

UNIVERSIDAD NACIONAL DE MOQUEGUA
FACULTAD DE INGENIERÍA Y ARQUITECTURA
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMÁTICA



**Evaluación de Algoritmos de Ordenamiento: Un
Enfoque Multilenguaje**

Informe laboratorio N° 1

Estudiantes:

Gersael Mathias Rojas
Quijano

Profesores:

Honorio Apaza Alanoca

22 de octubre de 2024

Índice

1. Introducción	3
1.1. Motivación y Contexto	3
1.2. Objetivo general	3
1.3. Objetivos específicos	3
1.4. Justificación	3
2. Marco teórico	4
2.1. Antecedentes	4
2.2. Marco conceptual	4
2.2.1. Bubble Sort	5
2.2.2. Counting Sort	5
2.2.3. Heap Sort	7
2.2.4. Insertion Sort	8
2.2.5. Merge Sort	8
2.2.6. Quick Sort	10
2.2.7. Selection Sort	11
3. Metodología	12
3.1. Comparación de Algoritmos de Ordenamiento en Tres Lenguajes de Programación	12
3.1.1. Implementación y Optimización de Algoritmos de Ordenamiento	12
3.1.2. Pruebas de Rendimiento: Medición de Tiempo de Ejecución y Uso de Memoria	12
3.1.3. Presentación y Análisis de Resultados: Comparación Visual y Estadística de Algoritmos	12
4. Propuesta	13
4.1. Implementación y Optimización de Algoritmos de Ordenamiento	13
4.1.1. Código en C++.	13
4.1.2. Código en Java.	18
4.1.3. Código en Python.	23
4.2. Pruebas de Rendimiento: Medición de Tiempo de Ejecución y Uso de Memoria	27
4.2.1. Definición de Escenarios de Prueba	27
4.2.2. Medición del Tiempo de Ejecución	27
4.3. Resultados	28
4.3.1. Resultados en Java.	28
4.3.2. Resultados en C++.	31
4.3.3. Resultados en Python.	34
4.3.4. Resultados Bubble Sort	37
4.3.5. Resultados Counting Sort	38

4.3.6. Resultados Heap Sort	39
4.3.7. Resultados Insertion Sort	40
4.3.8. Resultados Merge Sort	41
4.3.9. Resultados Quick Sort	42
4.3.10. Resultados Selection Sort	43
4.3.11. Graficos:	44
5. Conclusiones	49
6. Recomendaciones	50
Referencias	51

1. Introducción

1.1. Motivación y Contexto

En la era digital actual, el manejo eficiente de datos se ha convertido en una prioridad para las empresas y organizaciones. Los algoritmos de ordenamiento son fundamentales en la informática, ya que permiten organizar datos de manera que se puedan procesar y acceder de forma eficiente. Sin embargo, no todos los algoritmos son iguales; su rendimiento puede variar considerablemente dependiendo de su implementación y del lenguaje de programación utilizado. Por lo tanto, es crucial realizar una comparación exhaustiva de diferentes algoritmos de ordenamiento para entender mejor su eficacia en distintos contextos.

1.2. Objetivo general

El objetivo general de este estudio es evaluar la eficiencia de varios algoritmos de ordenamiento —Bubble Sort, Counting Sort, Heap Sort, Insertion Sort, Merge Sort, Quick Sort y Selection Sort— implementados en tres lenguajes de programación: Java, Python y C++. Esta evaluación se centrará en aspectos como el tiempo de ejecución y el uso de memoria de cada algoritmo.

1.3. Objetivos específicos

- Comparar el tiempo de ejecución de cada algoritmo en Java, Python y C++.
- Analizar el uso de memoria en cada implementación.
- Identificar qué algoritmos son más eficientes en cada lenguaje y en qué condiciones.

1.4. Justificación

La elección de los algoritmos y lenguajes para este estudio se basa en su popularidad y relevancia en el ámbito de la informática. Cada uno de los algoritmos seleccionados presenta características únicas que los hacen adecuados para diferentes tipos de datos y aplicaciones. Además, la comparación en lenguajes de programación ampliamente utilizados como Java, Python y C++ permitirá obtener una visión más completa de cómo el contexto de implementación afecta el rendimiento de los algoritmos.

2. Marco teórico

2.1. Antecedentes

Diversos estudios han abordado la comparación de algoritmos de ordenamiento en el contexto de lenguajes de programación. Sin embargo, la mayoría de estos estudios se centran en un número limitado de algoritmos o en un solo lenguaje. Este trabajo busca llenar ese vacío al proporcionar una comparación más amplia y detallada que incluya varios algoritmos y lenguajes.

2.2. Marco conceptual

Los algoritmos de ordenamiento son métodos utilizados para reorganizar los elementos de una lista o array en un orden específico. La eficiencia de un algoritmo de ordenamiento se mide comúnmente en términos de complejidad temporal y espacial. La complejidad temporal se refiere al tiempo que tarda el algoritmo en ejecutarse, mientras que la complejidad espacial se refiere a la cantidad de memoria que utiliza.

2.2.1. Bubble Sort

Bubble Sort es un algoritmo de ordenamiento sencillo que funciona comparando pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que no se requieren más intercambios, lo que indica que la lista está ordenada. Aunque Bubble Sort es fácil de entender e implementar, su rendimiento no es óptimo para listas grandes, ya que su complejidad temporal en el peor de los casos es $O(n^2)$. Sin embargo, es útil para listas pequeñas o para fines educativos.

Algorithm 1 Bubble Sort

```
1: procedure BUBBLESORT(array, size)
2:   cambio  $\leftarrow$  false
3:   for  $i = 0$  to size - 1 do
4:     cambio  $\leftarrow$  false
5:     for  $j = 0$  to size - 1 -  $i$  do
6:       if array[ $j$ ] > array[ $j + 1$ ] then
7:         aux  $\leftarrow$  array[ $j$ ]
8:         array[ $j$ ]  $\leftarrow$  array[ $j + 1$ ]
9:         array[ $j + 1$ ]  $\leftarrow$  aux
10:        cambio  $\leftarrow$  true
11:      end if
12:    end for
13:    if not cambio then
14:      break
15:    end if
16:  end for
17: end procedure
```

2.2.2. Counting Sort

Counting Sort es un algoritmo no comparativo que ordena contando la frecuencia de los elementos dentro de un rango específico. Utiliza un arreglo auxiliar para almacenar el conteo de cada valor y luego coloca los elementos en su posición correcta basándose en esos conteos. Es eficiente para conjuntos de datos con un rango acotado, con una complejidad de $O(n + k)$, donde n es el número de elementos y k es el valor máximo. Sin embargo, puede requerir más memoria, lo que lo hace menos adecuado para valores dispersos o rangos amplios.

Algorithm 2 Counting Sort

Input: array, size

Output: sorted array

Step 1: Encontrar el valor máximo y mínimo en el array

$\max \leftarrow \text{array}[0]$, $\min \leftarrow \text{array}[0]$

for $i = 0$ to $\text{size}-1$ **do**

if $\text{array}[i] \geq \max$ **then**

$\max \leftarrow \text{array}[i]$

end if

if $\text{array}[i] \leq \min$ **then**

$\min \leftarrow \text{array}[i]$

end if

end for

Step 2: Inicializar el rango y el array de conteo

$\text{range} \leftarrow \max - \min + 1$

$\text{conteo}[\text{range}] \leftarrow 0$

$\text{salida}[\text{size}]$

Step 3: Contar las ocurrencias de cada elemento

for $i = 0$ to $\text{size}-1$ **do**

$\text{conteo}[\text{array}[i] - \min] \leftarrow \text{conteo}[\text{array}[i] - \min] + 1$

end for

Step 4: Calcular las posiciones acumuladas

for $i = 1$ to $\text{range}-1$ **do**

$\text{conteo}[i] \leftarrow \text{conteo}[i] + \text{conteo}[i-1]$

end for

Step 5: Colocar los elementos en su lugar correspondiente

for $i = \text{size}-1$ to 0 **do**

$\text{salida}[\text{conteo}[\text{array}[i] - \min] - 1] \leftarrow \text{array}[i]$

$\text{conteo}[\text{array}[i] - \min] \leftarrow \text{conteo}[\text{array}[i] - \min] - 1$

end for

Step 6: Copiar los elementos ordenados al array original

for $i = 0$ to $\text{size}-1$ **do**

$\text{array}[i] \leftarrow \text{salida}[i]$

end for

2.2.3. Heap Sort

Heap Sort es un algoritmo de ordenamiento basado en un enfoque de estructura de datos de montículos (heaps). Primero convierte el arreglo en un montículo máximo, donde el mayor valor está en la raíz. Luego, intercambia el primer elemento (máximo) con el último y reduce el tamaño del montículo, repitiendo el proceso hasta que el arreglo esté ordenado. Su complejidad es $O(n \log n)$, lo que lo hace eficiente incluso para grandes conjuntos de datos. Aunque no es un algoritmo estable, es eficiente en términos de espacio, ya que no requiere arreglos auxiliares adicionales.

