UNIVERSIDAD NACIONAL DE MOQUEGUA

Facultad de Ingeniería y Arquitectura

Escuela Profesional de Ingeniería de Sistemas e Informática

Algoritmos de Ordenamiento

Practica 01

Estudiantes:
Parancco Ccoaricona, Milagros

Profesores:
Honorio Apaza Alanoca

29 de octubre de 2024

Índice

1. Introducción

El ordenamiento de datos es una tarea fundamental en la informática y se aplica en diversas áreas, desde la búsqueda eficiente hasta la presentación de información. Un algoritmo de ordenamiento organiza los elementos de una lista o matriz en un orden específico, ya sea ascendente o descendente. En este trabajo, se explorarán varios algoritmos de ordenamiento implementados en Java utilizando Apache NetBeans. Se analizarán sus principios de funcionamiento, ventajas y desventajas, así como ejemplos de código para ilustrar su implementación.

1.1. Motivación y Contexto

En la era del Big Data, la necesidad de manipular y procesar grandes volúmenes de información es más relevante que nunca. Elegir el algoritmo de ordenación adecuado puede tener un impacto significativo en la eficiencia del procesamiento de datos. Además, cada lenguaje de programación tiene características y optimizaciones que pueden influir en el rendimiento de los algoritmos. Esta investigación busca no solo comparar la complejidad teórica de cada algoritmo, sino también su rendimiento práctico en diferentes entornos de programación. La motivación detrás de este estudio radica en la necesidad de optimizar el manejo de datos en un mundo impulsado por la información. Los algoritmos de ordenación son fundamentales para muchas aplicaciones, desde bases de datos hasta análisis de datos en tiempo real. Al comparar estos algoritmos en distintos lenguajes de programación, se busca entender cómo las diferencias en implementación pueden afectar el rendimiento y la eficiencia.

1.2. Objetivo general

Analizar y comparar diferentes algoritmos de ordenamiento en términos de complejidad, eficiencia y tiempo de ejecución. Evaluar el rendimiento de estos algoritmos en tres lenguajes de programación: Python, Java y C++.

1.3. Objetivos específicos

- Describir el funcionamiento y las características de cada algoritmo de ordenamiento.
- Implementar cada algoritmo en Python, Java y C++.
- Medir y comparar el tiempo de ejecución de cada algoritmo para diferentes conjuntos de datos.
- Analizar la complejidad temporal y espacial de cada algoritmo.
- Evaluar cómo las diferencias en la implementación de los lenguajes de programación afectan el rendimiento de los algoritmos.

1.4. Justificación

Este estudio es relevante debido a la importancia de los algoritmos de ordenamiento en la optimización de procesos de manejo de datos. La comparación de estos algoritmos en tres lenguajes de programación permitirá entender no solo su eficiencia teórica, sino también su rendimiento práctico en entornos de desarrollo diversos. Los resultados de esta investigación pueden ser útiles para desarrolladores e investigadores que busquen mejorar el rendimiento de sus aplicaciones al seleccionar el algoritmo de ordenamiento más adecuado según el contexto y el lenguaje utilizado.

2. Marco Teórico

La búsqueda de datos es una de las operaciones más fundamentales en la informática, permitiendo a los usuarios y sistemas localizar información específica dentro de conjuntos de datos. Existen varios algoritmos de búsqueda, cada uno diseñado para diferentes escenarios y tipos de datos. En esta presentación, exploraremos siete pseudocódigos de búsqueda, que incluyen:

2.1. Bubble sort

Bubble Sort es un algoritmo de ordenamiento sencillo que compara elementos adyacentes en un arreglo y los intercambia si están en el orden incorrecto. Este proceso se repite, haciendo que los elementos más grandes "burbujeen"hacia el final del arreglo. Aunque es fácil de entender e implementar, su eficiencia es limitada, con una complejidad promedio de O(n^2), lo que lo hace menos adecuado para listas grandes.

Algorithm 1 BubbleSort

```
1:  procedure BubbleSort(arr)
2:      n ← length(arr)
3:      for i = 0 to n − 1 do
4:          swapped ← false
5:          for j = 0 to n − i − 2 do
6:              if arr[j] > arr[j + 1] then
7:                  temp ← arr[j]
8:                  arr[j] ← arr[j + 1]
9:                  arr[j + 1] ← temp
10:                 swapped ← true
11:             end if
12:         end for
13:         if not swapped then
14:             break
15:         end if
16:     end for
17: end procedure
```

2.2. Counting sort

Counting Sort es un algoritmo de ordenamiento eficiente que se utiliza principalmente para ordenar números enteros dentro de un rango específico. A diferencia de los algoritmos de comparación tradicionales, Counting Sort calcula la posición de cada elemento en la lista ordenada utilizando un array auxiliar, que cuenta la frecuencia de cada elemento. Este

método es especialmente efectivo cuando el rango de los datos es conocido y relativamente pequeño en comparación con la cantidad de elementos a ordenar, lo que permite lograr una complejidad temporal de O(n + k), donde n es el número de elementos y k es el rango de los elementos.

Algorithm 2 Counting Sort

```
    procedure CountingSort(A)
2:      n ← length(A)
        max ← max(A)
4:      count ← new array of size (max + 1)
        for i ← 0 to n − 1 do
6:          count[A[i]] ← count[A[i]] + 1
        end for
8:      for i ← 1 to max do
            count[i] ← count[i] + count[i − 1]
10:     end for
        output ← new array of size n
12:     for i ← n − 1 to 0 do
            output[count[A[i]] − 1] ← A[i]
14:         count[A[i]] ← count[A[i]] − 1
        end for
16:     A ← output
    end procedure
```

2.3. Heap sort

Heap Sort es un algoritmo de ordenamiento que utiliza la estructura de datos llamada "heap", que es un árbol binario completo que cumple con la propiedad de heap. En un heap, para cualquier nodo dado, el valor de este nodo es mayor o igual (en un max-heap) que el de sus hijos, lo que permite obtener el máximo elemento de manera eficiente. Heap Sort funciona en dos etapas: primero, convierte la lista en un max-heap, y luego extrae el elemento máximo repetidamente para construir la lista ordenada. Este algoritmo tiene una complejidad temporal de O(n log n) y es eficiente en términos de espacio, ya que opera en el lugar, lo que significa que no requiere memoria adicional significativa más allá del array original.

