UNIVERSIDAD NACIONAL DEL ALTIPLANO
FACULTAD DE INGENIERÍA MECÁNICA ELÉCTRICA Y DE SISTEMAS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

INFORME DE PROYECTO
CURSO:
ANÁLISIS Y DISEÑO DE ALGORITMOS
DOCENTE:
Ing. INGALUQUE ARAPA MARGA ISABEL
ALUMNO:
CARI MULLISACA ALEX
QUISPE CHATA CARLOS DAVID
SEMESTRE:
V SEMESTRE
PUNO – 2024

1. Introducción

En la actualidad, la gestión eficiente de la información es crucial para el éxito de cualquier empresa, y uno de los aspectos más importantes es el manejo de los datos de los clientes. La administración correcta de esta información permite a las empresas optimizar sus procesos internos, mejorar la atención al cliente y, en última instancia, aumentar sus ingresos. En este contexto, la implementación de algoritmos eficientes para la ordenación y búsqueda de datos se convierte en una herramienta fundamental.

El presente informe documenta el desarrollo de un programa para la gestión de clientes, que se llevó a cabo como parte del curso "Análisis y Diseño de Algoritmos" en la Universidad Nacional del Altiplano. Este proyecto se enfoca en la implementación de algoritmos de ordenación y búsqueda que permiten organizar y acceder de manera eficiente a los datos de compra de los clientes, como fechas y montos de transacciones.

Los algoritmos seleccionados para este proyecto fueron Heapsort, Quicksort y Búsqueda Binaria, cada uno con características y ventajas específicas que los hacen adecuados para distintos escenarios. Estos algoritmos fueron implementados para permitir la ordenación de los datos de los clientes por fecha de compra o por monto de compra, lo que facilita la gestión y optimización de los procesos de búsqueda en el sistema.

Además, el proyecto evaluó el rendimiento de estos algoritmos bajo diferentes circunstancias, como listas ordenadas aleatoriamente, listas preordenadas e inversamente ordenadas. También se analizaron factores como el uso de memoria y el tiempo de ejecución, lo cual es clave para determinar la eficiencia de los algoritmos en la práctica.

El objetivo principal de este informe es demostrar cómo la implementación correcta de algoritmos de ordenación y búsqueda puede mejorar significativamente la eficiencia en la administración de datos, proporcionando a los futuros ingenieros una base sólida sobre la cual construir soluciones para problemas de gestión de datos en diversos contextos empresariales.

2. Objetivos

El desarrollo de este proyecto tiene como finalidad la implementación y evaluación de algoritmos avanzados de ordenación y búsqueda, aplicados a la gestión de clientes en un entorno empresarial. Los objetivos específicos que se persiguen son los siguientes:

1. **Implementar algoritmos eficientes de ordenación y búsqueda**: Desarrollar e integrar en el programa de gestión de clientes los algoritmos Heapsort, Quicksort y Búsqueda Binaria, que permitan ordenar grandes volúmenes de datos (como fechas de compra y montos) de manera eficiente y con un uso óptimo de los recursos del sistema. La correcta implementación de estos algoritmos busca mejorar la rapidez y precisión en la organización y recuperación de datos.
2. **Evaluar el rendimiento de los algoritmos**: Realizar una comparación detallada del rendimiento de los algoritmos Quicksort, Heapsort y Búsqueda Binaria, considerando aspectos como el tiempo de ejecución, el uso de memoria y la eficiencia en diferentes escenarios (listas desordenadas, preordenadas e inversamente ordenadas). Este análisis busca identificar las fortalezas y debilidades de cada algoritmo en contextos específicos, proporcionando una guía para seleccionar el más adecuado según las necesidades de cada situación.
3. **Optimizar la gestión de clientes mediante la ordenación de sus datos**: Garantizar que el sistema de gestión de clientes pueda manejar eficientemente grandes volúmenes de información, permitiendo la rápida organización y búsqueda de datos de clientes, como sus compras, fechas y montos. Este objetivo incluye no solo la mejora en la velocidad de respuesta del sistema, sino también la reducción

del uso de recursos computacionales, lo que es clave para sistemas que operan con bases de datos grandes.

4. **Proporcionar flexibilidad en los criterios de ordenación y búsqueda**: Asegurar que el sistema sea capaz de ordenar y buscar datos utilizando diferentes criterios, como la fecha de compra o el monto gastado, dependiendo de los requisitos específicos de la empresa o del usuario. La flexibilidad en los criterios de búsqueda permitirá a los usuarios del sistema tomar decisiones más informadas basadas en el análisis eficiente de los datos.
5. **Desarrollar una plataforma escalable y eficiente**: Construir un sistema de gestión de clientes que sea capaz de escalar en función de las necesidades del negocio, manteniendo una alta eficiencia incluso a medida que crece el número de clientes y la cantidad de datos almacenados. El proyecto busca no solo optimizar el rendimiento actual, sino también sentar las bases para futuras ampliaciones del sistema.

Algoritmos Utilizados

En este programa de gestión de clientes, se implementaron los siguientes algoritmos para la ordenación de los datos de compra y la búsqueda eficiente de clientes. A continuación, se describen los algoritmos y sus principales características:

Heapsort: Algoritmo de Ordenación

El **Heapsort** es un algoritmo de ordenación basado en una estructura de datos llamada **montículo** o **heap**, la cual es un árbol binario completo que satisface la propiedad de montículo. Esta propiedad asegura que el valor de cada nodo sea mayor (o menor) que el de sus hijos, lo que permite acceder fácilmente al elemento máximo o mínimo de la estructura en tiempo constante O(1)O(1)O(1). A continuación, se describe más a fondo cómo funciona el algoritmo y algunas de sus principales características.