Algorithm 3 Heap Sort

Input: array, size

Output: sorted array

Step 1: Construir el heap máximo

for $i = n/2 - 1$ **to** 0 **do**

heapify(array, n, i)

end for

Step 2: Extraer elementos uno por uno del heap

for $i = n - 1$ **to** 1 **do**

 Intercambiar array[0] con array[i]

heapify(array, i, 0)

end for

function HEAPIFY(array, n, i)

 indice_mayor $\leftarrow i$

 hijo_izquierdo $\leftarrow 2 \times i + 1$

 hijo_derecho $\leftarrow 2 \times i + 2$

if hijo_izquierdo $\leq n$ **and** array[hijo_izquierdo] \geq array[indice_mayor] **then**

 indice_mayor \leftarrow hijo_izquierdo

end if

if hijo_derecho $\leq n$ **and** array[hijo_derecho] \geq array[indice_mayor] **then**

 indice_mayor \leftarrow hijo_derecho

end if

if indice_mayor $\neq i$ **then**

 Intercambiar array[i] con array[indice_mayor]

heapify(array, n, indice_mayor)

end if

end function

2.2.4. Insertion Sort

Insertion Sort es un algoritmo de ordenamiento que construye el arreglo ordenado de manera incremental, insertando cada elemento en su posición correcta dentro de una parte previamente ordenada. Comienza desde el segundo elemento, comparándolo con los elementos anteriores y desplazándolos hacia la derecha hasta encontrar su ubicación adecuada. Este enfoque tiene una complejidad de $O(n^2)$ en el peor de los casos, lo que lo hace menos eficiente para listas grandes, pero es eficiente para listas pequeñas o casi ordenadas. Además, es un algoritmo estable y no requiere memoria adicional, lo que lo hace fácil de implementar.

Algorithm 4 Insertion Sort

Input: array, size
Output: sorted array
for $i = 0$ to size - 1 **do**
 $pos \leftarrow i$
 $aux \leftarrow array[i]$
 while $pos \neq 0$ **and** $array[pos - 1] > aux$ **do**
 $array[pos] \leftarrow array[pos - 1]$
 $pos \leftarrow pos - 1$
 end while
 $array[pos] \leftarrow aux$
end for

2.2.5. Merge Sort

Merge Sort es un algoritmo de ordenamiento basado en la técnica de "divide y vencerás". Divide el arreglo en dos mitades, las ordena recursivamente y luego combina (merge) las mitades ordenadas para formar un arreglo ordenado completo. Su complejidad es $O(n \log n)$ en todos los casos, lo que lo hace eficiente para grandes conjuntos de datos. Además, Merge Sort es un algoritmo estable, lo que significa que preserva el orden relativo de elementos iguales. Sin embargo, requiere memoria adicional para el arreglo auxiliar utilizado en el proceso de combinación, lo que puede ser una desventaja en términos de uso de espacio.

Algorithm 5 Merge Sort

```
Input: array, size
Output: sorted array
Step 1: Dividir el arreglo en dos subarreglos
if size < 2 then
    return
end if
mid  $\leftarrow$  size / 2
left  $\leftarrow$  nuevo arreglo de tamaño mid
right  $\leftarrow$  nuevo arreglo de tamaño (size - mid)
for i = 0 to mid - 1 do
    left[i]  $\leftarrow$  array[i]
end for
for i = mid to size - 1 do
    right[i - mid]  $\leftarrow$  array[i]
end for
merge_sort(left, mid)
merge_sort(right, size - mid)
merge(array, left, mid, right, size - mid)

function MERGE(array, left, leftSize, right, rightSize)
    i, j, k  $\leftarrow$  0
    while i < leftSize and j < rightSize do
        if left[i]  $\leq$  right[j] then
            array[k]  $\leftarrow$  left[i]
            i  $\leftarrow$  i + 1
        else
            array[k]  $\leftarrow$  right[j]
            j  $\leftarrow$  j + 1
        end if
        k  $\leftarrow$  k + 1
    end while
    while i < leftSize do
        array[k]  $\leftarrow$  left[i]
        i  $\leftarrow$  i + 1
        k  $\leftarrow$  k + 1
    end while
    while j < rightSize do
        array[k]  $\leftarrow$  right[j]
        j  $\leftarrow$  j + 1
        k  $\leftarrow$  k + 1
    end while
end function
```

2.2.6. Quick Sort

Quick Sort es un algoritmo de ordenamiento basado en "divide y vencerás". Selecciona un pivote y reorganiza el arreglo en torno a él, colocando los menores a la izquierda y los mayores a la derecha. Luego, aplica recursivamente el mismo proceso a las subparticiones. Su complejidad promedio es $O(n \log n)$, pero en el peor de los casos puede ser $O(n^2)$. A diferencia de Merge Sort, es in-place, lo que significa que no requiere memoria adicional significativa, aunque no es estable, ya que no conserva el orden relativo de los elementos iguales.

Algorithm 6 Quick Sort

Input: array, low, high

Output: sorted array

if low < high **then**

 partitionIndex \leftarrow **partition**(array, low, high)

quickSort(array, low, partitionIndex - 1)

quickSort(array, partitionIndex + 1, high)

end if

function PARTITION(array, low, high)

 pivot \leftarrow array[high]

 i \leftarrow low - 1

for j = low to high - 1 **do**

if array[j] \leq pivot **then**

 i \leftarrow i + 1

 Intercambiar array[i] con array[j]

end if

end for

 Intercambiar array[i + 1] con array[high]

return i + 1

end function

2.2.7. Selection Sort

Selection Sort es un algoritmo de ordenamiento que funciona a través de una serie de selecciones. Divide el arreglo en dos partes: la parte ordenada y la parte no ordenada. En cada iteración, busca el elemento más pequeño de la parte no ordenada y lo intercambia con el primer elemento de esa sección. Este proceso se repite hasta que todo el arreglo está ordenado. Su complejidad en el peor y mejor caso es $O(n^2)$, lo que lo hace menos eficiente que algoritmos como Quick Sort o Merge Sort para grandes conjuntos de datos. Aunque Selection Sort no requiere memoria adicional significativa (es in-place), no es un algoritmo estable, lo que significa que no conserva el orden relativo de elementos iguales.

Algorithm 7 Selection Sort

Input: array, size
Output: sorted array
for $i = 0$ to size - 2 **do**
 minIndex $\leftarrow i$
 for $j = i + 1$ to size - 1 **do**
 if array[j] < array[minIndex] **then**
 minIndex $\leftarrow j$
 end if
 end for
 Intercambiar array[i] con array[minIndex]
end for

3. Metodología

3.1. Comparación de Algoritmos de Ordenamiento en Tres Lenguajes de Programación

La metodología utilizada para este estudio se basa en la implementación y comparación práctica de los algoritmos de ordenamiento seleccionados en los tres lenguajes de programación.

3.1.1. Implementación y Optimización de Algoritmos de Ordenamiento

En el primer paso, se implementarán cada uno de los algoritmos de ordenamiento en Java, Python y C++. Se prestará especial atención a la correcta optimización de cada implementación, siguiendo las mejores prácticas de programación en cada lenguaje.

3.1.2. Pruebas de Rendimiento: Medición de Tiempo de Ejecución y Uso de Memoria

Una vez que los algoritmos estén implementados, se llevarán a cabo pruebas de rendimiento. Estas pruebas consistirán en medir el tiempo de ejecución para diferentes tamaños de datos, desde listas pequeñas hasta grandes conjuntos de datos.

3.1.3. Presentación y Análisis de Resultados: Comparación Visual y Estadística de Algoritmos

Finalmente, los resultados se presentarán en forma de tablas y gráficos que facilitarán la comparación visual de los diferentes algoritmos en cada lenguaje. Se realizarán análisis estadísticos para interpretar los datos obtenidos.

4. Propuesta

4.1. Implementación y Optimización de Algoritmos de Ordenamiento

A continuación, se presentará la implementaciones de los diferentes algoritmos de ordenamiento en varios lenguajes de programación. Además, se incluirán diferentes métodos como leer archivos .txt que contienen una serie de números y para imprimir esos valores.

