Implementación de los Diferentes Tipos de Ordenamiento(Sorting) en el Lenguaje C++

Tejada Lazo Jordy Rolando[1]
[1]*Universidad Nacional de San Agustín, Perú, jtejadala@unsa.edu.pe*

Abstract– *En el presente estudio, se llevó a cabo una evaluación exhaustiva de varios algoritmos de ordenamiento implementados en un entorno de programación C++. El objetivo principal de este trabajo de investigación fue medir y comparar el rendimiento de diferentes algoritmos de ordenamiento en términos de tiempo de ejecución, cantidad de comparaciones realizadas y cantidad de escrituras en memoria. Se realizaron pruebas utilizando diferentes tamaños de arreglos como entrada para cada algoritmo de ordenamiento.*

Keywords-- Sort, C++, Sorting, Algorithms

I. Introducción

Los algoritmos de ordenamiento desempeñan un papel esencial en la informática, la ciencia de la computación y la ingeniería de sistemas. La capacidad de organizar datos de manera eficiente y rápida es fundamental para una amplia gama de aplicaciones, desde bases de datos hasta algoritmos de búsqueda y mucho más. En este contexto, los algoritmos de ordenamiento son una pieza fundamental en el desarrollo de software eficiente y en la optimización de procesos computacionales.

En este estudio, exploraremos varios algoritmos de ordenamiento implementados en C++, con el objetivo de comprender su funcionamiento, rendimiento y aplicaciones prácticas. Estos algoritmos son esenciales en la programación y se utilizan en una variedad de contextos, desde la clasificación de datos en aplicaciones empresariales hasta la optimización de algoritmos de búsqueda en proyectos de inteligencia artificial.

Para evaluar y comparar la eficiencia de estos algoritmos, se realizarán pruebas de rendimiento utilizando conjuntos de datos de diferentes tamaños. Además, se analizarán aspectos como el número de comparaciones y escrituras necesarias para completar cada ordenamiento. Esto nos permitirá no solo comprender cómo funcionan estos algoritmos, sino también identificar cuáles son los más adecuados para situaciones específicas.

II. Marco Teórico

En esta sección del marco teórico, se presentarán definiciones y conceptos fundamentales relacionados con los algoritmos de ordenamiento en el lenguaje de programación C++. Además, se describirán los dispositivos utilizados en la investigación, como Leap Motion, y las herramientas de evaluación de usabilidad, como el Sistema de Escala de Usabilidad (SUS).

A. Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son procedimientos utilizados en programación para organizar una colección de datos en un orden específico. En el contexto de C++, estos algoritmos son esenciales y se utilizan para clasificar datos de manera eficiente en aplicaciones informáticas. Algunos de los algoritmos de ordenamiento comunes en C++ incluyen Selection Sort, Heap Sort, Insertion Sort, Merge Sort, Quick Sort e Intro Sort. Cada uno de estos algoritmos tiene sus propias características y ventajas, lo que los hace adecuados para diferentes situaciones y tamaños de datos.

B. Selection Sort

Selection Sort es un algoritmo de ordenamiento simple pero ineficiente. Funciona dividiendo la lista de elementos en dos partes: la parte ordenada y la parte desordenada. En cada iteración, busca el elemento más pequeño en la parte desordenada y lo coloca al principio de la parte ordenada. Repite este proceso hasta que toda la lista esté ordenada. A pesar de su simplicidad, el Selection Sort no es adecuado para grandes conjuntos de datos, ya que su complejidad es de O(n^2), lo que significa que su tiempo de ejecución aumenta cuadráticamente con el tamaño de la lista.

C. Heap Sort

Heap Sort es un algoritmo de ordenamiento basado en estructuras de datos llamadas "heaps" (montículos). En este enfoque, se crea un montículo a partir de la lista de elementos y luego se extrae repetidamente el elemento máximo (en un montículo máximo) o mínimo (en un montículo mínimo) y se coloca al final de la lista ordenada. Este proceso se repite hasta que todos los elementos estén ordenados. Heap Sort tiene una complejidad de tiempo de O(n log n) en el peor de los casos, lo que lo hace más eficiente que Selection Sort para conjuntos de datos grandes.