Funcionamiento:

1. **Construcción del montículo**: Inicialmente, se reorganizan los elementos del arreglo a ordenar de manera que se convierta en un montículo. Existen dos tipos de montículos:
    - **Montículo máximo**: En el que cada nodo padre es mayor o igual que sus hijos.
    - **Montículo mínimo**: En el que cada nodo padre es menor o igual que sus hijos.
2. Para el Heapsort, se suele utilizar el montículo máximo, lo que permite que el valor máximo esté siempre en la raíz (primer elemento) del árbol binario.
3. **Extracción del máximo**: Una vez que el arreglo ha sido convertido en un montículo máximo, el siguiente paso es intercambiar el valor máximo (que está en la raíz) con el último elemento del arreglo. Este proceso reduce el tamaño del montículo en una unidad, ya que el último elemento ahora está en su posición correcta en el arreglo ordenado.

4. **Reajuste del montículo**: Después de extraer el valor máximo, el montículo se desorganiza, por lo que es necesario reajustar el árbol para restaurar la propiedad del montículo. Este proceso se denomina **heapify** y se aplica sobre el nodo raíz. El heapify reorganiza los elementos de manera que el mayor de los hijos pase a ser la nueva raíz, y el proceso se repite hasta que el montículo vuelva a ser válido.
5. **Repetición**: Los pasos de extracción y reajuste se repiten hasta que todos los elementos estén ordenados. Al final del proceso, el arreglo estará ordenado de manera ascendente (si se utilizó un montículo máximo).

Ejemplo:

Consideremos el arreglo [4,10,3,5,1][4, 10, 3, 5, 1][4,10,3,5,1]. El Heapsort lo ordenaría de la siguiente manera:

1. Convertir el arreglo en un montículo máximo: [10,5,3,4,1][10, 5, 3, 4, 1][10,5,3,4,1].
2. Intercambiar el valor máximo con el último elemento: [1,5,3,4,10][1, 5, 3, 4, 10][1,5,3,4,10].
3. Aplicar heapify para restaurar el montículo: [5,4,3,1,10][5, 4, 3, 1, 10][5,4,3,1,10].
4. Repetir el proceso hasta que el arreglo esté ordenado: [1,3,4,5,10][1, 3, 4, 5, 10][1,3,4,5,10].

Complejidad Computacional

La complejidad temporal del Heapsort es **O(nlogn)O(n \log n)O(nlogn)** en el mejor, promedio y peor de los casos. Esto se debe a que la operación de construir el montículo tiene una complejidad de O(n)O(n)O(n), y cada extracción del valor máximo (o mínimo) y posterior reajuste del montículo tiene una complejidad de O(logn)O(\log n)O(logn). Dado que se realiza una extracción por cada elemento, el número total de operaciones es O(nlogn)O(n \log n)O(nlogn).

Espacio:

El Heapsort es un algoritmo **in-place**, lo que significa que no requiere espacio adicional significativo más allá del que ya ocupa el propio arreglo. Esto lo convierte en una opción eficiente en términos de uso de memoria, ya que no necesita crear copias adicionales del arreglo ni estructuras de datos adicionales.

Comparación con otros algoritmos:

- **Quicksort**: El Quicksort también tiene una complejidad promedio de O(nlogn)O(n \log n)O(nlogn), pero en el peor de los casos (cuando la elección del pivote es mala), su complejidad puede degradarse a O(n2)O(n^2)O(n2). Sin embargo, en la práctica suele ser más rápido que el Heapsort debido a su menor número de comparaciones y movimientos de datos.
- **Mergesort**: Mergesort también tiene una complejidad de O(nlogn)O(n \log n)O(nlogn) en todos los casos, y es más eficiente para grandes volúmenes de datos

debido a su menor constante oculta. Sin embargo, no es un algoritmo in-place, ya que requiere memoria adicional para combinar los subarreglos.
- **Bubble Sort / Selection Sort**: Estos algoritmos tienen una complejidad de O(n2)O(n^2)O(n2), lo que los hace significativamente más lentos para grandes conjuntos de datos. A pesar de ser más fáciles de implementar, son ineficientes en la práctica.

Ventajas de Heapsort:

- **Complejidad garantizada**: Heapsort garantiza una complejidad de O(nlogn)O(n \log n)O(nlogn) en todos los casos (mejor, promedio y peor), lo que lo hace más predecible que Quicksort, que puede tener un peor caso de O(n2)O(n^2)O(n2).
- **Uso eficiente de memoria**: Como Heapsort es un algoritmo in-place, no requiere un uso excesivo de memoria adicional. Esto lo convierte en una buena opción cuando el espacio en memoria es limitado.
- **Aplicaciones en estructuras de datos**: El montículo es útil para implementar colas de prioridad, donde las operaciones de inserción y extracción del elemento máximo (o mínimo) se benefician de la estructura de montículo.

