Metodos de ordenamiento

Los métodos de ordenamiento son algoritmos que realizan la operación de arreglar los registros de una tabla en algún orden secuencial de acuerdo a un criterio de ordenamiento. El ordenamiento se efectúa con base en el valor de algún campo en un grupo de datos. El ordenamiento puede estar dado de forma iterativa o recursiva según la naturaleza y forma de ejecución del mismo.

Insertion sort

Es un algoritmo de clasificación que coloca un elemento sin clasificar en su lugar adecuado en cada iteración. Es util para ordenar elementos que van entrando poco a poco.

Ejemplo:

```cpp
// C++ program for insertion sort

#include <bits/stdc++.h>
using namespace std;

// Function to sort an array using
// insertion sort
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key,
        // to one position ahead of their
        // current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// A utility function to print an array
// of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
```

```
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, N);
    printArray(arr, N);

    return 0;
}
// This is code is contributed by rathbhupendra
```

Complejidad Temporal: La complejidad temporal del algoritmo de ordenamiento por inserción es de O(n^2) en el peor de los casos. Esto significa que el tiempo de ejecución del algoritmo aumenta cuadráticamente con el tamaño de la entrada. En el mejor caso, cuando la lista ya está ordenada, la complejidad temporal es O(n), haciendo que el algoritmo sea eficiente en este escenario.

Complejidad Espacial: En cuanto a la complejidad espacial, el algoritmo de ordenamiento por inserción tiene una complejidad O(1), es decir, utiliza una cantidad constante de espacio adicional independientemente del tamaño de la entrada. Esto se debe a que el algoritmo realiza las operaciones de ordenamiento directamente sobre la lista de entrada, sin requerir estructuras de datos adicionales de gran tamaño.

Ventajas

1. es simple
2. requerimiento minimo de memoria
3. Eficiente en listas pequeñas

Desventajas

1. Es deficiente en listas grandes
2. puede ser lento
3. Realiza numerosas comparaciones

Shellsort

El método ShellSort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está “casi ordenada”.
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez. El algoritmo ShellSort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga “pasos más grandes” hacia su posición esperada. Se utiliza cuando el número de elementos a ordenar es grande.

Ejemplo:

```
#include<iostream>
#include<conio.h>
```

```
using namespace std;
int Arreglo[100];
void LeerArreglo(int Numero);
void EscribeArreglo(int Numero);
void Shell(int Numero);

int main(){
    int Num;
    cout<<"Ingrese dimension del arreglo : ";
    cin>>Num;
    LeerArreglo(Num);
    Shell(Num);
    cout<<endl;
    EscribeArreglo(Num);
    return 0;
}

void LeerArreglo(int Numero){
    int i;
    for(i=1;i<=Numero;i++)
    {
        cout<<"Arreglo["<<i<<"]=";
        cin>>Arreglo[i];
    }
}

void EscribeArreglo(int Numero){
    int i;
    cout<<"elementos ordenados por metodo Shell sort"<<endl;
    for(i=1;i<=Numero;i++)
    {

        cout<<"\t"<<Arreglo[i];
    }
}

void Shell(int Numero){
    int i,j,k,incremento,aux;

    incremento=Numero/2;

    while(incremento>0){
        for(i=incremento+1;i<=Numero;i++){
            j=i-incremento;
            while(j>0){
                if(Arreglo[j]>=Arreglo[j+incremento]){
                    aux = Arreglo[j];
                    Arreglo[j] = Arreglo[j+incremento];
                    Arreglo[j+incremento] = aux;
                }
                else{
                        j=0;
                    }
                j=j-incremento;
```

```
            }
        }
        incremento=incremento/2;
    }
}
```

Complejidad Temporal: La complejidad temporal del algoritmo de ordenamiento Shellsort es más complicada de analizar de manera precisa, ya que depende de la secuencia de brechas utilizada. Sin embargo, en el caso promedio, la complejidad temporal se estima en O(n log^2 n) o incluso en O(n^(3/2)), lo que lo hace más eficiente que algoritmos cuadráticos como el Insertion Sort.

Complejidad Espacial: En términos de complejidad espacial, Shellsort también tiene una complejidad de O(1), lo que significa que utiliza una cantidad constante de espacio adicional independientemente del tamaño de la entrada. A diferencia de algunos algoritmos que requieren estructuras de datos adicionales, Shellsort opera directamente sobre la lista de entrada.

Ventajas: -No requiere memoria adicional. -Mejor rendimiento que el método de inserción rápido. -Fácil implantación.

Desventajas: -Su funcionamiento puede resultar confuso -Suele ser un poco lento. -Realiza numerosas comparaciones e intercambios.

Heap sort

Es una técnica de clasificación basada en comparaciones basada en la estructura de datos Binary Heap. Es similar a la ordenación por selección donde primero encontramos el elemento mínimo y colocamos el elemento mínimo al principio. Repita el mismo proceso para los elementos restantes.

