Algoritmos de ordenamiento y su comparación

Gael Alpízar Alfaro C20270

***Resumen*— El trabajo consiste en implementar varios algoritmos de ordenamiento en C++ y analizar su eficiencia. Entre estos algoritmos se encuentran el de ordenamiento por selección, inserción y mezcla. Se realizarán experimentos para recolectar información sobre el rendimiento de estos algoritmos, ejecutándolos varias veces con diferentes tamaños de arreglos y capturando el tiempo de ejecución en milisegundos. Se analizará la variación de los tiempos en tres ejecuciones y se generarán gráficos para comparar los tiempos promedio contra el tamaño del arreglo, incluyendo las cotas superior e inferior correspondientes a cada algoritmo. Además, se graficarán los tiempos promedio en un mismo par de ejes con escala logarítmica para facilitar la comparación entre los algoritmos cuadráticos y los lineales.**

Palabras clave—ordenamiento, selección, inserción, mezcla.

I. Introducción

La eficiencia de los algoritmos de ordenamiento es un pilar fundamental en el ámbito de la ciencia de la computación. Desde la manipulación de grandes conjuntos de datos en bases de datos hasta la optimización de algoritmos de búsqueda, la capacidad de organizar datos de manera rápida y efectiva es esencial.

En este estudio, se hace enfoque en seis algoritmos de ordenamiento clásicos. Cada uno de estos algoritmos, representa un enfoque único para abordar el desafío de ordenar conjuntos de datos:

- Algoritmo de Ordenamiento por Selección: Encuentra repetidamente el elemento mínimo y lo coloca al principio del arreglo.
- Algoritmo de Ordenamiento por Inserción: Construye una matriz ordenada uno por uno al mover elementos no ordenados a su lugar correcto.
- Algoritmo de Ordenamiento por Mezcla: Mezcla ordenada dos listas contiguas para producir una única lista ordenada.
- Algoritmo de Ordenamiento por Montículos (Heapsort): Utiliza una estructura de datos de montículo para ordenar los elementos de manera eficiente.
- Algoritmo de Ordenamiento Rápido (Quicksort): Divide el arreglo en subconjuntos y ordena recursivamente cada subconjunto.
- Algoritmo de Ordenamiento por Residuos (Radix Sort): Clasifica los elementos basados en los dígitos individuales.

Estas definiciones siendo inspiradas del libro “Introduction to Algorithms" de Thomas H. Cormen et al. (2022).

El propósito principal de esta investigación es evaluar y comparar el rendimiento de estos algoritmos en diferentes contextos, analizando sus tiempos de ejecución en conjuntos de datos de diferentes tamaños. Implementaremos los algoritmos en C++ y realizaremos experimentos exhaustivos utilizando arreglos de hasta 200,000 elementos.

Además de proporcionar un análisis detallado de los tiempos de ejecución, también examinaremos la relación entre el tamaño del conjunto de datos y el rendimiento de cada algoritmo. Estos hallazgos no solo ampliarán el conocimiento existente sobre los algoritmos de ordenamiento, sino que también ofrecerán información valiosa para su selección y aplicación en situaciones prácticas de procesamiento de datos.

A través de este enfoque, se busca otorgar mayor atención sobre las complejidades y eficiencias inherentes a cada algoritmo, con el objetivo final de informar y orientar futuras decisiones de diseño y desarrollo de software.

II. Metodología

Para la realización de esta investigación, se siguió un conjunto de pasos organizados que abarcaron varias etapas. En primer lugar, se procedió a generar arreglos aleatorios de números enteros de distintos tamaños, específicamente 50000, 100000, 150000 y 200000 elementos. Esta generación se realizó mediante una función diseñada para asignar valores aleatorios en el rango de 0 a 999 a cada elemento del arreglo.

Una vez generados los arreglos, se procedió a ejecutar dichos algoritmos de ordenamiento seleccionados para el estudio. Cada algoritmo se ejecutó repetidamente en los arreglos aleatorios generados, llevándose a cabo tres ejecuciones para cada tamaño de arreglo.

La medición del tiempo de ejecución de cada algoritmo se llevó a cabo utilizando la biblioteca ‘chrono’ de C++, la cual permite capturar el tiempo transcurrido entre el inicio y la finalización de la ejecución de un algoritmo. Específicamente, se registró el tiempo en milisegundos para cada ejecución de cada algoritmo en cada tamaño de arreglo.