Desventajas de Heapsort:

- **Rendimiento práctico**: Aunque Heapsort tiene una complejidad asintótica eficiente, en la práctica suele ser más lento que Quicksort debido a su mayor número de comparaciones y movimientos de datos, especialmente en arquitecturas donde el acceso a la memoria es costoso.
- **Requiere más operaciones de intercambio**: A diferencia de Quicksort, que puede reducir el número de intercambios a través de particiones eficientes, Heapsort requiere realizar múltiples intercambios para mantener la propiedad del montículo, lo que puede afectar su rendimiento en ciertas aplicaciones.
    -

2. Quicksort

- **Descripción**: El algoritmo Quicksort utiliza una estrategia de **división y conquista**. Consiste en elegir un elemento llamado **pivote**, dividir la lista en dos sublistas: una con elementos menores que el pivote y otra con elementos mayores, y luego aplicar recursivamente el proceso de ordenación en las sublistas. Finalmente, los resultados se combinan para formar la lista ordenada.
- **Ventajas**:
    - Es uno de los algoritmos más rápidos en la práctica para la mayoría de los casos debido a su eficiente partición de los datos.
    - Tiene una complejidad temporal promedio de **O(n log n)**, aunque en el peor de los casos puede ser **O(n²)** si no se elige bien el pivote.
- **Desventajas**:
    - **No es un algoritmo in-place** en su versión más sencilla, ya que puede requerir espacio adicional para las sublistas.

    - La eficiencia depende mucho de la elección del pivote. Si se elige un mal pivote repetidamente, el rendimiento puede verse afectado negativamente.

3. Búsqueda Binaria

- **Descripción**: La búsqueda binaria es un algoritmo de búsqueda eficiente que funciona dividiendo repetidamente la lista ordenada en dos mitades. Comienza comparando el valor buscado con el elemento central de la lista, eliminando la mitad que no contiene el valor y repitiendo el proceso con la mitad restante hasta encontrar el valor deseado o agotar las posibilidades.
- **Requisitos**:
    - La lista debe estar **previamente ordenada**. Para este programa, la búsqueda binaria solo es aplicable después de haber ordenado los datos con Heapsort o Quicksort.
- **Ventajas**:
    - Tiene una complejidad temporal de **O(log n)**, lo que lo convierte en uno de los métodos más rápidos para búsquedas en listas grandes.
    - Eficiente en términos de espacio, ya que no requiere estructuras adicionales significativas.
- **Desventajas**:
    - Solo es aplicable a listas ordenadas, lo que implica un paso previo de ordenación si los datos no lo están.

3. Algoritmos Utilizados

En este proyecto se han implementado tres algoritmos clave que permiten gestionar de manera eficiente los datos de los clientes, garantizando tanto la velocidad en la ordenación como la precisión en las búsquedas. A continuación, se describen en detalle los algoritmos utilizados y sus características principales:

1. Heapsort

El algoritmo **Heapsort** es un método de ordenación basado en la estructura de datos conocida como *montículo* o *heap*. Un heap es un árbol binario completo en el que cada nodo tiene un valor mayor (en un *max-heap*) o menor (en un *min-heap*) que sus nodos hijos. El algoritmo Heapsort se beneficia de esta estructura para garantizar que el valor máximo o mínimo esté siempre en la raíz del árbol, lo que permite extraerlo de manera eficiente.

Principales Características:

- **Complejidad Temporal**: Heapsort tiene una complejidad temporal de O(n log n), tanto en el mejor como en el peor de los casos. Esto se debe a que la construcción del heap inicial toma O(n) tiempo y la extracción de cada elemento del heap requiere O(log n).

- **Algoritmo In-place**: Heapsort es un algoritmo in-place, lo que significa que no requiere espacio adicional significativo para llevar a cabo la ordenación, más allá del necesario para el array original. Esto lo hace adecuado para sistemas con limitaciones de memoria.

- **Uso de un Heap**: La clave de su eficiencia es la construcción del heap, que reorganiza los elementos del array de manera que el mayor o menor elemento se pueda acceder y extraer rápidamente. Luego, se ajusta el heap restante para mantener esta propiedad hasta que todos los elementos estén ordenados.

Ventajas:

- Excelente uso de la memoria debido a su naturaleza in-place.
- Ofrece un rendimiento estable con O(n log n) en todos los casos, sin importar la distribución inicial de los datos.

Desventajas:

- Aunque es eficiente en términos de memoria, en la práctica puede ser más lento que Quicksort debido al mayor número de comparaciones realizadas en cada paso del proceso.

Quicksort: Algoritmo de Ordenación

El **Quicksort** es un algoritmo de ordenación muy eficiente que sigue el paradigma de **divide y vencerás**. Fue desarrollado por Tony Hoare en 1959 y, a pesar de su sencillez, es uno de los algoritmos más rápidos en la práctica para la mayoría de los casos. Su eficiencia proviene de su capacidad para dividir el problema en subproblemas más pequeños de manera recursiva, lo que permite ordenar grandes volúmenes de datos con rapidez.

Funcionamiento:

1. **Elección del pivote**: En cada llamada recursiva del algoritmo, se elige un elemento de la lista a ordenar llamado **pivote**. Este pivote se utiliza para dividir el arreglo en dos subarreglos.
2. **Partición**: El siguiente paso es reorganizar los elementos del arreglo de tal manera que todos los elementos menores que el pivote queden a su izquierda, y todos los mayores queden a su derecha. A este proceso se le conoce como **partición**.
    - Existen varias formas de realizar la partición, pero una de las más comunes es el **Esquema de Lomuto**, donde se recorre el arreglo y se intercambian los elementos conforme se detectan que están en el lado incorrecto del pivote.
3. **Ordenación recursiva**: Después de la partición, el pivote estará en su posición correcta en el arreglo. El Quicksort se aplica recursivamente a las dos sublistas formadas a cada lado del pivote (es decir, los elementos menores que el pivote y los elementos mayores que el pivote).
4. **Combinación**: Una vez que se han ordenado ambas sublistas, no es necesario realizar ninguna combinación explícita, ya que el propio proceso de partición garantiza que los elementos están en su lugar correcto.

