Métodos de Ordenamiento:

Antes de trabajar con los métodos de ordenamiento en C++, es importante tener una comprensión sólida de ciertos temas previos para facilitar el aprendizaje y la implementación de los algoritmos. Estos temas te proporcionarán una buena base para comprender cómo funcionan los algoritmos de ordenamiento, optimizarlos y adaptarlos a tus necesidades.

Temas Previos:
Variables y Tipos de Datos

Enteros, flotantes, caracteres, y bool son fundamentales para almacenar los elementos a ordenar.
Arreglos (Arrays): El conocimiento de cómo declarar y manipular arreglos es esencial, ya que los algoritmos de ordenamiento suelen trabajar con estos.
Ejemplo:

```
int arr[5] = {3, 1, 4, 1, 5};
```

Bucles (for, while, do-while)

Los algoritmos de ordenamiento casi siempre implican bucles anidados, por lo que es necesario comprender cómo funcionan los bucles en C++.

Ejemplo de bucle for:

```
for (int i = 0; i < 5; i++) {
     cout << arr[i] << " ";
}
```

Condicionales (if, else, switch)

Los algoritmos de ordenamiento necesitan tomar decisiones (por ejemplo, si un elemento es mayor que otro) utilizando condicionales.

Ejemplo:

```
if (arr[i] > arr[j]) {
     // Intercambiar los elementos
}
```

Funciones

Los algoritmos de ordenamiento se suelen implementar como funciones, lo que te permite reutilizar el código y modularizar tu programa.
También es útil conocer paso por valor y por referencia, ya que puedes pasar arreglos a funciones por referencia para modificar su contenido.

Ejemplo de una función simple:

```
void swap(int &a, int &b) {
     int temp = a;
     a = b;
     b = temp;
}
```

Manejo de Punteros y Referencias

En algunos algoritmos de ordenamiento, especialmente cuando trabajas con estructuras de datos dinámicas o deseas optimizar el uso de memoria, el manejo de punteros y referencias es clave.

Ejemplo básico de punteros:

```
int a = 5;
int* ptr = &a;   // puntero a la variable a
```

Estructuras de Datos Básicas

Arreglos (Arrays): Son la estructura de datos más básica que se utiliza en la mayoría de los algoritmos de ordenamiento.
Vectores (std::vector): Es recomendable entender cómo funcionan los vectores en C++ ya que son más flexibles que los arreglos.

Ejemplo de un vector:

```
#include <vector>
```

```
std::vector<int> vec = {3, 1, 4, 1, 5};
```

Complejidad Temporal y Espacial (Análisis de Algoritmos)

Es útil conocer los conceptos de complejidad temporal (cómo escala el tiempo de ejecución de un algoritmo a medida que aumentan los datos) y complejidad espacial (el uso de memoria de un algoritmo).
Familiarízate con notaciones como O(n), O(n²), y O(log n), que se usan para describir la eficiencia de los algoritmos de ordenamiento.
Recursividad

Algunos de los algoritmos de ordenamiento más eficientes, como Merge Sort y Quick Sort, utilizan recursividad. Por lo tanto, es esencial tener una buena comprensión de cómo funcionan las funciones recursivas.

Ejemplo de función recursiva:

```
int factorial(int n) {
     if (n <= 1) return 1;
     return n * factorial(n - 1);
}
```

Manejo de Memoria (Opcional)

Aunque no es imprescindible para todos los métodos de ordenamiento, comprender cómo gestionar la memoria, sobre todo en arreglos dinámicos y con el uso de punteros, te será útil al trabajar con grandes conjuntos de datos o en entornos donde el rendimiento y el uso de memoria son críticos.
Ejemplo de uso de new y delete:

```
int* arr = new int[10];
// ... trabajar con arr
delete[] arr;
```

Ejemplo Completo
Este ejemplo usa varios conceptos anteriores para implementar Bubble Sort en una función y pasar el arreglo por referencia:

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
     for (int i = 0; i < n - 1; i++) {
          for (int j = 0; j < n - i - 1; j++) {
               if (arr[j] > arr[j + 1]) {
                    // Intercambiar elementos usando una función
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
               }
          }
     }
}

void printArray(int arr[], int n) {
     for (int i = 0; i < n; i++)
          cout << arr[i] << " ";
     cout << endl;
}

int main() {
     int arr[] = {64, 34, 25, 12, 22, 11, 90};
     int n = sizeof(arr)/sizeof(arr[0]);

     cout << "Array original: \n";
     printArray(arr, n);

     bubbleSort(arr, n);

     cout << "Array ordenado: \n";
     printArray(arr, n);

     return 0;
}
```

Tema Ordenamiento:

Los métodos de ordenamiento son fundamentales en la programación, y en C++ existen varias técnicas para ordenar datos. Estos algoritmos se pueden dividir en dos grandes categorías:

Ordenamientos internos: Los datos se ordenan directamente en la memoria principal (RAM).
Ordenamientos externos: Se usan cuando los datos son muy grandes para caber en la memoria principal y se necesita trabajar con almacenamiento secundario.

Algoritmos más utilizados hoy en día
Los algoritmos más usados para trabajar en entornos modernos son aquellos que tienen un buen rendimiento en términos de tiempo y espacio, como:

Quicksort
Mergesort
Heapsort

Estos tres algoritmos son rápidos y tienen un tiempo promedio de ejecución de O(n log n). Dependiendo del contexto y los requisitos específicos, otros algoritmos como Bubble Sort, Insertion Sort, o Selection Sort también se usan en escenarios más específicos, aunque son menos eficientes en general.

Algoritmos de Ordenamiento en C++
Aquí te explico algunos de los algoritmos de ordenamiento más comunes con ejemplos en C++.

1. Bubble Sort (**Ordenamiento burbuja**)
Este algoritmo compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto. Se repite este proceso hasta que el arreglo esté ordenado.

Complejidad: O(n²)
Cuándo usarlo: Para pequeños conjuntos de datos o cuando la simplicidad del código es una prioridad.

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
     for (int i = 0; i < n - 1; i++) {
          for (int j = 0; j < n - i - 1; j++) {
               if (arr[j] > arr[j + 1]) {
                    // Intercambiar arr[j] y arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
               }
          }
     }
}

int main() {
     int arr[] = {64, 34, 25, 12, 22, 11, 90};
     int n = sizeof(arr)/sizeof(arr[0]);

     bubbleSort(arr, n);

     cout << "Array ordenado: \n";
     for (int i = 0; i < n; i++)
          cout << arr[i] << " ";
     return 0;
}
```

2. Selection Sort (**Ordenamiento por selección**)
Este algoritmo encuentra el elemento más pequeño del arreglo y lo coloca en la primera posición. Luego, encuentra el segundo más pequeño y lo coloca en la segunda posición, y así sucesivamente.

Complejidad: O(n²)
Cuándo usarlo: Ideal cuando el número de intercambios debe ser mínimo.

```
#include <iostream>
using namespace std;

void selectionSort(int arr[], int n) {
     for (int i = 0; i < n-1; i++) {
          int minIndex = i;
```

```
            for (int j = i+1; j < n; j++) {
                  if (arr[j] < arr[minIndex]) {
                        minIndex = j;
                  }
            }
            // Intercambiar arr[i] y arr[minIndex]
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
      }
}

int main() {
      int arr[] = {64, 25, 12, 22, 11};
      int n = sizeof(arr)/sizeof(arr[0]);

      selectionSort(arr, n);

      cout << "Array ordenado: \n";
      for (int i = 0; i < n; i++)
            cout << arr[i] << " ";
      return 0;
}
```

3. Insertion Sort (**Ordenamiento por inserción**)
Este algoritmo construye el arreglo ordenado uno a uno, tomando cada elemento y ubicándolo en la posición correcta en la porción previamente ordenada del arreglo.