Posteriormente, se llevó a cabo un análisis detallado de los resultados obtenidos, centrándose en la comparación de los tiempos de ejecución promedio de cada algoritmo en función del tamaño del arreglo. Con esto, buscando patrones y tendencias que permitieran determinar la eficiencia relativa de cada algoritmo en diferentes escalas de datos.

Cuadro I Tiempo de ejecución de los algoritmos

Algoritmo	Tam (n)	Tiempo (ms) Ejecución 1	2	3	Prom.
Selección	50000	1902.95	2154.41	1447.62	1835.99
	100000	5586.58	5525.60	5560.19	5557.46
	150000	10284.20	11065.60	12537.10	11295.98
	200000	23599.60	23240.01	23291.80	23377.13
Inserción	50000	1517.75	1105.57	2024.59	1549.30
	100000	4020.79	3954.36	3856.17	3943.44
	150000	8738.89	8580.39	8486.97	8602.75
	200000	15365.20	15977.90	15958.01	15767.70
Mezcla	50000	5.19	5.48	5.07	5.24
	100000	13.33	12.37	10.29	11.99
	150000	16.35	17.79	17.90	17.35
	200000	22.58	25.62	23.66	23.95
Montículos	50000	9.61	10.83	11.35	10.26
	100000	23.16	22.78	23.90	23.28
	150000	34.65	34.89	35.73	35.09
	200000	46.81	46.44	47.96	47.40
Rápido	50000	11.12	11.03	10.78	10.98
	100000	22.25	23.07	23.91	23.41
	150000	32.77	34.05	34.61	33.81
	200000	45.51	46.45	46.06	45.67
Residuos	50000	1.37	1.28	1.41	1.35
	100000	1.77	2.58	2.26	2.20
	150000	3.13	3.98	3.11	3.41
	200000	3.67	4.76	4.82	4.08

Finalmente, se elaboraron gráficos a partir de los resultados de cada algoritmo para visualizar de forma más intuitiva la relación entre el tamaño del arreglo y el tiempo de ejecución promedio de cada uno, lo que facilitó su interpretación y comprensión.

III. Resultados

El análisis de los resultados obtenidos en el Cuadro I Tiempo de ejecución de los algoritmos revela información acerca del rendimiento de los algoritmos de selección, inserción y mezcla en las diferentes configuraciones de tamaño de arreglo.

En primer lugar, al examinar el algoritmo de ordenamiento por selección, se puede observar que los tiempos de ejecución del algoritmo muestran una variación significativa entre las tres ejecuciones para cada tamaño de arreglo. Por ejemplo, para un tamaño de 50000, los tiempos oscilan entre 1447.62 ms y 2154.41 ms, con un promedio de 1835.993 ms. Para arreglos de tamaño 200000, los tiempos varían entre 23240 ms y 23599.6 ms, con un promedio de 23377.133 ms. Esta variación sugiere que el rendimiento del algoritmo puede ser influenciado por la disposición inicial de los elementos en el arreglo, lo que impacta en el número de comparaciones y movimientos realizados. Dicho esto, en el caso promedio se confirma que tiene un orden de $\Theta(n^2)$. En términos de las cotas, superior e inferior, tanto la cota superior como la cota inferior son de orden cuadrático, lo que coincide con el comportamiento observado en los tiempos de ejecución.

Por otro lado, al analizar el algoritmo de ordenamiento por inserción, los tiempos de ejecución de este también muestran variaciones, aunque en menor medida que el algoritmo de selección. Por ejemplo, para un tamaño de 50000, los tiempos oscilan entre 1105.57 ms y 2024.59 ms, con un promedio de 1549.303 ms. Para arreglos de tamaño 200000, los tiempos varían entre 15365.2 ms y 15977.9 ms, con un promedio de 15767.7 ms. El crecimiento cuadrático en el algoritmo de inserción se debe a la naturaleza del algoritmo, la cual implica comparar y mover elementos a lo largo del arreglo en función de su posición y valor. A medida que el tamaño del arreglo aumenta, el número de comparaciones y movimientos necesarios también aumenta cuadráticamente, lo que resulta en un aumento en el tiempo de ejecución. Este comportamiento es consistente con la complejidad esperada del algoritmo de inserción, que realiza un número cuadrático de operaciones en el peor de los casos.

