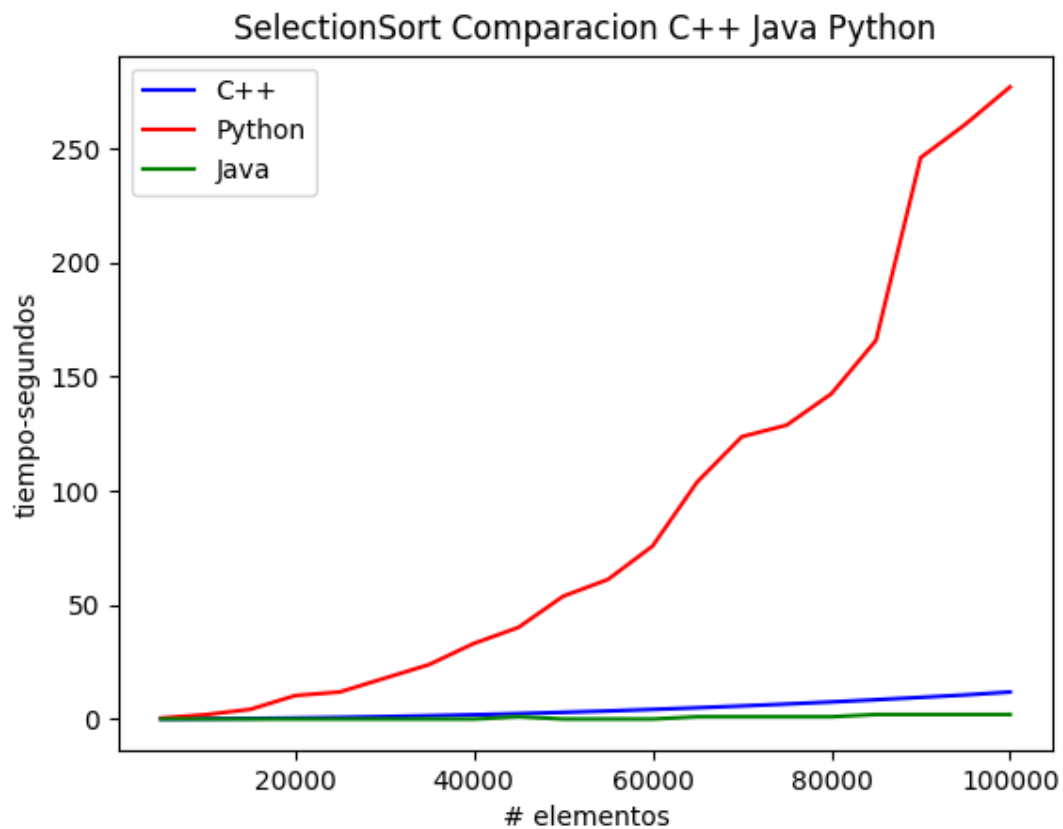
```
1  def selectionSort(a):
2      n=len(a)-1
3      while n>=0:
4          mayor=a[0]
5          pos=0
6          for i in range(1,n+1):
7              if mayor< a[i]:
8                  mayor = a[i]
9                  pos=i
10         a[pos],a[n]=a[n],a[pos]
11         n=n-1
```

JAVA



```
1  public class SelectionSort
2  {
3      public void selectionSort(int arr[])
4      {
5          int n = arr.length;
6          for (int i = 0; i < n-1; i++)
7          {
8              int min_idx = i;
9              for (int j = i+1; j < n; j++)
10                 if (arr[j] < arr[min_idx]) min_idx = j;
11
12                 int temp = arr[min_idx];
13                 arr[min_idx] = arr[i];
14                 arr[i] = temp;
15             }
16         }
17
18     }
```


GRÁFICA EN LOS 3 LENGUAJES



3.5 SHELLSORT

ShellSort es principalmente una variación de la ordenación por inserción. En la ordenación por inserción, movemos los elementos sólo una posición hacia adelante. Cuando un elemento tiene que ser movido muy adelante, muchos movimientos están involucrados. La idea de ShellSort es permitir el intercambio de elementos lejanos. En shellSort, hacemos que el array esté ordenado en h para un valor grande de h . Seguimos reduciendo el valor de h hasta que se convierta en 1. Se dice que un array está ordenado por h si todas las sublistas de cada elemento h están ordenadas.

12 34 54 2 3



Temp

Start with gap = $n/2$ (2 in this case)

One by one select elements to the right of gap and place them at their appropriate position.

12 34 2 3

54

Temp

Elements left of 54 are already smaller, so no change.

One by one select elements to the right of gap and place them at their appropriate position.

12 34 54 2 3

2

Temp

Compare 2 with $arr[3-2] = 34$ and shift it to $arr[gap+1 = 3]$.

12 54 34 3

2

Temp

Compare 2 with $arr[3-2] = 34$ and shift it to $arr[gap+1 = 3]$.

3 12 34 54

2

Temp

Since $3 > 2$

Now gap reduces to $1(n/4)$.

Select all elements starting from $arr[1]$ and compare them with elements within the distance of gap.