El proceso termina cuando las sublistas se han reducido a un único elemento o están vacías, momento en el que todos los elementos estarán ordenados.

Ejemplo:

Consideremos el arreglo [9,7,5,11,12,2,14,3,10,6][9, 7, 5, 11, 12, 2, 14, 3, 10, 6][9,7,5,11,12,2,14,3,10,6] y tomemos el último elemento como pivote.

1. Se elige el pivote: 6.
2. Se particiona el arreglo: después de reordenar los elementos en torno al pivote, obtenemos [5,2,3,6,12,14,11,9,10,7][5, 2, 3, 6, 12, 14, 11, 9, 10, 7][5,2,3,6,12,14,11,9,10,7], con los elementos menores que 6 a su izquierda y los mayores a su derecha.
3. Se aplica recursivamente el mismo proceso a las sublistas [5,2,3][5, 2, 3][5,2,3] y [12,14,11,9,10,7][12, 14, 11, 9, 10, 7][12,14,11,9,10,7], hasta que todo el arreglo esté ordenado: [2,3,5,6,7,9,10,11,12,14][2, 3, 5, 6, 7, 9, 10, 11, 12, 14][2,3,5,6,7,9,10,11,12,14].

Complejidad Computacional

La complejidad temporal de Quicksort depende en gran medida de cómo se elija el pivote:

- **Mejor caso**: Si el pivote siempre divide la lista en dos partes de tamaño aproximadamente igual, el número de comparaciones es aproximadamente O(nlogn)O(n \log n)O(nlogn), lo que ocurre cuando el pivote es cercano al valor mediano de la lista.
- **Caso promedio**: En la mayoría de las implementaciones, el caso promedio también es de O(nlogn)O(n \log n)O(nlogn), ya que en la mayoría de los escenarios prácticos los pivotes suelen dividir el arreglo de manera razonablemente equilibrada.
- **Peor caso**: Si el pivote se elige mal repetidamente (por ejemplo, si siempre se elige el valor más pequeño o más grande de la lista), el algoritmo puede degenerar a una complejidad de O(n2)O(n^2)O(n2). Esto ocurre, por ejemplo, cuando la lista ya está ordenada o casi ordenada, y se elige siempre el primer o el último elemento como pivote.

Espacio:

El Quicksort puede ser **in-place** si se implementa de manera que utilice la misma lista para hacer las particiones y no requiere sublistas adicionales. Sin embargo, la versión básica puede utilizar espacio adicional para almacenar las sublistas resultantes de cada partición. En su forma más eficiente, el uso de espacio adicional es O(logn)O(\log n)O(logn) debido a la recursividad.

Optimización del Pivote:

Para evitar el peor caso de O(n2)O(n^2)O(n2), se suelen emplear técnicas para elegir el pivote de manera más inteligente:

- **Elección aleatoria**: Se elige el pivote al azar, lo que reduce significativamente la probabilidad de que se repita el peor caso.

- **Mediana de tres**: Se selecciona el pivote como la mediana de tres elementos (por ejemplo, el primero, el último y el del medio), lo que también mejora la eficiencia en listas que ya están parcialmente ordenadas.

Ventajas de Quicksort:

- **Rendimiento rápido en la práctica**: A pesar de que el Quicksort tiene un peor caso de O(n2)O(n^2)O(n2), en la práctica se comporta de manera extremadamente eficiente y suele superar a otros algoritmos de ordenación como Heapsort y Mergesort en la mayoría de las implementaciones.
- **Divide y vencerás**: El enfoque recursivo de divide y vencerás permite que el algoritmo sea muy eficaz en la gestión de datos grandes y complejos.
- **In-place (en su versión optimizada)**: Con la implementación correcta, Quicksort puede ser un algoritmo in-place, lo que significa que no requiere espacio adicional significativo, haciendo un uso muy eficiente de la memoria.
- **Adaptabilidad**: El Quicksort se puede adaptar y optimizar para diferentes tipos de datos y aplicaciones. La elección del pivote es un ejemplo de cómo pequeñas modificaciones pueden mejorar drásticamente el rendimiento.

Desventajas de Quicksort:

- **Peor caso O(n2)O(n^2)O(n2)**: Aunque es raro en la práctica, el Quicksort puede degenerar a O(n2)O(n^2)O(n2) si los pivotes se eligen mal repetidamente. Esto puede ser problemático si no se optimiza adecuadamente la elección del pivote.
- **No siempre in-place en versiones simples**: Las versiones más sencillas de Quicksort no son in-place, ya que utilizan sublistas adicionales para almacenar los elementos menores y mayores que el pivote. Sin embargo, este problema se puede solucionar con implementaciones optimizadas.
- **Problemas con listas pequeñas**: Para listas pequeñas, el overhead de las llamadas recursivas puede hacer que Quicksort sea menos eficiente que algoritmos más simples como Insertion Sort. Una práctica común es cambiar a Insertion Sort cuando los subarreglos alcanzan un tamaño pequeño.
-

Búsqueda Binaria: Algoritmo de Búsqueda Eficiente

La **Búsqueda Binaria** es un algoritmo eficiente para encontrar un elemento en una lista **ordenada**. En lugar de recorrer la lista de manera secuencial, como haría una búsqueda lineal, la búsqueda binaria divide repetidamente la lista en dos mitades y se concentra en la mitad donde podría estar el valor buscado, descartando la otra mitad. Este proceso de división continua reduce significativamente el número de elementos a considerar, haciendo que la búsqueda sea extremadamente rápida, especialmente en listas grandes.

