LABORATORIO 1

Algoritmos y Costo Computacional

Docente: Rolando Jesús Cardenas Talavera

Alumno: Joao Franco Emanuel Chávez
Salas

Actividad

1. Utilizando los archivos adjuntos (DataGen1, DataGen05, DataGen025), utilice los datos para las pruebas de los algoritmos de ordenamiento. Tenga en cuenta la cantidad de datos de cada uno.
2. Implemente los siguientes algoritmos:
   - Bubble sort
   - Heap sort
   - Insertion sort
   - Selection sort
   - Shell sort
   - Merge sort
   - Quick sort
3. Analizar la complejidad computacional de cada uno.
4. Evaluar y comparar sus algoritmos usando los archivos de datos y elabore una(s) gráfica(s) comparativa(s). De utilizar c++, mida el tiempo de ejecucion con la funcion std::chrono::high_resolution_clock::now();

6 Entregables

Al finalizar el estudiante debera:

- Elaborar un documento, en donde se registre los algoritmos elaborados, el analisis realizado y las graficas elaborados
- Deberá de incluir el codigo en formato de texto (no coloque imágenes de los códigos empleados)
- Deberan de subir a la plataforma Classroom el documento elaborado en formato PDF (se recomienda el uso de LaTeX) y los codigos elaborados.
- IMPORTANTE En caso de copia o plagio o similares todos los alumnos implicados tendran sancion en toda la evaluacion del curso.

Implementación de los algoritmos

Bubble Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;

unsigned t0, t1;
void bubbleSort(vector<double>& arr);
void muestra_vector(const vector<double>&);

int main()
{
   ifstream fich("DataGen025.txt");
   if (!fich.is_open())
   {
      cout << "Error al abrir ejemplo.dat\n";
      exit(EXIT_FAILURE);
   }
   t0=clock();
   double valor;
   vector<double> datos;
   while (fich >> valor)
      datos.push_back(valor);
   cout<<"Mostrar vector desordenado: "<<endl;
   muestra_vector(datos);
   cout<<"Ordenar vector"<<endl;
   bubbleSort(datos);
   cout<<"Mostrar vector ordenado: "<<endl;
   muestra_vector(datos);
   t1 = clock();
   double time = (double(t1-t0)/CLOCKS_PER_SEC);
   cout << "Execution Time: " << time << endl;
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

void bubbleSort(vector<double>& arr) {
   bool swapped;
   for (size_t i = 0; i < arr.size() - 1; ++i) {
      swapped = false;
      for (size_t j = 0; j < arr.size() - i - 1; ++j) {
         if (arr[j] > arr[j + 1]) {
            swap(arr[j], arr[j + 1]);
            swapped = true;
         }
```

