UNIVERSIDAD DA VINCI DE GUATEMALA

FACULTAD DE INGENIERÍA, INDUSTRIA Y TECNOLOGÍA

CARRERA: LICENCIATURA EN INGENIERÍA EN SISTEMAS

CURSO: ESTRUCTURA DE DATOS

PARCIAL 1

Herberth Luis Ricardo Ortiz Cruz

Carné: 2010820470101

Guatemala, 27 febrero de 2026

INTRODUCCIÓN

En este proyecto se implementaron cuatro algoritmos clásicos en sus versiones iterativa y recursiva: factorial, Fibonacci, búsqueda lineal y ordenamiento burbuja. El objetivo fue comparar el rendimiento entre ambas estrategias y analizar su complejidad temporal mediante mediciones experimentales utilizando System.nanoTime(). Posteriormente se graficaron los resultados en Excel para observar su tendencia de crecimiento y determinar su comportamiento Big-O.

DESCRIPCIÓN DE ALGORITMOS

Factorial

- Iterativo: usa ciclo for.
- Recursivo: caso base n = 0 o n = 1.

El algoritmo del factorial calcula el producto de todos los números enteros positivos desde 1 hasta n. En la versión iterativa se implementó mediante un ciclo for, acumulando el resultado en una variable auxiliar. Esta versión ejecuta exactamente n multiplicaciones, lo que implica una complejidad temporal O(n).

La versión recursiva se basa en la definición matemática del factorial:

$n! = n \times (n-1)!$ con caso base $0! = 1$.

En cada llamada se reduce el problema en una unidad hasta alcanzar el caso base. Aunque la lógica es más expresiva, su complejidad temporal también es O(n), pero consume memoria adicional debido a la pila de llamadas.

```java
public class Factorial {

    // Factorial Iterativo
    public static long factorialIterativo(int n) {

        long resultado = 1;

        for (int i = 1; i <= n; i++) {
            resultado = resultado * i;
        }

        return resultado;
    }

    // Factorial Recursivo
    public static long factorialRecursivo(int n) {

        // Caso base
        if (n == 0 || n == 1) {
            return 1;
        }

        // Llamada recursiva
        return n * factorialRecursivo(n - 1);
    }

    // Método main para probar
    public static void main(String[] args) {

        int numero = 5;

        System.out.println("Factorial Iterativo de " + numero + " = "
                + factorialIterativo(numero));

        System.out.println("Factorial Recursivo de " + numero + " = "
                + factorialRecursivo(numero));
    }
}
```

Fibonacci

- Iterativo: usa acumuladores.
- Recursivo: sin memorización.

El algoritmo de Fibonacci calcula el término n de la sucesión donde cada elemento es la suma de los dos anteriores.

La versión iterativa utiliza variables acumuladoras y un ciclo que se ejecuta n veces, lo que produce una complejidad temporal O(n).

En contraste, la versión recursiva implementada no utiliza memorización. Esto provoca que se recalculen múltiples subproblemas repetidamente, generando un crecimiento exponencial en el número de llamadas. Por esta razón, su complejidad temporal es O(2^n), lo cual se refleja claramente en las gráficas obtenidas.

```java
public class Fibonacci {

    // Fibonacci Iterativo
    public static long fibonacciIterativo(int n) {

        if (n <= 1) {
            return n;
        }

        long a = 0;
        long b = 1;
        long resultado = 0;

        for (int i = 2; i <= n; i++) {
            resultado = a + b;
            a = b;
            b = resultado;
        }

        return resultado;
    }

    // Fibonacci Recursivo (sin memoización)
    public static long fibonacciRecursivo(int n) {

        if (n <= 1) {
            return n;
        }

        return fibonacciRecursivo(n - 1) + fibonacciRecursivo(n - 2);
    }

    // Método main para probar
    public static void main(String[] args) {

        int numero = 10;

        System.out.println("Fibonacci Iterativo de " + numero + " = "
                + fibonacciIterativo(numero));

        System.out.println("Fibonacci Recursivo de " + numero + " = "
                + fibonacciRecursivo(numero));
    }
}
```

Búsqueda Lineal

- Iterativo: recorre arreglo.
- Recursivo: reduce subproblema.

La búsqueda lineal consiste en recorrer un arreglo elemento por elemento hasta encontrar el valor deseado o llegar al final.

En la versión iterativa se utiliza un ciclo que compara cada elemento con el valor buscado.

