Algoritmos de Ordenamiento

Andres Murillo Murillo C15424

***Resumen* — En el contexto de la optimización de procesos computacionales, este reporte analiza la complejidad de seis algoritmos de ordenamiento (Selección, Inserción, Mezcla, Montículos, Rápido y de Residuos) implementados en C++. Mediante la evaluación de los tiempos de ejecución en arreglos de hasta 200,000 elementos, se observa que el algoritmo de Residuos no solo supera consistentemente a los demás en eficiencia de tiempo, sino que su ventaja aumenta conforme se incrementa el tamaño del arreglo. Estos hallazgos indican que el algoritmo de Residuos es particularmente adecuado para aplicaciones que manejan grandes volúmenes de datos, ofreciendo fundamentos sólidos para la selección de algoritmos en entornos de alto rendimiento.**

Palabras clave—ordenamiento, selección, inserción, mezcla, Montículos, Rápido y de Residuos

I. Introducción

En este estudio se lleva a cabo un análisis comparativo de varios algoritmos de ordenamiento clave en la informática: Selección (Selection Sort), Inserción (Insertion Sort), Mezcla (Merge Sort), Montículos (Heap Sort), Rápido (Quick Sort) y de Residuos (Radix Sort). Los algoritmos de ordenamiento son procedimientos utilizados para organizar elementos de una lista en un orden específico, ya sea ascendente o descendente. Cada uno de estos algoritmos tiene características únicas que afectan su eficiencia y aplicabilidad en diferentes escenarios. El algoritmo de Selección encuentra el elemento más pequeño de la lista y lo intercambia con el primer elemento, repitiendo este proceso para cada posición en la lista. Inserción construye una lista ordenada de forma incremental, insertando cada nuevo elemento en la posición correcta. Mezcla divide la lista en mitades, las ordena y luego las combina, ofreciendo una complejidad de $\Theta(n\log n)$. Montículos utiliza una estructura de datos llamada montículo para ordenar los elementos con una complejidad similar, mientras que Rápido selecciona un "pivote" y reordena los elementos alrededor de este, siendo muy eficiente. Finalmente, de Residuos ordena números o cadenas por sus dígitos o caracteres, procesando cada posición de menor a mayor.

La selección del algoritmo de ordenamiento adecuado es crucial, especialmente en aplicaciones como sistemas de bases de datos y aplicaciones en tiempo real, donde la eficiencia del procesamiento de datos puede significar la diferencia entre un rendimiento óptimo y retrasos considerables. El propósito de este análisis no es solo examinar estas variaciones, sino también validar la complejidad teórica de los algoritmos mediante su comportamiento en escenarios aplicados específicos. Este enfoque no solo profundiza en cómo las características inherentes de cada método influyen en su eficiencia, sino también en su aplicabilidad en diferentes entornos, tales como sistemas de bases de datos, aplicaciones en tiempo real y software de procesamiento masivo de datos. Además, comprender estas diferencias puede proporcionar una base sólida para optimizar tareas específicas y mejorar el rendimiento de sistemas complejos. Para una exploración más detallada de estos métodos y sus aplicaciones, se recomienda consultar "Introduction to Algorithms" de Thomas H. Cormen et al. (2022), una fuente ampliamente reconocida que proporciona una descripción exhaustiva de los principios y prácticas de los algoritmos de ordenamiento.

II. Metodología

Para alcanzar los objetivos planteados, se implementaron los algoritmos de ordenamiento utilizando C++ en un entorno Windows 11, ejecutándose en un procesador Intel i5 de 12ª generación con 12 núcleos. El código fuente, desarrollado a partir del pseudocódigo en " Introduction to Algorithms " de Cormen et al. (2022), está detallado en los apéndices de este documento. Para evaluar comparativamente la eficiencia de los algoritmos, se generaron arreglos de datos con números enteros aleatorios en tamaños desde 50,000 hasta 200,000 elementos, simulando diferentes magnitudes de datos. Esta metodología permite un análisis exhaustivo del rendimiento de los algoritmos bajo diversas condiciones de carga de datos, ofreciendo una visión integral de su eficacia operativa y además observar su complejidad. La generación de números aleatorios se realizó utilizando la biblioteca "random" de C++, asegurando la variabilidad necesaria para las pruebas. Los tiempos de ejecución se midieron utilizando herramientas específicas, como la biblioteca "chrono", y los resultados fueron promediados de varias ejecuciones para garantizar la precisión. Además, se realizaron pruebas adicionales para verificar la consistencia de los resultados y asegurar que no hubiera influencias externas. Los resultados obtenidos se presentaron en gráficos comparativos y tablas detalladas, proporcionando una visión clara de las diferencias entre los algoritmos.