D. Insertion Sort

Insertion Sort es un algoritmo de ordenamiento eficiente para conjuntos de datos pequeños o casi ordenados. Funciona dividiendo la lista en una parte ordenada y una parte desordenada, y luego inserta elementos de la parte desordenada en la parte ordenada en su posición correcta. En cada iteración, se toma un elemento de la parte desordenada y se compara con los elementos en la parte ordenada hasta encontrar su lugar adecuado. Insertion Sort tiene una complejidad de tiempo de O(n^2) en el peor de los casos, pero es rápido para listas pequeñas debido a su naturaleza incremental.

E. *Merge Sort*

Merge Sort es un algoritmo de ordenamiento basado en la técnica de "dividir y conquistar". Divide la lista en mitades más pequeñas hasta que se llega a sublistas de un solo elemento, luego combina estas sublistas en orden para obtener la lista ordenada final. Merge Sort es altamente eficiente y tiene una complejidad de tiempo de O(n log n) en todos los casos, lo que lo convierte en una excelente opción para ordenar grandes conjuntos de datos.

F. *Quick Sort*

Quick Sort es otro algoritmo de ordenamiento basado en la técnica "dividir y conquistar". Se elige un elemento pivote de la lista y se reorganizan los elementos para que los menores que el pivote estén a su izquierda y los mayores a su derecha. Luego, se aplica el mismo proceso de forma recursiva a las sublistas izquierda y derecha. Quick Sort es rápido y eficiente en la mayoría de los casos y tiene una complejidad de tiempo promedio de O(n log n). Sin embargo, su rendimiento puede degradarse en el peor de los casos si se elige un pivote inapropiado.

G. *Intro Sort*

Intro Sort es una variante del Quick Sort que combina las ventajas del Quick Sort, el Merge Sort, el Heap Sort y el Insertion Sort. Utiliza el Quick Sort para ordenar la lista inicialmente, pero si la profundidad de recursión excede un cierto límite, cambia a Heap Sort para evitar la degradación del rendimiento en casos extremos. Esto lo convierte en un algoritmo eficiente y versátil que es rápido en la mayoría de los casos y tiene una complejidad de tiempo promedio de O(n log n).

III. Metodología

El presente estudio tiene un enfoque cuantitativo debido a que este enfoque implica que el estudio se basará en datos numéricos y medibles. En el contexto de los algoritmos de ordenamiento en C++, se analizó el tiempo de ejecución de los algoritmos, la cantidad de comparaciones realizadas y/o la cantidad de movimientos (escrituras) de datos que se producen durante la ordenación. Estos son datos objetivos y cuantificables.

Se ha seguido este enfoque cuantitativo para analizar y evaluar exhaustivamente los algoritmos de ordenamiento en C++. Esta metodología se ha centrado en mediciones objetivas y análisis estadístico, lo que ha permitido obtener una comprensión profunda del rendimiento de estos algoritmos.

En primer lugar, se ha medido el tiempo de ejecución de cada algoritmo. Esta métrica es fundamental para evaluar la eficiencia en términos de velocidad. A través de pruebas en diferentes escenarios(elementos aleatorios) y tamaños de conjuntos de datos, hemos determinado qué algoritmos son más rápidos y cómo se comportan en diversas situaciones.

Además, se ha cuantificado la cantidad de escrituras realizadas por cada algoritmo. Este aspecto es crítico, especialmente cuando se manejan grandes conjuntos de datos, ya que las escrituras de datos pueden tener un impacto significativo en el rendimiento. Esta metodología nos ha permitido evaluar cómo cada algoritmo gestiona el movimiento de datos y compararlos en función de esta métrica.

Otra variable clave que se ha tenido en cuenta es el número de comparaciones realizadas por cada algoritmo. Esto es esencial para evaluar la eficiencia en términos de la complejidad de comparación de cada algoritmo, lo que puede influir en su rendimiento general. Este análisis ha proporcionado información valiosa sobre cómo se comportan estos algoritmos en función de la cantidad de comparaciones que realizan.