4.1.1. Código en C++.

```
1 #include <iostream>
2 #include <fstream>
3 #include <sstream>
4 #include <cstring>
5 #include <string>
6 #include <chrono>
7 using namespace std;
8
9 typedef void (*Sorteo)(int[], int);
10 void Bubble_sort(int array[], int size);
11 void Counting_sort(int array[], int size);
12 void heap_sort(int array[], int size);
13 void heapify(int array[], int n, int i);
14 void insertion_sort(int array[], int size);
15 void merge_sort(int array[], int size);
16 void merge(int array[], int left[], int leftSize, int right[], int
    rightSize);
17 void quickSort(int array[], int size);
18 void quickSort(int array[], int low, int high);
19 void selectionSort(int array[], int size);
20 int partition(int array[], int low, int high);
21
22 void imprimir(double tiempos[][21], int filas, int columnas);
23 int* LeerArchivo(int rango);
24
25 int main(){
26     int testing[]={100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000,
        8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000,
        90000, 100000};
27     const int cant_testing=sizeof(testing)/sizeof(testing[0]);
28     double tiempos[7][21];
29
30     Sorteo sortingMethods[] = {
31         Bubble_sort,
32         Counting_sort,
33         heap_sort,
34         insertion_sort,
```

```
35     merge_sort ,
36     quickSort ,
37     selectionSort
38 };
39     int cant_metodos=sizeof(sortingMethods)/sizeof(sortingMethods[0]);
40
41     for (int i=0;i<cant_testing;++i) {
42         int* numeros=LeerArchivo(testing[i]);
43         if(numeros==nullptr){
44             cerr<<"Error al leer el archivo del tama o: "<<testing[i]<<
endl;
45             continue;
46         }
47         for (int j=0;j<7;++j) {
48             int* numeros_copia=new int[testing[i]];
49             memcpy(numeros_copia, numeros, testing[i]*sizeof(int));
50             auto start=chrono::high_resolution_clock::now();
51             sortingMethods[j](numeros_copia, testing[i]);
52             auto end=chrono::high_resolution_clock::now();
53             chrono::duration<double> duration=end-start;
54             tiempos[j][i]=duration.count();
55             delete[] numeros_copia;
56         }
57         delete[] numeros; // Liberamos la memoria del arreglo original
58     }
59     imprimir(tiempos,cant_metodos,cant_testing);
60
61     return 0;
62 }
63 //1. Bubble sort
64 void Bubble_sort(int array[], int size){
65     bool cambio;
66     for(int i=0;i<size-1;i++){
67         cambio=false;
68         for(int j=0;j<size-1-i;j++){
69             if(array[j]>array[j+1]){
70                 int aux=array[j];
71                 array[j]=array[j+1];
72                 array[j+1]=aux;
73                 cambio=true;
74             }
75         }
76         if(!cambio) break;
77     }
78 }
79 //2. Counting sort
80 void Counting_sort(int array[], int size){
81     int max=array[0],min=array[0],range;
82     for(int i=0;i<size;i++){
83         if(array[i]>max){
```

```
84         max=array[i];
85     }
86     if(array[i]<min){
87         min=array[i];
88     }
89 }
90 range=max-min+1;
91 int conteo[range]={0};
92 int salida[size];
93
94 for(int i=0;i<size;i++){
95     conteo[array[i]-min]++;
96 }
97 for(int i=1;i<range;i++){
98     conteo[i]+=conteo[i-1];
99 }
100 for(int i=size-1;i>=0;i--){
101     salida[conteo[array[i]-min]-1]=array[i];
102     conteo[array[i]-min]--;
103 }
104 for(int i=0;i<size;i++){
105     array[i]=salida[i];
106 }
107
108 }
109 //3. Heap sort
110 void heap_sort(int array[],int size){
111     int n=size;
112     for(int i=n/2-1;i>=0;i--) {
113         heapify(array,n,i);
114     }
115
116     for (int i=n-1;i>0;i--) {
117         int aux=array[0];
118         array[0]=array[i];
119         array[i]=aux;
120         heapify(array, i, 0);
121     }
122 }
123 void heapify(int array[],int n,int i){
124     int indice_mayor= i;
125     int hijo_izquierdo=2*i+1;
126     int hijo_derecho=2*i+2;
127
128     if (hijo_izquierdo<n && array[hijo_izquierdo]>array[indice_mayor]) {
129         indice_mayor=hijo_izquierdo;
130     }
131     if (hijo_derecho<n && array[hijo_derecho] > array[indice_mayor]) {
132         indice_mayor=hijo_derecho;
133     }
```



```
134
135     if (indice_mayor!=i) {
136         int aux=array[i];
137         array[i]=array[indice_mayor];
138         array[indice_mayor]=aux;
139         heapify(array, n, indice_mayor);
140     }
141 }
142 //4. Insertion sort
143 void insertion_sort(int array[], int size){
144     for (int i=0;i<size;i++) {
145         int pos=i;
146         int aux=array[i];
147
148         while((pos>0 && (array[pos-1]>aux))){
149             array[pos]=array[pos-1];
150             pos--;
151         }
152         array[pos]=aux;
153     }
154 }
155 //5. Merge sort
156 void merge_sort(int array[], int size){
157     if (size< 2) {
158         return;
159     }
160     int mid=size/2;
161     int* left=new int[mid];
162     int* right=new int[size-mid];
163
164     for (int i=0;i<mid;i++) {
165         left[i]=array[i];
166     }
167
168     for (int i=mid;i<size;i++) {
169         right[i-mid]=array[i];
170     }
171
172     merge_sort(left,mid);
173     merge_sort(right,size-mid);
174
175     merge(array,left,mid,right,size-mid);
176     delete[] left;
177     delete[] right;
178 }
179 void merge(int array[], int left[], int leftSize, int right[], int
180 rightSize) {
181     int i=0,j=0,k=0;
182
183     while(i<leftSize && j<rightSize){
```

```
183         if(left[i]<=right[j]) {
184             array[k++]=left[i++];
185         }else{
186             array[k++]=right[j++];
187         }
188     }
189     while (i<leftSize) {
190         array[k++]=left[i++];
191     }
192
193     while (j<rightSize) {
194         array[k++]=right[j++];
195     }
196 }
197 //6. Quick sort
198 void quickSort(int array[],int size){
199     quickSort(array,0,size-1);
200 }
201 void quickSort(int array[],int low,int high){
202     if(low<high){
203         int partitionIndex=partition(array,low,high);
204         quickSort(array,low, partitionIndex-1);
205         quickSort(array,partitionIndex+1, high);
206     }
207 }
208 int partition(int array[], int low, int high){
209     int pivot=array[high];
210     int i=low-1;
211
212     for (int j=low;j<high;j++){
213         if(array[j]<=pivot) {
214             i++;
215             swap(array[i],array[j]);
216         }
217     }
218
219     swap(array[i+1],array[high]);
220     return i+1;
221 }
222 //7. Selection sort
223 void selectionSort(int array[], int size){
224     for(int i=0;i<size-1;i++){
225         int minIndex=i;
226         for(int j=i+1;j<size;j++){
227             if(array[j]<array[minIndex]){
228                 minIndex=j;
229             }
230         }
231         swap(array[i],array[minIndex]);
232     }
```

```

233 }
234 //Imprimir la matriz de tiempos
235 void imprimir(double tiempos[][21], int filas, int columnas){
236     for(int i=0; i<filas; i++){
237         for(int j=0; j<columnas; j++){
238             cout<<tiempos[i][j]<<" ";
239         }
240         cout<<endl;
241     }
242 }
243 int* LeerArchivo(int rango) {
244     string ruta="D:/gersael/Trabajo de ADA/Crear_numeros/"+to_string(rango)
245         + ".txt";
246     ifstream archivo(ruta);
247     if (!archivo.is_open()){
248         cerr<<"Error al abrir el archivo: "<<ruta<<endl;
249         return nullptr;
250     }
251     int* numeros=new int[rango];
252     string linea;
253     int i=0;
254     while(getline(archivo, linea) && i<rango) {
255         istringstream iss(linea);
256         int numero;
257         while (iss>>numero && i<rango){
258             numeros[i]=numero;
259             i++;
260         }
261     }
262     archivo.close();
263     return numeros;
264 }