Funcionamiento:

1. **Precondición: Lista Ordenada**: Para que la búsqueda binaria funcione correctamente, es imprescindible que la lista esté previamente ordenada (ya sea de manera ascendente o descendente). Si no lo está, se debe ordenar primero utilizando un algoritmo de ordenación, como **Quicksort** o **Heapsort**.
2. **División del intervalo**: El algoritmo comienza tomando los dos extremos de la lista, es decir, el índice del primer y último elemento. Luego calcula el índice del **elemento central** como la media de estos dos extremos.
3. **Comparación con el valor central**: El valor buscado se compara con el valor en el elemento central:
    - Si el valor buscado es **igual** al valor del elemento central, el algoritmo ha encontrado el valor, y retorna el índice de ese elemento.
    - Si el valor buscado es **menor** que el valor del elemento central, se repite el proceso en la **mitad izquierda** de la lista.
    - Si el valor buscado es **mayor** que el valor del elemento central, se continúa la búsqueda en la **mitad derecha** de la lista.
4. **Repetición**: Este proceso de división se repite hasta que se encuentra el valor o hasta que los extremos del intervalo se cruzan (es decir, cuando no quedan más elementos que comparar).
5. **Resultado**: Si el valor se encuentra en la lista, el algoritmo retorna su posición. Si no se encuentra, el algoritmo indica que el valor no está presente en la lista.

Ejemplo:

Supongamos que tenemos la lista ordenada [1,3,5,7,9,11,13,15][1, 3, 5, 7, 9, 11, 13, 15][1,3,5,7,9,11,13,15] y queremos encontrar el número 9.

1. El índice central es el valor en la posición 0+72=3 \frac{0 + 7}{2} = 320+7=3, por lo tanto, el valor central es 7.
2. Como 9 es mayor que 7, ahora buscamos en la mitad derecha de la lista, es decir, [9,11,13,15][9, 11, 13, 15][9,11,13,15].
3. El nuevo índice central es 4+72=5 \frac{4 + 7}{2} = 524+7=5, por lo tanto, el valor central ahora es 11.
4. Como 9 es menor que 11, ahora buscamos en la mitad izquierda de esta nueva sublista, es decir, [9][9][9].
5. Finalmente, 9 coincide con el valor central, por lo que la búsqueda binaria ha encontrado el valor en la posición 4.

Complejidad Computacional

La búsqueda binaria es muy eficiente gracias a su estrategia de dividir el problema a la mitad en cada iteración. La complejidad temporal de la búsqueda binaria es O(logn)O(\log n)O(logn), donde nnn es el número de elementos en la lista. Esto significa que incluso para listas grandes, el número de comparaciones necesarias para encontrar un valor es muy pequeño.

Por ejemplo, en una lista de n=1,000,000n = 1,000,000n=1,000,000 elementos, la búsqueda binaria solo necesita realizar aproximadamente 20 comparaciones en el peor de los casos para encontrar un elemento o confirmar que no está presente.

- **Mejor caso**: El mejor caso ocurre cuando el valor buscado es el valor central en la primera iteración, lo que tiene una complejidad de O(1)O(1)O(1).
- **Peor caso**: En el peor caso, se reducirá la lista a la mitad hasta que solo quede un elemento, lo que llevará O(logn)O(\log n)O(logn) comparaciones.

Ventajas de la Búsqueda Binaria:

1. **Extremadamente rápida para listas ordenadas**: La búsqueda binaria es mucho más eficiente que una búsqueda secuencial en listas grandes. Mientras que una búsqueda lineal tendría que comparar cada elemento de la lista uno por uno, la búsqueda binaria reduce el número de comparaciones a una fracción de los elementos originales.
2. **Bajo uso de memoria**: El algoritmo no necesita ninguna estructura de datos adicional compleja. Solo requiere algunas variables para los índices y el valor buscado, lo que significa que su uso de memoria adicional es muy bajo, ya que no necesita sublistas u otras estructuras auxiliares.
3. **Escalabilidad**: La búsqueda binaria funciona bien incluso para listas muy grandes. A medida que el tamaño de la lista crece, el número de comparaciones aumenta solo de manera logarítmica, lo que significa que incluso en listas con millones de elementos, la búsqueda binaria sigue siendo rápida.

Desventajas de la Búsqueda Binaria:

1. **Requiere lista ordenada**: Uno de los principales inconvenientes de la búsqueda binaria es que solo funciona en listas previamente ordenadas. Si la lista no está ordenada, se debe ordenar primero, lo que agrega un costo adicional. Dependiendo del tamaño de la lista y del algoritmo de ordenación que se utilice, el proceso de ordenar la lista podría aumentar considerablemente el tiempo total del proceso de búsqueda.
2. **Rendimiento en listas pequeñas**: Para listas muy pequeñas, la búsqueda binaria puede no ser mucho más eficiente que una búsqueda lineal simple. De hecho, en algunas implementaciones, el overhead de calcular el índice central y dividir la lista puede hacer que sea menos eficiente en comparación con una búsqueda lineal en listas muy cortas.
3. **Complejidad con listas dinámicas**: Si la lista cambia con frecuencia (se agregan o eliminan elementos), es necesario volver a ordenar la lista después de cada cambio para que la búsqueda binaria siga funcionando. En estos casos, puede que otros algoritmos de búsqueda sean más adecuados.

