Taller - Algoritmos de ordenamiento y búsqueda

Andrés Felipe Zuñiga Zuluaga
Código: 1094889063

Natalia Tejada Cardona
Código: 1092851258

Andrés Eduardo Pérez Martínez
Código: 1035972060

Entregado a:
Carlos Andrés Flórez

Universidad del Quindío
Facultad de Ingeniería
Ingeniería de sistemas y computación
Análisis de algoritmos
Armenia, Quindío. 2025

TALLER DE ORDENAMIENTO Y BÚSQUEDA

El objetivo de este taller es investigar, analizar y comparar el desempeño de diferentes algoritmos de ordenamiento y búsqueda, verificando que los resultados experimentales se correspondan con sus complejidades teóricas.

Punto 1. Análisis de Algoritmos de Ordenamiento

Se solicita registrar el **tiempo de ejecución** y el **orden de complejidad teórica** de los siguientes algoritmos de ordenamiento:

- **Shaker Sort =** **O(n2)**
- **Dual-Pivot QuickSort =** **O(n log n)**
- **Heap Sort =** **O(n log n)**
- **Merge Sort =** **O(n log n)**
- **Radix Sort =** **O(nk)**

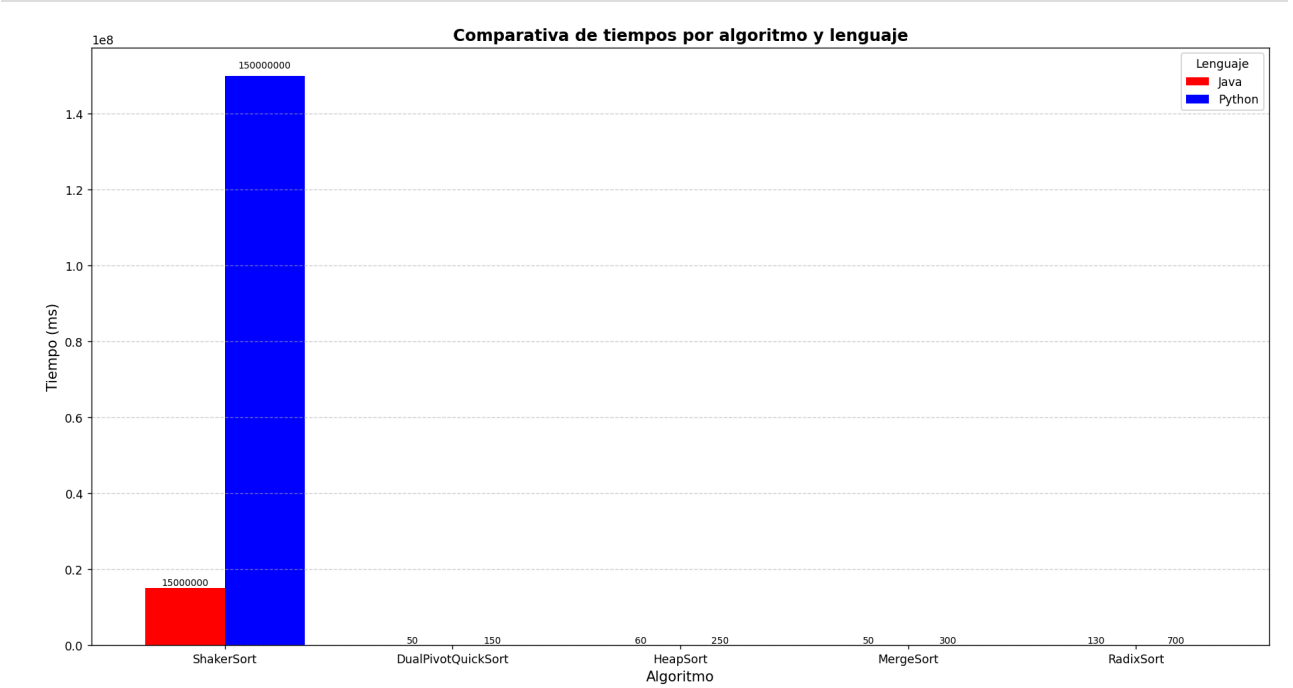
Cada algoritmo deberá probarse con **tres tamaños de arreglo diferentes**:

- **10.000 elementos**
- **100.000 elementos**
- **1.000.000 elementos**

Cada valor del arreglo debe ser un **número aleatorio de 8 dígitos**, generado de forma automática y cargado desde un **archivo de texto plano**. Durante las pruebas, se debe **medir únicamente el tiempo de ejecución del algoritmo de ordenamiento**, excluyendo los tiempos de lectura o escritura de archivos.

