ALGORITMOS DE ORDENAMIENTO Y SU APLICACIÓN CON EL LENGUAJE DE PROGRAMACIÓN JAVA

PROYECTO ACADÉMICO

ESTRUCTURAS DE DATOS

DOCENTE

JHON ARRIETA

AUTOR

IRINA BALLESTEROS OSPINO

UNIVERSIDAD DE CARTAGENA

FACULTAD DE INGENIERÍA

PROGRAMA DE INGENIERÍA DE SOFTWARE

CARTAGENA DE INDIAS

AÑO 2024

TABLA DE CONTENIDOS

INTRODUCCIÓN

En el mundo de la informática, los algoritmos de ordenamiento son herramientas fundamentales para organizar y manipular datos de manera eficiente. Estos algoritmos desempeñan un papel crucial en diversos aspectos de la ingeniería de software, desde el desarrollo de sistemas complejos hasta la optimización del rendimiento de aplicaciones.

¿Qué son los algoritmos de ordenamiento?

Los algoritmos de ordenamiento son secuencias de instrucciones diseñadas para reordenar una colección de elementos en un orden específico, como numérico, alfabético o por cualquier otro criterio definido. Estos algoritmos operan sobre conjuntos de datos de diversos tamaños y tipos, desde simples listas de números hasta estructuras de datos complejas.

¿Por qué son importantes los algoritmos de ordenamiento en la ingeniería de software?

La importancia de los algoritmos de ordenamiento en la ingeniería de software radica en su amplia aplicabilidad en diversas tareas y escenarios:

1. Organización de datos: Los algoritmos de ordenamiento permiten organizar y estructurar conjuntos de datos de manera eficiente, facilitando su acceso, análisis y procesamiento posterior.
2. Búsqueda de información: En combinación con algoritmos de búsqueda, los algoritmos de ordenamiento optimizan la búsqueda de elementos específicos dentro de grandes conjuntos de datos, reduciendo el tiempo y los recursos computacionales necesarios.
3. Optimización de algoritmos: Los algoritmos de ordenamiento se utilizan como componentes fundamentales en otros algoritmos más complejos, como algoritmos de clasificación, algoritmos de compresión de datos y algoritmos de análisis de datos.
4. Eficiencia en el manejo de datos: La elección del algoritmo de ordenamiento adecuado puede mejorar significativamente el rendimiento de aplicaciones

que manipulan grandes volúmenes de datos, reduciendo el tiempo de ejecución y optimizando el uso de recursos computacionales.

Ejemplos de aplicaciones de algoritmos de ordenamiento en la ingeniería de software:

1. Bases de datos: Los sistemas de bases de datos utilizan algoritmos de ordenamiento para organizar y recuperar registros de manera eficiente, permitiendo consultas rápidas y precisas.
2. Compiladores e intérpretes: Los compiladores e intérpretes de lenguajes de programación emplean algoritmos de ordenamiento para analizar y procesar el código fuente de manera eficiente, optimizando la traducción y la ejecución del programa.
3. Motores de búsqueda: Los motores de búsqueda utilizan algoritmos de ordenamiento para clasificar y organizar páginas web en función de su relevancia para una consulta específica, mejorando la experiencia del usuario.
4. Aplicaciones de análisis de datos: En el análisis de datos, los algoritmos de ordenamiento se utilizan para preparar y organizar conjuntos de datos antes de aplicar técnicas de análisis estadístico y de minería de datos.
5. Sistemas de recomendación: Los sistemas de recomendación utilizan algoritmos de ordenamiento para clasificar y presentar elementos a los usuarios en función de sus preferencias e historial de interacciones.

En resumen, los algoritmos de ordenamiento son herramientas esenciales en la ingeniería de software, proporcionando la base para una amplia gama de tareas de organización, procesamiento y análisis de datos. Su comprensión y aplicación eficiente son fundamentales para el desarrollo de sistemas de software robustos y escalables.

OBJETIVOS

Los algoritmos de ordenamiento tienen como objetivo principal organizar y estructurar conjuntos de datos de manera eficiente, facilitando su posterior manejo y procesamiento. Específicamente, los objetivos clave de los algoritmos de ordenamiento en la ingeniería de software son:

- Ordenar elementos de un conjunto de datos en un orden específico, como numérico, alfabético o por cualquier otro criterio definido.
- Agrupar elementos similares o relacionados entre sí.
- Facilitar el acceso a elementos específicos dentro de un conjunto de datos.
- Optimizar la búsqueda de elementos específicos dentro de grandes conjuntos de datos.
- Reducir el tiempo y los recursos computacionales necesarios para encontrar elementos específicos.
- Permitir la búsqueda eficiente de información relevante en bases de datos, sistemas de archivos y otras estructuras de almacenamiento.
- Servir como componentes fundamentales en otros algoritmos más complejos, como algoritmos de clasificación, algoritmos de compresión de datos y algoritmos de análisis de datos.
- Mejorar el rendimiento de aplicaciones que manipulan grandes volúmenes de datos, reduciendo el tiempo de ejecución y optimizando el uso de recursos computacionales.
- Facilitar el desarrollo de sistemas de software robustos y escalables.
- Permitir el procesamiento ordenado de datos, facilitando su análisis, manipulación y visualización.
- Reducir la complejidad computacional de tareas que involucran el manejo de grandes conjuntos de datos.
- Optimizar el uso de la memoria caché y otros recursos computacionales durante el procesamiento de datos.
- Presentar datos ordenados de manera clara y comprensible, facilitando su análisis e interpretación.
- Permitir la identificación de patrones, tendencias y anomalías en los datos.

- Apoyar la toma de decisiones basadas en datos y el desarrollo de estrategias efectivas.

En resumen, los objetivos de los algoritmos de ordenamiento en la ingeniería de software van más allá de la simple organización de datos. Estos algoritmos son herramientas esenciales para optimizar el procesamiento de información, mejorar el rendimiento de aplicaciones y facilitar la toma de decisiones basadas en datos. Su comprensión y aplicación eficiente son fundamentales para el desarrollo de sistemas de software modernos y eficientes.

DESARROLLO

1. Responda las siguientes preguntas, explicando y argumentando con sus propias palabras cada ítem, es ideal, pero no es necesario, que realice un esquema o gráfico que pueda complementar dicha explicación.

1.1. Explicar al menos 5 de los siguientes algoritmos de ordenamiento. Debe usar un único conjunto de datos (array o lista) y escribir un ejemplo de cada algoritmo en al menos 3 lenguajes de programación de los descritos anteriormente. Se debe adjuntar a este informe (subiendo a la plataforma SIMA-PESAD) un archivo comprimido con el código de los ejemplos. Hoy día no es necesario tener que instalar SDK o IDE en nuestro PC para poder desarrollar ejemplos simples de código, ya que existen muchas aplicaciones online y para el smartphone que nos permiten escribir el código, compilarlo, ejecutarlo y hasta depurarlo para hacer la respectiva prueba funcional.

ALGORITMO DE ORDAMIENTO
Ordenamiento de burbuja (Bubblesort)
Ordenamiento Aleatorio
Ordenamiento rápido (Quicksort)
Ordenamiento por inserción (Insertion sort)
Ordenamiento por selección (Selection sort)
Ordenamiento Shell (Shell sort)
Ordinamiento Distribution sort
Ordenamiento por mezcla (Merge sort)
Ordenamiento de burbuja bidireccional (Cocktail sort)
Ordenamiento por casilleros (Bucket sort)
Ordenamiento por cuentas (Counting sort)
Ordenamiento con árbol binario (Binary tree sort)
Ordenamiento Pigeonhole sort
Ordenamiento Radix (Radix sort)
Ordanamiento Gnome sort
Ordenamiento Comb sort
Ordenamiento por montículos (Heapsort)
Ordenamiento Smoothsort
Ordanamiento Several Unique Sort
Ordanamiento Bogosort
Ordenamiento de panqueques (Pancake sorting)

1. Ordenamiento de burbuja (Bubblesort) es uno de los más simples y fáciles de entender. Funciona comparando pares de elementos adyacentes e intercambiándose si están en el orden incorrecto. El algoritmo se repite hasta que no haya más intercambios.

Ejemplo:

Supongamos que tenemos una lista de números: {5, 2, 4, 1, 3}.

- ➢ Primera pasada:
  - Comparar 5 con 2: Intercambiar.
  - Comparar 2 con 4: Intercambiar.
  - Comparar 4 con 1: Intercambiar.
  - Comparar 1 con 3: Intercambiar.
  - La lista ahora es: {2, 5, 4, 1, 3}.
- ➢ Segunda pasada:
  - Comparar 2 con 5: Intercambiar.
  - Comparar 5 con 4: Intercambiar.
  - Comparar 4 con 1: Intercambiar.
  - Comparar 1 con 3: Intercambiar.
  - La lista ahora es: {1, 2, 3, 4, 5}.
- ➢ Tercera pasada:
  - Comparar 1 con 2: No es necesario intercambiar.
  - Comparar 2 con 3: No es necesario intercambiar.
  - Comparar 3 con 4: No es necesario intercambiar.
  - Comparar 4 con 5: No es necesario intercambiar.
  - La lista ya está ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Es un algoritmo simple y fácil de entender.
- No requiere memoria adicional.