Complejidad: O(n²)
Cuándo usarlo: Funciona bien con datos pequeños o parcialmente ordenados.

```
#include <iostream>
using namespace std;

void insertionSort(int arr[], int n) {
      for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            // Mover elementos que son mayores que key una posición adelante
            while (j >= 0 && arr[j] > key) {
                  arr[j + 1] = arr[j];
                  j = j - 1;
            }
            arr[j + 1] = key;
      }
}

int main() {
      int arr[] = {12, 11, 13, 5, 6};
      int n = sizeof(arr)/sizeof(arr[0]);

      insertionSort(arr, n);

      cout << "Array ordenado: \n";
      for (int i = 0; i < n; i++)
            cout << arr[i] << " ";
      return 0;
}
```

4. Merge Sort (**Ordenamiento por mezcla**)
Este es un algoritmo recursivo que divide el arreglo en dos mitades, ordena cada mitad y luego combina las mitades ordenadas en un solo arreglo.

Complejidad: O(n log n)
Cuándo usarlo: Para grandes conjuntos de datos.

```
#include <iostream>
using namespace std;

void merge(int arr[], int l, int m, int r) {
      int n1 = m - l + 1;
      int n2 = r - m;

      int L[n1], R[n2];

      for (int i = 0; i < n1; i++)
            L[i] = arr[l + i];
```

```
     for (int j = 0; j < n2; j++)
          R[j] = arr[m + 1 + j];

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

     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(int arr[], int l, int r) {
     if (l < r) {
          int m = l + (r - l) / 2;

          mergeSort(arr, l, m);
          mergeSort(arr, m + 1, r);
          merge(arr, l, m, r);
     }
}

int main() {
     int arr[] = {12, 11, 13, 5, 6, 7};
     int arr_size = sizeof(arr) / sizeof(arr[0]);

     mergeSort(arr, 0, arr_size - 1);

     cout << "Array ordenado: \n";
     for (int i = 0; i < arr_size; i++)
          cout << arr[i] << " ";
     return 0;
}
```

5. Quick Sort (**Ordenamiento rápido**)
Este algoritmo selecciona un elemento como pivote y particiona el arreglo en dos subarreglos, colocando los menores al pivote a la izquierda y los mayores a la derecha. Se aplica recursivamente a las subparticiones.

Complejidad: O(n log n) en promedio
Cuándo usarlo: Para grandes conjuntos de datos donde el ordenamiento en su lugar es importante.

```
#include <iostream>
using namespace std;

int partition(int arr[], int low, int high) {
     int pivot = arr[high];
     int i = (low - 1);

     for (int j = low; j < high; j++) {
          if (arr[j] <= pivot) {
               i++;
               int temp = arr[i];
               arr[i] = arr[j];
               arr[j] = temp;
          }
     }
     int temp = arr[i + 1];
     arr[i + 1] = arr[high];
     arr[high] = temp;
```

```
     return (i + 1);
}

void quickSort(int arr[], int low, int high) {
     if (low < high) {
          int pi = partition(arr, low, high);
          quickSort(arr, low, pi - 1);
          quickSort(arr, pi + 1, high);
     }
}

int main() {
     int arr[] = {10, 7, 8, 9, 1, 5};
     int n = sizeof(arr) / sizeof(arr[0]);

     quickSort(arr, 0, n - 1);

     cout << "Array ordenado: \n";
     for (int i = 0; i < n; i++)
          cout << arr[i] << " ";
     return 0;
}
```

Algoritmos más utilizados hoy en día
Quick Sort: Ideal para la mayoría de aplicaciones debido a su velocidad y eficiencia en la mayoría de los casos.
Merge Sort: Preferido en escenarios donde se requiere estabilidad en el ordenamiento y trabajar con grandes conjuntos de datos.
Heapsort: Utilizado en contextos donde es importante el uso eficiente de la memoria.
También, C++ tiene la función de biblioteca estándar `std::sort` que usa una mezcla de Quick Sort, Heap Sort y Insertion Sort para diferentes tamaños de datos, lo que lo convierte en una excelente opción predeterminada.