```

4.1.2. Código en Java.

```

1 package algoritmos;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6 import java.util.Arrays;
7
8 public class algoritmo{
9     @FunctionalInterface
10     interface SortingMethod {
11         void sort(int[] array);
12     }
13     public static void main(String[] args){
14         int []testing
          = {100,500,1000,2000,3000,4000,5000,6000,7000,8000,9000,10000,20000,30000,40000,50000};
    
```

```
15     double [][] tiempos=new double [7][21];
16
17     SortingMethod[] sortingMethods = new SortingMethod []{
18         algoritmo::bubblesort ,
19         algoritmo::countingsort ,
20         algoritmo::heap_sort ,
21         algoritmo::insertion_sort ,
22         algoritmo::merge_sort ,
23         algoritmo::quickSort ,
24         algoritmo::selectionSort ,
25     };
26
27     for(int i=0;i<testing.length;i++){
28         int [] numeros=LeerArchivo(testing[i]);
29         for(int j=0;j<sortingMethods.length;j++){
30             int [] numeros_copia=Arrays.copyOf(numeros,numeros.length
31 );
32             long startTime=System.nanoTime();
33             sortingMethods[j].sort(numeros_copia);
34             long endTime=System.nanoTime();
35             long duration=endTime-startTime;
36             tiempos[j][i]=duration/1_000_000_000.0;
37         }
38     }
39     imprimir(tiempos);
40 }
41 //algoritmos sort
42 //1. bubble sort
43 public static void bubblesort(int [] array){
44     int size=array.length;
45     boolean cambio;
46     for(int i=0;i<size-1;i++){
47         cambio=false;
48         for(int j=0;j<size-1-i;j++){
49             if(array[j]>array[j+1]){
50                 int aux=array[j];
51                 array[j]=array[j+1];
52                 array[j+1]=aux;
53                 cambio=true;
54             }
55         }
56         if(!cambio) break;
57     }
58 }
59 //2. counting sort
60 public static void countingsort(int [] array){
61     int max=array[0],min=array[0],range;
62     for(int i=0;i<array.length;i++){
63         if(array[i]>max){
```

```
63         max=array[i];
64     }
65     if(array[i]<min){
66         min=array[i];
67     }
68 }
69
70 range=max-min+1;
71 int[] conteo=new int[range];
72 int[] salida=new int[array.length];
73 for(int i=0;i<array.length;i++){
74     conteo[array[i]-min]++;
75 }
76 for(int i=1;i<range;i++){
77     conteo[i]+=conteo[i-1];
78 }
79 for(int i=array.length-1;i>=0;i--){
80     salida[conteo[array[i]-min]-1]=array[i];
81     conteo[array[i]-min]--;
82 }
83 for(int i=0;i<array.length;i++){
84     array[i]=salida[i];
85 }
86
87 }
88 //3. heap sort
89 public static void heap_sort(int [] array){
90     int n = array.length;
91
92     for(int i=n/2-1;i>=0;i--) {
93         heapify(array,n,i);
94     }
95
96     for (int i=n-1;i>0;i--) {
97         int aux=array[0];
98         array[0]=array[i];
99         array[i]=aux;
100         heapify(array, i, 0);
101     }
102
103 }
104 private static void heapify(int[] array, int n, int i) {
105     int indice_mayor= i;
106     int hijo_izquierdo=2*i+1;
107     int hijo_derecho=2*i+2;
108
109     if (hijo_izquierdo<n && array[hijo_izquierdo]>array[indice_mayor
110 ]) {
111         indice_mayor=hijo_izquierdo;
```

```
112     if (hijo_derecho<n && array[hijo_derecho] > array[indice_mayor])
113     {
114         indice_mayor=hijo_derecho;
115     }
116     if (indice_mayor!=i) {
117         int aux=array[i];
118         array[i]=array[indice_mayor];
119         array[indice_mayor]=aux;
120         heapify(array, n, indice_mayor);
121     }
122 }
123 //4. insertion sort
124 public static void insertion_sort(int[] array){
125     for (int i=0;i<array.length;i++) {
126         int pos=i;
127         int aux=array[i];
128
129         while((pos>0 && (array[pos-1]>aux))){
130             array[pos]=array[pos-1];
131             pos--;
132         }
133         array[pos]=aux;
134     }
135 }
136 //5. Merge sort
137 public static void merge_sort(int[] array){
138     if(array.length < 2){
139         return;
140     }
141     int mid=array.length / 2;
142     int[] left=Arrays.copyOfRange(array, 0, mid);
143     int[] right=Arrays.copyOfRange(array, mid, array.length);
144
145     merge_sort(left);
146     merge_sort(right);
147
148     merge(array, left, right);
149 }
150 private static void merge(int[] arr, int[] left, int[] right) {
151     int i=0,j=0,k=0;
152
153     while (i<left.length && j<right.length) {
154         if(left[i]<=right[j]) {
155             arr[k++]=left[i++];
156         }else{
157             arr[k++]=right[j++];
158         }
159     }
160 }
```

```
161     while (i<left.length) {
162         arr[k++]=left[i++];
163     }
164
165     while (j<right.length) {
166         arr[k++]=right[j++];
167     }
168 }
169 //6. Quick sort
170 public static void quickSort(int[] array){
171     quickSort(array,0,array.length-1);
172 }
173 public static void quickSort(int[] array, int low, int high) {
174     if(low<high){
175         int partitionIndex=partition(array, low, high);
176         quickSort(array, low, partitionIndex - 1);
177         quickSort(array, partitionIndex + 1, high);
178     }
179 }
180 private static int partition(int[] array, int low, int high) {
181     int pivot=array[high];
182     int i=(low-1);
183
184     for(int j=low;j<high;j++){
185         if(array[j]<=pivot){
186             i++;
187             int aux=array[i];
188             array[i]=array[j];
189             array[j]=aux;
190         }
191     }
192     int temp=array[i+1];
193     array[i+1]=array[high];
194     array[high]=temp;
195
196     return i+1;
197 }
198 //7. Selection sort
199 public static void selectionSort(int[] array) {
200     int n=array.length;
201
202     for(int i=0;i<n-1;i++){
203         int minIndex=i;
204         for(int j=i+1;j<n;j++){
205             if(array[j]<array[minIndex]){
206                 minIndex=j;
207             }
208         }
209
210         int temp=array[minIndex];
```

```
211         array[minIndex]=array[i];
212         array[i]=temp;
213     }
214 }
215 //leer archivos .txt
216 public static int[] LeerArchivo(int rango) {
217     String ruta="D:\\gersael\\Trabajo de ADA\\Crear_numeros"+"\\\\"+
String.valueOf(rango)+".txt";
218     int [] numeros=new int[rango];
219     try (BufferedReader br=new BufferedReader(new FileReader(ruta)))
    {
220         String linea;
221         int i=0;
222         while((linea=br.readLine())!=null && i<rango) {
223             String[] numerosComoString=linea.split(" ");
224             for(String numStr : numerosComoString){
225                 if(i<rango) {
226                     numeros[i] = Integer.parseInt(numStr);
227                     i++;
228                 }
229             }
230         }
231     } catch(IOException e) {
232         e.printStackTrace();
233     }
234     return numeros;
235 }
236 //imprimir matriz de tiempos
237 public static void imprimir(double[][] array){
238     for(int i = 0; i < array.length; i++){
239         for (int j = 0; j < array[0].length; j++) {
240             System.out.print(array[i][j]);
241             if(j<array[0].length-1){
242                 System.out.print(",");
243             }
244         }
245         System.out.println();
246     }
247 }
248 }
```

4.1.3. Código en Python.

```
1 import time
2 import copy
3 import numpy as np
4 def bubblesort(array):
5     size=len(array)
6     cambio=False
7
8     for i in range(size-1):
```



```
9      cambio=False
10
11      for j in range(size-1-i):
12          if array[j]>array[j+1]:
13              array[j], array[j + 1] = array[j + 1], array[j]
14              cambio=True
15
16      if not cambio:
17          break
18 def counting_sort(array):
19     mx=max(array)
20     mn=min(array)
21     rng=mx-mn+1
22     conteo=[0]*rng
23     salida=[0]*len(array)
24     for i in range(len(array)):
25         conteo[array[i]-mn]+=1
26
27     for i in range(1,rng):
28         conteo[i]+=conteo[i-1]
29
30     for i in range(len(array)-1,-1,-1):
31         salida[conteo[array[i]-mn]-1]=array[i]
32         conteo[array[i]-mn]-=1
33
34     for i in range(len(array)):
35         array[i]=salida[i]
36 def heap_sort(array):
37     n=len(array)
38     for i in range(n//2-1,-1,-1):
39         heapify(array,n,i)
40
41     for i in range(n-1,0,-1):
42         aux=array[0]
43         array[0]=array[i]
44         array[i]=aux
45         heapify(array, i, 0)
46 def heapify(array,n,i):
47     indice_mayor=i
48     hijo_derecho=2*i+1
49     hijo_izquierdo=2*i+2
50     if hijo_izquierdo<n and array[hijo_izquierdo]>array[indice_mayor]:
51         indice_mayor=hijo_izquierdo
52
53     if hijo_derecho<n and array[hijo_derecho] > array[indice_mayor]:
54         indice_mayor=hijo_derecho
55
56     if indice_mayor!=i:
57         aux=array[i]
58         array[i]=array[indice_mayor]
```

```
59     array[indice_mayor]=aux
60     heapify(array, n, indice_mayor)
61 def insertion_sort(array):
62     for i in range(len(array)):
63         pos=i
64         aux=array[i]
65
66         while pos>0 and array[pos-1]>aux:
67             array[pos]=array[pos-1]
68             pos-=1
69         array[pos]=aux
70 def merge_sort(array):
71     if len(array)<2:
72         return array
73     mid=len(array)//2
74     left=merge_sort(array[:mid])
75     right=merge_sort(array[mid:])
76
77     return merge(left, right)
78 def merge(left, right):
79     merged=[]
80     i=j=0
81
82     while i<len(left) and j<len(right):
83         if left[i]<=right[j]:
84             merged.append(left[i])
85             i+=1
86         else:
87             merged.append(right[j])
88             j+=1
89     merged.extend(left[i:])
90     merged.extend(right[j:])
91     return merged
92 def quick_sort(array):
93     if len(array) < 2:
94         return array
95
96     pivot=array[-1]
97     left=[]
98     right=[]
99
100    for i in range(len(array) - 1):
101        if array[i]<=pivot:
102            left.append(array[i])
103        else:
104            right.append(array[i])
105
106    return quick_sort(left)+[pivot]+quick_sort(right)
107 def selection_sort(array):
108     n=len(array)
```

```
109
110     for i in range(n):
111         min_index=i
112         for j in range(i+1,n):
113             if array[j]<array[min_index]:
114                 min_index=j
115
116         array[i],array[min_index]=array[min_index],array[i]
117 def leer_archivo(rango):
118     ruta=f"D:/gersael/Trabajo de ADA/Crear_numeros/{rango}.txt"
119     numeros=[]
120     try:
121         with open(ruta, 'r') as archivo:
122             for linea in archivo:
123                 numeros_como_string=linea.split()
124                 for num_str in numeros_como_string:
125                     if len(numeros)<rango:
126                         numeros.append(int(num_str))
127     except IOError as e:
128         print(f"Error en el archivo: {ruta}")
129         print(e)
130
131     return numeros
132
133 testing=[100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000,
134          10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000,
135          100000]
136 tiempos=np.zeros((7,len(testing)))
137
138 sorteo_metodos=[
139     bubblesort,
140     counting_sort,
141     heap_sort,
142     insertion_sort,
143     merge_sort,
144     quick_sort,
145     selection_sort
146 ]
147
148 for i in range(len(testing)):
149     numeros=leer_archivo(testing[i])
150     for j in range (len(sorteo_metodos)):
151         numeros_copia=copy.deepcopy(numeros)
152         start_time=time.time()
153         sorteo_metodos[j](numeros_copia)
154         end_time=time.time()
155         duracion=end_time-start_time
156         tiempos[j][i]=duracion
157
158 print(tiempos)
```

4.2. Pruebas de Rendimiento: Medición de Tiempo de Ejecución y Uso de Memoria

4.2.1. Definición de Escenarios de Prueba

Se establecerán varios escenarios que incluyan diferentes tamaños de datos, comenzando desde listas pequeñas (por ejemplo, de 100 a 1000) y escalando hasta grandes conjuntos de datos (por ejemplo, 100000). Estas listas estarán desordenadas para evaluar el rendimiento en esta situación.

4.2.2. Medición del Tiempo de Ejecución

Se utilizarán temporizadores para medir el tiempo que cada algoritmo tarda en procesar los diferentes tamaños de datos. Esto se puede hacer a través de funciones específicas en el lenguaje de programación utilizado (por ejemplo, `System.nanoTime()` en Java o `time()` en C++). Se ejecutarán múltiples pruebas para cada tamaño de dato y se calculará el tiempo promedio de ejecución para obtener resultados más representativos.