Cuadro I Tiempo de ejecución de los algoritmos

		Tiempo (ms) *Corrida*					
Algoritmo	Tam. (n)	1	2	3	4	5	Prom.
Selección	50000	978.875	997.458	998.049	1001.36	989.307	993.01
	100000	3898.62	3911.28	4051.78	4117.01	4005.22	3996.78
	150000	8951.81	9031.02	9022.42	9052.77	9232.67	9058.14
	200000	16529.2	16739.1	16391.5	15819.6	16234.2	16342.72
Inserción	50000	629.909	633.523	637.136	629.106	639.433	633.82
	100000	2556.47	2542.43	2567.26	2527.13	2537.96	2546.25
	150000	5746.79	5808.35	5813.27	5799.87	5891.93	5812.04
	200000	10104.1	10171.1	10155	10177.9	10229.9	10167.6
Mezcla	50000	5.296	5.819	5.664	5.248	5.702	5.55
	100000	11.141	11.264	11.254	10.92	11.824	11.40
	150000	17.575	17.52	17.917	17.922	17.384	17.66
	200000	23.431	23.05	25.041	23.902	23.561	23.80
Montículos	50000	9.569	10.317	11.479	11.899	11.523	10.557
	100000	21.864	22.047	22.508	22.128	22.384	22.186
	150000	33.88	35.54	35.571	34.345	35.365	34.940
	200000	47.661	47.728	47.096	46.901	48.033	47.084
Rápido	50000	11.313	10.265	11.017	10.96	11.007	10.912
	100000	23.051	22.679	23.208	22.634	22.437	22.602
	150000	33.669	34.11	35.122	34.905	34.747	34.912
	200000	46.572	46.22	47.378	48.778	47.287	47.047
Residuos	50000	1.036	0.515	1.089	1.029	1.06	0.946
	100000	1.521	2.141	2.103	2.145	2.182	2.018
	150000	3.622	3.227	3.781	3.086	3.638	3.471
	200000	3.566	3.185	4.638	4.715	4.114	4.044

Cada prueba mide el tiempo de ejecución de los algoritmos en milisegundos, enfocándose en su capacidad para ordenar números en secuencia ascendente. El procedimiento consiste en capturar el tiempo justo antes de iniciar el algoritmo, luego ejecutar el proceso de ordenamiento, y finalmente calcular el tiempo transcurrido restando el tiempo inicial del final. Para obtener resultados consistentes y reducir variabilidad, se promedian los tiempos obtenidos de cinco ejecuciones independientes para cada tamaño de conjunto de datos.

III. RESULTADOS

El "Cuadro I" presenta los tiempos de ejecución medidos para los algoritmos de ordenamiento: Selección, Inserción, Mezcla, Montículos, Rápido y de Residuos. Estos tiempos, registrados para distintos tamaños de entrada, varían desde 50,000 hasta 200,000 elementos. Esta recopilación de datos permite evaluar no solo la eficiencia de cada algoritmo, sino también su escalabilidad al manejar incrementos progresivos en el volumen de datos. Los resultados ofrecen una perspectiva clara sobre cómo cada algoritmo se adapta a diferentes magnitudes de datos, proporcionando una base comparativa para determinar la idoneidad de cada método en entornos de procesamiento variados.

El algoritmo de ordenamiento por Selección (Selection Sort) muestra un incremento notable en el tiempo de ejecución a medida que se aumenta el tamaño del arreglo, con una complejidad asintótica de $\Theta(n^2)$ en todos los casos. En las pruebas realizadas, el tiempo necesario para ordenar 50,000 elementos fue de 993.01 ms, mientras que para 200,000 elementos, el tiempo de ejecución ascendió a 16,342.72 ms. Estos resultados confirman que el tiempo necesario para completar la ordenación crece significativamente con el aumento del tamaño del arreglo, coherente con la complejidad cuadrática esperada del algoritmo.