La versión recursiva reduce el tamaño del subproblema en cada llamada, avanzando el índice hasta encontrar el elemento o alcanzar el límite del arreglo.

En ambos casos, en el peor escenario (cuando el elemento se encuentra al final), se deben realizar n comparaciones. Por ello, la complejidad temporal es O(n).

```java
public class BusquedaLineal {

    // Búsqueda Lineal Iterativa
    public static int busquedaIterativa(int[] arreglo, int valor) {

        for (int i = 0; i < arreglo.length; i++) {
            if (arreglo[i] == valor) {
                return i;
            }
        }

        return -1; // No encontrado
    }

    // Búsqueda Lineal Recursiva
    public static int busquedaRecursiva(int[] arreglo, int valor, int indice) {

        // Caso base: llegó al final
        if (indice >= arreglo.length) {
            return -1;
        }

        if (arreglo[indice] == valor) {
            return indice;
        }

        return busquedaRecursiva(arreglo, valor, indice + 1);
    }

    // Método main para probar
    public static void main(String[] args) {

        int[] datos = {3, 7, 10, 25, 40};
        int valor = 25;

        int resultadoIter = busquedaIterativa(datos, valor);
        int resultadoRec = busquedaRecursiva(datos, valor, 0);

        System.out.println("Iterativa encontró en índice: " + resultadoIter);
        System.out.println("Recursiva encontró en índice: " + resultadoRec);
    }
```

Burbuja

- Iterativo: ciclos anidados.
- Recursivo: misma lógica con llamadas.

El algoritmo de ordenamiento burbuja organiza los elementos de un arreglo comparando pares adyacentes e intercambiándolos cuando es necesario.

La versión iterativa utiliza dos ciclos anidados, lo que implica aproximadamente n^2 comparaciones en el peor caso.

La versión recursiva replica la misma lógica reduciendo progresivamente el tamaño del arreglo en cada llamada.

En ambos casos la complejidad temporal es $O(n^2)$, lo cual se evidenció experimentalmente en el crecimiento acelerado de los tiempos conforme aumentó el tamaño del arreglo.

```java
public class Burbuja {

    // Burbuja Iterativo
    public static void burbujaIterativo(int[] arreglo) {

        int n = arreglo.length;

        for (int i = 0; i < n - 1; i++) {

            for (int j = 0; j < n - i - 1; j++) {

                if (arreglo[j] > arreglo[j + 1]) {

                    int temp = arreglo[j];
                    arreglo[j] = arreglo[j + 1];
                    arreglo[j + 1] = temp;
                }
            }
        }
    }

    // Burbuja Recursivo
    public static void burbujaRecursivo(int[] arreglo, int n) {

        // Caso base
        if (n == 1) {
            return;
```

```
        }

        for (int i = 0; i < n - 1; i++) {

            if (arreglo[i] > arreglo[i + 1]) {

                int temp = arreglo[i];
                arreglo[i] = arreglo[i + 1];
                arreglo[i + 1] = temp;
            }
        }

        burbujaRecursivo(arreglo, n - 1);
    }

    // Método para imprimir arreglo
    public static void imprimir(int[] arreglo) {

        for (int num : arreglo) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    // Método main para probar
    public static void main(String[] args) {

        int[] datos1 = {5, 2, 9, 1, 3};
        int[] datos2 = {5, 2, 9, 1, 3};

        burbujaIterativo(datos1);
        burbujaRecursivo(datos2, datos2.length);

        System.out.print("Iterativo ordenado: ");
        imprimir(datos1);

        System.out.print("Recursivo ordenado: ");
        imprimir(datos2);
    }
}
```

METODOLOGÍA DE MEDICIÓN

Para la medición del rendimiento se utilizó el método System.nanoTime(), el cual permite obtener el tiempo en nanosegundos antes y después de ejecutar cada algoritmo.

