UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACION

Estructura de Datos

NOMBRE: Miguel Alejandro Cabrera Diaz

FECHA: 05/06/2023

NRC: 9898

Algoritmos de ordenación interna

¿Que son?

Los algoritmos de ordenación interna son algoritmos establecidos para ordenar datos de forma requerida como pueden ser numérica o alfabética, estos dependen de un criterio para ser implementados de tal manera que cada uno es mejor para ordenar los datos que otro algoritmo dependiendo del tipo de dato o información a ordenar.

A continuación, expondremos los principales algoritmos de ordenamiento:

1. **Ordenamiento de Intercambio (Burbuja):**

También conocido como ordenamiento burbuja se basa en "comparar elementos en pares hasta que los elementos más grandes "burbujean" hasta el final de la lista y los más pequeños permanecen al principio."(*Tipos de Algoritmos de Ordenación en Python - Másteres Online Nº 1 Empleabilidad*, 2021)

El funcionamiento de este algoritmo es bastante sencillo ya que toma los dos primeros elementos de la lista y los comparar, si el segundo elemento en menor al primer elemento se intercambia de posición y continúa con los dos elementos siguientes, repite el bucle hasta saber que la lista está completamente ordenada.

Este es el algoritmo más fácil de implementar pero también de los mas lentos e ineficientes debido a su gran consumo de tiempo y recursos como la memoria RAM.(*TPM | Tutorial de Programación Multiplataforma*, s/f)

Ahora veremos sus diferentes implementaciones en diferentes lenguajes de programación:

- Python:

```python
#algoritmo de burbuja

def burbuja(elementos):
    Intercambio = True                  #Dejamos el boolean en true para acceder almenos una vez dentro del bucle
    while Intercambio:
        Intercambio = False             #Cambiamos a false para que de hacer un intercambio cambie el valor y de no ser asi termine el bucle.
        for i in range(len(elementos) - 1):
            if elementos[i] >elementos[i+1]:
                elementos[i], elementos[i + 1] = elementos[i + 1], elementos[i] #intercambia posicion de las variables una con otra.
                Intercambio = True

#comprobamos que funcione
Lista = [5,4,9,3,2,7,11,6]
burbuja(Lista)
print(Lista)
```

- Java:

```java
public class Burbuja {
    public static void burbuja(int[] elementos) {
        int n = elementos.length;
        // Iterar a través del arreglo n - 1 veces
        for (int i = 0; i < n - 1; i++) {
            // Iterar a través de los elementos no ordenados
            for (int j = 0; j < n - i - 1; j++) {
                // Comparar elementos adyacentes y realizar un intercambio si es necesario
                if (elementos[j] > elementos[j + 1]) {
                    int temp = elementos[j];
                    elementos[j] = elementos[j + 1];
                    elementos[j + 1] = temp;
                }
            }
        }
    }

    // Ejemplo de uso
    Run | Debug
    public static void main(String[] args) {
        int[] numeros = {64, 34, 25, 12, 22, 11, 90};
        burbuja(numeros);                          // Ordenar el arreglo utilizando el método burbuja
        System.out.println(x:"Arreglo ordenado:"); // Imprimir el arreglo ordenado
        for (int i = 0; i < numeros.length; i++) {
            System.out.print(numeros[i] + " ");
        }
    }
}
```

- C++:

```cpp
void bubbleSort(int elementos[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (elementos[j] > elementos[j + 1]) {
                int temp = elementos[j];
                elementos[j] = elementos[j + 1];
                elementos[j + 1] = temp;
            }
        }
    }
}

// Ejemplo de uso
int main() {
    int numeros[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(numeros) / sizeof(numeros[0]);
    bubbleSort(numeros, n);
    cout << "Arreglo ordenado:" << endl;
    for (int i = 0; i < n; i++) {
        cout << numeros[i] << " ";
    }
    return 0;
}
```

2. **Ordenamiento Quicksort:**

El ordenamiento rápido (Quicksort) está basado en el termino divide y vencerás donde siguiendo el ejemplo divide el trabajo para poder ordenar tardando un tiempo igual al número de elementos por el logaritmo natural del número de elementos.

Para ordenar la lista selecciona un elemento pivote y organiza la lista de forma que los elementos menores al pivote estén antes que el y los mayores estén después, luego aplica la recursividad y se llama así mismo para repetir el proceso en las sub-listas.