```
        }
        if (!swapped) // Si no hubo cambios, el arreglo ya está ordenado
        {
            break;
        }
    }
}
```

Heap Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>
#include <algorithm>

using namespace std;

unsigned t0, t1;

void heapify(vector<double>& arr, int n, int i);
void heapSort(vector<double>& arr);
void muestra_vector(const vector<double>&);

int main()
{
    ifstream fich("DataGen025.txt");
    if (!fich.is_open())
    {
        cout << "Error al abrir ejemplo.dat\n";
        exit(EXIT_FAILURE);
    }
    t0=clock();
    double valor;
    vector<double> datos;
    while (fich >> valor)
        datos.push_back(valor);
    cout<<"Mostrar vector desordenado: "<<endl;
    muestra_vector(datos);
    cout<<"Ordenar vector"<<endl;
    heapSort(datos);
    cout<<"Mostrar vector ordenado: "<<endl;
    muestra_vector(datos);
    t1 = clock();
    double time = (double(t1-t0)/CLOCKS_PER_SEC);
    cout << "Execution Time: " << time << endl;
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

void heapify(vector<double>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
```

```
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<double>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; --i) {
        heapify(arr, n, i);
    }
    for (int i = n - 1; i > 0; --i) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Insertion Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;

unsigned t0, t1;
void insertionSort(vector<double>& arr);
void muestra_vector(const vector<double>&);

int main()
{
    ifstream fich("DataGen025.txt");
    if (!fich.is_open())
    {
        cout << "Error al abrir ejemplo.dat\n";
        exit(EXIT_FAILURE);
    }
    t0=clock();
    double valor;
    vector<double> datos;
    while (fich >> valor)
        datos.push_back(valor);
    cout<<"Mostrar vector desordenado: "<<endl;
    muestra_vector(datos);
    cout<<"Ordenar vector"<<endl;
    insertionSort(datos);
    cout<<"Mostrar vector ordenado: "<<endl;
    muestra_vector(datos);
    t1 = clock();
```

```
    double time = (double(t1-t0)/CLOCKS_PER_SEC);
    cout << "Execution Time: " << time << endl;
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

void insertionSort(vector<double>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        double key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

Selection Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;

unsigned t0, t1;
void insertionSort(vector<double>& arr);
void muestra_vector(const vector<double>&);

int main()
{
    ifstream fich("DataGen025.txt");
    if (!fich.is_open())
    {
        cout << "Error al abrir ejemplo.dat\n";
        exit(EXIT_FAILURE);
    }
    t0=clock();
    double valor;
    vector<double> datos;
    while (fich >> valor)
        datos.push_back(valor);
    cout<<"Mostrar vector desordenado: "<<endl;
    muestra_vector(datos);
    cout<<"Ordenar vector"<<endl;
    insertionSort(datos);
    cout<<"Mostrar vector ordenado: "<<endl;
    muestra_vector(datos);
    t1 = clock();
    double time = (double(t1-t0)/CLOCKS_PER_SEC);
```

```
    cout << "Execution Time: " << time << endl;
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

void insertionSort(vector<double>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        double key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = key;
    }
}
```

Shell Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;

unsigned t0, t1;
void shellSort(vector<double>& arr);
void muestra_vector(const vector<double>&);

int main()
{
    ifstream fich("DataGen025.txt");
    if (!fich.is_open())
    {
        cout << "Error al abrir ejemplo.dat\n";
        exit(EXIT_FAILURE);
    }
    t0=clock();
    double valor;
    vector<double> datos;
    while (fich >> valor)
        datos.push_back(valor);
    cout<<"Mostrar vector desordenado: "<<endl;
    muestra_vector(datos);
    cout<<"Ordenar vector"<<endl;
    shellSort(datos);
    cout<<"Mostrar vector ordenado: "<<endl;
    muestra_vector(datos);
    t1 = clock();
    double time = (double(t1-t0)/CLOCKS_PER_SEC);
    cout << "Execution Time: " << time << endl;
```

```
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

void shellSort(vector<double>& arr) {
    int n = arr.size();

    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; ++i) {
            double temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

Merge Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;

unsigned t0, t1;
void merge(vector<double>& arr, int left, int mid, int right);
void mergeSort(vector<double>& arr, int left, int right);
void muestra_vector(const vector<double>&);

int main()
{
    ifstream fich("DataGen025.txt");
    if (!fich.is_open())
    {
        cout << "Error al abrir ejemplo.dat\n";
        exit(EXIT_FAILURE);
    }
    t0=clock();
    double valor;
    vector<double> datos;
    while (fich >> valor)
        datos.push_back(valor);
    cout<<"Mostrar vector desordenado: "<<endl;
    muestra_vector(datos);
    cout<<"Ordenar vector"<<endl;
    mergeSort(datos, 0, datos.size()-1);
    cout<<"Mostrar vector ordenado: "<<endl;
    muestra_vector(datos);
    t1 = clock();
```

