

# Informe de Laboratorio 03

## Tema: Análisis de Algoritmos (II)

Nota

Estudiante	Escuela	Asignatura
Rodrigo Infanzón Acosta rinfanzona@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Lenguaje de Programación 3

Laboratorio	Tema	Duración
03	Análisis de Algoritmos (II)	06 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2024 - A	05/04/24	12/04/24

### 1. Actividades a realizar:

- Creación de un contenedor de casos de uso
- Elaborar, programar e implementar el algoritmo de ordenamiento InsertionSort.
- Concebir, codificar e implementar un algoritmo de ordenamiento según preferencia personal. Para este propósito, se explorarán diversos métodos, entre ellos: BubbleSort, QuickSort y MergeSort.
- Desarrollar una implementación en cualquier lenguaje de programación con el propósito de llevar a cabo un análisis comparativo exhaustivo, enfocado en los tiempos de ejecución medidos en "microsegundos".
- Utilizar un programa dedicado en realizar gráficas correspondientes y analizar las tendencias de cada algoritmo de ordenamiento.
- Responder: ¿Qué cantidad de arreglos fue necesaria para encontrar datos confiables?
- Responder: ¿Qué limitaciones presenta el lenguaje de programación utilizado para este tipo de simulaciones?

### 2. Consideraciones generales del docente:

- Enviar satisfactoriamente todas sus implementaciones hacia su repositorio privado lp3-24a provenientes de las plataformas Git y GitHub.
- Describir antecedentes previos que sean necesarios para desarrollar el laboratorio. Las entregadas por el docente y/o las que se buscaron personalmente.

- Elaborar la lista de commits que permitirán culminar el laboratorio, previo a la implementación.
- Explicar porciones de código fuente importantes, trascendentales que permitieron resolver el laboratorio y que reflejen su particularidad única, sólo en trabajos grupales se permite duplicidad
- Mostrar comandos, capturas de pantalla, explicando la forma de replicar y ejecutar el entregable del laboratorio.
- Toda información externa de fuente perteneciente a otro/a autor, debe ser citada y referenciada en el bloque: referencias y referencias bibliográficas.

### 3. Entregables:

- URL al directorio específico del laboratorio en su repositorio GitHub privado donde esté todo el código fuente y otros que sean necesarios. Evitar la presencia de archivos: binarios, objetos, archivos temporales. Incluir archivos de especificación como: packages.json, requirements.txt o README.md.
- No olvidar que el docente debe ser siempre colaborador a su repositorio que debe ser privado. Usuario del docente: @rescobedoulasalle
- Se debe de describir sólo los commits más importantes que marcaron hitos en su trabajo, adjuntando capturas de pantalla, del commit, porciones de código fuente, evidencia de sus ejecuciones y pruebas.
- Siempre se debe explicar las imágenes (código fuente, capturas de pantalla, commits, ejecuciones, pruebas, etc.) con descripciones puntuales pero precisas.
- Agregar la estructura de directorios y archivos de su laboratorio.

### 4. Recursos y herramientas utilizados:

- Sistema operativo utilizado: Windows 10 pro 22H2 de 64 bits SO. 19045.4170.
- Hardware: Ryzen 5 3550H 2.10 GHz, RAM 16 GB DDR4 2400 MHz.
- PseInt
- Git 2.44.0.
- Visual Studio Code 1.88.0.
- Cuenta de GitHub creada con el correo institucional proporcionado por la Universidad La Salle de Arequipa: rinfaazona@ulasalle.du.pe
- Conocimientos básicos en Git.
- Conocimientos intermedios en programación.
- Lenguaje de programación C++.
- Librerías y recursos en C++.
- GNUplot 5.4 patchlevel 8.
- GNU compiler 13.2.

## 5. URL de Repositorio Github:

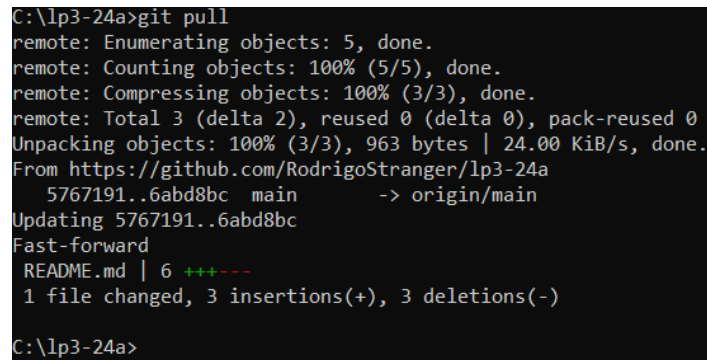
- URL del Repositorio GitHub para clonar o sincronizar:
- <https://github.com/RodrigoStranger/lp3-24a>
- URL para el laboratorio 03 en el Repositorio GitHub:
- <https://github.com/RodrigoStranger/lp3-24a/tree/main/lab03>

## 6. Actividades previas en el repositorio de GitHub

En el desarrollo del tercer laboratorio, desarrollamos el algoritmo de ordenamiento InsertionSort previamente, analizando su lógica y complejidad computacional, en ello, desarrollé mi implementación denominada InsertionSort.cpp, añadiéndolo al área de preparación y empujándolo al servidor. Al llegar a casa, solo tuve que hacer git pull para actualizar mi repositorio local:

Listing 1: Sincronizando el repositorio local

```
cd C:\lp3-24a
git pull
```



```
C:\lp3-24a>git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 963 bytes | 24.00 KiB/s, done.
From https://github.com/RodrigoStranger/lp3-24a
  5767191..6abd8bc  main       -> origin/main
Updating 5767191..6abd8bc
Fast-forward
 README.md | 6 +++---
 1 file changed, 3 insertions(+), 3 deletions(-)

C:\lp3-24a>
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

## 7. Creación de un contenedor de casos de uso:

Para poder crear un contenedor de casos de uso, es decir, crear un vector de vectores en la cual se almacenen los peores casos para analizar en los algoritmos de ordenamiento, necesitamos elaborar una función en la cual pueda crear una instancia `vector<vector<int>>` nombre de la función, y poder usar una variable `n`, en la cual represente el número total de casos:

Listing 2: Crear un contenedor de peores casos

```
// Definimos nuestra funcion para generar un contenedor de peores casos
vector<vector<int>> GenerarYAlmacenarPeoresCasos(int n) {
    vector<vector<int>> peoresCasos;
    for (int i = 1; i <= n; i++) {
        vector<int> peorCaso(i);
        int valor = i;
        for (int j = 0; j < i; j++) {
            peorCaso[j] = valor;
        }
    }
}
```