- Python:

```python
# Algoritmo de Quicksort:

def quicksort(elementos):
    if len(elementos) <= 1:  # Si la lista tiene 1 elemento o menos, está ordenada
        return elementos
    else:
        pivot = elementos[0]  # Selecciona el primer elemento como pivote
        less = [x for x in elementos[1:] if x <= pivot]  # Crea una lista con los elementos menores o iguales al pivote
        greater = [x for x in elementos[1:] if x > pivot]  # Crea una lista con los elementos mayores al pivote
        return quicksort(less) + [pivot] + quicksort(greater)  # Aplica Quicksort recursivamente en las sublistas

# Ejemplo de uso:
numeros = [5, 2, 9, 1, 7, 6, 3]
numeros2 = quicksort(numeros)  # Ordena la lista utilizando Quicksort
print(numeros2)  # Imprime la lista ordenada
```

- Java:

```java
public class QuickSort {
    public static void quicksort(int[] arr, int low, int high) {
        if (low < high) {
            int pivot = partition(arr, low, high); // Encuentra el índice del pivote
            quicksort(arr, low, pivot - 1); // Ordena recursivamente los elementos antes del pivote
            quicksort(arr, pivot + 1, high); // Ordena recursivamente los elementos después del pivote
        }
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // Elige el último elemento como pivote
        int i = low - 1; // Índice del elemento más pequeño

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) { // Si el elemento actual es menor o igual al pivote
                i++; // Incrementa el índice del elemento más pequeño
                swap(arr, i, j); // Intercambia arr[i] y arr[j]
            }
        }

        swap(arr, i + 1, high); // Intercambia el pivote con el elemento en la posición correcta
        return i + 1; // Retorna la posición del pivote
    }

    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    // Ejemplo de uso:
    Run | Debug
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 7, 6, 3};
        quicksort(arr, low:0, arr.length - 1); // Llama a la función quicksort para ordenar el arreglo

        for (int num : arr) {
            System.out.print(num + " "); // Imprime el arreglo ordenado
        }
    }
}
```

- C++:

```cpp
#include <iostream>
using namespace std;

void intercambiar(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int particion(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            intercambiar(&arr[i], &arr[j]);
        }
    }
    intercambiar(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = particion(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void imprimirArreglo(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

```cpp
int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Arreglo original: ";
    imprimirArreglo(arr, n);

    quickSort(arr, 0, n - 1);

    cout << "Arreglo ordenado: ";
    imprimirArreglo(arr, n);
    return 0;
}
```

3. Ordenamiento Shellsort:

El algoritmo ShellSort es una mejora del método de inserción directa. Divide la lista en subgrupos y ordena cada uno de ellos mediante inserción directa. Luego, reduce gradualmente el tamaño de los subgrupos y continúa con el proceso de ordenación hasta que el tamaño del subgrupo sea 1.

- Python:

```python
def shell_sort(arr):
    n = len(arr)
    gap = n // 2  # Establece el tamaño inicial del gap

    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i

            # Desplaza los elementos que son mayores que temp hacia la derecha
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap

            arr[j] = temp

        gap //= 2  # Reduce el tamaño del gap

# Ejemplo de uso:
arr = [5, 2, 9, 1, 7, 6, 3]
shell_sort(arr)
print(arr)
```

- Java:

```java
public class ShellSort {
    public static void shellSort(int[] arr) {
        int n = arr.length;
        int gap = n / 2; // Establece el tamaño inicial del gap

        while (gap > 0) {
            for (int i = gap; i < n; i++) {
                int temp = arr[i];
                int j = i;

                // Desplaza los elementos que son mayores que temp hacia la derecha
                while (j >= gap && arr[j - gap] > temp) {
                    arr[j] = arr[j - gap];
                    j -= gap;
                }

                arr[j] = temp;
            }

            gap /= 2; // Reduce el tamaño del gap
        }
    }

    // Ejemplo de uso:
    Run | Debug
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 7, 6, 3};
        shellSort(arr);

        for (int num : arr) {
            System.out.print(num + " "); // Imprime el arreglo ordenado
        }
    }
}
```

- C++:

```cpp
#include <iostream>
using namespace std;

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i += 1) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    shellSort(arr, n);

    cout << "Arreglo ordenado: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

4. Ordenamiento por distribución:

La ordenación por distribución se basa en la distribución de los elementos en baldes y su posterior recolección. Los elementos se distribuyen en baldes según alguna propiedad, como el valor numérico. Luego, los baldes se ordenan individualmente y se recolectan en orden para obtener la lista ordenada.(*métodos DE Ordenamiento interno - METODOS DE ORDENAMIENTO En esta Unidad explicaremos 4 algoritmos - Studocu*, s/f)

