

TEGNOLOGICO DE ESTUDIOS SUPERIORES JILOTEPEC

METODOS DE ORDENAMIENTO

Con valores numéricos en vectores

Alumna :

Vanessa Fabiola Hernández Ramírez

Docente:

Materia:

Matricula:

**202423330**

Salón

3101

El ordenamiento de valores numéricos es una operación fundamental en la informática y en muchas aplicaciones prácticas, desde la organización de datos hasta la optimización de algoritmos más complejos. Un vector, que es una estructura de datos lineal que almacena una colección de elementos del mismo tipo en un orden específico, es uno de los formatos más comunes para trabajar con datos numéricos. Ordenar un vector numérico implica reorganizar sus elementos según un criterio específico, como orden ascendente (de menor a mayor) o descendente (de mayor a menor).

El ordenamiento no solo facilita el análisis y visualización de datos, sino que también es crucial para tareas como la búsqueda eficiente, la eliminación de duplicados y el procesamiento estadístico. Dada su importancia, se han desarrollado numerosos métodos de ordenamiento, cada uno con sus características, ventajas y limitaciones. Estos métodos pueden clasificarse en algoritmos de ordenamiento interno, Que trabajan directamente en la memoria principal, y algoritmos de ordenamiento externo, diseñados para manejar grandes conjuntos de datos que no caben en la memoria principal.

1. **Ordenamiento por Burbuja (Bubble Sort)**

Algoritmo: Es uno de los algoritmos más simples y menos eficientes. Compara elementos adyacentes y los intercambiables si están en el orden incorrecto. Este proceso se repite hasta que no se realizan más intercambios.

1. Recorre el vector varias veces.
2. Compara cada par de elementos adyacentes.
3. Intercambia los elementos si están en el orden incorrecto.
4. Repite hasta que no haya más intercambios.

Código ejemplo (en pseudocódigo):

Para i desde 0 hasta n-1:

Para j desde 0 hasta n-i-1:

Si vector[j] > vector[j+1]:

Intercambiar(vector[j], vector[j+1])

Ventajas:

Fácil de implementar.

Útil para vectores muy pequeños o casi ordenados.

Desventajas:

Ineficiente para vectores grandes ().

Requiere múltiples pasadas, incluso si el vector ya está parcialmente ordenado.

1. **Ordenamiento por Selección (Selection Sort)**

Algoritmo: El algoritmo de selección divide el vector en dos partes: la parte ordenada y la no ordenada. En cada iteración, encuentra el elemento mínimo (o máximo, dependiendo del orden) de la parte no ordenada y lo coloca en la posición correspondiente.

1. Encuentra el elemento más pequeño en el vector y colócalo al principio.
2. Repite el proceso con el resto del vector.

Código ejemplo:

Para i desde 0 hasta n-1:

Min\_index ← i

Para j desde i+1 hasta n:

Si vector[j] < vector[min\_index]:

Min\_index ← j

Intercambiar(vector[i], vector[min\_index])

Ventajas:

No requiere memoria adicional (in-place).

Simple de implementar.

Desventajas:

Ineficiente para vectores grandes ().

Realiza comparaciones innecesarias incluso si el vector está ordenado.

1. **Ordenamiento por Inserción (Insertion Sort)**

Algoritmo: El algoritmo de inserción toma los elementos de un vector uno a uno y los inserta en su posición correcta dentro de la sublista ordenada. Funciona de manera similar a cómo ordenamos cartas en la mano.

1. Divide el vector en una parte ordenada y otra desordenada.
2. Inserta cada elemento de la parte desordenada en la posición correcta de la parte ordenada.

Código ejemplo:

Para i desde 1 hasta n-1:

Clave ← vector[i]

J ← i-1

Mientras j ≥ 0 y vector[j] > clave:

Vector[j+1] ← vector[j]

J ← j-1

Vector[j+1] ← clave

Ventajas:

Eficiente para vectores pequeños o casi ordenados ( en el mejor caso).

Fácil de implementar.

Desventajas:

Ineficiente para vectores grandes ( en el peor caso).

1. **Ordenamiento Rápido (Quick Sort)**

Algoritmo: El algoritmo de ordenamiento rápido es un algoritmo de tipo "divide y vencerás". Selecciona un elemento como pivote y divide el vector en dos sublistas: una con elementos menores que el pivote y otra con elementos mayores. Luego ordena recursivamente esas sublistas.

1. Elige un elemento como pivote.
2. Divide el vector en dos subgrupos: los menores y los mayores al pivote.
3. Aplica recursivamente el proceso a cada subgrupo.

Código ejemplo:

QuickSort(vector, inicio, fin):

Si inicio < fin:

P ← Particionar(vector, inicio, fin)

QuickSort(vector, inicio, p-1)

QuickSort(vector, p+1, fin)

Ventajas:

Muy rápido en promedio ().

In-place (en la mayoría de las implementaciones).

Desventajas:

En el peor caso, puede ser ineficiente ().

Puede requerir optimización para ser estable.

1. **Ordenamiento por Fusión (Merge Sort)**

Algoritmo: El ordenamiento por fusión también es un algoritmo "divide y vencerás". Divide el vector en dos mitades, las ordena recursivamente y luego las fusiona de manera ordenada.

