Programación Básica para Ingeniería en Sistemas de Información:
De los Sistemas de Información al Código

Gleiston Guerrero Ulloa

Índice general

Prefacio

Parte I

Fundamentos de Computación, Sistemas de Información y Algoritmos

Capítulo 1

Computadoras, Información y Sistemas de Programación

1.1. Ingeniería en Sistemas de Información y el rol de la programación

La Ingeniería en Sistemas de Información se ocupa del estudio, diseño, implementación y gestión de sistemas sociotécnicos que integran personas, procesos y tecnologías con el propósito de optimizar el flujo de información y apoyar la toma de decisiones organizacionales [1], [2]. Desde la perspectiva de obras fundamentales de computación [3], un sistema informático comprende hardware, software y datos, mientras que un sistema de información incorpora además procedimientos, políticas y factores humanos que condicionan su uso efectivo. En este marco, la programación se considera un mecanismo esencial para traducir requerimientos funcionales en procedimientos formales ejecutables por computadoras, permitiendo la automatización de tareas, la reducción de errores y la mejora de la eficiencia operativa [4], [5]. La capacidad de especificar algoritmos y construir software se convierte así en un elemento indispensable para materializar soluciones que respondan a necesidades reales de las organizaciones.

Asimismo, el rol de la programación dentro de la Ingeniería en Sistemas de Información se fundamenta en su potencial para modelar la lógica de los procesos y representar digitalmente las operaciones que se ejecutan en un entorno corporativo. Tal como señalan manuales y artículos técnicos [5], [6], un programa consiste en una secuencia de instrucciones que un procesador interpreta mediante ciclos de lectura, decodificación y ejecución, siendo este principio la base sobre la cual se construyen sistemas de gestión, plataformas empresariales

y aplicaciones orientadas al negocio [7]. La programación, por tanto, no solo habilita el funcionamiento técnico del sistema, sino que también estructura la interacción entre los distintos componentes organizacionales mediante reglas lógicas claras y reproducibles [7], [8].

1.1.1. La programación como competencia profesional

En el contexto profesional, la programación constituye una competencia central que habilita a los ingenieros de sistemas para diseñar soluciones computacionales alineadas con las necesidades estratégicas de la organización. Los programadores no se limitan a escribir código, sino que analizan problemas, diseñan modelos conceptuales y transforman esos modelos en implementaciones precisas y optimizadas. Esto encaja con la visión contemporánea de la ingeniería de software como motor de la transformación digital y la innovación en las organizaciones [9], [10].

Desde la práctica profesional, la programación exige habilidades de abstracción, pensamiento algorítmico, evaluación de alternativas tecnológicas y dominio de lenguajes formales que permitan expresar con claridad la solución propuesta. Esta competencia implica comprender cómo los datos fluyen a través de los sistemas, cómo se gestionan los recursos computacionales y cómo se implementan estructuras que garanticen robustez, escalabilidad y mantenibilidad. De este modo, el profesional se posiciona para traducir requerimientos operativos en sistemas fiables y sostenibles en el tiempo [11], [12].

Finalmente, la programación se consolida como una competencia estratégica debido a su papel en la innovación tecnológica. La capacidad para desarrollar software orientado a necesidades específicas permite crear ventajas competitivas, incorporar nuevas modalidades de servicio y adaptar organizaciones a entornos digitales en constante evolución [13], [14]. Así, la programación no solo responde a demandas existentes, sino que posibilita la creación de nuevas oportunidades en áreas como automatización, analítica, inteligencia artificial, sistemas distribuidos y modelos de negocio digitales.

1.1.2. Relación entre procesos organizacionales y software

Los procesos organizacionales describen la secuencia de actividades que transforman insumos en productos o servicios, y el software constituye el medio principal para estructurar, automatizar y controlar dichos procesos. En obras clásicas y recientes sobre gestión de procesos e integración empresarial, se reconoce que los sistemas de información son efectivos

cuando reflejan fielmente los flujos operativos de la organización, asegurando que la captura, procesamiento y salida de datos correspondan a la realidad empresarial [15], [16].

El software implementa reglas, restricciones y secuencias lógicas que garantizan que los procesos se ejecuten de manera consistente, uniforme y verificable. La correspondencia entre software y procesos requiere un análisis detallado de las actividades humanas, los puntos de decisión, las dependencias entre tareas y los mecanismos de control. Una vez modelados estos elementos (como en el contexto de Business Process Management – BPM), el software se convierte en la herramienta que soporta su ejecución digital, reduciendo tiempos, minimizando errores y permitiendo trazabilidad [17], [18]. Esto es evidente en sistemas como control de inventarios, nómina, admisión académica o gestión de ventas, donde el software representa y ejecuta reglas de negocio de forma automática.

Además, el software facilita la integración entre procesos que anteriormente funcionaban de forma aislada. Gracias a la conexión de módulos y bases de datos, los sistemas permiten compartir información entre departamentos y sincronizar operaciones en tiempo real. Este alineamiento entre tecnología y operación genera mejoras sustanciales en la eficiencia global, en la capacidad de respuesta ante cambios y en la calidad del servicio ofrecido por la organización [15], [19], [20].

1.2. El rol del programador en el desarrollo de sistemas de información

El programador cumple un rol fundamental en el desarrollo de sistemas de información al actuar como el puente entre los requerimientos del negocio y la implementación técnica. Su responsabilidad no se limita a codificar, sino que incluye analizar, interpretar y traducir reglas de negocio en instrucciones precisas que serán ejecutadas por la computadora. En efecto, un ingeniero de software aplica principios de ingeniería para diseñar, desarrollar y mantener sistemas de software que satisfacen necesidades reales de organizaciones [3], [21].

Asimismo, el programador participa activamente en la validación y mejora de los procesos organizacionales al identificar inconsistencias, redundancias y oportunidades de optimización durante el desarrollo del sistema. La implementación técnica obliga a examinar cada paso del proceso, lo que permite proponer mejoras basadas en la capacidad del software para gestionar flujos de trabajo de forma más eficiente, segura y consistente [19], [22].

Finalmente, el programador es responsable de garantizar la calidad del sistema mediante la aplicación de buenas prácticas de diseño, pruebas, documentación, refactorización y

mantenimiento. Su intervención asegura que el software sea confiable, extensible y alineado con los objetivos institucionales, contribuyendo con ello al funcionamiento integral del sistema de información [23], [24].

1.2.1. Automatización, digitalización y transformación digital

La automatización se refiere a la sustitución de tareas manuales por procedimientos ejecutados por computadoras mediante el uso de algoritmos, sensores, aplicaciones o sistemas automáticos. En este contexto, la automatización permite ejecutar operaciones repetitivas con alta velocidad, precisión y consistencia, reduciendo errores humanos y aumentando la productividad [25], [26]. Este enfoque se aplica en dominios como facturación, control de inventario, registros institucionales o procesamiento de transacciones.

La digitalización implica convertir objetos físicos o procesos manuales en representaciones digitales manipulables por sistemas computacionales. Su objetivo es permitir que la información fluya a través de plataformas tecnológicas, eliminando la dependencia del papel y habilitando nuevas formas de interacción y análisis. Por ejemplo, la digitalización de documentos permite su almacenamiento, búsqueda y procesamiento automático, mientras que la digitalización de procesos facilita su integración en sistemas más amplios [27], [28].

Por su parte, la transformación digital constituye un cambio organizacional profundo que integra tecnología, estrategias de innovación y rediseño de procesos para mejorar significativamente el desempeño institucional. Esta transformación requiere no solo digitalizar información o automatizar tareas, sino reimaginar modelos de operación, aprovechar capacidades analíticas y utilizar el software como motor de innovación. En este contexto, la programación y la ingeniería de software se convierten en componentes críticos para materializar soluciones que soporten nuevos modelos de negocio, experiencias mejoradas para el usuario y toma de decisiones basada en datos [9], [25], [29].

En síntesis, la evolución desde automatización hacia digitalización y transformación digital representa una progresión donde la tecnología deja de ser un soporte operativo para convertirse en un elemento estratégico de la organización. El ingeniero de sistemas participa activamente en este proceso mediante el desarrollo de software, la integración de plataformas y la implementación de soluciones que respondan a los desafíos del entorno digital [30], [31].