Además, se ha considerado cuidadosamente el tamaño de los elementos en los conjuntos de datos. Se reconoce que diferentes algoritmos pueden comportarse de manera distinta según el tamaño de los elementos. Por lo tanto, se han llevado a cabo pruebas con conjuntos de datos de diferentes tamaños para comprender cómo se adaptan y rinden en diversas situaciones.

IV. Resultados y Discusión

Esta parte de resultados se enfocó en base al tiempo de ejecución, las escrituras, comparaciones y al tamaño del arreglo que se insertan en los diferentes algoritmos de ordenamiento. Teniendo como resultado luego de haber implementado los siguientes algoritmos.

```
/* * * * * * * * IMPLEMENTACIONES * * * * * * * * */

template < class RandomIt >
void SelectionSort ( RandomIt b, RandomIt e ) {
  SelectionSort ( b, e, less<decltype(*(b))>() );
}

template < class RandomIt, class Compare >
void SelectionSort ( RandomIt b, RandomIt e, Compare comp ) {
  cont_comparaciones = cont_escrituras = 0;
  for (RandomIt s = b; s < e-1; s++) {
    RandomIt m = s;
    for ( RandomIt i = s+1; i < e; i++ ) {
      cont_comparaciones++;
      if ( comp ( *i, *m ) ) m = i;
    }
    if ( m != s ) {
      cont_escrituras+=3; // swap(a,b) escribe t=a;a=b;b=t;
      swap ( *s, *m );
    }
  }
}
```

Figura 1. Implementación del Selection Sort

```
template < class RandomIt >
void HeapSort ( RandomIt b, RandomIt e ) {
  HeapSort ( b, e, less<decltype(*(b))>() );
}

template < class RandomIt, class Compare >
void HeapSort ( RandomIt b, RandomIt e, Compare comp ) {
  cont_comparaciones = cont_escrituras = 0;
  // Utilizando la heap de std
  make_heap ( b, e );
  sort_heap ( b, e );
}
```

Figura 2. Implementación del Heap Sort

```
template < class RandomIt >
void InsertionSort ( RandomIt b, RandomIt e ) {
  InsertionSort ( b, e, less<decltype(*(b))>() );
}

template <class RandomIt, class Compare>
void InsertionSort(RandomIt b, RandomIt e, Compare comp) {
    cont_comparaciones = cont_escrituras = 0;
    for (RandomIt i = b + 1; i < e; ++i) {
        RandomIt j = i;
        while (j > b && comp(*j, *(j - 1))) {
            cont_comparaciones++;
            cont_escrituras++;
            std::swap(*j, *(j - 1));
            --j;
        }
    }
}
```

Figura 3. Implementación del Insertion Sort

```
template < class RandomIt >
void MergeSort ( RandomIt b, RandomIt e ) {
  MergeSort ( b, e, less<decltype(*(b))>() );
}

template <class RandomIt, class Compare>
void MergeSort(RandomIt b, RandomIt e, Compare comp) {
    if (std::distance(b, e) <= 1) {
        return;  // La lista ya está ordenada o tiene un solo elemento
    }

    RandomIt middle = b + std::distance(b, e) / 2;

    // Dividir la lista en dos partes
    MergeSort(b, middle, comp);
    MergeSort(middle, e, comp);

    // Fusionar las dos partes ordenadas
    std::vector<typename RandomIt::value_type> tmp(std::distance(b, e));
    RandomIt left = b, right = middle, tmp_it = tmp.begin();

    while (left < middle && right < e) {
        cont_comparaciones++;
        if (comp(*left, *right)) {
            cont_escrituras++;
            *tmp_it = *left;
            ++left;
        } else {
            cont_escrituras++;
            *tmp_it = *right;
            ++right;
        }
        ++tmp_it;
    }

    // Copiar los elementos restantes (si los hay)
    tmp_it = std::copy(left, middle, tmp_it);
    tmp_it = std::copy(right, e, tmp_it);

    // Copiar los elementos ordenados de vuelta a la lista original
    std::copy(tmp.begin(), tmp.end(), b);
}
```