Los resultados obtenidos deberán **almacenarse y visualizarse en un gráfico de barras comparativo**. Este gráfico debe mostrar **una barra por lenguaje de programación** (dos barras por algoritmo) y los **valores de tiempo visibles** sobre cada barra, para facilitar la interpretación.

Gráfico para algoritmos de ordenamiento



Punto 2. Análisis de Algoritmos de Búsqueda

Se solicita **repetir el procedimiento anterior**, pero ahora implementando y evaluando **algoritmos de búsqueda**. Los arreglos deben contener **números enteros de 8 dígitos**, y los tiempos deben medirse **únicamente durante la ejecución del algoritmo de búsqueda**, excluyendo los tiempos de lectura o escritura de archivos.

Los algoritmos a implementar y analizar son los siguientes:

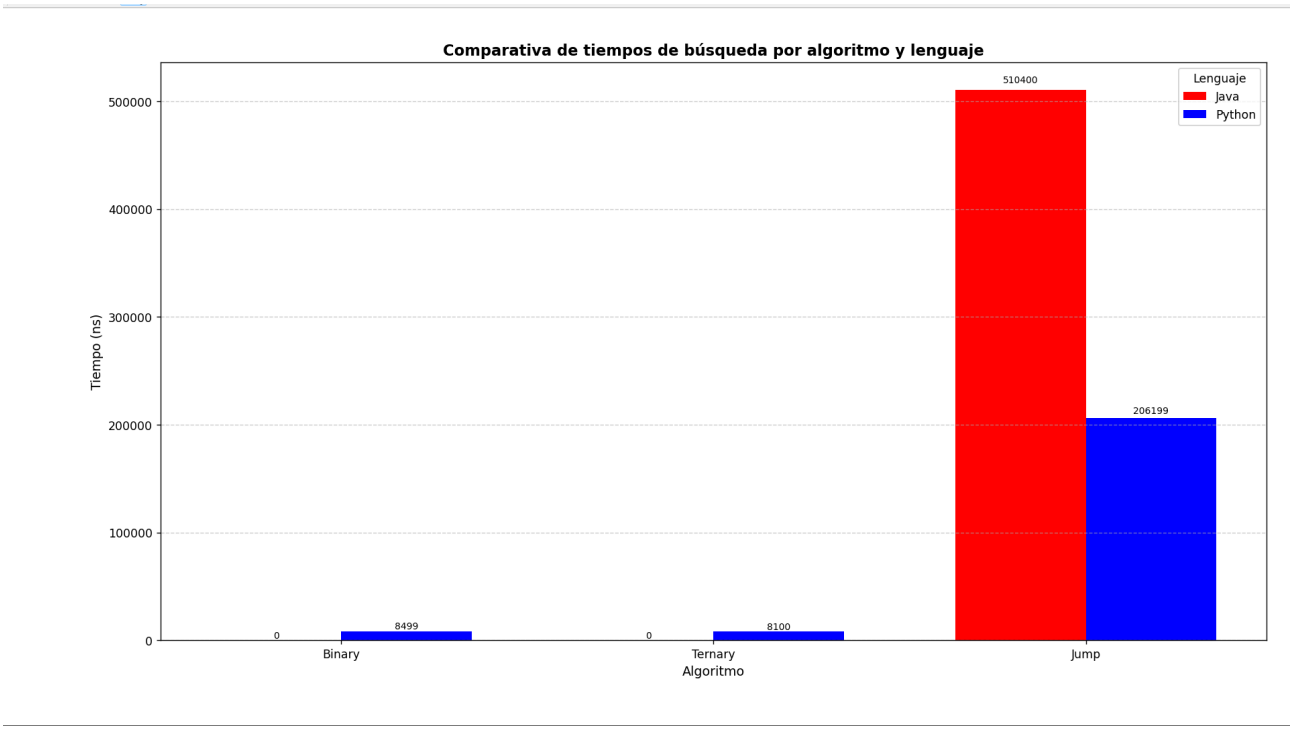
- **Búsqueda Binaria =** **O(log n)**
- **Búsqueda Ternaria =** **O(log n)**
- **Búsqueda por Saltos (Jump Search) =** **O(√n)**