1.3. Datos, información y conocimiento

Los datos, la información y el conocimiento constituyen niveles complementarios dentro del ciclo de procesamiento que permite que los sistemas computacionales aporten valor. Los datos corresponden a representaciones simbólicas sin interpretación, la información emerge cuando estos datos adquieren significado en un contexto, y el conocimiento se genera cuando la información se asimila, estructura y utiliza para la toma de decisiones. Autores como Gaddis [32] enfatizan que la computación tiene como propósito central transformar datos en información útil mediante operaciones de entrada, procesamiento, almacenamiento y salida.

1.3.1. Definiciones fundamentales

Los datos se definen como símbolos o valores que representan hechos, medidas u observaciones, pero que por sí mismos carecen de significado. Pueden adoptar la forma de números, letras, sonidos, imágenes o señales capturadas en bruto por sensores o dispositivos de entrada [33], [34]. Su valor depende del modo en que se organizan, almacenan y procesan para cumplir un propósito específico dentro del sistema.

La información surge cuando los datos son procesados, estructurados o contextualizados de manera que adquieren relevancia para un usuario o una actividad. Este procesamiento puede incluir cálculos, clasificaciones, comparaciones, transformaciones o agregaciones [35]-[37]. La información constituye la base para la toma de decisiones y la ejecución de acciones dentro de una organización.

Por tanto, la distinción entre dato e información no es mera formalidad: señala la transición entre materia prima (datos) y producto útil (información) — un proceso que depende de métodos de almacenamiento, procesamiento, reglas, contexto y estructuras semánticas. Este entendimiento es clave en el diseño de sistemas de información eficientes y confiables.

Dato como representación simbólica

Los datos constituyen la forma más elemental de representación de hechos dentro de un sistema computacional. Según los fundamentos de almacenamiento digital y representación de datos en computación, un dato puede expresarse como un número, una letra, un símbolo o un patrón binario que el sistema aún no ha interpretado [38], [39]. En esta etapa, los datos no poseen un significado por sí mismos; simplemente existen como elementos aislados

que esperan ser procesados. Por ejemplo, el valor `45`, el carácter `A` o una cadena como `2025-03-10` son datos sin contexto, pero pueden adquirir interpretación cuando se insertan en un proceso específico, como un sistema de inventario o un registro académico.

Asimismo, los datos se caracterizan por su capacidad de ser capturados, almacenados y transmitidos mediante diferentes dispositivos computacionales. Los sistemas informáticos representan los datos en estructuras binarias o en tipos de datos definidos, almacenándolos en memoria principal o secundaria, lo que permite su manipulación sistemática [39], [40]. Este ciclo evidencia que los datos funcionan como la materia prima indispensable para generar información y conocimiento en entornos digitales, tras un proceso de interpretación, estructuración o contexto que les da significado [41], [42].

Información: contexto y significado

La información se genera a partir de los datos cuando estos son organizados, procesados o contextualizados para responder a una necesidad concreta. En marcos teóricos ampliamente aceptados, se enfatiza que la información es el resultado de aplicar operaciones tales como clasificación, comparación, cálculo o interpretación sobre los datos crudos [43]-[45]. Por ejemplo, una lista de números puede convertirse en información cuando se calcula un promedio, se ordena la secuencia o se determina la variación entre elementos. Así, la información aporta significado y facilita la comprensión de fenómenos o situaciones específicas.

El valor de la información está estrechamente ligado al contexto. El mismo dato puede ser irrelevante en un entorno, pero crucial en otro. Por ejemplo, el número `98` podría no comunicar nada por sí mismo; sin embargo, si se interpreta como la calificación promedio de un estudiante o como la temperatura corporal de un paciente, adquiere sentido inmediato. Este fenómeno coincide con la discusión epistemológica sobre la transformación de datos en información en función del contexto, la estructura y el uso previsto [44], [46].

Finalmente, la información se convierte en un recurso estratégico para organizaciones gracias a su capacidad de apoyar decisiones, automatizar procesos, mejorar la eficiencia organizacional y orientar políticas o acciones informadas. En entornos digitales y de sistemas de información, la generación de información pertinente depende de la calidad de los datos, la adecuación de los algoritmos de procesamiento y la correcta implementación del software que transforma esos datos en insumos útiles para la toma de decisiones [45], [47]. .

Conocimiento: reglas, estructuras y experiencia

El conocimiento representa el nivel más sofisticado dentro del ciclo de transformación de datos. Surge cuando la información es interpretada, conectada con experiencias previas y organizada en estructuras que permiten comprender situaciones, anticipar problemas y tomar decisiones [43], [48]. Aunque los sistemas computacionales pueden almacenar y procesar grandes volúmenes de información, su capacidad de generar conocimiento depende de modelos diseñados por seres humanos o de técnicas avanzadas como aprendizaje automático o sistemas inteligentes [49]. En este sentido, el conocimiento implica integración, generalización y aplicación real de la información.

En ingeniería de sistemas, el conocimiento se manifiesta como reglas de negocio, políticas organizacionales, modelos de decisión o patrones de comportamiento que guían el funcionamiento del software. Por ejemplo, un sistema académico no solo almacena datos de estudiantes e información de calificaciones; también incorpora conocimiento institucional al aplicar reglas como "aprobar con mínimo 7/10", "permitir matrícula si no existen deudas" o "generar alertas si el estudiante reprueba tres asignaturas consecutivas". Estas reglas, diseñadas a partir de experiencia y normativas, permiten que el sistema actúe de manera coherente y confiable [50].

El conocimiento también se asocia con estructuras mentales y modelos que permiten interpretar la información de manera más profunda. Mientras la información responde al "qué", el conocimiento responde al "por qué" y "para qué". De esta forma, un sistema de información no solo debe presentar datos, sino proveer mecanismos para generar conocimiento útil para gestores, analistas y tomadores de decisiones [35], [51].

La literatura identifica el conocimiento como algo más que datos procesados: es una entidad emergente que requiere interpretación, contexto, experiencia o reglas explícitas — atributos humanos o estructurales — para existir. Por ello, la calidad de los datos y la pertinencia del procesamiento influyen directamente en la confiabilidad del conocimiento generado [44], [48]. Finalmente, el conocimiento es esencial para retroalimentar sistemas inteligentes y procesos de mejora continua dentro de organizaciones modernas. Un sistema que aprende de errores, patrones o tendencias puede evolucionar y adaptarse a nuevas condiciones, convirtiéndose en un activo estratégico para la institución [47].

1.3.2. Flujo de datos y transformación

El flujo de datos describe el movimiento de la información desde su origen hasta su utilización final dentro de un sistema computacional. De acuerdo con los modelos de

procesamiento estudiados en textos introductorios [52], los datos ingresan al sistema a través de dispositivos de entrada, son transformados mediante operaciones lógicas y aritméticas, se almacenan temporalmente y finalmente se presentan como información útil. Este flujo garantiza que el sistema pueda operar de manera continua y coherente.

Durante la transformación, los datos pueden ser validados, normalizados, agrupados o convertidos en otros formatos. Por ejemplo, una fecha ingresada como `10-03-25` puede convertirse en `2025-03-10` para garantizar uniformidad. Asimismo, valores numéricos pueden ser sumados, promediados o comparados para generar indicadores útiles. Estas transformaciones permiten adaptar los datos a los requerimientos de los procesos internos.

El análisis del flujo de datos es fundamental en el diseño de sistemas porque permite identificar posibles redundancias, cuellos de botella o riesgos de inconsistencia. Una representación adecuada del flujo garantiza que la información llegue oportunamente a los usuarios o módulos que la requieren, evitando pérdidas o duplicación de datos y mejorando la eficiencia global del sistema.

Procesos de captura, procesamiento y salida

El proceso de captura consiste en recoger datos desde fuentes internas o externas mediante dispositivos como teclados, sensores, micrófonos, cámaras o sistemas conectados. Según [52], la captura debe asegurar precisión, completitud y coherencia para evitar errores en etapas posteriores. Un ejemplo común es el registro de productos en un sistema de inventarios mediante un lector de código de barras, que garantiza rapidez y precisión en la recolección de datos.