Figura 4. Implementación del Merge Sort

```cpp
template < class RandomIt >
void QuickSort ( RandomIt b, RandomIt e ) {
  MergeSort ( b, e, less<decltype(*(b))>() );
}
template <class RandomIt, class Compare>
void QuickSort(RandomIt b, RandomIt e, Compare comp) {
    if (std::distance(b, e) <= 1) {
        return;  // La lista ya está ordenada o tiene un solo elemento
    }

    RandomIt pivot = b + std::distance(b, e) / 2;
    typename RandomIt::value_type pivot_value = *pivot;

    RandomIt left = b;
    RandomIt right = e - 1;

    while (left <= right) {
        while (comp(*left, pivot_value)) {
            cont_comparaciones++;
            left++;
        }

        while (comp(pivot_value, *right)) {
            cont_comparaciones++;
            right--;
        }

        if (left <= right) {
            cont_escrituras++;
            std::swap(*left, *right);
            left++;
            right--;
        }
    }

    // Llamada recursiva para las dos mitades
    if (b < right) {
        QuickSort(b, right + 1, comp);
    }
    if (left < e) {
        QuickSort(left, e, comp);
    }
}
```

Figura 5. Implementación del Quick Sort

```cpp
// INTROSORT
template <class RandomIt>
void IntroSort(RandomIt b, RandomIt e) {
    IntroSort(b, e, std::less<decltype(*(b))>());
}

template <class RandomIt, class Compare>
void IntroSort(RandomIt b, RandomIt e, Compare comp) {
    cont_comparaciones = cont_escrituras = 0;
    int maxDepth = 2 * log2(std::distance(b, e));
    IntroSortImpl(b, e, comp, maxDepth);
}

template <class RandomIt, class Compare>
void IntroSortImpl(RandomIt b, RandomIt e, Compare comp, int maxDepth) {
    if (maxDepth <= 0) {
        // Si alcanzamos la profundidad máxima, cambiamos a Heap Sort
        HeapSort(b, e, comp);
        return;
    }

    if (std::distance(b, e) <= 16) {
        // Si el rango es pequeño, cambiamos a Insertion Sort
        InsertionSort(b, e, comp);
        return;
    }

    RandomIt pivot = Partition(b, e, comp);
    IntroSortImpl(b, pivot, comp, maxDepth - 1);
    IntroSortImpl(pivot + 1, e, comp, maxDepth - 1);
}
```

Figura 6. Implementación del Intro Sort Parte 1

```cpp
template <class RandomIt, class Compare>
RandomIt Partition(RandomIt b, RandomIt e, Compare comp) {
    RandomIt pivot = b + std::distance(b, e) / 2;
    typename RandomIt::value_type pivot_value = *pivot;

    RandomIt left = b;
    RandomIt right = e - 1;

    while (true) {
        while (comp(*left, pivot_value)) {
            cont_comparaciones++;
            left++;
        }

        while (comp(pivot_value, *right)) {
            cont_comparaciones++;
            right--;
        }

        if (left >= right) {
            return right;
        }

        cont_escrituras++;
        std::swap(*left, *right);
        left++;
        right--;
    }
}
```

Figura 7. Implementación del Intro Sort Parte 2

Donde luego de implementar estos diversos algoritmos de ordenamiento en C++ se obtuvieron los siguientes resultados:

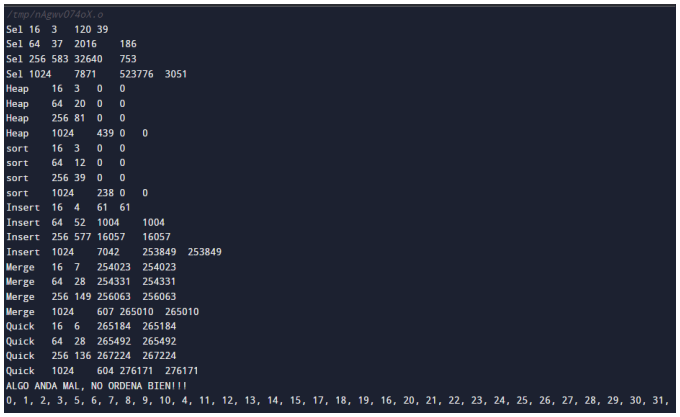

Figura 8. Resultados de los algoritmos de ordenamiento dado un tamaño con su respectivos datos (tiempo de ejecución, escrituras, intercambios) con CUI personal.

