6. Algoritmos de Ordenación

Los **algoritmos de ordenación** son algoritmos que colocan elementos de una lista, array o cualquier otra estructura de datos en un orden específico, como numérico o alfabético. La ordenación es una operación fundamental en informática, ya que facilita la búsqueda, fusión, y otras operaciones sobre los datos.

—

0.1. Conceptos Clave en Ordenación

- **Estabilidad:** Un algoritmo de ordenación es estable si mantiene el orden relativo de los elementos con valores iguales. Es decir, si dos elementos tienen el mismo valor, su orden en el array ordenado será el mismo que en el array original.
- **Ordenación In-place (In Situ):** Un algoritmo es in-place si no requiere una cantidad significativa de espacio de memoria adicional aparte de la entrada para realizar la ordenación.
- **Ordenación Comparativa vs. No Comparativa:**
  - **Comparativa:** Se basan en comparar los elementos entre sí (ej., Bubble Sort, Merge Sort, Quick Sort).
  - **No Comparativa:** No se basan en comparaciones y a menudo tienen un rendimiento mejor en casos específicos, pero con restricciones sobre el rango o tipo de datos (ej., Counting Sort, Radix Sort).

—

0.2. Tipos y Funcionamiento de Algoritmos de Ordenación

Existen numerosos algoritmos de ordenación, cada uno con sus propias características de rendimiento y aplicabilidad. Aquí cubriremos algunos de los más representativos.

Algoritmos de Ordenación Simples (Generalmente $\mathcal{O}(N^2)$)

Estos algoritmos son fáciles de entender e implementar, pero su eficiencia los hace imprácticos para grandes conjuntos de datos.

- **Bubble Sort (Ordenación de Burbuja):** Recorre repetidamente la lista, compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Las pasadas se repiten hasta que no se necesiten más intercambios, indicando que la lista está ordenada.

- **Selection Sort (Ordenación por Selección):** Divide la lista en una parte ordenada y una no ordenada. En cada paso, encuentra el elemento más pequeño (o más grande) de la parte no ordenada y lo coloca al final de la parte ordenada.
- **Insertion Sort (Ordenación por Inserción):** Construye la lista ordenada un elemento a la vez. Cada nuevo elemento se inserta en su posición correcta dentro de la parte ya ordenada de la lista. Eficiente para listas pequeñas o casi ordenadas.

Algoritmos de Ordenación Eficientes (Generalmente $\mathcal{O}(N \log N)$)

Estos algoritmos son preferibles para ordenar grandes volúmenes de datos.

- **Merge Sort (Ordenación por Fusión):** Un algoritmo de "Divide y Vencerás". Divide recursivamente el array en dos mitades, las ordena de forma independiente y luego fusiona las mitades ordenadas para producir un array completamente ordenado. Es un algoritmo estable.
- **Quick Sort (Ordenación Rápida):** También un algoritmo de "Divide y Vencerás". Selecciona un elemento como "pivote" y particiona los otros elementos en dos sub-arrays, según sean menores o mayores que el pivote. Los sub-arrays se ordenan recursivamente. No es un algoritmo estable, pero es muy rápido en promedio.
- **Heap Sort (Ordenación por Montículos):** Utiliza la estructura de datos "Heap" (montículo binario) para ordenar. Primero construye un max-heap (o min-heap) de la entrada, y luego extrae repetidamente el elemento máximo (la raíz) del heap, colocándolo al final del array, y reconstruye el heap.

—

0.3. Ejemplos de Algoritmos de Ordenación en C++

Veamos implementaciones básicas de algunos algoritmos de ordenación, incluyendo los eficientes.