```
        --valor;  
    }  
    peoresCasos.push_back(peorCaso);  
}  
return peoresCasos;  
}
```

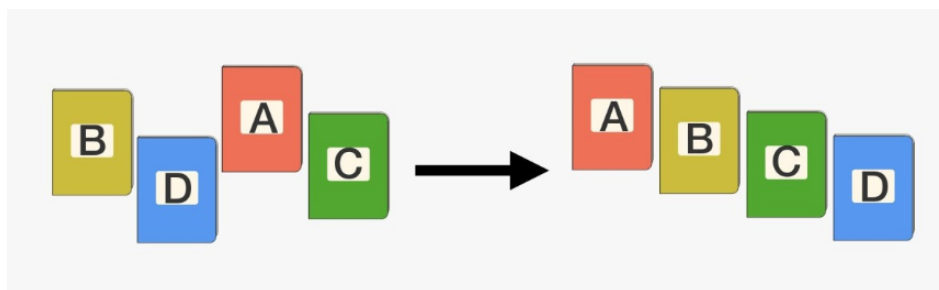
En la función `GenerarYAlmacenarPeoresCasos`, se itera desde 1 hasta  $n$ , donde  $n$  representa el número de peores casos que se desean generar. Dentro de cada iteración, se crea un vector llamado `peorCaso` de tamaño  $i$ , donde  $i$  es el valor actual de la iteración. Luego, se inicializa una variable `valor` con el valor de  $i$ . A continuación, se utiliza un bucle interno `for` con la variable  $j$  para llenar el vector `peorCaso` con números ordenados de manera descendente. En cada iteración de este bucle interno, el valor de `valor` se asigna al elemento en la posición  $j$  del vector `peorCaso`, y luego se decrementa en 1.

Esto garantiza que el vector `peorCaso` esté lleno con números ordenados de manera descendente, comenzando desde  $i$  y decrementando hasta 1. Una vez que el vector `peorCaso` está completamente lleno, se agrega al contenedor `peoresCasos`. Finalmente, cuando se completan todas las iteraciones, la función devuelve el contenedor `peoresCasos`, que contiene todos los peores casos generados.

## 8. Algoritmos de Ordenamiento:

Un algoritmo de ordenamiento es una secuencia de pasos lógicos diseñada para organizar los elementos de una secuencia o conjunto de datos en un orden específico. Este proceso de organización es fundamental en la informática y las ciencias de la computación, ya que permite acceder y manipular los datos de manera más eficiente.

Los algoritmos de ordenamiento pueden variar en complejidad y eficiencia, dependiendo de diversos factores como el tamaño de los datos, la estructura de los mismos y el contexto en el que se utilizan. A grandes rasgos, estos algoritmos pueden dividirse en dos categorías principales: algoritmos de ordenamiento internos y externos:



■ Imagen con propiedad de ©Eduardo Reichel, ©LinkedIn

**Algoritmos de ordenamiento internos:** estos algoritmos operan sobre datos que pueden ser almacenados completamente en la memoria principal del sistema. Son adecuados para conjuntos de datos pequeños o moderados. Ejemplos comunes de estos algoritmos incluyen Bubble Sort, InsertionSort, SelectionSort, QuickSort y MergeSort.

**Algoritmos de ordenamiento externos:** estos algoritmos son diseñados para manejar con-

juntos de datos demasiado grandes para almacenar completamente en la memoria principal. En lugar de eso, los datos se almacenan en dispositivos externos, como discos duros, y se accede a ellos en bloques más pequeños. Ejemplos de estos algoritmos incluyen MergeSort externo y QuickSort externo.

En este laboratorio, nos enfocaremos en realizar los siguientes algoritmos de ordenamiento:

## 8.1. Ordenamiento InsertionSort:

El algoritmo de ordenación por inserción es una técnica que construye la secuencia ordenada de elementos de manera incremental. En cada paso, compara un elemento con los elementos ya ordenados a su izquierda, insertándolo en la posición adecuada. Es un algoritmo simple y eficaz para arreglos pequeños, pero su eficiencia decrece considerablemente en arreglos con un gran número de elementos.

Para lograr esta lógica en el ordenamiento de vectores, mi lógica se influye en estas características:

### 8.1.1. Diseño del algoritmo InsertionSort:

- 1.- Iteración sobre el arreglo:** avanzamos a través del arreglo, comenzando desde el segundo elemento.
- 2.- Comparación e inserción:** en cada iteración, tomamos el elemento actual y lo comparamos con los elementos de la sublista ordenada, comenzando desde el último elemento de la sublista y avanzando hacia la izquierda.
- 3.- Desplazamiento y espacio para la inserción:** si encontramos un elemento en la sublista que es mayor que el elemento actual, desplazamos ese elemento hacia la derecha para hacer espacio para la inserción del elemento actual.
- 4.- Inserción del elemento:** una vez que encontramos el lugar adecuado en la sublista ordenada, insertamos el elemento actual en esa posición.
- 5.- Avance y repetición:** procedemos avanzando al siguiente elemento del arreglo y repetimos el proceso de comparación e inserción en la sublista ordenada.
- 6.- Finalización:** continuamos este proceso hasta que hayamos recorrido todos los elementos del arreglo. Una vez que hemos procesado el último elemento, el arreglo estará completamente ordenado.

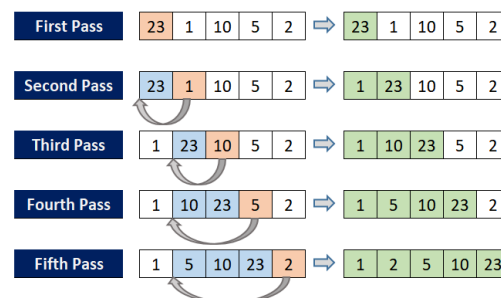


Imagen con propiedad de ©Medium

### 8.1.2. Pseudocódigo del algoritmo InsertionSort:

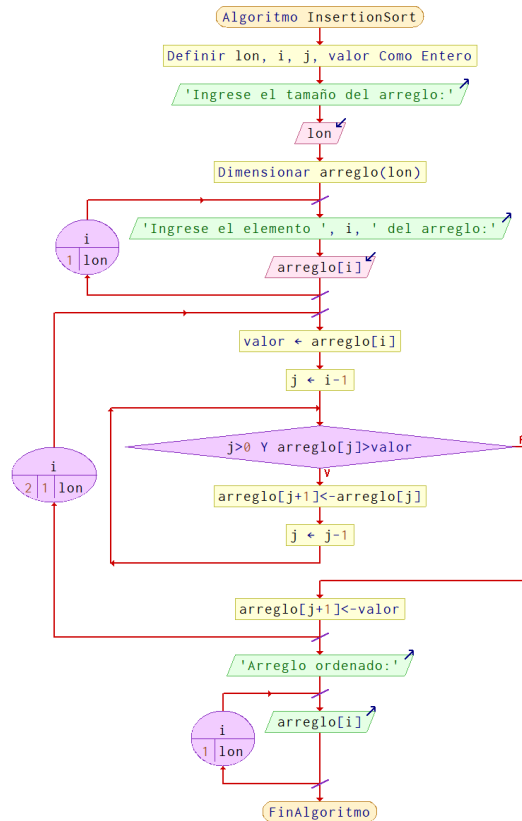


Imagen con propiedad de ©Rodrigo Infanzon Acosta, ©PseInt

### 8.1.3. Implementación del algoritmo InsertionSort:

Previamente, hice la elaboración del InsertionSort en el laboratorio, por lo cual mi primer commit representa el diseño principal de la función, lo podemos ver gracias al comando git log:

#### Commit 1: c88f6a2

En este commit se realizó la implementación de la función "InsertionSort", con una función adicional llamada "PrintArray", en la cual se encarga de imprimir el array inicial y el array ordenado después de su ordenamiento.

```
commit c88f6a2aff482d9e955c82ccf6ae424f990b03e3
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Wed Apr 3 16:04:55 2024 -0500

Agregando la funcion de InsertionSort, con su seguimiento de las iteraciones que hace para ordenar un vector
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

**Output de prueba:** se realiza una prueba para determinar el funcionamiento del algoritmo InsertionSort, en la cual resulta en un éxito.

```
PS C:\Users\riexd> cd 'c:\lp3-24a\lab03\exercises\output'
PS C:\lp3-24a\lab03\exercises\output> .\InsertionSort.exe
Array Inicial: 5 2 4 6 1 3

Ordenando el Array:
Iteracion 1: 2 5 4 6 1 3
Iteracion 2: 2 4 5 6 1 3
Iteracion 3: 2 4 5 6 1 3
Iteracion 4: 1 2 4 5 6 3
Iteracion 5: 1 2 3 4 5 6

Resultado Final: 1 2 3 4 5 6
PS C:\lp3-24a\lab03\exercises\output> |
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Visual Studio Code

### Commit 2: cc937eb

En este commit agrege las librerías necesarias para efectuar tanto el calculo del tiempo, ecepciones y manejo de vectores en general, asu vez, tambien mejore la funcion InsertionSort.

