ResumenEDA2

October 23, 2025

1 Resumen de Algoritmos de Ordenamiento - Parte 2

1.1 Bucket Sort

1.1.1 ¿Qué hace?

Bucket Sort (ordenamiento por cubetas) divide los elementos en varios grupos o "cubetas", ordena cada cubeta individualmente y luego combina todas las cubetas. Es como cuando clasificas monedas por su denominación: separas en diferentes montones (cubetas) y luego ordenas cada montón.

1.1.2 Cómo funciona

- Divide el rango de valores en intervalos iguales (cubetas)
- Distribuye los elementos en las cubetas correspondientes
- Ordena cada cubeta individualmente (usando otro algoritmo)
- Combina todas las cubetas en orden

1.1.3 Código con ArrayList

```
import java.util.ArrayList;
import java.util.Collections;

public static void bucketSort(ArrayList<Double> lista) {
    // Caso base: si la lista está vacía o tiene un elemento, ya está ordenada
    if (lista.size() <= 1) return;

    // Encontramos el valor máximo y mínimo para determinar el rango
    double min = Collections.min(lista);
    double max = Collections.max(lista);

    // Calculamos el número de cubetas (usamos la raíz cuadrada del tamaño como regla general)
    int numCubetas = (int) Math.sqrt(lista.size());

    // Creamos las cubetas (cada cubeta es un ArrayList)
    ArrayList<ArrayList<Double>> cubetas = new ArrayList<>();
    for (int i = 0; i < numCubetas; i++) {
        cubetas.add(new ArrayList<>());
    }

    // FASE 1: DISTRIBUIR - colocamos cada elemento en su cubeta correspondiente
```

```
    for (int i = 0; i < lista.size(); i++) {
        double valor = lista.get(i);

        // Calculamos en qué cubeta debe ir este valor
        // Fórmula: (valor - min) / (max - min) * (numCubetas - 1)
        int indiceCubeta = (int) ((valor - min) / (max - min) * (numCubetas - 1));

        // Aseguramos que el índice esté en el rango válido [0, numCubetas-1]
        indiceCubeta = Math.min(indiceCubeta, numCubetas - 1);

        // Agregamos el valor a la cubeta correspondiente
        cubetas.get(indiceCubeta).add(valor);
    }

    // FASE 2: ORDENAR - ordenamos cada cubeta individualmente
    for (int i = 0; i < numCubetas; i++) {
        // Usamos Collections.sort() que implementa TimSort (híbrido de Merge Sort e Insertion
        Collections.sort(cubetas.get(i));
    }

    // FASE 3: COMBINAR - juntamos todas las cubetas en orden
    int indice = 0; // Índice para reconstruir la lista original

    // Recorremos cada cubeta en orden
    for (int i = 0; i < numCubetas; i++) {
        ArrayList<Double> cubetaActual = cubetas.get(i);

        // Copiamos todos los elementos de esta cubeta a la lista original
        for (int j = 0; j < cubetaActual.size(); j++) {
            lista.set(indice, cubetaActual.get(j));
            indice++; // Avanzamos al siguiente espacio en la lista
        }
    }
    // Al terminar, la lista original está completamente ordenada
}
```

1.1.4 Código con Array Normal

```
import java.util.ArrayList;
import java.util.Collections;

public static void bucketSort(double[] arr) {
    if (arr.length <= 1) return;

    // Encontrar min y max
    double min = arr[0];
    double max = arr[0];
    for (int i = 1; i < arr.length; i++) {
```

```
        if (arr[i] < min) min = arr[i];
        if (arr[i] > max) max = arr[i];
    }

    int numCubetas = (int) Math.sqrt(arr.length);
    ArrayList<ArrayList<Double>> cubetas = new ArrayList<>();
    for (int i = 0; i < numCubetas; i++) {
        cubetas.add(new ArrayList<>());
    }

    // Distribuir elementos en cubetas
    for (int i = 0; i < arr.length; i++) {
        int indiceCubeta = (int) ((arr[i] - min) / (max - min) * (numCubetas - 1));
        indiceCubeta = Math.min(indiceCubeta, numCubetas - 1);
        cubetas.get(indiceCubeta).add(arr[i]);
    }

    // Ordenar cada cubeta
    for (int i = 0; i < numCubetas; i++) {
        Collections.sort(cubetas.get(i));
    }

    // Combinar cubetas
    int indice = 0;
    for (int i = 0; i < numCubetas; i++) {
        for (double valor : cubetas.get(i)) {
            arr[indice] = valor;
            indice++;
        }
    }
}
```

1.1.5 Demostración Paso a Paso

```
ArrayList original: [0.42, 0.32, 0.33, 0.52, 0.37, 0.47, 0.51]
Rango: 0.32 - 0.52, Número de cubetas: 3 (√7  2.6 → 3)

CUBETAS:
Cubeta 0 [0.32-0.40]: [0.32, 0.33, 0.37]
Cubeta 1 [0.40-0.48]: [0.42, 0.47]
Cubeta 2 [0.48-0.52]: [0.52, 0.51]

ORDENAR CADA CUBETA:
Cubeta 0: [0.32, 0.33, 0.37]
Cubeta 1: [0.42, 0.47]
Cubeta 2: [0.51, 0.52]

COMBINAR:
```

```
[0.32, 0.33, 0.37, 0.42, 0.47, 0.51, 0.52]
```

1.2 Radix Sort

1.2.1 ¿Qué hace?