Algorithm 3 Heap Sort

```
    procedure HeapSort(A)
        n ← length(A)
 3:     for i ← n/2 − 1 to 0 do
            Heapify(A, n, i)
        end for
 6:     for i ← n − 1 to 1 do
            temp ← A[0]
            A[0] ← A[i]
 9:         A[i] ← temp
            Heapify(A, i, 0)
        end for
12: end procedure
    procedure Heapify(A, n, i)
        largest ← i
15:     left ← 2 · i + 1
        right ← 2 · i + 2
        if left < n and A[left] > A[largest] then
18:         largest ← left
        end if
        if right < n and A[right] > A[largest] then
21:         largest ← right
        end if
        if largest ≠ i then
24:         temp ← A[i]
            A[i] ← A[largest]
            A[largest] ← temp
27:         Heapify(A, n, largest)
        end if
    end procedure
```

2.4. Insertion sort

Insertion Sort es un algoritmo de ordenamiento simple y eficiente que construye una lista ordenada de manera incremental. Funciona dividiendo la lista en dos partes: una parte ordenada y otra desordenada. El algoritmo toma un elemento de la parte desordenada y lo inserta en la posición correcta de la parte ordenada, repitiendo este proceso hasta que todos los elementos están ordenados. Su complejidad temporal es O(n2) en el peor caso, pero puede ser muy eficiente para listas pequeñas o listas que ya están parcialmente ordenadas, lo que lo convierte en una opción práctica en diversas aplicaciones. Además, es un algoritmo estable, lo que significa que preserva el orden de los elementos iguales.

Algorithm 4 Insertion Sort

```
    procedure InsertionSort(A)
        for i ← 1 to length(A) − 1 do
            key ← A[i]
4:          j ← i − 1
            while j ≥ 0 and A[j] > key do
                A[j + 1] ← A[j]
                j ← j − 1
8:          end while
            A[j + 1] ← key
        end for
    end procedure
```

2.5. Merge sort

Merge Sort es un algoritmo de ordenamiento eficiente que utiliza la técnica de divide y vencerás. Divide repetidamente una lista en dos mitades hasta que cada sublista contiene un solo elemento, lo que se considera ordenado. Luego, combina estas sublistas de manera ordenada para formar listas más grandes hasta que toda la lista esté reunida en un solo arreglo ordenado. Su complejidad temporal es O(n log n) en todos los casos, lo que lo convierte en una opción preferida para manejar grandes conjuntos de datos. Además, Merge Sort es estable y funciona bien en listas enlazadas y en arreglos, aunque requiere espacio adicional para la combinación de las sublistas.

Algorithm 5 Merge Sort

```
    procedure MergeSort(A, left, right)
        if left < right then
            mid ← ⌊(left+right)/2⌋
            MergeSort(A, left, mid)
 5:         MergeSort(A, mid + 1, right)
            Merge(A, left, mid, right)
        end if
    end procedure
    procedure Merge(A, left, mid, right)
10:     n1 ← mid − left + 1
        n2 ← right − mid
        L ← new array of size n1
        R ← new array of size n2
        for i ← 0 to n1 − 1 do
15:         L[i] ← A[left + i]
        end for
        for j ← 0 to n2 − 1 do
            R[j] ← A[mid + 1 + j]
        end for
20:     i ← 0, j ← 0, k ← left
        while i < n1 and j < n2 do
            if L[i] ≤ R[j] then
                A[k] ← L[i]
                i ← i + 1
25:         else
                A[k] ← R[j]
                j ← j + 1
            end if
            k ← k + 1
30:     end while
        while i < n1 do
            A[k] ← L[i]
            i ← i + 1
            k ← k + 1
35:     end while
        while j < n2 do
            A[k] ← R[j]
            j ← j + 1
            k ← k + 1
40:     end while
    end procedure
```

2.6. Quick sort

Quick Sort es un algoritmo de ordenamiento que también utiliza la técnica de divide y vencerás, siendo conocido por su eficiencia en la práctica. Funciona seleccionando un elemento de la lista como "pivotez particionando los demás elementos en dos sublistas: aquellos menores que el pivote y aquellos mayores. Luego, se ordenan recursivamente estas sublistas. Su complejidad temporal promedio es O(n log n), aunque en el peor de los casos puede llegar a O(n^2), especialmente si el pivote se elige de manera ineficiente. Sin embargo, Quick Sort es muy eficiente en la mayoría de las implementaciones, es in situ (no requiere espacio adicional significativo) y es ampliamente utilizado en diversas aplicaciones de ordenamiento.

Algorithm 6 Quick Sort

```
    procedure QuickSort(A, low, high)
        if low < high then
            pi ← Partition(A, low, high)
            QuickSort(A, low, pi - 1)
            QuickSort(A, pi + 1, high)
 6:     end if
    end procedure
    procedure Partition(A, low, high)
        pivot ← A[high]
        i ← low − 1
        for j ← low to high − 1 do
12:         if A[j] ≤ pivot then
                i ← i + 1
                temp ← A[i]
                A[i] ← A[j]
                A[j] ← temp
            end if
18:     end for
        temp ← A[i + 1]
        A[i + 1] ← A[high]
        A[high] ← temp
        return i + 1
    end procedure
```

2.7. Selection sort

Selection Sort es un algoritmo de ordenamiento simple y directo que trabaja de manera iterativa. Funciona dividiendo la lista en una parte ordenada y otra desordenada. En cada iteración, el algoritmo busca el elemento más pequeño de la parte desordenada y lo intercambia con el primer elemento de esta sección. Este proceso se repite, reduciendo la parte desordenada en uno, hasta que toda la lista está ordenada. Aunque su complejidad temporal es O(n^2), lo que lo hace menos eficiente para listas grandes, su simplicidad y el hecho de que no requiere espacio adicional lo convierten en una opción viable para listas pequeñas o casi ordenadas.