Desventajas:

- Es un algoritmo lento, especialmente para listas grandes.

- Es ineficiente para listas casi ordenadas.

```java
/* Algoritmo de Ordenamiento Bubble Sort Java*/
/* Declaración de la función */
public static void bubbleSort(int[] arr) {
    // Definición de variables
    int n = arr.length;         //Se almacena la longitud del Array con el método length
    // Creamos un bucle for que itera desde la posición 0 hasta la posición n - 2 del array
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            // Se realiza una comparación e intercambio de elementos usando una variable temporal
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;

            }
        }
    }
}
/*
int[] arr = {64, 34, 25, 12, 22, 11, 90};
bubbleSort(arr);
System.out.println(Arrays.toString(arr));
*/
```

```python
# Algoritmo de Ordenamiento Bubble Sort Python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]


arr = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(arr)
```

```
print(arr)

// Algoritmo de Ordenamiento Bubble Sort Javascript
function bubbleSort(arr) {
    let n = arr.length;
    for (let i = 0; i < n-1; i++) {
        for (let j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                let temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
let arr = [64, 34, 25, 12, 22, 11, 90];
bubbleSort(arr);
console.log(arr);
```

2. Algoritmo de Ordenamiento aleatorio funciona seleccionando aleatoriamente elementos de la lista y colocándolos en su posición correcta. El algoritmo se repite hasta que todos los elementos estén en su lugar.

Ejemplo:

Supongamos que tenemos una lista de números: {5, 2, 4, 1, 3}.

- Seleccionar aleatoriamente un elemento: 3.
- Colocar el elemento seleccionado en su posición correcta: {3, 5, 2, 4, 1}.
- Seleccionar aleatoriamente otro elemento: 1.
- Colocar el elemento seleccionado en su posición correcta: {1, 3, 5, 2, 4}.
- Continuar hasta que todos los elementos estén en su lugar: {1, 2, 3, 4, 5}.

Ventajas:

- Es un algoritmo simple y fácil de implementar.
- Es eficiente para listas casi ordenadas.

Desventajas:

- Es un algoritmo inestable, es decir, el tiempo de ejecución puede variar mucho dependiendo de la entrada.
- No es eficiente para listas grandes.

3. Ordenamiento rápido (Quicksort)

El algoritmo de ordenamiento rápido es uno de los más eficientes para ordenar listas grandes. Funciona dividiendo la lista en dos sublistas más pequeñas, ordenando cada sublista recursivamente y luego uniendo las sub listas ordenadas.

Ejemplo:

Supongamos que tenemos una lista de números: {5, 2, 4, 1, 3}.

- Seleccionar un elemento pivote (por ejemplo, el elemento central): 4.
- Dividir la lista en dos sublistas: {1, 2, 3} y {5}.
- Ordenar recursivamente las sublistas: {1, 2, 3} y {5}.
- Unir las sub listas ordenadas: {1, 2, 3, 5}.

Ventajas:

- Es uno de los algoritmos de ordenamiento más eficientes para listas grandes.
- Es eficiente para listas casi ordenadas.

Desventajas:

- Es un algoritmo inestable, es decir, el tiempo de ejecución puede variar mucho dependiendo de la entrada.
- Puede ser ineficiente para listas pequeñas.

```java
// Algoritmo de Ordenamiento Quick Sort Java
public static 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);
    }
```

```
}

public static 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;
}

// Ejemplo de uso
int[] arr = {10, 7, 8, 9, 1, 5};
quickSort(arr, 0, arr.length - 1);
System.out.println(Arrays.toString(arr));
```

```
# Algoritmo de Ordenamiento QuickSort Python
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

```
# Ejemplo de uso
arr = [10, 7, 8, 9, 1, 5]
quick_sort(arr, 0, len(arr) - 1)
print(arr)
```

```
// Algoritmo de ordenamiento Quicksort Javascript
function quickSort(arr, left = 0, right = arr.length - 1) {
    if (left < right) {
        let pivotIndex = partition(arr, left, right);
        quickSort(arr, left, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, right);
    }
}

function partition(arr, left, right) {
    let pivot = arr[right];
    let i = left - 1;
    for (let j = left; j < right; j++) {
        if (arr[j] < pivot) {
            i++;
            [arr[i], arr[j]] = [arr[j], arr[i]];
        }
    }
    [arr[i + 1], arr[right]] = [arr[right], arr[i + 1]];
    return i + 1;
}

// Ejemplo de uso
let arr = [10, 7, 8, 9, 1, 5];
quickSort(arr);
console.log(arr);
```

4. Algoritmo de ordenamiento por inserción (Insertion sort): funciona insertando cada elemento de la lista en su posición correcta en la lista ordenada. El algoritmo se repite hasta que todos los elementos estén en su lugar.

Supongamos que tenemos una lista de números: {5, 2, 4, 1, 3}.

- Insertar el segundo elemento (2) en la lista ordenada: {2, 5}.
- Insertar el tercer elemento (4) en la lista ordenada: {2, 4, 5}.
- Insertar el cuarto elemento (1) en la lista ordenada: {1, 2, 4, 5}.
- Insertar el quinto elemento (3) en la lista ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Es un algoritmo simple y fácil de entender.
- Es eficiente para listas pequeñas.
- Es un algoritmo estable, es decir, el orden relativo de los elementos iguales se mantiene.

Desventajas:

- Es un algoritmo lento, especialmente para listas grandes.
- Es ineficiente para listas casi ordenadas.

```java
// Algoritmo Insertion Sort Java
public static void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Ejemplo de uso
int[] arr = {12, 11, 13, 5, 6};
insertionSort(arr);
System.out.println(Arrays.toString(arr));
```

```python
# Algoritmo de Ordenamiento Insertion Sort Python
def insertion_sort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key