El procesamiento involucra transformaciones lógicas y aritméticas que convierten los datos capturados en información con significado para los usuarios. Estas operaciones pueden incluir sumar, restar, ordenar, clasificar, comparar o aplicar fórmulas más complejas. Por ejemplo, en un sistema académico, el cálculo automático del promedio de un estudiante a partir de sus calificaciones constituye un proceso típico de procesamiento.

La salida corresponde a la presentación de la información procesada al usuario o a otro sistema. Puede materializarse mediante reportes en pantalla, documentos impresos, gráficos o almacenamiento estructurado. Ejemplos comunes incluyen la generación de facturas, la actualización de dashboards o la emisión de certificados. De acuerdo con [32], los dispositivos de salida como monitores e impresoras traducen la información digital en representaciones comprensibles para el usuario.

Criterios de calidad de datos

La calidad de los datos se evalúa mediante diversos criterios, entre ellos precisión, completitud, consistencia, actualidad y validez. A continuación, se desarrolla cada criterio con su respectivo ejemplo.

Precisión. La precisión se refiere a que los datos reflejen correctamente la realidad. Por ejemplo, registrar que un estudiante obtuvo `85` en un examen cuando realmente obtuvo `58` constituye un dato impreciso que afectará todos los cálculos posteriores. Según [52], la precisión es esencial porque errores pequeños pueden amplificarse durante el procesamiento.

Completitud. Se refiere a que no falten datos necesarios para procesar la información. Un registro de cliente sin número de identificación o sin dirección puede impedir la validación de operaciones o la entrega de productos. La falta de completitud ocasiona fallos en transacciones o bloqueos en procesos de negocio.

Consistencia. Los datos deben mantener coherencia entre diferentes fuentes o módulos del sistema. Por ejemplo, si un empleado aparece con salario `1200` en el módulo de nómina y `1500` en el módulo de recursos humanos, existe inconsistencia. Esta situación puede deberse a errores de actualización o duplicación de registros.

Actualidad. Los datos deben encontrarse actualizados para reflejar correctamente el estado del sistema. Una tarifa de servicio desactualizada puede generar cobros incorrectos. Según [32], la actualidad es crítica en sistemas de comercio y banca, donde los datos cambian frecuentemente.

Validez. Los datos deben cumplir reglas específicas o restricciones definidas por la organización. Por ejemplo, una fecha de nacimiento futura como `2030-05-10` no es válida en un registro personal. La validez garantiza que los datos se ajusten a dominios y formatos aceptables.

1.3.3. Datos en sistemas computacionales

Los datos almacenados en sistemas computacionales están sujetos a restricciones propias del hardware y del software que los manipulan. En términos de representación digital, todos los datos deben convertirse a secuencias de bits, tal como se explica en la literatura de fundamentos de programación y arquitectura [32]. Esta representación binaria permite que el procesador interprete, almacene y transforme los datos mediante operaciones lógicas y aritméticas. En un sistema computacional, los datos pueden corresponder a valores numéricos, texto, imágenes, audio o cualquier forma de información digital.

El manejo de datos dentro del sistema implica su ubicación en memoria primaria, su

posible transferencia a memoria secundaria y su uso durante la ejecución de programas. Los textos de introducción a la computación [52] destacan que los datos fluyen entre CPU, memoria y dispositivos de entrada/salida mediante buses especializados, asegurando una comunicación eficiente entre los componentes del sistema. La forma en que se organizan estos datos influye en la eficiencia, seguridad y consistencia del sistema.

Finalmente, los datos en sistemas computacionales se encuentran estructurados según tipos específicos definidos por el lenguaje de programación y por el sistema operativo. Estos tipos permiten especificar el tamaño, formato y operaciones válidas para cada dato, lo que garantiza integridad y minimiza errores durante la ejecución. Esta estructura formal es esencial para el diseño de algoritmos robustos y sistemas confiables.

Representación interna

La representación interna de los datos en una computadora se basa en el sistema binario, donde cada valor se almacena como una combinación de bits. De acuerdo con [32], un byte está compuesto por ocho bits, y cada patrón binario representa un número, un carácter o una instrucción dependiendo del contexto. Por ejemplo, el carácter `A` se representa mediante el código ASCII `65`, cuyo equivalente binario es `01000001`. Este proceso permite que la información textual sea manejada de manera uniforme.

Asimismo, los datos numéricos pueden representarse mediante diferentes codificaciones internas. Los enteros utilizan frecuentemente la representación en complemento a dos para manejar valores positivos y negativos, mientras que los números reales se representan mediante notación de punto flotante basada en el estándar IEEE 754. Este método divide el número en signo, exponente y mantisa, permitiendo una representación amplia pero aproximada del valor real. Como señalan los manuales de arquitectura computacional, la precisión depende directamente del número de bits asignados.

Finalmente, la representación interna también afecta la forma en que los programas interpretan y manipulan los datos. Un conjunto de bits puede corresponder a un número, un carácter o una instrucción dependiendo del módulo que lo interprete. Esta versatilidad es fundamental para la ejecución de programas, pero requiere disciplina en su manejo para evitar errores de interpretación y corrupción de datos.

Persistencia y acceso

La persistencia de datos se refiere a su capacidad de mantenerse almacenados incluso cuando el sistema se apaga. Los textos de fundamentos [32] identifican los discos duros,

unidades de estado sólido y memorias flash como medios principales para lograr esta finalidad. A diferencia de la memoria RAM, que es volátil, los dispositivos de almacenamiento secundario preservan los datos utilizando tecnologías magnéticas, ópticas o electrónicas. Esto permite que los programas y documentos permanezcan disponibles a largo plazo.

El acceso a los datos puede realizarse de manera secuencial o aleatoria. El acceso secuencial implica recorrer los datos en orden, como ocurre en cintas magnéticas, mientras que el acceso aleatorio permite recuperar cualquier dato directamente mediante su dirección, como en los discos modernos. La eficiencia del acceso influye en el rendimiento del sistema, especialmente en aplicaciones que requieren grandes volúmenes de lectura y escritura.

Finalmente, los mecanismos de persistencia y acceso se integran mediante sistemas de archivos, que organizan la información en directorios, bloques y metadatos. Estos sistemas, gestionados por el sistema operativo, garantizan integridad, seguridad y recuperación ante fallos. La correcta comprensión de estos mecanismos es esencial para el diseño de aplicaciones que gestionen datos de manera segura y eficiente.

1.4. La computadora como sistema

Una computadora puede entenderse como un sistema compuesto por elementos interrelacionados que trabajan de manera coordinada para procesar información. Según [32], este sistema incluye la unidad central de procesamiento (CPU), la memoria principal, los dispositivos de almacenamiento, los dispositivos de entrada y salida, y el software que gestiona la interacción entre los componentes. La computadora opera bajo un modelo secuencial de ejecución que permite transformar datos en resultados útiles para el usuario.

1.4.1. Arquitectura básica

La arquitectura básica de una computadora se fundamenta en el modelo de Von Neumann, que establece que tanto los datos como las instrucciones se almacenan en la misma memoria. Este enfoque permite que el procesador obtenga instrucciones y datos desde ubicaciones similares, simplificando el diseño del hardware. Los elementos principales incluyen la unidad de control, la unidad aritmético-lógica (ALU), los registros internos y la memoria principal.

En los textos de fundamentos de computación [52], se establece que la CPU coordina todas las operaciones, mientras que la memoria sirve como espacio temporal para guardar instrucciones y datos que están siendo procesados. Los dispositivos de entrada permiten

introducir datos al sistema y los dispositivos de salida permiten presentar resultados en forma visual, auditiva o impresa. Esta estructura modular facilita la expansión del sistema mediante nuevos dispositivos o componentes.

Finalmente, la arquitectura básica contempla buses que interconectan los componentes principales. El bus de direcciones especifica dónde se encuentran los datos, el bus de datos transporta la información y el bus de control coordina las señales necesarias para ejecutar operaciones. Estos elementos garantizan la integridad y sincronización del sistema.

CPU y unidad de control

La Unidad Central de Procesamiento (CPU) es el componente encargado de ejecutar instrucciones y coordinar las operaciones del sistema. Según [32], la CPU se compone principalmente de la unidad de control y la unidad aritmético-lógica (ALU). La unidad de control interpreta las instrucciones del programa, genera señales de control y dirige el flujo de datos dentro del sistema. Esta unidad determina qué operación debe realizarse en cada ciclo y qué componentes deben activarse.