Cada algoritmo deberá probarse con **tres tamaños de arreglo diferentes**:

- **10.000 elementos**
- **100.000 elementos**
- **1.000.000 elementos**

Los resultados deberán **presentarse y almacenarse del mismo modo que en el punto anterior**, generando un gráfico de barras comparativo que muestre el tiempo de ejecución en ambos lenguajes de programación y los valores visibles sobre cada barra.

Gráfico para algoritmos de búsqueda



Punto 3. Análisis de Casos Específicos

Dados los siguientes problemas, realice una explicación del algoritmo y una explicación del tiempo de ejecución T(n).

1. https://www.geeksforgeeks.org/sort-n-numbers-range-0-n2-1-linear-time/

Sort n numbers in range from 0 to n^2 - 1 in linear time

```
void countSort(int arr[], int n, int exp)
{
    int output[] = new int[n]; // output array
    int i, count[] = new int[n] ;
    for (i=0; i < n; i++)
        count[i] = 0;

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%n ]++;

    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < n; i++)
        count[i] += count[i - 1];

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

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}


// The main function to that sorts arr[] of size n using Radix Sort
public void sort(int arr[], int n)
{
    // Do counting sort for first digit in base n. Note that
    // instead of passing digit number, exp (n^0 = 1) is passed.
    countSort(arr, n, 1);

    // Do counting sort for second digit in base n. Note that
    // instead of passing digit number, exp (n^1 = n) is passed.
    countSort(arr, n, n);
}
```

Este método tiene una complejidad de **T(n) = 16n+15,** esto se demuestra en la siguiente tabla

Sección	Costo

Fijo	6
Primer for	3n + 2
Segundo for	3n + 2
Tercer for	3n + 1
Cuarto for	4n + 2
Quinto for	3n + 2
TOTAL	**(3 + 3 + 3 + 4 + 3)n + (6 + 2 + 2 + 1 + 2 + 2)**
	16n + 15

Sin embargo, este es un algoritmo auxiliar, que va a ser utilizado por el algoritmo que viene a continuación.

```
// The main function to that sorts arr[] of size n using Radix Sort
public void sort(int arr[], int n)
{
   // Do counting sort for first digit in base n. Note that
   // instead of passing digit number, exp (n^0 = 1) is passed.
   countSort(arr, n, 1);

   // Do counting sort for second digit in base n. Note that
   // instead of passing digit number, exp (n^1 = n) is passed.
   countSort(arr, n, n);
}
```

Por lo que el método de arriba se debe de ejecutar dos veces, sumandole los dos parámetros con los que cuenta este método **sort,** así que el resultado sería.

$$T(n) = 2 * (16n + 15) + 2$$
$$T(n) = 32n + 32$$

El método countSort es una versión del ordenamiento por conteo usada dentro del Radix Sort.
Su función es ordenar los elementos de un arreglo según un dígito específico (por ejemplo, unidades o decenas), sin hacer comparaciones directas entre ellos.

Primero cuenta cuántas veces aparece cada dígito, luego calcula las posiciones acumuladas y finalmente coloca los elementos en orden dentro de un arreglo auxiliar (output). Después, copia el resultado al arreglo original.

El método sort llama a countSort varias veces, una por cada posición de dígito, logrando así un ordenamiento completo.

Es útil porque tiene complejidad lineal O(n), es estable y muy eficiente para ordenar números enteros grandes o con varios dígitos.

2. https://www.geeksforgeeks.org/sort-array-according-order-defined-another-array/

Sort an array according to the order defined by another array

En este algoritmo se presentan dos métodos para solucionar un problema de ordenar un arreglo con respecto al ordenamiento que solicita un segundo arreglo, es decir, que al ordenar siga una secuencia

solicitada por un segundo arreglo, en caso de que existan elementos que no existan en el segundo arreglo, el ordenamiento va a ser ascendente. A continuación se presenta la primera forma de completar este problema.

