# 24 días, 24 desafíos de código

Desde “mi código funciona”
hasta la solución profesional

👉 **MUESTRA DEL LIBRO - EDICIÓN MULTILENGUAJE**

**Patricia E. Miguel**

Programación Desde Cero

## 📖 VISTA PREVIA - EDICIÓN MULTILENGUAJE

---

Este vista previa muestra el contenido de la edición que incluye **Python**, **Java** y **C#** (la edición más completa del libro).

También disponible en ediciones individuales:

- Solo Python
- Solo Java
- Solo C#

Todas las ediciones contienen el mismo análisis y explicaciones.
La diferencia está únicamente en los lenguajes de implementación.

# Índice de contenidos

Tu nombre - tu@email.com

# Introducción

## Estructura del libro

Este libro contiene **24 desafíos de programación** diseñados para fortalecer la capacidad de análisis algorítmico y la lógica de resolución de problemas. Los ejercicios se ubican en un nivel principiante-intermedio, pensados para lectores que ya dominan los fundamentos de la programación: variables, tipos de datos, estructuras de control, funciones, arreglos unidimensionales y bidimensionales, y colecciones básicas (conjuntos y tablas hash).

Los desafíos pueden resolverse en el orden propuesto, que introduce conceptos progresivamente, o de manera independiente según el área que quieras reforzar. Cada ejercicio desarrolla estrategias de resolución transferibles a problemas similares: uso de tablas hash para conteos, técnicas de manipulación de arreglos, patrones de trabajo con cadenas de texto, entre otros.

## Por *qué* los ejercicios de algoritmia siguen siendo fundamentales en la era de la IA

Hoy, cualquier modelo de lenguaje genera en segundos el código que resuelve un ejercicio bien planteado. Entonces, ¿para qué dedicar tiempo a aprender a resolverlos por tu cuenta? Porque generar código y resolver problemas son habilidades completamente diferentes.

Los modelos de inteligencia artificial pueden generar código algorítmico, sí. Pero es más importante **lo que NO pueden hacer sin un humano** que sepa lo que está pidiendo:

- **Identificar** qué algoritmo se necesita cuando el problema real no se parece a ningún ejercicio.
- **Evaluar** si la solución es correcta o tiene casos especiales que fallan en producción.
- **Decidir** ante disyuntivas (¿es preferible optimizar tiempo o memoria? ¿simplicidad o rendimiento?).
- **Adaptar** el algoritmo estándar cuando las restricciones difieren del caso teórico.
- ***Debuggear*** cuando falla, sin tener que pedirle ayuda a la IA en cada línea.

Un programador que delega todo a la IA sin entender lo que está pasando es como un conductor que solo sabe manejar con piloto automático: **funciona bien hasta que algo sale mal**.

Este libro no te enseña a competir con la IA en velocidad de escritura de código. Te enseña a ser el tipo de programador que sabe qué pedirle, cómo evaluarlo, y qué hacer cuando te entrega algo que no funciona.

## Para *quién* es este libro

Este libro es para el lector que:

- Ya sabe programar lo básico.
- Quiere entender cómo pensar algorítmicamente, no solo generar código con IA.

- Busca prepararse para entrevistas técnicas donde se evalúa razonamiento, no solo sintaxis.

Este libro no es para quien:

- Busca copiar y pegar código sin entender.
- Quiere solo usar frameworks sin comprender los fundamentos.
- Cree que no es necesario aprender algoritmia porque "la IA lo hace todo".

### Si necesitas repasar conceptos: curso gratuito

Si te falta dominar algo de los fundamentos de programación, podrás aprenderlos de forma gratuita en el **curso de programación desde cero** que se encuentra disponible en el canal de YouTube (www.youtube.com/c/ProgramacionDesdeCero).

Aunque en ese curso las explicaciones y ejercicios están implementados en Python, el contenido no se centra en el lenguaje, sino en **desarrollar una base sólida** en lógica y construcción de algoritmos. El curso te prepara para abordar los desafíos de este libro.

## Sobre las ediciones de este libro

Están disponibles cuatro ediciones:

- **Edición Python**: todos los ejercicios implementados exclusivamente en Python.
- **Edición Java**: todos los ejercicios implementados exclusivamente en Java.
- **Edición C#**: todos los ejercicios implementados exclusivamente en C#.
- **Edición Multilenguaje**: incluye las implementaciones en los tres lenguajes para cada ejercicio.

Las ediciones mono-lenguaje son ideales para quien estudia o trabaja principalmente con uno de estos lenguajes y prefiere un contenido más conciso. La edición multilenguaje te permite comparar diferentes enfoques sintácticos.

El análisis algorítmico, los casos de prueba y las explicaciones son idénticos en todas las ediciones. Lo que varía es únicamente el código de implementación.

## Cómo aprovechar este libro

Este libro está diseñado para dos formas de uso complementarias:

**Resolución activa:** la idea es que, antes de revisar cualquier solución, te tomes un momento para trabajar cada desafío por tu cuenta. Esto implica: leer el enunciado con atención, examinar los casos de prueba, apoyarte en las pistas cuando sientas que no estás avanzando al ritmo que te gustaría y, finalmente, desarrollar tu propio código. Después tendrás la oportunidad de comparar tu propuesta con las soluciones que se presentan aquí. Este enfoque favorece un aprendizaje más sólido y una mayor autonomía.

**Lectura analítica:** si un ejercicio se vuelve particularmente exigente o si te resulta más interesante revisar el razonamiento antes de implementar, podrías explorar la sección "Del análisis a la solución" completa. Allí verás cómo evoluciona la idea inicial, pasando por alternativas poco eficientes hasta llegar a la versión validada. Una vez que la lógica te resulte clara, tendrás la posibilidad de implementar la solución sin necesidad de mirar el código directamente.

Ambas estrategias son completamente válidas, y podrías alternarlas según lo que necesites en cada momento. Habrá ejercicios que quieras resolver de inmediato y otros que te convenga estudiar antes. Lo esencial es evitar copiar código sin comprenderlo. Si un ejercicio te resulta muy sencillo, una lectura rápida será suficiente para continuar; si en cambio te plantea un reto mayor, vale la pena dedicar el tiempo necesario.

Los ejercicios están organizados de manera que cada uno se enfoque en una habilidad algorítmica o un tipo de estructura a practicar. Podrías seguir la secuencia sugerida o avanzar de manera distinta si hay algún ejercicio que despierte más tu interés.

## Estructura de cada desafío

Todos los desafíos de este libro se basan en **patrones clásicos de práctica algorítmica**, ampliamente utilizados en educación, preparación técnica y ejercicio profesional. Estos tipos de problemas, con sus variaciones, aparecen en entrevistas laborales, exámenes universitarios y plataformas de entrenamiento, porque permiten evaluar con claridad la capacidad de análisis, diseño y validación de soluciones.