Ejemplo:

```
// C++ program for implementation of Heap Sort

#include <iostream>
using namespace std;

// To heapify a subtree rooted with node i
// which is an index in arr[].
// n is size of heap
void heapify(int arr[], int N, int i)
{

    // Initialize largest as root
    int largest = i;

    // left = 2*i + 1
    int l = 2 * i + 1;

    // right = 2*i + 2
    int r = 2 * i + 2;
```

```
    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest
    // so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int N)
{

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element
    // from heap
    for (int i = N - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// A utility function to print array of size n
void printArray(int arr[], int N)
{
    for (int i = 0; i < N; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver's code
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
```

```
    heapSort(arr, N);

    cout << "Sorted array is \n";
    printArray(arr, N);
}
```

Complejidad Temporal: La complejidad temporal de Heap Sort es O(n log n) en todos los casos, ya sea en el peor caso, el mejor caso o el caso promedio. Heap Sort utiliza un montículo (heap) como estructura de datos subyacente para organizar los elementos, y la construcción del montículo inicial toma O(n) tiempo, mientras que cada operación de extracción máxima (eliminar el elemento máximo del montículo) toma O(log n) tiempo. Dado que se realizan n operaciones de extracción máxima, la complejidad total es O(n log n).

Complejidad Espacial: En cuanto a la complejidad espacial, Heap Sort tiene una complejidad O(1) adicional. Esto significa que utiliza una cantidad constante de espacio adicional independientemente del tamaño de la entrada. A diferencia de algunos algoritmos que pueden requerir estructuras de datos adicionales proporcionando una complejidad espacial de O(n), Heap Sort opera directamente sobre la lista de entrada y no necesita espacio adicional proporcional al tamaño de la entrada.

Ventajas:

1. Su desempeño es en promedio tan buen como el Quicksort
2. No utiliza memoria adicional

Desventajas:

1. No es estable
2. Es un metodo mas complejo que los demas

MergeSort

es un algoritmo de ordenación fácilmente paralelizable. Este algoritmo consiste en dividir el vector a ordenar en varias partes, estas partes se ordenan y, posteriormente, se mezclan entre ellas de forma ordenada. Es útil cuando se tiene una estructura ordenada y los nuevos datos a añadir se almacenan en una estructura temporal para después agregarlos a la estructura original de manera que vuelva a quedar ordenada.

Ejemplo:

```
#include <iostream>
#include <vector>

// Función para combinar dos subarreglos de arr[]
// El primer subarreglo es arr[l..m]
// El segundo subarreglo es arr[m+1..r]
void merge(std::vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;  // Tamaño del primer subarreglo
    int n2 = r - m;      // Tamaño del segundo subarreglo

    // Crear arreglos temporales L[] y R[]
    std::vector<int> L(n1), R(n2);
```

```
    // Copiar datos a los arreglos temporales L[] y R[]
    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];

    // Combinar los arreglos temporales de vuelta en arr[l..r]
    int i = 0;  // Índice inicial del primer subarreglo
    int j = 0;  // Índice inicial del segundo subarreglo
    int k = l;  // Índice inicial del subarreglo combinado
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copiar los elementos restantes de L[], si los hay
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copiar los elementos restantes de R[], si los hay
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Función principal que ordena arr[l..r] usando Merge Sort
void mergeSort(std::vector<int>& arr, int l, int r) {
    if (l < r) {
        // Encuentra el punto medio del arreglo
        int m = l + (r - l) / 2;

        // Ordena la primera y segunda mitad recursivamente
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Combina las mitades ordenadas
        merge(arr, l, m, r);
    }
}

int main() {
    std::vector<int> arr = {12, 11, 13, 5, 6, 7};
    int arr_size = arr.size();
```

```
    std::cout << "Arreglo original: ";
    for (int i = 0; i < arr_size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;

    // Aplica el Merge Sort
    mergeSort(arr, 0, arr_size - 1);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < arr_size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;

    return 0;
}
```

Complejidad temporal: La función mergeSort tiene una complejidad temporal de O(n log n), donde "n" es el número de elementos en el arreglo. La función merge también tiene una complejidad temporal de O(n), ya que combina dos subarreglos ordenados. La recursión en mergeSort realiza divisiones del arreglo hasta que cada subarreglo tenga un solo elemento, lo que lleva a log(n) divisiones. En cada nivel de la recursión, hay O(n) operaciones debido a la combinación de subarreglos en la función merge. Por lo tanto, el tiempo total es O(n log n).

Complejidad espacial: La complejidad espacial del algoritmo Merge Sort es O(n), donde "n" es el número de elementos en el arreglo. Esto se debe al uso de arreglos temporales L[] y R[] para almacenar los subarreglos durante el proceso de combinación en la función merge. Además, la recursión en mergeSort utiliza la pila de llamadas, ocupando espacio proporcional al log(n) en el peor caso. En resumen, la complejidad espacial es O(n) + O(log n), que se simplifica a O(n).