```
    double time = (double(t1-t0)/CLOCKS_PER_SEC);
    cout << "Execution Time: " << time << endl;
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

void merge(vector<double>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<double> L(n1);
    vector<double> R(n2);

    for (int i = 0; i < n1; ++i)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            ++i;
        } else {
            arr[k] = R[j];
            ++j;
        }
        ++k;
    }

    while (i < n1) {
        arr[k] = L[i];
        ++i;
        ++k;
    }

    while (j < n2) {
        arr[k] = R[j];
        ++j;
        ++k;
    }
}

void mergeSort(vector<double>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Quick Sort

```
#include <iostream>
#include <fstream>
#include <vector>
#include <chrono>
#include <string>

using namespace std;

unsigned t0, t1;
int partition(vector<double>& arr, int low, int high);
void quickSort(vector<double>& arr, int low, int high);
void muestra_vector(const vector<double>&);

int main()
{
   ifstream fich("DataGen025.txt");
   if (!fich.is_open())
   {
      cout << "Error al abrir ejemplo.dat\n";
      exit(EXIT_FAILURE);
   }
   t0=clock();
   double valor;
   vector<double> datos;
   while (fich >> valor)
      datos.push_back(valor);
   cout<<"Mostrar vector desordenado: "<<endl;
   muestra_vector(datos);
   cout<<"Ordenar vector"<<endl;
   quickSort(datos, 0, datos.size()-1);
   cout<<"Mostrar vector ordenado: "<<endl;
   muestra_vector(datos);
   t1 = clock();
   double time = (double(t1-t0)/CLOCKS_PER_SEC);
   cout << "Execution Time: " << time << endl;
}

void muestra_vector(const vector<double>& v)
{
  for (auto x : v)
    cout << x <<endl;
  cout << endl;
}

int partition(vector<double>& arr, int low, int high) {
   double pivot = arr[high];
   int i = low - 1;

   for (int j = low; j < high; ++j) {
      if (arr[j] <= pivot) {
         ++i;
         swap(arr[i], arr[j]);
      }
   }
```

```
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<double>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Analizar la complejidad computacional de cada uno de los algoritmos

Bubble Sort

```
void bubbleSort(vector<double>& arr) {
    int n = arr.size();  // O(1) -> Obtener el tamaño del arreglo es una operación constante

    for (int i = 0; i < n - 1; ++i) {  // O(n) -> Este bucle se ejecuta 'n-1' veces
        for (int j = 0; j < n - i - 1; ++j) {  // O(n) -> Este bucle anidado se ejecuta cada vez menos ('n-i-1' veces)
            if (arr[j] > arr[j + 1]) {  // O(1) -> Comparación de dos elementos
                swap(arr[j], arr[j + 1]);  // O(1) -> Intercambiar dos elementos es una operación constante
            }
        }
    }
}

int main() {
    vector<double> arr = {64.0, 34.0, 25.0};  // O(1) -> Inicialización de un vector de tamaño constante

    bubbleSort(arr);  // O(n^2) -> Llamada al algoritmo con un arreglo de 'n' elementos

    return 0;  // O(1) -> Finalización del programa
}
```

Mejor Caso: O(n). El algoritmo está ordenado.

Caso Promedio: O(n^2) El algoritmo está en desorden.

Peor Caso: O(n^2) El algoritmo está en orden inverso.

Heap Sort

```
#include <iostream>
#include <vector>

// Función heapify para asegurar que el subárbol con raíz en i sea un max-heap
void heapify(vector<double>& arr, int n, int i) {
    int largest = i;            // O(1) -> Asignación de un índice
    int left = 2 * i + 1;       // O(1) -> Cálculo del hijo izquierdo
    int right = 2 * i + 2;      // O(1) -> Cálculo del hijo derecho

    // Si el hijo izquierdo es mayor que la raíz
    if (left < n && arr[left] > arr[largest])  // O(1) -> Comparación y verificación de índices
        largest = left;                        // O(1) -> Asignación condicional

    // Si el hijo derecho es mayor que el más grande hasta ahora
```

```
    if (right < n && arr[right] > arr[largest]) // O(1) -> Comparación y verificación de índices
        largest = right;                    // O(1) -> Asignación condicional

    // Si el más grande no es la raíz
    if (largest != i) {                     // O(1) -> Comparación
        std::swap(arr[i], arr[largest]);    // O(1) -> Intercambio de dos elementos
        heapify(arr, n, largest);           // O(log n) -> Llamada recursiva en el subárbol con altura log(n)
    }
}
    void heapSort(vector<double>& arr) {
    int n = arr.size();             // O(1) -> Obtener el tamaño del arreglo