Considerando el algoritmo de ordenamiento por Inserción (Insertion Sort), cuya complejidad asintótica en el peor caso es $\Theta(n^2)$, muestra un comportamiento cuadrático en sus tiempos de ejecución. Este inicia con un tiempo promedio de 639.433 ms para arreglos de 50,000 elementos y asciende a 10,167.6 ms para los de 200,000 elementos. Este patrón de crecimiento cuadrático está en línea con las expectativas teóricas y revela limitaciones en contextos de grandes volúmenes de datos, resultando en una menor eficiencia para tales situaciones. Este comportamiento subraya su idoneidad para conjuntos de datos más pequeños donde su sencillez puede ser ventajosa, pero señala limitaciones cuando se manejan volúmenes mayores.

En cambio, el algoritmo de ordenamiento por Mezcla (Merge Sort) se caracteriza por una complejidad teórica constante de $\Theta(n\log n)$ en todos los casos: mejor, peor y promedio. Esta uniformidad se manifiesta en un incremento de tiempo de ejecución más gradual y controlado. Registrando tiempos promedio de solo 5.55 ms para ordenar arreglos de 50,000 elementos y aumentando a 23.80 ms para los de 200,000 elementos, el algoritmo de Mezcla demuestra mantener un alto rendimiento incluso al escalar a grandes cantidades de datos. Esto lo convierte en una opción significativamente eficaz para manejar grandes volúmenes de información.

El algoritmo de ordenamiento por Montículos (Heap Sort) exhibe un crecimiento controlado en los tiempos de ejecución, en consonancia con su complejidad teórica constante de $\Theta(n\log n)$ en todos los escenarios. Iniciando con un tiempo promedio de 10.557 ms para arreglos de 50,000 elementos, se registra un aumento hasta 47.084 ms para arreglos de 200,000 elementos. Este incremento moderado en los tiempos de ejecución establece al algoritmo de Montículos como una opción eficiente para manejar grandes volúmenes de datos, demostrando su buena escalabilidad y evitando los incrementos exponenciales característicos de los algoritmos con complejidad cuadrática.

Por su parte, el algoritmo de ordenamiento Rápido (Quick Sort), aunque altamente eficiente en una amplia variedad de contextos, registra tiempos de ejecución que comienzan en 10.912 ms para arreglos de 50,000 elementos y aumentan hasta 47.047 ms para arreglos de 200,000 elementos. A pesar de su complejidad promedio de $O(n\log n)$, en el peor caso este algoritmo puede escalar a $\Theta(n^2)$, especialmente bajo

condiciones adversas derivadas de la distribución inicial de los datos. No obstante, sigue siendo una opción robusta y eficaz para la mayoría de las aplicaciones de ordenamiento, demostrando adaptabilidad y eficiencia en diversos escenarios.

Finalmente, el algoritmo de Residuos (Radix Sort) destaca por su rendimiento excepcionalmente alto y tiempos de ejecución consistentemente bajos, registrando un promedio de 0.946 ms para arreglos de 50,000 elementos y apenas alcanzando 4.044 ms para arreglos de 200,000 elementos. Este comportamiento se debe a su complejidad de $O(k\cdot(n+b))$, adecuada para cuando n es el número de elementos y b representa la base numérica usada, lo que lo hace ideal para datos con un rango limitado de valores posibles o donde la longitud de los dígitos es controlada. Este algoritmo mantiene su eficiencia sin verse significativamente afectado por incrementos en el tamaño del arreglo, haciéndolo extremadamente eficiente para grandes volúmenes de datos con baja variabilidad numérica.

Los datos del "Cuadro I" demuestran variaciones significativas en la adaptabilidad de los algoritmos de ordenamiento al manejar grandes volúmenes de datos. Los algoritmos de Mezcla y Montículos, con complejidades de $\Theta(n\log n)$ así como el algoritmo de Residuos con su eficiencia en contextos de baja variabilidad numérica, muestran un rendimiento excepcional en escalabilidad y control del tiempo de ejecución. En contraste, los algoritmos de Selección e Inserción enfrentan incrementos sustanciales en tiempos de ejecución con el aumento del tamaño de los datos, destacando su utilidad en conjuntos más pequeños debido a su simplicidad. Aunque el algoritmo de ordenamiento Rápido es generalmente eficaz, su rendimiento puede degradarse significativamente bajo condiciones adversas. Esta diversidad en el comportamiento enfatiza la importancia de elegir el algoritmo adecuado basado en las especificidades del conjunto de datos y las exigencias del entorno de procesamiento.