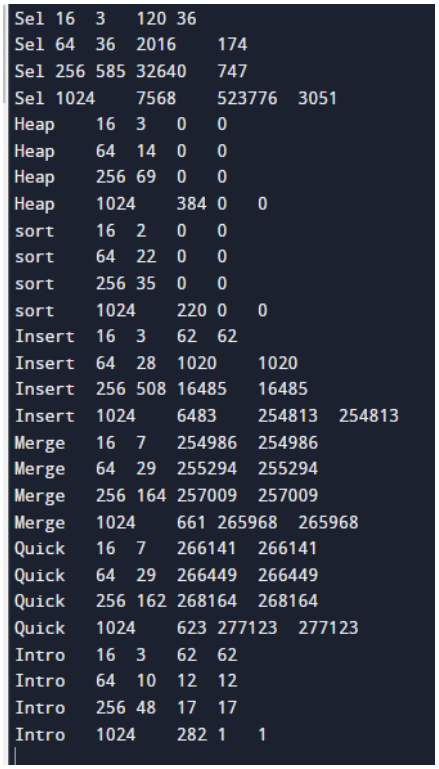

Figura 9. Resultados de los algoritmos de ordenamiento dado un tamaño con su respectivos datos (tiempo de ejecución, escrituras, intercambios) con CUI por defecto (20000000).

Resultados obtenidos mediante un compilador online de C++ (https://www.programiz.com/cpp-programming/online-compiler/)

Referencias

[1] N. Josuttis, C++ Standard Library: A Tutorial and Reference. Pearson Educ. Ltd.

[2] S. S. Skiena, The Algorithm Design Manual. Cham: Springer Int. Publishing, 2020.. [En línea]. Disponible: https://doi.org/10.1007/978-3-030-54256-6 h

[3] Diseño y construcción de algoritmos - 1. edición. Ed. Univ. Del Norte, 2015.

[4] K. Guillén Díaz. “ANÁLISIS Y DISEÑO DE ALGORITMOS I”. ReUNED. [En línea]. Disponible: https://repositorio.uned.ac.cr/bitstream/handle/120809/335/GE50289%20Análisis%20y%20diseño%20de%20algoritmos%20-%202009%20-%20Informática.pdf?sequence=1

[5] A. Dev Mishra y D. Garg. “SELECTION OF THE BEST ALGORITHM”. [En línea]. Disponible: https://d1wqtxts1xzle7.cloudfront.net/28569137/Selection_of_best_sorting_algorithm-libre.pdf?1390874589=&response-content-disposition=inline%3B+filename%3DSelection_of_Best_Sorting_Algorithm.pdf&Expires=1696119920&Signature=Xk6zIYcNjAMz5kE46tSNfekklqe8gEKnfOs2Jptca1nnKLRMOgwqjz-ywO6zB8ppL3cspnpJlmSl6vDCbhNussgQphaj14gGY3Y7w-qYY6XfbYLg-8Vs8e2zmCKZ-wVYyXP3XTbMwQmBceEUiVfThfxId9HoSe-EdR-z31m3vXj62pVYD01fnq~T5BLhejjf3qNNstGgRR02ANgJHI56bH~XaOc0fUubZ9QxS~Fr3FbEKtQwqo1zJ9~-GaOgVWJYRvXpnXs0pV1jla95kLCtUkc0MMuI3nOSe~Ku8IOzxLnfv8Ha2Z9tJUcHTXACP8grVyDnNDmYRuhMoiIgv9jB8Q__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

[6] “The Case for a Learned Sorting Algorithm”, SIGMOD '20: Proc. 2020 ACM SIGMOD Int. Conf. Manage. Data, 2020.