# Ejemplo de uso
arr = [12, 11, 13, 5, 6]
insertion_sort(arr)
print(arr)
```

```javascript
// Algoritmo de ordenamiento Insertion Sort Javascript
function insertionSort(arr) {
    let n = arr.length;
    for (let i = 1; i < n; ++i) {
        let key = arr[i];
        let j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Ejemplo de uso
let arr = [12, 11, 13, 5, 6];
insertionSort(arr);
console.log(arr);
```

5. Ordenamiento por selección (Selection sort) funciona encontrando el elemento mínimo de la lista y colocándolo en la primera posición, luego encontrando el segundo elemento mínimo y colocándolo en la segunda posición, y así sucesivamente hasta que todos los elementos estén en su lugar.

Ejemplo:

Supongamos que tenemos una lista de números: {5, 2, 4, 1, 3}.

- Encontrar el elemento mínimo (1) y colocarlo en la primera posición: {1, 5, 2, 4, 3}.
- Encontrar el segundo elemento mínimo (2) y colocarlo en la segunda posición: {1, 2, 5, 4, 3}.
- Encontrar el tercer elemento mínimo (3) y colocarlo en la tercera posición: {1, 2, 3, 5, 4}.
- Encontrar el cuarto elemento mínimo (4) y colocarlo en la cuarta posición: {1, 2, 3, 4, 5}.
- La lista ya está ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Es un algoritmo simple y fácil de entender.
- Es un algoritmo estable, es decir, el orden relativo de los elementos iguales se mantiene.

Desventajas:

- Es un algoritmo lento, especialmente para listas grandes.
- Requiere memoria adicional para almacenar el elemento mínimo encontrado.

6. Ordenamiento Shell (Shell sort) es una variante del algoritmo de ordenamiento por inserción que utiliza un incremento para dividir la lista en sublistas más pequeñas antes de insertar los elementos. El incremento se reduce a medida que se avanza en el algoritmo, lo que hace que el algoritmo sea más eficiente que el ordenamiento por inserción simple.

Ejemplo:

Supongamos que tenemos una lista de números: {5, 2, 4, 1, 3}.

- Usar un incremento de 2 para dividir la lista en sublistas: {{5}, {2, 4}, {1, 3}}.
- Ordenar cada sublista usando el algoritmo de ordenamiento por inserción.
- Reducir el incremento a 1 y dividir la lista en sublistas: {{5}, {2}, {4}, {1}, {3}}.
- Ordenar cada sublista usando el algoritmo de ordenamiento por inserción.
- La lista ya está ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Es un algoritmo más eficiente que el ordenamiento por inserción simple para listas grandes.
- Es un algoritmo estable, es decir, el orden relativo de los elementos iguales se mantiene.

Desventajas:

- Es un algoritmo más complejo que el ordenamiento por inserción simple.
- Requiere memoria adicional para almacenar el incremento.

7. **Ordenamiento por Distribución (Distribution Sort)** es un tipo de algoritmo de ordenamiento que funciona aprovechando el conocimiento previo del rango de valores posibles en los datos a ordenar. Este conocimiento se usa para distribuir los elementos a posiciones específicas en una estructura auxiliar, y luego se recolectan en el orden deseado.

De acuerdo con lo citado anteriormente cabe destacar que el algoritmo de ordenamiento por Distribución nos permite:

- Determinar el valor mínimo y máximo que pueden tomar los elementos en la lista.
- Crear una estructura auxiliar, como un arreglo, del tamaño del rango (máximo - mínimo + 1). Esta estructura se utiliza para contar la frecuencia de aparición de cada valor.

- Recorrer la lista original e incrementar el contador correspondiente a cada valor en la estructura auxiliar.
- Calcular la estructura auxiliar para determinar la posición final de cada elemento en la lista ordenada. Esto se puede hacer utilizando prefijos de suma.
- Recorrer la lista original nuevamente y se usa el valor de cada elemento como índice para colocarlo en su posición final dentro de la lista ordenada.

Ventajas:

- Puede ser muy eficiente para listas donde el número de valores distintos es pequeño comparado con el tamaño de la lista.
- En el mejor de los casos, su complejidad temporal es lineal O(n + k), donde n es el tamaño de la lista y k es el número de valores distintos.

Desventajas:

- No es aplicable para casos donde el rango de valores posibles es muy grande comparado con el tamaño de la lista.
- Requiere memoria adicional para la estructura auxiliar.

Ejemplo:

Supongamos que tenemos una lista de números {3, 2, 1, 4, 1, 3} con un rango de valores de 1 a 4.

1. Creamos una estructura auxiliar de tamaño 4 (máximo - mínimo + 1) inicializada con ceros: [0, 0, 0, 0].
2. Recorremos la lista original e incrementamos los contadores en la estructura auxiliar: [0, 2, 2, 1].
3. Procesamos la estructura auxiliar para calcular las posiciones finales. Por ejemplo, usando prefijos de suma, tendríamos: [0, 2, 4, 5].
4. Recorremos nuevamente la lista original y colocamos cada elemento en su posición final de acuerdo a la estructura auxiliar:
    - Elemento 3 (valor 0): Ya hay 2 elementos menores (en las posiciones 0 y 1), lo colocamos en la posición 2.

- Elemento 2 (valor 1): Ya hay 2 elementos menores (en las posiciones 0 y 1), lo colocamos en la posición 3.
- Y así sucesivamente.

La lista ordenada final sería: {1, 1, 2, 3, 3, 4}.

8. **Ordenamiento por mezcla (Merge sort)** es un algoritmo de ordenamiento con un rendimiento promedio y en el peor de los casos de **O(n log n)**, donde n es la cantidad de elementos en la lista. Es un algoritmo estable, lo que significa que conserva el orden relativo de elementos iguales.

El algoritmo funciona de manera recursiva dividiendo la lista en dos sublistas más pequeñas hasta que las sublistas tengan un solo elemento o ningún elemento (caso base). Luego, combina las sublistas ordenadas para formar una lista ordenada general.

Pasos:

1. **Dividir:** Dividir la lista original en dos sublistas de aproximadamente igual tamaño.
2. **Conquistar:** Ordenar recursivamente cada sublista usando el mismo algoritmo de ordenamiento por mezcla.
3. **Combinar:** Fusionar las dos sublistas ordenadas en una sola lista ordenada.

La combinación es la parte crucial del algoritmo:

- Se comparan los primeros elementos de cada sublista.
- El elemento más pequeño se agrega a la lista ordenada final.
- Se elimina el elemento agregado de la sublista correspondiente.
- Se repiten los pasos anteriores hasta que una de las sublistas esté vacía.
- Se agregan los elementos restantes de la otra sublista a la lista ordenada final.

Ejemplo:

Supongamos que queremos ordenar la lista {5, 2, 4, 1, 3}.

Dividir:

- Sublista 1: {5, 2}
- Sublista 2: {4, 1, 3}

Conquistar:

- Ordenar recursivamente la sublista 1: {2, 5}
- Ordenar recursivamente la sublista 2: {1, 3, 4}

Combinar:

- Comparar 2 con 1: Se agrega 1 a la lista ordenada final.
- Comparar 2 con 3: Se agrega 2 a la lista ordenada final.
- Comparar 5 con 3: Se agrega 3 a la lista ordenada final.
- Comparar 5 con 4: Se agrega 4 a la lista ordenada final.
- La lista ordenada final es: {1, 2, 3, 4, 5}.

Ventajas:

- Eficiente en el caso promedio y en el peor de los casos (O(n log n)).
- Algoritmo estable (preserva el orden de elementos iguales).
- Se puede paralelizar fácilmente.
- Buena opción para listas grandes.

Desventajas:

- Requiere memoria adicional para las sublistas durante la recursión.
- Puede ser menos eficiente que otros algoritmos para listas pequeñas (como ordenamiento por inserción).

Ejemplos:

```
// Algoritmo Merge Sort Java
public static void mergeSort(int[] arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
```

```
            merge(arr, l, m, r);
        }
    }

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

        int L[] = new int[n1];
        int R[] = new int[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;

        int 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++;
        }
    }

    // Ejemplo de uso
    int[] arr = {38, 27, 43, 3, 9, 82, 10};
    mergeSort(arr, 0, arr.length - 1);
    System.out.println(Arrays.toString(arr));
```

```
# Algoritmo de Ordenamiento Merge Sort Python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
```

```
        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Ejemplo de uso
arr = [38, 27, 43, 3, 9, 82, 10]
```

```
merge_sort(arr)
print(arr)

// Algoritmo de ordenamiento Merge Sort Javascript
function mergeSort(arr) {
    if (arr.length <= 1) {
        return arr;
    }

    const mid = Math.floor(arr.length / 2);
    const left = arr.slice(0, mid);
    const right = arr.slice(mid);