    // Construir el heap (reorganizar el vector)
    for (int i = n / 2 - 1; i >= 0; --i) {  // O(n) -> El bucle recorre la mitad del arreglo (n/2 veces)
        heapify(arr, n, i);             // O(log n) -> Cada llamada a heapify toma tiempo log(n) en el peor caso
    }

    // Extraer un elemento a la vez del heap
    for (int i = n - 1; i > 0; --i) {   // O(n) -> El bucle se ejecuta 'n' veces
        swap(arr[0], arr[i]);           // O(1) -> Intercambiar el primer elemento con el último
        heapify(arr, i, 0);             // O(log n) -> Llamar a heapify para reajustar el heap después de cada extracción
    }
}
int main() {
    vector<double> arr = {12.0, 11.0, 13.0, 5.0, 6.0, 7.0};  // O(1) -> Inicialización del vector

    heapSort(arr);  // O(n log n) -> Llamada a Heap Sort

    return 0;       // O(1) -> Finalización del programa
}
```

Mejor Caso: O(nlogn) (el orden inicial no afecta significativamente el rendimiento de Heap Sort).

Caso Promedio: O(nlogn) (se comporta consistentemente sin importar el orden inicial de los datos).

Peor Caso: O(nlogn) (incluso si los datos están en el orden inverso, el rendimiento sigue siendo O(nlogn).

Insertion Sort

```
void insertionSort(vector<double>& arr) {
    int n = arr.size();  // O(1) -> Obtener el tamaño del vector es una operación constante

    for (int i = 1; i < n; ++i) {  // O(n) -> El bucle se ejecuta 'n-1' veces
        double key = arr[i];       // O(1) -> Asignación de la clave
        int j = i - 1;             // O(1) -> Asignación del índice 'j'

        // Mover los elementos de arr[0..i-1], que son mayores que la clave, una posición adelante
        while (j >= 0 && arr[j] > key) {  // O(j) -> En el peor caso, el bucle se ejecuta j veces (cercano a 'i')
            arr[j + 1] = arr[j];          // O(1) -> Desplazamiento de un elemento
            j = j - 1;                    // O(1) -> Decremento del índice
        }
        arr[j + 1] = key;                 // O(1) -> Colocar la clave en la posición correcta
    }
}
int main() {
    vector<double> arr = {12.0, 11.0, 13.0, 5.0, 6.0};  // O(1) -> Inicialización del vector

    insertionSort(arr);  // O(n^2) -> Llamada al algoritmo Insertion Sort
```

```
    return 0;  // O(1) -> Finalización del programa
}
```

Mejor Caso: O(n) Si el arreglo ya está ordenado, la condición arr[j] > key nunca se cumple, por lo que el bucle while no se ejecuta.

Caso Promedio: O(n^2) El bucle while se ejecuta aproximadamente la mitad del tiempo en cada iteración.

Peor Caso: O(n^2) Cada nueva clave debe ser comparada con todos los elementos anteriores, y cada elemento debe ser movido un lugar hacia la derecha.

Selection Sort

```
void selectionSort(vector<double>& arr) {
    int n = arr.size();  // O(1) -> Obtener el tamaño del vector

    for (int i = 0; i < n - 1; ++i) {  // O(n) -> El bucle externo se ejecuta 'n-1' veces
        int minIndex = i;              // O(1) -> Asignar el índice del mínimo inicial

        // Bucle interno para encontrar el mínimo en la sublista no ordenada
        for (int j = i + 1; j < n; ++j) {  // O(n - i - 1) -> El bucle interno busca el mínimo
            if (arr[j] < arr[minIndex])    // O(1) -> Comparación entre el elemento actual y el mínimo
                minIndex = j;              // O(1) -> Actualización del índice del mínimo
        }