1. Divide el vector en mitades recursivamente hasta obtener subvectores de un elemento.
2. Fusiona los subvectores de manera ordenada.

Código ejemplo:

MergeSort(vector):

Si tamaño(vector) > 1:

Mitad ← tamaño(vector)/2

Izquierda ← vector[0:mitad]

Derecha ← vector[mitad:]

MergeSort(izquierda)

MergeSort(derecha)

Fusionar(izquierda, derecha, vector)

Ventajas:

Eficiente () y estable.

Adecuado para grandes conjuntos de datos.

Desventajas:

Requiere memoria adicional para la fusión.

Más complejo de implementar.

1. **Ordenamiento por Montículo (Heap Sort)**

Algoritmo: El ordenamiento por montículo convierte el vector en una estructura de montículo (heap), un tipo de árbol binario que satisface la propiedad del montículo. Luego, extrae el elemento máximo (o mínimo) y lo coloca en su posición final.

1. Construye un montículo (heap) máximo a partir del vector.
2. Intercambia el primer y último elemento, reduce el tamaño del montículo y ajusta el heap.
3. Repite hasta que el vector esté ordenado.

Código ejemplo:

HeapSort(vector):

ConstruirHeap(vector)

Para i desde n-1 hasta 1:

Intercambiar(vector[0], vector[i])

AjustarHeap(vector, 0, i)

Ventajas:

Complejidad en el peor caso.

In-place (no requiere memoria adicional significativa).

Desventajas:

No es estable.

Más lento que Quick Sort en promedio.

1. **Ordenamiento por Conteo (Counting Sort)**

Algoritmo: Este algoritmo es adecuado cuando los valores de los elementos en el vector son enteros y se encuentran dentro de un rango limitado. Cuenta cuántas veces aparece cada valor en el vector y luego reconstruye el vector ordenado utilizando esta información.

1. Cuenta las ocurrencias de cada valor en un rango específico.
2. Usa esta información para colocar los elementos en el orden correcto.

Código ejemplo:

CountingSort(vector, rango):

Conteo ← array(rango)

Para cada elemento en vector:

Conteo[elemento]++

Posicionar elementos en el vector usando conteo

Ventajas:

Muy rápido para valores en rangos pequeños (, donde es el rango).

Estable y no comparativo.

Desventajas:

Ineficiente para rangos grandes.

Requiere memoria adicional proporcional al rango.

CONCLUSION

Estos métodos no solo optimizan el rendimiento de otros algoritmos, como los de búsqueda y análisis, sino que también facilitan el manejo de grandes volúmenes de información, mejoran la experiencia del usuario y contribuyen a la eficiencia de los sistemas. Algunos algoritmos, como **Quick Sort** y \*\*Merge S**Merge Sort** , son especialmente útiles en aplicaciones de procesamiento de grandes cantidades de datos, mientras que otras, como **Insertion Sort** y \*\*Bubble So.**Bubble Sort** , son más adecuados para vectores pequeños o casi ordenados

En conclusión, los **métodos de ordenamiento con vectores** son **importancia de estos métodos** radica en su capacidad para optimizar la **búsqueda** , la \*\*clasificación y el **análisis** de información, impactando de manera significativa en la velocidad y eficiencia de las aplicaciones informáticas. La **elección del algoritmo adecuado** depende de diversos factores, como el tamaño del conjunto de datos. **Bubble Sort** o **Insertion Sort** , que son simples y fáciles de implementar, pero poco eficientes para grandes volúmenes de datos debido a su como. **Ordenación rápida** y \*\*Combinar **Merge Sort** son mucho más eficientes en el Además, los algoritmos de ordenamiento no solo mejoran el rendimiento de los sistemas en términos de tiempo, sino que también optimizan el **uso de recursos .**

El ordenamiento también es crucial en **la toma de decisiones** y **Análisis de datos.** Muchas **búsqueda binaria** o en algoritmos de \*\*análisis esta **análisis estadístico** , donde se necesita identificación **y recomendación** y **minería de datos** , el ordenamiento también juega un p

Otro punto importante es que, aunque algunos algoritmos de ordenamiento, como el **Merge Sort**o **Quick Sort** , son eficientes en términos de tiempo, requieren más recursos computacionales, especiales **Combinar Ordenar** . En aplicaciones donde el uso de memoria es **Ordenar por selección** o **Ordenar por inserción** podrían ser más

Por otro lado**, algoritmos no comparativos** como **Counting Sort** y **\*Radix Sort** tiene ventajas en situaciones específicas donde los datos cum

En el contexto de **programación competitiva** , el conocimiento de los métodos de orden. Finalmente, el uso adecuado de los métodos de ordenamiento impacta directamente en la **experiencia del usuario .** En aplicaciones interactivas, como bases de datos, motores de búsqueda, o interfaces gráficas, los algoritmos de ordenamiento efectivo.

En resumen, los algoritmos de ordenamiento en vectores son fundamentales en **todas las áreas de la informática .** Su capacidad para organizar datos de manera eficiente mejora el rendimiento de los sistemas, optimiza el uso de recursos, facilita la toma de decisiones y as