4.3. Resultados

4.3.1. Resultados en Java.

Cuadro 1: Tiempos de Ejecución para Bubble Sort, Counting Sort y Heap Sort

Datos	Tiempo (s)	Datos	Tiempo (s)	Datos	Tiempo (s)
100	1.938E-4	100	4.82E-5	100	9.82E-5
500	0.0052466	500	8.28E-5	500	1.252E-4
1000	0.0019102	1000	1.477E-4	1000	7.215E-4
2000	0.0033435	2000	2.535E-4	2000	6.04E-4
3000	0.00756	3000	3.389E-4	3000	0.0034017
4000	0.0182686	4000	4.952E-4	4000	7.469E-4
5000	0.0310904	5000	6.049E-4	5000	8.62E-4
6000	0.0458611	6000	8.997E-4	6000	0.0010608
7000	0.1694462	7000	2.599E-4	7000	0.0013233
8000	0.1456742	8000	2.325E-4	8000	0.0020899
9000	0.1068609	9000	1.926E-4	9000	0.0015766
10000	0.1218712	10000	4.95E-5	10000	0.0010235
20000	0.5882881	20000	8.25E-5	20000	0.0026338
30000	1.4120705	30000	8.38E-5	30000	0.0032856
40000	2.5066909	40000	1.596E-4	40000	0.0044329
50000	3.910809	50000	1.926E-4	50000	0.0055629
60000	5.7167568	60000	1.847E-4	60000	0.0068137
70000	7.9613456	70000	2.166E-4	70000	0.0079103
80000	10.2269335	80000	3.147E-4	80000	0.0145751
90000	13.0583245	90000	3.576E-4	90000	0.0111748
100000	16.5934335	100000	2.908E-4	100000	0.0122198

Cuadro 2: Tiempos de Ejecución para Insertion Sort, Merge Sort y Quick Sort

Datos	Tiempo (s)	Datos	Tiempo (s)	Datos	Tiempo (s)
100	7.51E-5	100	1.069E-4	100	3.51E-5
500	0.0017098	500	5.449E-4	500	2.494E-4
1000	0.004813	1000	2.937E-4	1000	1.501E-4
2000	7.13E-4	2000	4.328E-4	2000	2.499E-4
3000	0.0014505	3000	0.004513	3000	0.0011097
4000	0.0025567	4000	7.808E-4	4000	3.412E-4
5000	0.0034699	5000	9.918E-4	5000	0.0010325
6000	0.0037654	6000	0.0011652	6000	0.0013077
7000	0.0065304	7000	0.0012765	7000	5.49E-4
8000	0.0113495	8000	0.0013764	8000	0.0032757
9000	0.0098525	9000	0.0044043	9000	0.0045521
10000	0.0072307	10000	0.0012917	10000	0.0020985
20000	0.032786	20000	0.0023942	20000	0.0089473
30000	0.0744814	30000	0.003163	30000	0.0199478
40000	0.1276205	40000	0.0050708	40000	0.0428479
50000	0.2018817	50000	0.008667	50000	0.0719473
60000	0.2802187	60000	0.0071439	60000	0.1102892
70000	0.3742836	70000	0.0082791	70000	0.1490283
80000	0.5036482	80000	0.0116453	80000	0.1948991
90000	0.626857	90000	0.0117112	90000	0.2434876
100000	0.7833399	100000	0.0115384	100000	0.2857284

Cuadro 3: Tiempos de Ejecución para Selection Sort

Datos	Tiempo (s)
100	1.032E-4
500	0.0017503
1000	9.688E-4
2000	0.0019033
3000	0.003021
4000	0.005599
5000	0.0119343
6000	0.0378161
7000	0.0227534
8000	0.0313945
9000	0.0273526
10000	0.0260107
20000	0.1040738
30000	0.2458172
40000	0.4537899
50000	0.6387729
60000	0.9261224
70000	1.2553659
80000	1.6705828
90000	2.1714559
100000	2.5882197

4.3.2. Resultados en C++.

Cuadro 4: Tiempos de Ejecución para Bubble Sort, Counting Sort y Heap Sort

Datos	Tiempo (s)	Datos	Tiempo (s)	Datos	Tiempo (s)
100	5.1E-05	100	1E-05	100	2.3E-05
500	0.000895	500	1.7E-05	500	0.000101
1000	0.001999	1000	1.6E-05	1000	0.000147
2000	0.006897	2000	3.3E-05	2000	0.000261
3000	0.015624	3000	4.8E-05	3000	0.00039
4000	0.027715	4000	7E-05	4000	0.000724
5000	0.048263	5000	9.4E-05	5000	0.001113
6000	0.06145	6000	8.1E-05	6000	0.000846
7000	0.093108	7000	0.000131	7000	0.001946
8000	0.116891	8000	0.000101	8000	0.001149
9000	0.145785	9000	0.000112	9000	0.001435
10000	0.205865	10000	0.000134	10000	0.001737
20000	1.07046	20000	0.000335	20000	0.004005
30000	3.035	30000	0.000574	30000	0.005924
40000	6.08519	40000	0.000507	40000	0.007463
50000	7.34785	50000	0.00059	50000	0.011972
60000	10.788	60000	0.000699	60000	0.011844
70000	16.4769	70000	0.001071	70000	0.024467
80000	20.0166	80000	0.000945	80000	0.01628
90000	27.4799	90000	0.001399	90000	0.026597
100000	35.105	100000	0.00118	100000	0.029002

Cuadro 5: Tiempos de Ejecución para Insertion Sort, Merge Sort y Quick Sort

Datos	Tiempo (s)	Datos	Tiempo (s)	Datos	Tiempo (s)
100	1.7E-05	100	0.000692	100	1.5E-05
500	0.000343	500	0.002457	500	6.9E-05
1000	0.000756	1000	0.004415	1000	0.000108
2000	0.002736	2000	0.012983	2000	0.000223
3000	0.006325	3000	0.021476	3000	0.000353
4000	0.010934	4000	0.034222	4000	0.000486
5000	0.023192	5000	0.070327	5000	0.00089
6000	0.023856	6000	0.054044	6000	0.000806
7000	0.035036	7000	0.024959	7000	0.00092
8000	0.042897	8000	0.014907	8000	0.001127
9000	0.056109	9000	0.242166	9000	0.002709
10000	0.067696	10000	0.204689	10000	0.00148
20000	0.291069	20000	1.39013	20000	0.003437
30000	0.681393	30000	0.834906	30000	0.006382
40000	1.1476	40000	3.03014	40000	0.010957
50000	1.75028	50000	0.08031	50000	0.014118
60000	2.71887	60000	2.35224	60000	0.019319
70000	3.94584	70000	15.8345	70000	0.024315
80000	4.33925	80000	0.098109	80000	0.034413
90000	5.84094	90000	14.9677	90000	0.0353
100000	7.63002	100000	10.8174	100000	0.045908

Cuadro 6: Tiempos de Ejecución para Selection Sort

Datos	Tiempo (s)
100	3.6E-05
500	0.000503
1000	0.001355
2000	0.005289
3000	0.012503
4000	0.021077
5000	0.035123
6000	0.048005
7000	0.073991
8000	0.084943
9000	0.112187
10000	0.137571
20000	0.581428
30000	1.46434
40000	2.50922
50000	3.33035
60000	5.9876
70000	6.9184
80000	8.58591
90000	12.8355
100000	14.7765

4.3.3. Resultados en Python.

Cuadro 7: Tiempos de Ejecución para Bubble Sort, Counting Sort y Heap Sort

Datos	Tiempo (s)	Datos	Tiempo (s)	Datos	Tiempo (s)
0.0	0.00000000e+00	0.0	0.00000000e+00	0.0	0.00000000e+00
1	1.56257153e-02	1	0.00000000e+00	1	0.00000000e+00
2	4.69267368e-02	2	0.00000000e+00	2	0.00000000e+00
3	2.30518103e-01	3	9.98258591e-04	3	5.97953796e-03
4	5.01944542e-01	4	0.00000000e+00	4	0.00000000e+00
5	9.05662537e-01	5	0.00000000e+00	5	1.56233311e-02
6	1.38176894e+00	6	0.00000000e+00	6	1.80084705e-02
7	1.99803042e+00	7	0.00000000e+00	7	3.12466621e-02
8	2.70447922e+00	8	0.00000000e+00	8	3.12292576e-02
9	3.52774048e+00	9	0.00000000e+00	9	1.56230927e-02
10	4.54441500e+00	10	0.00000000e+00	10	3.55217457e-02
11	5.73008585e+00	11	0.00000000e+00	11	3.12469006e-02
12	2.21260002e+01	12	1.56240463e-02	12	9.31251049e-02
13	5.17758090e+01	13	1.56219006e-02	13	1.25024557e-01
14	8.89954815e+01	14	1.56280994e-02	14	1.71806574e-01
15	1.43525966e+02	15	1.56228542e-02	15	2.19347239e-01
16	1.98997158e+02	16	3.12445164e-02	16	2.90744543e-01
17	2.80856373e+02	17	1.56772137e-02	17	3.12468529e-01
18	3.56218605e+02	18	2.39198208e-02	18	3.83722067e-01
19	4.50912164e+02	19	2.69045830e-02	19	4.83337402e-01
20	5.55684031e+02	20	3.08969021e-02	20	4.89367723e-01

Cuadro 8: Tiempos de Ejecución para Insertion Sort, Merge Sort y Quick Sort)