```
C:\lp3-24a\lab03\exercises>git show cc937eb47d7c9bd495664bbf33e02855e98c172f
commit cc937eb47d7c9bd495664bbf33e02855e98c172f
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Tue Apr 9 23:53:07 2024 -0500

    agregando librerias y mejorando la funcion
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 3: 43efb6e

En este commit se añadió la biblioteca chrono para medir el tiempo de ejecución con precisión. Se registró el tiempo antes y después de la ejecución de InsertionSort, y se calculó la duración en microsegundos. Este cambio permite una mejor comprensión del rendimiento de InsertionSort.

```
commit 43efb6e87993c83449a50d7f8e59482d585aa0ca
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Tue Apr 9 23:54:38 2024 -0500

    agregando funcion para calcular el tiempo
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 4: 81e701e

Se han definido dos funciones: PrintArray para imprimir un vector de enteros y PrintArrayContenedor para imprimir un vector de vectores de enteros. Ambas funciones utilizan un bucle for para recorrer los elementos de los vectores y los imprimen en la consola. La función PrintArrayContenedor también imprime el índice del vector de vectores antes de imprimir sus elementos.

```
commit 81e701e196709d1c56918e35b5ad8266a37b914e
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Tue Apr 9 23:55:34 2024 -0500

    agregando funcion para imprimir contenedor y array
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 5: f6acea3

En este commit se agrega una función contenedor que genera y almacena los peores casos para

el algoritmo de ordenamiento, produciendo vectores de vectores de enteros. Para cada tamaño de vector de entrada de 1 a n, crea un vector ordenado en orden descendente, donde el primer elemento tiene el valor i, el segundo i - 1, y así sucesivamente hasta llegar a 1. Luego, estos vectores de peor caso se almacenan en el vector peoresCasos y se devuelve este vector de vectores como resultado.

```
commit f6acea3f888786c681c76505a11753b26465e0c2
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Tue Apr 9 23:57:23 2024 -0500

    agregando funcion para generar un contenedor de vectores y agregando chrono
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 6: ee85f6b

En este commit se desarrolla la función main(), se establece el número de casos a 10000 y se genera un vector de vectores llamado Contenedor2 mediante la función generarYAlmacenarPeoresCasos(). Posteriormente, se abre un archivo llamado "InsertionSort.dat" para escritura y se ordena cada caso del vector de vectores utilizando la función MedirTiempo() para medir el tiempo de ejecución de InsertionSort, guardando los resultados en el archivo. Finalmente, se imprime un mensaje indicando que los datos han sido generados.

```
commit ee85f6b2cbbdf29d21a45a93bd2bf72b8ff2bbc8
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Wed Apr 10 00:00:22 2024 -0500

    agregando el int main donde se genera el .dat para la medicion
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit final 7: 31fa36f

En este commit final se pulen y mejoran detalles del int main, como permitir que el usuario pueda ingresar por consola el número de casos, la optimización para agregar los datos al archivo .dat. Por otro lado, tambien se agrego la función PrintArrayContenedor, dicha función no es importante, pero se puede usar para imprimir y verificar si el algoritmo esta haciendo su trabajo. Por ultimo, se agregó la medición del tiempo de ejecución del ordenamiento de todos los casos establecidos por el usuario.

```
commit 31fa36faa36ff1ffabd20adc51c3afefd0c970e8 (HEAD -> main, origin/main, origin/HEAD)
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Wed Apr 10 21:14:38 2024 -0500

    Mejorando el int main, agregando la funcion de imprimir el cotenedor en caso se necesite, agregando medicion general del tiempo de ejecucion del programa
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

#### 8.1.4. Código fuente del algoritmo InsertionSort:

Listing 3: Implementación InsertionSort

```
1 // Commit 2: cc937eb4f
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 #include <fstream>
```



```
6  #include <chrono>
7  #include <thread>
8
9  using namespace std;
10
11  // Commit 5: f6acea3
12  using namespace std::chrono;
13
14  // Commmmit 1: c88f6a2
15  // Commit 2: cc937eb4
16  // Definimos nuestra funcion InsertionSort
17  void InsertionSort(vector<int>& arr, int n) {
18      for (int i = 1; i < n; i++) {
19          int key = arr[i];
20          int j = i - 1;
21          while (j > -1 && arr[j] > key) {
22              arr[j + 1] = arr[j];
23              j = j - 1;
24          }
25          arr[j + 1] = key;
26      }
27  }
28
29  // Commit 3: 43efb6e
30  // Definimos nuestra funcion en la cual mide el tiempo en microsegundos
31  chrono::microseconds MedirTiempo(vector<int>& arr) {
32      typedef chrono::high_resolution_clock clock;
33      auto inicio = clock::now();
34      InsertionSort(arr, arr.size());
35      auto fin = clock::now();
36      auto duracion = chrono::duration_cast<chrono::microseconds>(fin - inicio);
37      return duracion;
38  }
39
40  // Commit 4: 81e701e
41  // Definimos nuestra funcion para imprimir un vector
42  void PrintArray(const vector<int>& arr, const int longitud) {
43      for (int i = 0; i < longitud; i++) {
44          cout << arr[i] << " ";
45      }
46  }
47
48  // Commit 7: 31fa36f
49  // Definimos nuestra funcion para imprimir un vector de vectores
50  void PrintArrayContenedor(const vector<vector<int>>& arr, const int longitud) {
51      for (int i = 0; i < longitud; ++i) {
52          cout << i + 1 << ": ";
53          PrintArray(arr[i], arr[i].size());
54          cout<<endl;
55      }
56  }
57
58  // Commit 5: f6acea3
59  // Definimos nuestra funcion para generar un contenedor de peores casos
60  vector<vector<int>> GenerarYAlmacenarPeoresCasos(int n) {
61      vector<vector<int>> peoresCasos;
```