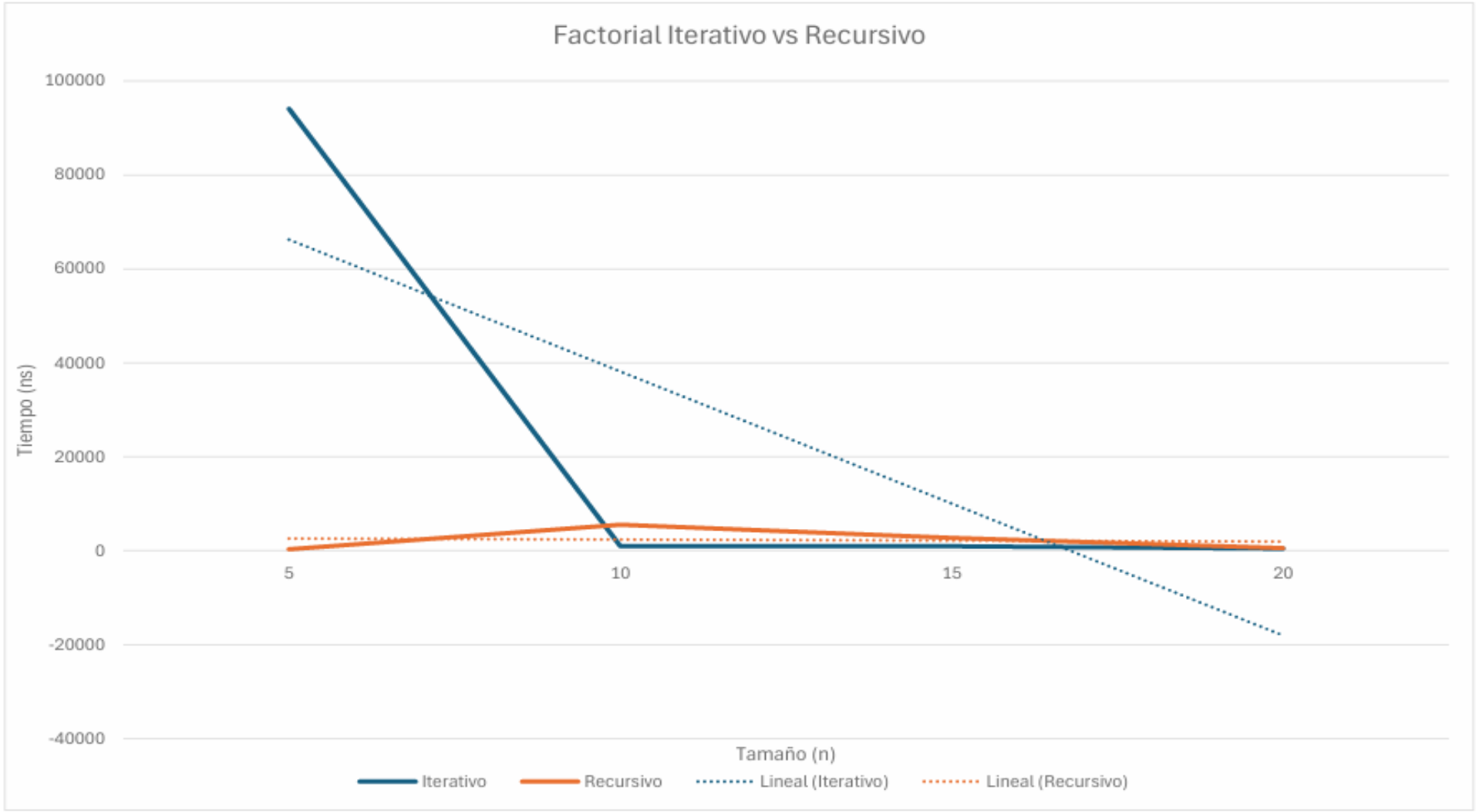
Con el fin de reducir variaciones producidas por procesos internos de la máquina virtual de Java (JVM), cada prueba se ejecutó cinco veces para cada tamaño de entrada y se calculó el promedio aritmético.

La inicialización de arreglos y la generación de datos de prueba no fueron incluidos dentro del tiempo medido, garantizando que únicamente se evaluara el tiempo correspondiente a la ejecución del algoritmo.

