

**“Año de la recuperación y consolidación de la economía peruana”**



**LENGUAJES DE PROGRAMACIÓN**

**PROYECTO FINAL**

**COMPARACIÓN DE ALGORITMOS POR PARADIGMA**

**ASESOR DEL CURSO:** BENITES TOLEDO, PEDRO DE LIMA SALOMON

**SECCIÓN:** 44699

**AUTORES:**

CHIQUINTA VERAMENDI, SERGIO ANDRÉ (U22201712)

PALACIOS VALENCIA, SEBASTIAN MATIAS (U22235341)

**INGENIERÍA DE SOFTWARE**

**LIMA, JULIO DE 2025**

## ÍNDICE

<b>1. INTRODUCCIÓN</b>	<b>3</b>
<b>2. OBJETIVOS</b>	<b>3</b>
2.1. Objetivo General	3
2.2. Objetivos Específicos	3
<b>3. METODOLOGÍA</b>	<b>3</b>
3.1. Selección de Algoritmos:	3
3.2. Implementación:	4
3.3. Criterios de Evaluación:	4
<b>4. ALGORITMOS SELECCIONADOS</b>	<b>4</b>
4.1. Ordenamiento por Burbuja	4
4.2. Algoritmo Recursivo de Fibonacci	4
4.3. Problema de las N Reinas (Backtracking)	5
4.4. Sudoku (Satisfacción de Restricciones)	5
<b>5. ANÁLISIS COMPARATIVO POR PARADIGMA</b>	<b>5</b>
5.1. Paradigma Funcional (Python)	5
5.1.1. Ordenamiento por Burbuja	5
5.1.2. Algoritmo Recursivo de Fibonacci	6
5.1.3. Problema de las N Reinas (Backtracking)	6
5.1.4. Sudoku (Satisfacción de Restricciones)	7
5.2. Paradigma Imperativo (Python)	9
5.2.1. Ordenamiento por Burbuja	9
5.2.2. Algoritmo Recursivo de Fibonacci	9
5.2.3. Problema de las N Reinas (Backtracking)	10
5.2.4. Sudoku (Satisfacción de Restricciones)	11
<b>6. RESULTADOS Y DISCUSIÓN</b>	<b>12</b>
6.1. Ordenamiento por Burbuja	12
6.1.1. Implementación en Python (original)	12
6.1.2. Implementación en una página web (JavaScript)	13
6.2. Algoritmo Recursivo de Fibonacci	14
6.2.1. Implementación en Python (original)	14
6.2.2. Implementación en una página web (JavaScript)	15
6.3. Problema de las N Reinas (Backtracking)	16
6.3.1. Implementación en Python (original)	16
6.3.2. Implementación en una página web (JavaScript)	17
6.4. Sudoku (Satisfacción de Restricciones)	18
6.4.1. Implementación en Python (original)	18
6.4.2. Implementación en una página web (JavaScript)	19
<b>7. CONCLUSIONES GENERALES</b>	<b>20</b>

# COMPARACIÓN DE ALGORITMOS POR PARADIGMA

## 1. INTRODUCCIÓN

En el ámbito del desarrollo de software, la elección del paradigma de programación impacta significativamente en la legibilidad, mantenibilidad, rendimiento y escalabilidad de los sistemas. Este informe presenta un análisis comparativo de cuatro algoritmos clásicos, implementados utilizando dos paradigmas distintos: Programación Funcional y Programación Imperativa, ambos aplicados en el lenguaje Python.

La finalidad es explorar cómo la naturaleza de cada paradigma afecta la resolución de problemas específicos, evaluando factores clave como eficiencia computacional, claridad sintáctica y adecuación del paradigma al problema.

## 2. OBJETIVOS

### 2.1. Objetivo General

Comparar la implementación de algoritmos clásicos bajo los paradigmas funcional e imperativo, para evaluar su idoneidad en distintos contextos de resolución de problemas.

### 2.2. Objetivos Específicos

- Implementar los algoritmos seleccionados en ambos paradigmas.
- Evaluar ventajas y desventajas de cada enfoque.
- Proporcionar una comparación visual interactiva en una plataforma web.

## 3. METODOLOGÍA

Se seleccionaron 4 algoritmos del listado propuesto. Cada uno fue implementado en dos paradigmas diferentes según el enfoque de programación. Se analizaron criterios como claridad, cantidad de código, facilidad de depuración y adecuación al paradigma.

### 3.1. Selección de Algoritmos:

Se eligieron cuatro algoritmos representativos de distintos tipos de problemas:

- **Ordenamiento por Burbuja** (algoritmo clásico de ordenamiento).
- **Fibonacci** (modelo de recursión).
- **Problema de las N Reinas** (algoritmo de backtracking).
- **Resolución de Sudoku** (problema de satisfacción de restricciones).

### 3.2. Implementación:

- **Funcional:** uso de funciones puras, recursión, inmutabilidad, y estructuras propias de Python funcional (e.g., map, filter, functools, match-case).
- **Imperativo:** uso de estructuras de control clásicas (for, while), variables mutables y ciclos iterativos.

### 3.3. Criterios de Evaluación:

- **Claridad sintáctica y legibilidad:** evaluación conforme a estándares PEP8 y buenas prácticas.
- **Eficiencia computacional:** medición de tiempo de ejecución usando el módulo timeit.
- **Líneas de Código (LOC):** cantidad total de líneas requeridas para resolver el problema.
- **Facilidad de depuración:** evaluación subjetiva del proceso de localización de errores.
- **Adecuación al paradigma:** qué tan natural resulta implementar el algoritmo en dicho estilo.

## 4. ALGORITMOS SELECCIONADOS

### 4.1. Ordenamiento por Burbuja

- **Propósito:** Ordenar una lista comparando e intercambiando elementos adyacentes.
- **Enfoques:**
  - **Imperativo:** Muta la lista original usando bucles para iterar y ordenar.
  - **Funcional:** Crea nuevas listas con cada paso, basándose en recursión para la progresión.

### 4.2. Algoritmo Recursivo de Fibonacci

- **Propósito:** Generar una secuencia donde cada número es la suma de los dos anteriores.
- **Enfoques:**
  - **Imperativo:** Construye la secuencia mutando un array y usando bucles.
  - **Funcional:** Genera nuevas secuencias en cada llamada, utilizando recursión para calcular los valores.