```
62     for (int i = 1; i <= n; i++) {
63         vector<int> peorCaso(i);
64         int valor = i;
65         for (int j = 0; j < i; j++) {
66             peorCaso[j] = valor;
67             --valor;
68         }
69         peoresCasos.push_back(peorCaso);
70     }
71     return peoresCasos;
72 }
73
74 // Commit 6: ee85f6b
75 // Commit 7: 31fa36f
76 // Funcion principal de ejecucion
77 int main() {
78     // Creamos nuestro numero de casos
79     int casos;
80     cout<< "Ingrese el numero de casos a tratar: ";
81     cin >> casos;
82
83     // Generamos nuestro vector de vectores llamado Contenedor2 con el numero de casos
84     // establecidos
85     vector<vector<int>> Contenedor2 = GenerarYAlmacenarPeoresCasos(casos);
86
87     // Generando Datos....
88     cout << "Generando datos...." << endl;
89
90     // Abriendo el archivo para escritura
91     ofstream myfile("InsertionSort.dat");
92
93     // Ordenamos cada caso y medimos el tiempo de ejecucion de cada caso y duracion
94     // final del programa, escribiendo en el archivo
95     typedef chrono::high_resolution_clock clock;
96     auto inicio1 = clock::now();
97     for (int i = 0; i < casos; i++) {
98         auto tiempo = MedirTiempo(Contenedor2[i]);
99         myfile << i + 1 << "\t" << tiempo.count() << endl;
100     }
101     auto fin1 = clock::now();
102     auto duracion1 = chrono::duration_cast<chrono::seconds>(fin1 - inicio1);
103     myfile.close();
104
105     // Datos generados....
106     cout << "Datos generados...." << endl;
107     cout << "Duracion final de los ordenamientos: " << duracion1.count() << " segundos";
108     return 0;
109 }
```

#### 8.1.5. Salidas del algoritmo InsertionSort:

- Archivo denominado InsertionSort.dat.
- Respuesta en consola del tiempo de ejecución total del programa.

## 8.2. Algoritmo QuickSort:

Creador: Charles Antony Richard Hoare (Científico Británico en computación)

El algoritmo de ordenamiento QuickSort es un método de (divide y vencerás) que elige un elemento como pivote y particiona la lista alrededor del pivote, de manera que los elementos menores que el pivote estén a su izquierda y los mayores estén a su derecha. Luego, aplica recursivamente QuickSort a las sublistas generadas. Es eficiente para grandes conjuntos de datos y generalmente más rápido que otros algoritmos de ordenamiento.

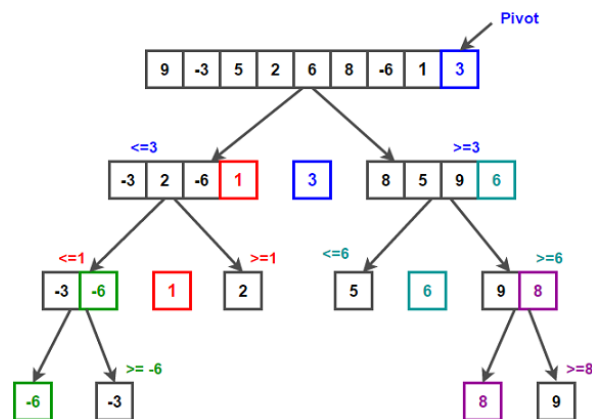


Imagen con propiedad de ©Medium

### 8.2.1. Diseño del algoritmo Quicksort:

- 1.- Selección del pivote:** seleccionamos un elemento de la lista como pivote. Este pivote dividirá la lista en dos subconjuntos: uno con elementos menores o iguales que el pivote y otro con elementos mayores que el pivote.
- 2.- Partición de la lista:** reordenamos los elementos de la lista de manera que los elementos menores que el pivote estén a su izquierda y los mayores a su derecha. Esto se logra moviendo los elementos menores que el pivote a la parte izquierda y los mayores a la parte derecha.
- 3.- Recursión:** aplicamos Quicksort de manera recursiva a las sublistas generadas por la partición. Esto se hace hasta que las sublistas tengan tamaño 0 o 1, en cuyo caso ya estén ordenadas.
- 4.- combinación de las sub listas ordenadas:** Finalmente, combinamos las sub listas ordenadas para formar la lista completa ordenada.

### 8.2.2. Pseudocódigo del algoritmo QuickSort:

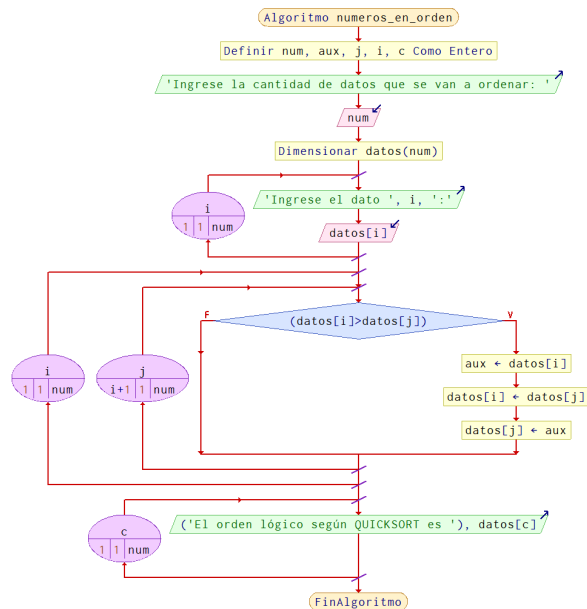


Imagen con propiedad de ©Rodrigo Infanzon Acosta, ©PseInt

### 8.2.3. Implementación del algoritmo QuickSort:

En esta implementación de código, procederé a crear un archivo .cpp denominado QuickSort.cpp, ya que lo desarrollaré en C++:

Listing 4: Creando QuickSort.cpp

```
1 C:\lp3-24a\lab03\exercises
2 echo. > QuickSort.cpp && notepad QuickSort.cpp
```

#### Commit 1: 6e4a179

En el primer commit, se introdujo las librerías necesarias, las funciones Partition y QuickSort. En Partition, se realiza el particionamiento del arreglo, seleccionando un pivote y reorganizando los elementos de manera que los menores que el pivote estén a su izquierda y los mayores a su derecha. Por su parte, QuickSort implementa el algoritmo de ordenamiento QuickSort de manera recursiva, dividiendo el arreglo en subarreglos más pequeños y aplicando el particionamiento hasta que el arreglo esté completamente ordenado.

```
commit 6e4a1791983f897410b6c9325c870e8e980ab566
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Wed Apr 10 22:37:07 2024 -0500

    agregando librerias, añadiendo la funcion partition y el quicksort recursivo
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

#### Commit 2: c302d58

En el segundo commit, se agregaron las funciones MedirTiempo, PrintArray y PrintArrayCon-

tenedor. MedirTiempo se encarga de calcular el tiempo de ejecución del algoritmo QuickSort en microsegundos utilizando la librería chrono. PrintArray y PrintArrayContenedor fueron añadidas para imprimir vectores y vectores de vectores respectivamente, facilitando la visualización de los datos.

```
commit c302d580b24a42d927d98f559a9151c729b5b599
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 22:40:13 2024 -0500

    agregando la funcion para calcular el tiempo de cada ordenamiento, imprimir un array e imprimir un vector de vectores
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 3: b6ae861

En el tercer commit, se implementó la función GenerarYAlmacenarPeoresCasos. Esta función genera casos de prueba específicos diseñados para evaluar el peor rendimiento del algoritmo QuickSort. Cada caso de prueba está compuesto por un vector de tamaño creciente, donde los elementos están ordenados de manera inversa.

```
commit b6ae861d749c8b17557bdb43515b9da6494b32f
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 22:41:23 2024 -0500

    agregando la funcion para crar un vector de vectores
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit final 4: ec995f6

Finalmente, en el cuarto commit se introdujo la función principal main(). Esta función solicita al usuario el número de casos que desea analizar, genera los casos de prueba utilizando la función previamente implementada, ejecuta el algoritmo QuickSort para ordenar cada caso de prueba, mide el tiempo de ejecución de cada uno y lo escribe en un archivo llamado "QuickSort.dat". Además, muestra por pantalla la duración total de la ejecución del programa en segundos para ofrecer al usuario una perspectiva del tiempo total invertido.

```
commit ec995f694c4c465a537d9ad459b4b4917c2d07e9
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 22:48:49 2024 -0500

    agregado el desarrollo de la funcion principal int main, añadiendo las mediciones de tiempo para agregarlas al archivo saliente .dat juntamente con la salida del tiempo total de la ejecucion del programa
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

## 8.2.4. Código fuente del algoritmo QuickSort:

Listing 5: Implementación QuickSort

```
1 // Commit 1: 6e4a179
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 #include <fstream>
6 #include <chrono>
7 #include <thread>
8
```