Bubble Sort ($\mathcal{O}(N^2)$)

Listing 1: Implementación de Bubble Sort

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::swap

using namespace std;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; ++i) {
        bool swapped = false; // Optimizaci n: Si no hay
            intercambios en una pasada, ya est  ordenado
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
        if (swapped == false) {
            break; // No hubo intercambios, el array
                est  ordenado
        }
    }
}

int main() {
    vector<int> nums = {64, 34, 25, 12, 22, 11, 90};
    cout << "Array original: ";
    for (int x : nums) {
        cout << x << " ";
    }
    cout << endl;

    bubbleSort(nums);

    cout << "Array ordenado (Bubble Sort): ";
    for (int x : nums) {
        cout << x << " ";
    }
    cout << endl;
    return 0;
}
```

Merge Sort ($\mathcal{O}(N \log N)$)

Listing 2: Implementación de Merge Sort

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::min y std::max (no usados en este merge, pero comunes)

using namespace std;

// Funci n para fusionar dos subarrays ordenados
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Crear arrays temporales
    vector<int> L(n1);
    vector<int> R(n2);

    // Copiar datos a arrays temporales L[] y R[]
    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];

    // Fusionar los arrays temporales de vuelta en arr[left..right]
    int i = 0; //  ndice  inicial del primer subarray
    int j = 0; //  ndice  inicial del segundo subarray
    int k = left; //  ndice  inicial del array fusionado

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

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

```
44
45      // Copiar los elementos restantes de R[], si los hay
46      while (j < n2) {
47          arr[k] = R[j];
48          j++;
49          k++;
50      }
51  }
52
53  // Funci n principal de Merge Sort
54  void mergeSort(vector<int>& arr, int left, int right) {
55      if (left >= right) {
56          return; // Caso base: array de 0 o 1 elemento ya
                est  ordenado
57      }
58      int mid = left + (right - left) / 2;
59      mergeSort(arr, left, mid);       // Ordena la primera
            mitad
60      mergeSort(arr, mid + 1, right);  // Ordena la segunda
            mitad
61      merge(arr, left, mid, right);    // Fusiona las dos
            mitades ordenadas
62  }
63
64  int main() {
65      vector<int> nums = {38, 27, 43, 3, 9, 82, 10};
66      cout << "Array original: ";
67      for (int x : nums) {
68          cout << x << " ";
69      }
70      cout << endl;
71
72      mergeSort(nums, 0, nums.size() - 1);
73
74      cout << "Array ordenado (Merge Sort): ";
75      for (int x : nums) {
76          cout << x << " ";
77      }
78      cout << endl;
79      return 0;
80  }
```

Quick Sort ($\mathcal{O}(N \log N)$ promedio)

Listing 3: Implementación de Quick Sort

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::swap

using namespace std;

// Funci n para particionar el array alrededor de un pivote
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Tomamos el  ltimo  elemento como pivote
    int i = (low - 1); //  ndice  del elemento m s peque o

    for (int j = low; j <= high - 1; j++) {
        // Si el elemento actual es m s peque o o igual que el pivote
        if (arr[j] <= pivot) {
            i++; // Incrementa el  ndice  del elemento m s peque o
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// Funci n principal de Quick Sort
void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        // pi es el  ndice  de partici n, arr[pi] est ahora en su lugar correcto
        int pi = partition(arr, low, high);

        // Ordenar recursivamente los elementos antes y despu s de la partici n
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> nums = {10, 7, 8, 9, 1, 5};
    cout << "Array original: ";
    for (int x : nums) {
        cout << x << " ";
```

```
40      }
41      cout << endl;
42
43      quickSort(nums, 0, nums.size() - 1);
44
45      cout << "Array ordenado (Quick Sort): ";
46      for (int x : nums) {
47          cout << x << " ";
48      }
49      cout << endl;
50      return 0;
51  }
```

Heap Sort ($\mathcal{O}(N \log N)$)

Listing 4: Implementación de Heap Sort

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Para std::swap

using namespace std;

// Funci n para heapificar un sub rbol con ra z en el ndice i
// n es el tama o del heap
void heapify(vector<int>& arr, int n, int i) {
    int largest = i; // Inicializar largest como la ra z
    int left = 2 * i + 1; // Hijo izquierdo
    int right = 2 * i + 2; // Hijo derecho