Radix Sort (ordenamiento por raíz) ordena los números procesando **dígito por dígito**, comenzando por el dígito menos significativo hasta el más significativo. Es como cuando ordenas papeles por fechas: primero por día, luego por mes, finalmente por año.

1.2.2 Cómo funciona

- Ordena por el dígito menos significativo (unidades)
- Luego por el siguiente dígito (decenas)
- Continúa hasta el dígito más significativo
- Usa un algoritmo estable (como Counting Sort) para cada dígito

1.2.3 Código con ArrayList

```java
import java.util.ArrayList;
import java.util.Collections;

public static void radixSort(ArrayList<Integer> lista) {
    // Caso base: si la lista está vacía o tiene un elemento, ya está ordenada
    if (lista.size() <= 1) return;

    // Encontramos el número máximo para saber cuántos dígitos tiene
    int max = Collections.max(lista);

    // Realizamos Counting Sort para cada dígito, empezando por el menos significativo
    // Exp representa la posición del dígito: 1 (unidades), 10 (decenas), 100 (centenas), etc.
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSortPorDigito(lista, exp);
    }
}

// Método auxiliar que realiza Counting Sort para un dígito específico
private static void countingSortPorDigito(ArrayList<Integer> lista, int exp) {
    int n = lista.size();

    // Array para almacenar el resultado temporal
    ArrayList<Integer> resultado = new ArrayList<>(Collections.nCopies(n, 0));

    // Array de conteo para los dígitos (0-9)
    int[] conteo = new int[10];

    // FASE 1: CONTAR - contamos la frecuencia de cada dígito en la posición actual
    for (int i = 0; i < n; i++) {
```

```
        // Obtenemos el dígito en la posición exp: (número / exp) % 10
        int digito = (lista.get(i) / exp) % 10;
        conteo[digito]++; // Incrementamos el contador para este dígito
    }

    // FASE 2: ACUMULAR - modificamos el array de conteo para contener posiciones
    for (int i = 1; i < 10; i++) {
        // Cada posición ahora contiene la posición final de ese dígito
        conteo[i] += conteo[i - 1];
    }

    // FASE 3: CONSTRUIR - construimos el array resultado
    // IMPORTANTE: recorremos de derecha a izquierda para mantener la estabilidad
    for (int i = n - 1; i >= 0; i--) {
        // Obtenemos el dígito del elemento actual
        int digito = (lista.get(i) / exp) % 10;

        // Calculamos la posición donde debe ir este elemento en el resultado
        int posicion = conteo[digito] - 1;

        // Colocamos el elemento en la posición calculada
        resultado.set(posicion, lista.get(i));

        // Decrementamos el contador para este dígito
        conteo[digito]--;
    }

    // Copiamos el resultado ordenado por este dígito de vuelta a la lista original
    for (int i = 0; i < n; i++) {
        lista.set(i, resultado.get(i));
    }
}
```

1.2.4 Código con Array Normal

```
public static void radixSort(int[] arr) {
    if (arr.length <= 1) return;

    // Encontrar el valor máximo
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) max = arr[i];
    }

    // Aplicar Counting Sort para cada dígito
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSortPorDigito(arr, exp);
    }
```

```
}

private static void countingSortPorDigito(int[] arr, int exp) {
    int n = arr.length;
    int[] resultado = new int[n];
    int[] conteo = new int[10];

    // Contar frecuencia de dígitos
    for (int i = 0; i < n; i++) {
        int digito = (arr[i] / exp) % 10;
        conteo[digito]++;
    }

    // Acumular posiciones
    for (int i = 1; i < 10; i++) {
        conteo[i] += conteo[i - 1];
    }

    // Construir resultado (de derecha a izquierda para estabilidad)
    for (int i = n - 1; i >= 0; i--) {
        int digito = (arr[i] / exp) % 10;
        resultado[conteo[digito] - 1] = arr[i];
        conteo[digito]--;
    }

    // Copiar resultado al array original
    for (int i = 0; i < n; i++) {
        arr[i] = resultado[i];
    }
}
```