EXPLICACION DEL CODIGO

Función `construirHeap`

El objetivo de esta función es construir un **heap máximo** (en el que el nodo raíz siempre será el mayor valor). El heap se construye a partir del array que se desea ordenar.

```
// Heapsort

export const ordenarPorHeapSort = (array, criterio) => {

  const construirHeap = (array) => {

    let i = Math.floor(array.length / 2 - 1);

    while (i >= 0) {

      heapify(array, array.length, i);

      i--;

    }

  };



  const heapify = (array, length, i) => {

    let mayor = i;

    let izquierda = 2 * i + 1;

    let derecha = 2 * i + 2;



    // Comparar según el criterio de ordenación

    if (criterio === "fecha") {

      if (

        izquierda < length &&

        convertirFecha(array[izquierda].fechaCompra) >

          convertirFecha(array[mayor].fechaCompra)

      ) {
```

```
        mayor = izquierda;
      }
      if (
        derecha < length &&
        convertirFecha(array[derecha].fechaCompra) >
          convertirFecha(array[mayor].fechaCompra)
      ) {
        mayor = derecha;
      }
    } else if (criterio === "monto") {
      if (
        izquierda < length &&
        parseFloat(array[izquierda].monto) >
parseFloat(array[mayor].monto)
      ) {
        mayor = izquierda;
      }
      if (
        derecha < length &&
        parseFloat(array[derecha].monto) >
parseFloat(array[mayor].monto)
      ) {
        mayor = derecha;
      }
    }
```

```
    if (mayor !== i) {
      [array[i], array[mayor]] = [array[mayor], array[i]];
      heapify(array, length, mayor);
    }
  };

  construirHeap(array);

  let final = array.length - 1;
  while (final >= 0) {
    [array[0], array[final]] = [array[final], array[0]];
    heapify(array, final, 0);
    final--;
  }
  return array;
};
```

Construcción del Heap:

- Se construye un **heap máximo** a partir del array original.
- Para ello, se usa la función `heapify`, que ajusta la estructura del heap asegurando que cada subárbol cumpla con la propiedad de heap.

Ordenación:

- Se intercambia el elemento más grande (la raíz del heap) con el último elemento del array.
- Luego, se "reduce" el tamaño del heap y se vuelve a aplicar `heapify` para restaurar la propiedad de heap en la parte restante del array.
- Este proceso se repite hasta que todos los elementos están ordenados.

Criterios de Ordenación:

- El algoritmo es flexible y puede ordenar tanto por **fecha de compra** (utilizando objetos `Date`) como por **monto de compra** (comparando números flotantes).

QUICKSORT

```
export const ordenarPorQuickSort = (array, criterio) => {
  if (array.length <= 1) {
    return array;
  }

  const pivote = array[array.length - 1]; // Tomamos el último elemento como pivote
  const izquierda = [];
  const derecha = [];

  // Comparar según el criterio de ordenación
  for (let i = 0; i < array.length - 1; i++) {
    if (criterio === "fecha") {
      if (
        convertirFecha(array[i].fechaCompra) <
        convertirFecha(pivote.fechaCompra)
      ) {
        izquierda.push(array[i]);
      } else {
        derecha.push(array[i]);
      }
    } else if (criterio === "monto") {
```

```
      if (parseFloat(array[i].monto) < parseFloat(pivote.monto)) {
        izquierda.push(array[i]);
      } else {
        derecha.push(array[i]);
      }
    }
  }

  return [
    ...ordenarPorQuickSort(izquierda, criterio),
    pivote,
    ...ordenarPorQuickSort(derecha, criterio),
  ];
};
```

Función `ordenarPorQuickSort`

Esta función implementa el Quicksort, que ordena el array según el **criterio** de ordenación (ya sea por fecha o monto). La función sigue los pasos típicos del algoritmo:

a. Caso base: array de un solo elemento o vacío

```
if (array.length <= 1) {
  return array;
}
```

- Si el array tiene uno o menos elementos, ya está ordenado, así que simplemente se retorna el array tal como está.

b. Elección del pivote

```
const pivote = array[array.length - 1]; // Tomamos el último elemento como pivote
```

- El pivote es el último elemento del array. Este elemento se utilizará como referencia para dividir el array en dos partes: elementos menores al pivote y elementos mayores al pivote.

c. Dividir el array en dos sublistas: `izquierda` y `derecha`

```
const izquierda = [];
const derecha = [];
```

- Se inicializan dos arrays vacíos: `izquierda` y `derecha`. El array `izquierda` contendrá todos los elementos menores o iguales al pivote, y el array `derecha` contendrá todos los elementos mayores que el pivote.