    // Si el hijo izquierdo es m s grande que la ra z
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // Si el hijo derecho es m s grande que el actual largest
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // Si largest no es la ra z
    if (largest != i) {
        swap(arr[i], arr[largest]);
        // Llamar recursivamente a heapify en el sub rbol afectado
        heapify(arr, n, largest);
    }
}

// Funci n principal de Heap Sort
void heapSort(vector<int>& arr) {
    int n = arr.size();

    // Construir un max-heap (reorganizar array)
    // El ltimo nodo padre est en el ndice (n/2 - 1)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // Extraer elementos uno por uno del heap
```

```
    for (int i = n - 1; i > 0; i--) {
        // Mover la ra z actual al final
        swap(arr[0], arr[i]);

        // Llamar a heapify en el heap reducido
        heapify(arr, i, 0);
    }
}

int main() {
    vector<int> nums = {12, 11, 13, 5, 6, 7};
    cout << "Array original: ";
    for (int x : nums) {
        cout << x << " ";
    }
    cout << endl;

    heapSort(nums);

    cout << "Array ordenado (Heap Sort): ";
    for (int x : nums) {
        cout << x << " ";
    }
    cout << endl;
    return 0;
}
```

—

0.4. Análisis de Eficiencia de los Algoritmos de Ordenación

La eficiencia se mide en términos de complejidad temporal (número de comparaciones y/o movimientos) y complejidad espacial (memoria adicional requerida).

- **Algoritmos $\mathcal{O}(N^2)$ (ej., Bubble Sort, Selection Sort, Insertion Sort):**
  - **Complejidad Temporal:** $\mathcal{O}(N^2)$ en el peor y caso promedio. Son ineficientes para grandes conjuntos de datos.
  - **Complejidad Espacial:** Mayormente $\mathcal{O}(1)$ (in-place), excepto Insertion Sort que puede ser $\mathcal{O}(1)$ o $\mathcal{O}(N)$ si se usa lista enlazada.
- **Algoritmos $\mathcal{O}(N \log N)$ (ej., Merge Sort, Quick Sort, Heap Sort):**
  - **Complejidad Temporal:** $\mathcal{O}(N \log N)$ en el peor y caso promedio (Quick Sort tiene $\mathcal{O}(N^2)$ en el peor caso, pero es raro). Son los más eficientes para uso general.

- **Complejidad Espacial:**
  - **Merge Sort:** $\mathcal{O}(N)$ debido a los arrays temporales usados en la fusión. No es in-place.
  - **Quick Sort:** $\mathcal{O}(\log N)$ en promedio (debido a la recursión de la pila), $\mathcal{O}(N)$ en el peor caso. Es in-place.
  - **Heap Sort:** $\mathcal{O}(1)$ (in-place).

—

0.5. Consideraciones al Elegir un Algoritmo de Ordenación

La elección del algoritmo de ordenación depende de varios factores:

- **Tamaño de los datos:** Para datos pequeños, la simplicidad de un algoritmo $\mathcal{O}(N^2)$ puede ser aceptable. Para grandes datos, se prefieren los $\mathcal{O}(N \log N)$.
- **¿Los datos ya están casi ordenados?:** Insertion Sort es muy eficiente en este caso.
- **Restricciones de memoria:** Algoritmos in-place como Quick Sort o Heap Sort son mejores si la memoria es limitada.
- **Estabilidad requerida?:** Si el orden relativo de elementos iguales es importante, se debe usar un algoritmo estable como Merge Sort.
- **Rendimiento en el peor caso:** Quick Sort puede degradarse a $\mathcal{O}(N^2)$, mientras que Merge Sort siempre es $\mathcal{O}(N \log N)$.

La ordenación es una operación fundamental que impacta directamente el rendimiento de muchas otras operaciones de procesamiento de datos.En la práctica usaremos la función sort de la biblioteca algorithms. Sin embargo es importante conocer su funcionamiento.