La ALU, por su parte, realiza operaciones aritméticas como suma, resta, multiplicación y división, así como operaciones lógicas como comparación, conjunción y disyunción. Estas operaciones permiten que los programas ejecuten cálculos, evaluaciones de condiciones y manipulación de datos. La ALU trabaja estrechamente con los registros internos, que almacenan valores temporales necesarios para las operaciones.

Los registros son pequeñas ubicaciones de memoria extremadamente rápidas que permiten guardar direcciones, resultados intermedios e instrucciones. Entre ellos destacan el contador de programa (PC), que indica la siguiente instrucción a ejecutar, y el registro de instrucción (IR), que almacena la instrucción actual. Estos componentes permiten que el ciclo de ejecución se realice de manera eficiente.

Finalmente, la CPU ejecuta el ciclo de *fetch–decode–execute*, descrito ampliamente en la literatura [32]. En este ciclo, la CPU obtiene una instrucción desde memoria, la decodifica para determinar la operación requerida y luego la ejecuta. Este proceso se repite miles de millones de veces por segundo en los procesadores modernos.

Memoria principal

La memoria principal, conocida como RAM, almacena temporalmente los programas y datos que se encuentran en ejecución. Según [32], la RAM es volátil, lo que implica que

su contenido se pierde cuando el sistema se apaga. Su principal ventaja es la rapidez de acceso, ya que permite que la CPU lea y escriba datos de manera casi instantánea.

La memoria se organiza en celdas numeradas mediante direcciones, lo que permite recuperar datos específicos de manera eficiente. Cada celda almacena un byte y varias celdas pueden combinarse para representar valores más grandes. Esta organización facilita la ejecución de programas complejos que requieren acceso frecuente a diferentes porciones de memoria.

Por último, la capacidad y velocidad de la memoria influyen directamente en el rendimiento del sistema. Sistemas con mayor cantidad de RAM pueden ejecutar múltiples programas de manera simultánea y manejar conjuntos de datos más extensos. Por ello, la memoria es un componente crítico para aplicaciones científicas, empresariales y multimedia.

Dispositivos de entrada y salida

Los dispositivos de entrada permiten que el usuario o un entorno externo introduzcan datos en el sistema computacional; estos mecanismos incluyen teclados, ratones, escáneres, micrófonos y sensores especializados, cuya función es convertir fenómenos físicos en señales digitales interpretables por el procesador [53]. Cada dispositivo transforma información analógica o simbólica en datos binarios, asegurando que pueda ser procesada según el modelo de arquitectura de Von Neumann [54].

Por otro lado, los dispositivos de salida presentan información procesada al usuario en formatos visuales, impresos o auditivos. Pantallas, impresoras y proyectores traducen los resultados binarios en representaciones comprensibles mediante controladores especializados que ejecutan rutinas en coordinación con el sistema operativo [55]. La correcta operación de estos dispositivos depende de un flujo continuo entre CPU, memoria y buses, conforme describen los modelos clásicos de organización computacional [53].

Asimismo, muchos dispositivos cumplen funciones mixtas, como discos, memorias USB o redes, que pueden actuar simultáneamente como entrada y salida al transferir datos entre sistemas mediante protocolos estandarizados [56]. Esta característica permite la interoperabilidad y soporta sistemas distribuidos en los que la información fluye de manera bidireccional.

Almacenamiento

El almacenamiento secundario conserva datos de forma persistente aun cuando el sistema pierde energía, a diferencia de la memoria principal que es volátil [32]. Discos duros,

unidades de estado sólido y memorias flash emplean tecnologías magnéticas, electrónicas u ópticas para asegurar la retención de la información, siguiendo estándares como SATA, NVMe o USB [53]. La capacidad de almacenamiento y su velocidad influyen directamente en el rendimiento de sistemas de información, especialmente en aplicaciones de alta demanda como bases de datos o sistemas transaccionales [3].

Los modelos de dispositivos descritos en la literatura señalan que el acceso puede ser secuencial como en cintas magnéticas o aleatorio, como en los discos modernos, lo que permite recuperar cualquier bloque de datos mediante su dirección física o lógica [54]. Esta característica posibilita algoritmos de entrada/salida eficientes y estructuras de almacenamiento optimizadas como árboles B o índices hash.

Finalmente, los sistemas operativos gestionan el almacenamiento mediante sistemas de archivos, asignación de bloques, metadatos y mecanismos de journaling que permiten recuperar el estado del sistema después de fallos, tal como explican los textos canónicos de sistemas operativos [55].

1.4.2. Software del sistema y de aplicación

El software del sistema coordina los recursos computacionales y actúa como interfaz entre hardware y programas de usuario. Se compone principalmente del sistema operativo, controladores de dispositivos y utilidades que permiten gestionar procesos, memoria, almacenamiento y dispositivos externos [55]. Este conjunto de programas es esencial para que el hardware opere según lo previsto y para que los programas de aplicación se ejecuten correctamente.

Por su parte, el software de aplicación corresponde a programas diseñados para resolver necesidades específicas del usuario o de una organización. Procesadores de texto, hojas de cálculo, navegadores web y sistemas empresariales se incluyen en esta categoría, cada uno construido sobre abstracciones proporcionadas por el sistema operativo [3]. La estructura modular del software moderno facilita su reutilización y mantenimiento mediante principios de ingeniería de software [12].

La correcta interacción entre software del sistema y software de aplicación permite que la computadora funcione como un sistema coherente capaz de ejecutar múltiples tareas y gestionar recursos de forma segura y eficiente [53].

Sistema operativo

El sistema operativo es el componente esencial del software del sistema. Sus funciones incluyen administrar procesos, asignar memoria, controlar dispositivos, gestionar almacenamiento, proporcionar seguridad y ofrecer una interfaz entre el usuario y el hardware [55]. Sistemas ampliamente difundidos como Windows, Linux y macOS implementan estas funciones mediante arquitecturas de núcleo monolítico, híbrido o microkernel, cada una con ventajas particulares en rendimiento y modularidad [54].

Una tarea crítica del sistema operativo es el manejo de procesos, donde se administra su creación, suspensión, sincronización y terminación. Esta gestión se realiza mediante algoritmos de planificación como Round Robin, SJF o prioridades, los cuales determinan cómo se asigna el CPU a las tareas [53].

Asimismo, el sistema operativo gestiona el almacenamiento secundario mediante sistemas de archivos como NTFS, ext4 o APFS, asegurando integridad, eficiencia y control de acceso [55]. Cada uno utiliza estructuras diferentes para organizar directorios, bloques y metadatos, lo que afecta directamente el rendimiento del sistema.

Programas utilitarios

Los programas utilitarios realizan tareas especializadas que complementan las funciones del sistema operativo. Entre estas herramientas destacan los programas de compresión, antivirus, software de respaldo y herramientas de diagnóstico, que permiten optimizar el rendimiento y proteger la integridad del sistema [12]. A diferencia de las aplicaciones generales, los utilitarios se centran en tareas específicas de mantenimiento o soporte técnico.

Por ejemplo, los programas antivirus analizan patrones de código y comportamientos sospechosos mediante bases de datos actualizadas según modelos de detección descritos en la literatura de seguridad informática [53]. Igualmente, las herramientas de respaldo generan copias de seguridad periódicas para prevenir la pérdida de datos, siguiendo políticas de retención y estrategias como incremental, diferencial o completa [3].

Aplicaciones orientadas a negocio

Las aplicaciones orientadas a negocio permiten automatizar procesos empresariales tales como contabilidad, ventas, logística, recursos humanos y gestión de clientes. Estas aplicaciones se diseñan a partir de modelos de procesos de negocio y requisitos de usuario definidos mediante metodologías de ingeniería de software [12]. Su arquitectura puede incluir

módulos integrados, bases de datos relacionales y servicios distribuidos que garantizan consistencia e interoperabilidad.