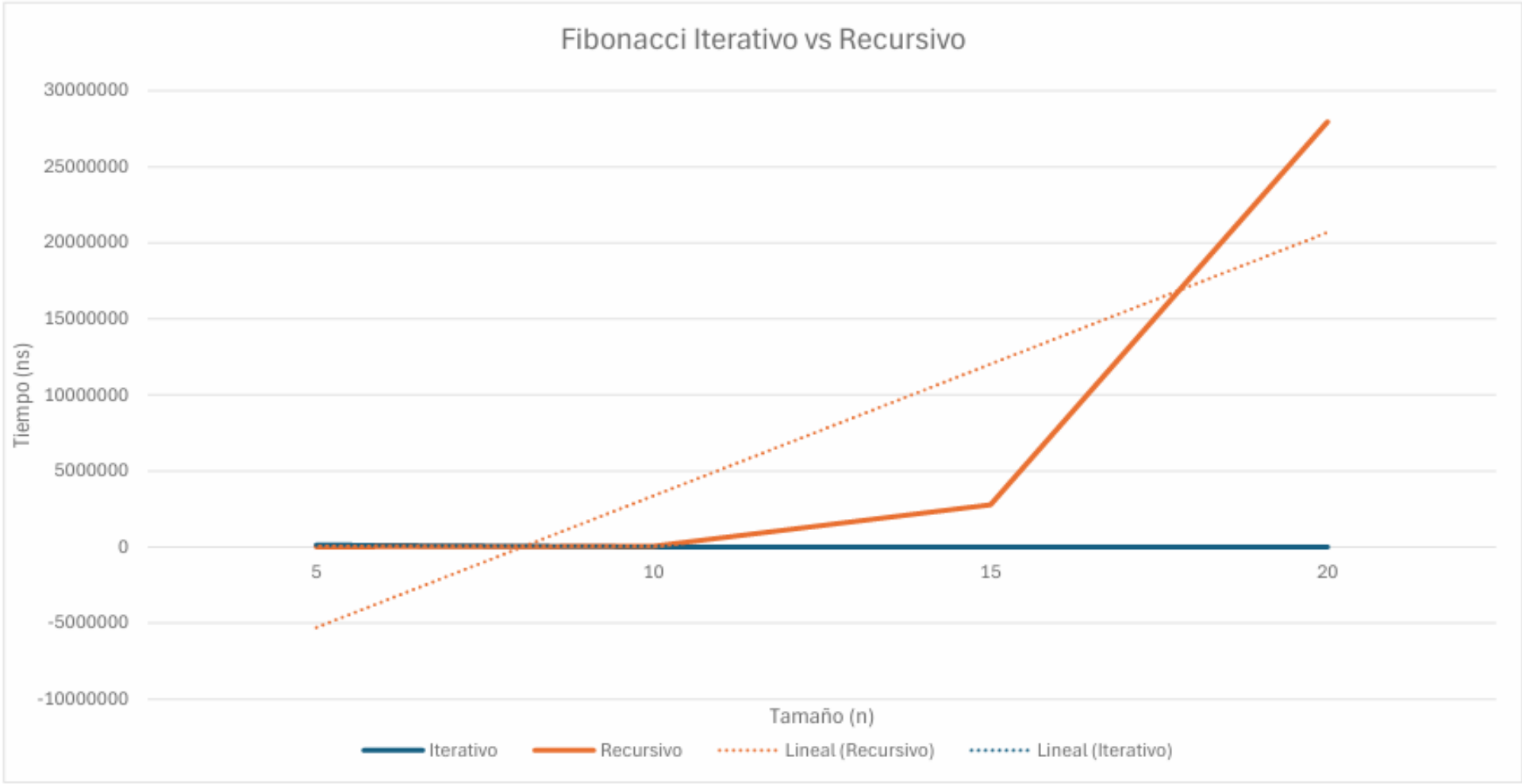
Las pruebas se realizaron con distintos tamaños de entrada con el objetivo de observar el comportamiento del crecimiento temporal y contrastarlo con su análisis teórico de complejidad Big-O.

```
PS C:\Users\hlrortiz\OneDrive - Fundacion Genesis Empresarial\Escritorio\java-tarea\IterativoVsRecursivo> javac src/*.java
>> java -cp src MedicionTiempos
FACTORIAL
n = 5 | Iterativo: 94100 ns | Recursivo: 300 ns
n = 10 | Iterativo: 920 ns | Recursivo: 5520 ns
n = 15 | Iterativo: 940 ns | Recursivo: 2700 ns
n = 20 | Iterativo: 420 ns | Recursivo: 520 ns

FIBONACCI
n = 10 | Iterativo: 141260 ns | Recursivo: 7220 ns
n = 20 | Iterativo: 460 ns | Recursivo: 54300 ns
n = 30 | Iterativo: 8540 ns | Recursivo: 2783720 ns
n = 35 | Iterativo: 680 ns | Recursivo: 27946380 ns

BUSQUEDA LINEAL
n = 500 | Iterativo: 166780 ns | Recursivo: 43760 ns
n = 1000 | Iterativo: 5420 ns | Recursivo: 5800 ns
n = 2000 | Iterativo: 11640 ns | Recursivo: 14920 ns
n = 3000 | Iterativo: 19580 ns | Recursivo: 20340 ns

BURBUJA
n = 1000 | Iterativo: 1358700 ns | Recursivo: 2447300 ns
n = 2000 | Iterativo: 914200 ns | Recursivo: 898660 ns
n = 3000 | Iterativo: 1853280 ns | Recursivo: 2059000 ns
n = 4000 | Iterativo: 3305540 ns | Recursivo: 3173080 ns
PS C:\Users\hlrortiz\OneDrive - Fundacion Genesis Empresarial\Escritorio\java-tarea\IterativoVsRecursivo>
```