Sección	Operaciones (desglose)	Coste
Parámetros y estructuras iniciales	paso de a1,a2 (2) + m = ... (1) + n = ... (1) + new HashMap() (1)	**5**
1) Primer for (contar freq en a1)	for i=0..m-1: init(1) + (m+1) comps + m increments + m × (getOrDefault+put) (2×m)	**4m + 2**
index = 0	asignación	**1**
2a) for externo sobre a2	init(1) + (n+1) comps + n increments	**2n + 2**
2b) while interno (todas las iteraciones)	suma total sobre todos i: chequeos ≈ (k + n) + cuerpo por iteración ≈ 4 per iter → **~5k + n** (v. explicación)	**5k + n**
2c) freq.remove(a2[i])	un remove por cada i → **n**	**n**
Subtotal bloque 2 (for+while+remove)	suma de 2a+2b+2c	**5k + 4n + 2**
3a) Crear remaining	new ArrayList()	**1**
3b) Rellenar remaining desde freq	para todas las entradas: inicializaciones d + comparaciones totales r + d + increments r + cuerpo por iteración remaining.add(entry.getKey()) (2×r) → total **4r + 2d**	**4r + 2d**
Subtotal construir remaining	3a + 3b	**4r + 2d + 1**
4) Collections.sort(remaining)	coste de ordenar r elementos	**Sort(r)** (≈ Θ(r log r))
5) Append remaining a a1	init(1) + (r+1) comps + r increments + por iter. a1[index++]=num (2×r) → **4r + 2**	**4r + 2**

$$T(m, n, k, r, d) = 4m + 4n + 5k + 8r + 2d + Collections.sort(r) + 13$$

Sustituyendo r=m−k:

$$T(n, m, k, d) = 12m + 4n - 3k + 2d + Collections.sort(r) + 13$$

El método **relativeSort** ordena el arreglo a1 siguiendo el orden definido por otro arreglo a2.
Primero cuenta cuántas veces aparece cada elemento de a1, luego coloca en a1 todos los que están en a2 manteniendo ese orden, y finalmente agrega los elementos restantes (los que no están en a2) ordenados de menor a mayor.

Es útil porque permite ordenar un arreglo de forma personalizada, respetando un orden de referencia específico y dejando el resto de los elementos ordenados naturalmente. Esto se aplica, por ejemplo, en sistemas donde ciertos valores tienen prioridad o jerarquía predefinida.

En el link que se proporciona, existe otra forma de ejecutar este problema de ordenar un arreglo con respecto a algunas directrices de otro. Este es el método y su complejidad temporal.

```java
public void relativeSort(int[] a1, int[] a2) {

    // Create a map to store the index of each
    // element in a2
    Map<Integer, Integer> mp = new HashMap<>();
    for (int i = 0; i < a2.length; i++) {
        if (!mp.containsKey(a2[i]))
            mp.put(a2[i], i);
    }

    // Convert int[] to Integer[] for custom sorting
    Integer[] temp = Arrays.stream(a1).boxed().toArray(Integer[]::new);

    // Custom comparator using lambda
    Arrays.sort(temp, (a, b) -> {
        boolean aIn = mp.containsKey(a), bIn = mp.containsKey(b);
        if (!aIn && !bIn) return Integer.compare(a, b);
        if (!aIn) return 1;
        if (!bIn) return -1;
        return Integer.compare(mp.get(a), mp.get(b));
    });

    // Copy sorted values back to original array
    for (int i = 0; i < a1.length; i++) {
        a1[i] = temp[i];
    }
}
```

Sección	Operaciones (desglose)	Costo
Parámetros y declaración de mp	Paso de parámetros (2) + new HashMap() (1)	**3**
for sobre a2	Init (1) + (n+1) comps + n increments + cuerpo: containsKey (1) + posible put (1) por iteración → 2×n	**4n + 2**
Conversión a1 a temp	Arrays.stream(a1) (1) + boxed() (m) + toArray() (m) + asignación (1)	**2m + 2**
Arrays.sort(temp, comparator)	Comparaciones y accesos promedio: m log m comparaciones, cada una con hasta 4 operaciones (containsKey, get, compare)	**≈ 4m log m**
for final (copiar resultados)	Init (1) + (m+1) comps + m increments + m asignaciones	**3m + 2**

$$T(m,n) = 4mlog(m) + 5m + 4n + 9$$

El método relativeSort ordena el arreglo a1 según el orden definido por otro arreglo a2, usando un comparador personalizado.