Algorithm 7 Selection Sort

```
procedure SelectionSort(A)
    n ← length(A)
    for i ← 0 to n − 2 do
        minIndex ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[minIndex] then
7:              minIndex ← j
            end if
        end for
        if minIndex ≠ i then
            temp ← A[i]
            A[i] ← A[minIndex]
            A[minIndex] ← temp
14:     end if
    end for
end procedure
```

3. Metodología

La metodología adoptada para este trabajo de investigación se centra en la comparación de siete algoritmos de búsqueda: Búsqueda Lineal, Búsqueda Binaria, Búsqueda Interpolativa, Búsqueda Exponencial, Búsqueda de Fibonacci, Búsqueda de Ternaria y Búsqueda de Índice. Estos algoritmos se implementarán en tres lenguajes de programación: C++, Python y Java. Se evaluarán sus tiempos de ejecución al buscar elementos en conjuntos de datos de diferentes tamaños, variando de 100 a 100,000 elementos.

3.1. Algoritmos de busqueda

Cada uno de los algoritmos de búsqueda mencionados será analizado en términos de su complejidad temporal y espacial. Se implementarán utilizando estructuras de datos adecuadas para cada lenguaje, y se prestará especial atención a las particularidades de cada implementación, como el manejo de errores y la eficiencia en el uso de recursos. Esta fase busca no solo codificar los algoritmos, sino también comprender sus fundamentos teóricos y prácticos.

3.1.1. Implementacion de los algoritmos

Los algoritmos se implementarán en C++, Python y Java, garantizando que cada versión mantenga una estructura similar para facilitar la comparación. Se utilizarán funciones específicas de cada lenguaje para manejar la entrada y salida de datos, así como la medición del tiempo de ejecución. Los datos de prueba serán generados aleatoriamente para cubrir una variedad de escenarios y asegurar una evaluación justa.

3.1.2. Evaluacion de los algoritmos

La evaluación se realizará midiendo los tiempos de ejecución de cada algoritmo para diferentes tamaños de datos. Se utilizarán métricas como el tiempo promedio, el tiempo máximo y la variabilidad en las ejecuciones para obtener un análisis detallado. Los resultados serán representados gráficamente para facilitar la comparación visual entre los lenguajes y algoritmos, permitiendo conclusiones sobre la eficiencia relativa de cada enfoque en función del tamaño de los datos.

4. Propuesta

4.1. Implementación y Optimización de Algoritmos de Ordenamiento

A continuación, se presentará la implementaciones de los diferentes algoritmos de ordanamiento 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. Codigo en C++.

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstring>
#include <string>
#include <chrono>
using namespace std;

typedef void (*Sorteo)(int[], int);
void Bubble_sort(int array[], int size);
void Counting_sort(int array[],int size);
void heap_sort(int array[],int size);
void heapify(int array[],int n,int i);
void insertion_sort(int array[], int size);
void merge_sort(int array[], int size);
void merge(int array[], int left[], int leftSize, int right[], int
    rightSize);
void quickSort(int array[],int size);
void quickSort(int array[], int low, int high);
void selectionSort(int array[], int size);
int partition(int array[], int low, int high);

void imprimir(double tiempos[][21], int filas,int columnas);
int* LeerArchivo(int rango);

int main(){
  int testing[]={100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000,
    8000, 9000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000,
    90000, 100000};
     const int cant_testing=sizeof(testing)/sizeof(testing[0]);
     double tiempos[7][21];

  Sorteo sortingMethods[] = {
        Bubble_sort,
        Counting_sort,
        heap_sort,
        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
    rightSize) {
180     int i=0,j=0,k=0;
181 
182     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     }
```

```
}
//Imprimir la matriz de tiempos
void imprimir(double tiempos[][21], int filas,int columnas){
  for(int i=0;i<filas;i++){
    for(int j=0;j<columnas;j++){
      cout<<tiempos[i][j]<<" ";
    }
    cout<<endl;
  }
}
int* LeerArchivo(int rango) {
  string ruta="D:/gersael/Trabajo de ADA/Crear_numeros/"+to_string(rango
   ) + ".txt";
    ifstream archivo(ruta);
    if (!archivo.is_open()){
        cerr<<"Error al abrir el archivo: "<<ruta<<endl;
        return nullptr;
    }
    int* numeros=new int[rango];
  string linea;
    int i=0;
    while(getline(archivo, linea) && i<rango) {
        istringstream iss(linea);
        int numero;
        while (iss>>numero && i<rango){
            numeros[i]=numero;
            i++;
        }
    }

    archivo.close();
    return numeros;
}
```

4.1.2. Codigo en Java.

```
package algoritmos;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Arrays;

public class algoritmo{
    @FunctionalInterface
    interface SortingMethod {
        void sort(int[] array);
    }
    public static void main(String[] args){
        int []testing
   ={100,500,1000,2000,3000,4000,5000,6000,7000,8000,9000,10000,20000,30000,40000,5
```

```
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                  long startTime=System.nanoTime();
32                  sortingMethods[j].sort(numeros_copia);
33                  long endTime=System.nanoTime();
34                  long duration=endTime-startTime;
35                  tiempos[j][i]=duration/1_000_000_000.0;
36              }
37          }
38          imprimir(tiempos);
39      }
40      //algoritmos sort
41      //1. bubble sort
42      public static void bubblesort(int[]array){
43          int size=array.length;
44          boolean cambio;
45          for(int i=0;i<size-1;i++){
46              cambio=false;
47              for(int j=0;j<size-1-i;j++){
48                  if(array[j]>array[j+1]){
49                      int aux=array[j];
50                      array[j]=array[j+1];
51                      array[j+1]=aux;
52                      cambio=true;
53                  }
54              }
55              if(!cambio) break;
56          }
57      }
58      //2. counting sort
59      public static void countingsort(int[] array){
60          int max=array[0],min=array[0],range;
61          for(int i=0;i<array.length;i++){
62              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             indice_mayor=hijo_izquierdo;
111         }
```