        // Intercambiar el mínimo encontrado con el primer elemento de la sublista no ordenada
        swap(arr[i], arr[minIndex]);  // O(1) -> Intercambio de dos elementos
    }
}

int main() {
    vector<double> arr = {64.0, 25.0, 12.0, 22.0, 11.0};  // O(1) -> Inicialización del vector

    selectionSort(arr);  // O(n^2) -> Llamada a Selection Sort

    return 0;  // O(1) -> Finalización del programa
}
```

Mejor Caso: O(n^2) El algoritmo siempre debe comparar cada elemento para encontrar el mínimo en la sublista no ordenada. Por lo tanto, el número de comparaciones y operaciones es el mismo independientemente del orden inicial.

Caso Promedio: O(n^2) El algoritmo siempre realiza el mismo número de comparaciones, sin importar el orden de los elementos.

Peor Caso: O(n^2) el peor caso también requiere realizar todas las comparaciones posibles.

Shell Sort

```
void shellSort(vector<double>& arr) {
    int n = arr.size();  // O(1) -> Obtener el tamaño del vector

    // Comenzamos con un gap grande, y lo reducimos gradualmente
    for (int gap = n / 2; gap > 0; gap /= 2) {  // O(log n) -> El gap se reduce por la mitad en cada iteración
        // Realizamos una especie de "insertion sort" para cada gap
        for (int i = gap; i < n; ++i) {         // O(n) -> Iteramos sobre los elementos del array
            double temp = arr[i];               // O(1) -> Asignación temporal del elemento actual
            int j;

            // Desplazamos los elementos ordenados del subarray de tamaño gap hacia adelante
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {  // O(n / gap) -> Comparaciones dentro del subarray
```

```
            arr[j] = arr[j - gap];                    // O(1) -> Desplazamiento del elemento
        }

        arr[j] = temp;  // O(1) -> Colocar el elemento temporal en su posición correcta
    }
  }
}

int main() {
  vector<double> arr = {12.0, 34.0, 54.0, 2.0, 3.0};  // O(1) -> Inicialización del vector

  shellSort(arr);  // O(n log n) en promedio -> Llamada a Shell Sort

  return 0;  // O(1) -> Finalización del programa
}
```

Mejor Caso: O(nlogn) El número de comparaciones es reducido, ya que el bucle while no realiza muchos intercambios.

Caso Promedio: O(nlog^2n) La secuencia de gaps original de Shell divide el tamaño del arreglo entre 2 en cada paso.

Peor Caso: O(n^2) El peor caso ocurre cuando la secuencia de gaps no logra reducir significativamente el número de comparaciones.

Merge Sort

```
// Función para fusionar dos subarreglos
void merge(vector<double>& arr, int left, int mid, int right) {
  int n1 = mid - left + 1;  // O(1) -> Tamaño del primer subarreglo
  int n2 = right - mid;     // O(1) -> Tamaño del segundo subarreglo

  vector<double> L(n1), R(n2);  // O(n1 + n2) -> Crear arreglos temporales

  // Copiamos los datos a los arreglos temporales L[] y R[]
  for (int i = 0; i < n1; ++i)  // O(n1)
    L[i] = arr[left + i];      // O(1) por cada copia
  for (int j = 0; j < n2; ++j)  // O(n2)
    R[j] = arr[mid + 1 + j];   // O(1) por cada copia

  // Fusionamos los subarreglos temporales de nuevo en arr[left..right]
  int i = 0, j = 0, k = left;
  while (i < n1 && j < n2) {   // O(n1 + n2) -> Comparar y fusionar los subarreglos
    if (L[i] <= R[j]) {
      arr[k] = L[i];       // O(1)
      i++;
    } else {
      arr[k] = R[j];       // O(1)
      j++;
    }
    k++;
  }

  // Copiamos los elementos restantes de L[], si hay
  while (i < n1) {            // O(n1)
    arr[k] = L[i];          // O(1)
    i++;
    k++;
  }

  // Copiamos los elementos restantes de R[], si hay
```

```
    while (j < n2) {            // O(n2)
        arr[k] = R[j];          // O(1)
        j++;
        k++;
    }
}

// Función Merge Sort
void mergeSort(vector<double>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;  // O(1) -> Calcular el índice del medio