2 3 12 34 54

Now gap reduces to 0

Sorting stops and array is sorted.

Complejidad temporal: La complejidad temporal de la implementación anterior de shellsort es $O(n^2)$.

IMPLEMENTACIÓN

C++

```
1  int shellSort(int arr[], int n){
2      for (int gap = n/2; gap > 0; gap /= 2){
3          for (int i = gap; i < n; i += 1){
4              int temp = arr[i];
5
6              int j;
7              for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
8                  arr[j] = arr[j - gap];
9
10             arr[j] = temp;
11         }
12     }
13     return 0;
14 }
```

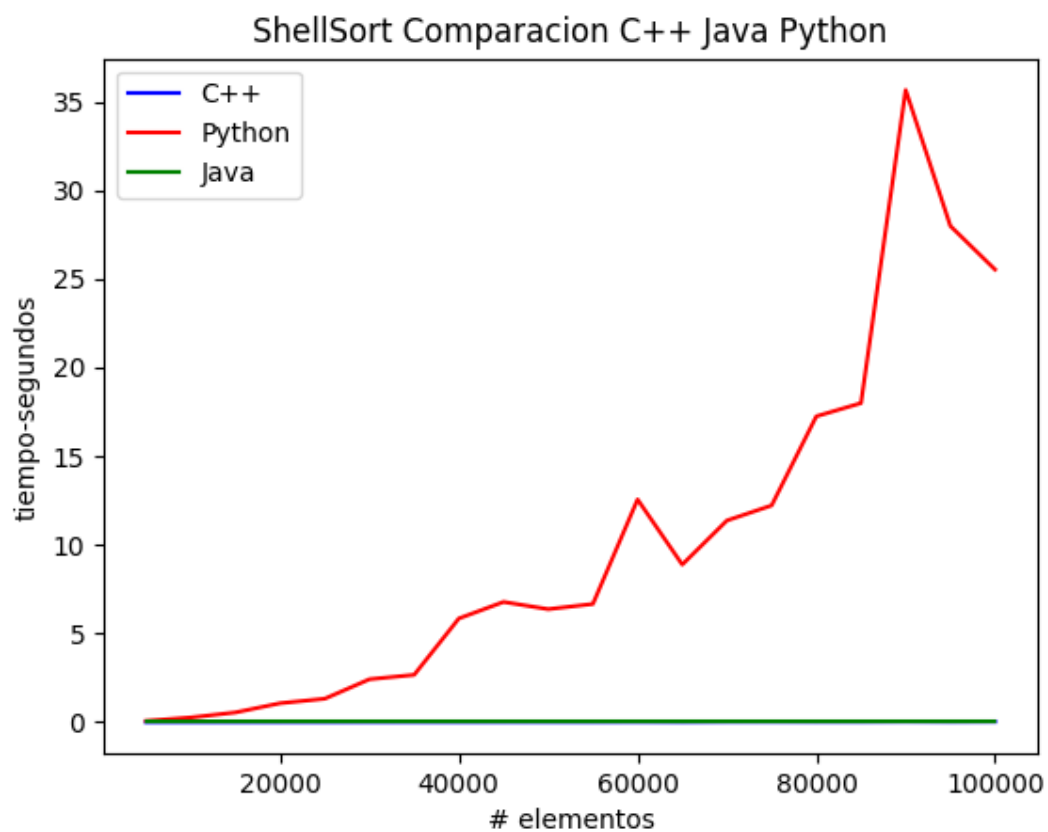
PYTHON

```
1  def shellSort(a):
2      n=len(a)
3      intervalo=n//2
4
5      while intervalo>0:
6          j=intervalo
7          for i in range(0,n):
8              if(j<n):
9                  if a[i]>a[j]:
10                     a[i],a[j]=a[j],a[i]
11                     if(intervalo==1):
12                         key=a[i]
13                         k=i-1
14                         while k>=0 and key<a[k]:
15                             a[k+1]=a[k]
16                             k-=1
17                         a[k+1]=key
18             j+=1
19             intervalo=intervalo//2
```

JAVA

```
1 public class ShellSort
2 {
3     public void shellSort(int arr[])
4     {
5         int n = arr.length;
6         for (int gap = n/2; gap > 0; gap /= 2)
7         {
8             for (int i = gap; i < n; i += 1)
9             {
10                 int temp = arr[i];
11                 int j;
12                 for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
13                     arr[j] = arr[j - gap];
14                 arr[j] = temp;
15             }
16         }
17     }
18 }
```

GRÁFICA EN LOS 3 LENGUAJES



3.6 MERGESORT

Merge Sort es un algoritmo de división y conquista. Divide la matriz de entrada en dos mitades, se llama a sí mismo para las dos mitades, y luego combina las dos mitades ordenadas. La función `merge()` se utiliza para fusionar dos mitades. La función `merge(arr, l, m, r)` es un proceso clave que asume que `arr[l..m]` y `arr[m+1..r]` están ordenados y fusiona las dos submatrices ordenadas en una sola.

```
MergeSort(arr[], l, r)
```

```
Si r > l
```

```
1. Encuentra el punto medio para dividir el array en dos mitades:
```

```
    medio m = l+ (r-l)/2
```

```
2. Llamar a mergeSort para la primera mitad:
```

```
    Llamar a mergeSort(arr, l, m)
```