```
112         if (hijo_derecho<n && array[hijo_derecho] > array[indice_mayor])
    {
113             indice_mayor=hijo_derecho;
114         }
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];
```

```
            array[minIndex]=array[i];
            array[i]=temp;
        }
    }
    //leer archivos .txt
    public static int[] LeerArchivo(int rango) {
        String ruta="D:\\gersael\\Trabajo de ADA\\Crear_numeros"+"\\"+
   String.valueOf(rango)+".txt";
        int [] numeros=new int[rango];
        try (BufferedReader br=new BufferedReader(new FileReader(ruta)))
    {
            String linea;
            int i=0;
            while((linea=br.readLine())!=null && i<rango) {
                String[] numerosComoString=linea.split(" ");
                for(String numStr : numerosComoString){
                    if(i<rango) {
                        numeros[i] = Integer.parseInt(numStr);
                        i++;
                    }
                }
            }
        } catch(IOException e) {
            e.printStackTrace();
        }
        return numeros;
    }
    //imprimir matriz de tiempos
    public static void imprimir(double[][] array){
        for(int i = 0; i <array.length; i++){
            for (int j = 0; j < array[0].length; j++) {
                System.out.print(array[i][j]);
                if(j<array[0].length-1){
                    System.out.print(",");
                }
            }
            System.out.println();
        }
    }
}
```

4.1.3. Codigo en Python.

```
import time
import copy
import numpy as np
def bubblesort(array):
    size=len(array)
    cambio=False

    for i in range(size-1):
```

```
        cambio=False

        for j in range(size-1-i):
            if array[j]>array[j+1]:
                array[j], array[j + 1] = array[j + 1], array[j]
                cambio=True

        if not cambio:
            break
def counting_sort(array):
    mx=max(array)
    mn=min(array)
    rng=mx-mn+1
    conteo=[0]*rng
    salida=[0]*len(array)
    for i in range(len(array)):
        conteo[array[i]-mn]+=1

    for i in range(1,rng):
        conteo[i]+=conteo[i-1]

    for i in range(len(array)-1,-1,-1):
        salida[conteo[array[i]-mn]-1]=array[i]
        conteo[array[i]-mn]-=1

    for i in range(len(array)):
        array[i]=salida[i]
def heap_sort(array):
    n=len(array)
    for i in range(n//2-1,-1,-1):
        heapify(array,n,i)

    for i in range(n-1,0,-1):
        aux=array[0]
        array[0]=array[i]
        array[i]=aux
        heapify(array, i, 0)
def heapify(array,n,i):
    indice_mayor=i
    hijo_derecho=2*i+1
    hijo_izquierdo=2*i+2
    if hijo_izquierdo<n and array[hijo_izquierdo]>array[indice_mayor]:
        indice_mayor=hijo_izquierdo

    if hijo_derecho<n and array[hijo_derecho] > array[indice_mayor]:
        indice_mayor=hijo_derecho

    if  indice_mayor!=i:
        aux=array[i]
        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,
      10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000,
     100000]
134 tiempos=np.zeros((7,len(testing)))
135
136 sorteo_metodos=[
137     bubblesort,
138     counting_sort,
139     heap_sort,
140     insertion_sort,
141     merge_sort,
142     quick_sort,
143     selection_sort
144 ]
145
146 for i in range(len(testing)):
147     numeros=leer_archivo(testing[i])
148     for j in range (len(sorteo_metodos)):
149         numeros_copia=copy.deepcopy(numeros)
150         start_time=time.time()
151         sorteo_metodos[j](numeros_copia)
152         end_time=time.time()
153         duracion=end_time-start_time
154         tiempos[j][i]=duracion
155
156 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 estaran desordenadas para evaluar el rendimiento en esta situacion.

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.

5. Resultados

5.1. Resultado del Paso 1 de la metodología

5.1.1. 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

5.1.2. 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

5.1.3. 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)**
0.0	0.00000000e+00
1	0.00000000e+00
2	0.00000000e+00
3	1.56297684e-02
4	1.02324486e-01
5	2.49690294e-01
6	4.55706596e-01
7	6.90638304e-01
8	9.99198437e-01
9	1.36985087e+00
10	1.76438975e+00
11	2.26423597e+00
12	2.82991290e+00
13	1.12059011e+01
14	2.53226099e+01
15	4.54007187e+01
16	7.09822209e+01
17	1.04037443e+02
18	1.38776112e+02
19	1.82618368e+02
20	2.28714460e+02
21	2.83626488e+02

Datos	**Tiempo (s)**
0.0	0.00000000e+00
1	0.00000000e+00
2	0.00000000e+00
3	0.00000000e+00
4	1.56242847e-02
5	0.00000000e+00
6	0.00000000e+00
7	1.56216621e-02
8	1.56238079e-02
9	3.21872234e-02
10	3.12447548e-02
11	3.12471390e-02
12	4.68714237e-02
13	7.81240463e-02
14	1.24988317e-01
15	1.40601397e-01
16	1.71808720e-01
17	2.11305141e-01
18	2.42140293e-01
19	2.74088144e-01
20	3.07026863e-01

Datos	**Tiempo (s)**
0.0	0.00000000e+00
1	0.00000000e+00
2	0.00000000e+00
3	1.56173706e-02
4	1.56238079e-02
5	0.00000000e+00
6	1.56321526e-02
7	1.63121223e-02
8	1.56230927e-02
9	1.15544796e-02
10	1.56230927e-02
11	1.56235695e-02
12	1.56238079e-02
13	4.68819141e-02
14	9.36980247e-02
15	1.41101837e-01
16	1.80535793e-01
17	2.52150536e-01
18	2.95016289e-01
19	3.79733562e-01
20	4.54482317e-01
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