    return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
    let result = [];
    let leftIndex = 0;
    let rightIndex = 0
```

9. **El ordenamiento de burbuja bidireccional**, también conocido como **Cocktail sort**, es una variante del algoritmo de ordenamiento de burbuja que mejora su rendimiento al realizar pasadas de ordenamiento en ambas direcciones de la lista.

El algoritmo funciona de manera similar al ordenamiento de burbuja simple, pero con la diferencia de que realiza dos pasadas por la lista en cada iteración:

1. **Pasada hacia adelante:** Compara elementos adyacentes de izquierda a derecha y los intercambia si están en el orden incorrecto.
2. **Pasada hacia atrás:** Compara elementos adyacentes de derecha a izquierda y los intercambia si están en el orden incorrecto.

Las pasadas se repiten hasta que no se realizan intercambios en ninguna de las dos direcciones, lo que indica que la lista está ordenada.

Ejemplo:

Supongamos que tenemos una lista de números {5, 2, 4, 1, 3}.

Primera iteración:

- ➢ Pasada hacia adelante:
  - Comparar 5 con 2: Intercambiar.
  - Comparar 2 con 4: Intercambiar.
  - Comparar 4 con 1: Intercambiar.
  - Comparar 1 con 3: Intercambiar.

  La lista ahora es: {2, 5, 4, 1, 3}.
- ➢ Pasada hacia atrás:
  - Comparar 3 con 1: Intercambiar.
  - Comparar 1 con 4: Intercambiar.
  - Comparar 4 con 5: Intercambiar.
  - La lista ahora es: {2, 1, 3, 4, 5}.

Segunda iteración:

- ➢ Pasada hacia adelante:
  - Comparar 2 con 1: Intercambiar.
  - Comparar 1 con 3: No es necesario intercambiar.
  - Comparar 3 con 4: No es necesario intercambiar.
  - Comparar 4 con 5: No es necesario intercambiar.
- ➢ Pasada hacia atrás:
  - Comparar 5 con 4: No es necesario intercambiar.
  - Comparar 4 con 3: No es necesario intercambiar.

- Comparar 3 con 1: No es necesario intercambiar.
- Comparar 1 con 2: No es necesario intercambiar.

  La lista ya está ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Al realizar pasadas en ambas direcciones, se detectan y corrigen más desordenamientos en cada iteración lo que mejora el rendimiento del ordenamiento de burbuja simple.
- La lógica del algoritmo es similar al ordenamiento de burbuja simple, lo que lo hace fácil de entender y programar.
- Puede ser una buena opción para ordenar listas pequeñas donde otros algoritmos más complejos podrían tener una sobrecarga significativa.

Desventajas:

- Su rendimiento en el peor de los casos es O(n^2), lo que lo hace ineficiente para ordenar listas muy grandes.
- Para listas grandes, se suelen utilizar algoritmos como ordenamiento rápido o ordenamiento por mezcla que tienen un mejor rendimiento promedio y en el peor de los casos.

10. **Ordenamiento por casilleros (Bucket sort)** también conocido como **Bucket sort** o **Bin sort**, es un algoritmo de ordenamiento que funciona distribuyendo los elementos a ordenar en un conjunto finito de casilleros. Cada casillero solo puede contener elementos que cumplan con ciertas condiciones, generalmente un rango de valores específico.

Funcionamiento:

- Se determina el valor mínimo y máximo que pueden tomar los elementos en la lista.

- Se crea una estructura de datos que represente los casilleros, como un arreglo o una lista. El tamaño de la estructura debe ser igual al rango (máximo - mínimo + 1).
- Se recorre la lista original y se coloca cada elemento en el casillero correspondiente de acuerdo a su valor.
- Se utiliza un algoritmo de ordenamiento simple, como el ordenamiento por inserción o el ordenamiento por selección, para ordenar los elementos dentro de cada casillero.
- Se recorren los casilleros en orden y se extraen los elementos ordenados, colocándolos en una nueva lista ordenada.

Ejemplo:

Supongamos que tenemos una lista de números {5, 2, 4, 1, 3} con un rango de valores de 1 a 5.

1. Creamos 5 casilleros vacíos (arreglo o lista) para representar el rango de valores.
2. Recorremos la lista original y colocamos cada elemento en su casillero correspondiente:
    - Elemento 5 (valor 5): Se coloca en el casillero 4 (índice 4).
    - Elemento 2 (valor 2): Se coloca en el casillero 1 (índice 1).
    - Elemento 4 (valor 4): Se coloca en el casillero 3 (índice 3).
    - Elemento 1 (valor 1): Se coloca en el casillero 0 (índice 0).
    - Elemento 3 (valor 3): Se coloca en el casillero 2 (índice 2).
3. Ordenamos los elementos dentro de cada casillero:
    - Casillero 0: {1}.
    - Casillero 1: {2}.
    - Casillero 2: {3}.
    - Casillero 3: {4}.
    - Casillero 4: {5}.
4. Recorremos los casilleros en orden y extraemos los elementos ordenados, colocándolos en una nueva lista ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Si el rango de valores es pequeño comparado con el tamaño de la lista, el algoritmo puede ser muy eficiente.
- La lógica del algoritmo es relativamente simple y fácil de entender.
- Preserva el orden relativo de elementos iguales.

Desventajas:

- Si el rango de valores es grande, se requiere una gran cantidad de memoria para crear los casilleros, lo que puede hacerlo ineficiente.
- El algoritmo requiere conocer el rango de valores de antemano para crear los casilleros.
- Para listas con un rango de valores grande o casos generales, existen algoritmos de ordenamiento como el ordenamiento rápido o el ordenamiento por mezcla que suelen ser más eficientes.

11. **Ordenamiento por cuentas (Counting Sort)** es un algoritmo de ordenamiento que funciona contando el número de ocurrencias de cada elemento en la lista a ordenar. Esta información se utiliza para colocar cada elemento en su posición correcta en la lista ordenada.

Funcionamiento:

- Se determina el valor mínimo y máximo que pueden tomar los elementos en la lista.
- Se crea un arreglo de tamaño igual al rango (máximo - mínimo + 1). Este arreglo se utilizará para almacenar el número de ocurrencias de cada valor.
- Se recorre la lista original y se incrementa el contador correspondiente en el arreglo de conteo para cada valor encontrado.
- Se procesa el arreglo de conteo para calcular las posiciones finales de cada elemento en la lista ordenada. Esto se puede hacer utilizando prefijos de suma.
- Se recorre la lista original nuevamente y se usa el valor de cada elemento como índice para colocarlo en su posición final dentro de la lista ordenada.

Ejemplo:

Supongamos que tenemos una lista de números {3, 2, 1, 4, 1, 3} con un rango de valores de 1 a 4.

1. Creamos un arreglo de conteo de tamaño 4 (máximo - mínimo + 1) inicializado con ceros: [0, 0, 0, 0].
2. Recorremos la lista original e incrementamos los contadores en el arreglo de conteo: [0, 2, 2, 1].
3. Procesamos el arreglo de conteo para calcular las posiciones finales. Por ejemplo, usando prefijos de suma, tendríamos: [0, 2, 4, 5].
4. Recorremos nuevamente la lista original y colocamos cada elemento en su posición final de acuerdo a la estructura auxiliar:
    - Elemento 3 (valor 0): Ya hay 2 elementos menores (en las posiciones 0 y 1), lo colocamos en la posición 2.
    - Elemento 2 (valor 1): Ya hay 2 elementos menores (en las posiciones 0 y 1), lo colocamos en la posición 3.
    - Y así sucesivamente.

    La lista ordenada final sería: {1, 1, 2, 3, 3, 4}.

Ventajas:

- Si el rango de valores es pequeño comparado con el tamaño de la lista, el algoritmo puede ser muy eficiente, con una complejidad temporal de O(n + k), donde n es el tamaño de la lista y k es el tamaño del rango.
- La lógica del algoritmo es relativamente simple y fácil de entender.
- Preserva el orden relativo de elementos iguales.

Desventajas:

- Si el rango de valores es grande, se requiere una gran cantidad de memoria para crear el arreglo de conteo, lo que puede hacerlo ineficiente.
- El algoritmo requiere conocer el rango de valores de antemano para crear el arreglo de conteo.

- Para listas con un rango de valores grande o casos generales, existen algoritmos de ordenamiento como el ordenamiento rápido o el ordenamiento por mezcla que suelen ser más eficientes.

  Casos de uso:

  - Ordenar listas de números enteros no negativos.
  - Contar el número de ocurrencias de cada elemento en una lista.
  - Generar histogramas de datos.

12. **Ordenamiento con árbol binario (Binary Tree Sort)** es un algoritmo de ordenamiento que aprovecha las propiedades de un árbol binario de búsqueda (BST) para ordenar una lista de elementos.

Funcionamiento:

1. Construir el árbol binario: Se recorre la lista original e inserta cada elemento en el árbol binario de búsqueda. Al insertar en un BST, los elementos se colocan automáticamente en orden ascendente.
2. Recorrido inorden**:** Se realiza un recorrido inorden del árbol binario de búsqueda.

El **Recorrido inorden:** En un árbol binario de búsqueda, un recorrido inorden visita los nodos del árbol en el siguiente orden:

- Primero, se visita el subárbol izquierdo.
- Luego, se visita el nodo actual.
- Finalmente, se visita el subárbol derecho.

Como los elementos en un BST se insertan en orden ascendente, un recorrido inorden del árbol visitará los nodos en el mismo orden.

Ejemplo:

Supongamos que tenemos una lista de números {5, 2, 4, 1, 3}.

1. **Construir el árbol binario:**
   - Insertar 5: El árbol queda con un nodo raíz con valor 5.

- Insertar 2: Se inserta a la izquierda del nodo con valor 5 como el hijo izquierdo.
- Insertar 4: Se inserta a la derecha del nodo con valor 5 como el hijo derecho.
    - Insertar 1: Se inserta a la izquierda del nodo con valor 2 como el hijo izquierdo del hijo izquierdo.
    - Insertar 3: Se inserta a la derecha del nodo con valor 2 como el hijo derecho del hijo izquierdo.

El árbol binario resultante tendrá la siguiente estructura:

2. **Recorrido inorden:** Al recorrer este árbol inorden, se visitarán los nodos en el orden: 1, 2, 3, 4, 5.

Ventajas:

- Aprovecha la estructura ordenada de un árbol binario de búsqueda.
- Se puede combinar con otros algoritmos de ordenamiento para listas parcialmente ordenadas.

Desventajas:

- Si la lista de entrada ya está ordenada en el orden inverso, el árbol se degenerará en una lista enlazada, y el algoritmo tendrá una complejidad temporal de O(n^2).
- Existen algoritmos de ordenamiento con mejor rendimiento promedio, como el ordenamiento rápido o el ordenamiento por mezcla.
- Requiere memoria adicional para la estructura del árbol binario.

13. **Ordenamiento Pigeonhole Sort (Ordenamiento por casilleros)** es un algoritmo de ordenamiento que funciona distribuyendo los elementos a ordenar en un conjunto finito de casilleros. Cada casillero solo puede contener elementos que cumplan con ciertas condiciones, generalmente un rango de valores específico.

- Se determina el valor mínimo y máximo que pueden tomar los elementos en la lista.
- Se crea una estructura de datos que represente los casilleros, como un arreglo o una lista. El tamaño de la estructura debe ser igual al rango (máximo - mínimo + 1).
- Se recorre la lista original y se coloca cada elemento en el casillero correspondiente de acuerdo a su valor.
- Dependiendo de la implementación, se puede aplicar un algoritmo de ordenamiento simple (como ordenamiento por inserción o selección) a cada casillero individualmente para garantizar que los elementos dentro de cada casillero estén completamente ordenados. En algunos casos, si la cantidad de elementos por casillero es pequeña, puede que no sea necesario ordenarlos explícitamente.
- Se recorren los casilleros en orden y se extraen los elementos ordenados, colocándolos en una nueva lista ordenada final.

Ejemplo:

Supongamos que tenemos una lista de números {5, 2, 4, 1, 3} con un rango de valores de 1 a 5.

1. Creamos 5 casilleros vacíos (arreglo o lista) para representar el rango de valores.
2. Recorremos la lista original y colocamos cada elemento en su casillero correspondiente:
    - Elemento 5 (valor 5): Se coloca en el casillero 4 (índice 4).
    - Elemento 2 (valor 2): Se coloca en el casillero 1 (índice 1).
    - Elemento 4 (valor 4): Se coloca en el casillero 3 (índice 3).
    - Elemento 1 (valor 1): Se coloca en el casillero 0 (índice 0).

- Elemento 3 (valor 3): Se coloca en el casillero 2 (índice 2).

3. En este ejemplo, como el número de elementos por casillero es pequeño, podemos omitir el paso de ordenar individualmente cada casillero.
4. Recorremos los casilleros en orden y extraemos los elementos ordenados, colocándolos en una nueva lista ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Si el rango de valores es pequeño comparado con el tamaño de la lista, el algoritmo puede ser muy eficiente. En el mejor de los casos, su complejidad temporal es lineal O(n + k), donde n es el tamaño de la lista y k es el número de valores distintos.
- La lógica del algoritmo es relativamente simple y fácil de entender.

Desventajas:

- Si el rango de valores es grande, se requiere una gran cantidad de memoria para crear los casilleros, lo que puede hacerlo ineficiente.
- Para listas con un rango de valores grande o casos generales, existen algoritmos de ordenamiento como el ordenamiento rápido o el ordenamiento por mezcla que suelen ser más eficientes.

14. **Ordenamiento Radix (Radix Sort)** también conocido como ordenamiento por base o por dígitos, es un algoritmo de ordenamiento no comparativo que funciona procesando los elementos dígito por dígito. Es particularmente eficiente para ordenar elementos con claves numéricas representadas en un sistema de numeración posicional (como decimal, binario, etc.).

➢ Se determina la cantidad máxima de dígitos que pueda tener cualquier elemento en la lista.
➢ Se realizan iteraciones desde el dígito menos significativo (unidades) hasta el dígito más significativo. En cada iteración:
    - Se crea una estructura auxiliar (como un arreglo o lista) del tamaño igual a la base del sistema de numeración (generalmente 10 para decimal).

- Se recorre la lista original y se distribuyen los elementos en la estructura auxiliar según el valor de su dígito correspondiente a la posición actual (la unidad en la primera pasada, decenas en la segunda, etc.).
- Se utiliza un conteo o técnicas de recolección para reunir los elementos de la estructura auxiliar en el orden original de la lista.

➢ Se repiten las pasadas de ordenamiento por dígito hasta que se haya procesado el dígito más significativo. Una vez finalizadas todas las pasadas, la lista estará completamente ordenada.

Ejemplo:

Supongamos que tenemos una lista de números {170, 45, 750, 90} con un sistema de numeración decimal (d = 3).

Primera pasada (ordenamiento por unidades):

1. Creamos un arreglo auxiliar de tamaño 10 (0-9).
2. Recorremos la lista original y colocamos cada elemento en la casilla correspondiente según su dígito de las unidades:
   - 170 -> Casilla 0
   - 45 -> Casilla 5
   - 750 -> Casilla 0
   - 90 -> Casilla 0
3. Recolectamos los elementos del arreglo auxiliar en el orden original de la lista: {90, 750, 170, 45}

Segunda pasada (ordenamiento por decenas):

1. Creamos un arreglo auxiliar de tamaño 10 (0-9).
2. Recorremos la lista (ya parcialmente ordenada) y colocamos cada elemento en la casilla correspondiente según su dígito de las decenas:
   - 90 -> Casilla 9
   - 750 -> Casilla 7
   - 170 -> Casilla 1
   - 45 -> Casilla 4

3. Recolectamos los elementos del arreglo auxiliar en el orden original de la lista: {45, 170, 750, 90}

Tercera pasada (ordenamiento por centenas):

1. Creamos un arreglo auxiliar de tamaño 10 (0-9).
2. Recorremos la lista (casi ordenada) y colocamos cada elemento en la casilla correspondiente según su dígito de las centenas:
    - 45 -> Casilla 0
    - 170 -> Casilla 1
    - 750 -> Casilla 7
    - 90 -> Casilla 0
3. Recolectamos los elementos del arreglo auxiliar en el orden original de la lista: {45, 90, 170, 750} (la lista ya está completamente ordenada).

Ventajas:

- Es un algoritmo muy eficiente para ordenar enteros o elementos con claves numéricas representadas en un sistema de numeración posicional.
- Si todas las claves tienen la misma longitud (d), la complejidad temporal es lineal O(n + d), donde n es el tamaño de la lista y d es el número de dígitos.
- Dependiendo de la implementación del conteo o recolección de elementos en las pasadas, el algoritmo puede ser estable, preservando el orden relativo de elementos iguales.

Desventajas:

- Si las claves tienen longitudes variables, puede requerir ajustes para manejar ceros a la izquierda y su rendimiento puede disminuir.
- Requiere memoria adicional para las estructuras auxiliares utilizadas en cada pasada.

15. **Gnome Sort (Ordenamiento Gnomo)** también conocido como ordenamiento estúpido (Stupid sort), es un algoritmo de ordenamiento simple que funciona de