[7] A. Kumari, N. K. Singh y S. Chakraborty, “A Statistical Comparative Study of Some Sorting Algorithms”, Int. J. Found. Comput. Sci. & Technol., vol. 5, n.º 4, pp. 21–29, julio de 2015. [En línea]. Disponible: https://doi.org/10.5121/ijfcst.2015.5403

[8] K. Suleiman Al-Kharabsheh, I. Mahmoud AlTurani, A. Mahmoud Ibrahim AlTurani y N. Imhammed Zanoon. “Review on Sorting Algorithms A Comparative Study”. [En línea]. Disponible: https://d1wqtxts1xzle7.cloudfront.net/83532208/research-paper-lecture-5-bubble-sort-4-libre.pdf?1649488629=&response-content-disposition=inline%3B+filename%3DReview_on_Sorting_Algorithms_A_Comparati.pdf&Expires=1696120446&Signature=GXNdEECqPZf6DIAIUxXTIfWULvHpREzySCYK8y2dW6ena7bSTR1cTJn6gRrXMOrTzCAE2pEEwg6liiYNfRLVFm-JnucllVmdFWoHc4n01AFWK-xmIYuc-OE0hVNlHDMa7uSjV-WNotSOoE9Wso-1Oxm9ot1aSPjFiAec7c9wlZKFVlbSgtk~jiZJlJo~Hw5Fws2efPo8ivwp-stOByEU-tYeFqIIK~if0hEFdxMq5q-Do4xcHRTVp-4C9c5fxgHCzaGUferuUgvUV2SYr88MBAq~qKp7m7TisGG00nZ4f40TsUJ2FuwmdNzWm3c2P~kZst3f~DJJxAfZ~kFSNBKIuw__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA.