d. Comparar y clasificar los elementos según el criterio

```
for (let i = 0; i < array.length - 1; i++) {
  if (criterio === "fecha") {
    if (convertirFecha(array[i].fechaCompra) < convertirFecha(pivote.fechaCompra)) {
      izquierda.push(array[i]);
    } else {
      derecha.push(array[i]);
    }
  } else if (criterio === "monto") {
    if (parseFloat(array[i].monto) < parseFloat(pivote.monto)) {
      izquierda.push(array[i]);
    } else {
      derecha.push(array[i]);
    }
  }
}
```

- El array se recorre elemento por elemento (excepto el pivote), y según el criterio:
  - **Fecha**: Si la fecha de compra del elemento actual es menor que la fecha del pivote, se coloca en el array `izquierda`. Si es mayor o igual, se coloca en el array `derecha`.
  - **Monto**: Similar a la fecha, si el monto es menor que el del pivote, el elemento se coloca en el array `izquierda`; de lo contrario, en el array `derecha`.

e. Llamada recursiva para ordenar las sublistas

```
return [
    ...ordenarPorQuickSort(izquierda, criterio), // Ordenar los
elementos menores
  pivote,                                    // Colocar el pivote
en el medio
    ...ordenarPorQuickSort(derecha, criterio),   // Ordenar los
elementos mayores
];
```

- El algoritmo se aplica recursivamente:
    - Se ordena primero el array `izquierda`.
    - Se coloca el pivote en su lugar correcto.
    - Luego se ordena el array `derecha`.
- El resultado es la combinación de la sublista izquierda ordenada, el pivote y la sublista derecha ordenada.

Pruebas Prácticas del Programa: Evaluación del Rendimiento

Para evaluar el rendimiento de los algoritmos implementados en el programa, se llevaron a cabo una serie de pruebas prácticas que permitieron analizar tanto el **tiempo de ejecución** como el **uso de memoria** de los algoritmos **Heapsort**, **Quicksort** y **Búsqueda Binaria**. Estas pruebas se realizaron utilizando listas de clientes de diferentes tamaños y en diferentes condiciones de ordenación para obtener una visión integral de la eficiencia y rendimiento en escenarios diversos.

Escenarios de Pruebas

Las pruebas se realizaron con listas de clientes de **100, 500 y 1000 clientes**, utilizando tres escenarios diferentes:

1. **Escenario aleatorio**: Las listas de clientes estaban desordenadas de forma aleatoria.
2. **Escenario preordenado**: Las listas de clientes ya estaban ordenadas antes de aplicar el algoritmo.
3. **Escenario inversamente ordenado**: Las listas de clientes estaban ordenadas en el orden inverso al que se buscaba ordenar.

El criterio de ordenación utilizado fue tanto la **fecha** de las compras como el **monto** de las mismas.

Resultados del Tiempo de Ejecución

El tiempo de ejecución se midió en milisegundos (ms) y se compararon tanto **Quicksort** como **Heapsort** bajo los tres escenarios mencionados. Los resultados se dividen en dos tablas: uno para el escenario aleatorio y otro para el peor caso (inversamente ordenado).

Tabla 1: Tiempo de Ejecución en el Escenario Aleatorio

Tamaño de la Lista	Quicksort (aleatorio)	Heapsort (aleatorio)
100	1.4 ms	2.3 ms
500	6.9 ms	10.6 ms
1000	17.8 ms	11.5 ms

Tabla 2: Tiempo de Ejecución en el Peor Caso (Inversamente Ordenado)

Tamaño de la Lista	Quicksort (peor caso)	Heapsort (peor caso)
100	6.4 ms	1.3 ms
500	144.4 ms	9.6 ms
1000	473.2 ms	15.6 ms

Análisis de los Resultados del Tiempo de Ejecución

1. **Escenario Aleatorio**:
    - En el caso de listas de clientes con un orden aleatorio, **Quicksort** mostró un rendimiento más rápido que **Heapsort** en las listas pequeñas (100 y 500 clientes). Esto es coherente con la expectativa de que **Quicksort** es más eficiente en la mayoría de los casos prácticos, debido a su enfoque de partición rápida y eficaz.
    - Sin embargo, a medida que el tamaño de la lista aumenta (1000 clientes), el rendimiento de **Heapsort** comienza a igualar o incluso superar al de **Quicksort**. Esto se debe a que **Heapsort** tiene un comportamiento más

consistente en diferentes escenarios, mientras que **Quicksort** puede sufrir una ligera desaceleración en listas más grandes si no se selecciona un pivote adecuado.

2. **Escenario Inversamente Ordenado (Peor Caso)**:
    - En este escenario, se observa claramente el impacto de un mal pivote en **Quicksort**. Cuando la lista está ordenada en sentido inverso (el peor caso para Quicksort), su tiempo de ejecución se incrementa drásticamente, alcanzando **473.2 ms** para la lista de 1000 clientes.
    - En contraste, **Heapsort** mantiene un rendimiento significativamente mejor y más estable en este escenario, debido a su estructura basada en montículos que no se ve tan afectada por el orden inicial de los elementos. Para la lista de 1000 clientes, el tiempo de **Heapsort** fue solo **15.6 ms**, mucho más rápido que Quicksort en este caso.
3. **Comparación General**:
    - **Quicksort** es el algoritmo más rápido en situaciones donde los datos están desordenados de manera aleatoria o en escenarios de listas más pequeñas. Sin embargo, en situaciones donde los datos están ordenados inversamente (peor caso), **Heapsort** se desempeña mucho mejor y con mayor estabilidad.
    - **Heapsort**, por su parte, es más predecible y confiable en todos los escenarios, aunque su rendimiento puede ser ligeramente inferior en casos de listas pequeñas y desordenadas aleatoriamente.