manera similar al ordenamiento por burbuja pero con una optimización. Recorre la lista comparando elementos adyacentes y los intercambia si están en el orden incorrecto. Sin embargo, a diferencia del ordenamiento por burbuja, el Gnome Sort avanza dos posiciones en cada comparación, lo que le permite evitar comparaciones innecesarias.

- ➢ Se comienza con dos índices:
  - i (inicializado en 1) que representa el elemento actual a comparar.
  - j (inicializado en 0) que representa el elemento anterior al actual.
- ➢ Se comparan los elementos en las posiciones i y j:
  - Si A[i] >= A[j], se incrementa i en 1 para avanzar a la siguiente comparación.
  - Si A[i] < A[j], se intercambian los elementos en las posiciones i y j-1 (retrocede un paso).
  - Se repiten los pasos 2 y 3 hasta que i llegue al final de la lista (i = n-1).

En el ordenamiento por burbuja, se realizan comparaciones completas de toda la lista en cada iteración. El Gnome Sort aprovecha el hecho de que, una vez que se intercambian elementos, la sublista anterior a la posición actual ya está ordenada. Por eso, en lugar de retroceder por toda la lista comparando con todos los elementos anteriores, sólo retrocede un paso y compara con el elemento anterior.

Ejemplo:

Supongamos que tenemos una lista de números desordenada: {5, 2, 4, 1, 3}.

Primera iteración:

- i = 1, j = 0
- Comparar A[1] (2) con A[0] (5): Como 2 < 5, no es necesario intercambiar.
- Incrementar i a 2.

Segunda iteración:

- i = 2, j = 1
- Comparar A[2] (4) con A[1] (2): Como 4 > 2, no es necesario intercambiar.
- Incrementar i a 3.

Tercera iteración:

- i = 3, j = 2
- Comparar A[3] (1) con A[2] (4): Como 1 < 4, se intercambian A[2] y A[1] ({5, 2, 1, 4, 3}).
- Decrementar j a 1.

Cuarta iteración:

- i = 3, j = 1
- Comparar A[3] (1) con A[1] (2): Como 1 < 2, no es necesario intercambiar.
- Incrementar i a 4.

Quinta iteración:

- i = 4, j = 3
- Comparar A[4] (3) con A[3] (1): Como 3 > 1, se intercambian A[3] y A[2] ({5, 2, 3, 4, 1}).
- Decrementar j a 2.

Sexta iteración:

- i = 4, j = 2
- Comparar A[4] (3) con A[2] (3): Como 3 = 3, no es necesario intercambiar.
- Incrementar i a 5 (final de la lista).

La lista final ordenada es: {1, 2, 3, 4, 5}.

Ventajas:

- La lógica del algoritmo es relativamente simple y fácil de entender.
- Se puede combinar con otros algoritmos de ordenamiento para listas parcialmente ordenadas.
- Puede ser eficiente para listas pequeñas comparado con otros algoritmos como el ordenamiento por burbuja.

Desventajas:

- Su rendimiento en el peor de los casos es O(n^2), similar al ordenamiento por burbuja.
- Existen algoritmos de ordenamiento con mejor rendimiento promedio, como el ordenamiento rápido o el ordenamiento por mezcla, que son preferibles para listas grandes.

16. Ordenamiento Comb Sort (Ordenamiento por Peine) es un algoritmo de ordenamiento híbrido que combina elementos del ordenamiento por burbuja y del ordenamiento por saltos (Shell sort). Se diseñó para mejorar el rendimiento del ordenamiento por burbuja al reducir el número de swaps innecesarios que se realizan en listas casi ordenadas.

- ➢ Se calcula un valor inicial para el "gap" (salto) utilizando una fórmula específica (generalmente el valor inicial se define como el 1.3 - 1.25 * n, donde n es el tamaño de la lista).
- ➢ Se itera sobre la lista utilizando el gap actual. En cada iteración:
  - Se comparan elementos separados por el gap actual.
  - Si los elementos están en el orden incorrecto, se intercambian.
- ➢ Se reduce el valor del gap para la siguiente iteración. Una estrategia común es dividir el gap por 1.3.
- ➢ Se repiten los pasos de recorrer la lista con el gap actual y reducir el gap hasta que el gap sea igual a 1 (un salto de una posición).

Explicación:

El gap inicial permite realizar saltos más grandes en la lista, comparando elementos más alejados y corrigiendo el orden a mayor escala. A medida que se reduce el gap, las comparaciones se van acercando y se corrigen los desórdenes más pequeños, similar al ordenamiento por burbuja.

Fórmula para el gap inicial:

Existen diferentes fórmulas para calcular el gap inicial. Una opción común es:

gap = floor(n / 1.3)

donde:

gap es el valor inicial del salto.

- n es el tamaño de la lista.
- floor es una función matemática que redondea hacia abajo al entero más cercano.

Ejemplo:

Supongamos que tenemos una lista de números desordenada: {5, 2, 4, 1, 3}.

Primera iteración:

➢ Calcular gap inicial: gap = floor(5 / 1.3) = 3 (redondeado hacia abajo).
➢ Recorrer la lista con gap = 3:
    - Comparar A[0] (5) con A[3] (1): Intercambiar (lista parcial: {2, 5, 4, 1, 3}).

Segunda iteración:

➢ Reducir gap: gap = floor(3 / 1.3) = 2 (redondeado hacia abajo).
➢ Recorrer la lista con gap = 2:
    - Comparar A[0] (2) con A[2] (4): No es necesario intercambiar.
    - Comparar A[1] (5) con A[3] (1): Intercambiar (lista parcial: {2, 1, 5, 4, 3}).

Tercera iteración:

➢ Reducir gap: gap = floor(2 / 1.3) = 1 (redondeado hacia abajo).
➢ Recorrer la lista con gap = 1 (similar al ordenamiento por burbuja):
    - Se realizan comparaciones y swaps para completar el ordenamiento final: {1, 2, 3, 4, 5}.

Ventajas:

- Puede ser más eficiente que el ordenamiento por burbuja en listas que ya estén parcialmente ordenadas.
- La lógica del algoritmo es relativamente simple y fácil de entender.

Desventajas:

- En el caso promedio, su rendimiento puede ser similar al ordenamiento por burbuja (O(n^2)).
- Existen algoritmos de ordenamiento con mejor rendimiento promedio en el caso general, como el ordenamiento rápido o el ordenamiento por mezcla.

17. **Ordenamiento por montículos (Heapsort)** es un algoritmo de ordenamiento eficiente que utiliza la estructura de datos de un montículo para ordenar una lista de elementos. Un montículo es un árbol binario completo con la propiedad de que el valor de cada nodo es mayor o igual que el valor de sus hijos (para un montículo máximo) o menor o igual (para un montículo mínimo).

- ➢ Se utiliza la lista original para construir un montículo máximo (o mínimo, según se requiera). Esto se puede lograr utilizando algoritmos específicos para la construcción de montículos, como el algoritmo de construcción ascendente o el algoritmo de construcción descendente.
- ➢ Se extrae el elemento raíz del montículo, que es el elemento de mayor valor (o menor valor, en el caso de un montículo mínimo). Esto reduce el tamaño del montículo en una unidad.
- ➢ Se reordena el montículo restante para mantener la propiedad del montículo. Esto implica mover elementos hacia abajo en el árbol hasta que se restaure la propiedad de orden.
- ➢ Se repiten los pasos de extraer el elemento máximo (o mínimo) y reordenar el montículo hasta que el montículo esté vacío. En este punto, la lista original estará ordenada.

Explicación:

La eficiencia del ordenamiento por montículos se basa en la estructura del montículo y las operaciones eficientes que se pueden realizar sobre él. La construcción inicial del montículo requiere O(n log n) de tiempo, pero las extracciones y

reordenamientos posteriores del montículo solo requieren O(log n) de tiempo por cada elemento.

Ejemplo:

Supongamos que tenemos una lista de números desordenada: {5, 2, 4, 1, 3}.

Construcción del montículo máximo:

Extracción del máximo y reordenamiento:

- Se extrae el elemento raíz (5) del montículo, quedando la lista ordenada parcialmente: {1, 2, 3, 4}.
- Se reordena el montículo restante para mantener la propiedad de montículo:

Extracción del siguiente máximo y reordenamiento:

- Se extrae el nuevo elemento raíz (4) del montículo, quedando la lista ordenada parcialmente: {1, 2, 3}.

- Se reordena el montículo restante:

Se repiten los pasos de extraer el máximo elemento del montículo y reordenar hasta que el montículo esté vacío, obteniendo la lista completamente ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Su complejidad temporal en el peor de los casos es O(n log n), lo que lo convierte en uno de los algoritmos de ordenamiento más eficientes.
- El algoritmo no requiere memoria adicional para estructuras auxiliares, ya que realiza la ordenación sobre la misma lista original.
- Dependiendo de la implementación, el algoritmo puede ser estable, preservando el orden relativo de elementos iguales.

Desventajas:

- La estructura y el funcionamiento del algoritmo pueden ser menos intuitivos para principiantes comparado con otros algoritmos de ordenamiento.
- La construcción inicial del montículo requiere O(n log n) de tiempo, lo que puede ser un inconveniente para listas muy pequeñas.