```
9 using namespace std;
10 using namespace std::chrono;
11
12 // Commit 1: 6e4a179
13 // Definimos nuestra Particion
14 int Partition(vector<int>& arr, int low, int high) {
15     int pivot = arr[high];
16     int i = (low - 1);
17     for(int j = low; j <= high; j++) {
18         if(arr[j]<pivot) {
19             i++;
20             swap(arr[i], arr[j]);
21         }
22     }
23     swap(arr[i+1],arr[high]);
24     return (i+1);
25 }
26
27 // Commit 1: 6e4a179
28 // Definimos nuestro algoritmo QuickSort
29 void QuickSort(vector<int>& arr, int low, int high) {
30     if(low < high) {
31         int pi = Partition(arr, low, high);
32         QuickSort(arr,low, pi - 1);
33         QuickSort(arr, pi + 1, high);
34     }
35 }
36
37 // Commit 2: c302d58
38 // Definimos nuestra funcion en la cual mide el tiempo en microsegundos
39 chrono::microseconds MedirTiempo(vector<int>& arr) {
40     typedef chrono::high_resolution_clock clock;
41     auto inicio = clock::now();
42     QuickSort(arr, 0, arr.size() - 1);
43     auto fin = clock::now();
44     auto duracion = chrono::duration_cast<chrono::microseconds>(fin - inicio);
45     return duracion;
46 }
47
48 // Commit 2: c302d58
49 // Definimos nuestra funcion para imprimir un vector
50 void PrintArray(const vector<int>& arr,const int longitud) {
51     for (int i = 0; i < longitud; i++) {
52         cout <<arr[i] << " ";
53     }
54 }
55
56 // Commit 2: c302d58
57 // Definimos nuestra funcion para imprimir un vector de vectores
58 void PrintArrayContenedor(const vector<vector<int>>& arr,const int longitud) {
59     for (int i = 0; i < longitud; ++i) {
60         cout << i + 1 << ": ";
61         PrintArray(arr[i], arr[i].size());
62         cout<<endl;
63     }
64 }
```

```
65
66 // Commit 3: b6ae861
67 // Definimos nuestra funcion para generar un contenedor de peores casos
68 vector<vector<int>> GenerarYAlmacenarPeoresCasos(int n) {
69     vector<vector<int>> peoresCasos;
70     for (int i = 1; i <= n; i++) {
71         vector<int> peorCaso(i);
72         int valor = i;
73         for (int j = 0; j < i; j++) {
74             peorCaso[j] = valor;
75             --valor;
76         }
77         peoresCasos.push_back(peorCaso);
78     }
79     return peoresCasos;
80 }
81
82 // Commit 4: ec995f6
83 // Funcion principal de ejecucion
84 int main() {
85     // Creamos nuestro numero de casos
86     int casos;
87     cout<< "Ingrese el numero de casos a tratar: ";
88     cin >> casos;
89
90     // Generamos nuestro vector de vectores llamado Contenedor2 con el numero de casos
91     // establecidos
92     vector<vector<int>> Contenedor1 = GenerarYAlmacenarPeoresCasos(casos);
93
94     // Generando Datos....
95     cout << "Generando datos...." << endl;
96
97     // Abriendo el archivo para escritura
98     ofstream myfile("QuickSort.dat");
99
100    // Ordenamos cada caso y medimos el tiempo de ejecucion de cada caso y duracion
101    // final del programa, escribiendo en el archivo
102    typedef chrono::high_resolution_clock clock;
103    auto inicio1 = clock::now();
104    for (int i = 0; i < casos; i++) {
105        auto tiempo = MedirTiempo(Contenedor1[i]);
106        myfile << i + 1 << "\t" << tiempo.count() << endl;
107    }
108    auto fin1 = clock::now();
109    auto duracion1 = chrono::duration_cast<chrono::seconds>(fin1 - inicio1);
110    myfile.close();
111
112    // Datos generados....
113    cout << "Datos generados...." << endl;
114    cout << "Duracion final de los ordenamientos: "<<duracion1.count()<<" segundos";
115    return 0;
116 }
```

### 8.2.5. Salidas del algoritmo QuickSort:

- Archivo denominado QuickSort.dat.
- Respuesta en consola del tiempo de ejecución total del programa.

### 8.3. Algoritmo BubbleSort:

El ordenamiento burbuja es un algoritmo simple de ordenamiento que recorre repetidamente una lista, compara elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso continúa hasta que no se requieren más intercambios, lo que significa que la lista está ordenada. Es uno de los algoritmos de ordenamiento más básicos y fáciles de entender, pero no es eficiente para grandes conjuntos de datos debido a su complejidad cuadrática. Sin embargo, es útil para enseñar conceptos básicos de algoritmos de ordenamiento y para ordenar pequeñas cantidades de datos.

Original:	03	07	11	02	09	01	08	05	10	06	04
Pasada 1:	03	07	02	09	01	08	05	10	06	04	11
Pasada 2:	03	02	07	01	08	05	09	06	04	10	11
Pasada 3:	02	03	01	07	05	08	06	04	09	10	11
Pasada 4:	02	01	03	05	07	06	04	08	09	10	11
Pasada 5:	01	02	03	05	06	04	07	08	09	10	11

Imagen con propiedad de ©Cidecame

#### 8.3.1. Diseño del algoritmo BubbleSort:

**1.- Comparación y reordenamiento de elementos adyacentes:** en el algoritmo de ordenamiento burbuja, se comparan los elementos adyacentes de la lista uno por uno. Si un elemento es mayor que el siguiente, se intercambian de lugar. Este proceso se repite hasta que se completa una pasada completa por la lista sin realizar ningún intercambio, lo que indica que la lista está ordenada.

**2.- Iteración y avance en la lista:** durante cada iteración, el algoritmo burbuja avanza a través de la lista, comparando y reordenando los elementos adyacentes según sea necesario. A medida que avanza, los elementos más grandes "burbujean" hacia el final de la lista, mientras que los elementos más pequeños "flotan" hacia el principio.

**3.- Repetición del proceso:** este proceso de comparación y reordenamiento se repite varias veces hasta que la lista esté completamente ordenada. Cada iteración garantiza que al menos un elemento esté en su posición correcta, lo que gradualmente lleva a una lista ordenada.

**4.- Finalización del algoritmo:** una vez que se completa una pasada completa sin realizar ningún intercambio, se considera que la lista está ordenada y el algoritmo termina. No hay necesidad de combinar sublistas ordenadas, ya que el algoritmo de burbuja opera directamente sobre la lista completa en cada iteración.



### 8.3.2. Pseudocódigo del algoritmo BubbleSort:

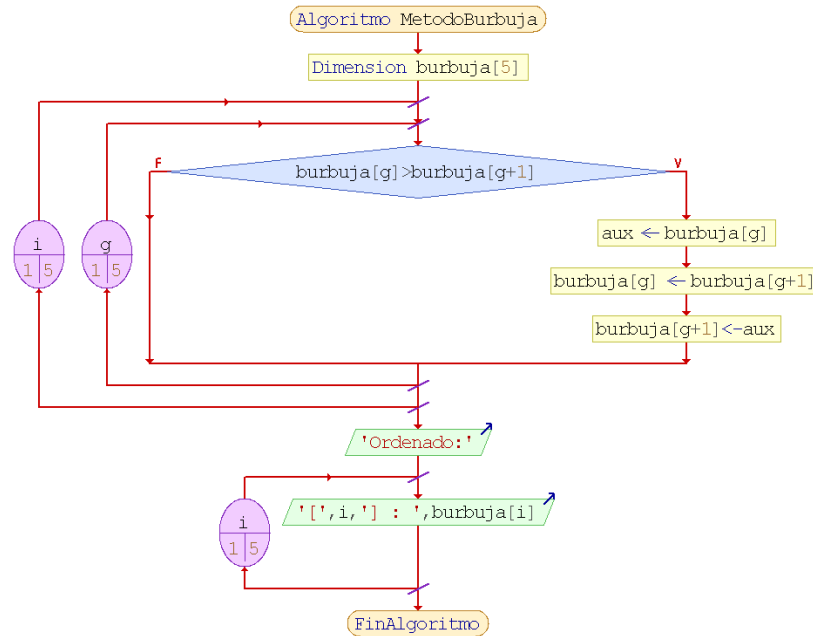


Imagen con propiedad de ©GitHub Wiki

### 8.3.3. Implementación del algoritmo BubbleSort:

En esta implementación de código, procederé a crear un archivo .cpp denominado BubbleSort.cpp, ya que lo desarrollaré en C++:

Listing 6: Creando QuickSort.cpp

```
1 C:\lp3-24a\lab03\exercises
2 echo. > BubbleSort.cpp && notepad BubbleSort.cpp
```

#### Commit 1: 3a59951

En este commit, se implementa las librerías necesarias y el algoritmo de ordenamiento burbuja (BubbleSort). Este algoritmo compara elementos adyacentes y los intercambia si están en el orden incorrecto, repitiendo este proceso hasta que la lista esté completamente ordenada.