Conclusiones

El desarrollo del programa de gestión de clientes basado en algoritmos de ordenación y búsqueda ha permitido obtener una comprensión más profunda de cómo estas técnicas pueden ser aplicadas de manera efectiva para resolver problemas prácticos en la gestión de datos empresariales. A lo largo del proyecto, se implementaron y evaluaron tres algoritmos clave —Heapsort, Quicksort y Búsqueda Binaria— con el objetivo de optimizar el acceso, organización y recuperación de los datos de clientes en un entorno de comercio.

Uno de los aprendizajes más importantes obtenidos es que no existe un algoritmo que sea ideal para todos los escenarios. Cada algoritmo tiene fortalezas y debilidades que lo hacen más o menos adecuado dependiendo de las características específicas de los datos que se estén procesando.

Rendimiento de los Algoritmos

Heapsort se destacó por su eficiencia en términos de uso de memoria, siendo un algoritmo in-place que no requiere espacio adicional significativo. Además, ofreció un rendimiento estable y predecible en todos los escenarios, manteniendo una complejidad temporal de O(n log n) en el mejor, peor y promedio de los casos. Esto lo convierte en una opción confiable para situaciones en las que se necesita un rendimiento constante y donde el uso de memoria es una preocupación.

Quicksort, por otro lado, fue más rápido en listas aleatorias, lo que lo hace ideal para situaciones donde los datos no tienen una estructura predefinida. Su enfoque basado en la técnica de división y conquista lo convierte en uno de los algoritmos más rápidos en la práctica para la mayoría de los casos. Sin embargo, su principal desventaja se observó en

listas inversamente ordenadas, donde la elección ineficiente del pivote lo llevó a un rendimiento degradado con una complejidad temporal de $O(n^2)$. Esto subraya la importancia de la selección adecuada del pivote para evitar un rendimiento subóptimo.

Impacto en la Gestión de Clientes

La implementación de estos algoritmos en el sistema de gestión de clientes demostró que la optimización de la ordenación de los datos tiene un impacto directo en la velocidad y eficiencia del acceso a la información. La posibilidad de ordenar los datos de los clientes por criterios como fecha de compra o monto de compra permitió no solo una gestión más estructurada, sino también búsquedas más rápidas y precisas utilizando la Búsqueda Binaria.

La Búsqueda Binaria demostró ser una herramienta extremadamente eficiente para la recuperación de datos en listas previamente ordenadas. Con una complejidad temporal de $O(\log n)$, permitió acceder a datos específicos de clientes en tiempos considerablemente más cortos que los métodos de búsqueda secuencial, lo que es esencial para sistemas que manejan grandes volúmenes de información.

Consideraciones sobre el Uso de Memoria

Una de las conclusiones más importantes derivadas de las pruebas es que el uso de memoria varía significativamente entre los algoritmos. Heapsort, siendo un algoritmo in-place, utilizó menos recursos, lo que lo hace ideal para sistemas donde la limitación de memoria es un factor crítico. Quicksort, aunque más rápido en muchos escenarios, requiere memoria adicional para las particiones, lo que puede ser un inconveniente en sistemas con recursos limitados.

Elección del Algoritmo Adecuado

El análisis realizado en este proyecto resalta la importancia de elegir el algoritmo adecuado para cada situación específica. En aplicaciones del mundo real, donde los datos pueden tener patrones específicos o pueden llegar en diferentes formatos (ordenados, desordenados o inversamente ordenados), la elección del algoritmo debe basarse en una evaluación cuidadosa de los requisitos de rendimiento y los recursos disponibles.

Heapsort es ideal para aplicaciones que requieren un uso eficiente de memoria y donde el rendimiento estable es una prioridad.

Quicksort es más adecuado para escenarios donde se espera que los datos lleguen en forma aleatoria y el tiempo de ejecución promedio sea más importante que el rendimiento en el peor caso.

Búsqueda Binaria es fundamental para aplicaciones que requieren realizar búsquedas rápidas en listas grandes que ya están ordenadas, mejorando drásticamente la eficiencia de las operaciones de búsqueda.

Futuras Mejoras

El sistema desarrollado ha sentado las bases para futuras mejoras. A medida que el volumen de datos crece en los sistemas de gestión de clientes, la optimización tanto en términos de tiempo de ejecución como en uso de memoria se vuelve cada vez más crítica. Se recomienda continuar explorando otras técnicas avanzadas de optimización, como el uso de algoritmos híbridos que combinen lo mejor de varios enfoques, o incluso la implementación de estructuras de datos avanzadas, como árboles balanceados o tablas hash, para mejorar aún más la eficiencia del sistema.

Conclusión Final

En conclusión, este proyecto ha demostrado la importancia de seleccionar y aplicar los algoritmos de manera adecuada según las necesidades específicas de la aplicación. La implementación de Heapsort, Quicksort y Búsqueda Binaria permitió una gestión más eficiente de los datos de clientes, proporcionando una solución flexible y escalable para un entorno empresarial. La clave del éxito en este tipo de proyectos radica en comprender tanto las características de los datos como las limitaciones del sistema, lo que permitirá seguir optimizando el rendimiento de las aplicaciones en el futuro.

Captura del programa:

link del repositorio mostrando el codigo y el video mostrando el programa:
https://github.com/carlos5845/gestion-de-clientes-/settings