1.2.5 Demostración Paso a Paso

```
ArrayList original: [170, 45, 75, 90, 2, 802, 24, 66]

PRIMERA ITERACIÓN (dígito de unidades - exp=1):
Números por dígito de unidades: [0, 5, 5, 0, 2, 2, 4, 6]
Ordenado por unidades: [170, 90, 2, 802, 24, 45, 75, 66]

SEGUNDA ITERACIÓN (dígito de decenas - exp=10):
Números por dígito de decenas: [7, 9, 0, 0, 2, 4, 7, 6]
Ordenado por decenas: [2, 802, 24, 45, 66, 170, 75, 90]

TERCERA ITERACIÓN (dígito de centenas - exp=100):
Números por dígito de centenas: [0, 8, 0, 0, 0, 1, 0, 0]
Ordenado por centenas: [2, 24, 45, 66, 75, 90, 170, 802]

Lista ordenada: [2, 24, 45, 66, 75, 90, 170, 802]
```

1.3 Quick Sort

1.3.1 ¿Qué hace?

Quick Sort (ordenamiento rápido) usa la estrategia **"divide y vencerás"** seleccionando un elemento como "pivote" y particionando la lista alrededor de este pivote. Los elementos menores van a la izquierda, los mayores a la derecha, y luego se repite el proceso recursivamente.

1.3.2 Cómo funciona

- Elige un elemento como pivote (puede ser el primero, último, medio, o aleatorio)
- Reorganiza la lista: elementos < pivote a la izquierda, elementos > pivote a la derecha
- Aplica recursivamente el mismo proceso a las sublistas izquierda y derecha

1.3.3 Código con ArrayList

```
import java.util.ArrayList;

public static void quickSort(ArrayList<Integer> lista) {
    // Llamamos al método recursivo con los índices inicial y final
    quickSortRecursivo(lista, 0, lista.size() - 1);
}

// Método recursivo que implementa el algoritmo Quick Sort
private static void quickSortRecursivo(ArrayList<Integer> lista, int low, int high) {
    // Caso base: si el segmento tiene 0 o 1 elemento, ya está ordenado
    if (low < high) {
        // Particionamos la lista y obtenemos la posición final del pivote
        int indicePivote = particion(lista, low, high);

        // Ordenamos recursivamente los elementos antes del pivote (subarray izquierdo)
        quickSortRecursivo(lista, low, indicePivote - 1);

        // Ordenamos recursivamente los elementos después del pivote (subarray derecho)
        quickSortRecursivo(lista, indicePivote + 1, high);
    }
}

// Método que particiona la lista alrededor de un pivote
private static int particion(ArrayList<Integer> lista, int low, int high) {
    // Elegimos el último elemento como pivote (estrategia común)
    int pivote = lista.get(high);

    // Índice que indica la posición donde colocaremos el siguiente elemento menor que el pivo
    int i = low - 1;

    // Recorremos todos los elementos desde low hasta high-1
```

```
    for (int j = low; j < high; j++) {
        // Si el elemento actual es menor o igual al pivote
        if (lista.get(j) <= pivote) {
            i++; // Incrementamos el índice de los elementos menores

            // Intercambiamos el elemento actual con el elemento en la posición i
            int temp = lista.get(i);
            lista.set(i, lista.get(j));
            lista.set(j, temp);
        }
    }

    // Colocamos el pivote en su posición correcta
    // Todos los elementos antes de i+1 son <= pivote, todos después son > pivote
    int temp = lista.get(i + 1);
    lista.set(i + 1, lista.get(high)); // Movemos el pivote a su posición final
    lista.set(high, temp);

    // Retornamos la posición final del pivote
    return i + 1;
}
```

1.3.4 Código con Array Normal

```
public static void quickSort(int[] arr) {
    quickSortRecursivo(arr, 0, arr.length - 1);
}

private static void quickSortRecursivo(int[] arr, int low, int high) {
    if (low < high) {
        int indicePivote = particion(arr, low, high);
        quickSortRecursivo(arr, low, indicePivote - 1);
        quickSortRecursivo(arr, indicePivote + 1, high);
    }
}

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

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivote) {
            i++;
            // Intercambiar arr[i] y arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
```

```
    }

    // Colocar el pivote en su posición correcta
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}
```

1.3.5 Demostración Paso a Paso

```
ArrayList original: [10, 80, 30, 90, 40, 50, 70]
Pivote: 70 (último elemento)

PRIMERA PARTICION:
[10, 80, 30, 90, 40, 50, 70]
↓
Elementos <= 70: [10, 30, 40, 50]
Elementos > 70: [80, 90]
Pivote en posición correcta: [10, 30, 40, 50, 70, 80, 90]

PARTICION IZQUIERDA: [10, 30, 40, 50]
Pivote: 50
[10, 30, 40, 50] → ya está ordenado

PARTICION DERECHA: [80, 90]
Pivote: 90
[80, 90] → ya está ordenado

Lista final ordenada: [10, 30, 40, 50, 70, 80, 90]
```

1.4 Heap Sort

1.4.1 ¿Qué hace?