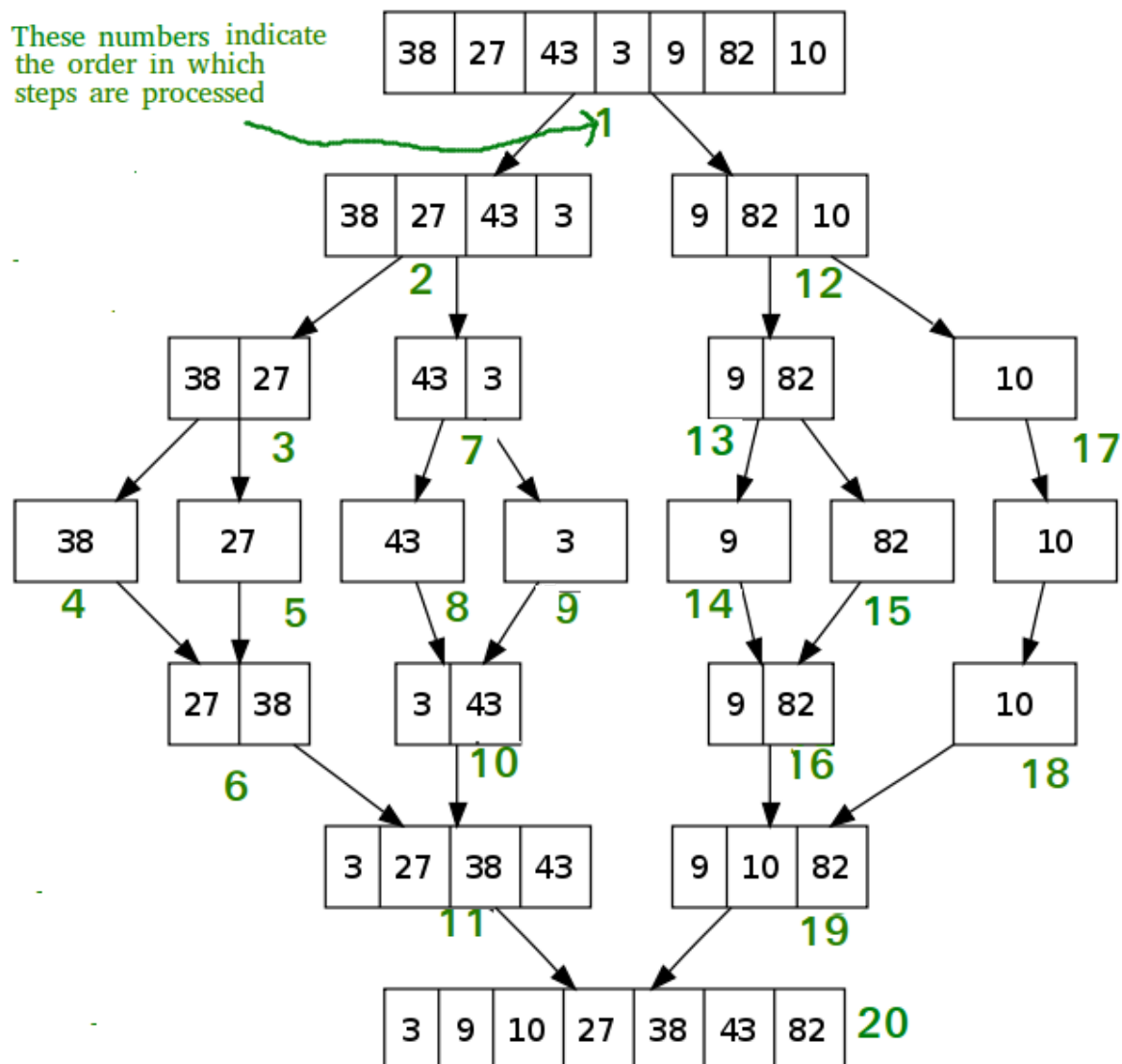
```
3. Llamar a mergeSort para la segunda mitad:
```

```
    Llamar a mergeSort(arr, m+1, r)
```

```
4. Combina las dos mitades ordenadas en los pasos 2 y 3:
```

```
    Llamar a merge(arr, l, m, r)
```

These numbers indicate the order in which steps are processed



Complejidad temporal: $O(n \log n)$.