```
// Algoritmo de ordenamiento Heap Sort Java
public static void heapSort(int[] arr) {
    int n = arr.length;
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
```

```
    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

public static void heapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}

// Ejemplo de uso
int[] arr = {12, 11, 13, 5, 6, 7};
heapSort(arr);
```

```
System.out.println(Arrays.toString(arr));
```

```
# Algoritmo de Ordenamiento Heap Sort Python
def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

# Ejemplo de uso
arr = [12, 11, 13, 5, 6, 7]
heap_sort(arr)
print(arr)
```

```
// Algoritmo de ordenamiento Heap Sort Javascript
```

```
function heapSort(arr) {
    let n = arr.length;
    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }
    for (let i = n - 1; i > 0; i--) {
        [arr[0], arr[i]] = [arr[i], arr[0]];
        heapify(arr, i, 0);
    }
}

function heapify(arr, n, i) {
    let largest = i;
    let left = 2 * i + 1;
    let right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
    if (largest !== i) {
        [arr[i], arr[largest]] = [arr[largest], arr[i]];
        heapify(arr, n, largest);
    }
}

// Ejemplo de uso
let arr = [12, 11, 13, 5, 6, 7];
```

```
heapSort(arr);
console.log(arr);
```

18. **Ordenamiento Smoothsort (Ordenamiento Suave)** es un algoritmo de ordenamiento por comparación basado en montículos (Heapsort) que fue inventado por Edsger Dijkstra en 1981. Es una variante del ordenamiento por montículos diseñada para minimizar el número de comparaciones realizadas durante el proceso de ordenamiento.

- ➢ Se construye un montículo Leonardo a partir de la lista de entrada. Un montículo Leonardo es un tipo especial de montículo binario completo con la propiedad de que la altura de cada nodo es un número de Fibonacci.
- ➢ Se extrae repetidamente el elemento máximo del montículo Leonardo y lo coloca en su posición final en la lista ordenada.
- ➢ Después de extraer un elemento, se reordena el montículo Leonardo para mantener la propiedad del montículo y la altura de los nodos.

La clave de la eficiencia de Smoothsort reside en la construcción del montículo Leonardo. Los montículos Leonardo tienen la propiedad de que la altura de cada nodo es un número de Fibonacci, lo que permite realizar extracciones y reordenamientos del montículo de manera más eficiente que en un montículo binario estándar.

Ejemplo:

Supongamos que tenemos una lista de números desordenada: {5, 2, 4, 1, 3}.

Construcción del montículo Leonardo:

5

/ \

2 4

/

1

/

3

Extracción del máximo y reordenamiento:

- Se extrae el elemento raíz (5) del montículo, quedando la lista ordenada parcialmente: {1, 2, 3, 4}.
- Se reordena el montículo Leonardo para mantener la propiedad de montículo y altura de los nodos:

4

/ \

2 3

/

1

Extracción del siguiente máximo y reordenamiento:

- Se extrae el nuevo elemento raíz (4) del montículo, quedando la lista ordenada parcialmente: {1, 2, 3}.
- Se reordena el montículo Leonardo:

3

/ \

2 1

Se repiten los pasos de extraer el máximo elemento del montículo Leonardo y reordenar hasta que el montículo esté vacío, obteniendo la lista completamente ordenada: {1, 2, 3, 4, 5}.

Ventajas:

- Smoothsort puede tener un rendimiento promedio mejor que el ordenamiento por montículos clásico, especialmente para listas con algunos elementos ordenados.
- En general, Smoothsort realiza menos comparaciones que el ordenamiento por montículos, lo que puede ser beneficioso para optimizar el rendimiento.
- A medida que la lista se ordena, el algoritmo realiza transiciones suaves entre diferentes comportamientos de ordenamiento, lo que le da su nombre característico.

Desventajas:

- Al igual que el ordenamiento por montículos, Smoothsort tiene una complejidad temporal en el peor de los casos de O(n log n).
- La construcción del montículo Leonardo puede ser más compleja que la construcción de un montículo binario estándar.

19. **El algoritmo de ordenamiento Several Unique Sort** es un método de clasificación que se centra en ordenar una lista de elementos de manera que los elementos únicos aparezcan primero, seguidos de los elementos duplicados. Este algoritmo es útil cuando se desea preservar los elementos únicos en una lista, mientras se ordenan los elementos duplicados.

El algoritmo Several Unique Sort se puede implementar de la siguiente manera:

- Se recorre la lista y se identifican los elementos únicos, manteniendo un registro de cuántas veces aparece cada elemento.

- Se ordenan los elementos únicos utilizando un algoritmo de ordenamiento como Merge Sort o Quick Sort.
- Se recorre la lista original y se reemplazan los elementos duplicados por los elementos ordenados únicos.

El tiempo de ejecución de Several Unique Sort depende del algoritmo de ordenamiento utilizado para ordenar los elementos únicos. En general, tiene una complejidad de O(n log n), donde n es el número de elementos en la lista.

20. **Ordenamiento Bogosort (Ordenamiento Aleatorio)** es un algoritmo de ordenamiento humorístico e ineficiente que funciona barajando aleatoriamente una lista hasta que esté ordenada. Se le considera un ejemplo de un algoritmo de "tiempo infinito" o "complejidad aleatoria", ya que no hay garantía de que termine en un tiempo finito.

Funcionamiento:

- ➢ Se baraja la lista de elementos aleatoriamente utilizando un algoritmo de barajado aleatorio.
- ➢ Se verifica si la lista está ordenada comparando cada elemento con el siguiente.
- ➢ Si la lista no está ordenada, se repiten los pasos 1 y 2 hasta que la lista finalmente se ordene aleatoriamente.

La idea detrás de Bogosort es que, dado un número suficiente de barajados aleatorios, la lista eventualmente llegará a estar ordenada por pura casualidad. La probabilidad de que esto suceda en un número finito de barajados es extremadamente baja, lo que hace que Bogosort sea un algoritmo impracticable para cualquier lista de tamaño considerable.

Ejemplo:

Supongamos que tenemos una lista desordenada de números: {5, 2, 4, 1, 3}.

Primer barajado:

La lista se baraja aleatoriamente, dando como resultado: {4, 3, 5, 1, 2}.

Verificación de ordenamiento:

Se comprueba la lista, y se encuentra que no está ordenada.

Repetición:

Se repiten los pasos de barajado y verificación hasta que la lista se ordene aleatoriamente:

- {2, 1, 3, 4, 5}
- {3, 5, 1, 4, 2}
- ...
- {1, 2, 3, 4, 5} (¡Lista ordenada!)

Ventajas:

- La lógica del algoritmo es extremadamente simple y fácil de entender.
- No se requiere memoria adicional para estructuras auxiliares, ya que la ordenación se realiza sobre la misma lista original.

Desventajas:

- No hay garantía de que el algoritmo termine en un tiempo finito, ya que depende de la aleatoriedad de los barajados.
- El tiempo de ejecución esperado es extremadamente largo, incluso para listas pequeñas, lo que lo hace inviable para cualquier aplicación práctica.
- No hay forma de predecir cuánto tiempo tardará el algoritmo en terminar, ya que depende del azar.
-

21. **Ordenamiento de Panqueques (Pancake sorting)** también conocido como ordenamiento de crepes o hotcakes, es un algoritmo de ordenamiento lúdico que simula el proceso de ordenar una pila de panqueques utilizando solo una espátula. La espátula solo se puede insertar en la parte superior de la pila y se usa para

voltear todos los panqueques encima de ella. El objetivo es ordenar la pila de panqueques de menor a mayor tamaño utilizando sólo este método de volteo.

Funcionamiento:

- ➢ Se identifica el panqueque más grande sin ordenar en la pila.
- ➢ Se usa la espátula para voltear todos los panqueques encima del panqueque más grande, colocándolo en la parte superior de la pila.
- ➢ Se repiten los pasos 1 y 2 hasta que el panqueque más grande se encuentre en la posición correcta (en la parte inferior de la pila), lo que indica que la pila está ordenada.

La clave del ordenamiento de panqueques es que cada volteo puede reorganizar un subconjunto de la pila sin afectar el orden de los panqueques ya ordenados en la parte inferior. Al identificar y voltear repetidamente el panqueque más grande, se puede ir moviendo gradualmente los panqueques a sus posiciones correctas.

Ejemplo:

Supongamos que tenemos una pila desordenada de panqueques: [4, 3, 2, 1].

Primer volteo:

Se identifica el panqueque más grande (4) y se voltea la pila, quedando: [4, 3, 2, 1].

Segundo volteo:

Se identifica el nuevo panqueque más grande (3) y se voltea la pila superior, quedando: [3, 4, 2, 1].

Tercer volteo:

Se identifica el panqueque más grande (4) y se voltea la pila superior, quedando: [4, 3, 1, 2].

Cuarto volteo:

Se identifica el último panqueque sin ordenar (2) y se voltea la pila superior, ordenando la pila completa: [1, 2, 3, 4].

Ventajas:

- La lógica del algoritmo es simple y fácil de comprender, incluso para principiantes.
- La idea de ordenar panqueques con una espátula lo hace divertido y atractivo.
- Para pilas pequeñas, el ordenamiento de panqueques puede ser eficiente, ya que solo requiere un número relativamente pequeño de volteos.

Desventajas:

- La complejidad temporal en el peor de los casos es O(n^3), lo que lo hace muy lento para pilas grandes.
- En la práctica, la simulación del volteo de panqueques con una espátula puede tener limitaciones físicas, como la posibilidad de que los panqueques se peguen o se rompan.
- Debido a su ineficiencia en el peor de los casos, el ordenamiento de panqueques no es un algoritmo viable para ordenar grandes conjuntos de datos en aplicaciones prácticas.

1.2 ¿Qué es una Lista?

Es una estructura de datos abstracta que representa una colección ordenada de elementos. Esta estructura se caracteriza por las siguientes propiedades:

- ➢ Orden: Los elementos de la lista se encuentran ordenados según un criterio específico, como el orden numérico, alfabético o por cualquier otro criterio definido. Este orden es fundamental para diferenciar una lista de otros tipos de estructuras de datos como conjuntos o diccionarios.
- ➢ Acceso secuencial: Los elementos de la lista se pueden acceder y recorrer de manera secuencial, es decir, uno a la vez, siguiendo el orden establecido. Esto significa que para acceder a un elemento específico, es necesario recorrer los elementos anteriores de la lista.

- ➢ Operaciones básicas: La lista proporciona operaciones básicas para manipular sus elementos, como:
    - Inserción: Agregar un nuevo elemento a la lista en una posición específica.
    - Eliminación: Remover un elemento existente de la lista.
    - Acceso: Obtener el valor de un elemento en una posición determinada de la lista.
    - Búsqueda: Encontrar la posición de un elemento específico dentro de la lista.
    - Recorrido: iterar a través de todos los elementos de la lista en el orden en que están almacenados.
- ➢ Implementaciones: Existen diferentes formas de implementar una lista en lenguajes de programación, siendo las más comunes.
    - Listas enlazadas: los elementos se almacenan en nodos independientes que se conectan entre sí mediante referencias. Esta es flexible y permite la inserción y eliminación de elementos en cualquier posición sin necesidad de reordenar la lista.
    - Arreglos: Los elementos se almacenan en posiciones contiguas de la memoria. Esta implementación ofrece acceso rápido a elementos por índice, pero la inserción y eliminación de elementos en posiciones intermedias puede requerir reordenar la lista.
- ➢ Aplicaciones: Las listas son estructuras de datos versátiles y ampliamente utilizadas en diversos escenarios de programación, como:
    - Almacenamiento de datos ordenados: Se utilizan para almacenar y organizar conjuntos de datos ordenados, como listas de nombres, registros de transacciones o secuencias de números.
    - Implementación de Algoritmos: Sirven como base para la implementación de algoritmos de ordenamiento, búsqueda y procesamiento de datos.
    - Desarrollo de aplicaciones: Se utilizan en el desarrollo de aplicaciones que requieren almacenar, organizar y manipular colecciones de datos ordenados, como reproductores de música, editores de texto o sistemas de gestión de bases de datos.

1.2.1. Desarrollar un ejemplo en al menos 3 lenguajes de programación, en el cual se pueda usar una lista personalizada y sus operaciones básicas (no es la que trae el lenguaje dentro de su SDK). El ejemplo debe ser el mismo escrito en los 3 lenguajes que usted elija.