Anexos

Anexo 1. Código del programa en C++

```
// Example program
#include <iostream>
#include <algorithm>
#include <random>
#include <ctime>
// #include <iterator>
// #include <numeric>
// #include <cassert>
using namespace std;

/* \ \ \ \ \ / / / / / */
```

```
constexpr int CUI = 20171770; //Reemplace por su propio
CUI
/* / / / / / \ \ \ \ \ */

/* * * * * * * * DECLARACIONES * * * * * * * * */

// Máquina de generación de números aleatorios
mt19937 Rand;

// Variables de conteo estadístico
unsigned long long int cont_comparaciones = 0,
cont_escrituras = 0;

// Medición de tiempo (en microsegundos) desde
transcurrido(true) hasta transcurrido()
typedef uint_fast64_t t_transcurrido; // Definido en
microsegundos
constexpr double __MedidaTiempo = 1000000.0;    // Define
microsegundos
clock_t __inicio_transcurrido = clock();
inline t_transcurrido transcurrido(bool reiniciar);

inline t_transcurrido transcurrido(bool reiniciar = false) {
  if ( reiniciar ) __inicio_transcurrido = clock();
  return
((__MedidaTiempo*double(clock()-__inicio_transcurrido))/do
uble(CLOCKS_PER_SEC));
};

template < class RandomIt >
void SelectionSort ( RandomIt b, RandomIt e ) ;
template < class RandomIt, class Compare >
void SelectionSort ( RandomIt b, RandomIt e, Compare comp
);

template < class RandomIt >
void HeapSort ( RandomIt b, RandomIt e ) ;
template < class RandomIt, class Compare >
void HeapSort ( RandomIt b, RandomIt e, Compare comp );

template < class RandomIt >
void InsertionSort ( RandomIt b, RandomIt e ) ;
template < class RandomIt, class Compare >
void InsertionSort ( RandomIt b, RandomIt e, Compare comp
);

template < class RandomIt >
void MergeSort ( RandomIt b, RandomIt e ) ;
template < class RandomIt, class Compare >
void MergeSort ( RandomIt b, RandomIt e, Compare comp );

template < class RandomIt >
void QuickSort ( RandomIt b, RandomIt e ) ;
template < class RandomIt, class Compare >
void QuickSort ( RandomIt b, RandomIt e, Compare comp );

template < class RandomIt >
void IntroSort ( RandomIt b, RandomIt e ) ;
template < class RandomIt, class Compare >
void IntroSort ( RandomIt b, RandomIt e, Compare comp );
//CLASES EXTRA PARA EL INTROSORT
template <class RandomIt, class Compare>
void IntroSortImpl(RandomIt b, RandomIt e, Compare comp,
int maxDepth);

template <class RandomIt, class Compare>
RandomIt Partition(RandomIt b, RandomIt e, Compare
comp);

typedef void ( *t_ordenamiento ) ( vector<int>::iterator,
vector<int>::iterator ); // tipo de función de ordenamiento

template < t_ordenamiento >
void EvaluaOrdenamiento ();


/* * * * * * * * IMPLEMENTACIONES * * * * * * * * */

template < class RandomIt >
void SelectionSort ( RandomIt b, RandomIt e ) {
  SelectionSort ( b, e, less<decltype(*(b))>() );
}

template < class RandomIt, class Compare >
void SelectionSort ( RandomIt b, RandomIt e, Compare comp
) {
  cont_comparaciones = cont_escrituras = 0;
  for (RandomIt s = b; s < e-1; s++) {
    RandomIt m = s;
    for ( RandomIt i = s+1; i < e; i++ ) {
      cont_comparaciones++;
      if ( comp ( *i, *m ) ) m = i;
    }
    if ( m != s ) {
      cont_escrituras+=3; // swap(a,b) escribe t=a;a=b;b=t;
      swap ( *s, *m );
    }
  }
}

template < class RandomIt >
void HeapSort ( RandomIt b, RandomIt e ) {
  HeapSort ( b, e, less<decltype(*(b))>() );
}

template < class RandomIt, class Compare >
void HeapSort ( RandomIt b, RandomIt e, Compare comp ) {
  cont_comparaciones = cont_escrituras = 0;
  // Utilizando la heap de std
  make_heap ( b, e );
```

```
  sort_heap ( b, e );
}

template < class RandomIt >
void InsertionSort ( RandomIt b, RandomIt e ) {
  InsertionSort ( b, e, less<decltype(*(b))>() );
}

template <class RandomIt, class Compare>
void InsertionSort(RandomIt b, RandomIt e, Compare comp)
{
    cont_comparaciones = cont_escrituras = 0;
    for (RandomIt i = b + 1; i < e; ++i) {
        RandomIt j = i;
        while (j > b && comp(*j, *(j - 1))) {
            cont_comparaciones++;
            cont_escrituras++;
            std::swap(*j, *(j - 1));
            --j;
        }
    }
}


template < class RandomIt >
void MergeSort ( RandomIt b, RandomIt e ) {
  MergeSort ( b, e, less<decltype(*(b))>() );
}

template <class RandomIt, class Compare>
void MergeSort(RandomIt b, RandomIt e, Compare comp) {
    if (std::distance(b, e) <= 1) {
        return;  // La lista ya está ordenada o tiene un solo
elemento
    }

    RandomIt middle = b + std::distance(b, e) / 2;

    // Dividir la lista en dos partes
    MergeSort(b, middle, comp);
    MergeSort(middle, e, comp);

    // Fusionar las dos partes ordenadas
    std::vector<typename RandomIt::value_type>
tmp(std::distance(b, e));
    RandomIt left = b, right = middle, tmp_it = tmp.begin();