Datos	Tiempo (s)	Datos	Tiempo (s)	Datos	Tiempo (s)
0.0	0.00000000e+00	0.0	0.00000000e+00	0.0	0.00000000e+00
1	0.00000000e+00	1	0.00000000e+00	1	0.00000000e+00
2	0.00000000e+00	2	0.00000000e+00	2	0.00000000e+00
3	1.56297684e-02	3	0.00000000e+00	3	1.56173706e-02
4	1.02324486e-01	4	1.56242847e-02	4	1.56238079e-02
5	2.49690294e-01	5	0.00000000e+00	5	0.00000000e+00
6	4.55706596e-01	6	0.00000000e+00	6	1.56321526e-02
7	6.90638304e-01	7	1.56216621e-02	7	1.63121223e-02
8	9.99198437e-01	8	1.56238079e-02	8	1.56230927e-02
9	1.36985087e+00	9	3.21872234e-02	9	1.15544796e-02
10	1.76438975e+00	10	3.12447548e-02	10	1.56230927e-02
11	2.26423597e+00	11	3.12471390e-02	11	1.56235695e-02
12	2.82991290e+00	12	4.68714237e-02	12	1.56238079e-02
13	1.12059011e+01	13	7.81240463e-02	13	4.68819141e-02
14	2.53226099e+01	14	1.24988317e-01	14	9.36980247e-02
15	4.54007187e+01	15	1.40601397e-01	15	1.41101837e-01
16	7.09822209e+01	16	1.71808720e-01	16	1.80535793e-01
17	1.04037443e+02	17	2.11305141e-01	17	2.52150536e-01
18	1.38776112e+02	18	2.42140293e-01	18	2.95016289e-01
19	1.82618368e+02	19	2.74088144e-01	19	3.79733562e-01
20	2.28714460e+02	20	3.07026863e-01	20	4.54482317e-01
21	2.83626488e+02	21		21	5.64116478e-01

Cuadro 9: Tiempos de Ejecución para Selection Sort)

Datos	Tiempo (s)
0	0.00000000e+00
1	1.56655312e-02
2	1.56219006e-02
3	1.56202316e-02
4	7.81700611e-02
5	1.87481642e-01
6	3.08452368e-01
7	4.97750044e-01
8	7.31855392e-01
9	9.54062700e-01
10	1.26771402e+00
11	1.58076668e+00
12	1.96320724e+00
13	7.86725283e+00
14	1.77597535e+01
15	3.17364285e+01
16	4.91117587e+01
17	7.18987072e+01
18	9.72151854e+01
19	1.26917149e+02
20	1.59406486e+02
21	1.99318351e+02

4.3.4. Resultados Bubble Sort

Cuadro 10: Tiempos de Ejecución para Bubble Sort en Diferentes Lenguajes

Datos	Tiempo (Java) (s)	Tiempo (C++) (s)	Tiempo (Python) (s)
100	1.938E-4	5.1e-05	0.00000000e+00
500	0.0052466	0.000895	1.56257153e-02
1000	0.0019102	0.001999	4.69267368e-02
2000	0.0033435	0.006897	2.30518103e-01
3000	0.00756	0.015624	5.01944542e-01
4000	0.0182686	0.027715	9.05662537e-01
5000	0.0310904	0.048263	1.38176894e+00
6000	0.0458611	0.06145	1.99803042e+00
7000	0.1694462	0.093108	2.70447922e+00
8000	0.1456742	0.116891	3.52774048e+00
9000	0.1068609	0.145785	4.54441500e+00
10000	0.1218712	0.205865	5.73008585e+00
20000	0.5882881	1.07046	2.21260002e+01
30000	1.4120705	3.035	5.17758090e+01
40000	2.5066909	6.08519	8.89954815e+01
50000	3.910809	7.34785	1.43525966e+02
60000	5.7167568	10.788	1.98997158e+02
70000	7.9613456	16.4769	2.80856373e+02
80000	10.2269335	20.0166	3.56218605e+02
90000	13.0583245	27.4799	4.50912164e+02
100000	16.5934335	35.105	5.55684031e+02

4.3.5. Resultados Counting Sort

Cuadro 11: Tiempos de Ejecución para Counting Sort

Datos	Java (s)	C++ (s)	Python (s)
100	4.82E-5	1.0E-5	0.00000000
500	8.28E-5	1.7E-5	0.00000000
1000	1.477E-4	1.6E-5	0.00000000
2000	2.535E-4	3.3E-5	9.98258591E-04
3000	3.389E-4	4.8E-5	0.00000000
4000	4.952E-4	7E-5	0.00000000
5000	6.049E-4	9.4E-5	0.00000000
6000	8.997E-4	8.1E-5	0.00000000
7000	2.599E-4	0.000131	0.00000000
8000	2.325E-4	0.000101	0.00000000
9000	1.926E-4	0.000112	0.00000000
10000	4.95E-5	0.000134	0.00000000
20000	8.25E-5	0.000335	1.56240463E-02
30000	8.38E-5	0.000574	1.56219006E-02
40000	1.596E-4	0.000507	1.56280994E-02
50000	1.926E-4	0.00059	1.56228542E-02
60000	1.847E-4	0.000699	3.12445164E-02
70000	2.166E-4	0.001071	1.56772137E-02
80000	3.147E-4	0.000945	2.39198208E-02
90000	3.576E-4	0.001399	2.69045830E-02
100000	2.908E-4	0.00118	3.08969021E-02

4.3.6. Resultados Heap Sort

Cuadro 12: Tiempos de Ejecución para Heap Sort

Datos	Java (s)	C++ (s)	Python (s)
100	9.82E-5	2.3e-05	0.00000000e+00
500	1.252E-4	0.000101	0.00000000e+00
1000	7.215E-4	0.000147	0.00000000e+00
2000	6.04E-4	0.000261	5.97953796e-03
3000	0.0034017	0.00039	0.00000000e+00
4000	7.469E-4	0.000724	1.56233311e-02
5000	8.62E-4	0.001113	1.80084705e-02
6000	0.0010608	0.000846	3.12466621e-02
7000	0.0013233	0.001946	3.12292576e-02
8000	0.0020899	0.001149	1.56230927e-02
9000	0.0015766	0.001435	3.55217457e-02
10000	0.0010235	0.001737	3.12469006e-02
20000	0.0026338	0.004005	9.31251049e-02
30000	0.0032856	0.005924	1.25024557e-01
40000	0.0044329	0.007463	1.71806574e-01
50000	0.0055629	0.011972	2.19347239e-01
60000	0.0068137	0.011844	2.90744543e-01
70000	0.0079103	0.024467	3.12468529e-01
80000	0.0145751	0.01628	3.83722067e-01
90000	0.0111748	0.026597	4.83337402e-01
100000	0.0122198	0.029002	4.89367723e-01

4.3.7. Resultados Insertion Sort

Cuadro 13: Tiempos de Ejecución para Insertion Sort

Datos	Java (s)	C++ (s)	Python (s)
100	7.51E-5	1.7e-05	0.00000000e+00
500	0.0017098	0.000343	0.00000000e+00
1000	0.004813	0.000756	1.56297684e-02
2000	7.13E-4	0.002736	1.02324486e-01
3000	0.0014505	0.006325	2.49690294e-01
4000	0.0025567	0.010934	4.55706596e-01
5000	0.0034699	0.023192	6.90638304e-01
6000	0.0037654	0.023856	9.99198437e-01
7000	0.0065304	0.035036	1.36985087e+00
8000	0.0113495	0.042897	1.76438975e+00
9000	0.0098525	0.056109	2.26423597e+00
10000	0.0072307	0.067696	2.82991290e+00
20000	0.032786	0.291069	1.12059011e+01
30000	0.0744814	0.681393	2.53226099e+01
40000	0.1276205	1.1476	4.54007187e+01
50000	0.2018817	1.75028	7.09822209e+01
60000	0.2802187	2.71887	1.04037443e+02
70000	0.3742836	3.94584	1.38776112e+02
80000	0.5036482	4.33925	1.82618368e+02
90000	0.626857	5.84094	2.28714460e+02
100000	0.7833399	7.63002	2.83626488e+02

4.3.8. Resultados Merge Sort

Cuadro 14: Tiempos de Ejecución para Merge Sort

Datos	Java (s)	C++ (s)	Python (s)
100	1.069E-4	0.000692	0.00000000e+00
500	5.449E-4	0.002457	0.00000000e+00
1000	2.937E-4	0.004415	0.00000000e+00
2000	4.328E-4	0.012983	0.00000000e+00
3000	0.004513	0.021476	0.015624
4000	7.808E-4	0.034222	0.00000000e+00
5000	9.918E-4	0.070327	0.00000000e+00
6000	0.0011652	0.054044	1.56216621e-02
7000	0.0012765	0.024959	1.56238079e-02
8000	0.0013764	0.014907	3.21872234e-02
9000	0.0044043	0.242166	3.12447548e-02
10000	0.0012917	0.204689	3.12471390e-02
20000	0.0023942	1.39013	4.68714237e-02
30000	0.003163	0.834906	7.81240463e-02
40000	0.0050708	3.03014	1.24988317e-01
50000	0.008667	0.08031	1.40601397e-01
60000	0.0071439	2.35224	1.71808720e-01
70000	0.0082791	15.8345	2.11305141e-01
80000	0.0116453	0.098109	2.42140293e-01
90000	0.0117112	14.9677	2.74088144e-01
100000	0.0115384	10.8174	3.07026863e-01