- C++:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void bucketSort(float elementos[], int n) {
    vector<float> buckets[n];

    for (int i = 0; i < n; i++) {
        int bucketIndex = elementos[i] * n;
        buckets[bucketIndex].push_back(elementos[i]);
    }

    for (int i = 0; i < n; i++) {
        sort(buckets[i].begin(), buckets[i].end());
    }

    int index = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < buckets[i].size(); j++) {
            elementos[index++] = buckets[i][j];
        }
    }
}

void printArray(float elementos[], int n) {
    for (int i = 0; i < n; i++) {
        cout << elementos[i] << " ";
    }
    cout << endl;
}

int main() {
    float numeros[] = {0.78, 0.41, 0.51, 0.14, 0.94, 0.28, 0.62};
    int n = sizeof(numeros) / sizeof(numeros[0]);

    cout << "Arreglo original:" << endl;
    printArray(numeros, n);

    bucketSort(numeros, n);

    cout << "Arreglo ordenado:" << endl;
    printArray(numeros, n);

    return 0;
}
```

5. Ordenamiento por Ratix:

El ordenamiento Radix es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, radix no está limitado sólo a los enteros.(*TPM | Tutorial de programación Multiplataforma*, s/f)

- Python:

```python
def countingSort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    # Cuenta la frecuencia de ocurrencia de cada dígito en la posición "exp"
    for i in range(n):
        index = arr[i] // exp
        count[index % 10] += 1

    # Calcula las posiciones finales de cada dígito
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Construye el arreglo de salida en orden
    i = n - 1
    while i >= 0:
        index = arr[i] // exp
        output[count[index % 10] - 1] = arr[i]
        count[index % 10] -= 1
        i -= 1

    # Copia el arreglo de salida al arreglo original
    for i in range(n):
        arr[i] = output[i]


def radixSort(arr):
    max_value = max(arr)  # Encuentra el valor máximo en el arreglo

    # Realiza el ordenamiento para cada dígito, comenzando desde el dígito menos significativo
    exp = 1
    while max_value // exp > 0:
        countingSort(arr, exp)
        exp *= 10


# Ejemplo de uso:
arr = [170, 45, 75, 90, 802, 24, 2, 66]
radixSort(arr)
print(arr)
```

- Java:

```java
import java.util.Arrays;

public class RadixSort {
    public static void radixSort(int[] arr) {
        int max = getMax(arr); // Obtiene el valor máximo en el arreglo

        // Realiza el ordenamiento para cada dígito, comenzando desde el dígito menos significativo
        for (int exp = 1; max / exp > 0; exp *= 10) {
            countingSort(arr, exp);
        }
    }

    public static void countingSort(int[] arr, int exp) {
        int n = arr.length;
        int[] output = new int[n];
        int[] count = new int[10];

        // Cuenta la frecuencia de ocurrencia de cada dígito en la posición "exp"
        for (int i = 0; i < n; i++) {
            int index = arr[i] / exp % 10;
            count[index]++;
        }

        // Calcula las posiciones finales de cada dígito
        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }

        // Construye el arreglo de salida en orden
        for (int i = n - 1; i >= 0; i--) {
            int index = arr[i] / exp % 10;
            output[count[index] - 1] = arr[i];
            count[index]--;
        }

        // Copia el arreglo de salida al arreglo original
        System.arraycopy(output, srcPos:0, arr, destPos:0, n);
    }

    public static int getMax(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }

    // Ejemplo de uso:
    Run | Debug
    public static void main(String[] args) {
        int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
        radixSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

- C++:

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

void countingSort(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};

    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

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

void radixSort(int arr[], int n) {
    int max_num = *max_element(arr, arr + n);

    for (int exp = 1; max_num / exp > 0; exp *= 10) {
        countingSort(arr, n, exp);
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

```cpp
int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Arreglo original:" << endl;
    printArray(arr, n);

    radixSort(arr, n);

    cout << "Arreglo ordenado:" << endl;
    printArray(arr, n);

    return 0;
}
```

Bibliografía:

Métodos DE Ordenamiento interno—METODOS DE ORDENAMIENTO En esta Unidad explicaremos 4 algoritmos—Studocu. (s/f). Recuperado el 5 de junio de 2023, de https://www.studocu.com/es-mx/document/universidad-politecnica-del-estado-de-morelos/base-de-datos/metodos-de-ordenamiento-interno/41957892

Tipos de Algoritmos de Ordenación en Python—Másteres Online Nº 1 Empleabilidad. (2021, junio 29). https://eiposgrados.com/blog-python/tipos-de-algoritmos-de-ordenacion-en-python/

TPM | Tutorial de Programacion Multiplataforma. (s/f). Recuperado el 5 de junio de 2023, de https://www.itslr.edu.mx/archivos2013/TPM/temas/s3u5.html