5.2. Resultado del Paso 2 de la metodología

5.2.1. Resultados Bubble Sort

Cuadro 10: Tiempos de Ejecución0̆0f3n 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

5.2.2. 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

5.2.3. 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

5.2.4. 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

5.2.5. 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

5.2.6. 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

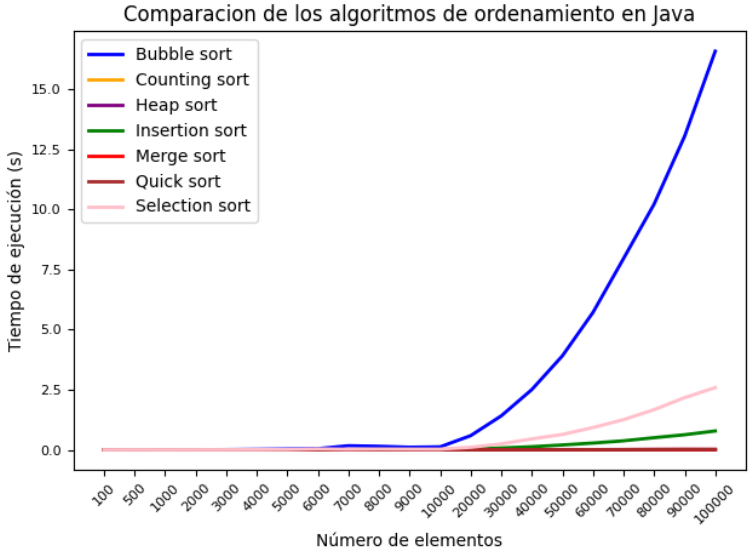
5.2.7. Resultados Selection Sort

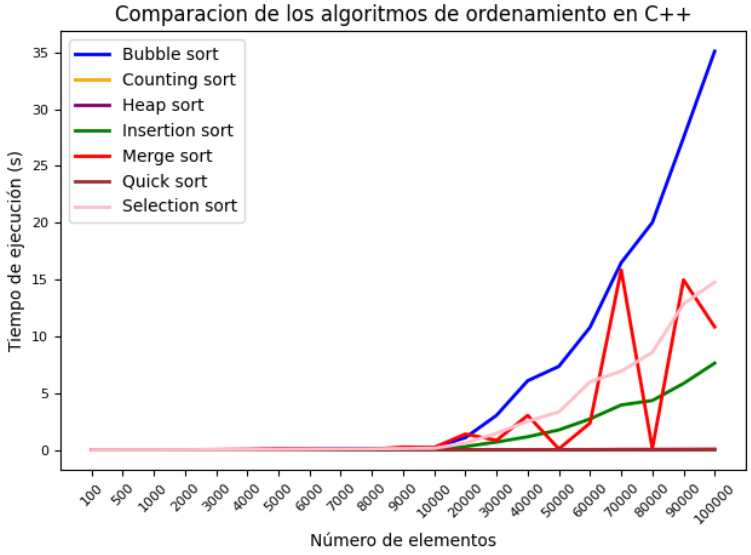
Cuadro 16: Tiempos de Ejecución para Selection Sort

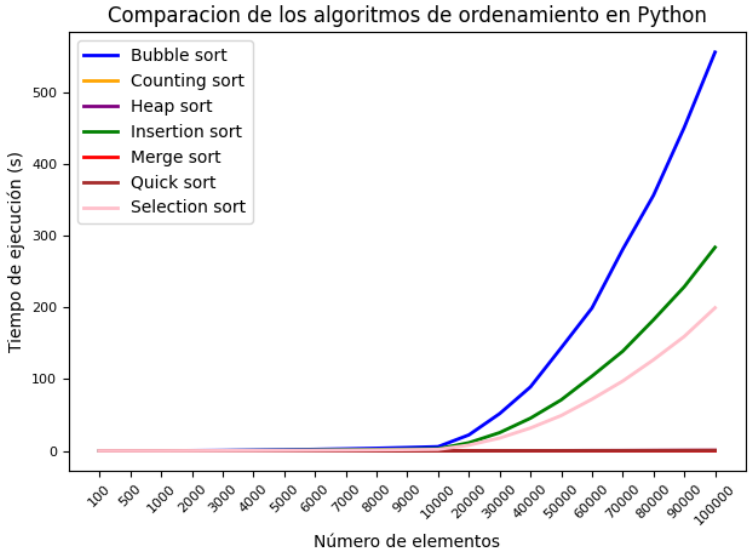
Datos	**Columna 1**	**Columna 2**	**Columna 3**
100	1.032E-4	3.6e-05	1.56655312e-02
500	0.0017503	0.000503	1.56219006e-02
1000	9.688E-4	0.001355	1.56202316e-02
2000	0.0019033	0.005289	7.81700611e-02
3000	0.003021	0.012503	1.87481642e-01
4000	0.005599	0.021077	3.08452368e-01
5000	0.0119343	0.035123	4.97750044e-01
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

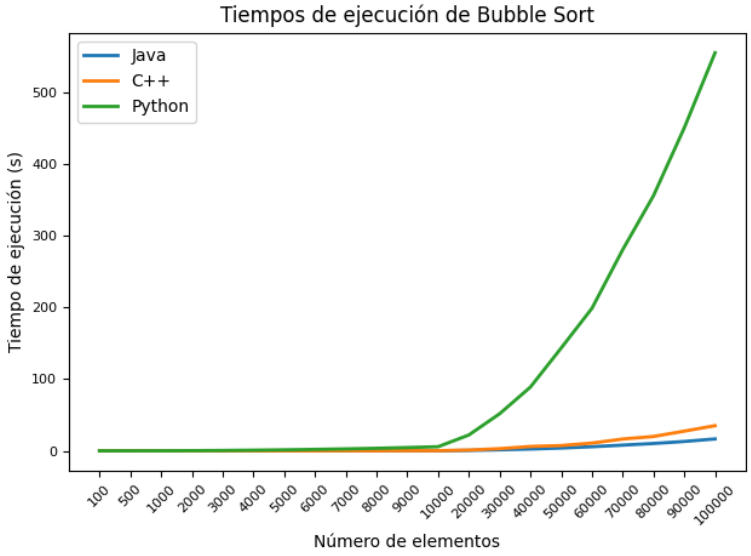
5.3. Resultado del Paso N de la metodología