Ventajas:

1. Metodo estable mientras la operacion de mezclas este bien implementada
2. Es efectivo para conjuntos de datos que se puedan acceder como arreglos, vectores y listas ligadas

Desventajas:

1. Esta definido recursivamente y su implementacion no recursiva emplea una pila, por lo que requiere un espacio adicional de memoria

Quicksort

Es actualmente el mas eficiente y veloz de los método de ordenación interna. Es tambien conocido con el nombre del método rápido y de ordenamento por partición. Este método es una mejora sustancial del método de intercambio directo y recibe el nombre de Quick Sort, por la velocidad con la que ordena los elementos del arreglo. Quicksort es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n.

Ejemplo:

```cpp
#include <iostream>
#include <vector>

// Función para dividir el arreglo y devolver el índice del elemento pivote
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];  // Selecciona el último elemento como pivote
    int i = (low - 1);      // Índice del menor elemento

    // Recorre el subarreglo
    for (int j = low; j <= high - 1; j++) {
        // Si el elemento actual es menor o igual al pivote
        if (arr[j] <= pivot) {
            i++;

            // Intercambia arr[i] y arr[j]
            std::swap(arr[i], arr[j]);
        }
    }

    // Intercambia arr[i + 1] y arr[high] para colocar el pivote en su posición
correcta
    std::swap(arr[i + 1], arr[high]);

    return (i + 1);  // Devuelve el índice del pivote
}

// Función principal que implementa el algoritmo Quicksort
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        // Encuentra el índice del pivote, arr[pi] está en la posición correcta
        int pi = partition(arr, low, high);

        // Ordena los elementos antes y después del pivote
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    std::vector<int> arr = {10, 7, 8, 9, 1, 5};
    int arr_size = arr.size();

    std::cout << "Arreglo original: ";
    for (int i = 0; i < arr_size; i++)
        std::cout << arr[i] << " ";
    std::cout << std::endl;

    // Aplica el Quicksort
    quickSort(arr, 0, arr_size - 1);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < arr_size; i++)
        std::cout << arr[i] << " ";
```

```
    std::cout << std::endl;

    return 0;
}
```

Complejidad temporal: La función partition tiene una complejidad temporal de O(n), donde "n" es el número de elementos en el subarreglo. La función quickSort tiene una complejidad temporal promedio de O(n log n). En el peor caso, la elección del pivote puede llevar a una complejidad temporal de O(n^2), pero esto es poco probable en situaciones prácticas si se selecciona un pivote de manera adecuada. En el mejor caso y en la media, quickSort realiza log(n) particiones, y en cada partición, realiza O(n) comparaciones y swaps. Por lo tanto, la complejidad temporal promedio es O(n log n).

Complejidad espacial: La complejidad espacial del algoritmo Quicksort es O(log n) en la pila de llamadas para la versión recursiva. Esto se debe a la recursión, ya que cada llamada recursiva agrega un marco de pila. En el peor caso, la complejidad espacial puede ser O(n) debido a la recursión en el peor caso, pero en el caso promedio es O(log n).

Ventajas:

1. Requiere de pocos recursos en comparación a otros métodos de ordenamiento.
2. En la mayoría de los casos, se requiere aproximadamente N log N operaciones.
3. Ciclo interno es extremadamente corto.
4. No se requiere de espacio adicional durante ejecución.

Desventajas:

1. Se complica la implementación si la recursión no es posible.
2. Peor caso, se requiere N2
3. Un simple error en la implementación puede pasar sin detección, lo que provocaría un rendimiento pésimo.
4. No es útil para aplicaciones de entrada dinámica, donde se requiere reordenar una lista de elementos con nuevos valores.
5. Se pierde el orden relativo de elementos idénticos.

Referencias

GitHub - gbaudino/MetodosDeOrdenamiento: Métodos de ordenamiento con sus especificaciones, características y su código en Python 3. Comparativa final entre los algoritmos de ordenamiento. (s.f.). GitHub. https://github.com/gbaudino/MetodosDeOrdenamiento

Insertion Sort (With Code in Python/C++/Java/C). (s.f.). Programiz: Learn to Code for Free. https://www.programiz.com/dsa/insertion-sort

Insertion sort. (s.f.). Share & Discover Presentations | SlideShare. https://www.slideshare.net/M1k3Z/insertion-sort-55547752

(s.f.). Universidad Don Bosco © | Inicio. https://www.udb.edu.sv/udb_files/recursos_guias/informatica-ingenieria/programacion-iv/2019/ii/guia-4.pdf

Heap Sort - Data Structures and Algorithms Tutorials - GeeksforGeeks. (s.f.). GeeksforGeeks. https://www.geeksforgeeks.org/heap-sort/

Algoritmo de ordenamiento: Heap Sort. (s.f.). Share & Discover Presentations | SlideShare. https://www.slideshare.net/DanielGmez3/algoritmo-de-ordenamiento-heap-sort

MÃ©todo Quick Sort. (s.f.). CIDECAME UAEH. http://cidecame.uaeh.edu.mx/lcc/mapa/proyecto/libro9/mtodo_quick_sort.html