```
commit 3a599513a37f68fd4c3d4398df382504dd81368f
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date: Wed Apr 10 23:49:16 2024 -0500

    implementacion de librerias y funcion principal BubbleSort
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

#### Commit 2: b25af76

Se define la función MedirTiempo, que utiliza la librería `chrono` para medir el tiempo de ejecución del algoritmo de ordenamiento burbuja en microsegundos. Esto nos permitirá analizar el rendimiento del algoritmo.

```
commit b25af76157ee7e55a83afaf59f53af70d8058227
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 23:51:17 2024 -0500

    agregando la funcion de medir el tiempo
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 3: e5b362f

Se crean las funciones PrintArray y PrintArrayContenedor para imprimir vectores y vectores de vectores respectivamente. Estas funciones serán útiles para visualizar los datos y resultados durante la ejecución del programa.

```
commit e5b362f12a97b58cb8a83cf54e617d1f7002ba30
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 23:52:52 2024 -0500

    agregando funciones de imprimir un array e imprimir un contenedor
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 4: 52df16a

Se agrega la función GenerarYAlmacenarPeoresCasos, que genera y almacena casos de prueba para el peor escenario de ordenamiento. Estos casos se utilizarán para evaluar el rendimiento del algoritmo en situaciones desfavorables.

```
commit 52df16ad292775c145fcf2c9636191862df50d9d
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 23:53:50 2024 -0500

    agregando funcion de crear un contenedor
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### Commit 5: 973f491

En el main(), se solicita al usuario el número de casos a tratar y se generan los casos de prueba utilizando la función creada anteriormente. Luego, se ordenan estos casos utilizando el algoritmo de burbuja y se mide el tiempo de ejecución de cada uno. Los resultados se escriben en un archivo llamado BubbleSort.dat para su posterior análisis. Finalmente, se muestra la duración total de la ejecución del programa en segundos.

```
commit 973f4912ec51164fa5cd0e6be16ca34e72890ee2
Author: Rodrigo Infanzon Acosta <rinfanzona@ulasalle.edu.pe>
Date:   Wed Apr 10 23:57:01 2024 -0500

    agregando la funcion principal con las implementaciones correspondientes de medir el tiempo de cada ordenamiento, medir el tiempo general y la creacion del .dat
```

Imagen con propiedad de ©Rodrigo Infanzón Acosta, ©Windows

### 8.3.4. Código fuente del algoritmo BubbleSort:

Listing 7: Implementación BubbleSort

```
1  // Commit 1: 3a59951
2  #include <iostream>
3  #include <string>
4  #include <vector>
5  #include <fstream>
6  #include <chrono>
7  #include <thread>
8
9  using namespace std;
10 using namespace std::chrono;
11
12 // Commit 1: 3a59951
13 // Definimos el Ordenamiento Burbuja
14 void BubbleSort(vector<int>& arr, int size) {
15     for (int i = 0; i < size - 1; i++) {
16         for (int j = i + 1; j < size; j++) {
17             if(arr[i] > arr[j]) {
18                 int temp = arr[i];
19                 arr[i] = arr[j];
20                 arr[j] = temp;
21             }
22         }
23     }
24 }
25
26 // Commit 2: b25af76
27 // Definimos nuestra funcion en la cual mide el tiempo en microsegundos
28 chrono::microseconds MedirTiempo(vector<int>& arr) {
29     typedef chrono::high_resolution_clock clock;
30     auto inicio = clock::now();
31     BubbleSort(arr, arr.size());
32     auto fin = clock::now();
33     auto duracion = chrono::duration_cast<chrono::microseconds>(fin - inicio);
34     return duracion;
35 }
36
37 // Commit 3: e5b362f
38 // Definimos nuestra funcion para imprimir un vector
39 void PrintArray(const vector<int>& arr, const int longitud) {
40     for (int i = 0; i < longitud; i++) {
41         cout << arr[i] << " ";
42     }
43 }
44
45 // Commit 3: e5b362f
46 // Definimos nuestra funcion para imprimir un vector de vectores
47 void PrintArrayContenedor(const vector<vector<int>>& arr, const int longitud) {
48     for (int i = 0; i < longitud; ++i) {
49         cout << i + 1 << ": ";
50         PrintArray(arr[i], arr[i].size());
51         cout<<endl;
52     }
```