Espacio auxiliar: $O(n)$

Paradigma algorítmico: Divide y vencerás


Estable: Sí

IMPLEMENTACIÓN

C++

```
1 void merge(int array[], int const left, int const mid, int const right)
2 {
3     auto const subArrayOne = mid - left + 1;
4     auto const subArrayTwo = right - mid;
5     auto *leftArray = new int[subArrayOne],
6         *rightArray = new int[subArrayTwo];
7     for (auto i = 0; i < subArrayOne; i++) leftArray[i] = array[left + i];
8     for (auto j = 0; j < subArrayTwo; j++)
9         rightArray[j] = array[mid + 1 + j];
10    auto indexOfSubArrayOne = 0,
11        indexOfSubArrayTwo = 0;
12    int indexOfMergedArray = left;
13    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo) {
14        if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo]) {
15            array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
16            indexOfSubArrayOne++;
17        }
18        else {
19            array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
20            indexOfSubArrayTwo++;
21        }
22        indexOfMergedArray++;
23    }
24    while (indexOfSubArrayOne < subArrayOne) {
25        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
26        indexOfSubArrayOne++;
27        indexOfMergedArray++;
28    }
29    while (indexOfSubArrayTwo < subArrayTwo) {
30        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
31        indexOfSubArrayTwo++;
32        indexOfMergedArray++;
33    }
34 }
35 void mergeSort(int array[], int const begin, int const end)
36 {
37     if (begin >= end) return;
38     auto mid = begin + (end - begin) / 2;
39     mergeSort(array, begin, mid);
40     mergeSort(array, mid + 1, end);
41     merge(array, begin, mid, end);
42 }
43
```

PYTHON

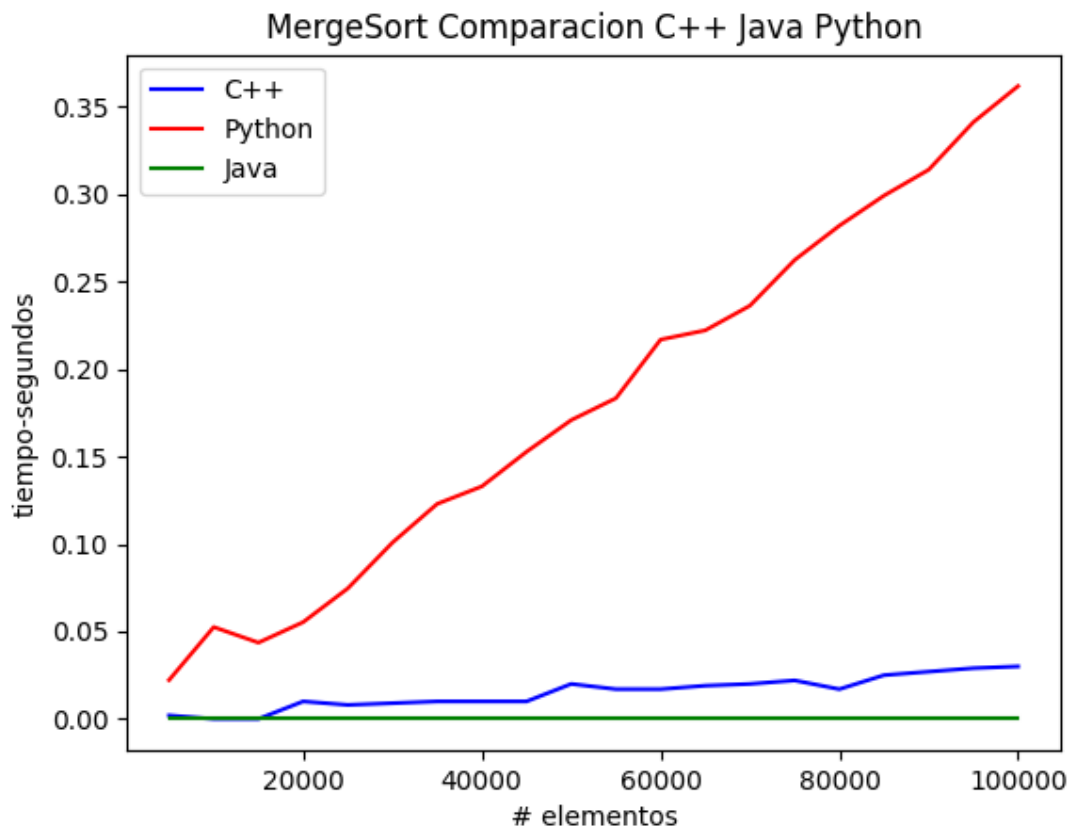


```
1  def mergeSort(arr):
2      if len(arr) > 1:
3          mid = len(arr)//2
4          L = arr[:mid]
5          R = arr[mid:]
6          mergeSort(L)
7          mergeSort(R)
8          i = j = k = 0
9          while i < len(L) and j < len(R):
10             if L[i] < R[j]:
11                 arr[k] = L[i]
12                 i += 1
13             else:
14                 arr[k] = R[j]
15                 j += 1
16             k += 1
17         while i < len(L):
18             arr[k] = L[i]
19             i += 1
20             k += 1
21         while j < len(R):
22             arr[k] = R[j]
23             j += 1
24             k += 1
```