Cada desafío sigue una **estructura** común y coherente:

- Enunciado.
- Ficha técnica (Objetivo de aprendizaje, Etiquetas y Etapa de aprendizaje).
- Casos de prueba.
- Ayuda o pistas.
- Del análisis a la solución, paso a paso.
- Conclusiones técnicas
- Implementación de la solución validada en Python, Java y C#.
- Otras alternativas de resolución.

Por cada solución mostrada (sea la solución validada o alguna de las soluciones intermedias) se incluye un breve análisis de su complejidad algorítmica y en qué se basa el cálculo.

## Por qué algunos ejercicios incluyen "Otra alternativa" y otros no

No todos los desafíos requieren una solución adicional. La sección "Otra alternativa" aparece únicamente cuando aporta valor pedagógico real y no cuando se trata simplemente de "otra forma de escribir lo mismo".

Se incluye solo en los siguientes casos:

- Uso de **estructuras o estrategias** más allá de las herramientas que forman parte del nivel esperado, pero cuya inclusión ayuda a mostrar cómo evolucionaría la solución con recursos más avanzados.
- Existencia de un **algoritmo óptimo** cuya dificultad excede el alcance del libro (por ejemplo, un algoritmo lineal o sublineal que requiere ideas avanzadas). La solución final del ejercicio siempre se mantiene dentro del nivel objetivo; la alternativa se muestra únicamente como referencia para el lector que quiere explorar más.
- Cuando una variante ilustra una **estrategia distinta** (recursión, "dividir y conquistar", uso explícito de tablas auxiliares, etc.) que, aunque no sea la más eficiente, enriquece la comprensión del problema y la construcción mental del algoritmo.

La idea no es acumular implementaciones, sino mostrar cuándo una solución alternativa enseña algo que la solución principal no enseña. Si una alternativa no agrega comprensión, no se incluye.

## Etapas del aprendizaje

Los 24 desafíos están organizados en tres etapas de aprendizaje, que no representan niveles de dificultad, sino tipos de razonamiento que los desafíos permiten ejercitar:

| Etapa | Ejercicios | Enfoque |
|---|---|---|
| Dominio de estructuras básicas | 1–12 | Consolidar el manejo de strings, arreglos y matrices mediante operaciones directas y patrones fundamentales. |
| Razonamiento con estructuras auxiliares | 13–19 | Aplicar estructuras como conjuntos y tablas hash para optimizar búsquedas, conteos y detección de patrones. |
| Pensamiento algorítmico integral | 20–24 | Resolver problemas que combinan múltiples técnicas y requieren estrategias algorítmicas no obvias. |

Esta progresión acompaña el camino natural de aprendizaje de la programación: desde la comprensión sintáctica, pasando por la aplicación estructurada, hasta la resolución de problemas más desafiantes que exigen abstraer, combinar y refinar ideas.

## Sobre el análisis e implementación de cada desafío

Cada desafío está pensado no solo para mostrar una solución correcta, sino para guiar el razonamiento que lleva hasta ella.

En programación, llegar a una buena solución suele implicar pasar por varias ideas previas: algunas incompletas, otras ineficientes o ingenuas, pero todas necesarias para comprender el problema a fondo. Por eso, en la sección "**Del análisis a la solución, paso a paso**", se presenta un apartado sobre la comprensión del problema, seguido de varios enfoques progresivos y soluciones alternativas:

- **Comprensión del problema:** análisis inicial para identificar aspectos clave que ayuden a resolver el problema. Muestra el pensamiento crítico necesario para empezar a plantear una solución. Para un programador, es práctica esencial; para entrevistas, es lo que diferencia a candidatos que tienen un razonamiento sólido de candidatos que solo escriben código.
- **Primer enfoque, segundo enfoque, etc.:** representan los intentos iniciales de resolución. Pueden ser ineficientes o incompletos, pero son valiosos para mostrar cómo evoluciona el razonamiento. Su función es ilustrar cómo un programador puede analizar, probar y mejorar una idea.
- **Solución validada:** es la versión recomendada y probada, que cumple todas las restricciones y pasa los casos de prueba.

- **Otras alternativas de implementación:** opcionalmente, muestra variantes que exploran estrategias diferentes o algoritmos más avanzados, con potenciales mejoras en eficiencia o legibilidad.

El formato de análisis refleja la evolución natural del razonamiento de un programador: se parte de **intentos iniciales incompletos o menos eficientes** y se avanza hacia una **solución óptima**. Esta metodología no solo replica el aprendizaje real en la práctica de la algoritmia, sino también el proceso que suele verse en entrevistas técnicas con ejercicios de "código en vivo" (o "*live coding*"), donde se valora la capacidad de explicar el razonamiento y luego optimizar el código. De la misma forma, se incluyen casos de prueba y una implementación básica de ellos, no solo para permitir probar fácilmente la solución validada de cada ejercicio sino para reforzar la idea de que las **pruebas** también son parte importante del desarrollo de un algoritmo.

El objetivo es que aprendas no solo qué hacer, sino por qué y cómo se llega a una solución sólida. Ver el proceso de refinamiento es clave para desarrollar pensamiento algorítmico y criterio técnico, dos competencias esenciales en la práctica profesional.

## Terminología y notaciones

En el libro se privilegia el uso de términos en español siempre que sea posible, pero se mantienen algunas palabras en inglés que son de uso universal en la programación.

El término "arreglo" se usa para referirse a una estructura de datos que almacena una colección de elementos del mismo tipo, accesibles por su posición o índice.

Para los números, se utiliza el punto como separador de miles y la coma como separador de decimales.

## Reglas del pseudocódigo

El pseudocódigo se presenta con un estilo claro y neutral, pensado para expresar la lógica de los algoritmos sin depender de ningún lenguaje de programación real. Se adopta un estilo **estructurado y académico**, legible para principiantes y coherente con las convenciones utilizadas en la enseñanza formal de algoritmia.

Su propósito es mostrar la **lógica del algoritmo**, no los detalles de implementación. Por eso, las secciones en pseudocódigo están pensadas para que puedas traducirlas mentalmente a un lenguaje concreto.

Las palabras clave se muestran en mayúsculas y se usa una sintaxis próxima a la que podría encontrarse en textos académicos o en concursos de programación.

Las siguientes reglas garantizan la uniformidad y legibilidad de todos los ejemplos:

- **Palabras clave en español:** `SI`, `ENTONCES`, `SI NO`, `MIENTRAS`, `PARA`, `RETORNAR`, `ESCRIBIR`, `LEER`, `FUNCION`, etc.
- **Palabras clave en mayúsculas:** cada bloque de control se cierra explícitamente: `FIN SI`, `FIN PARA`, `FIN MIENTRAS`, `FIN FUNCION`, etc.
- **Identificadores:** los nombres de variables y funciones se escriben en minúsculas, con palabras separadas por guion bajo si es necesario. Ejemplo: `suma_total`, `mayor_elemento`.
- **Variables:** se usan directamente, sin declaraciones ni tipos.