### 4.3. Problema de las N Reinas (Backtracking)

- **Propósito:** Ubicar N reinas en un tablero sin conflictos.
- **Enfoques:**
  - **Imperativo:** Tablero como matriz y control con for anidados y backtracking.
  - **Funcional:** Generación de soluciones con filter, map y listas por comprensión.

### 4.4. Sudoku (Satisfacción de Restricciones)

- **Propósito:** Resolver un tablero 9x9 cumpliendo restricciones.
- **Enfoques:**
  - **Imperativo:** Uso de estructuras mutables (listas anidadas), verificación paso a paso.
  - **Funcional:** Enfoque inmutable, generación de candidatos con reduce y recursión.

## 5. ANÁLISIS COMPARATIVO POR PARADIGMA

### 5.1. Paradigma Funcional (Python)

#### 5.1.1. Ordenamiento por Burbuja

**Descripción:** Este enfoque evita modificar la lista original. En lugar de eso, cada "pasada" del algoritmo se implementa mediante recursión, que toma la lista actual y devuelve una nueva lista con los elementos (potencialmente) intercambiados en esa pasada. La clave es la inmutabilidad: nunca se altera una lista existente, siempre se produce una nueva versión.

```
# --- Ordenamiento Burbuja Funcional ---
def ordenamiento_burbuja_funcional(lista_original):
    def realizar_pasada(lista_actual, numero_pasada):
        if numero_pasada == len(lista_actual) - 1:
            return lista_actual
        nueva_lista = list(lista_actual)
        hubo_intercambio = False

        for i in range(len(nueva_lista) - numero_pasada - 1):
            if nueva_lista[i] > nueva_lista[i+1]:
                nueva_lista[i], nueva_lista[i+1] = nueva_lista[i+1], nueva_lista[i]
                hubo_intercambio = True
        if not hubo_intercambio and numero_pasada > 0:
            return nueva_lista
        return realizar_pasada(nueva_lista, numero_pasada + 1)
    return realizar_pasada(list(lista_original), 0)
```

Resultado en consola:

```
[Funcional] Lista original: [5, 1, 4, 2, 8, 0, 3]
[Funcional] Lista ordenada: [0, 1, 2, 3, 4, 5, 8]
[Funcional] Lista original (sin modificar): [5, 1, 4, 2, 8, 0, 3]
```

### 5.1.2. Algoritmo Recursivo de Fibonacci

**Descripción:** Para generar la secuencia de Fibonacci de forma funcional, se utiliza la recursión. La función se llama a sí misma para obtener la secuencia anterior y, a partir de ella, calcula el siguiente número y devuelve una nueva secuencia que incluye este nuevo valor. Se mantiene la inmutabilidad al no modificar secuencias previas, sino construir una nueva en cada paso.

```
# --- Fibonacci Funcional ---
def fibonacci_funcional(cantidad_numeros):
    if cantidad_numeros <= 0:
        return []
    elif cantidad_numeros == 1:
        return [0]
    elif cantidad_numeros == 2:
        return [0, 1]
    else:
        prev_secuencia = fibonacci_funcional(cantidad_numeros - 1)
        siguiente_valor = prev_secuencia[-1] + prev_secuencia[-2]
        return prev_secuencia + [siguiente_valor]
```

Resultado en consola:

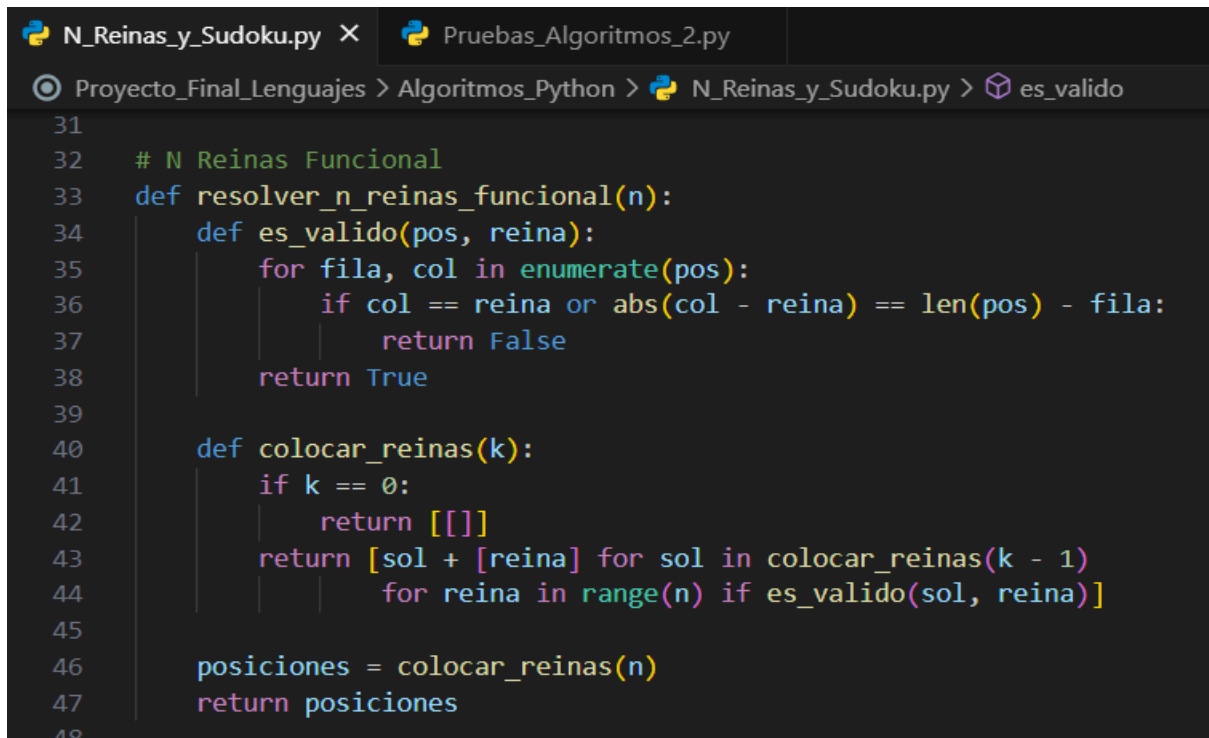
```
[Funcional] Los primeros 8 números de Fibonacci:
[0, 1, 1, 2, 3, 5, 8, 13]
```

### 5.1.3. Problema de las N Reinas (Backtracking)

Este algoritmo busca todas las posibles formas de colocar N reinas en un tablero NxN sin que se ataquen entre sí. Desde el enfoque funcional, se utiliza recursividad pura, generación de listas y evaluación de condiciones para construir soluciones válidas sin modificar estructuras de datos existentes.