Ejemplos comunes incluyen sistemas ERP, CRM y plataformas de comercio electrónico, que integran datos y procesos en una estructura centralizada gestionada por un motor transaccional [3]. Estas aplicaciones implementan reglas de negocio específicas que permiten controlar flujos de trabajo, validar transacciones y generar reportes estratégicos.

El éxito de estas aplicaciones depende de su alineación con los objetivos organizacionales y de su capacidad para adaptarse a cambios en el entorno empresarial. La literatura enfatiza que la mantenibilidad, escalabilidad y seguridad del software son elementos decisivos para su adopción efectiva [12].

1.4.3. Modelo de ejecución de un programa

El modelo de ejecución de un programa describe cómo la CPU obtiene instrucciones, las interpreta y las ejecuta según el ciclo de instrucción, un concepto fundamental en arquitectura computacional [54]. Toda ejecución comienza con la carga del programa en memoria principal, proceso gestionado por el sistema operativo mediante rutinas de asignación de memoria y creación de procesos [55]. Esta estructura garantiza que cada programa cuente con un espacio de direcciones protegido, propiedad descrita ampliamente en los modelos de gestión de memoria [53].

Durante la ejecución, la CPU sigue el ciclo clásico *fetch–decode–execute*, en el cual el contador de programa indica la dirección de la siguiente instrucción a recuperar [32]. Posteriormente, la instrucción se decodifica mediante la unidad de control, responsable de interpretar los códigos de operación definidos por la arquitectura del procesador [53]. Finalmente, la ALU o las unidades especializadas ejecutan la operación solicitada, ya sea aritmética, lógica o de transferencia de datos, siguiendo el conjunto de instrucciones documentado en microarquitecturas modernas [54].

Además, los sistemas operativos gestionan el estado del programa mediante contextos de ejecución que incluyen registros, pila, contador de programa y tabla de páginas [55]. Esto permite suspender y reanudar procesos, facilitando la multitarea y la ejecución concurrente. En sistemas multiprocesador y multinúcleo, la planificación se distribuye para mejorar el rendimiento, conforme a las técnicas de escalamiento descritas por Sommerville [3].

El correcto funcionamiento del modelo de ejecución depende de la integridad del código, del manejo adecuado de interrupciones y del uso eficiente de la jerarquía de memoria, cuya estructura se ha documentado de manera exhaustiva en la literatura de organización de

computadores [53]. Sin estos mecanismos, la ejecución confiable y segura sería imposible.

Etapas: código → compilación → ejecución

El proceso inicia con la escritura del código fuente en un lenguaje de alto nivel, el cual debe cumplir reglas sintácticas y semánticas documentadas por los lenguajes de programación [32]. Posteriormente, un compilador traduce dicho código a lenguaje máquina siguiendo análisis léxico, sintáctico y semántico, procesos descritos por los compiladores clásicos como los de Aho, Sethi y Ullman [54]. Esta traducción genera archivos objeto que contienen instrucciones binarias alineadas con la arquitectura del procesador [53].

Durante la fase de enlace, los módulos compilados se integran con bibliotecas y rutinas del sistema para producir un ejecutable, siguiendo las especificaciones del sistema operativo [55]. Cuando el usuario solicita la ejecución, el cargador (*loader*) ubica el ejecutable en memoria, prepara el contexto inicial del proceso y transfiere el control a la primera instrucción, conforme a los mecanismos descritos por Stallings [53].

Finalmente, el programa se ejecuta dentro del modelo de multitarea del sistema operativo, donde los planificadores asignan tiempo de CPU según algoritmos como Round Robin, prioridades o planificación multinivel [55]. Este flujo garantiza portabilidad, seguridad y desempeño, objetivos centrales en ingeniería de software [12].

Gestión de memoria y procesos

La gestión de memoria es una función crítica del sistema operativo y permite asignar, proteger y liberar espacio para programas en ejecución [55]. Técnicas como segmentación, paginación y memoria virtual posibilitan que los programas utilicen más memoria de la disponible físicamente, simulando un espacio unificado mediante tablas de páginas y mecanismos de intercambio (*paging*) [53]. Estos métodos aseguran aislamiento entre procesos, propiedad fundamental para la seguridad del sistema [54].

Por otra parte, la gestión de procesos controla su creación, suspensión y finalización mediante estructuras como PCB (*Process Control Block*), que almacenan estados de ejecución, registros, contadores de programa y descriptores de memoria [55]. Este control permite que múltiples programas coexistan de manera eficiente mediante planificación distribuida, un principio documentado en sistemas multiprocesador [53].

La sincronización entre procesos se implementa mediante semáforos, monitores o exclusión mutua, mecanismos ampliamente estudiados en los problemas clásicos de concurrencia

[54]. Sin estas herramientas, los procesos podrían interferir entre sí, produciendo condiciones de carrera, interbloqueos o inconsistencias lógicas.

1.5. Representación de la información en la computadora

La representación de la información en una computadora se basa en estructuras binarias gestionadas por el hardware, donde todo dato —números, caracteres, colores, sonidos o instrucciones— se codifica mediante patrones de bits que siguen reglas específicas para cada tipo de información [53], [54]. Estos patrones binarios deben ser interpretados por la arquitectura del computador, que define tamaños de palabra, formatos de almacenamiento, esquemas de direccionamiento y mecanismos de manipulación, garantizando que la información pueda procesarse con precisión y eficiencia [3], [12].

1.5.1. Unidades de información

Las unidades fundamentales para representar información en un sistema digital son el bit, el byte y la palabra, cada una con funciones específicas dentro de la arquitectura del computador [54]. Un bit (*binary digit*) es la unidad mínima de información, capaz de representar dos estados: 0 y 1, utilizados para modelar valores lógicos, señales electrónicas y condiciones booleanas [53]. Los bits son la base para cualquier codificación, y su combinación permite expresar cantidades más complejas. Por ejemplo, el patrón binario `1011` representa el número decimal 11 [32].

El byte está compuesto por 8 bits y constituye la unidad estándar para representar caracteres, pequeñas cantidades de datos y direcciones básicas de memoria [53]. Muchos sistemas contemporáneos consideran la palabra como una unidad compuesta por 16, 32 o 64 bits, dependiendo de la arquitectura, lo que determina la cantidad de datos que la CPU puede procesar en una única operación [54]. Por ejemplo, en arquitecturas de 64 bits, un entero con precisión estándar se almacena típicamente en una palabra de 64 bits, lo que permite representar un rango numérico significativamente mayor que en arquitecturas de 32 bits [12].

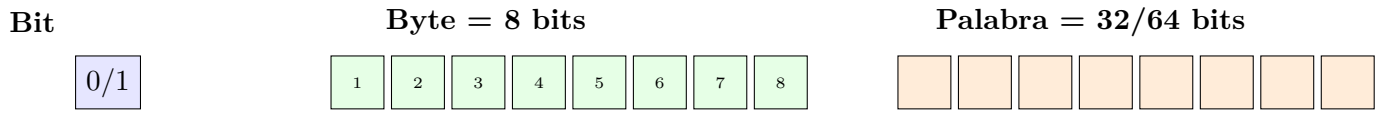

Figura 1.1: Representación de bit, byte y palabra en niveles crecientes de agrupación [53], [54].

Bit, byte y palabra

El bit es la unidad más elemental de información digital y sirve como base para la representación de todos los datos en la computadora [53]. Su estado puede modelar fenómenos físicos como presencia o ausencia de corriente, polaridad magnética o niveles de voltaje, dependiendo de la tecnología de hardware empleada [54]. Debido a su simplicidad y capacidad de combinación, los bits son elementos fundamentales para estructuras más complejas como registros, instrucciones y formatos multimedia [3].

El byte, compuesto por 8 bits, permite representar 256 combinaciones distintas (de 0 a 255 en decimal), siendo suficiente para codificar caracteres según estándares como ASCII o para representar colores básicos en modelos RGB [32]. Por ejemplo, el carácter ‘’A’‘ corresponde al valor decimal 65, cuyo equivalente binario es `01000001` [53]. El byte constituye una frontera natural para el direccionamiento de memoria en la mayoría de arquitecturas modernas [54].