- **Sangría:** se usa una sangría de 4 espacios por nivel de anidamiento para reflejar cada bloque de código.
- **Asignación:** se utiliza el símbolo ← para asignar valores. Ejemplo: `contador ← contador + 1`.
- **Operadores de comparación:** se usan los símbolos clásicos: `=`, `≠`, `<`, `>`, `≤`, `≥`.
- **Operadores lógicos:** usan las palabras `Y` ("and"), `O` ("or") y `NO` ("not").
- **Comentarios:** indicados con `//` y ubicados en la misma línea o en líneas separadas.
- **Bucles numéricos:** se escriben con la estructura `PARA i DESDE 1 HASTA n HACER`, asumiendo que `i` se incrementa en una unidad, salvo que se indique otra cosa. En el caso de iteraciones que decrementan el valor, se indica como `HACIA ATRÁS`.
- **Acceso a elementos de un arreglo o una cadena (string):** se usa la notación con corchetes para acceder a caracteres o elementos: `texto[i]`, `arreglo[j]`.
- **Pertenencia:** se usa la notación `elemento EN estructura` para verificar si un elemento se encuentra en una estructura.
- **Definición de funciones:** toda función se declara con la palabra clave y la estructura `FUNCION nombre(parámetros)` y finaliza con `FIN FUNCION`.
- **Funciones:** si se usan funciones auxiliares no definidas en el código, éstas se encuentran descriptas a continuación. Ejemplos: la función `longitud` para retornar cantidad de caracteres en una cadena o de elementos en un arreglo; o `conjunto_de_caracteres` para crear un conjunto cuyos elementos son caracteres.
- **Inicializar estructura de datos:** se usan los elementos listados entre llaves, en una función constructora que se asume existente. Ejemplo: `vocales ← conjunto_de_caracteres({"a", "e", "i", "o", "u"})`

Este formato te permite concentrarte en la lógica esencial del algoritmo, sin distracciones sintácticas propias de un lenguaje específico. Además, facilita la comparación entre distintos algoritmos y la posterior traducción de la solución recomendada a Python, Java y C#.

Ejemplo de función que verifica si todos los caracteres de una cadena son vocales:

```
1   FUNCION todas_son_vocales(texto)
2       vocales ← conjunto_de_caracteres({'a', 'e', 'i', 'o', 'u'})  // Vocales
3
4       // Recorre la cadena de texto verificando vocales
5       PARA i DESDE 0 HASTA longitud(texto) - 1 HACER
6           SI NO (texto[i] EN vocales) ENTONCES
7               RETORNAR FALSO
8           FIN SI
9       FIN PARA
10
11      RETORNAR VERDADERO
12  FIN FUNCION
```

## Decisiones sobre la implementación

En todos los ejercicios se asume que las **especificaciones de entrada** se cumplen exactamente tal como se describen en el enunciado. Por lo tanto, ni los casos de prueba ni los algoritmos incluyen validaciones de entrada (por ejemplo, comprobaciones de tipo o de rango). El objetivo es concentrar la atención en la lógica del problema y en la construcción del algoritmo, no en el manejo de errores o entradas inválidas.

En las soluciones implementadas:

- **En Python:** el código se presenta como un *script* ejecutable directamente, seguido de una breve sección de pruebas de validación. Toda la solución se muestra como funciones y llamadas directas, sin definir clases.
- **En Java:** se incluye la clase mínima con un método `main`, ya que el lenguaje lo requiere para la ejecución.
- **En C#**: el código se presenta como un *script* ejecutable directamente, seguido de una breve sección de pruebas de validación. Para ello se utiliza el entorno `dotnet-script`, que permite ejecutar código C# secuencial, de manera similar a muchos lenguajes de *scripting*.

Las soluciones alternativas solo muestran la parte relevante con el algoritmo, evitando pruebas de validación. Este formato busca centrar la atención en la lógica algorítmica.

En los ejercicios basados en **arreglos** (problemas sobre indexación, subarreglos, matrices, etc.):

- **En Python** se utiliza el tipo `list` por su correspondencia directa con el pseudocódigo. Además, en las implementaciones se utiliza *type hinting* tanto en los parámetros de las funciones como, en algunos casos, en variables locales cuyo tipo podría prestarse a confusión. El objetivo es favorecer la legibilidad y ayudarte a reconocer de inmediato qué tipo de dato espera cada función, sin que esto implique una obligación de verificación estricta en tiempo de ejecución.
- **En Java y C#** se utilizan los tipos nativos (como `int[]`, `String[]`, `int[][]`) por su claridad y su correspondencia directa con el pseudocódigo. En cambio, en los ejercicios donde los datos requieren estructuras dinámicas se emplean las clases específicas que las implementan, para aprovechar las colecciones modernas provistas por el lenguaje.

En algunos ejercicios, las soluciones presentadas no emplean las formas más resumidas, idiomáticas o propias del lenguaje. Esto es intencional: se prioriza la claridad del razonamiento y la correspondencia directa con el pseudocódigo, de modo que puedas entender paso a paso cómo se llega al resultado. En la práctica profesional podrían existir versiones más compactas o expresivas de ciertos algoritmos, pero el propósito aquí es hacer visible la **estructura lógica** subyacente, no la optimización sintáctica.

Finalmente, debe aclararse que la programación orientada a objetos no forma parte del contenido de este libro: el foco está en funciones puras y datos simples, para que puedas comprender y optimizar algoritmos con independencia del paradigma.

### Pruebas

Solo las **soluciones validadas** incluyen en su implementación un bloque de pruebas para verificar si los casos de prueba pasan o fallan. Las implementaciones alternativas no repiten

esas pruebas, ya que se asume que es posible reutilizarlas simplemente reemplazando la implementación del algoritmo.

Cuando se incluye, el **bloque de pruebas** sigue un formato uniforme para facilitar su lectura y su ejecución directa. La salida de su ejecución mostrará los datos de entrada, el resultado obtenido por el programa, el resultado esperado y una indicación de si la prueba fue superada. Cuando la función recibe más de un argumento, éstos se presentan separados por comas: `Entrada: arg1, arg2, arg3 → Resultado: valor (esperado: valor). Pasa/Falla.`. Si algún argumento o valor de salida es de tipo arreglo, los valores se mostrarán entre corchetes: `Entrada: [arg1], [arg2], arg3 → Resultado: [valor] (esperado: [valor]). Pasa/Falla.`

## Complejidad algorítmica

La complejidad algorítmica es una herramienta fundamental que permite analizar **cuánto tiempo y memoria requiere un algoritmo** independientemente del hardware donde se ejecute. En lugar de medir en segundos o megabytes (que varían según la computadora), se mide **en función del tamaño de la entrada**: cuántas operaciones se necesitan y cuánto espacio adicional se requiere cuando los datos crecen. Esta métrica es importante en dos contextos principales:

- **En la práctica profesional:** permite identificar cuellos de botella, comparar alternativas y elegir la solución más eficiente antes de implementarla. Un algoritmo con mala complejidad puede funcionar bien con pocos datos pero colapsar cuando la entrada crece.
- **En entrevistas técnicas:** es uno de los criterios más evaluados. Saber analizar y optimizar la complejidad demuestra capacidad para diseñar soluciones escalables, no solo código que "funciona" con casos pequeños.