```
53 }
54
55 // Commit 4: 52df16a
56 // Definimos nuestra funcion para generar un contenedor de peores casos
57 vector<vector<int>> GenerarYAlmacenarPeoresCasos(int n) {
58     vector<vector<int>> peoresCasos;
59     for (int i = 1; i <= n; i++) {
60         vector<int> peorCaso(i);
61         int valor = i;
62         for (int j = 0; j < i; j++) {
63             peorCaso[j] = valor;
64             --valor;
65         }
66         peoresCasos.push_back(peorCaso);
67     }
68     return peoresCasos;
69 }
70
71 // Commit 5: 973f491
72 // Funcion principal
73 int main() {
74     // Creamos nuestro numero de casos
75     int casos;
76     cout<< "Ingrese el numero de casos a tratar: ";
77     cin >> casos;
78
79     // Generamos nuestro vector de vectores llamado Contenedor con el numero de casos
80     // establecidos
81     vector<vector<int>> Contenedor = GenerarYAlmacenarPeoresCasos(casos);
82
83     // Generando Datos....
84     cout << "Generando datos...." << endl;
85
86     // Abriendo el archivo para escritura
87     ofstream myfile("BubbleSort.dat");
88
89     // Ordenamos cada caso y medimos el tiempo de ejecucion de cada caso y duracion
90     // final del programa, escribiendo en el archivo
91     typedef chrono::high_resolution_clock clock;
92     auto inicio1 = clock::now();
93     for (int i = 0; i < casos; i++) {
94         auto tiempo = MedirTiempo(Contenedor[i]);
95         myfile << i + 1 << "\t" << tiempo.count() << endl;
96     }
97     auto fin1 = clock::now();
98     auto duracion1 = chrono::duration_cast<chrono::seconds>(fin1 - inicio1);
99     myfile.close();
100
101     // Datos generados....
102     cout << "Datos generados...." << endl;
103     cout << "Duracion final de los ordenamientos: "<<duracion1.count()<<" segundos";
104     return 0;
105 }
```

### 8.3.5. Salidas algoritmo BubbleSort:

- Archivo denominado BubbleSort.dat.
- Respuesta en consola del tiempo de ejecución total del programa.

## 9. Gnuplot

Es una herramienta de trazado de gráficos que permite generar gráficos bidimensionales y tridimensionales de funciones matemáticas y conjuntos de datos. Utilizando comandos simples y scripts, Gnuplot puede producir una amplia variedad de gráficos, incluyendo gráficos de dispersión, gráficos de líneas, gráficos de barras, histogramas, superficies y más.

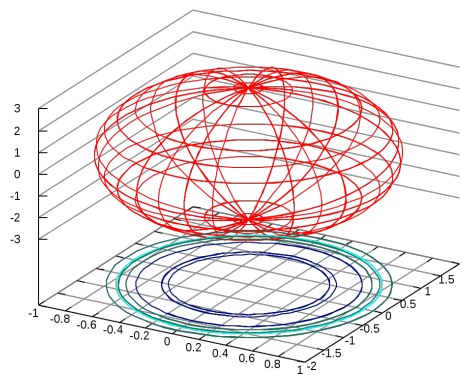


Imagen con propiedad de ©Wikipedia

### 9.1. Comandos comunes en gnuplot:

- 1.- **plot:** Este comando se utiliza para trazar gráficos. Por ejemplo, `plot sin(x)` traza el gráfico de la función seno.
- 2.- **set:** este comando se utiliza para configurar diversas opciones del gráfico, como el rango de los ejes, los títulos de los ejes, los estilos de línea, los colores, etc. Por ejemplo, `set xlabel Eje X` establece la etiqueta del eje X.
- 3.- **unset:** este comando se utiliza para desactivar opciones previamente configuradas. Por ejemplo, `unset key` desactiva la leyenda del gráfico.
- 4.- **splot:** similar al comando `plot`, pero se utiliza para trazar gráficos tridimensionales.
- 5.- **replot:** este comando se utiliza para volver a trazar el gráfico actual. Es útil cuando se realizan cambios en el gráfico y se desea actualizar la visualización.
- 6.- **pause:** este comando pausa la ejecución del script durante un número específico de segundos. Es útil para controlar la velocidad de la visualización.
- 7.- **help:** este comando muestra la documentación de Gnuplot. Por ejemplo, `help plot` muestra información sobre el comando `plot`.

## 9.2. Uso de gnuplot en nuestro laboratorio:

En este laboratorio, utilizaremos Gnuplot para generar gráficos y visualizar datos de experimentos y simulaciones. Gnuplot nos permite representar de forma clara y precisa los resultados obtenidos, lo que facilita el análisis y la interpretación de los datos en los tiempos de los algoritmos de ordenamiento.

Para utilizar Gnuplot en nuestro laboratorio, seguimos los siguientes pasos:

- 1.- Instalar gnuplot en la computadora local.
- 2.- Preparación de los datos .dat.
- 3.- Ejecutar los comandos de gnuplot para la elaboración de gráficos 2D.
- 4.- Analizar e interpretar.

### 9.2.1. Instalar gnuplot en la computadora local:

Para Windows, podemos descargar el instalador ejecutable de la página oficial de gnuplot: <https://sourceforge.net/projects/gnuplot/>. Una vez descargado, ejecutamos el instalador y seguimos las instrucciones en pantalla para completar la instalación.

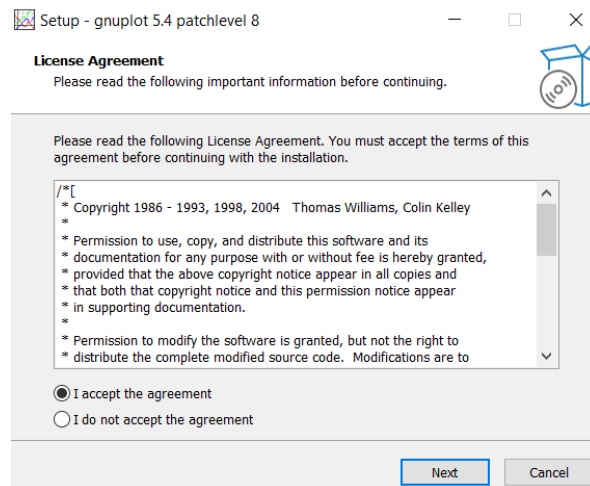


Imagen con propiedad de ©gnuplot

### 9.2.2. Preparación de los datos .dat.

Para preparar los datos en formato .dat que necesitamos para realizar las gráficas con Gnuplot, debemos ejecutar nuestros programas en C++ que contienen los algoritmos de ordenamiento Insertion Sort, QuickSort y BubbleSort de la siguiente manera:

**InsertionSort:**

Listing 8: Generar InsertionSort.dat

```
1 C:\lp3-24a\lab03\exercises
2 g++ -o InsertionSort.exe InsertionSort.cpp
3 InsertionSort.exe
```

### QuickSort:

Listing 9: Generar QuickSort.dat

```
1 C:\lp3-24a\lab03\exercises
2 g++ -o QuickSort.exe QuickSort.cpp
3 QuickSort.exe
```

### BubbleSort:

Listing 10: Generar BubbleSort.dat

```
1 C:\lp3-24a\lab03\exercises
2 g++ -o BubbleSort.exe BubbleSort.cpp
3 BubbleSort.exe
```

Una vez esperado los tiempos de ejecución, cada uno de los programas nos arrojan los siguientes resultados, para  $n = 10000$ :

### InsertionSort:

Listing 11: Tiempo de ejecución en InsertionSort.dat

```
1 Duracion final de los ordenamientos: 1569 segundos
```

### QuickSort:

Listing 12: Tiempo de ejecución en QuickSort.dat

```
1 Duracion final de los ordenamientos: 978 segundos
```

### BubbleSort:

Listing 13: Tiempo de ejecución en BubbleSort.dat

```
1 Duracion final de los ordenamientos: 2089 segundos
```

Podemos observar que el algoritmo QuickSort tuvo el menor tiempo de ejecución, seguido por InsertionSort, y finalmente BubbleSort con el mayor tiempo de ejecución. Esto sugiere que QuickSort es el más eficiente en términos de tiempo de ejecución para ordenar un conjunto de datos de tamaño 10000. Por otro lado, BubbleSort demostró ser el menos eficiente entre los tres algoritmos para este tamaño de datos.

### 9.2.3. Ejecutar los comandos de gnuplot para la elaboracion de graficos 2D:

Una vez obtenido los .dat de cada ejecución de programa correspondiente, podemos utilizar el gnuplot para graficar e interpretar los gráficos correspondientes:

Listing 14: Cargar los datos a gnu plot

```
1 Terminal type is now 'qt'
2 gnuplot> cd 'C:\lp3-24a\lab03\dat'
3 gnuplot> set title "Análisis y gnuplot> comportamiento de algoritmos de ordenacion"
4 gnuplot> set xlabel "Tamaño del vector (n)"
5 gnuplot> set ylabel "Tiempo de ordenamiento (microsegundos)"
```