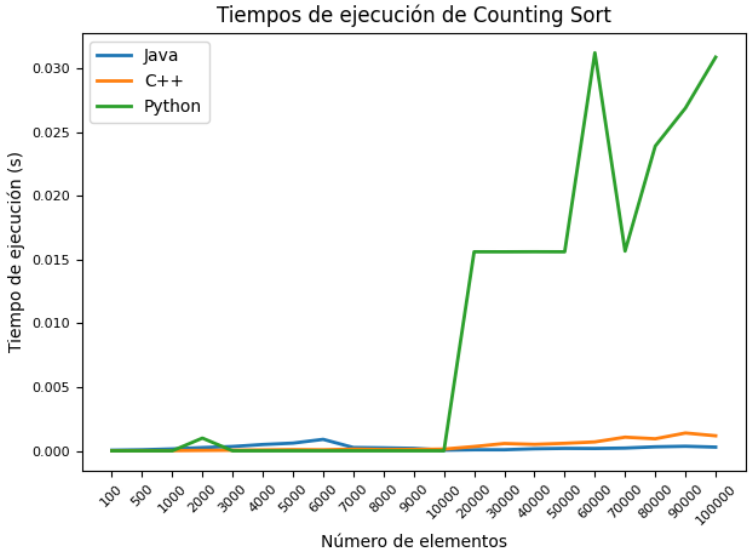
5.3.1. Graficos:

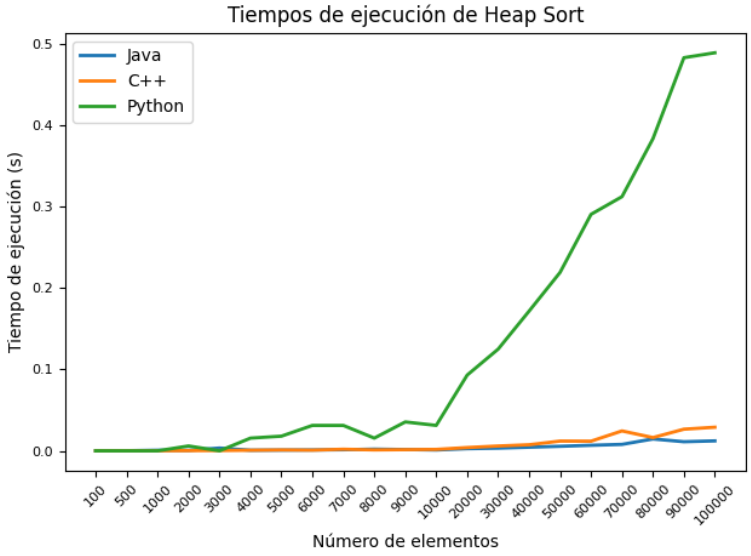


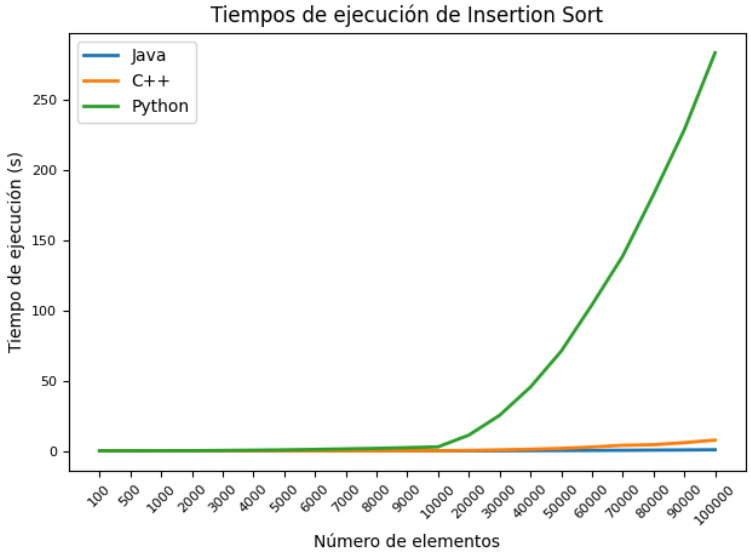


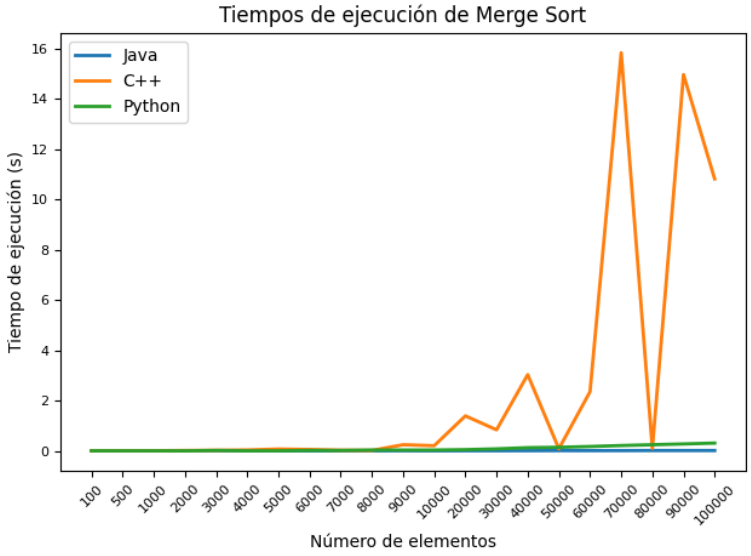

Comparacion de los algoritmos de ordenamiento en Python
Bubble sort
Counting sort
Heap sort
Insertion sort
Merge sort
Quick sort
Selection sort
Tiempo de ejecución (s)
0
100
200
300
400
500
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos

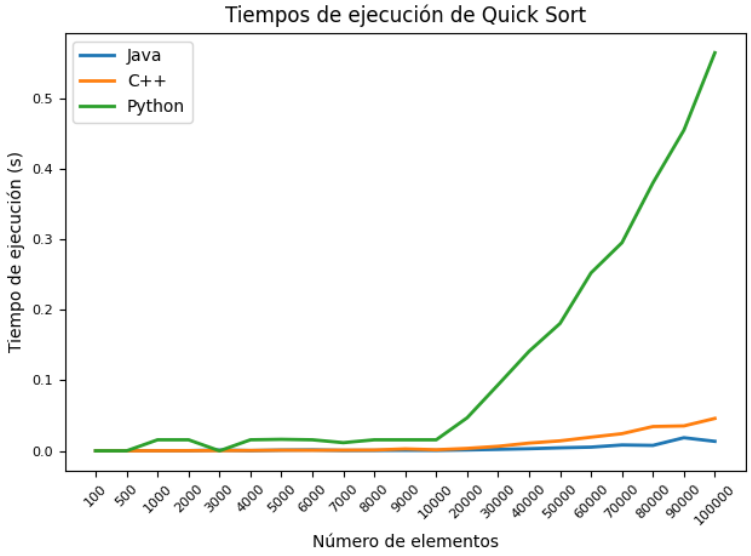

Tiempos de ejecución de Bubble Sort
Java
C++
Python
Tiempo de ejecución (s)
0
100
200
300
400
500
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos

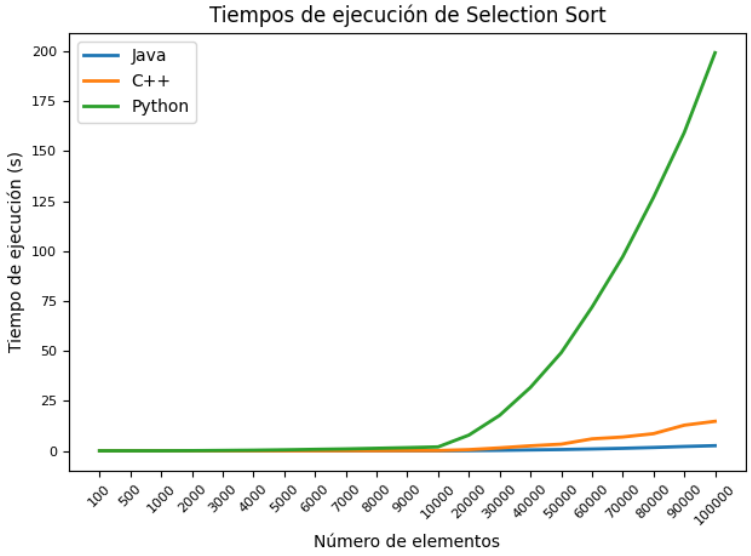

Tiempos de ejecución de Counting Sort
Java
C++
Python
Tiempo de ejecución (s)
0.030
0.025
0.020
0.015
0.010
0.005
0.000
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos


Tiempos de ejecución de Heap Sort
Java
C++
Python
Tiempo de ejecución (s)
0.5
0.4
0.3
0.2
0.1
0.0
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos


Tiempos de ejecución de Insertion Sort
Java
C++
Python
Tiempo de ejecución (s)
0
50
100
150
200
250
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos


Tiempos de ejecución de Merge Sort
Java
C++
Python
Tiempo de ejecución (s)
0
2
4
6
8
10
12
14
16
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos


Tiempos de ejecución de Quick Sort
Java
C++
Python
Tiempo de ejecución (s)
0.0
0.1
0.2
0.3
0.4
0.5
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos


Tiempos de ejecución de Selection Sort
Java
C++
Python
Tiempo de ejecución (s)
0
25
50
75
100
125
150
175
200
100
500
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Número de elementos

6. Conclusiones

[5]

1. Conclusion numero uno : Al analizar los tiempos de ejecución de diferentes algoritmos, se puede observar que la selección adecuada del algoritmo tiene un impacto significativo en el rendimiento general del programa, especialmente con conjuntos de datos grandes.

   En Java, los algoritmos de ordenamiento como Merge Sort, Quick Sort, y Heap Sort, que tienen complejidad O(n log n), demostraron ser significativamente más eficientes que los algoritmos cuadráticos (Bubble, Insertion y Selection) a medida que el tamaño del conjunto de datos aumentó. Para entradas grandes (1,000,000 elementos), los algoritmos cuadráticos mostraron un aumento drástico en el tiempo de ejecución, lo que los hace imprácticos para aplicaciones reales donde se manejan grandes volúmenes de datos. Aunque algunos algoritmos son teóricamente más eficientes, la implementación y optimización del código también juegan un papel crucial en el rendimiento real observado.

   En C++, los algoritmos de ordenamiento como Counting Sort, Heap Sort y Quick sort demostraron ser mas eficientes al emplear menor tiempo de ejecución, a diferencia de algoritmos cuadraticos como Bubble Sort, que incremento su tiempo de ejecucion conforme aumentaban los datos, lo cual lo vuelve menos practico a diferencia de los inicialmente mencionados.

   En Pyhton, los algoritmos de ordenamiento como Quick Sort, Merge Sort y Heap Sort mostraron mayor eficiencia a diferencia de los demás empleados, esto debido a su estructura y funcionamiento. Al igual que en los anteriores, Bubble Sort se encabeza como el algoritmo menos eficiente debido a su estructura cuadratica, empleando mayor tiempo de ejecucion.