JAVA

```
1  public class MergeSort
2  {
3      void merge(int arr[], int l, int m, int r)
4      {
5          int n1 = m - l + 1;
6          int n2 = r - m;
7          int L[] = new int[n1];
8          int R[] = new int[n2];
9          for (int i = 0; i < n1; ++i)
10             L[i] = arr[l + i];
11          for (int j = 0; j < n2; ++j)
12             R[j] = arr[m + 1 + j];
13          int i = 0, j = 0;
14          int k = l;
15          while (i < n1 && j < n2) {
16             if (L[i] <= R[j]) { arr[k] = L[i]; i++; }
17             else { arr[k] = R[j]; j++; }
18             k++;
19         }
20         while (i < n1) {
21             arr[k] = L[i];
22             i++; k++;
23         }
24         while (j < n2) {
25             arr[k] = R[j];
26             j++; k++;
27         }
28     }
29     public void mergeSort(int arr[], int l, int r)
30     {
31         if (l < r) {
32             int m = l + (r-l)/2;
33             mergeSort(arr, l, m);
34             mergeSort(arr, m + 1, r);
35             merge(arr, l, m, r);
36         }
37     }
38 }
```


GRÁFICA EN LOS 3 LENGUAJES



3.7 QUICKSORT

QuickSort es un algoritmo de división y conquista. Escoge un elemento como pivote y divide la matriz dada alrededor del pivote escogido. Hay muchas versiones diferentes de quickSort que eligen el pivote de diferentes maneras.

Siempre escoge el primer elemento como pivote.

Elegir siempre el último elemento como pivote (implementado a continuación)

Elegir un elemento al azar como pivote.

Elegir la mediana como pivote.

El proceso clave en quickSort es partition(). El objetivo de las particiones es, dado un array y un elemento x del array como pivote, poner x en su posición correcta en el array ordenado y poner todos los elementos más pequeños (menores que x) antes de x, y poner todos los elementos mayores (mayores que x) después de x. Todo esto debería hacerse en tiempo lineal.

```
/* low --> índice inicial, high --> índice final */  
quickSort(arr[], low, high)
```

```

{
    si (bajo < alto)
    {
        /* pi es el índice de partición, arr[pi] está ahora
           en el lugar correcto */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Antes de pi
        quickSort(arr, pi + 1, high); // Después de pi
    }
}

```

PARTICIÓN:

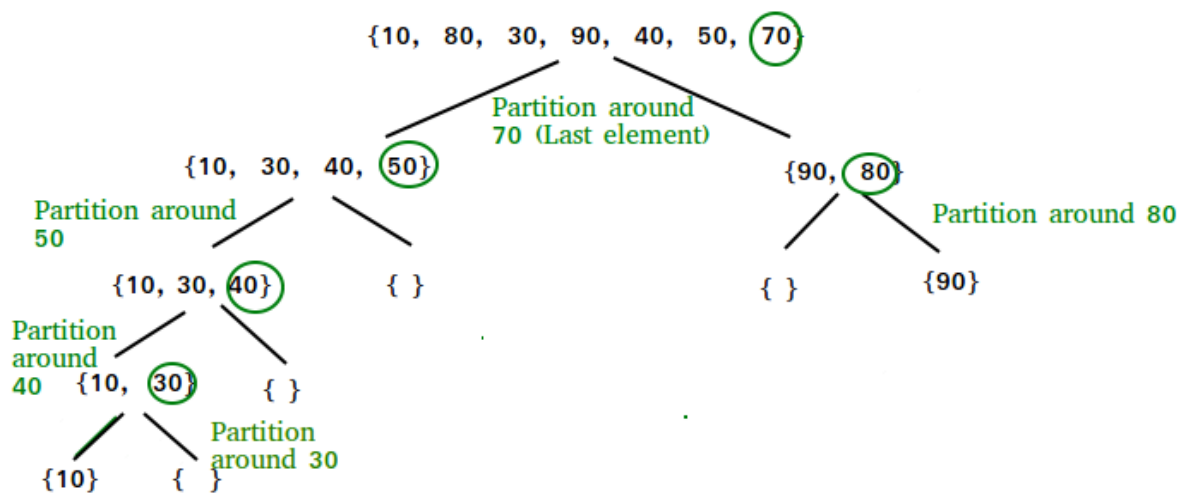
```

/* Esta función toma el último elemento como pivote, coloca
   el elemento pivote en su posición correcta en la matriz
   ordenada, y coloca todos los elementos menores (más pequeños
   que el pivote)
   a la izquierda del pivote y todos los elementos mayores a la
   derecha
   del pivote */
partición (arr[], bajo, alto)
{
    // pivote (elemento que se coloca a la derecha)
    pivote = arr[high];

    i = (low - 1) // Índice del elemento más pequeño e indica la
                  // posición derecha del pivote encontrado hasta
ahora

    for (j = bajo; j <= alto 1; j++)
    {
        // Si el elemento actual es más pequeño que el pivote
        if (arr[j] < pivote)
        {
            i++; // incrementa el índice del elemento más pequeño
            intercambiar arr[i] y arr[j]
        }
    }
    swap arr[i + 1] y arr[high])
    return (i + 1)
}

```



Complejidad: $O(n)$

El peor caso: El peor caso ocurre cuando el proceso de partición siempre elige el elemento más grande o más pequeño como pivote. Si consideramos la estrategia de partición anterior, en la que el último elemento se elige siempre como pivote, el peor caso se produciría cuando la matriz ya está ordenada en orden creciente o decreciente.