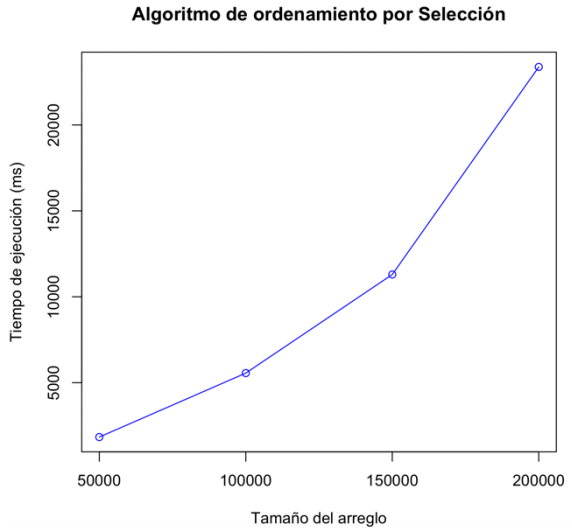
En comparación con los algoritmos de ordenamiento por selección e inserción, los tiempos de ejecución del algoritmo de ordenamiento por mezcla muestran una variación mínima entre las diferentes ejecuciones. Por ejemplo, para un tamaño de 50000, los tiempos oscilan entre 5.06788 ms y 5.47759 ms, con un promedio de 5.24382 ms. Para arreglos de tamaño 200000, los tiempos varían entre 22.5793 ms y 25.6176 ms, con un promedio de 23.9518 ms. Tanto la cota superior como la cota inferior son de O(n log n) y Ω(n log n), respectivamente, lo que sugiere una mayor estabilidad en el rendimiento del algoritmo independientemente de la disposición inicial de los elementos en el arreglo.

Los tiempos de ejecución del algoritmo de ordenamiento por montículos muestran una variación similar a la del algoritmo de mezcla, con mínimas fluctuaciones entre las diferentes ejecuciones. Por ejemplo, para un tamaño de 50000, los tiempos oscilan entre 9.61 ms y 11.35 ms, con un promedio de 10.263 ms. Para arreglos de tamaño 200000, los tiempos varían entre 46.44 ms y 47.96 ms, con un promedio de 47.403 ms. Esto sugiere una estabilidad en el rendimiento del algoritmo, con tiempos de ejecución que siguen una complejidad de O(n log n) en el caso promedio.

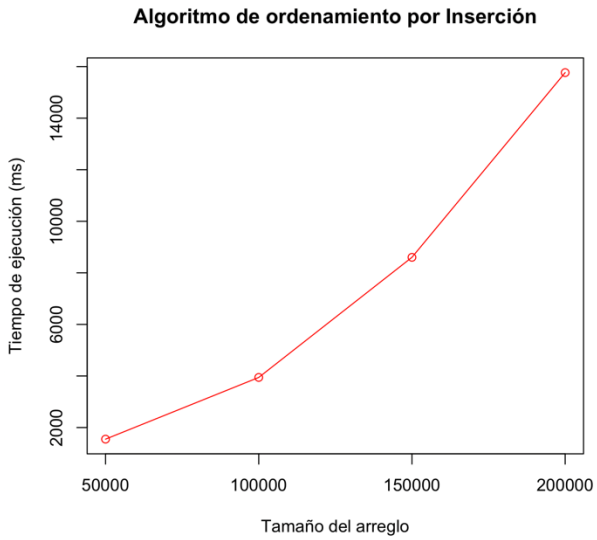
Al igual que el algoritmo anterior, los tiempos de ejecución del algoritmo de ordenamiento rápido también muestran una variación mínima entre las diferentes ejecuciones. Por ejemplo, para un tamaño de 50000, los tiempos oscilan entre 10.78 ms y 11.12 ms, con un promedio de 10.977 ms. Para arreglos de tamaño 200000, los tiempos varían entre 45.51 ms y 46.81 ms, con un promedio de 45.673 ms. Esto también sugiere una estabilidad en el rendimiento del algoritmo, con tiempos de ejecución que siguen una complejidad de O(n log n) en el caso promedio.