En cada ejercicio se analiza la **complejidad temporal y espacial** del algoritmo, utilizando la notación *Big-O*. Esta medida permite comparar soluciones y entender cómo crece el tiempo de ejecución o el consumo de memoria a medida que aumenta el tamaño de la entrada. En términos prácticos: si el número de operaciones crece linealmente con la entrada, la complejidad temporal es O(n). Si crece con el cuadrado del tamaño, es $O(n^2)$, y así sucesivamente.

### Convención sobre complejidad espacial

Para la complejidad espacial se sigue la convención de considerar si el espacio en memoria a utilizar crece de manera proporcional a la entrada o no:

- La **entrada** no se cuenta como espacio extra.
- La **salida** tampoco se cuenta cuando es una modificación de la estructura de entrada (por ejemplo, ordenar un arreglo "*in-place*").
- Sin embargo, cuando el algoritmo debe crear una **estructura de salida completamente nueva y diferente** (en tamaño o tipo), esa estructura sí se cuenta en el análisis espacial. Esto incluye arreglos nuevos, matrices, listas, conjuntos, tablas hash, o cadenas de texto que crecen con la entrada.
- **Variables simples** (números, booleanos, índices) y cadenas de longitud fija siempre se consideran O(1) porque no dependen del tamaño de la entrada.

Por ejemplo: si un algoritmo recibe un arreglo de `n` elementos y retorna ese mismo arreglo ordenado (modificado en el lugar), la complejidad espacial es O(1) porque no creó estruc-

turas nuevas. Pero, si recibe un arreglo y retorna un arreglo completamente nuevo con los resultados, la complejidad espacial es O(n).

### ¿Cómo se calcula la complejidad temporal en la práctica?

La forma más directa es contar bucles:

- Un bucle que recorre `n` elementos: **O(n)**
- Dos bucles anidados sobre `n` elementos: **O(n²)**
- Un bucle dentro de otro, pero el interno no depende del externo: se suman, por ejemplo **O(n + m)**
- Operaciones que dividen el problema a la mitad en cada paso: **O(log n)** (como la búsqueda binaria).
- Operaciones sobre tablas hash o conjuntos: **O(1)** en promedio para búsqueda, inserción y eliminación

No todos los algoritmos se pueden analizar contando bucles (algunos requieren análisis matemático más avanzado), pero esta heurística funciona para la gran mayoría de los casos que se verán en este libro.

### Simplificación en *Big-O*: qué se ignora

La notación *Big-O* se enfoca en el comportamiento dominante cuando la entrada crece mucho, por lo que se ignoran ciertos elementos:

1. **Las constantes se descartan:**
   - Si un algoritmo hace `5n` operaciones, se escribe como O(n), no O(5n).
   - Si hace `3n + 100`, sigue siendo O(n).
   - Las constantes no importan porque cuando `n` es muy grande (por ejemplo, un millón), la diferencia entre `5n` y `n` es proporcional, no cambia el orden de magnitud.
2. **Se mantiene solo el término dominante:**
   - Si un algoritmo tiene complejidad O(n²) + O(n), se simplifica a O(n²).
   - Cuando `n` crece, el término `n²` domina completamente sobre `n`. Por ejemplo: con `n = 1000`, `n² = 1.000.000` versus `n = 1000`. El segundo término se vuelve despreciable.
   - En general: O(1) < O(log n) < O(n) < O(n log n) < O(n²) < O(n³) < O(2ⁿ).
3. **Términos constantes pequeños también se ignoran:**
   - Una tabla hash de tamaño fijo (por ejemplo, 26 letras del alfabeto) tiene complejidad O(1), no O(26).
   - Si un algoritmo usa O(n) + O(26), el resultado es O(n) porque 26 es constante y despreciable frente a `n` cuando `n` crece.

Ejemplos prácticos de simplificación:

- `O(2n)` → O(n) (la constante 2 se descarta).
- `O(n) + O(n)` → O(2n) → O(n) (se suman, luego se descarta la constante).
- `O(n²) + O(n)` → O(n²) (se mantiene solo el término dominante).
- `O(n * m) + O(n)` → O(n * m) si `m` es comparable a `n` o mayor.

Algunos algoritmos dependen de **más de una variable**. Por ejemplo, si la entrada tiene dos componentes, una cadena de longitud `n` y una de longitud `m`:

- **Si ambas variables son importantes y pueden crecer independientemente**, se expresan ambas: O(n + m) o O(n * m).
- **Si una es mucho menor que la otra**, a veces se expresa solo la dominante para simplificar, aclarando la relación.

Por ejemplo: "La división de la cadena es O(m) y la validación es O(n). El costo total es O(n + m)" significa que ambas operaciones se ejecutan en secuencia, no anidadas, por lo que se suman.

### Cuándo importa la complejidad

El objetivo no es realizar cálculos formales, sino **desarrollar la intuición necesaria** para decidir si un algoritmo es adecuado según el tamaño estimado de los datos. Por ejemplo, si se sabe que una entrada nunca superará los mil elementos, una solución $O(n^2)$ puede ser perfectamente aceptable, incluso más legible que una versión O(n log n) más complicada de entender.

Como referencia aproximada, una computadora moderna puede ejecutar alrededor de mil millones ($10^9$) de operaciones elementales por segundo. Esto permite estimar de manera intuitiva el impacto de cada complejidad. Por ejemplo, un algoritmo $O(n^2)$ con una entrada de un millón de elementos requeriría alrededor de $10^{12}$ operaciones, lo que equivaldría a unos 1.000 segundos (≈16,6 minutos) de ejecución, que normalmente es inaceptable en la práctica.

El aprendizaje de la complejidad algorítmica no busca memorizar fórmulas, sino entender los límites prácticos de cada enfoque y **aprender a elegir** conscientemente la solución más adecuada.

## Repositorio de código adjunto

Este libro se entrega con un **repositorio** que incluye todas las implementaciones organizadas por ejercicio. El repositorio contiene el código completo y ejecutable: todas las **soluciones validadas** con sus casos de prueba, listas para ejecutar directamente, además de las **soluciones alternativas** (cuando un ejercicio presenta alternativas de implementación, éstas se incluyen como archivos separados claramente identificados).

Un archivo `readme.md` en la raíz del repositorio explica los detalles.

El repositorio te permite:

- Ejecutar los programas sin necesidad de copiar código del PDF.
- Modificar las implementaciones para experimentar con variaciones.
- Comparar tu código con las soluciones presentadas.
- Usar los casos de prueba como base para probar otras soluciones.

## Cómo probar el código

Las implementaciones presentadas en el libro están diseñadas para que puedas probarlas directamente:

- **Soluciones validadas:** cada una forma un programa completo que incluye casos de prueba y puede ejecutarse tal como está.

- **Soluciones alternativas:** proveen el algoritmo que puede reemplazar la implementación de la solución validada, reutilizando el mismo bloque de pruebas.