Mejor caso: El mejor caso ocurre cuando el proceso de partición siempre elige el elemento del medio como pivote. La siguiente es la recurrencia para el mejor caso.

Caso medio:

Para hacer el análisis del caso promedio, necesitamos considerar todas las permutaciones posibles del arreglo y calcular el tiempo que toma cada permutación, lo cual no parece fácil.

IMPLEMENTACIÓN

C++



```
1 void swap_q(int* a, int* b){
2     int t = *a;
3     *a = *b;
4     *b = t;
5 }
6 int partition (int arr[], int low, int high){
7     int pivot = arr[high];
8     int i = (low - 1);
9
10    for (int j = low; j <= high - 1; j++)
11    {
12        if (arr[j] < pivot)
13        {
14            i++;
15            swap_q(&arr[i], &arr[j]);
16        }
17    }
18    swap_q(&arr[i + 1], &arr[high]);
19    return (i + 1);
20 }
21 void quickSort(int arr[], int low, int high){
22     if (low < high){
23         int pi = partition(arr, low, high);
24         quickSort(arr, low, pi - 1);
25         quickSort(arr, pi + 1, high);
26     }
27 }
```

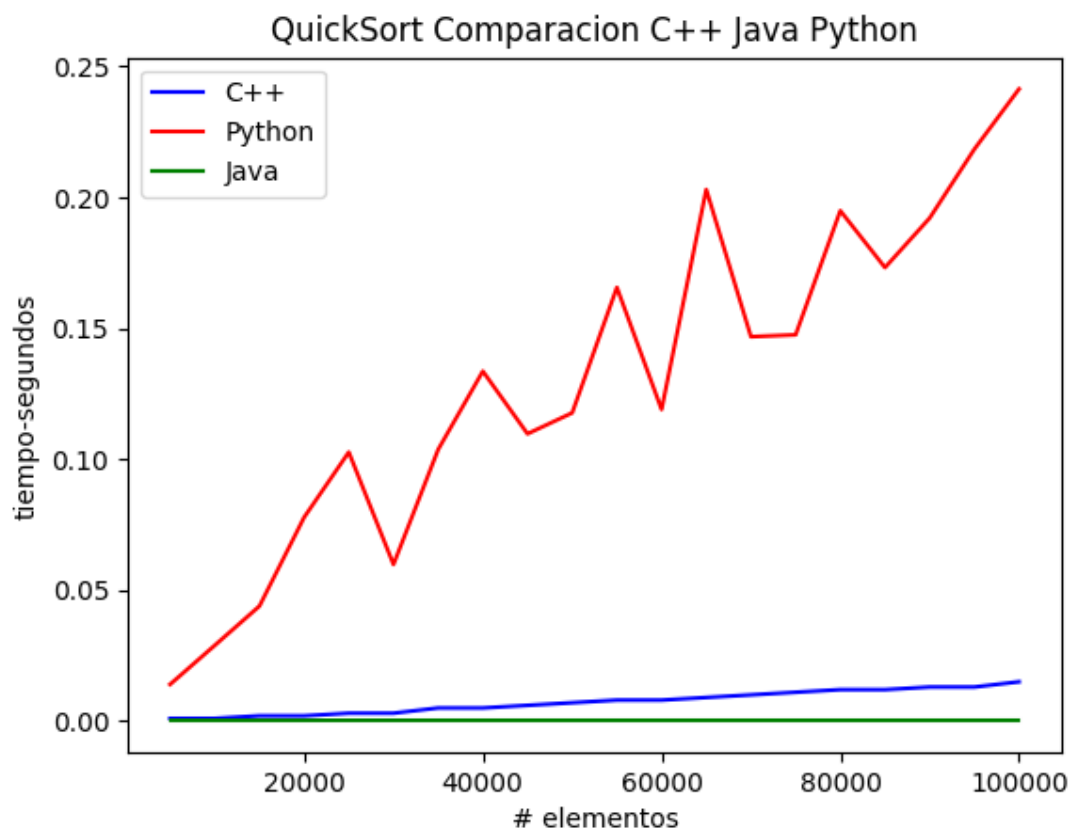
PYTHON

```
1  def partition(a,low,high):
2      i = ( low-1 )
3      pivot = a[high]
4      for j in range(low , high):
5          if a[j] < pivot:
6              i = i+1
7              a[i],a[j] = a[j],a[i]
8      a[i+1],a[high] = a[high],a[i+1]
9      return ( i+1 )
10 def quickSort(a,low,high):
11     if low < high:
12         pi = partition(a,low,high)
13         quickSort(a, low, pi-1)
14         quickSort(a, pi+1, high)
```