Para complementar el análisis cuantitativo presentado, a continuación, se incluyen una serie de gráficos que ilustran visualmente el comportamiento de los algoritmos de ordenamiento bajo diferentes condiciones de carga de datos. Estos gráficos permiten apreciar de manera más directa y comparativa cómo varían los tiempos de ejecución en función del tamaño de los arreglos, destacando las diferencias en eficiencia y escalabilidad entre cada método. Esto proporciona una perspectiva más clara y accesible del rendimiento relativo de los algoritmos en distintos escenarios operativos.

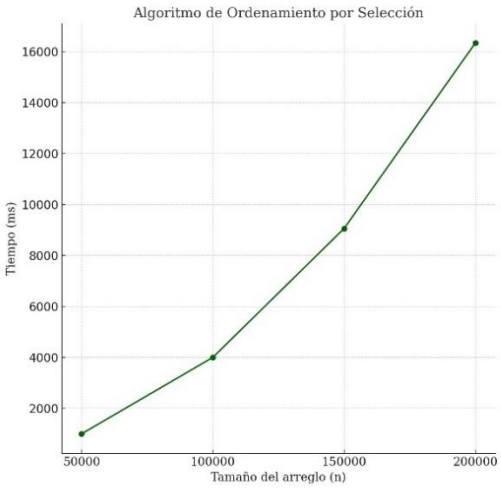

Figura *1* Tiempos promedios de ejecución Algoritmo de ordenamiento por Selección

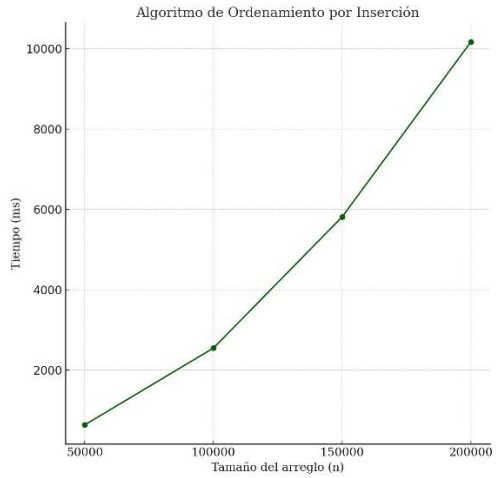

Figura 2 Tiempos promedio de ejecución Algoritmo de ordenamiento por Inserción

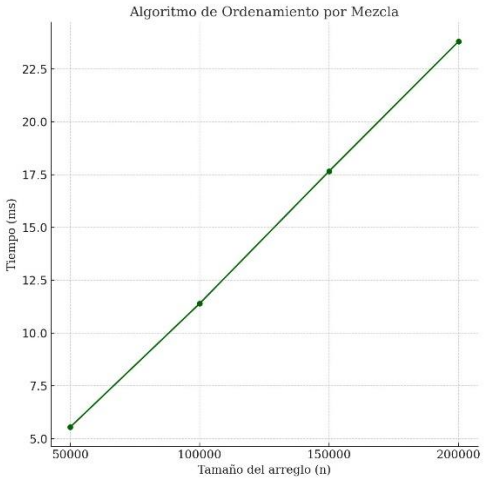

Figura *3* Tiempos promedios de ejecución Algoritmo de ordenamiento por Mezcla

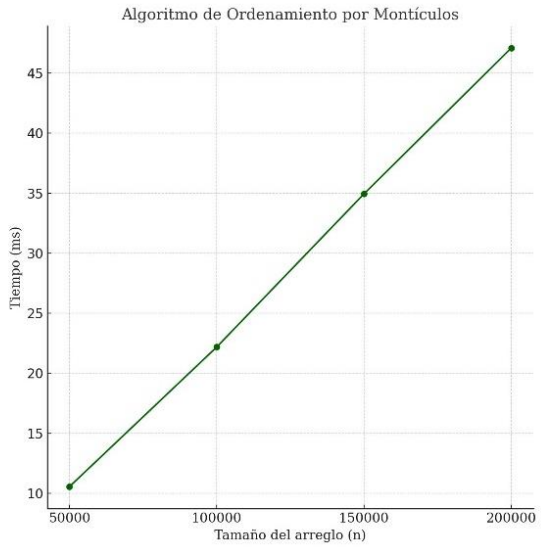

Figura *4* Tiempos promedios de ejecución Algoritmo de ordenamiento por Montículos.