No es necesario configurar proyectos complejos ni instalar dependencias adicionales. Los programas utilizan solo las bibliotecas estándar del lenguaje.

En el repositorio de código se encuentran las instrucciones detalladas de instalación del entorno y dos formas de ejecutar el código: desde la terminal o desde Visual Studio Code con la extensión "Code Runner".

## Sobre futuras actualizaciones

Este libro podría ser actualizado para corregir erratas, incorporar mejoras o añadir nuevos comentarios didácticos.

La versión actual es 1.0 (noviembre 2025). Las versiones futuras se identificarán con un número incremental (1.1, 2.0, etc.) y un registro de cambios.

Los compradores de la versión 1.x recibirán gratuitamente todas las actualizaciones menores (correcciones de erratas, mejoras de formato, aclaraciones) publicadas dentro de la misma serie. Las actualizaciones con contenido nuevo sustancial podrían identificarse como una nueva versión mayor (2.0) y se evaluará su modelo de distribución en ese momento.

Tu nombre - tu@email.com

# Desafío 1 - Mensajes simétricos

## 1.1. Enunciado

Estás trabajando en una aplicación de mensajería donde piden detectar si un mensaje puede considerarse "simétrico". Un mensaje es simétrico si puede leerse igual de izquierda a derecha que de derecha a izquierda (lo que normalmente se llama "palíndromo"), eliminando, como máximo una letra. La aplicación trabaja con mensajes que solo contienen letras minúsculas, sin espacios. Tu tarea es verificar si un mensaje es simétrico. Si el mensaje es un palíndromo, la respuesta es afirmativa; y si quitando una sola letra llegara a serlo, también se debe considerar simétrico. Por ejemplo, si alguien escribe "reconocer", es simétrico y no hace falta eliminar nada. Si escribe "reconcer", eliminando una letra se puede lograr que el texto sea palíndromo, entonces también es simétrico. Pero si el mensaje no tiene manera de volverse palíndromo quitando solo una letra, se debe indicar que no es posible. Solo se puede eliminar una letra como máximo, y no se permiten más cambios.

### 1.1.1. Especificaciones de la entrada

- Longitud del mensaje (L): $1 \leq L \leq 10^5$.
- Caracteres: el mensaje solo contiene letras minúsculas del alfabeto español (a-z).

### 1.1.2. Aclaraciones

- La entrada es una cadena (*string*).
- La salida debe ser un valor booleano.
- Solo se puede eliminar, como máximo, un carácter. No se permiten inserciones ni reordenamientos.

## 1.2. Ficha técnica

**Objetivo de aprendizaje:** Aprender a resolver problemas comparando información desde ambos extremos de una estructura, detectando diferencias y tomando decisiones bajo restricciones específicas.

**Etiquetas:** strings, cadenas, palíndromos, dos_índices, validación.

**Etapa de aprendizaje:** Dominio de estructuras básicas.

## 1.3. Casos de prueba

```
Entrada: "aba"
Salida: VERDADERO
Explicación: ya es un palíndromo, no hace falta eliminar ninguna letra.
```

```
Entrada: "abca"
Salida: VERDADERO
Explicación: si se elimina la 'c', queda "aba", que sí es palíndromo.
```

```
Entrada: "abc"
Salida: FALSO
Explicación: aunque se elimine una letra, nunca quedará un palíndromo ("ab" o
"bc" no lo son).
```

```
Entrada: "a"
Salida: VERDADERO
Explicación: un solo carácter siempre es palíndromo por definición.
```

```
Entrada: "deeee"
Salida: VERDADERO
Explicación: eliminando la 'd', queda "eeee", que sí es palíndromo.
```

```
Entrada: "abcdefgfedcbaa"
Salida: VERDADERO
Explicación: toda la cadena es simétrica salvo por la 'a' extra al final;
eliminándola queda un palíndromo perfecto.
```

```
Entrada: "abcdefg"
Salida: FALSO
Explicación: la cadena no tiene forma de ser palíndromo eliminando solo una
letra, pues hay múltiples diferencias.
```



## 1.4. Ayuda

- Podrías intentar comparar letras desde los extremos hacia el centro; si hay una diferencia, podrías eliminar una de las dos y verificar el resto.
- Un error frecuente es seguir comprobando después de más de una eliminación. No olvides que solo está permitido borrar una letra como máximo.

## 1.5. Del análisis a la solución, paso a paso

A continuación se presentan distintos enfoques progresivos. Los primeros intentos pueden ser ineficientes o incompletos; el objetivo es mostrar posibles etapas del razonamiento hasta llegar a una solución final validada como aceptable.

### 1.5.1. Comprensión del problema

Un **palíndromo** es una palabra que se lee igual de izquierda a derecha que de derecha a izquierda. Ejemplos clásicos son "ana", "radar" o "reconocer". El desafío aquí va un paso

más allá: no solo se busca identificar palíndromos perfectos, sino determinar si una palabra **puede convertirse en palíndromo eliminando como máximo una letra**.

Esto significa que hay tres escenarios posibles:

- La palabra **ya es un palíndromo**: no hace falta eliminar nada. Ejemplo: "ana" → ya es palíndromo.
- La palabra **se convierte en palíndromo** eliminando exactamente una letra: ejemplo: "abca" → eliminando la 'b' queda "aca" (palíndromo), o eliminando la 'c' queda "aba" (palíndromo).
- La palabra **no puede ser palíndromo** ni eliminando una letra: ejemplo: "abc" → ninguna eliminación individual produce un palíndromo.

Observaciones importantes:

- Solo se permite eliminar una letra como máximo (no dos, ni tres).
- La letra a eliminar puede estar en cualquier posición (no necesariamente al inicio o al final).
- Si hay múltiples formas de convertir la palabra en palíndromo eliminando diferentes letras, cualquiera es válida (solo importa que exista al menos una).

La pregunta clave es: ¿cómo detectar eficientemente cuál letra debe eliminarse (si es que alguna debe eliminarse), sin tener que probar eliminar cada letra y verificar todas las posibilidades?

### 1.5.2. Primer enfoque (intento inicial)

Si un palíndromo es texto que se lee igual al derecho y al revés, entonces se podría invertir ese texto y compararlo con la versión "al derecho". Si son iguales, es palíndromo. Algunos lenguajes de programación tienen funcionalidad para **invertir cadenas**, y si no existe se podría crear esa función manualmente (tal vez recorriendo la cadena desde el final hacia el principio), así que se podría dar por hecha la función `invertir(cadena)` para plantear una solución inicial que verifica si el mensaje es simétrico mediante la comprobación de si es palíndromo:

**Pseudocódigo:**

```
FUNCION palindromo_modificado(cadena)
    // Compara la cadena con su versión invertida
    SI cadena = invertir(cadena) ENTONCES
        RETORNAR VERDADERO
    SI NO
        RETORNAR FALSO
    FIN SI
FIN FUNCION
```

**Reflexión didáctica:** Este sería un punto de partida natural. Es simple y correcto para detectar palíndromos exactos. Pero aún falta considerar el caso de tener que eliminar una letra. La complejidad temporal de este algoritmo depende del tamaño de la cadena porque invierte la cadena una vez. Si `n` es la longitud de la cadena, la complejidad temporal expresada en notación *Big O* es O(n). Además, el algoritmo utiliza, para la comparación, una cadena invertida de igual tamaño que la cadena original (longitud `n`), por lo que su complejidad espacial es también O(n).