Heap Sort (ordenamiento por montículos) utiliza una estructura de datos llamada **heap** (montículo) para ordenar elementos. Primero convierte la lista en un max-heap (donde el padre es mayor que los hijos), luego extrae repetidamente el elemento máximo y lo coloca al final.

1.4.2 Cómo funciona

- Construye un max-heap a partir de la lista desordenada
- El elemento máximo está siempre en la raíz (posición 0)
- Intercambia la raíz con el último elemento y reduce el tamaño del heap
- Reconstruye el heap y repite el proceso

1.4.3 Código con ArrayList

```
import java.util.ArrayList;

public static void heapSort(ArrayList<Integer> lista) {
    int n = lista.size();

    // FASE 1: CONSTRUIR el max-heap
    // Empezamos desde el último nodo que tiene hijos (n/2 - 1) y vamos hacia la raíz
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(lista, n, i);
    }

    // FASE 2: EXTRAER elementos uno por uno del heap
    for (int i = n - 1; i > 0; i--) {
        // Movemos la raíz actual (máximo) al final
        int temp = lista.get(0);
        lista.set(0, lista.get(i));
        lista.set(i, temp);

        // Llamamos heapify en el heap reducido (excluyendo el último elemento ya ordenado)
        heapify(lista, i, 0);
    }
}

// Método para mantener la propiedad de max-heap
// n: tamaño del heap, i: índice del nodo raíz del subárbol
private static void heapify(ArrayList<Integer> lista, int n, int i) {
    int mayor = i; // Inicializamos el mayor como la raíz
    int izquierdo = 2 * i + 1; // índice del hijo izquierdo = 2*i + 1
    int derecho = 2 * i + 2;   // índice del hijo derecho = 2*i + 2

    // Si el hijo izquierdo existe y es mayor que la raíz
    if (izquierdo < n && lista.get(izquierdo) > lista.get(mayor)) {
        mayor = izquierdo;
    }

    // Si el hijo derecho existe y es mayor que el mayor actual
    if (derecho < n && lista.get(derecho) > lista.get(mayor)) {
        mayor = derecho;
    }

    // Si el mayor no es la raíz
    if (mayor != i) {
        // Intercambiamos la raíz con el mayor de sus hijos
        int temp = lista.get(i);
        lista.set(i, lista.get(mayor));
        lista.set(mayor, temp);
```

```
        // Recursivamente aplicamos heapify al subárbol afectado
        heapify(lista, n, mayor);
    }
}
```

1.4.4 Código con Array Normal

```
public static void heapSort(int[] arr) {
    int n = arr.length;

    // Construir max-heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // Extraer elementos del heap
    for (int i = n - 1; i > 0; i--) {
        // Mover raíz al final
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // heapify en el heap reducido
        heapify(arr, i, 0);
    }
}

private static void heapify(int[] arr, int n, int i) {
    int mayor = i;
    int izquierdo = 2 * i + 1;
    int derecho = 2 * i + 2;

    if (izquierdo < n && arr[izquierdo] > arr[mayor]) {
        mayor = izquierdo;
    }

    if (derecho < n && arr[derecho] > arr[mayor]) {
        mayor = derecho;
    }

    if (mayor != i) {
        int temp = arr[i];
        arr[i] = arr[mayor];
        arr[mayor] = temp;

        heapify(arr, n, mayor);
    }
```

```
}
```

1.4.5 Demostración Paso a Paso

```
ArrayList original: [4, 10, 3, 5, 1]

FASE 1: CONSTRUIR MAX-HEAP:
Array inicial: [4, 10, 3, 5, 1]
Heapify en índice 1: [4, 10, 3, 5, 1] → 10 > 4 y 10 > 5,
intercambia 4 y 10 → [10, 4, 3, 5, 1]
Heapify en índice 0: [10, 4, 3, 5, 1] → 10 > 4 y 10 > 3,
ya es max-heap → [10, 5, 3, 4, 1]

Max-heap construido:
      10
     /  \
    5    3
   / \
  4   1

FASE 2: EXTRAER ELEMENTOS:
Intercambiar 10 y 1: [1, 5, 3, 4, 10]
Heapify en raíz: [5, 4, 3, 1, 10]

Intercambiar 5 y 1: [1, 4, 3, 5, 10]
Heapify en raíz: [4, 1, 3, 5, 10]

Intercambiar 4 y 3: [3, 1, 4, 5, 10]
Heapify en raíz: [3, 1, 4, 5, 10]

Intercambiar 3 y 1: [1, 3, 4, 5, 10]

Lista ordenada: [1, 3, 4, 5, 10]
```