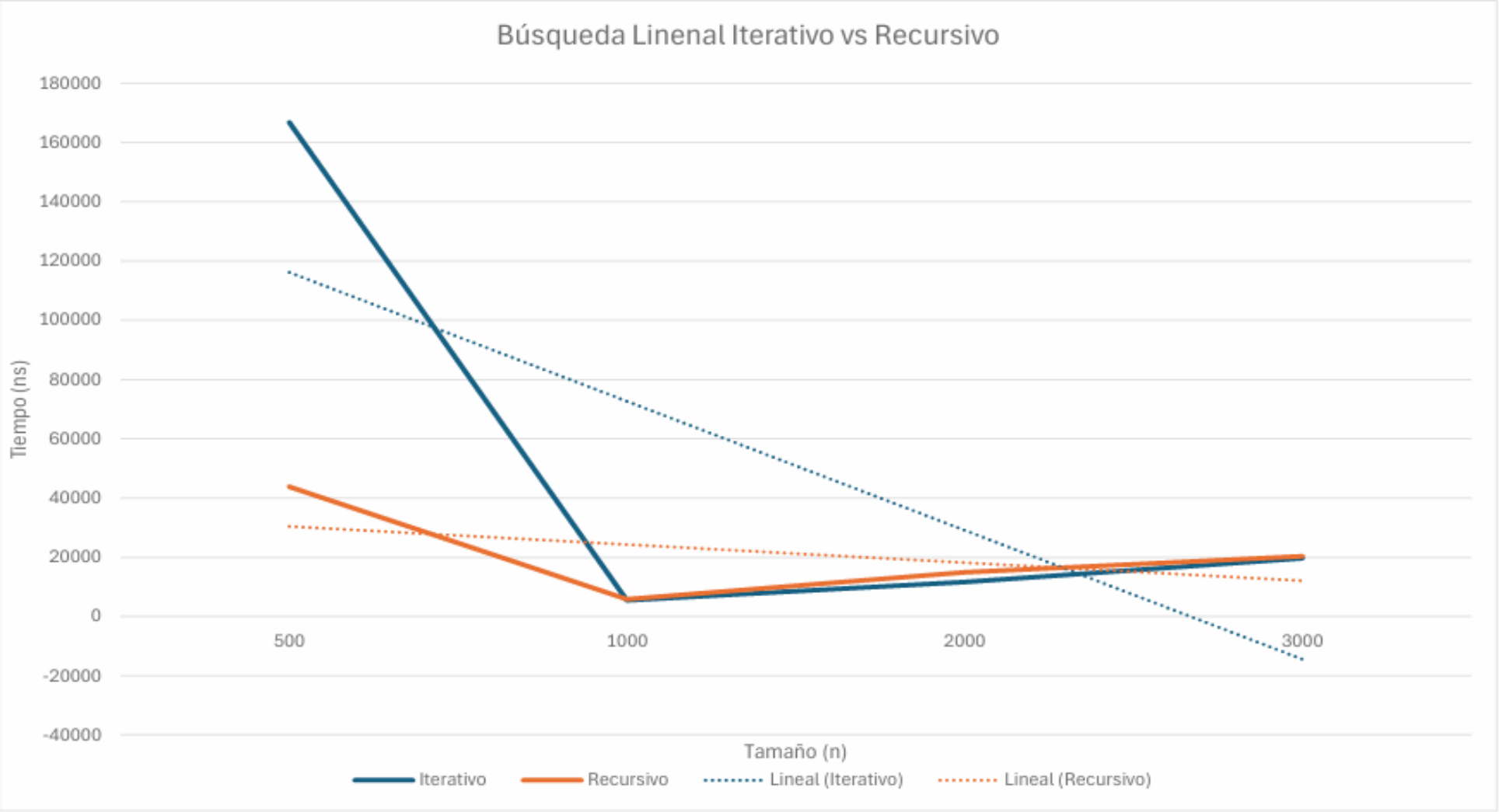
	Iterativo	Recursivo
5	94100	300
10	920	5520
15	940	2700
20	420	520