### 1.5.3. Segundo enfoque (intento de mejora)

Ahora es necesario considerar el caso de "una letra menos": si no es palíndromo, se podría intentar **borrar una letra** y ver si eso lo arregla. Para lograrlo, se podría recorrer la cadena letra por letra, de principio a fin, comprobando si, ignorando la letra actual, el resto de la cadena es palíndromo.

**Pseudocódigo:**

```
1   FUNCION palindromo_modificado(cadena)
2       // Si la cadena ya es palíndromo, no es necesario eliminar nada
3       SI cadena = invertir(cadena) ENTONCES
4           RETORNAR VERDADERO
5       FIN SI
6
7       // Prueba eliminar cada carácter y verificar si la subcadena que queda
    es palíndromo
8       PARA i DESDE 0 HASTA longitud(cadena) - 1 HACER
9           subcadena ← cadena sin el carácter en posición i
10          SI subcadena = invertir(subcadena) ENTONCES
11              RETORNAR VERDADERO
12          FIN SI
13      FIN PARA
14
15      // Si ninguna eliminación funcionó, no puede ser palíndromo
16      RETORNAR FALSO
17  FIN FUNCION
```

Se asume la existencia de una función llamada `longitud` que, dada una cadena, devuelve la cantidad de caracteres o de elementos que contiene.

**Reflexión didáctica:** Esta opción de buscar "a fuerza bruta" todas las posibilidades suele ser el paso lógico después del anterior. Ahora el algoritmo contempla la idea de "eliminar una letra", pero es ineficiente, porque invierte casi toda la cadena (excepto una letra) en cada intento, es decir que se ejecuta el algoritmo de invertir la cadena por

cada letra del mensaje. En el peor caso (que se daría cuando la cadena debe recorrerse por completo porque no se encuentra una diferencia antes), el bucle externo se ejecuta `n` veces y, dentro del bucle, `invertir(cadena)` debe recorrer los `n` caracteres de la cadena (entonces iterará otras `n` veces). En total, todo esto hace que la complejidad del algoritmo sea $O(n^2)$, porque en cada una de las `n` iteraciones del bucle se invierte una cadena de longitud aproximada `n`. La complejidad espacial es O(n) porque crea una subcadena nueva en cada iteración del bucle externo (hasta `n` veces). Un siguiente intento debería buscar optimizar el algoritmo para hacerlo más eficiente.

### 1.5.4. Solución validada

Si se usa un índice que empieza desde el inicio (o la izquierda) y otro desde el final (o la derecha) de la cadena y ambos se van moviendo hacia el centro, comparando los extremos, solo es necesario borrar una letra en el momento exacto donde no coinciden.

¿Por qué funciona esta técnica? Si una cadena es palíndromo, cada carácter en la posición `i` desde la izquierda debe coincidir con el carácter en la posición `i` desde la derecha. Al **comparar simultáneamente desde ambos extremos hacia el centro**, se reduce a la mitad el número de comparaciones necesarias. Cuando se encuentra la primera diferencia, se sabe que al menos uno de esos dos caracteres debe ser eliminado para que el resto forme un palíndromo. Solo hay dos opciones: eliminar el izquierdo o el derecho, y basta con verificar si alguna de esas dos alternativas resulta en un palíndromo.

Podría probarse con los siguientes pasos:

1. Empezar con `izquierda = 0` (incrementándose en cada iteración) y `derecha = longitud - 1` (decrementándose).
2. Comparar los caracteres en esas posiciones.
   - Si son iguales, avanzar.
   - Si no lo son, probar ignorar uno de los dos
     - Definir si lo que queda es palíndromo.

**Pseudocódigo:**

```
1   FUNCION palindromo_modificado(cadena)
2       // Define dos índices: uno que empieza desde la izquierda y otro desde
    la derecha de la cadena
3       izquierda ← 0
4       derecha ← longitud(cadena) - 1
5
6       MIENTRAS izquierda < derecha HACER
7           SI cadena[izquierda] = cadena[derecha] ENTONCES
8               izquierda ← izquierda + 1
9               derecha ← derecha - 1
10          SI NO
```

```
            // Si se encuentra una diferencia, prueba ignorando uno de los caracteres
            RETORNAR es_palindromo(cadena, izquierda + 1, derecha)
                  O es_palindromo(cadena, izquierda, derecha - 1)
        FIN SI
    FIN MIENTRAS

    RETORNAR VERDADERO
FIN FUNCION


// Función auxiliar que comprueba si la parte de una cadena comprendida entre inicio y fin es palíndromo
FUNCION es_palindromo(cadena, inicio, fin)
    MIENTRAS inicio < fin HACER
        SI cadena[inicio] ≠ cadena[fin] ENTONCES
            RETORNAR FALSO
        FIN SI
        inicio ← inicio + 1
        fin ← fin - 1
    FIN MIENTRAS

    RETORNAR VERDADERO
FIN FUNCION
```

**Ejemplo de ejecución** con `cadena = "abca"`:

```
Iteración 1: izquierda = 0; derecha = 3
           cadena[0] = cadena[3] → 'a' = 'a' → VERDADERO
           → izquierda = 1
           → derecha = 2
Iteración 2: izquierda = 1; derecha = 2
           cadena[1] = cadena[2] → 'b' = 'c' → FALSO
es_palindromo(cadena, 2, 2) → VERDADERO
es_palindromo(cadena, 1, 1) → VERDADERO
→ VERDADERO O VERDADERO → VERDADERO

Resultado final: VERDADERO
```

## 1.6. Complejidad algorítmica de la solución validada

### 1.6.1. Temporal

Complejidad de ambas funciones: **O(n)**.

- Función `palindromo_modificado`: O(n) en el peor caso, donde n es la longitud del mensaje.

- Función `es_palindromo`: O(n) en el peor caso, donde n es la longitud de la subcadena analizada.

En el mejor caso (cuando la cadena ya es palíndromo), se recorre la cadena una vez desde ambos extremos hasta el centro. El peor caso sería que el bucle principal de `palindromo_modificado` recorra n/2 caracteres, que es el máximo posible (porque itera con dos índices en paralelo que recorren, cada uno, la mitad de la cadena desde un extremo hacia el centro), entonces la complejidad sería O(n / 2). Como las constantes se ignoran en la notación *Big O*, el resultado final es O(n).