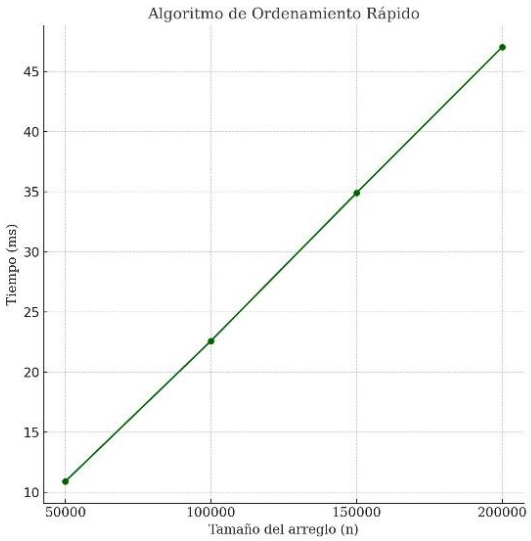

Figura *5* Tiempos promedios de ejecución Algoritmo de ordenamiento Rápido.

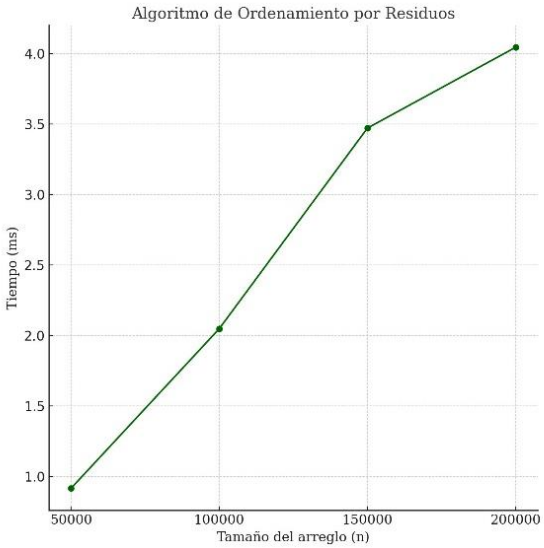

Figura *6* Tiempos promedios de ejecución Algoritmo de ordenamiento por Residuos.

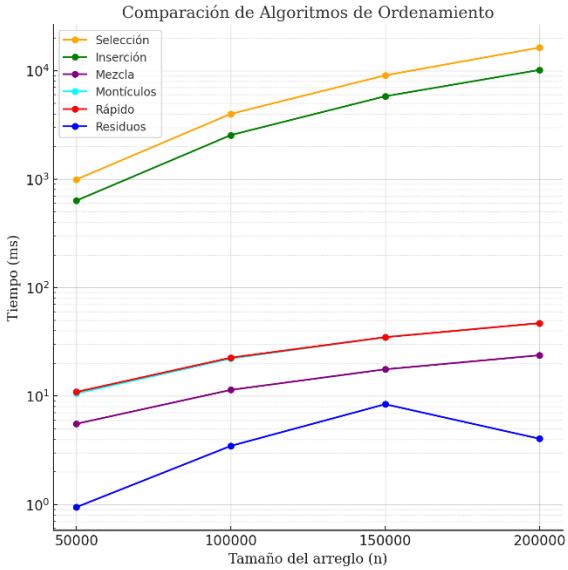

Figura *7* Comparación con escala logarítmica de los algoritmos de Ordenamiento.

La Figura 7 muestra el tiempo de ejecución de seis algoritmos de ordenamiento, proporcionando una comparativa de su rendimiento. Destaca el algoritmo de Mezcla por su eficiencia y estabilidad, mostrando un crecimiento casi lineal en el tiempo de ejecución, lo que lo hace adecuado para grandes volúmenes de datos. Los algoritmos de Montículos y Residuos también presentan buen rendimiento, especialmente el de Residuos que mantiene tiempos bajos consistentemente. En contraste, los algoritmos de Selección e Inserción exhiben un crecimiento cuadrático, resultando menos eficaces para grandes conjuntos de datos. El algoritmo Rápido, aunque más rápido para arreglos menores, también experimenta un aumento en el tiempo con arreglos más grandes, pero sigue siendo competitivo frente a Selección e Inserción.