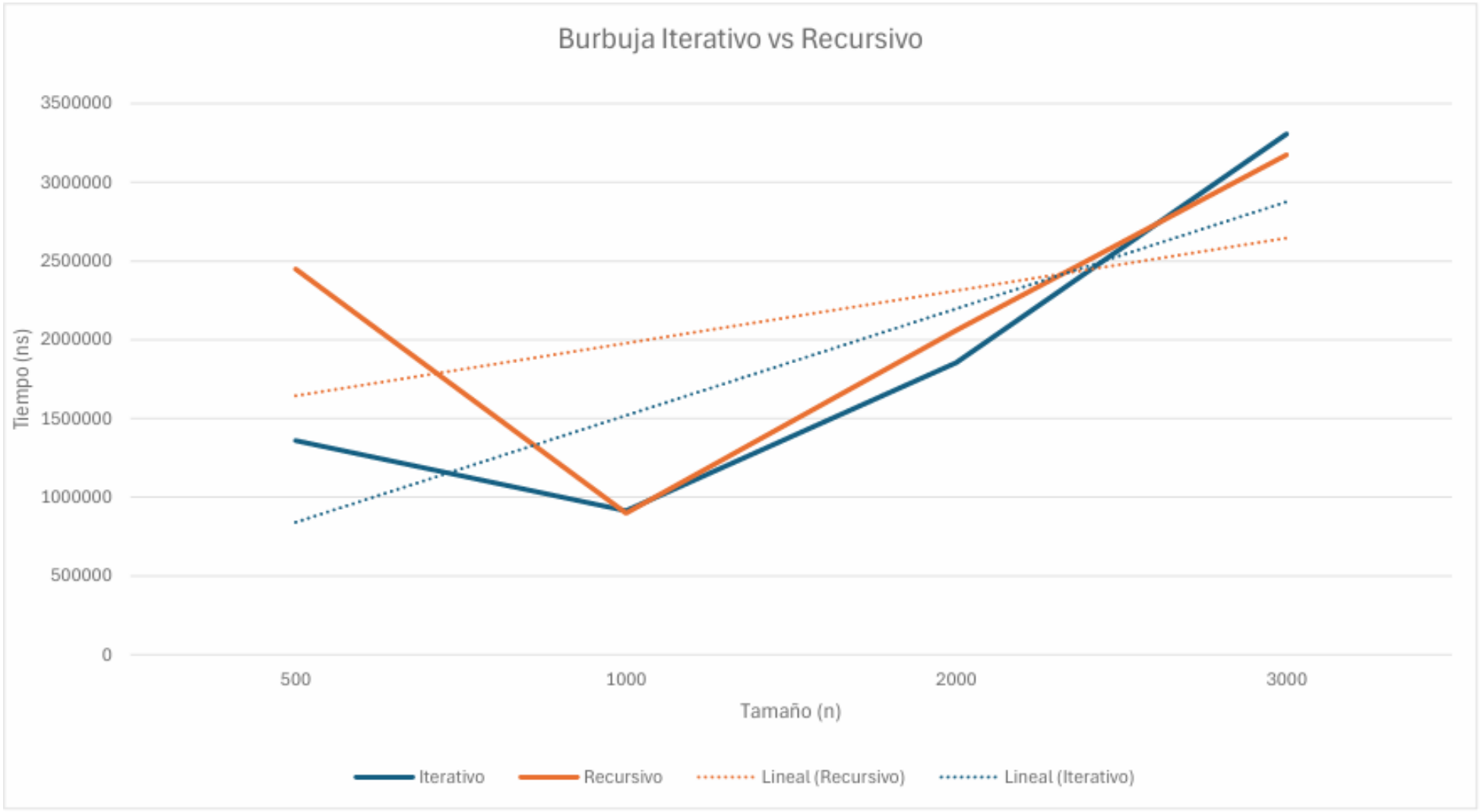
	Iterativo	Recursivo
5	141260	7220
10	460	54300
15	8540	2783720
20	680	27946380



	Iterativo	Recursivo
500	166780	43760
1000	5420	5800
2000	11640	14920
3000	19580	20340



	Iterativo	Recursivo
500	1358700	2447300
1000	914200	898660
2000	1853280	2059000
3000	3305540	3173080



ANÁLISIS DE RESULTADOS

1. Factorial

En las pruebas realizadas para el algoritmo factorial, tanto en su versión iterativa como recursiva, se observa que el tiempo de ejecución se mantiene relativamente estable conforme aumenta el valor de n.