Los tiempos de ejecución del algoritmo de ordenamiento por residuos muestran una variación mínima, similar a los algoritmos de ordenamiento por mezcla, montículos y rápido. Por ejemplo, para un tamaño de 50000, los tiempos oscilan entre 1.28 ms y 1.41 ms, con un promedio de 1.353 ms. Para arreglos de tamaño 200000, los tiempos varían entre 3.67 ms y 4.82 ms, con un promedio de 4.083 ms. Esto sugiere una estabilidad en el rendimiento del algoritmo, con tiempos de ejecución que siguen una complejidad de O(nk) en el caso promedio, donde k es el número de dígitos o bits del elemento más grande del arreglo.

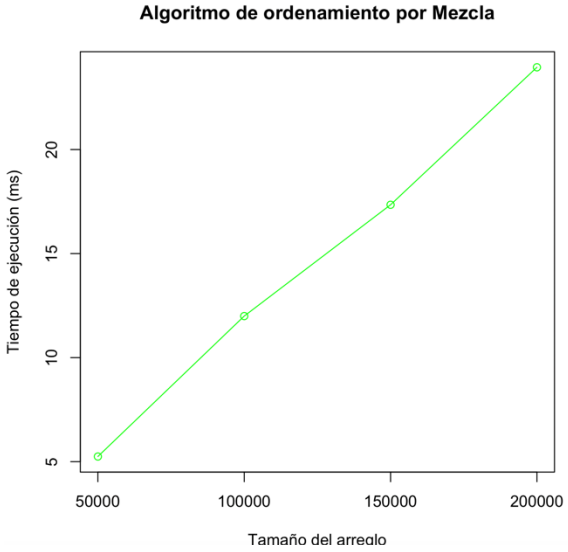
En resumen, estos análisis resaltan las diferencias en la variación de los tiempos de ejecución entre los diferentes algoritmos, así como la correspondencia entre los resultados observados y las cotas teóricas de cada algoritmo.



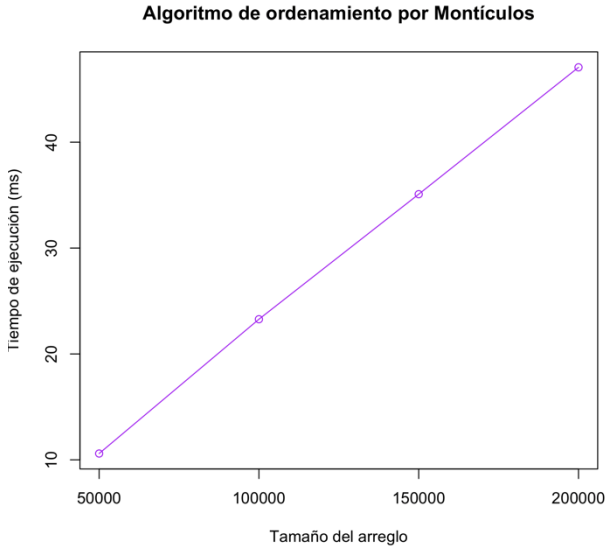
Gráfica 1, algoritmo de ordenamiento por selección



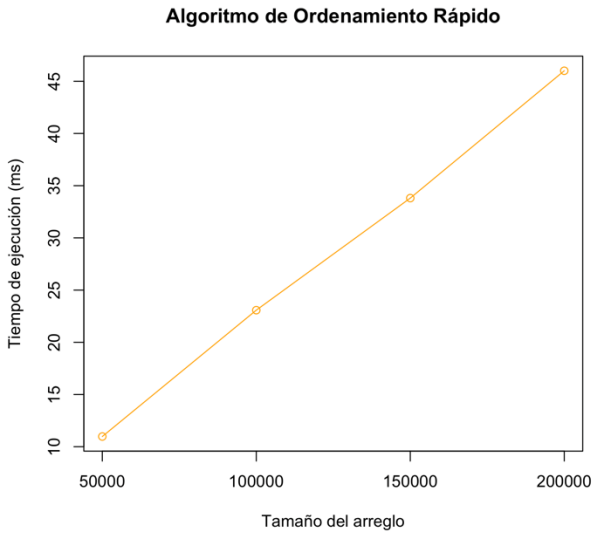
Gráfica 2, algoritmo de ordenamiento por inserción