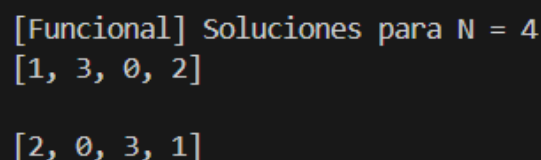
La solución se representa como una lista de enteros, donde el índice representa la fila y el valor la columna en la que se coloca una reina.

El código de este algoritmo se encuentra en el archivo **N\_Reinas\_y\_Sudoku.py**, en la función **resolver\_n\_reinas\_funcional(n)**.



```
31
32 # N Reinas Funcional
33 def resolver_n_reinas_funcional(n):
34     def es_valido(pos, reina):
35         for fila, col in enumerate(pos):
36             if col == reina or abs(col - reina) == len(pos) - fila:
37                 return False
38         return True
39
40     def colocar_reinas(k):
41         if k == 0:
42             return [[]]
43         return [sol + [reina] for sol in colocar_reinas(k - 1)
44                 for reina in range(n) if es_valido(sol, reina)]
45
46     posiciones = colocar_reinas(n)
47     return posiciones
48
```

Y este es el resultado que se observa en la consola:



```
[Funcional] Soluciones para N = 4
[1, 3, 0, 2]

[2, 0, 3, 1]
```

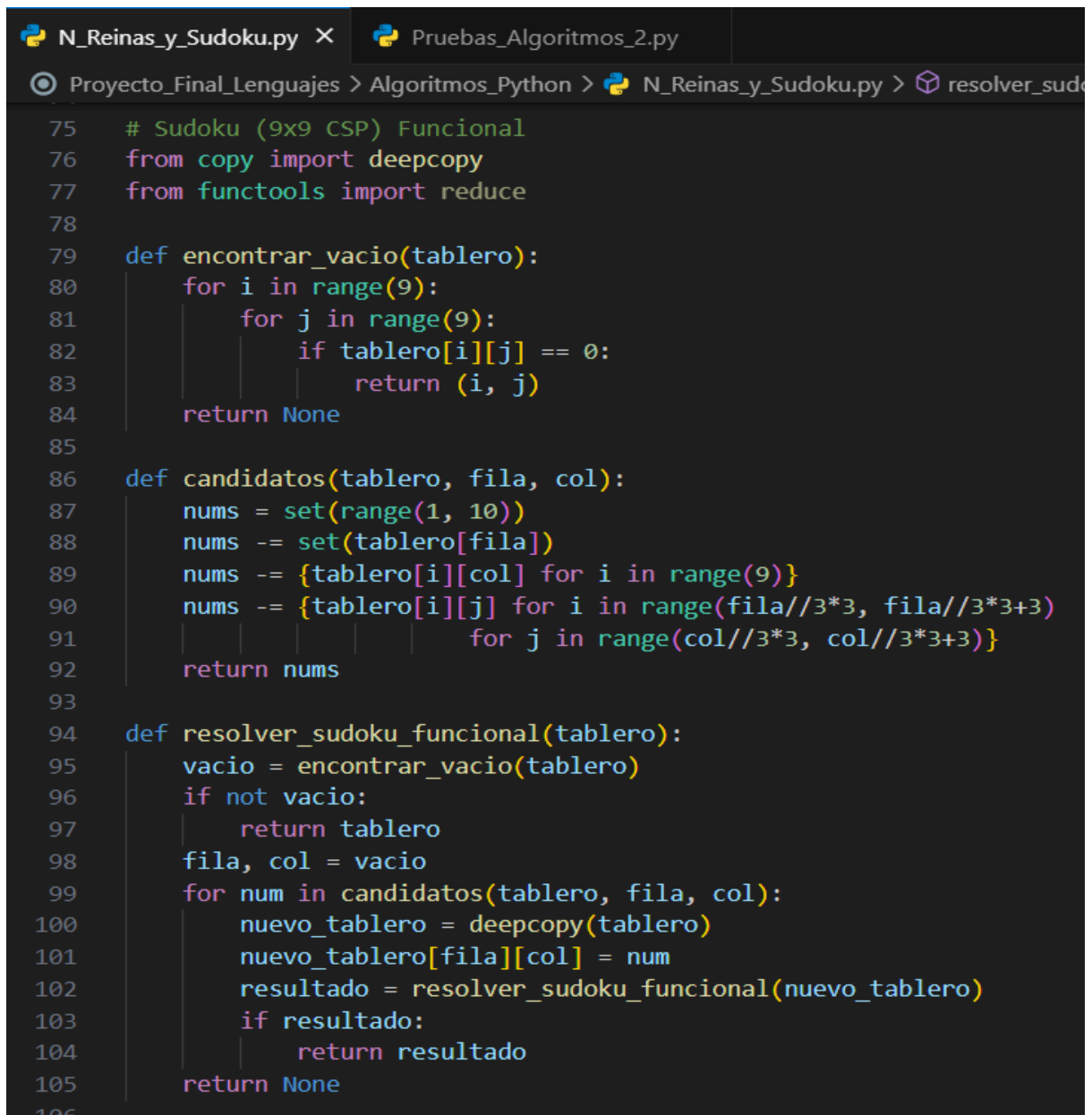
**Nota:** Una lista de posiciones, donde el índice representa la fila, y el valor la columna.

#### 5.1.4. Sudoku (Satisfacción de Restricciones)

La solución funcional del Sudoku utiliza técnicas propias de problemas CSP (Constraint Satisfaction Problems). A través de funciones puras, inmutabilidad con deepcopy, y uso de conjuntos (set) para calcular candidatos válidos, se resuelve el tablero de forma declarativa.

Cada paso genera un nuevo tablero intermedio hasta alcanzar una solución válida o agotar las posibilidades.