        // Ordenar la primera y la segunda mitad
        mergeSort(arr, left, mid);            // O(n log n) -> Recursión en la primera mitad
        mergeSort(arr, mid + 1, right);       // O(n log n) -> Recursión en la segunda mitad

        // Fusionar las dos mitades ordenadas
        merge(arr, left, mid, right);         // O(n) -> Fusión de los subarreglos
    }
}

int main() {
    vector<double> arr = {12.0, 11.0, 13.0, 5.0, 6.0, 7.0};  // O(1) -> Inicialización del vector

    mergeSort(arr, 0, arr.size() - 1);  // O(n log n) -> Llamada al Merge Sort

    return 0;  // O(1) -> Finalización del programa
}
```

Mejor Caso: O(nlogn) Incluso si el arreglo ya está ordenado, el algoritmo aún debe dividirlo y fusionarlo, por lo que la complejidad sigue siendo.

Caso Promedio: O(nlogn) Merge Sort realiza un número similar de comparaciones y fusiones similar al mejor caso.

Peor Caso: O(nlogn) Merge Sort no se ve afectado por el orden de los elementos, ya que siempre divide y fusiona los subarreglos de la misma manera.

Quick Sort

```
// Función para hacer la partición del array
int partition(vector<double>& arr, int low, int high) {
    double pivot = arr[high];  // O(1) -> Asignación del pivote
    int i = low - 1;           // O(1) -> Inicialización del índice de partición

    for (int j = low; j < high; ++j) {  // O(n) -> Iteración a través del subarreglo
        if (arr[j] <= pivot) {          // O(1) -> Comparación del elemento con el pivote
            ++i;                        // O(1) -> Incremento del índice
            swap(arr[i], arr[j]);  // O(1) -> Intercambio de elementos
        }
    }

    swap(arr[i + 1], arr[high]);   // O(1) -> Colocar el pivote en su posición correcta
    return i + 1;                  // O(1) -> Retornar el índice del pivote
}

// Función recursiva para Quick Sort
void quickSort(vector<double>& arr, int low, int high) {
    if (low < high) {                  // O(1) -> Comparación
        int pi = partition(arr, low, high);  // O(n) en promedio -> Llamada a la función de partición
```

```
        // Ordenar recursivamente los subarreglos
        quickSort(arr, low, pi - 1);     // T(n/2) -> Recursión en el subarreglo izquierdo
        quickSort(arr, pi + 1, high);    // T(n/2) -> Recursión en el subarreglo derecho
    }
}
int main() {
    vector<double> arr = {10.0, 7.0, 8.0, 9.0, 1.0, 5.0};  // O(1) -> Inicialización del vector
    quickSort(arr, 0, arr.size() - 1);  // O(n log n) en promedio -> Llamada a Quick Sort
    return 0;  // O(1) -> Finalización del programa
}
```

Mejor Caso: O(nlogn) El mejor caso ocurre cuando el pivote divide el arreglo en dos subarreglos de tamaño aproximadamente igual en cada recursión.

Caso Promedio: O(nlogn) la partición tiende a dividir el arreglo en subarreglos de tamaños razonablemente balanceados. Aunque el algoritmo realiza O(n) operaciones en cada nivel de recursión, hay O(logn) niveles de recursión.

Peor Caso: O(n^2) El peor caso ocurre cuando el pivote es el menor o el mayor elemento en cada partición, lo que significa que uno de los subarreglos tiene tamaño cero y el otro tiene tamaño n−1.

Evaluar y comparar sus algoritmos usando los archivos de datos y elabore una(s) gráfica(s) comparativa(s).

	DataGen1	DataGen05	DataGen025		DataGen1	DataGen05	DataGen025
Bubble Sort	5489.87	1340.81	353.18	**Selection sort**	1475.09	391.636	117.284
Heap Sort	154.572	72.263	38.251	**Shell sort**	140.214	74.65	38.997
Insertion sort	1127.91	310.757	96.402	**Merge sort**	148.655	72.135	38.964
				Quick sort	142.36	72.85	36.315