2. Conclusion numero dos: En el siguiente apartado, podemos analizar los 7 diferentes algoritmos de ordenamiento, junto a su ejecucion en los 3 programas mas comunmente usados (Pyhton, Java y C++). Analizaremos su tiempo de ejecucion en diferentes tamaños de datos.

   En Bubble Sort, podemos observar como Java es el lenguaje mas efectivo al emplear menor tiempo de ejecucion, caso contrario a Python, cuyas estadisticas incrementan a niveles alarmantes.

   En Counting Sort, si bien se llega a la misma conclusion, podemos observar incrementos y disminuciones en Java y C++, observando como ambos, en cierto punto, pelean por cual programa es mas efectivo, quedando Java como ganador pues resulta mas efectivo con datos de gran magnitud.

   En Heap Sort, se repite el mismo patron que los anteriores algoritmos, siendo Python el lenguaje menos efectivo al, nuevamente, emplear mayor tiempo de ejecucion con-

forme los datos incrementaban. De igual manera con los algoritmos Insertion Sort, Quick Sort y Selection Sort.

El unico algoritmo que no sigue con el anterior patron es el Merge Sort, el cual es menos eficiente en el lenguaje C++, esto se debe porque C++ requiere un manual de gestión de la memoria. Si hay problemas de asignación o liberación de memoria, esto puede afectar el rendimiento. En cambio, Python y Java tienen recolección de basura, lo que puede facilitar el manejo de la memoria.

3. Conclusion numero tres : A partir del análisis de los tiempos de ejecución de diversos algoritmos de ordenamiento en lenguajes como Java, C++ y Python, se pueden extraer conclusiones complementarias que refuerzan la importancia de la selección del algoritmo y su implementación. La tendencia observada en el rendimiento de los algoritmos sugiere que, independientemente del lenguaje, los algoritmos con complejidad O(n log n) tienden a ofrecer un rendimiento superior a medida que aumenta el tamaño de los datos. Esto enfatiza la necesidad de utilizar algoritmos eficientes para aplicaciones que manejan grandes volúmenes de información.

   Aunque ciertos algoritmos son intrínsecamente más eficientes, el comportamiento observado varía entre lenguajes. Por ejemplo, Java mostró un rendimiento consistentemente mejor en varios algoritmos, mientras que Python enfrentó desafíos significativos con algoritmos cuadráticos. Esto resalta la importancia de considerar el contexto del lenguaje al seleccionar un algoritmo. Además, la gestión de memoria y recursos es un factor crítico que afecta el rendimiento, especialmente en C++. La necesidad de una gestión manual puede introducir errores y afectar la eficiencia, lo que no ocurre en lenguajes con recolección de basura como Java y Python. Esto sugiere que la facilidad de manejo de memoria puede ser un criterio importante al elegir un lenguaje para desarrollar aplicaciones que requieran un ordenamiento eficiente.

   La implementación y optimización del código son esenciales para maximizar el rendimiento. Aunque algunos algoritmos son teóricamente más rápidos, una implementación ineficiente puede anular estas ventajas. Por lo tanto, es fundamental no solo seleccionar el algoritmo correcto, sino también optimizar su implementación según las características específicas del lenguaje y del entorno. Finalmente, las conclusiones extraídas son altamente relevantes para desarrolladores e ingenieros de software al diseñar sistemas que requieren procesamiento eficiente de datos. La elección del algoritmo adecuado no solo mejora el rendimiento, sino que también puede influir en la escalabilidad y mantenibilidad del software a largo plazo. En resumen, el análisis destaca la importancia crítica de seleccionar algoritmos eficientes y considerar factores como la gestión de memoria y la optimización del código para lograr un rendimiento óptimo en aplicaciones prácticas.

7. Recomendaciones

1. Recomendación numero uno: Opta por algoritmos eficientes, eligiendo aquellos con complejidad O(n log n), como Merge Sort, Quick Sort o Heap Sort, para manejar grandes volúmenes de datos, ya que ofrecen un rendimiento superior en comparación con los algoritmos cuadráticos. Asegúrese de que su implementación sea eficiente; elimine operaciones innecesarias, utilice estructuras de datos adecuadas y minimice el uso de memoria para mejorar el rendimiento general.

2. Recomendación numero dos: Antes de implementar un algoritmo en producción, realice pruebas exhaustivas con diferentes tamaños de datos y condiciones para identificar cuellos de botella. Además, implemente herramientas de monitoreo para observar el rendimiento del algoritmo mientras está en funcionamiento, lo que le permitirá detectar problemas y optimizar el código de manera continua.

3. Recomendación numero tres: Mantente actualizado con nuevas técnicas, ya que la informática evoluciona constantemente; infórmate sobre nuevas investigaciones y técnicas en algoritmos y optimización. Por último, considere utilizar bibliotecas o frameworks existentes que implementen algoritmos eficientes, lo que puede ahorrarle tiempo y esfuerzo en el desarrollo. Siguiendo estas recomendaciones, podrás mejorar el rendimiento de tus aplicaciones al seleccionar y optimizar adecuadamente los algoritmos de ordenamiento utilizados.

Referencias

[1] Sultan Alanazi, James Goulding, and Derek McAuley. Cross-system recommendation: User-modelling via social media versus self-declared preferences. In *Proceedings of the 27th ACM Conference on Hypertext and Social Media*, HT '16, page 183–188, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Aditya Bhargava. *Grokking Algorithms: An illustrated guide for programmers and other curious people*. Manning Publications, 2016.

[3] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[5] George D. Greenwade. The Comprehensive Tex Archive Network (CTAN). *TUGBoat*, 14(3):342–351, 1993.

[6] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*. CareerMonk Publications, 2011.

[7] Kitti Puritat and Kannikar Intawong. Development of an open source automated library system with book recommedation system for small libraries. In *2020 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT NCON)*, pages 128–132, 2020.

[8] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.

[9] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2008.