La palabra, que puede ser de 16, 32 o 64 bits según la arquitectura, determina el ancho del bus de datos, el tamaño de los registros de la CPU y la cantidad de información que puede manipularse en una sola operación [53]. Por ejemplo, una palabra de 32 bits permite representar enteros hasta aproximadamente $2^{31} - 1$, mientras que una palabra de 64 bits incrementa ese límite a $2^{63} - 1$ [54]. Esta diferencia tiene impacto directo en el rendimiento y en la capacidad de direccionamiento del sistema [12].

Concepto de dirección de memoria

Una dirección de memoria es un identificador numérico que permite localizar un byte específico dentro del espacio de memoria del sistema [54]. Por ejemplo, si un arreglo comienza en la dirección `1000`, y cada elemento ocupa 4 bytes, entonces el quinto elemento se ubicará en la dirección `1000 + 4 * 4 = 1016` [53]. Este mecanismo de direccionamiento secuencial es esencial para el acceso eficiente a estructuras de datos y para la implementación de punteros en lenguajes como C o C++ [12].

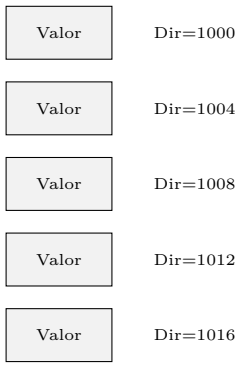

Figura 1.2: Direccionamiento secuencial en memoria para un arreglo de enteros de 4 bytes [53], [54].

1.5.2. Sistemas de numeración

Los sistemas de numeración permiten representar valores de forma simbólica según una base o radix que determina el número de dígitos disponibles [53]. En informática, los sistemas más utilizados son el decimal (base 10), el binario (base 2) y el hexadecimal (base 16), siendo el binario el utilizado internamente por las computadoras debido a su compatibilidad con circuitos digitales basados en dos estados [54]. Comprender estos sistemas es fundamental para interpretar direcciones de memoria, códigos de instrucción y formatos internos de datos [3].

La conversión entre sistemas numéricos constituye una habilidad esencial en programación y arquitectura de computadoras [32]. Por ejemplo, el número decimal 25 corresponde a `11001` en binario, y a `19` en hexadecimal [53]. Estas conversiones permiten interpretar adecuadamente valores internos y depurar programas que manipulan niveles bajos de representación [54].

Decimal

El sistema decimal, basado en diez símbolos (0–9), tiene sus raíces históricas en civilizaciones antiguas como los hindúes, quienes desarrollaron el concepto de valor posicional que luego sería difundido por los árabes hacia Occidente [53]. Este sistema utiliza potencias de 10 para determinar el valor de cada dígito según su posición, propiedad que facilita la representación de cantidades grandes y la enseñanza matemática [54]. Su uso extendido en la vida cotidiana lo convierte en el sistema principal para la interacción humano–computadora [3].

En programación, los valores numéricos suelen ingresarse y mostrarse en decimal para

facilitar su comprensión, aunque internamente se almacenen en binario [32]. Por ejemplo, el número decimal 47 se interpreta como $4 \times 10^1 + 7 \times 10^0$ [53]. Esta interpretación posicional es la base para convertir números decimales a otras bases [54].

La literatura destaca que, aunque el decimal no se utilice internamente para cálculos en hardware, comprender su relación con el binario y el hexadecimal es esencial para interpretar conversiones, depurar programas y trabajar con formatos de datos multinivel [12]. Esta comprensión también es importante para representar cantidades en sistemas financieros, científicos y estadísticos [3].

Binario

El sistema binario utiliza dos dígitos (0 y 1), lo que lo hace ideal para representar estados físicos de circuitos digitales [53]. Cada posición de un número binario representa una potencia de 2, de modo que el valor de un número binario se obtiene sumando las potencias correspondientes a los dígitos 1 presentes [54]. Por ejemplo, `10110` equivale a $1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 22$ en decimal [32].

La conversión de decimal a binario puede hacerse mediante divisiones sucesivas entre 2, registrando los residuos, mientras que la conversión inversa se basa en expansiones posicionales [3]. Por ejemplo, para convertir el decimal 13 a binario: 13/2=6 r1, 6/2=3 r0, 3/2=1 r1, 1/2=0 r1, obteniendo `1101` al leer los residuos en orden inverso [53]. Este método permite comprender la relación entre valores y representaciones digitales [54].

El binario se utiliza ampliamente para representar contenido computacional como máscaras de bits, permisos, colores, direcciones de memoria y códigos máquina [12]. Asimismo, la representación de enteros con signo (complemento a dos), números reales (IEEE 754) y caracteres (ASCII, Unicode) utiliza internamente formatos binarios específicos [3]. Por ello, el dominio del sistema binario es esencial para cualquier profesional de computación [53].

Hexadecimal

El sistema hexadecimal utiliza dieciséis símbolos (0–9 y A–F), lo que permite representar grandes valores en pocas posiciones y facilita la lectura de patrones binarios [53]. Cada dígito hexadecimal equivale exactamente a 4 bits, lo que permite convertir entre hexadecimal y binario agrupando bits en conjuntos de cuatro [54]. Por ejemplo, `A3` equivale a `1010 0011` en binario [32].

La conversión de hexadecimal a decimal se basa en potencias de 16. Por ejemplo, `2F` equivale a $2 \times 16^1 + 15 \times 16^0 = 47$ [53]. De binario a hexadecimal, basta agrupar los bits

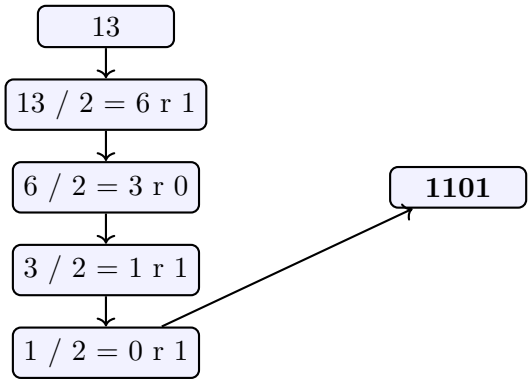

Figura 1.3: Conversión de número decimal a binario mediante divisiones sucesivas [53].

en grupos de cuatro desde la derecha; por ejemplo, `11010111` se agrupa como `1101 0111`, que corresponde a `D7` [54]. Esta correspondencia directa simplifica tareas de depuración y análisis de memoria [3].

La conversión de decimal a hexadecimal puede realizarse mediante divisiones sucesivas entre 16. Por ejemplo, para convertir 254: 254/16=15 r 14; así $15 = F$, $14 = E$, lo que produce `FE` [53]. La representación hexadecimal se emplea ampliamente en direcciones de memoria, códigos de color, instrucciones máquina y formatos binarios [12].

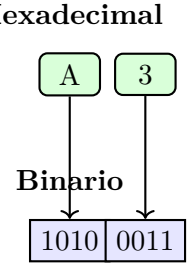

Figura 1.4: Conversión directa entre hexadecimal y binario agrupando bits en conjuntos de 4 [53], [54].

1.5.3. Representación de tipos básicos

Los tipos básicos de datos constituyen las unidades fundamentales para la representación y manipulación de información en los programas, y su codificación está estrechamente vinculada con la arquitectura del computador y los estándares internacionales de representación binaria [54]. Entre los tipos elementales se incluyen los enteros, los números reales, los caracteres y los valores lógicos, cada uno con reglas de codificación específicas que determinan su rango, precisión y comportamiento ante operaciones aritméticas y lógicas [53].

Comprender estos formatos es esencial para evitar errores sutiles como desbordamientos, pérdidas de precisión o resultados inesperados en comparaciones [3].

La representación interna de estos tipos se fundamenta en configuraciones binarias interpretadas bajo convenciones establecidas, como el complemento a dos para enteros con signo, el estándar IEEE 754 para números reales y los códigos ASCII o Unicode para caracteres [12]. Esta organización permite que las computadoras procesen información de forma uniforme, garantizando compatibilidad entre plataformas y lenguajes de programación [32]. El dominio de estas representaciones facilita el uso correcto de los tipos y promueve la escritura de programas más fiables y eficientes [3].