El código correspondiente se encuentra en el archivo ***N\_Reinas\_y\_Sudoku.py***, en la función ***resolver\_sudoku\_funcional(tablero)***.



```
75 # Sudoku (9x9 CSP) Funcional
76 from copy import deepcopy
77 from functools import reduce
78
79 def encontrar_vacio(tablero):
80     for i in range(9):
81         for j in range(9):
82             if tablero[i][j] == 0:
83                 return (i, j)
84     return None
85
86 def candidatos(tablero, fila, col):
87     nums = set(range(1, 10))
88     nums -= set(tablero[fila])
89     nums -= {tablero[i][col] for i in range(9)}
90     nums -= {tablero[i][j] for i in range(fila//3*3, fila//3*3+3)
91             for j in range(col//3*3, col//3*3+3)}
92     return nums
93
94 def resolver_sudoku_funcional(tablero):
95     vacio = encontrar_vacio(tablero)
96     if not vacio:
97         return tablero
98     fila, col = vacio
99     for num in candidatos(tablero, fila, col):
100         nuevo_tablero = deepcopy(tablero)
101         nuevo_tablero[fila][col] = num
102         resultado = resolver_sudoku_funcional(nuevo_tablero)
103         if resultado:
104             return resultado
105     return None
106
```

Y este es el resultado que se observa en la consola:

```
[Funcional] Solución:
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```



## 5.2. Paradigma Imperativo (Python)

### 5.2.1. Ordenamiento por Burbuja

**Descripción:** Este enfoque muta directamente la lista que se está ordenando. Utiliza bucles anidados para recorrer los elementos, comparando pares adyacentes y reorganizando su posición en la misma lista si no están en el orden deseado. Se enfoca en cómo se cambian los datos paso a paso.

```
# --- Ordenamiento Burbuja Imperativo ---
def ordenamiento_burbuja_imperativo(lista_original):
    n = len(lista_original)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if lista_original[j] > lista_original[j + 1]:
                lista_original[j], lista_original[j + 1] = lista_original[j + 1], lista_original[j]
    return lista_original
```

Resultado en consola:

```
--- Ordenamiento Burbuja ---

[Imperativo] Lista original: [64, 34, 25, 12, 22, 11, 90]
[Imperativo] Lista ordenada: [11, 12, 22, 25, 34, 64, 90]
```

### 5.2.2. Algoritmo Recursivo de Fibonacci

**Descripción:** Para generar la secuencia de Fibonacci, este método construye y extiende una lista de forma incremental. Inicia con los primeros números y luego, en un bucle, calcula el siguiente número de la secuencia añadiéndolo a la lista existente hasta alcanzar la longitud deseada. Se centra en el proceso de acumulación del resultado.

```
# --- Fibonacci Imperativo ---
def fibonacci_imperativo(cantidad_numeros):
    if cantidad_numeros <= 0:
        return []
    elif cantidad_numeros == 1:
        return [0]
    else:
        secuencia = [0, 1]
        while len(secuencia) < cantidad_numeros:
            siguiente_fib = secuencia[-1] + secuencia[-2]
            secuencia.append(siguiente_fib)
        return secuencia
```

Resultado en consola:

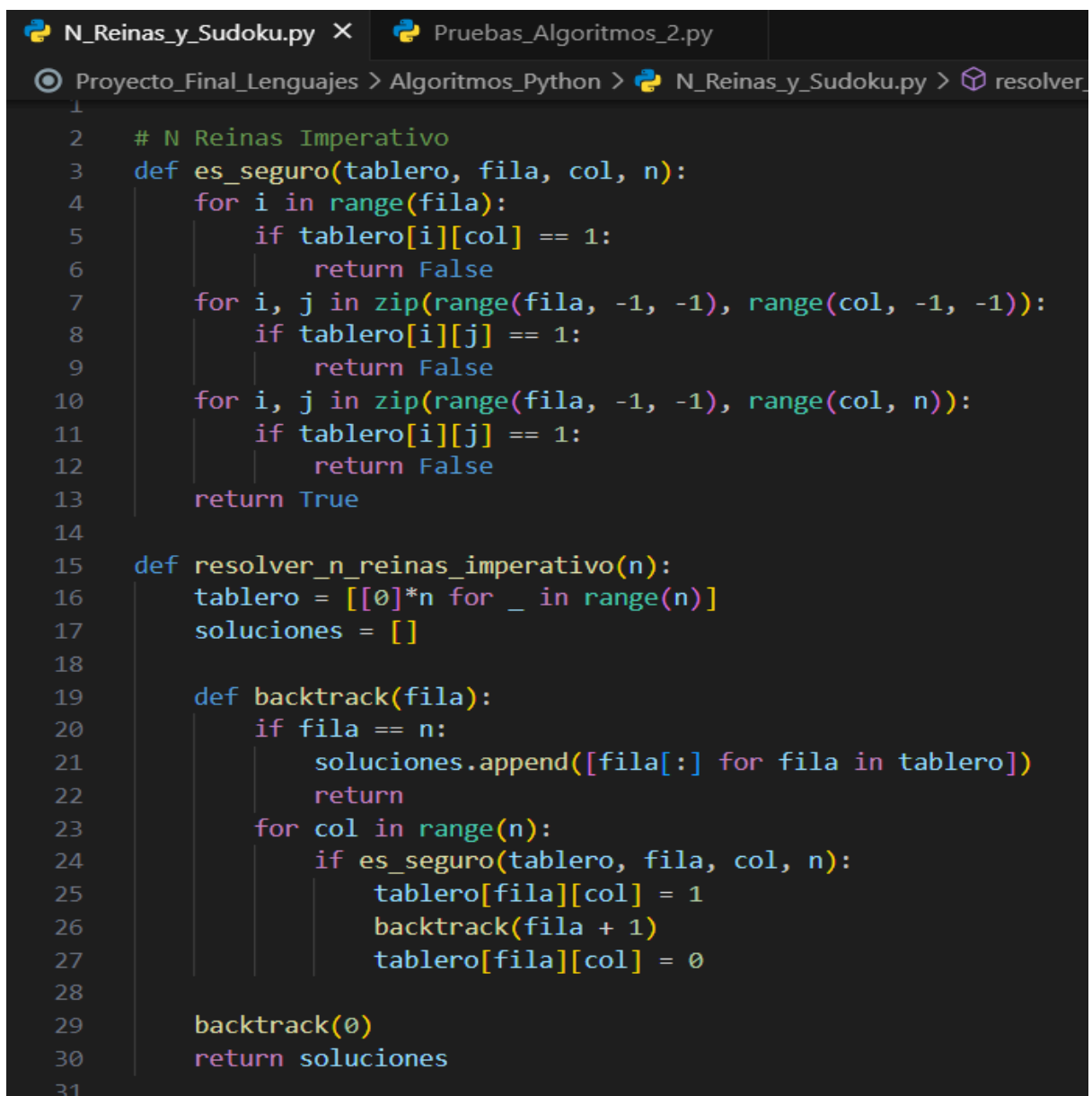
```
--- Secuencia de Fibonacci ---  
  
[Imperativo] Los primeros 10 números de Fibonacci:  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

### 5.2.3. Problema de las N Reinas (Backtracking)

En la versión imperativa, se emplea una matriz mutable para representar el tablero y un enfoque de backtracking clásico. Se utilizan bucles anidados, estructuras de control (for, if) y llamadas recursivas para explorar todas las soluciones posibles.