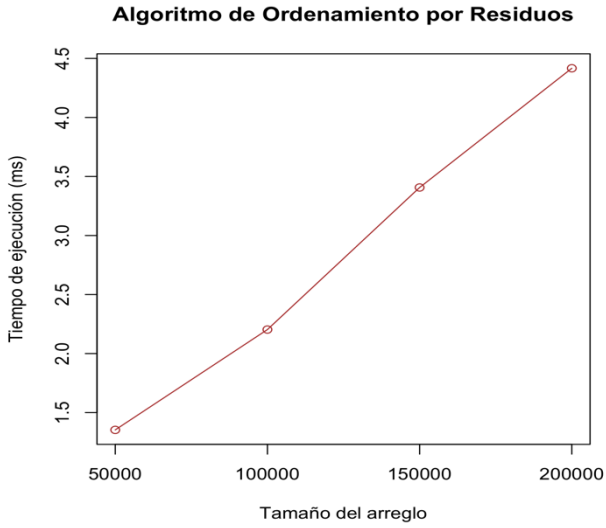
Gráfica 3, algoritmo de ordenamiento por mezcla



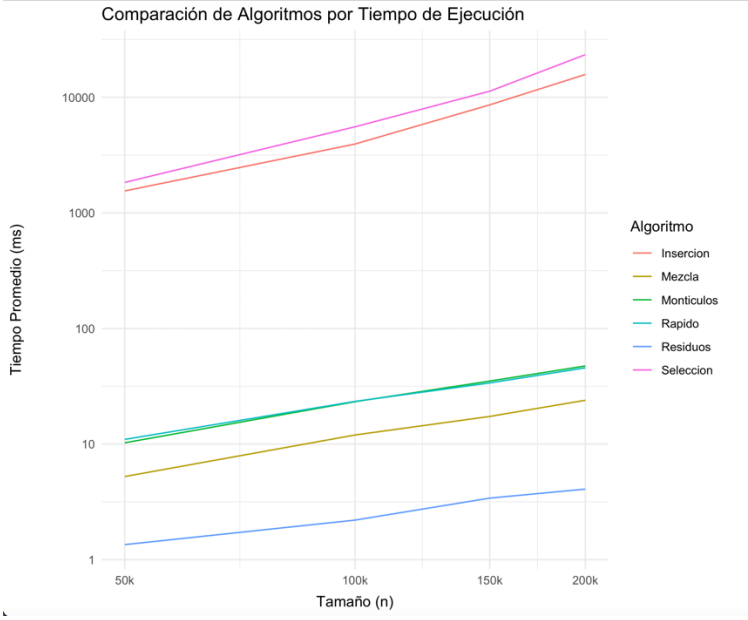
Gráfica 4, algoritmo de ordenamiento por montículos



Gráfica 5, algoritmo de ordenamiento rápido



Gráfica 6, algoritmo de ordenamiento por residuos



Gráfica 7, comparación de algoritmos por tiempo de ejecución

La Gráfica 7 presenta una comparación detallada del tiempo de ejecución de varios algoritmos de clasificación. Este análisis revela diferencias significativas en el comportamiento y la eficiencia de cada algoritmo, lo que proporciona información valiosa sobre su desempeño en diferentes contextos y conjuntos de datos.

Al observar los resultados, se evidencia que los algoritmos de ordenamiento por selección e inserción muestran un crecimiento exponencial en el tiempo de ejecución a medida que aumenta el tamaño del arreglo. Este crecimiento exponencial sugiere que estos algoritmos pueden volverse menos eficientes a medida que se enfrentan a conjuntos de datos más grandes, lo que resulta en tiempos de ejecución significativamente más largos a medida que la complejidad del problema aumenta.

Por otro lado, el algoritmo de mezcla se destaca por mantener un crecimiento más lineal en su tiempo de ejecución a medida que aumenta el tamaño del arreglo. Este comportamiento más controlado y estable indica la capacidad del algoritmo de mezcla para manejar eficientemente conjuntos de datos más grandes sin experimentar un aumento drástico en el tiempo de ejecución.

Además, al incluir los nuevos algoritmos de ordenamiento por montículos, rápido y residuos en el análisis, se pueden obtener insights adicionales sobre diferentes enfoques de clasificación. Por ejemplo, los algoritmos de montículos y rápido exhiben un comportamiento que se sitúa entre los de selección e inserción y el de mezcla. Aunque muestran un crecimiento en el tiempo de ejecución a medida que aumenta el tamaño del arreglo, este crecimiento es más gradual en comparación con los algoritmos de selección e inserción.