IV. Conclusiones

Los resultados obtenidos de la comparación entre los algoritmos de Selección, Inserción, Mezcla, Montículos, Rápido y Residuos revelan diferencias significativas en su capacidad para escalar con tamaños de entrada de 50,000 a 200,000 elementos, reflejando directamente sus complejidades teóricas. Los algoritmos de Selección e Inserción, con una complejidad

teórica de $\Theta(n^2)$ en el peor caso, exhiben un incremento exponencial en los tiempos de ejecución, lo cual limita su utilidad a conjuntos de datos menores donde su simplicidad puede ser beneficiosa. En contraste, los algoritmos de Mezcla y Montículos, ambos con complejidades de $\Theta(n\log n)$, muestran un incremento controlado de los tiempos de ejecución, lo que los convierte en opciones robustas y eficientes para manejar grandes volúmenes de datos. El algoritmo Rápido, aunque eficiente en la mayoría de los casos, puede degradar su rendimiento bajo configuraciones de datos adversas, acercándose a $\Theta(n^2)$ en el peor de los casos. Por otro lado, el algoritmo de Residuos destaca por su rendimiento excepcionalmente alto y tiempos de ejecución consistentemente bajos, debido a su complejidad lineal $O(k\cdot(n+b))$, lo que lo hace ideal para datos con un rango limitado de valores o donde la longitud de los dígitos es controlada. Estos resultados subrayan la importancia de seleccionar un algoritmo de ordenamiento adecuado basado en las características específicas del conjunto de datos, y cómo la teoría de la complejidad algorítmica se refleja en el rendimiento práctico, optimizando así el rendimiento en entornos de procesamiento de datos reales.

V. Referencias

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. Introduction to Algorithms, IV ed. MIT Press, 2022.

Andres Murillo Murillo

Estudiante de Ingeniería en Computacion en la Universidad de Costa Rica.

VI. Apéndice A

Código de los Algoritmos

Algoritmo 1 El algoritmo de ordenamiento por selección perfecciona la organización de una colección de elementos a través de un proceso iterativo que sistemáticamente identifica el elemento mínimo dentro del segmento aún no ordenado. En cada iteración, este elemento es trasladado al inicio de la sección ya ordenada. Este procedimiento se repite sucesivamente, avanzando el límite entre las partes ordenada y no ordenada, hasta que se alcanza un estado en el que toda la serie de datos se encuentra secuencialmente organizada.

```
void Ordenador::seleccion(int *A, int n) {
  int m;
  for (int i = 0; i < n - 1; i++){
    m = i;
    for (int j = i + 1 ; j < n; j++){
      if (A[j] < A[m]) {
        m=j;
      }
    }
    std::swap(A[i], A[m]);
  }
}
```

Algoritmo 2 El algoritmo de ordenamiento por inserción trabaja seleccionando secuencialmente cada elemento de la porción no ordenada del conjunto de datos y lo incorpora en la ubicación exacta dentro de la subsecuencia ya organizada. Este método asegura que, tras cada inserción, la sección ordenada se expanda manteniendo su orden hasta que todo el conjunto esté sistemáticamente ordenado. Con cada operación, el algoritmo compara el elemento seleccionado con los precedentes, desplazándolos si es necesario, para encontrar la posición precisa donde el elemento debe ser ubicado, garantizando así la continuidad de la ordenación.

```
void Ordenador::insercion(int *A, int n) {
  int key;
  int i;
  for (int j = 1; j < n; j++) {
    key = A[j];
    i = j - 1;
    while (i >= 0 && A[i] > key) {
      A[i + 1] = A[i];
      i = i - 1;
    }
    A[i + 1] = key;
  }
}
```

Algoritmo 3 El algoritmo de ordenamiento por mezcla adopta un enfoque dividir para conquistar, dividiendo el arreglo original en mitades más manejables. Estas subsecciones se ordenan de manera independiente mediante llamadas recursivas. Una vez que cada mitad está ordenada, el algoritmo procede a fusionarlas meticulosamente, ensamblando así el conjunto completo en un arreglo finalmente ordenado. Este proceso garantiza que los elementos dispersos se reintegren en una secuencia coherente y organizada.

```
void Ordenador::merge(int *A, int p, int q, int r) {
  int n1 = q - p + 1;
  int n2 = r - q;
  int L[n1], R[n2];

  for (int i = 0; i < n1; i++)
    L[i] = A[p + i];
  for (int j = 0; j < n2; j++)
    R[j] = A[q + 1 + j];

  int i = 0, j = 0, k = p;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      A[k++] = L[i++];
    } else {
      A[k++] = R[j++];
    }
  }

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

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

void Ordenador::mergeSortAux(int *A, int p, int r) {
  if (p < r) {
    int q = (p + r) / 2;
    mergeSortAux(A, p, q);
    mergeSortAux(A, q + 1, r);
    merge(A, p, q, r);
  }
}

void Ordenador::mergesort(int *A, int n) {
  mergeSortAux(A, 0, n - 1);
}
```