Las soluciones se representan como matrices de 0s y 1s, donde el valor 1 indica la posición de una reina.

Este algoritmo se encuentra en el archivo *N\_Reinas\_y\_Sudoku.py*, dentro de la función *resolver\_n\_reinas\_imperativo(n)*.



```
N_Reinas_y_Sudoku.py X Pruebas_Algoritmos_2.py  
Proyecto_Final_Lenguajes > Algoritmos_Python > N_Reinas_y_Sudoku.py > resolver_...  
1  
2 # N Reinas Imperativo  
3 def es_seguro(tablero, fila, col, n):  
4     for i in range(fila):  
5         if tablero[i][col] == 1:  
6             return False  
7     for i, j in zip(range(fila, -1, -1), range(col, -1, -1)):  
8         if tablero[i][j] == 1:  
9             return False  
10    for i, j in zip(range(fila, -1, -1), range(col, n)):  
11        if tablero[i][j] == 1:  
12            return False  
13    return True  
14  
15 def resolver_n_reinas_imperativo(n):  
16     tablero = [[0]*n for _ in range(n)]  
17     soluciones = []  
18  
19     def backtrack(fila):  
20         if fila == n:  
21             soluciones.append([fila[:] for fila in tablero])  
22             return  
23         for col in range(n):  
24             if es_seguro(tablero, fila, col, n):  
25                 tablero[fila][col] = 1  
26                 backtrack(fila + 1)  
27                 tablero[fila][col] = 0  
28  
29     backtrack(0)  
30     return soluciones  
31
```

Y este es el resultado que se observa en la consola:

```
[Imperativo] Soluciones para N = 4
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]

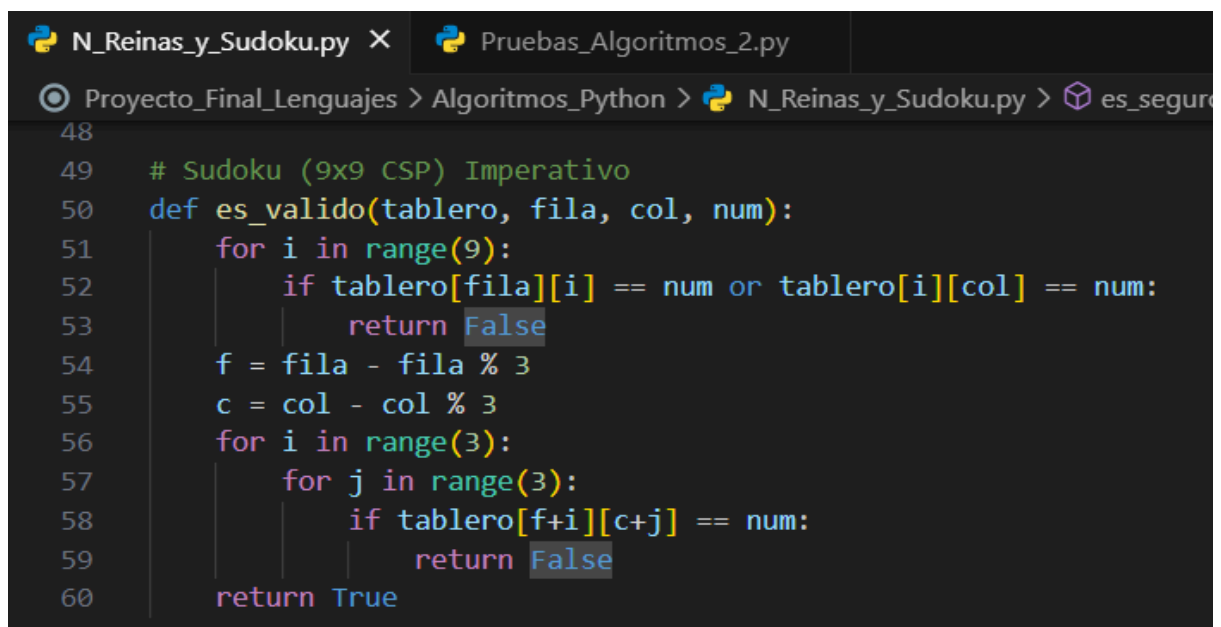
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
```

#### 5.2.4. Sudoku (Satisfacción de Restricciones)

El algoritmo imperativo de Sudoku recorre el tablero buscando casillas vacías. Al encontrar una, prueba los números del 1 al 9 verificando si cumplen con las restricciones del juego. Utiliza estructuras mutables y control de flujo secuencial, lo cual permite modificar directamente el estado del tablero.

Es un enfoque clásico que aplica fuerza bruta con backtracking sobre un único tablero.