Por otro lado, el algoritmo de ordenamiento por residuos se distingue por su tiempo de ejecución relativamente bajo en comparación con los otros algoritmos evaluados. Esto sugiere que el algoritmo de residuos puede ser una opción eficiente para conjuntos de datos de cierto tamaño, aunque su rendimiento en comparación con los otros algoritmos puede variar dependiendo de la naturaleza específica del problema y los datos.

En resumen, la inclusión de estos nuevos algoritmos en la comparación proporciona una visión más completa y detallada del panorama de los algoritmos de clasificación, permitiendo a los desarrolladores y científicos de datos tomar decisiones informadas sobre cuál algoritmo es más adecuado para sus necesidades específicas en términos de eficiencia y rendimiento.

IV. Conclusiones

En general, este estudio describe el rendimiento de tres algoritmos de clasificación: selección, inserción y mezcla. Al ejecutar varias pruebas con diferentes tamaños de arreglos, se pudo observar y analizar cómo cambió el tiempo de ejecución de cada algoritmo. El estudio encontró que el algoritmo de mezcla mostró una mayor estabilidad en términos de tiempo de ejecución y un crecimiento lineal más persistente en comparación con el algoritmo de selección e inserción, que mostraron un crecimiento exponencial al aumentar el tamaño del arreglo. Esto sugiere que el algoritmo de mezcla es una opción más eficiente y fiable para aplicaciones que necesitan ordenar grandes conjuntos de datos.

Además, al incorporar los algoritmos de ordenamiento por montículos, rápido y residuos en el análisis, se pudo observar una variedad de comportamientos en cuanto a su tiempo de ejecución y capacidad para manejar conjuntos de datos de distintos tamaños y complejidades. Por ejemplo, el algoritmo de montículos mostró un rendimiento consistente en términos de tiempo de ejecución y una capacidad para manejar conjuntos de datos de tamaño moderado a grande de manera eficiente. Por otro lado, el algoritmo de ordenamiento rápido también demostró ser altamente eficiente en la mayoría de los casos, especialmente para conjuntos de datos grandes y aleatorios. De igual forma, el algoritmo de residuos exhibió un tiempo de ejecución relativamente bajo en comparación con otros algoritmos, lo que lo convierte en una opción atractiva para conjuntos de datos de tamaño moderado. Estos hallazgos sugieren que la elección del algoritmo más adecuado dependerá de varios factores, como el tamaño y la naturaleza de los datos, así como los requisitos específicos de la aplicación en cuestión.

Estos resultados resaltan la importancia de elegir un algoritmo apropiado en función de las necesidades específicas de la aplicación, teniendo en cuenta factores como el tamaño y la complejidad de los datos a ordenar. En última instancia, esta investigación proporciona una base sólida para tomar decisiones informadas al diseñar e implementar algoritmos de clasificación en una variedad de situaciones.

Referencias

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

Gael Alpízar Alfaro
Facultad de ingeniería, Computación con Varios Énfasis
Universidad de Costa Rica (UCR).
Carné C20270.

APÉNDICE A

Código de los Algoritmos

El código se muestra en los algoritmos 1, 2 y 3.

Algoritmo 1 El algoritmo de selección es un método de ordenación que busca el elemento más pequeño en cada iteración y lo coloca en la posición correcta, intercambiándolo con el elemento actual en la lista. Este proceso se repite hasta que toda la lista está ordenada. En cada iteración, busca el elemento más pequeño en la parte no ordenada y lo coloca al principio de la lista ordenada. Este proceso continúa hasta que todos los elementos están en su lugar. Aunque no es tan eficiente como otros algoritmos para grandes conjuntos de datos, el algoritmo de selección es simple y fácil de implementar.

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

Algoritmo 2 El algoritmo de inserción es un método de ordenamiento que recorre un arreglo de izquierda a derecha, y en cada iteración, toma un elemento de la lista y lo inserta en su posición correcta en el subarreglo ya ordenado a la izquierda. El proceso de inserción se repite hasta que todos los elementos están en su posición correcta.

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

Algoritmo 3 El algoritmo de mezcla es un método para organizar una lista de elementos. Funciona dividiendo la lista en partes más pequeñas hasta que cada parte tenga solo un elemento. Luego, combina estas partes gradualmente, asegurándose de que estén en orden mientras las une. Este proceso continúa hasta que se vuelva a formar la lista completa, pero esta vez en orden. Es como organizar un montón de cartas: primero divides las cartas en pilas pequeñas, luego las mezclas de nuevo, asegurándote de que estén en orden.