Algoritmo 4 Heap Sort es un algoritmo de ordenamiento que organiza los elementos de una lista convirtiéndola en un heap máximo. Un heap máximo es una estructura de datos en la que el valor de cada nodo padre es mayor o igual que los valores de sus hijos. El algoritmo comienza construyendo un heap máximo a partir de los elementos dados. Luego, extrae repetidamente el elemento máximo del heap (ubicado en la raíz) y lo coloca al final de la lista, reduciendo así el tamaño del heap en uno. Este proceso se repite hasta que todos los elementos hayan sido extraídos del heap y la lista esté completamente ordenada en orden ascendente.

```
void Ordenador::maxHeapify(int *A, int i, int n) {
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;

  if (l < n && A[l] > A[largest]) {
    largest = l;
  }

  if (r < n && A[r] > A[largest]) {
    largest = r;
  }

  if (largest != i) {
```

```
    std::swap(A[i], A[largest]);
    maxHeapify(A, largest, n);
  }
}

void Ordenador::buildMaxHeap(int *A, int n) {
  int startIdx = (n / 2) – 1;

  for (int i = startIdx; i >= 0; i--) {
    maxHeapify(A, i, n);
  }
}

void Ordenador::heapsort(int *A, int n) {
  buildMaxHeap(A, n);
  for (int i = n - 1; i >= 0; i--) {
    std::swap(A[0], A[i]);
    maxHeapify(A, 0, i);
  }
}
```

Algoritmo 5 Quick Sort es un algoritmo de ordenamiento eficiente que sigue el enfoque de "divide y conquista". Selecciona un elemento como pivote y particiona la lista de manera que los elementos menores que el pivote estén a su izquierda y los mayores estén a su derecha. Luego, aplica recursivamente el mismo proceso a las sublistas generadas hasta que la lista esté ordenada.

```
void Ordenador::quickSortAux(int *A, int p, int r) {
  if (p < r) {
    int q = partition(A, p, r);
    quickSortAux(A, p, q - 1);
    quickSortAux(A, q + 1, r);
  }
}

int Ordenador::partition(int *A, int p, int r) {
  int x = A[r];
  int i = p - 1;

  for (int j = p; j < r; j++) {
    if (A[j] <= x) {
      i++;
      std::swap(A[i], A[j]);
    }
  }
  std::swap(A[i + 1], A[r]);
  return i + 1;
}

void Ordenador::quicksort(int *A, int n) {
  quickSortAux(A, 0, n - 1);
}
```

Algoritmo 6 Radix Sort es un algoritmo de ordenamiento que organiza los elementos de una lista basándose en sus dígitos individuales. Utiliza un enfoque de ordenamiento estable, como Counting Sort, para clasificar los elementos en cada iteración, comenzando por el dígito menos significativo y avanzando hacia los dígitos más significativos. Este proceso se repite hasta que todos los dígitos hayan sido considerados, resultando en una lista completamente ordenada.

```
void Ordenador::countSort(int *A, int n, int exp, int base) {
  std::vector<int> output(n);
  std::vector<int> count(base, 0);

  for (int i = 0; i < n; i++) {
    int index = (A[i] / exp) % base;
    count[index]++;
  }

  for (int i = 1; i < base; i++) {
    count[i] += count[i - 1];
```

```
  }

  for (int i = n - 1; i >= 0; i--) {
    int index = (A[i] / exp) % base;
    output[count[index] - 1] = A[i];
    count[index]--;
  }

  for (int i = 0; i < n; i++) {
    A[i] = output[i];
  }
}

void Ordenador::radixSortAux(int *A, int n) {
  int m = *std::max_element(A, A + n);
  int base = std::pow(2, std::ceil(std::log2(n)));

  for (int exp = 1; m / exp > 0; exp *= base) {
    countSort(A, n, exp, base);
  }
}
void Ordenador::radixsort(int *A, int n) {
  radixSortAux(A, n);
}
```