Enteros

Los enteros pueden representarse en formato sin signo o con signo, siendo este último manejado con frecuencia mediante el método de complemento a dos, que permite operar con valores negativos utilizando aritmética binaria estándar [53]. En un entero de 8 bits con complemento a dos, el rango va de –128 a 127, y el bit más significativo actúa como bit de signo [54]. Por ejemplo, el valor decimal –5 se representa como `11111011` en complemento a dos [32].

El complemento a dos simplifica el hardware al permitir que la suma y la resta se realicen de la misma manera tanto para números positivos como negativos [53]. Para obtener el complemento a dos de un número, se invierten sus bits y se suma uno. Así, el valor binario de 5 (`00000101`) se invierte (`11111010`) y se suma uno (`11111011`), obteniéndose la representación de –5 [54]. Este método evita las ambigüedades inherentes a otras formas de representar números negativos [3].

Los enteros también pueden causar desbordamientos cuando el valor resultante de una operación supera la capacidad del tipo. Por ejemplo, sumar 1 al valor máximo en un entero de 8 bits (`01111111` = 127) produce un desbordamiento que genera `10000000`, interpretado como –128 en complemento a dos [53]. Comprender estos comportamientos es esencial para escribir programas seguros y confiables [12].

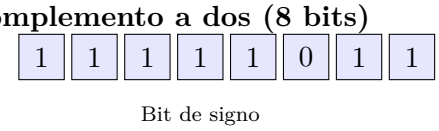

Figura 1.5: Representación de enteros con signo mediante complemento a dos [53], [54].

Reales y precisión

Los números reales se representan habitualmente mediante el estándar IEEE 754, que utiliza una estructura de punto flotante compuesta por tres campos: signo, exponente y mantisa [53]. En precisión simple (32 bits), estos campos ocupan 1 bit, 8 bits y 23 bits respectivamente, lo que permite representar un amplio rango de valores con distintos niveles de precisión [54]. La normalización de la mantisa y el sesgo del exponente son aspectos clave en esta representación [3].

La precisión limitada provoca que algunos números no puedan representarse exactamente. Por ejemplo, valores como 0.1 o 0.2 no tienen una representación binaria finita, lo que genera errores acumulativos cuando se realizan sumas sucesivas [12]. Esto explica por qué en algunos lenguajes una expresión como `0.1 + 0.2` produce resultados como `0.30000000000000004` en lugar de 0.3 [32]. La falta de precisión debe considerarse al trabajar con cálculos sensibles o financieros [3].

Los números IEEE 754 también incluyen valores especiales como `NaN` (Not a Number), ∞, $-\infty$ y cero positivo o negativo, utilizados para representar desbordamientos, divisiones por cero o resultados indefinidos [53]. Estos valores tienen patrones binarios específicos para distinguirlos de números normales y subnormales [54]. Comprender estas particularidades es fundamental para evitar errores de cálculo y comportamientos inesperados [12].

IEEE 754 (32 bits)

Figura 1.6: Distribución de campos en el formato IEEE 754 de precisión simple [53], [54].

Caracteres (ASCII y Unicode)

Los caracteres se representan mediante códigos numéricos que asocian cada símbolo con un valor entero [53]. ASCII es uno de los estándares más antiguos, utilizando 7 bits para representar 128 caracteres, incluyendo letras mayúsculas y minúsculas, dígitos y signos de puntuación [54]. Por ejemplo, la letra ''A'' se codifica como 65 (`01000001`) y ''a'' como 97 (`01100001`) [32]. Aunque limitado, ASCII sigue siendo ampliamente utilizado por su simplicidad.

Unicode surgió para superar las restricciones de ASCII y permitir la representación de caracteres de todos los idiomas, símbolos técnicos, emojis y más [3]. El estándar Unicode

asigna un punto de código único a cada carácter, que puede codificarse mediante formatos como UTF-8, UTF-16 o UTF-32 [53]. Por ejemplo, el carácter ''ñ'' tiene punto de código U+00F1, que en UTF-8 se representa como `C3 B1` [54].

La coexistencia de ASCII y Unicode implica que algunos caracteres tienen equivalencias directas en ambas codificaciones, mientras que otros existen únicamente en Unicode [12]. Por ejemplo, el carácter ''A'' mantiene el mismo valor en ASCII y UTF-8, mientras que ''ñ'' y miles de otros caracteres no pueden representarse en ASCII [3]. Esto hace que la elección del formato de codificación sea crucial en aplicaciones multilingües.

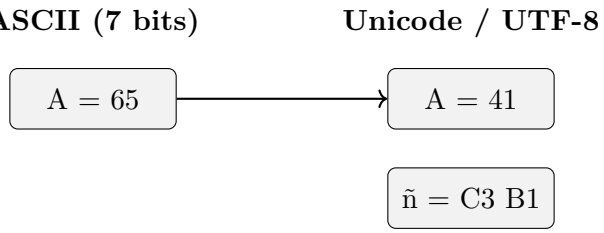

Figura 1.7: Equivalencias entre ASCII y Unicode para caracteres comunes y extendidos [53], [54].

Valores lógicos (booleanos)

Los valores lógicos representan estados booleanos verdaderos o falsos, fundamentales para el control del flujo de programas mediante estructuras como condicionales y bucles [32]. Internamente, suelen codificarse mediante un bit, utilizando 0 para falso y 1 para verdadero, aunque algunos lenguajes emplean representaciones más amplias por razones de alineación en memoria [53]. Esta simplicidad permite implementar operadores lógicos de forma eficiente mediante compuertas digitales [54].

Los operadores lógicos como AND, OR y NOT se interpretan mediante reglas precisas definidas en la lógica booleana [53]. Por ejemplo, la expresión `(x > 10) && (y < 5)` sólo es verdadera cuando ambas condiciones se cumplen [32]. Los valores booleanos también se utilizan en máscaras de bits, activación de señales y condiciones de parada [3].

Los booleanos pueden provocar errores cuando se confunden con enteros o cuando se aplican operaciones inadecuadas, como comparar directamente valores lógicos con números fuera del rango permitido [12]. Por ejemplo, en algunos lenguajes, cualquier valor distinto de cero se interpreta como verdadero, lo que puede generar confusión si no se utiliza una comparación explícita [3]. Por ello, se recomienda utilizar variables booleanas específicas en lugar de valores numéricos en expresiones lógicas [53].

1.5.4. Implicaciones en programación

La forma en que se representan internamente los tipos básicos tiene implicaciones directas en el diseño, la corrección y el rendimiento de los programas [32]. El programador debe conocer los rangos de los tipos enteros, las limitaciones de precisión de los números en punto flotante, las codificaciones de caracteres y la semántica de los valores booleanos para evitar errores sutiles que no siempre son detectados por el compilador [54]. Una comprensión superficial de estos aspectos puede conducir a resultados incorrectos, desbordamientos silenciosos y comparaciones engañosas [53].

Estas implicaciones se manifiestan especialmente en operaciones críticas como cálculos financieros, simulaciones científicas, algoritmos de control y evaluación de condiciones complejas [3]. Por ejemplo, el uso inadecuado de un tipo de dato con rango insuficiente puede provocar fallos intermitentes que aparecen solo con determinados valores de entrada [12]. De manera análoga, la comparación directa de valores en punto flotante o la conversión implícita entre tipos puede introducir errores difíciles de depurar [53].

Desbordamientos

El desbordamiento ocurre cuando el resultado de una operación aritmética excede el rango que puede representar el tipo de dato utilizado [54]. En un entero con signo de 8 bits, el rango va de –128 a 127; si se intenta sumar 1 a 127, el resultado se desborda y se interpreta como –128 en complemento a dos [53]. Este comportamiento es definido por la arquitectura y no suele generar errores de tiempo de ejecución, lo que complica su detección [32].

En muchos lenguajes de bajo nivel, como C o C++, el desbordamiento de enteros se considera comportamiento no definido o bien se produce sin advertencia, lo que puede dar lugar a vulnerabilidades de seguridad y resultados incorrectos [12]. Por ejemplo, contadores, índices de arreglos o acumuladores pueden sobrepasar su límite y producir accesos fuera de rango o datos incoherentes [3]. En lenguajes de más alto nivel, algunas implementaciones generan excepciones o amplían dinámicamente la capacidad de representación, pero esto depende de la plataforma [53].