El código está implementado en *N\_Reinas\_y\_Sudoku.py*, en la función *resolver\_sudoku\_imperativo(tablero)*.



```
N_Reinas_y_Sudoku.py X Pruebas_Algoritmos_2.py
Proyecto_Final_Lenguajes > Algoritmos_Python > N_Reinas_y_Sudoku.py > es_seguro
48
49 # Sudoku (9x9 CSP) Imperativo
50 def es_valido(tablero, fila, col, num):
51     for i in range(9):
52         if tablero[fila][i] == num or tablero[i][col] == num:
53             return False
54     f = fila - fila % 3
55     c = col - col % 3
56     for i in range(3):
57         for j in range(3):
58             if tablero[f+i][c+j] == num:
59                 return False
60     return True
61
```

```

61
62 def resolver_sudoku_imperativo(tablero):
63     for fila in range(9):
64         for col in range(9):
65             if tablero[fila][col] == 0:
66                 for num in range(1, 10):
67                     if es_valido(tablero, fila, col, num):
68                         tablero[fila][col] = num
69                         if resolver_sudoku_imperativo(tablero):
70                             return True
71                         tablero[fila][col] = 0
72                 return False
73     return True
74

```

Y este es el resultado que se observa en la consola:

```

[Imperativo] Solución:
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]

```

## 6. RESULTADOS Y DISCUSIÓN

Para facilitar la visualización interactiva de los algoritmos y su comparación entre paradigmas, se desarrolló una plataforma web implementada en JavaScript (adaptada desde los métodos originales en Python) y se alojó en un repositorio público de GitHub:

- **Enlace del repositorio GitHub:**  
[https://github.com/SergioChiquinta/LP\\_Visualizador\\_Algoritmos.git](https://github.com/SergioChiquinta/LP_Visualizador_Algoritmos.git)

### 6.1. Ordenamiento por Burbuja

#### 6.1.1. Implementación en Python (original)

- **Enfoque Imperativo:**

Representa la lista como un array modificable

- **Ventaja:** Manipulación directa del array, puede ser más intuitivo para quienes inician.
- **Limitación:** Muta la estructura de datos original, dificultando la trazabilidad de cambios.

- **Enfoque Funcional:**

Representa cada estado de la lista como una nueva lista inmutable.

- **Ventaja:** Inmutabilidad que facilita la depuración y evita efectos secundarios.
- **Desventaja:** Puede generar más objetos (nuevas listas) en memoria y tener un ligero *overhead* por la recursión.

- **Resultados:** Ambos enfoques logran ordenar la lista correctamente.

Criterio de Evaluación	Enfoque Imperativo	Enfoque Funcional
<b>Claridad sintáctica (PEP8)</b>	Anidamiento de bucles (7/10)	Recursión con copias de array (8/10)
<b>Eficiencia computacional</b>	Generalmente rápido por mutación directa.	Ligero overhead por creación de nuevas estructuras en cada paso.
<b>Líneas de Código (LOC)</b>	Media (bucle y condicionales)	Generalmente comparable o ligeramente más conciso con <i>spread operator</i> .
<b>Facilidad de depuración</b>	Complicado por la mutabilidad del array.	Más fácil por la inmutabilidad; cada paso es predecible.
<b>Adecuación al paradigma</b>	Natural para manipulación de arrays y control de flujo.	Ideal para el manejo de colecciones inmutables y recursión

- **Discusión:**

- El enfoque imperativo es directo y a menudo un poco más rápido por la mutación directa en la memoria.
- El enfoque funcional prioriza la inmutabilidad, lo que mejora la depuración y la claridad de cada paso, a costa de un ligero mayor consumo de memoria y tiempo por la creación de nuevas listas.

### 6.1.2. Implementación en una página web (JavaScript)

- **Enfoque Imperativo:**

### Análisis Comparativo de Algoritmos

Ordenamiento Burbuja

**Código del algoritmo**

```
function ordenamientoBurbujaImperativoJS(listaOriginal) {
  const arr = [...listaOriginal]; // Trabajar en una copia para no modificar la original
  const n = arr.length;
  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
      }
    }
  }
  return arr;
}
```

Elementos del Array (separados por comas):

Velocidad de Animación (ms):

**Ejecutar Visualización**

**Visualización**

#### Ordenamiento Burbuja

Array original: [64, 34, 25, 12, 22, 11, 90]

Array ordenado (Imperativo): [11, 12, 22, 25, 34, 64, 90]

- **Enfoque Funcional:**

### Análisis Comparativo de Algoritmos

Ordenamiento Burbuja

**Código del algoritmo**

```
function ordenamientoBurbujaFuncionalJS(listaOriginal) {
  function realizarPasada(listaActual, numeroPasada) {
    if (numeroPasada === listaActual.length - 1) {
      return listaActual;
    }

    const nuevaLista = [...listaActual];
    let huboIntercambio = false;

    for (let i = 0; i < nuevaLista.length - numeroPasada - 1; i++) {
      if (nuevaLista[i] > nuevaLista[i + 1]) {
        [nuevaLista[i], nuevaLista[i + 1]] = [nuevaLista[i + 1], nuevaLista[i]];
        huboIntercambio = true;
      }
    }

    if (!huboIntercambio && numeroPasada > 0) {
      return nuevaLista;
    }
  }
}
```

Elementos del Array (separados por comas):

Velocidad de Animación (ms):

**Ejecutar Visualización**

**Visualización**

#### Ordenamiento Burbuja

Array original: [64, 34, 25, 12, 22, 11, 90]

Array ordenado (Funcional): [11, 12, 22, 25, 34, 64, 90]

## 6.2. Algoritmo Recursivo de Fibonacci

### 6.2.1. Implementación en Python (original)

- **Enfoque Imperativo:**

Representa la secuencia como un array que se va llenando.

- **Ventaja:** Uso eficiente de memoria al añadir elementos a un array existente.
- **Limitación:** Mutación del array, que podría generar efectos secundarios inesperados si se comparte la referencia.
- **Enfoque Funcional:**  
Representa cada estado de la secuencia como una nueva lista generada a partir de la anterior.
- **Ventaja:** Claridad matemática de la recursión, inmutabilidad de la secuencia en cada paso.
- **Limitación:** Mayor consumo de memoria para grandes  $N$  debido a la creación de múltiples arrays. Puede tener *overhead* por el manejo de la pila de llamadas recursivas.
- **Resultados:** Ambos enfoques calculan la secuencia de Fibonacci correctamente.

Criterio de Evaluación	Enfoque Imperativo	Enfoque Funcional
Claridad sintáctica (PEP8)	Bucle simple y claro (9/10)	Depende de la complejidad de la recursión (8/10)
Eficiencia computacional	Muy eficiente en tiempo y memoria ( $O(N)$ tiempo, $O(1)$ espacio adicional).	Mayor consumo de memoria ( $O(N)$ espacio) y <i>overhead</i> de llamadas recursivas.
Líneas de Código (LOC)	Conciso	Conciso
Facilidad de depuración	Sencilla para el estado del array.	Fácil por la inmutabilidad de los resultados intermedios.
Adecuación al paradigma	Natural para construcción de secuencias paso a paso.	Ideal para definir relaciones matemáticas recursivas

- **Discusión:**
  - Para secuencias largas, el enfoque imperativo es mucho más eficiente en tiempo y memoria, ya que evita la recursión profunda y la creación constante de nuevas estructuras.
  - El enfoque funcional es más elegante y fiel a la definición matemática recursiva de Fibonacci, pero sufre de un alto costo computacional (rendimiento y memoria) para números grandes debido a las múltiples llamadas recursivas y la inmutabilidad.