Primero guarda en un HashMap la posición de cada elemento de a2, luego convierte a1 a un arreglo de objetos Integer y lo ordena con una lambda que compara según ese orden.
Finalmente, copia los valores ordenados de nuevo en a1.

Es útil cuando se necesita ordenar datos siguiendo un orden de prioridad o referencia definido por otro arreglo, sin perder estabilidad ni usar estructuras adicionales complejas.

3. https://www.geeksforgeeks.org/sort-a-linked-list-of-0s-1s-or-2s/

```
public void sortList(Node head) {

    // Initialize count of '0', '1' and '2' as 0
    int[] cnt = { 0, 0, 0 };
    Node ptr = head;

    // Traverse and count total number of '0', '1' and '2'
    // cnt[0] will store total number of '0's
    // cnt[1] will store total number of '1's
    // cnt[2] will store total number of '2's
    while (ptr != null) {
        cnt[ptr.data] += 1;
        ptr = ptr.next;
    }

    int idx = 0;
    ptr = head;

    // Fill first cnt[0] nodes with value 0
    // Fill next cnt[1] nodes with value 1
    // Fill remaining cnt[2] nodes with value 2
    while (ptr != null) {
        if (cnt[idx] == 0)
            idx += 1;
        else {
            ptr.data = idx;
            cnt[idx] -= 1;
            ptr = ptr.next;
        }
    }
}
```

Sección	Descripción	Costo
Parámetro y declaración de cnt	Paso de head (1) + creación del arreglo cnt (1) + asignación (1)	**3**
Asignación inicial ptr = head	1	**1**

Primer while (recorrer lista y contar)	Init (1) + (n+1) comparaciones ptr != null + n accesos a cnt[ptr.data] (1 cada uno) + n incrementos +=1 + n actualizaciones ptr = ptr.next	**4n + 2**
Asignación idx = 0	1	**1**
Reinicio ptr = head	1	**1**
Segundo while (reescribir valores)	Init (1) + (n+1) comparaciones ptr != null + En cada iteración puede ejecutar una de dos ramas:	**—**
— Rama if (cnt[idx] == 0) → ejecuta en total ≤ 3 veces (por valores 0,1,2), así que constante	3	
— Rama else: ptr.data=idx + cnt[idx]-- + ptr=ptr.next (3 operaciones por nodo) → ejecutada n veces	3n	
Subtotal segundo while	Suma de todo lo anterior	**4n + 4**

$$T(n) = 8n + 11$$

El método sortList ordena una lista enlazada que solo contiene los valores 0, 1 y 2, sin usar estructuras adicionales complejas.

Primero cuenta cuántos nodos hay de cada tipo y luego reescribe los valores de la lista en orden (todos los 0, luego los 1 y al final los 2).

Es útil porque permite ordenar una lista con valores limitados en una sola pasada lineal, sin necesidad de crear nuevas listas o usar algoritmos de ordenamiento comparativo.

Este enfoque es eficiente y optimizado para estructuras como listas enlazadas con pocos valores posibles, típico en problemas de clasificación o conteo de categorías.

Referencias:

GeeksforGeeks. (s.f.). *Binary Search*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/binary-search/

GeeksforGeeks. (s.f.). *Ternary Search*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/ternary-search/

GeeksforGeeks. (s.f.). *Jump Search*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/jump-search/

GeeksforGeeks. (s.f.). Cocktail Sort (Shaker Sort). GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/cocktail-sort/

GeeksforGeeks. (s.f.). *Dual Pivot QuickSort*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/dual-pivot-quicksort/

GeeksforGeeks. (s.f.). *Heap Sort*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/heap-sort/

GeeksforGeeks. (s.f.). *Merge Sort*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/merge-sort/

GeeksforGeeks. (s.f.). *Radix Sort*. GeeksforGeeks. Recuperado el 13 de octubre de 2025, de https://www.geeksforgeeks.org/radix-sort/