4.3.9. Resultados Quick Sort

Cuadro 15: Tiempos de Ejecución para Quick sort

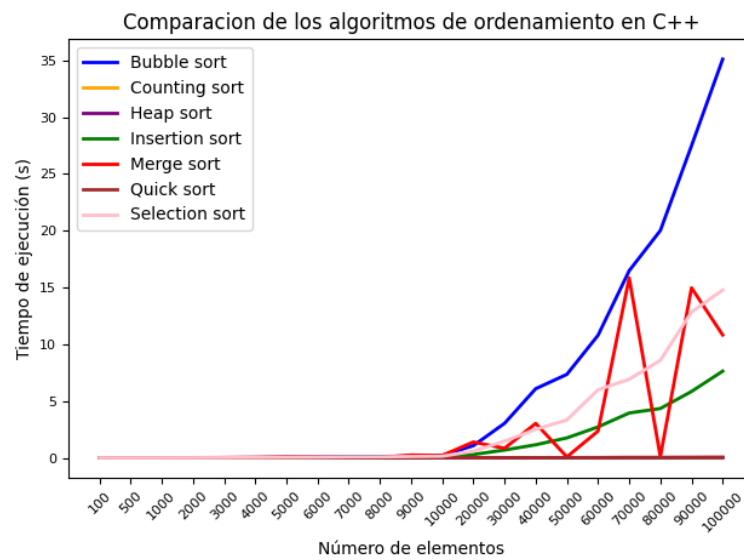
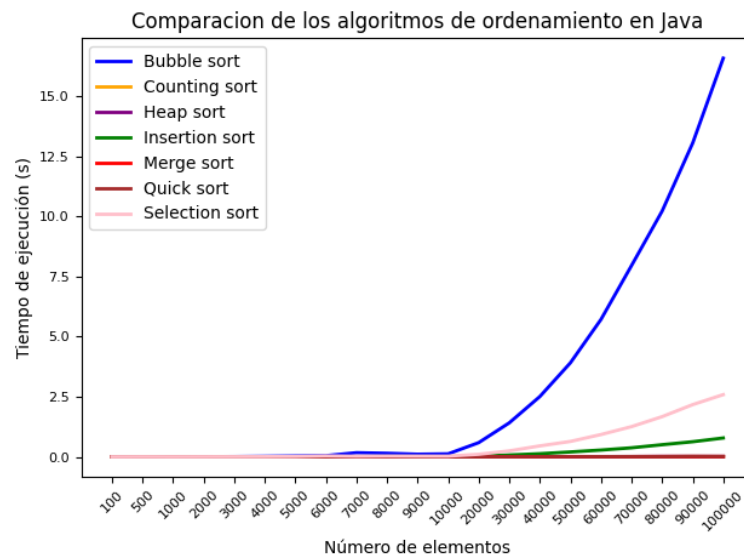
Datos	Columna 1	Columna 2	Columna 3
100	3.51E-5	1.5e-05	0.00000000e+00
500	2.494E-4	6.9e-05	0.00000000e+00
1000	1.501E-4	0.000108	1.56173706e-02
2000	2.499E-4	0.000223	1.56238079e-02
3000	0.0011097	0.000353	0.00000000e+00
4000	3.412E-4	0.000486	1.56321526e-02
5000	0.0010325	0.00089	1.63121223e-02
6000	0.0013077	0.000806	1.56230927e-02
7000	5.49E-4	0.00092	1.15544796e-02
8000	5.915E-4	0.001127	1.56230927e-02
9000	7.253E-4	0.002709	1.56235695e-02
10000	5.945E-4	0.00148	1.56238079e-02
20000	0.0012494	0.003437	4.68819141e-02
30000	0.0020551	0.006382	9.36980247e-02
40000	0.002999	0.010957	1.41101837e-01
50000	0.0042619	0.014118	1.80535793e-01
60000	0.0052183	0.019319	2.52150536e-01
70000	0.0083029	0.024315	2.95016289e-01
80000	0.0077096	0.034413	3.79733562e-01
90000	0.0185676	0.0353	4.54482317e-01
100000	0.013429	0.045908	5.64116478e-01

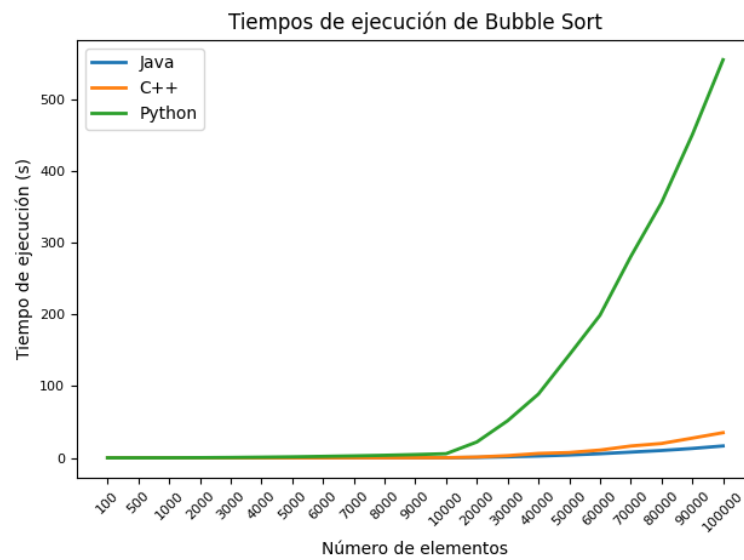
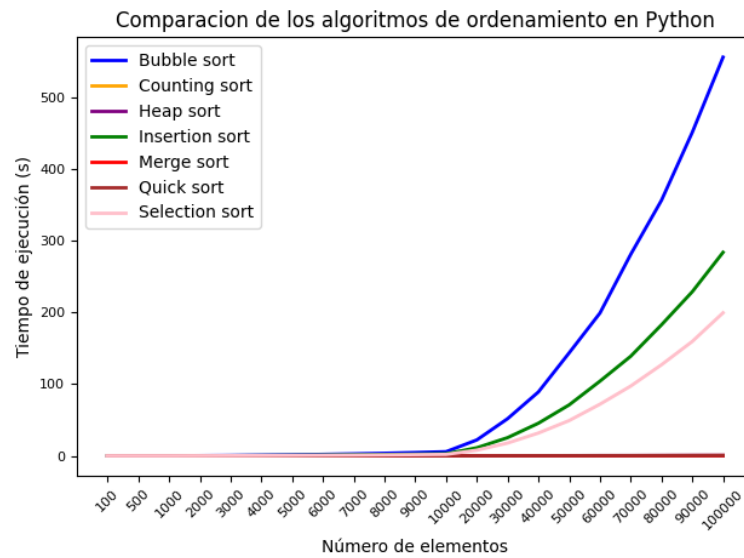
4.3.10. Resultados Selection Sort

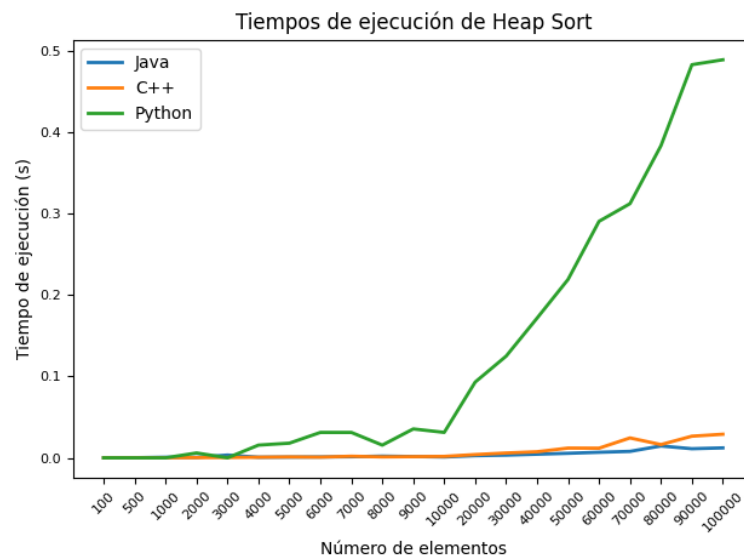
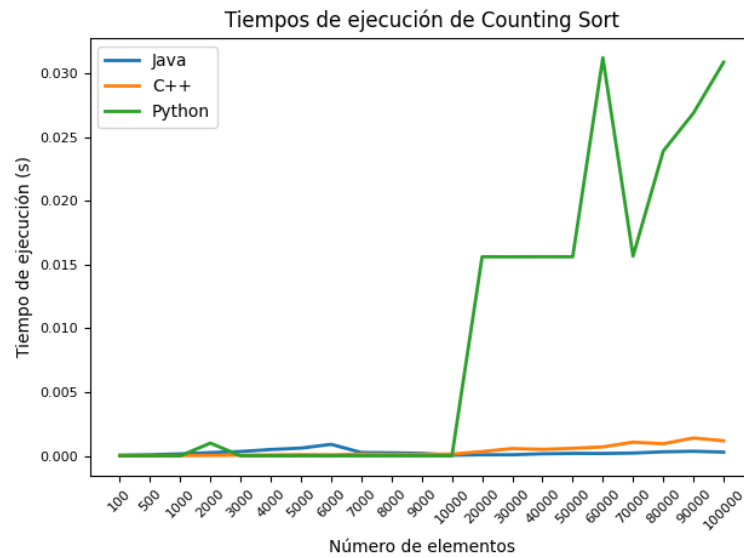
Cuadro 16: Tiempos de Ejecución para Selection Sort