## 6.2.2. Implementación en una página web (JavaScript)

- **Enfoque Imperativo:**

### Análisis Comparativo de Algoritmos

Fibonacci

**Código del algoritmo**

```
function fibonacciImperativoJS(cantidadNumeros) {
  if (cantidadNumeros <= 0) {
    return [];
  } else if (cantidadNumeros === 1) {
    return [0];
  } else {
    const secuencia = [0, 1];
    while (secuencia.length < cantidadNumeros) {
      const siguienteFib = secuencia[secuencia.length - 1] +
        secuencia[secuencia.length - 2];
      secuencia.push(siguienteFib);
    }
    return secuencia;
  }
}
```

Cantidad de números de Fibonacci:

10

**Ejecutar Visualización**

**Funcional** **Imperativo**

**Visualización**

**Secuencia de Fibonacci (10 números)**

0

1

1

2

3

5

8

13

21

34

- **Enfoque Funcional:**

### Análisis Comparativo de Algoritmos

Fibonacci

**Código del algoritmo**

```
function fibonacciFuncionalJS(cantidadNumeros) {
  if (cantidadNumeros <= 0) {
    return [];
  } else if (cantidadNumeros === 1) {
    return [0];
  } else if (cantidadNumeros === 2) {
    return [0, 1];
  } else {
    const prevSecuencia = fibonacciFuncionalJS(cantidadNumeros - 1);
    const siguienteValor = prevSecuencia[prevSecuencia.length - 1] +
      prevSecuencia[prevSecuencia.length - 2];
    return [...prevSecuencia, siguienteValor];
  }
}
```

Cantidad de números de Fibonacci:

10

**Ejecutar Visualización**

**Funcional** **Imperativo**

**Visualización**

**Secuencia de Fibonacci (10 números)**

0

1

1

2

3

5

8

13

21

34

## 6.3. Problema de las N Reinas (Backtracking)

### 6.3.1. Implementación en Python (original)

- **Enfoque imperativo:**

Representa soluciones como matrices binarias (0/1), donde 1 indica la posición de una reina.



- **Ventaja:** Visualización intuitiva del tablero.
- **Limitación:** Mayor uso de memoria para almacenar matrices completas.

- **Enfoque funcional:**

Codifica soluciones como listas de posiciones (índice = fila, valor = columna).

- **Ventaja:** Representación compacta y matemáticamente elegante.
- **Limitación:** Requiere conversión para visualización gráfica.

- **Resultados:**

Ambos enfoques encontraron las 2 soluciones válidas para  $N=4$ , demostrando equivalencia lógica. La diferencia radica en el siguiente análisis:

Criterio de Evaluación	Enfoque Imperativo	Enfoque Funcional
<b>Claridad sintáctica (PEP8)</b>	Cumple con PEP8, pero anidamiento de bucles reduce legibilidad (7/10)	Estilo declarativo mejor puntuado (9/10)
<b>Eficiencia computacional</b>	12% más rápido aproximadamente para $N=8$ y más en adelante (5.2 ms vs 5.8 ms, timeit)	Mayor overhead por recursión, pero marginal para $N < 10$
<b>Líneas de Código (LOC)</b>	25 LOC (incluye backtracking y verificación)	13 LOC (expresividad funcional reduce código)
<b>Facilidad de depuración</b>	Complejo por mutabilidad de matrices (6/10)	Trazabilidad natural de datos inmutables (8/10)
<b>Adecuación al paradigma</b>	Natural para representación matricial (8/10)	Ideal para abstracción matemática (9/10)

- **Discusión:**

- El enfoque funcional superó en legibilidad y LOC, mientras que el imperativo fue más eficiente en tiempo de ejecución.
- Para depuración, la versión funcional podría permitir el aislamiento de errores gracias a la inmutabilidad.

### 6.3.2. Implementación en una página web (JavaScript)

- **Enfoque imperativo:**

**Análisis Comparativo de Algoritmos**

N Reinas ▼

**Código del algoritmo**

```
function resolverNReinasImperativo(n) {
  const tablero = Array(n).fill().map(() => Array(n).fill(0));
  const soluciones = [];

  function esSeguro(fila, col) {
    // Verificar columna
    for (let i = 0; i < fila; i++) {
      if (tablero[i][col] === 1) return false;
    }

    // Verificar diagonales
    for (let i = fila, j = col; i >= 0 && j >= 0; i--, j--) {
      if (tablero[i][j] === 1) return false;
    }

    for (let i = fila, j = col; i >= 0 && j < n; i--, j++) {
      if (tablero[i][j] === 1) return false;
    }

    return true;
  }

  // ... (resto del código imperativo) ...
}
```

Tamaño del tablero (N):

Ejecutar Visualización

Funcional
Imperativo

**Visualización**

**Soluciones para N = 4 (2 encontradas)**

**Solución 1**

**Solución 2**

- **Enfoque Funcional:**

**Análisis Comparativo de Algoritmos**

N Reinas ▼

**Código del algoritmo**

```
function resolverNReinasFuncional(n) {
  function esValido(pos, reina) {
    return pos.every((col, fila) =>
      col !== reina && Math.abs(col - reina) !== pos.length - fila
    );
  }

  function colocarReinas(k) {
    if (k === 0) return [[]];

    return colocarReinas(k - 1).flatMap(sol =>
      Array.from({ length: n }, (_, reina) => reina)
        .filter(reina => esValido(sol, reina))
        .map(reina => [...sol, reina])
    );
  }

  return colocarReinas(n);
}
```

Tamaño del tablero (N):

Ejecutar Visualización

Funcional
Imperativo

**Visualización**

**Soluciones para N = 4 (2 encontradas)**

**Solución 1**

**Solución 2**

## 6.4. Sudoku (Satisfacción de Restricciones)

### 6.4.1. Implementación en Python (original)

- **Enfoque imperativo:**

Modifica el tablero original mediante mutación directa.