JAVA

```
1  public class QuickSort{
2
3      void swap(int[] arr, int i, int j)
4      {
5          int temp = arr[i];
6          arr[i] = arr[j];
7          arr[j] = temp;
8      }
9      public int partition(int[] arr, int low, int high)
10     {
11         int pivot = arr[high];
12         int i = (low - 1);
13         for(int j = low; j <= high - 1; j++)
14         {
15             if (arr[j] < pivot)
16             {
17                 i++; swap(arr, i, j);
18             }
19         }
20         swap(arr, i + 1, high);
21         return (i + 1);
22     }
23
24     public void quickSort(int[] arr, int low, int high)
25     {
26         if (low < high)
27         {
28             int pi = partition(arr, low, high);
29             quickSort(arr, low, pi - 1);
30             quickSort(arr, pi + 1, high);
31         }
32     }
33 }
```

GRÁFICA EN LOS 3 LENGUAJES



3. CONCLUSIONES

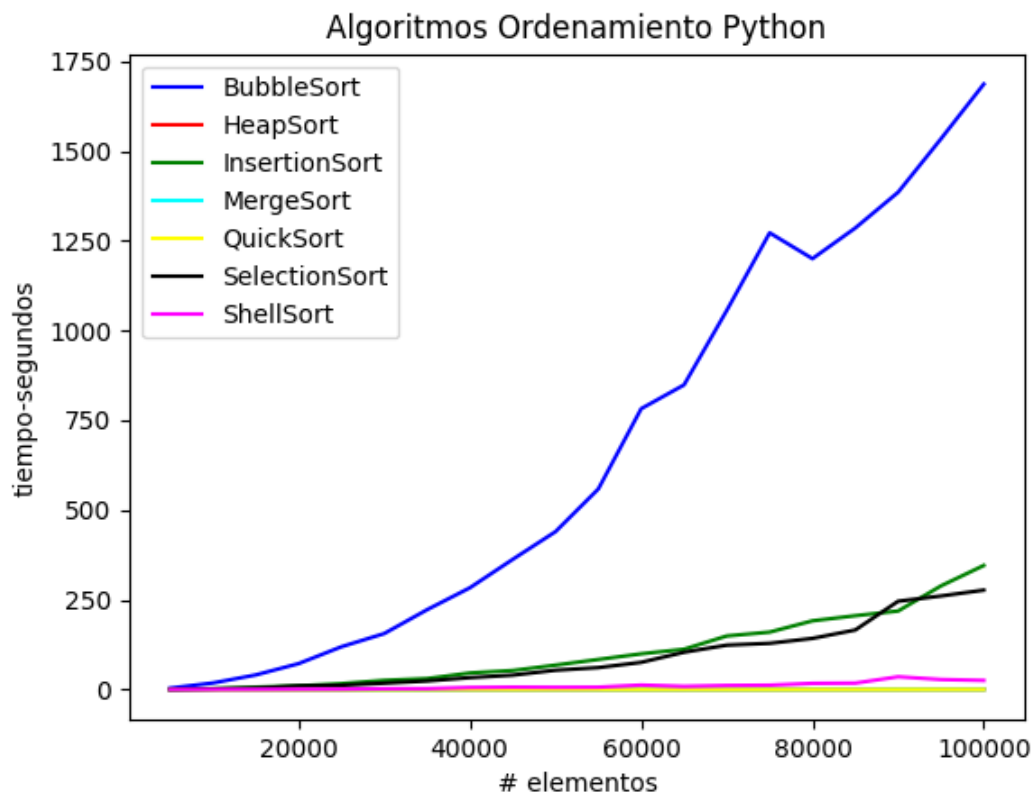
En esta sección observamos el rendimiento de los algoritmos en 3 lenguajes de programación:

- C++
- Python
- Java

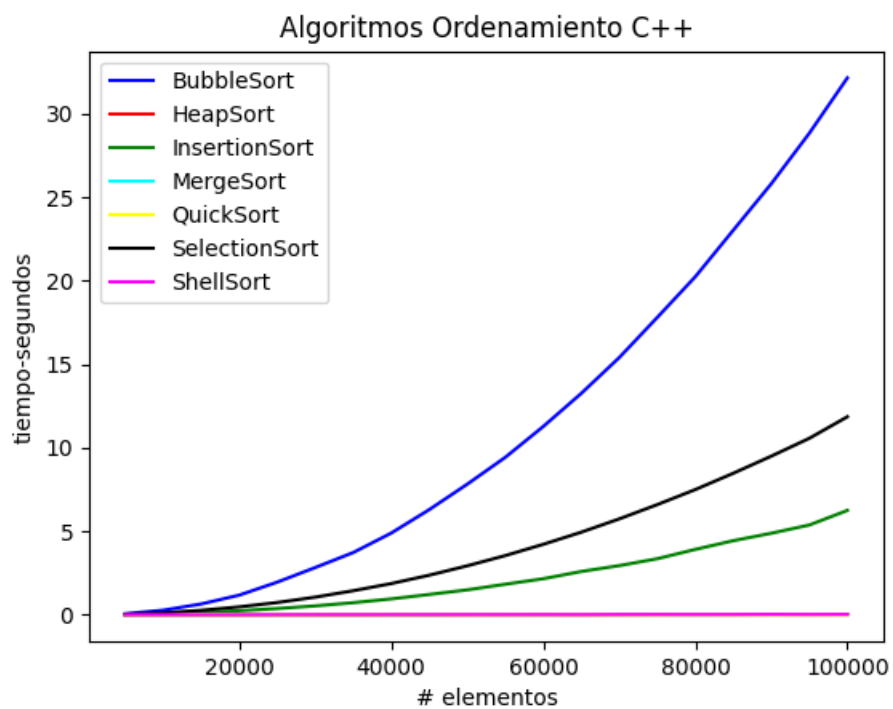
En cuanto a la memoria Python tiene mayor consumo que en C++ y JAVA, esto se debe a que C++ y JAVA convierte el código fuente a código máquina y lo ejecuta, mientras que Python convierte el código línea por línea en código de bytes y luego lo convierte en código máquina, después de ello ejecuta el programa línea por línea.