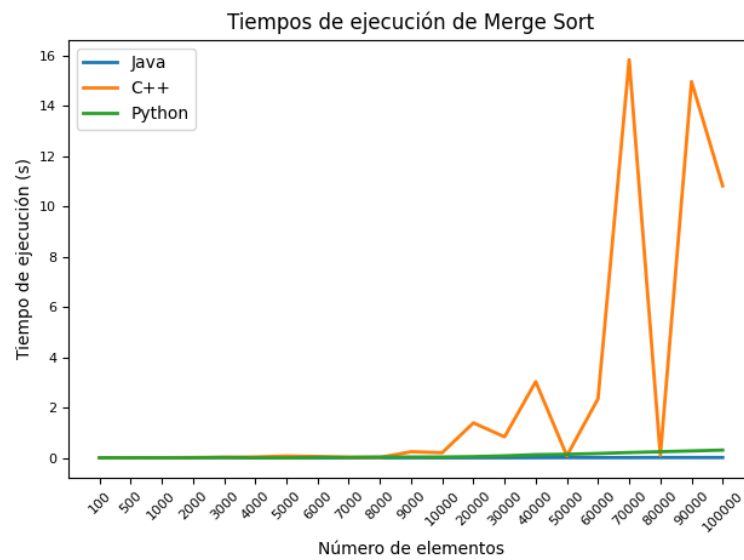
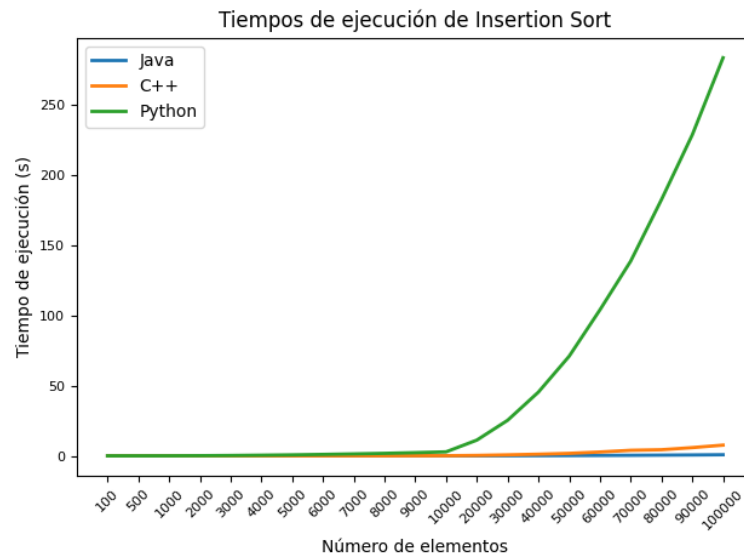
Datos	Columna 1	Columna 2	Columna 3
100	1.032E-4	3.6e-05	1.56655312e-02
500	0.0017503	0.000503	1.56219006e-02
1000	9.688E-4	0.001355	1.56202316e-02
2000	0.0019033	0.005289	7.81700611e-02
3000	0.003021	0.012503	1.87481642e-01
4000	0.005599	0.021077	3.08452368e-01
5000	0.0119343	0.035123	4.97750044e-01
6000	0.0378161	0.048005	7.31855392e-01
7000	0.0227534	0.073991	9.54062700e-01
8000	0.0313945	0.084943	1.26771402e+00
9000	0.0273526	0.112187	1.58076668e+00
10000	0.0260107	0.137571	1.96320724e+00
20000	0.1040738	0.581428	7.86725283e+00
30000	0.2458172	1.46434	1.77597535e+01
40000	0.4537899	2.50922	3.17364285e+01
50000	0.6387729	3.33035	4.91117587e+01
60000	0.9261224	5.9876	7.18987072e+01
70000	1.2553659	6.9184	9.72151854e+01
80000	1.6705828	8.58591	1.26917149e+02
90000	2.1714559	12.8355	1.59406486e+02
100000	2.5882197	14.7765	1.99318351e+02

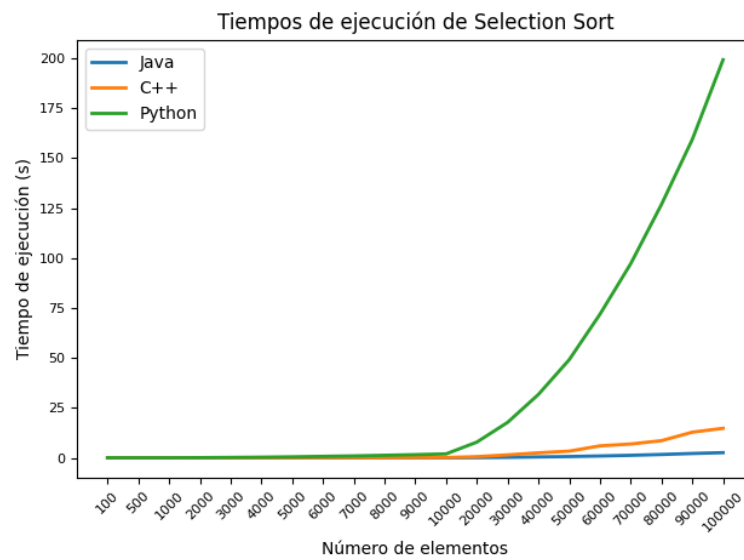
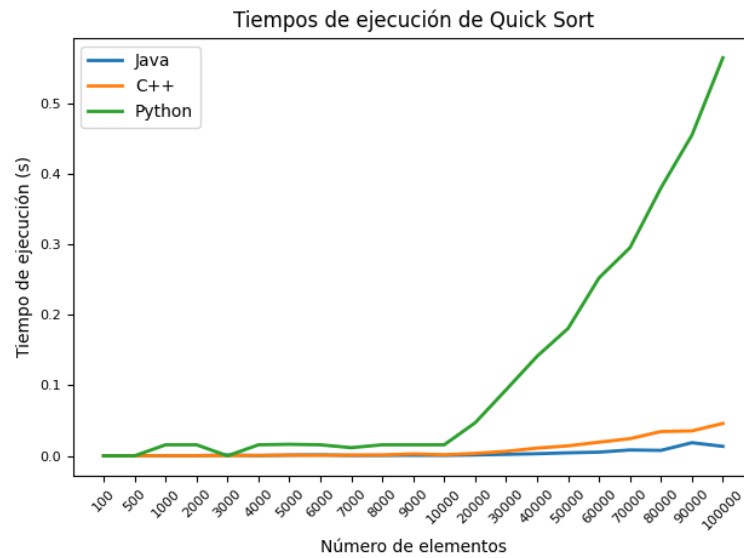
4.3.11. Graficos:











5. Conclusiones

1. Rendimiento de Algoritmos de Ordenamiento en Diferentes Lenguajes: La implementación de los algoritmos en Java, C++ y Python mostró variaciones importantes en términos de tiempo de ejecución. En general, C++ tiende a ser más rápido que Java y Python en la mayoría de los casos, debido a su proximidad al hardware y su gestión eficiente de la memoria. Python, por otro lado, fue consistentemente el más lento, lo que se atribuye a su naturaleza interpretada y la sobrecarga del intérprete.
2. Optimización y Eficiencia: En cuanto a la optimización, se observó que en Python las optimizaciones son menos efectivas en comparación con Java y C++, lo cual resalta la importancia de elegir el lenguaje adecuado según las necesidades de rendimiento. Java se comportó de manera intermedia, con un rendimiento aceptable, aunque ligeramente más lento que C++, especialmente en casos de conjuntos de datos más grandes.
3. Diferencias entre Algoritmos: Los algoritmos como Bubble Sort y Selection Sort, aunque simples y fáciles de implementar, son ineficientes para grandes volúmenes de datos, mostrando tiempos de ejecución significativamente mayores en comparación con Merge Sort y Quick Sort. Esto refuerza la recomendación de utilizar algoritmos más avanzados para escenarios donde se manejen grandes volúmenes de datos, dado que el aumento exponencial en el tiempo de ejecución de los algoritmos menos eficientes es considerable.
4. Recomendación de Algoritmos por Lenguaje: Para implementaciones donde la velocidad y la eficiencia son críticas, se recomienda el uso de C++ junto con algoritmos como Merge Sort o Quick Sort. En escenarios donde la simplicidad es prioritaria, y el rendimiento no es un factor crítico, Python puede ser útil. Sin embargo, para aplicaciones de propósito general, Java sigue siendo una opción equilibrada, proporcionando una buena combinación de rendimiento y facilidad de implementación.

6. Recomendaciones

1. Seleccionar el Lenguaje de Programación Basado en el Contexto del Proyecto: Si la eficiencia y el rendimiento son cruciales, como en aplicaciones en tiempo real o donde se manejen grandes volúmenes de datos, se recomienda optar por C++, ya que generalmente ofrece un mejor manejo de memoria y tiempos de ejecución más rápidos. Para proyectos que requieren desarrollo rápido y facilidad de uso, pero donde el rendimiento no es tan crítico, Python es una opción viable. Java puede ser utilizado cuando se necesita un equilibrio entre rendimiento y portabilidad.
2. Utilizar Algoritmos con Complejidad $O(n \log n)$ para Grandes Volúmenes de Datos: Siempre que se trabajen con grandes conjuntos de datos, se recomienda utilizar algoritmos eficientes como Merge Sort o Quick Sort, ya que ofrecen tiempos de ejecución significativamente más bajos que los algoritmos de ordenación más simples, como Bubble Sort o Selection Sort. Además, en aplicaciones que requieran estabilidad en el orden de los datos, Merge Sort es una opción ideal debido a su naturaleza estable.
3. Optimizar y Probar en Diferentes Entornos de Hardware: Es fundamental realizar pruebas de rendimiento y optimización no solo en un entorno de desarrollo, sino también en el hardware y las condiciones reales en las que se ejecutará el proyecto final. Además, se recomienda realizar pruebas utilizando diferentes configuraciones de hardware (CPU, memoria, almacenamiento) para obtener una evaluación integral del rendimiento del algoritmo. De esta manera, se puede identificar el mejor algoritmo y lenguaje de programación para el entorno específico en que será implementado.

Referencias

- [1] Jesús Mario Rodríguez López. Análisis comparativo de algoritmos de ordenamiento, 2014. Consultado el 22 de octubre de 2024.
- [2] Roberto Sánchez García. Algoritmos de ordenamiento: Análisis y comportamiento en diferentes lenguajes de programación, 2016. Consultado el 22 de octubre de 2024.
- [2]. [1]