En el peor caso (cuando necesita verificar ambos caminos):

1. Primero recorre hasta encontrar el carácter que difiere: O(k), donde `k` es la longitud que necesitó recorrer (aproximadamente n/2 si el carácter diferente está cerca del centro).
2. Luego llama a `es_palindromo` dos veces con subcadenas de longitud `m` y `p`, donde tanto `m` como `p` son menores que `n`, pero en el peor caso serán aproximadamente similares a `n`.
3. Total: O(n / 2) + O(n) + O(n) = O(2,5n) = O(n).

### 1.6.2. Espacial

Complejidad de ambas funciones: **O(1)**.

Ambas funciones usan solo variables enteras y no asignan memoria adicional que dependa del tamaño de la entrada. Crea la variable que acumula toda la cadena invertida, que es una cadena de longitud `n` pero es el dato de salida exigido por el algoritmo, por lo que no se cuenta como espacio extra. Las llamadas a `es_palindromo` no añaden complejidad espacial porque son llamadas iterativas simples, no recursivas (la función no se llama a sí misma).

## 1.7. Conclusiones técnicas

La solución validada muestra cómo funciona la técnica de "**dos índices**" o "**dos punteros**" ("*two pointers*", por el nombre en inglés más común) para problemas de verificación simétrica (aquí "puntero" significa índice, no referencia en memoria). En lugar de generar y comparar múltiples subcadenas (enfoque de fuerza bruta), se aprovecha la naturaleza bidireccional del problema: un palíndromo debe ser simétrico desde ambos extremos.

La evolución de los tres enfoques muestra la optimización paso a paso:

- **Inversión simple**: O(n) tiempo, O(n) espacio. Verifica palíndromos exactos. Resuelve solo parte del problema.
- **Fuerza bruta con eliminaciones**: O($n^2$) tiempo, O(n) espacio. Funciona correctamente pero es ineficiente: invierte la cadena `n` veces. En algunos lenguajes podría comportarse aún peor y llegar a O($n^3$) en el tiempo, debido al costo de crear nuevas subcadenas.
- **Dos índices con verificación selectiva**: O(n) tiempo, O(1) espacio. Óptima: solo verifica subcadenas cuando encuentra una diferencia.

La clave del algoritmo está en detectar el primer punto donde los extremos no coinciden. En ese momento, hay exactamente dos posibilidades: eliminar el carácter izquierdo o el derecho. La técnica de dos índices permite verificar ambas opciones de manera eficiente, recorriendo cada posible subcadena una sola vez.

Esta es una estrategia que se puede aplicar a otros problemas similares: comparar desde los extremos, detenerse en la primera diferencia, y verificar las dos opciones posibles. La restricción de eliminar solo una letra convierte un problema complejo en algo que se resuelve con un solo recorrido.

Otra ventaja es que el algoritmo trabaja directamente sobre la cadena original: no necesita crear copias ni usar estructuras auxiliares, lo que es importante para cadenas largas (hasta $10^5$ caracteres según las especificaciones).

Este patrón de **recorrer una secuencia desde ambos extremos** aparece en tareas reales donde interesa detectar simetría, complementariedad o coincidencias de borde. En **bioinformática**, por ejemplo, se usa para comprobar secuencias palindrómicas o repeticiones invertidas asociadas a sitios de restricción. También se aplica en el procesamiento eficiente de strings para **normalización**, **eliminación de caracteres** no deseados en los extremos o **detección de prefijos o sufijos** coincidentes. En análisis de datos y señales, este enfoque permite **verificar simetrías** en imágenes o series temporales comparando desde los extremos hacia el centro.

## 1.8. Implementación de la solución validada

**Código Python (con pruebas):**

```python
# --- Implementación del algoritmo ---
def es_palindromo(cadena: str, inicio: int, fin: int) -> bool:
    while inicio < fin:
        if cadena[inicio] != cadena[fin]:
            return False
        inicio += 1
        fin -= 1
    return True

def palindromo_modificado(cadena: str) -> bool:
    izquierda = 0
    derecha = len(cadena) - 1

    while izquierda < derecha:
        if cadena[izquierda] == cadena[derecha]:
            izquierda += 1
            derecha -= 1
        else:
            puede_ser_palindromo = es_palindromo(
                cadena, izquierda + 1, derecha) or es_palindromo(
                    cadena, izquierda, derecha - 1)
            return puede_ser_palindromo
    return True


# --- Verificación de casos de prueba ---
casos = {
    "aba": True,
    "abca": True,
    "abc": False,
    "a": True,
```

```
    "deeee": True,
    "abcdefgfedcbaa": True,
    "abcdefg": False
    }

for entrada, salida in casos.items():
    resultado = palindromo_modificado(entrada)

    print(f"Entrada: {entrada} "
        f"-> Resultado: {resultado} "
        f"(esperado: {salida}).", end=" ")

    if resultado == salida:
        print("Pasa.")
    else:
        print("Falla.")
```

**Código Java (con pruebas):**

```
class Main {
    // --- Implementación del algoritmo ---
    static boolean esPalindromo(String cadena, int inicio, int fin) {
        while (inicio < fin) {
            if (cadena.charAt(inicio) != cadena.charAt(fin)) {
                return false;
            }
            inicio++;
            fin--;
        }
        return true;
    }

    static boolean palindromoModificado(String cadena) {
        int izquierda = 0;
        int derecha = cadena.length() - 1;

        while (izquierda < derecha) {
            if (cadena.charAt(izquierda) == cadena.charAt(derecha)) {
                izquierda++;
                derecha--;
            }
            else {
                return esPalindromo(cadena, izquierda + 1, derecha)
                    || esPalindromo(cadena, izquierda, derecha - 1);
            }
        }
        return true;
    }


    // --- Verificación de casos de prueba ---
    public static void main(String[] args) {
        String[] entradas = {
            "aba",
```

```
            "abca",
            "abc",
            "a",
            "deeee",
            "abcdefgfedcbaa",
            "abcdefg"
        };
        boolean[] salidas = {
            true, true, false, true, true, true, false
        };

        for (int i = 0; i < entradas.length; i++) {
            boolean resultado = palindromoModificado(entradas[i]);

            System.out.print(
                "Entrada: " + entradas[i] +
                " -> Resultado: " + resultado +
                " (esperado: " + salidas[i] + "). ");

            if (resultado == salidas[i]) {
                System.out.println("Pasa.");
            }
            else {
                System.out.println("Falla.");
            }
        }
    }
}
```

