Resumen Anotado: Pseudocódigo y Pruebas de Escritorio

Brayan Stiven Ortiz Medina

16/09/2025

Índice

1. Definiciones

1.1. Pseudocódigo

El pseudocódigo es una herramienta fundamental para la programación, ya que permite representar algoritmos de manera sencilla sin necesidad de dominar un lenguaje formal. Puede entenderse como un puente entre el pensamiento lógico y la codificación real. Su objetivo es expresar con claridad la secuencia de pasos que resuelven un problema y, al mismo tiempo, facilitar la comunicación entre personas con diferentes niveles de conocimiento en programación.

En la guía se resalta que:

- Todo pseudocódigo comienza con la palabra reservada **INICIO** y finaliza con **FIN**.
- Se emplean palabras reservadas en mayúsculas como **LEER**, **ESCRIBIR**, **SI**, **MIENTRAS**.
- Es posible definir variables con tipos de datos como **ENTERO**, **REAL**, **BOOLEANO** o **CADENA**.
- La claridad es esencial, por lo que se recomienda usar sangrías, orden visual y un lenguaje cercano al natural.

Ejemplo: Un pseudocódigo que solicita al usuario un número y lo muestra:

```
INICIO
num: ENTERO
ESCRIBIR "Ingrese un número"
LEER num
ESCRIBIR "El número ingresado es:", num
FIN
```

Este sencillo ejemplo refleja la estructura clásica: entrada, procesamiento mínimo y salida.

1.2. Pruebas de escritorio

Las pruebas de escritorio consisten en simular manualmente la ejecución de un algoritmo, llenando una tabla donde se registran los valores de las variables en cada paso. Según la guía, su importancia radica en que permiten validar si un algoritmo produce el resultado esperado antes de implementarlo. Ayudan a detectar errores de lógica, a confirmar que el flujo de decisiones es el correcto y a anticipar problemas antes de codificar. Se convierten, por tanto, en un mecanismo de verificación inicial muy accesible.

Ejemplo tomado de la guía: Determinar si un número es positivo o negativo.

Iteración	X	Salida
1	5	El número es positivo
2	-29	El número es negativo
3	100	El número es positivo

Este tipo de prueba garantiza que el algoritmo clasifica correctamente valores positivos y negativos, y al mismo tiempo evidencia la necesidad de decidir cómo manejar el caso cero. Además, permite mostrar con transparencia la evolución de las variables, algo que resulta clave para estudiantes y programadores principiantes.

2. Análisis y comentario de ejemplos

2.1. Ejemplo 1: Determinar si un número es positivo o negativo

Análisis: El algoritmo se apoya en una condición sencilla para clasificar un número. Es directo, aunque deja dudas respecto a cómo manejar el cero y carece de validación sobre la entrada. Es útil como introducción a las estructuras condicionales, ya que enseña a pensar en caminos alternativos según el valor ingresado. También refleja la importancia de considerar casos límite que, aunque parecen menores, pueden comprometer la utilidad del programa.

Mejoras: Incluir un caso específico para el cero, añadir validaciones que eviten datos no numéricos y diseñar mensajes claros que guíen al usuario. Así se logra que el algoritmo sea más robusto, fácil de usar y aplicable en más contextos.

2.2. Ejemplo 2: Obtener el mayor de dos números

Análisis: Su lógica consiste en comparar dos valores y mostrar el mayor. La restricción de que no sean iguales limita su aplicabilidad. Es un ejemplo adecuado para comprender decisiones binarias, aunque poco flexible. Este tipo de algoritmo permite trabajar el razonamiento lógico y la construcción de condiciones compuestas, pero debe ajustarse para resultar útil en escenarios reales.

Mejoras: Permitir la igualdad e informar al usuario, validar que ambos datos sean numéricos y, de ser necesario, ampliar la idea para trabajar con más de dos números. Con estas mejoras se enriquece su alcance, permitiendo que el mismo esquema resuelva problemas más amplios y realistas.

2.3. Ejemplo 3: Calcular el factorial de un número

Análisis: Se apoya en la repetición de multiplicaciones sucesivas para obtener el factorial. Sirve para practicar ciclos y acumuladores, pero presenta limitaciones si la entrada es negativa o demasiado grande. Además, es un buen ejemplo de cómo los algoritmos sencillos pueden convertirse en una puerta de entrada a conceptos más avanzados, como eficiencia computacional y crecimiento exponencial de resultados.

Mejoras: Validar que el número sea un entero no negativo, advertir sobre límites de cálculo y usar métodos más eficientes o bibliotecas especializadas en caso de números muy grandes. También puede incluirse un sistema de retroalimentación al usuario, que advierta sobre los riesgos de cálculos excesivamente grandes o sugiera límites razonables.

3. Conclusión

Los ejemplos analizados muestran cómo estructuras básicas como condicionales y ciclos pueden convertirse en piezas clave para resolver problemas comunes. El pseudocódigo y las pruebas de escritorio son recursos que permiten abstraer la lógica de un problema y validar la solución antes de codificarla. Mejorar los algoritmos no solo significa optimizarlos, sino también hacerlos más claros, inclusivos en sus casos de uso y resistentes a errores, lo cual resulta esencial en cualquier proceso de programación.

4. Referencias

Manual de Prácticas del Laboratorio de Fundamentos de Programación (MADO-17, Versión 05), Facultad de Ingeniería, 22/enero/2025. Secciones: *Solución de Problemas y Algoritmos* (pp. 36–51).