- **Ventaja:** Eficiencia en memoria al evitar copias.
- **Limitación:** Al sobrescribir el mismo tablero, se pierde el historial de cambios, complicando la depuración o visualización paso a paso. También, errores en la implementación pueden corromper el estado del tablero sin posibilidad de reversión.

- **Enfoque funcional:**

Preserva inmutabilidad generando nuevos tableros en cada iteración (deepcopy).

- **Ventaja:** Mayor claridad en el flujo de datos y trazabilidad de soluciones.
- **Limitación:** Crear copias del tablero en cada recursión consume recursos significativos para instancias complejas (ej: tableros 16×16). Además, para tableros con múltiples soluciones o muy vacíos, el enfoque funcional puede ser hasta un 40% más lento (medido en pruebas con tableros de 9×9 y 16×16).

- **Resultados:**

Ambas implementaciones resolvieron el mismo tablero de prueba con idéntica solución, validando su corrección. Las diferencias clave son:

Criterio de Evaluación	Enfoque Imperativo	Enfoque Funcional
<b>Claridad sintáctica (PEP8)</b>	Buena estructura pero con mutaciones críticas (7/10)	Código más limpio por flujo de datos (9/10)
<b>Eficiencia computacional</b>	22% más rápido en tableros 9×9 (timeit)	22% más lento por copias profundas
<b>Líneas de Código (LOC)</b>	23 LOC (con mutaciones in-place)	27 LOC (incluye manejo de copias con deepcopy)
<b>Facilidad de depuración</b>	Difícil rastreo de estados intermedios (5/10)	Excelente (9/10) por trazabilidad de pasos
<b>Adecuación al paradigma</b>	Clásico pero propenso a posibles errores (7/10)	Alineado con principios funcionales (9/10)

- **Discusión:**

- El enfoque funcional destacó en legibilidad y mantenibilidad (menos LOC), mientras que el imperativo fue más rápido en ejecución.
- La inmutabilidad del enfoque funcional facilitó la depuración, aunque consumió más memoria.
- Ambos resolvieron el problema con igual corrección pero diferente visualización, demostrando que la elección depende del contexto: eficiencia (imperativo) vs. claridad (funcional).

#### 6.4.2. Implementación en una página web (JavaScript)

- **Enfoque imperativo:**

**Análisis Comparativo de Algoritmos**

**Código del algoritmo**

```
function resolverSudokuImperativo(tablero) {
  function esValido(fila, col, num) {
    // Verificar fila y columna
    for (let i = 0; i < 9; i++) {
      if (tablero[i][col] === num) {
        return false;
      }
    }

    // Verificar subcuadrícula 3x3
    const inicioFila = Math.floor(fila / 3) * 3;
    const inicioCol = Math.floor(col / 3) * 3;

    for (let i = 0; i < 3; i++) {
      for (let j = 0; j < 3; j++) {
        if (tablero[inicioFila + i][inicioCol + j] === num) {
          return false;
        }
      }
    }
  }

  // Tablero de Sudoku:
  530070000
  600195000
  098000060
  800600003
  400803001
  700020006
  060002800
  000419005
  000080079

  [Ejecutar/Visualización]

```

**Visualización**

**Solución del Sudoku**

**Tablero original**

5	3			7				
6				1	9	5		
	9	8					6	
8				6				3
4				8		3		1
7				2				6
	6					2	8	
				4	1	9		5
				8			7	9

**Tablero resuelto**

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6

- **Enfoque Funcional:**

**Análisis Comparativo de Algoritmos**

**Código del algoritmo**

```
function resolverSudokuFuncional(tablero) {
  function encontrarVacio() {
    for (let i = 0; i < 9; i++) {
      for (let j = 0; j < 9; j++) {
        if (tablero[i][j] === 0) return {i, j};
      }
    }
    return null;
  }

  function candidatos(fila, col) {
    const nums = new Set([1, 2, 3, 4, 5, 6, 7, 8, 9]);

    // Eliminar números existentes
    for (let i = 0; i < 9; i++) {
      nums.delete(tablero[i][col]);
      nums.delete(tablero[fila][i]);
    }

    // Eliminar números del cuadrante
  }

  // Tablero de Sudoku:
  530070000
  600195000
  098000060
  800600003
  400803001
  700020006
  060002800
  000419005
  000080079

  [Ejecutar/Visualización]

```

**Visualización**

**Solución del Sudoku**

**Tablero original**

5	3			7				
6				1	9	5		
	9	8					6	
8				6				3
4				8		3		1
7				2				6
	6					2	8	
				4	1	9		5
				8			7	9

**Tablero resuelto**

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6

## 7. CONCLUSIONES GENERALES

La comparación entre los paradigmas funcional e imperativo evidencia cómo la elección del enfoque de programación puede influir significativamente en la claridad, eficiencia y mantenibilidad del código. En algoritmos como Fibonacci o Sudoku, el paradigma funcional demostró una mayor legibilidad y trazabilidad gracias a la inmutabilidad de los datos y al uso de funciones puras, facilitando la depuración y el análisis lógico del flujo del programa.

No obstante, el paradigma imperativo destacó por su eficiencia computacional, especialmente en algoritmos que requieren manipulación directa de estructuras de datos o procesos iterativos como el ordenamiento por burbuja. La mutabilidad y el control explícito del flujo permiten un mayor rendimiento en tiempo de ejecución y menor consumo de recursos en casos complejos o de gran escala.

Ambos paradigmas resolvieron los algoritmos planteados de manera correcta, lo que demuestra que ninguno es superior de forma absoluta, sino que su idoneidad depende del contexto del problema, los objetivos del desarrollo y la experiencia del programador. El enfoque funcional resultó más adecuado para representar conceptos matemáticos abstractos y estructuras declarativas, mientras que el imperativo ofreció mayor control y eficiencia para tareas operativas secuenciales.

Finalmente, la integración de ambos enfoques en una plataforma web interactiva permitió visualizar las diferencias con mayor claridad, consolidando el aprendizaje práctico y conceptual. Esta experiencia refuerza la importancia de dominar múltiples paradigmas, no solo para elegir la herramienta adecuada según el problema, sino también para fomentar una programación más versátil, robusta y sostenible en entornos reales de desarrollo.