ALGORITMO	COMPLEJIDAD		
	MEJOR	PROMEDIO	PEOR
BUBBLE SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$
HEAP SORT	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
INSERTION SORT	$O(n)$	$O(n^2)$	$O(n^2)$
SELECTION SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$
SHELL SORT	$O(n \log(n))$		$O(n^2)$
MERGE SORT	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
QUICK SORT	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

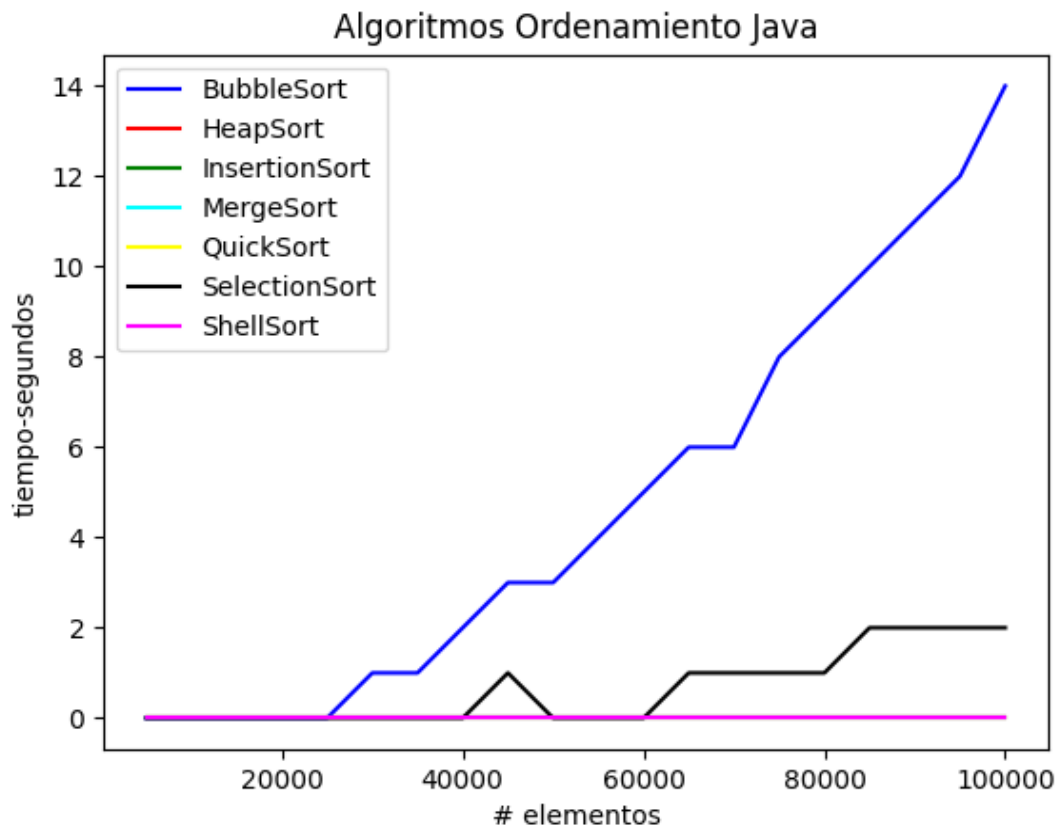
3.1 Rendimiento en Python



3.2 Rendimiento en C++



3.3 Rendimiento en Java



- Se puede ver que en los 3 lenguajes usados para este proyecto el algoritmo con mayor costo computacional es el BubbleSort que es seguido por el InsertionSort e selectionSort; luego le siguen los algoritmos de divide y vencerás: MergeSort y QuickSort
- Finalmente concluimos que el algoritmo de menor costo computacional es un tipo No Comparativo: para el caso C++ sería el QuickSort

References

GeeksforGeeks | A computer science portal for geeks. (2021) from. (n.d.).

<https://www.geeksforgeeks.org/>

Moyano, N. (2021). ¿Qué es la Complejidad Computacional? (n.d.).

<https://medium.com/@nelramoyano/qu%C3%A9-es-la-complejidad-computacional-3a556e557973>

Teoría de la complejidad computacional - Wikipedia, la enciclopedia libre. (2021). (n.d.).

https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_complejidad_computacional