    while (left < middle && right < e) {
        cont_comparaciones++;
        if (comp(*left, *right)) {
            cont_escrituras++;
            *tmp_it = *left;
            ++left;
        } else {
            cont_escrituras++;
            *tmp_it = *right;
            ++right;
        }
        ++tmp_it;
    }

    // Copiar los elementos restantes (si los hay)
    tmp_it = std::copy(left, middle, tmp_it);
    tmp_it = std::copy(right, e, tmp_it);

    // Copiar los elementos ordenados de vuelta a la lista
original
    std::copy(tmp.begin(), tmp.end(), b);
}

template < class RandomIt >
void QuickSort ( RandomIt b, RandomIt e ) {
  MergeSort ( b, e, less<decltype(*(b))>() );
}
template <class RandomIt, class Compare>
void QuickSort(RandomIt b, RandomIt e, Compare comp) {
    if (std::distance(b, e) <= 1) {
        return;  // La lista ya está ordenada o tiene un solo
elemento
    }

    RandomIt pivot = b + std::distance(b, e) / 2;
    typename RandomIt::value_type pivot_value = *pivot;

    RandomIt left = b;
    RandomIt right = e - 1;

    while (left <= right) {
        while (comp(*left, pivot_value)) {
            cont_comparaciones++;
            left++;
        }

        while (comp(pivot_value, *right)) {
            cont_comparaciones++;
            right--;
        }

        if (left <= right) {
            cont_escrituras++;
            std::swap(*left, *right);
            left++;
            right--;
        }
    }

    // Llamada recursiva para las dos mitades
    if (b < right) {
        QuickSort(b, right + 1, comp);
    }
```

```
    if (left < e) {
        QuickSort(left, e, comp);
    }
}
// INTROSORT
template <class RandomIt>
void IntroSort(RandomIt b, RandomIt e) {
    IntroSort(b, e, std::less<decltype(*(b))>());
}

template <class RandomIt, class Compare>
void IntroSort(RandomIt b, RandomIt e, Compare comp) {
    cont_comparaciones = cont_escrituras = 0;
    int maxDepth = 2 * log2(std::distance(b, e));
    IntroSortImpl(b, e, comp, maxDepth);
}

template <class RandomIt, class Compare>
void IntroSortImpl(RandomIt b, RandomIt e, Compare comp,
int maxDepth) {
    if (maxDepth <= 0) {
        // Si alcanzamos la profundidad máxima, cambiamos a
Heap Sort
        HeapSort(b, e, comp);
        return;
    }

    if (std::distance(b, e) <= 16) {
        // Si el rango es pequeño, cambiamos a Insertion Sort
        InsertionSort(b, e, comp);
        return;
    }

    RandomIt pivot = Partition(b, e, comp);
    IntroSortImpl(b, pivot, comp, maxDepth - 1);
    IntroSortImpl(pivot + 1, e, comp, maxDepth - 1);
}

template <class RandomIt, class Compare>
RandomIt Partition(RandomIt b, RandomIt e, Compare comp)
{
    RandomIt pivot = b + std::distance(b, e) / 2;
    typename RandomIt::value_type pivot_value = *pivot;

    RandomIt left = b;
    RandomIt right = e - 1;

    while (true) {
        while (comp(*left, pivot_value)) {
            cont_comparaciones++;
            left++;
        }

        while (comp(pivot_value, *right)) {
            cont_comparaciones++;
            right--;
        }

        if (left >= right) {
            return right;
        }

        cont_escrituras++;
        std::swap(*left, *right);
        left++;
        right--;
    }
}

template < t_ordenamiento _ordenamiento >
void EvaluaOrdenamiento ( string s_ordenamiento ) {
  vector<int> V;

  // Unit Test: verifica si ordena bien
  V.resize(32);
  iota ( V.begin(), V.end(), 0 );
  shuffle ( V.begin(), V.end(), Rand );
  _ordenamiento (V.begin(), V.end());
  for(auto i=V.begin()+1;i<V.end();i++) if (*(i-1)!=*i-1) {
    cout << "ALGO ANDA MAL, NO ORDENA BIEN!!!\n";
    for ( auto &i: V ) cout << i << ", ";
    cout << endl;
    exit(1);
  }

  //Inicia pruebas
  // vector<int> T =
{4096,32768,262144,2097152,16777216,134217728,1073741
824};
  //,4096,16384,65536,262144,1048576}
  vector<int> T = {16,64,256,1024};
  Rand.seed(CUI);
  for ( auto t : T ) {
    V.resize(t);
    iota ( V.begin(), V.end(), 0 );
    shuffle ( V.begin(), V.end(), Rand );
    cout << s_ordenamiento << '\t' << t << '\t';
    transcurrido(true);
    _ordenamiento (V.begin(), V.end());
    cout << transcurrido() << '\t';
    cout << cont_comparaciones << '\t';
    cout << cont_escrituras << endl;
  }
}


int main() {
  vector<int> V;

  EvaluaOrdenamiento<&SelectionSort>("Sel");
```

```
  EvaluaOrdenamiento<&HeapSort>("Heap");
  EvaluaOrdenamiento<&sort>("sort");
  EvaluaOrdenamiento<&InsertionSort>("Insert");
  EvaluaOrdenamiento<&MergeSort>("Merge");
  EvaluaOrdenamiento<&QuickSort>("Quick");

  // EvaluaOrdenamiento<&InsertionSort>("Ins");
  // EvaluaOrdenamiento<&MergeSort>("Merge");
  // EvaluaOrdenamiento<&QuickSort>("Quick");
  EvaluaOrdenamiento<&IntroSort>("Intro");

}
```