**Código C# (con pruebas):**

```
// --- Implementación del algoritmo ---
bool EsPalindromo(string cadena, int inicio, int fin)
{
    while (inicio < fin)
    {
        if (cadena[inicio] != cadena[fin])
        {
            return false;
        }
        inicio++;
        fin--;
    }
    return true;
}

bool PalindromoModificado(string cadena)
{
    int izquierda = 0;
    int derecha = cadena.Length - 1;

    while (izquierda < derecha)
    {
        if (cadena[izquierda] == cadena[derecha])
```

```
        {
            izquierda++;
            derecha--;
        }
        else
        {
            return EsPalindromo(cadena, izquierda + 1, derecha)
                    || EsPalindromo(cadena, izquierda, derecha - 1);
        }
    }
    return true;
}


// --- Verificación de casos de prueba ---
var entradas = new string[] {
    "aba",
    "abca",
    "abc",
    "a",
    "deeee",
    "abcdefgfedcbaa",
    "abcdefg"
};
var salidas = new bool[] {
    true, true, false, true, true, true, false
};

for (int i = 0; i < entradas.Length; i++)
{
    var resultado = PalindromoModificado(entradas[i]);

    Console.Write(
        $"Entrada: {entradas[i]} " +
        $"-> Resultado: {resultado} " +
        $"(esperado: {salidas[i]}). ");

    if (resultado == salidas[i])
    {
        Console.WriteLine("Pasa.");
    }
    else
    {
        Console.WriteLine("Falla.");
    }
}
```

Tu nombre - tu@email.com

# Desafío 2 - Organizando los productos de una tienda online

## 2.1. Enunciado

Se está desarrollando un sistema para organizar productos por nombre en una tienda online. Los nombres de los productos suelen compartir una parte inicial común (por ejemplo, "camisaroja", "camisaverde", "camisanegra"), y el sistema necesita identificar automáticamente esa sección compartida para clasificar los artículos en grupos. Tu tarea consiste en escribir un algoritmo que reciba un arreglo de nombres de productos (cadenas de texto) y determine cuál es el prefijo más largo que todos ellos tienen en común. Si los nombres no comparten ningún inicio común, el algoritmo debe devolver una cadena vacía.

### 2.1.1. Especificaciones de la entrada

- Longitud del arreglo (L1): $1 \leq L1 \leq 200$.
- Longitud de cada nombre de producto (L2): $0 \leq L2 \leq 200$.
- Un nombre de producto puede ser una cadena vacía.
- Cada nombre de producto no vacío solo puede estar compuesto de letras minúsculas del alfabeto español (a-z).

### 2.1.2. Aclaraciones

- La entrada es un arreglo de cadenas (*strings*).
- La salida debe ser una cadena (*string*).
- Si no hay prefijo común, debe retornarse una cadena vacía.

## 2.2. Ficha técnica

**Objetivo de aprendizaje:** Desarrollar la lógica para comparar elementos y encontrar patrones comunes, como habilidad fundamental en tareas de procesamiento de datos y agrupamiento de información.

**Etiquetas:** strings, cadenas, patrones, comparación.

**Etapa de aprendizaje:** Dominio de estructuras básicas.

## 2.3. Casos de prueba

```
Entrada: ["camisaroja", "camisaverde", "camisanegra"]
Salida: "camisa"
Explicación: Hay un prefijo común claro en todos los elementos.
```

```
Entrada: ["pantalon", "camisa", "abrigo"]
Salida: ""
Explicación: Las cadenas del arreglo son completamente diferentes.
```

```
Entrada: ["abc", "abcd", "ab"]
Salida: "ab"
Explicación: El prefijo está limitado por el elemento más corto, "ab".
```

```
Entrada: ["bufanda"]
Salida: "bufanda"
Explicación: Caso especial donde el prefijo común es la cadena completa.
```

```
Entrada: ["", "abc"]
Salida: ""
Explicación: La presencia de una cadena vacía hace que no haya prefijo común.
```

```
Entrada: ["sol", "solo", "solido"]
Salida: "sol"
Explicación: El prefijo común coincide exactamente con el primer elemento.
```

```
Entrada: ["abc", "abc", "abc"]
Salida: "abc"
Explicación: Todas las cadenas son iguales.
```

```
Entrada: ["a", "a", "b"]
Salida: ""
Explicación: Todas son cadenas muy cortas, donde algunas coinciden pero no todas.
```

```
Entrada: ["", "", ""]
Salida: ""
Explicación: Todas son cadenas vacías.
```



## 2.4. Ayuda

- Podrías comparar carácter por carácter en la misma posición para todas las cadenas, deteniéndote en la primera discrepancia que encuentres.
- Una estrategia sería usar la cadena más corta como referencia y verificar cuántos caracteres iniciales coinciden en todas las demás elementos del arreglo.

## 2.5. Del análisis a la solución, paso a paso

A continuación se presentan distintos enfoques progresivos. Los primeros intentos pueden ser ineficientes o incompletos; el objetivo es mostrar posibles etapas del razonamiento hasta llegar a una solución final validada como aceptable.

### 2.5.1. Comprensión del problema

Un "prefijo" es la **parte inicial** de una palabra que comienza desde el primer carácter. Por ejemplo, los prefijos de "flor" son: “","f”, "fl", "flo" y "flor" (la cadena vacía y la palabra completa también son prefijos válidos).

El término "común" significa que el prefijo debe estar presente en **todas las cadenas del arreglo**. No es suficiente con que aparezca en la mayoría: si al menos una cadena no lo contiene, ese prefijo no es válido.

Por tanto, el objetivo es encontrar el prefijo común más largo: el **mayor tramo inicial de caracteres que compartan todas las palabras**. Es importante destacar que:

- Si una cadena es más corta que las demás, el prefijo común no puede ser más largo que ella.
- Si las cadenas no comparten ningún carácter inicial, el prefijo común es la cadena vacía “”.
- El prefijo debe ser continuo desde el inicio: no se pueden "saltar" caracteres.

Algunos ejemplos:

- Para `["flor", "fluir", "flotar"]`, el prefijo común más largo es "fl" (todas empiezan con "fl").
- Para `["perro", "gato", "ave"]`, no hay prefijo común (ni siquiera comparten la primera letra), por lo que la respuesta es “”.
- Para `["auto", "autopista", "autobus"]`, el prefijo común más largo es "auto".

La clave del problema está en determinar hasta dónde se puede extender el prefijo manteniendo la coincidencia en todas las cadenas.

### 2.5.2. Primer enfoque (intento inicial)

*Esta sección está disponible en la versión completa del libro.*

### 2.5.3. Segundo enfoque (intento de mejora)

*Esta sección está disponible en la versión completa del libro.*

### 2.5.4. Solución validada

*Esta sección está disponible en la versión completa del libro.*

## 2.6. Complejidad algorítmica de la solución validada

*Esta sección está disponible en la versión completa del libro.*

## 2.7. Conclusiones técnicas

*Esta sección está disponible en la versión completa del libro.*

## 2.8. Implementación de la solución validada

*Código disponible en la versión completa, según edición (Python / C# / Java / Multilenguaje).*

## 2.9. Otra alternativa

*Enfoques alternativos disponibles en la versión completa.*

# ¿Qué incluye la versión completa?

Cada uno de los 24 ejercicios incluye:

- Análisis completo desde enfoques básicos hasta soluciones óptimas
- Justificación de cada decisión algorítmica
- Análisis de complejidad temporal y espacial
- Implementación en Python, Java y C# (según la edición)
- Enfoques alternativos y comparaciones

📖 290+ páginas (ediciones mono-lenguaje) / 380+ páginas (edición multilenguaje)

**Disponible en PDF en https://payhip.com/b/WSi0z**