```java
// Ejemplo de una lista personalizada genérica en Java
import java.util.ArrayList;

class CustomList<T> {       // Clase genérica
    private ArrayList<T> list;

    public CustomList() {
        list = new ArrayList<>();   // Inicializa la lista
    }

    public void add(T element) {
        list.add(element);  // Agrega un elemento a la lista
    }

    public void remove(T element) {
        list.remove(element);   // Remueve un elemento de la lista
    }

    public int size() {
        return list.size();    // Devuelve el tamaño de la lista
    }
```

```
    public boolean contains(T element) {
        return list.contains(element);  // Devuelve true si el elemento está en la lista, false en caso contrario
    }
}

public class Main {   // Clase principal
    public static void main(String[] args) {    // Método principal
        CustomList<Integer> customList = new CustomList<>();    // Instancia de la clase CustomList
        customList.add(1);  // Agrega elementos a la lista
        customList.add(2);
        customList.add(3);
            System.out.println("Size of list: " + customList.size());    // Imprime el tamaño de la lista
        System.out.println("Contains 2: " + customList.contains(2));
        customList.remove(2);   // Remueve el elemento 2 de la lista
        System.out.println("Contains 2 after removal: " + customList.contains(2));
    }
}

# Ejemplo lista personalizada con Python
class CustomList:
    def __init__(self):
        self.list = []

    def add(self, element):     # Método para agregar un elemento a la lista
```

```
        self.list.append(element)

    def remove(self, element):  # Método para remover un elemento de la lista
        self.list.remove(element)

    def size(self):           # Método para obtener el tamaño de la lista
        return len(self.list)

    def contains(self, element):    # Método para verificar si un elemento está en la lista
        return element in self.list

custom_list = CustomList()  # Crear una lista personalizada
custom_list.add(1)          # Agregar elementos a la lista
custom_list.add(2)
custom_list.add(3)
print("Size of list:", custom_list.size())
print("Contains 2:", custom_list.contains(2))
custom_list.remove(2)
print("Contains 2 after removal:", custom_list.contains(2))

// Ejemplo lista personalizada Javascript
class CustomList {
    constructor() {
        this.list = [];
    }
```

```
    add(element) {      // Agregar un elemento a la lista
        this.list.push(element);
    }

    remove(element) {   // Remover un elemento de la lista
        const index = this.list.indexOf(element);
        if (index !== -1) {
            this.list.splice(index, 1);
        }
    }

    size() {        // Obtener el tamaño de la lista
        return this.list.length;
    }

    contains(element) {     // Verificar si un elemento está en la lista
        return this.list.includes(element);
    }
}

let customList = new CustomList();
customList.add(1);
customList.add(2);
customList.add(3);
console.log("Size of list:", customList.size());
console.log("Contains 2:", customList.contains(2));
customList.remove(2);
```

```
console.log("Contains 2 after removal:", customList.contains(2));
```

Realizar un programa con listas que permita simular el siguiente proceso manual (la lista construida en el punto #1.2):

Una persona llamada Fulanito de Tal realiza una actividad comercial basada en la compra y venta de productos.

● Fulanito anota en una libreta información importante por cada producto que compra, no sin antes revisar si dicho producto ya había sido comprado (anotado) anteriormente, algunos de los datos son: código, nombre, marca, color, precio de compra, precio de venta, porcentaje de descuento máximo, unidades en existencia, métricas de medida (kg, litros, metros, unidad) y categoría.

● Fulanito anota en otra libreta información importante sobre cada compra que hace por cada producto: consecutivo, fecha, proveedor al que le compró, el código del producto que ha comprado, el precio que lo ha comprado, las cantidades (kg, litros, metros, unidades) y realiza el cálculo de la cantidad de dinero que debe pagar por esa compra, para eso tiene en cuenta el impuesto del IVA que es del 19% para todos los productos, anotando el valor a pagar sin IVA, el

valor a pagar por concepto de IVA y el valor total a pagar.

● Por cada compra que Fulanito realiza toma la libreta donde tiene anotados los productos que ha comprado, busca el producto por su código para actualizar el precio de compra, el precio de venta (el 40% del precio de compra), el porcentaje máximo de descuento y las unidades de existencia (suma las unidades compradas a las a unidades actuales)

● Fulanito inicia su proceso de mercadeo de sus productos con el fin de obtener personas (Clientes) interesados en comprarle. Para evitar publicitar productos que no tiene en su inventario, Fulanito toma como base la información que ha ido anotando en sus libretas, entonces, busca la información anotada previamente en sus libretas para consultar los productos cuyas unidades de existencias tienen un valor mayo que 0. Luego de haber hecho ese filtro de datos, los ordena

alfabéticamente y los agrupa por categorías, consignando la información en un archivo el cual envía a una empresa de publicidad para que hagan difusión de esa información de forma masiva.

● Fulanito posee una libreta donde anota todos los clientes que ha conseguido, algunos son solo referidos (no han comprado productos aun), en esa libreta va anotando la cédula, el nombre y los dos apellidos, el género, la fecha de nacimiento, el número telefónico, el email y la dirección, pero también anota el estado de cada cliente como VIP (los que han comprando algo) y como Referido (a los que no han comprado)

● Fulanito toma esos datos para enviar información de los productos a sus clientes por medio de mensajes de texto, correos electrónicos, folletos de impresos o llamadas telefónicas.

● Cuando un cliente está interesado en comprar algún producto, Fulanito toma otra libreta y anota la siguiente información: consecutivo, fecha, cédula del cliente, medio de pago, modalidad (directa o domicilio). Por cada artículo a comprar Fulanito anota: código del producto, precio de venta, cantidad a comprar según las unidades de medida, el valor de descuento aplicado. Luego por cada producto realiza el cálculo de la cantidad de dinero que debe cobrar por esa venta, teniendo en cuenta el impuesto del IVA, anotando el resultado de los siguientes cálculos: El valor a cobrar sin IVA, el valor que debe descontar, el valor a cobrar por IVA, y el valor total a cobrar.

● En caso de que la venta se realice, Fulanito consulta la libreta de ventas para buscar el código de esa venta y modifica al estado de la venta para cambiarle el valor por éxito, pero, si la venta no se realiza, entonces, Fulanito cambia el estado a cancelada y agrega un breve motivo.

● Luego de actualizar la venta, Fulanito toma la libreta de productos para buscar cada uno de los productos vendidos para disminuir las cantidades existentes de acuerdo a las cantidades vendidas.

● Por último, Fulanito toma la libreta de clientes para cambiarle pasar el cliente a VIP en caso de que esta sea su primera compra.

● Al finalizar cada venta, Fulanito revisa la información de los productos mostrando sólo aquellos cuya existencia es inferior a 5 (según la unidad de medida) y toma esa información para realizar compras y abastecerse su inventario con esos artículos.

● Con el objeto de motivar a los clientes y aumentar las ventas, Fulanito revisa la libreta de ventas para buscar cuales son los clientes que más compras han hecho (superior al promedio de compras) para informarles que tienen un 10% de descuento en la próxima compra

● Los clientes que han comprado poco (menos del promedio de compras) les informa que tienen un 15% en su próxima compra

● Y a los clientes que no han comprado les informa que tienen un 25% en su primera compra

● Por último, Fulanito consulta la información que tiene en sus libretas para poder calcular el balance financiero de su negocio, pues le interese conocer la siguiente información:

- Total de dinero invertido en los productos que ha comprado desde el inicio.
- Total de dinero que ha recuperado en ventas desde el inicio.
- Total de dinero ganancia que ha obtenido desde el inicio.
- Total de dinero en descuento que ha generado desde el inicio.
- Total de dinero que ha pagado como impuesto IVA

● Para saber en qué productos debe invertir más en las próximas compras, Fulanito consulta los productos que superan el 70% de las ventas.

● Los productos que no superan el 70% de las ventas les aplica un 35% de descuento

IrinaBallesteros/Estructuras_de_Datos (github.com)

CONCLUSIONES

En este viaje de aprendizaje, hemos explorado 21 algoritmos de ordenamiento haciendo énfasis en 5 seleccionados según indicación y las estructuras de datos tipo lista, adquiriendo habilidades valiosas para la programación eficiente.

Los algoritmos de ordenamiento nos han brindado herramientas para organizar conjuntos de datos de manera ascendente o descendente, permitiéndonos analizar y procesar información de forma más rápida y efectiva. Dominamos técnicas como Bubble Sort, Insertion Sort, Selection Sort, Merge Sort y Quick Sort, cada una con sus fortalezas, desventajas y aplicaciones específicas.

La elección del algoritmo de ordenamiento adecuado para una aplicación específica depende de varios factores, como el tamaño del conjunto de datos, la distribución de los datos, las restricciones de tiempo y memoria, y las preferencias del desarrollador. Es importante analizar estos factores y considerar las características y el rendimiento de cada algoritmo antes de tomar una decisión.

Las estructuras de datos tipo lista, por su parte, nos han permitido almacenar y manipular secuencias de elementos de manera dinámica. Aprendimos a crear, insertar, eliminar y buscar elementos en listas enlazadas y listas estáticas, comprendiendo sus ventajas y desventajas en diferentes escenarios.

La combinación de estos dos conceptos fundamentales nos ha equipado para abordar problemas de programación con mayor solvencia. Podemos ordenar listas de datos, buscar elementos específicos, procesar información de manera eficiente y optimizar el rendimiento de nuestros programas.

Este aprendizaje nos abre las puertas a un mundo de posibilidades en el ámbito de la programación. Podemos aplicar lo aprendido en el desarrollo de aplicaciones web, sistemas de gestión de datos, algoritmos de búsqueda, análisis de información y mucho más.

Sin embargo, la programación es un campo en constante evolución, y la actualización constante de nuestros conocimientos es esencial para seguir avanzando. El camino del conocimiento no termina aquí

BIBLIOGRAFÍA

➢ Introducción a los algoritmos" de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein.

➢ Algoritmos" de Robert Sedgewick y Kevin Wayne

➢ https://www.geeksforgeeks.org/sorting-algorithms/