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

    // Copiar elementos a los subarreglos L y R
    for (int i = 0; i < nL; i++)
        L[i] = A[p + i];
    for (int j = 0; j < nR; j++)
        R[j] = A[q + j + 1];

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

    // Mezclar los subarreglos L y R en el arreglo A
    while (i < nL && j < nR) {
        if (L[i] <= R[j]) {
            A[k] = L[i];
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
        k++;
    }

    // Copiar los elementos restantes de L (si hay alguno)
    while (i < nL) {
        A[k] = L[i];
        i++;
        k++;
    }

    // Copiar los elementos restantes de R (si hay alguno)
    while (j < nR) {
        A[k] = R[j];
        j++;
        k++;
```

```
        }
        // Liberar memoria
        delete[] L;
        delete[] R;
    }

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

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

    string ImprimirDatosDeTarea(){
        return "c20270 Tarea 1 Etapa 1";
}
```

Algoritmo 4 El algoritmo de ordenamiento por montículos, o heapsort, organiza los elementos de una lista construyendo y manipulando una estructura de datos llamada montículo. Consiste en construir un montículo a partir de la lista desordenada y luego extraer repetidamente el elemento máximo o mínimo del montículo para obtener una lista ordenada. La ventaja principal de heapsort es su eficiencia en tiempo y espacio, con un rendimiento de O(n log n) en el peor de los casos y sin requerir espacio adicional más allá de la lista original.

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

        if (l < heap_size && A[l] > A[i])
            largest = l;
        if (r < heap_size && A[r] > A[largest])
            largest = r;

        if (largest != i) {
            swap(A[i], A[largest]);
            maxHeapify(A, largest, heap_size);
        }
    }

    void Ordenador::buildMaxHeap(int *A, int n) {
        int heap_size = n;
```

```
        for (int i = n / 2 - 1; i >= 0; i--)
            maxHeapify(A, i, heap_size);
    }

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

Algoritmo 5 El algoritmo de ordenamiento rápido, o quicksort, funciona dividiendo una lista en subconjuntos más pequeños usando un elemento pivote. Luego, ordena recursivamente cada subconjunto y combina los resultados para obtener una lista ordenada. Esto se logra mediante el proceso de partición, donde los elementos se organizan de manera que los menores que el pivote estén a la izquierda y los mayores a la derecha. El pivote se coloca en su posición final y se aplican estas operaciones de manera recursiva en los subconjuntos. Quicksort tiene un rendimiento promedio eficiente de O(n log n), aunque en el peor de los casos puede llegar a O(n^2). Sin embargo, sigue siendo ampliamente utilizado debido a su eficiencia y simplicidad de implementación.

```
    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++;
                swap(A[i], A[j]);
            }
        }
        swap(A[i + 1], A[r]);
        return i + 1;
    }

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

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

Algoritmo 6 El algoritmo de ordenamiento por residuos, utilizando counting sort, es eficiente cuando el rango de los elementos es pequeño y conocido. Funciona contando las ocurrencias de cada elemento único y luego reconstruyendo la lista ordenada usando esta información. En el caso del counting sort, se crea un arreglo de conteo para cada elemento único, luego se calcula su posición en la lista ordenada y se coloca en su lugar correspondiente. Es especialmente rápido cuando el rango de los elementos es pequeño en comparación con el tamaño de la lista, con una complejidad temporal lineal ($O(n + k)$), donde n es el tamaño de la lista y k es el rango de los elementos.

```
void Ordenador::countingSort(int* A, int n, int exp, int base) {
    int output[n];
    int count[base];

    if(n > 0) {
        count[0] = 0;
        for(int i = 1; i < base; i++){
            count[i] = 0;
        }
    }

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

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

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

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

int Ordenador::getMax(int* A, int n) {
    int max = A[0];
    for (int i = 1; i < n; i++) {
        if (A[i] > max)
            max = A[i];
    }
    return max;
}

void Ordenador::radixsort(int* A, int n) {
    int max = getMax(A, n);
    int base = pow(2, (int)log2(max));
```

```
        for (int exp = 1; max / exp > 0; exp *= base)
            countingSort(A, n, exp, base);
}
```