Esto se debe a que ambas implementaciones ejecutan exactamente n operaciones multiplicativas, por lo que su crecimiento es lineal.

La diferencia de tiempos entre ambas versiones no es significativa en términos de complejidad, aunque la versión recursiva puede presentar una ligera sobrecarga debido a la gestión de la pila de llamadas.

Las gráficas confirman que el comportamiento es proporcional al tamaño de entrada, validando su complejidad O(n).

2. Fibonacci

En el caso del algoritmo Fibonacci se observa una diferencia considerable entre la versión iterativa y la recursiva.

La versión iterativa muestra un crecimiento lineal moderado conforme aumenta n, lo cual coincide con su complejidad O(n).

En contraste, la versión recursiva sin memoización presenta un crecimiento exponencial, evidenciado en el incremento abrupto de los tiempos para valores mayores como n = 30 y n = 35.

Este comportamiento confirma que la implementación recursiva realiza múltiples cálculos repetidos, lo que provoca una complejidad O(2^n).

La gráfica muestra claramente esta diferencia en pendiente, siendo el ejemplo más representativo del impacto que puede tener la recursividad no optimizada.

3. Búsqueda Lineal

Para el algoritmo de búsqueda lineal, ambas implementaciones muestran un crecimiento proporcional al tamaño del arreglo.

En el peor caso, cuando el elemento se encuentra al final del arreglo, es necesario recorrer todos los elementos. Las gráficas reflejan un comportamiento aproximadamente lineal, lo cual confirma la complejidad O(n).

No se observa una diferencia significativa entre la versión iterativa y recursiva en términos de crecimiento temporal, aunque la versión recursiva puede implicar un ligero costo adicional por las llamadas sucesivas.

4. Ordenamiento Burbuja

En el algoritmo de ordenamiento burbuja se aprecia un crecimiento acelerado conforme aumenta el tamaño del arreglo.

Tanto la versión iterativa como la recursiva muestran una tendencia cuadrática, lo cual se explica por el uso de ciclos anidados o llamadas recursivas equivalentes. A medida que el tamaño del arreglo se duplica, el tiempo de ejecución aumenta de manera considerable, confirmando una complejidad $O(n^2)$. Las gráficas muestran de manera clara esta tendencia, especialmente en los tamaños mayores evaluados.

Comparación General

A partir de los resultados experimentales se puede concluir que la complejidad teórica Big-O se refleja claramente en los datos obtenidos.

Los algoritmos lineales presentan crecimiento proporcional al tamaño de entrada, mientras que los algoritmos cuadráticos y exponenciales muestran incrementos mucho más pronunciados.

La recursividad no implica necesariamente mayor eficiencia; su desempeño depende directamente de la estructura del algoritmo y de si se optimizan o no los subproblemas repetidos.

CONCLUSIONES

A partir del análisis experimental realizado, se comprobó que la complejidad teórica Big-O coincide con el comportamiento observado en las mediciones empíricas.

Los algoritmos factorial y búsqueda lineal mostraron un crecimiento lineal O(n), manteniendo tiempos proporcionales al tamaño de entrada.

El algoritmo de Fibonacci evidenció la mayor diferencia entre implementaciones, donde la versión recursiva sin optimización presentó un crecimiento exponencial O(2^n), confirmando que la recursividad puede generar costos computacionales elevados cuando no se aplican técnicas como memoización.

En el caso del ordenamiento burbuja, ambas versiones mostraron un crecimiento cuadrático $O(n^2)$, lo cual se refleja en el aumento considerable de los tiempos conforme crece el tamaño del arreglo.

Se concluye que la elección entre enfoque iterativo y recursivo no debe basarse únicamente en claridad del código, sino también en el impacto que tiene en la eficiencia temporal y en el consumo de memoria.

Este proyecto permitió validar de manera práctica los conceptos teóricos de complejidad temporal y reforzar la importancia del análisis algorítmico en el desarrollo de software eficiente.