```
6  gnuplot> plot "BubbleSort.dat" with lines
7  gnuplot> replot "InsertionSort.dat" with lines
8  gnuplot> replot "QuickSort.dat" with lines
```

Salida del programa gnuplot:

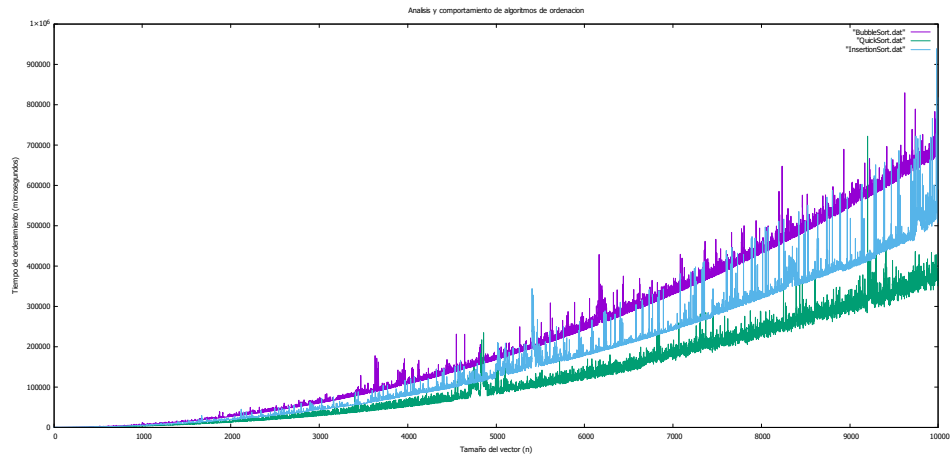


Imagen con propiedad de ©gnuplot

#### 9.2.4. Analizar e interpretar los datos obtenidos:

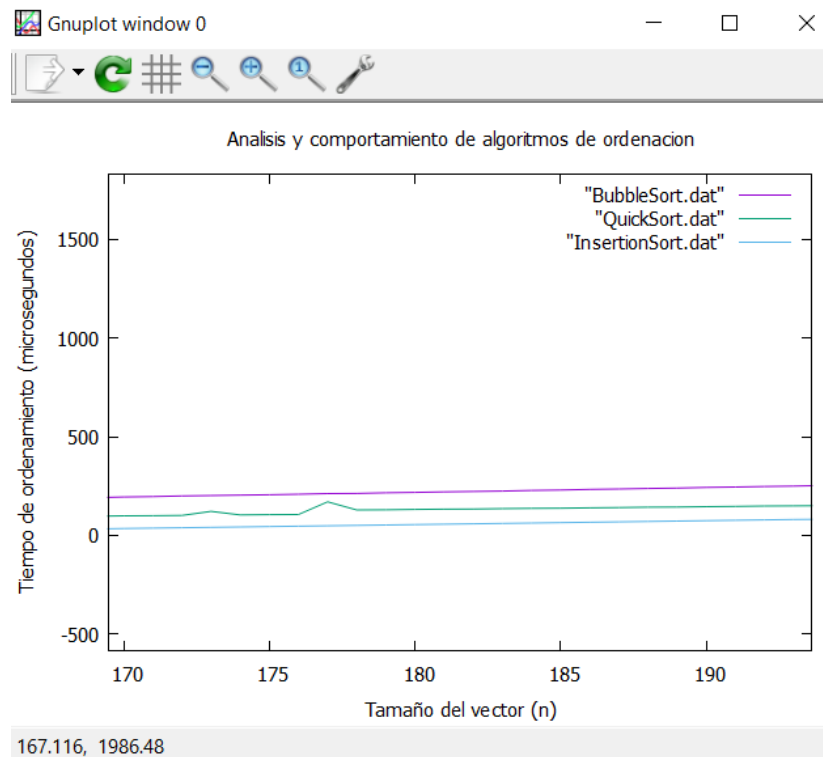


Imagen con propiedad de ©gnuplot



El análisis revela que para conjuntos de datos pequeños, Insertion Sort supera a Quick Sort en términos de velocidad. Esto puede atribuirse a que Insertion Sort tiene una complejidad temporal más baja para conjuntos pequeños, debido a su eficiencia en el manejo de datos de tamaño reducido.

Sin embargo, a medida que el tamaño del conjunto de datos aumenta, Quick Sort toma la delantera. Esto se debe a que Quick Sort tiene una complejidad temporal promedio de  $O(n \log n)$ , lo que lo hace más eficiente para conjuntos de datos más grandes. En resumen, la elección del algoritmo de ordenamiento depende del tamaño del conjunto de datos: Insertion Sort para conjuntos pequeños y Quick Sort para conjuntos más grandes, optimizando así el rendimiento en diferentes escalas de datos.

Por otro lado, el algoritmo BubbleSort en cualquier análisis de caso, es el más ineficiente. Esto radica en su enfoque de comparación y intercambio directo entre elementos adyacentes.

## 10. ¿Qué cantidad de arreglos fue necesaria para encontrar datos confiables?

Seleccionar 10,000 datos para obtener resultados confiables es una decisión que considero aceptable. Como estudiante de ingeniería de software, entiendo que para evaluar adecuadamente el rendimiento de algoritmos como QuickSort, InsertionSort y BubbleSort, es esencial trabajar con conjuntos de datos lo suficientemente grandes como para representar una variedad de casos posibles.

Con 10,000 datos, podemos explorar diferentes tamaños de conjuntos, desde pequeños hasta bastante grandes, lo que nos brinda una visión más completa de cómo se comportan los algoritmos en una variedad de situaciones.

Además, al trabajar con un conjunto de datos grande, podemos reducir el impacto de las fluctuaciones aleatorias y obtener resultados más confiables y consistentes en nuestros análisis.

## 11. ¿Qué limitaciones presenta el lenguaje de programación utilizado para este tipo de simulaciones?

Al emplear C++ para medir el tiempo de algoritmos de ordenación, se encuentran diversas limitaciones. Una de las principales es la precisión de las mediciones temporales. Aunque C++ proporciona mecanismos para medir el tiempo de ejecución, la exactitud puede verse comprometida por la carga del sistema y la resolución del reloj del sistema.

Además, la ausencia de un entorno gráfico integrado dificulta la visualización y análisis eficientes de los resultados temporales, lo que podría requerir soluciones adicionales para representar y comprender los datos recopilados.

La gestión manual de la memoria y la complejidad inherente del lenguaje también pueden impactar en la fiabilidad de las mediciones, ya que errores en la gestión de la memoria o en la implementación del código de medición podrían generar resultados inexactos.

Esto resalta la importancia de una cuidadosa implementación y análisis de los procedimientos de medición de tiempo en C++.

## 12. Estructura del laboratorio 03

```
1 |--- lab03
2 |   |--- data [DIRECTORY]
3 |       |--- InsertionSort.dat
4 |       |--- QuickSort.dat
5 |       |--- BubbleSort.dat
6 |   |--- report [DIRECTORY]
7 |       |--- report03.pdf
8 |       |--- report03.zip
9 |   |--- exercises [DIRECTORY]
10 |       |--- InsertionSort.cpp
11 |       |--- QuickSort.cpp
12 |       |--- BubbleSort.cpp
13 3 directories, 8 files
```

## 13. Referencias:

- [https://www.onlinegdb.com/online\\_c++\\_compiler](https://www.onlinegdb.com/online_c++_compiler)
- <https://sourceforge.net/projects/gnuplot/>
- <https://github.com/>
- <https://pseint.sourceforge.net/>
- <https://www.wikipedia.org/>

## 14. Referencias Bibliográficas:

1. GeeksforGeeks. (2020). Bubble Sort – Data Structure and Algorithm Tutorials. Recuperado de: <https://www.geeksforgeeks.org/bubble-sort/>
2. GeeksforGeeks. (2020). QuickSort – Data Structure and Algorithm Tutorials. Recuperado de: <https://www.geeksforgeeks.org/quick-sort/>
3. GeeksforGeeks. (2020). Orden de inserción – Estructura de Datos y Tutoriales de Algoritmos. Recuperado de: <https://www.geeksforgeeks.org/insertion-sort/>
4. Checa, B. (2018). Cómo usar la librería Chrono en C++. OpenWebinars. Recuperado de: <https://openwebinars.net/blog/como-usar-la-libreria-chrono-en-c/>

## 15. Calificación del docente:

Tabla 1: Rúbrica para contenido del Informe y evidencias

Contenido y demostración		Puntos	Checklist	Estudiante	Profesor
<b>1. GitHub</b>	El repositorio se pudo clonar y se evidencia la estructura adecuada para revisar los entregables. (Se descontará puntos por error o observación)	4	X	4	
<b>2. Commits</b>	Hay porciones de código fuente asociado a los commits planificados con explicaciones detalladas. (El profesor puede preguntar para refrendar calificación).	4	X	4	
<b>3. Ejecución</b>	Se incluyen comandos para ejecuciones y pruebas del código fuente explicadas gradualmente que permitirían replicar el proyecto. (Se descontará puntos por cada omisión)	4	X	4	
<b>4. Pregunta</b>	Se responde con completitud a la pregunta formulada en la tarea. (El profesor puede preguntar para refrendar calificación).	2	X	2	
<b>5. Ortografía</b>	El documento no muestra errores ortográficos. (Se descontará puntos por error encontrado)	2	X	2	
<b>6. Madurez</b>	El Informe muestra de manera general una evolución de la madurez del código fuente con explicaciones puntuales pero precisas, agregando diagramas generados a partir del código fuente y refleja un acabado impecable. (El profesor puede preguntar para refrendar calificación).	4	X	4	
<b>Total</b>		20		20	