Para mitigar el desbordamiento se recomienda validar los rangos antes de realizar operaciones críticas, utilizar tipos de mayor capacidad cuando se anticipan valores grandes y, en contextos de alto riesgo, recurrir a bibliotecas de aritmética de precisión arbitraria [12]. Asimismo, herramientas de análisis estático pueden detectar patrones de posible desbordamiento durante la fase de desarrollo [3]. Estas estrategias contribuyen a la robustez

y confiabilidad de los sistemas [53].

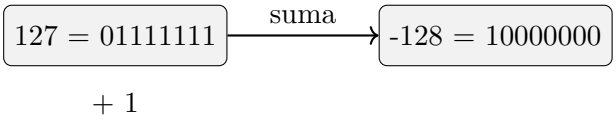

Figura 1.8: Desbordamiento al sumar 1 al valor máximo en un entero con signo de 8 bits [53], [54].

Errores de comparación

Los errores de comparación surgen cuando dos valores que deberían considerarse equivalentes no lo son desde el punto de vista de su representación interna, o cuando se emplean operadores relacionales de forma incorrecta [32]. Este fenómeno es especialmente frecuente con números en punto flotante, debido a los errores de redondeo introducidos por el estándar IEEE 754 [53]. Por ejemplo, la expresión `0.1 + 0.2 == 0.3` puede evaluar a falso en muchos lenguajes, pese a la expectativa intuitiva de igualdad [54].

La literatura recomienda evitar las comparaciones directas de igualdad entre valores flotantes y, en su lugar, utilizar comparaciones con tolerancia, verificando si la diferencia absoluta entre dos valores es menor que un umbral pequeño (ϵ) [3]. Por ejemplo, resulta más seguro comprobar `|x - 0.3| < 1e-6` que utilizar `x == 0.3` [12]. Esta técnica reduce el impacto de los errores de representación y mejora la robustez de los programas numéricos [53].

Los errores de comparación también pueden presentarse con cadenas de caracteres cuando existen diferencias invisibles, como espacios en blanco adicionales, mayúsculas y minúsculas o combinaciones Unicode equivalentes visualmente pero distintas internamente [53]. Por ejemplo, la cadena ‘”café”‘ puede representarse de forma precompuesta o descompuesta en Unicode, lo que afecta el resultado de las comparaciones byte a byte [54]. Para mitigar estos problemas se recomienda normalizar cadenas y aplicar comparaciones que consideren reglas locales de mayúsculas y acentos [3].

En lenguajes con punteros o referencias, otro error común es comparar direcciones de memoria en lugar del contenido al que apuntan [12]. Dos punteros pueden referirse a distintas ubicaciones que contengan valores iguales, pero la comparación de las direcciones devolverá falso [53]. Por ello, se debe distinguir entre igualdad de referencias e igualdad de valores, utilizando las operaciones adecuadas según el lenguaje y el contexto [3].

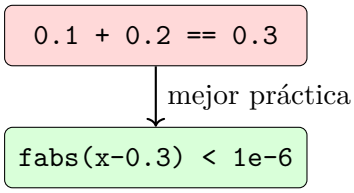

Figura 1.9: Comparación directa vs. comparación con tolerancia en números en punto flotante [12], [53].

Pérdida de precisión

La pérdida de precisión ocurre cuando un valor no puede representarse exactamente con el número de bits disponible o cuando se convierte desde un tipo más preciso a otro menos preciso [53]. Esto sucede con frecuencia al almacenar un número real con muchos decimales en un formato de precisión simple o al convertir un número de punto flotante a entero mediante truncamiento [54]. Por ejemplo, convertir 3.999 a entero puede producir 3 si se aplica truncamiento, aun cuando el valor esté muy próximo a 4 [32].

En cálculos acumulativos, pequeñas pérdidas de precisión pueden propagarse y amplificarse, afectando significativamente los resultados finales [3]. Por este motivo, en aplicaciones científicas o financieras se recomienda utilizar tipos de mayor precisión, algoritmos numéricamente estables y, cuando sea necesario, bibliotecas de precisión arbitraria [12]. Además, debe prestarse especial atención al orden de las operaciones, ya que sumas de números muy grandes con números muy pequeños pueden provocar que estos últimos se pierdan en el redondeo [53].

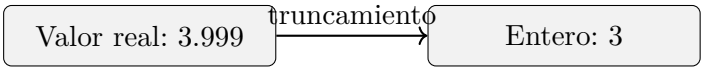

Figura 1.10: Pérdida de precisión al convertir de real a entero por truncamiento [32], [53].

1.6. El problema computacional

El problema computacional se define como la formulación precisa, en términos de entradas, salidas y restricciones, de una situación del mundo real que se desea resolver mediante un procedimiento ejecutable por una computadora [12], [32]. Esta formulación implica abstraer los detalles irrelevantes, identificar los datos disponibles y determinar el resultado esperado siguiendo criterios de corrección y eficiencia, tal como enfatizan los textos de ingeniería de software y algoritmia [3], [54]. La calidad de esta definición

condiciona directamente la posibilidad de diseñar algoritmos correctos y de implementar programas que respondan adecuadamente a las necesidades de usuarios y organizaciones [53].

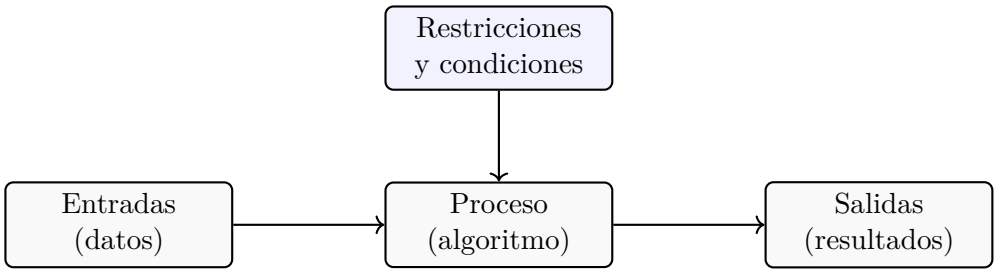

Figura 1.11: Esquema general de un problema computacional: entradas, proceso, salidas y restricciones [12], [32].

1.6.1. Análisis del problema

El análisis del problema consiste en estudiar en detalle la situación que se desea resolver, identificando actores, objetivos, datos relevantes, reglas de negocio y restricciones operativas [3], [12]. Esta etapa antecede al diseño algorítmico y al desarrollo del programa, y tiene como propósito comprender el dominio del problema para evitar interpretaciones ambiguas o incompletas [32]. La literatura de ingeniería de requisitos enfatiza que un análisis deficiente conduce a soluciones técnicamente correctas pero funcionalmente inadecuadas [12].

Durante el análisis se describen explícitamente las entradas, salidas y transformaciones esperadas, así como los escenarios típicos y excepcionales que el sistema deberá manejar [3]. Se utilizan modelos informales (descripciones en lenguaje natural) y modelos formales (diagramas de flujo de datos, casos de uso, modelos de procesos) para estructurar la información y validar el entendimiento con los interesados [53]. Este enfoque reduce la brecha de comunicación entre usuarios y desarrolladores, minimizando el riesgo de malentendidos [12].

Además, el análisis del problema permite evaluar la viabilidad técnica, económica y organizacional de la solución propuesta [3]. En esta etapa se identifican dependencias con otros sistemas, requisitos de rendimiento, restricciones legales o normativas, y criterios de éxito medibles [54]. Esta información sirve de base para el diseño de algoritmos y para la planificación del desarrollo del software [12].

Entradas

Las entradas de un problema computacional corresponden a los datos que el sistema requiere para ejecutar el procedimiento de solución [32]. Pueden provenir de usuarios (mediante formularios), de otros sistemas (mediante interfaces de software) o de dispositivos físicos (sensores, lectores de códigos de barras, etc.) [55]. La identificación precisa de estas entradas es fundamental para garantizar que el algoritmo disponga de la información necesaria para producir resultados correctos [3].

En el contexto de sistemas de información, las entradas suelen asociarse con entidades del negocio como clientes, productos, estudiantes, transacciones o documentos, que se describen mediante atributos específicos (nombre, fecha, monto, identificador, etc.) [12]. Por ejemplo, en un sistema de matrícula académica, las entradas típicas incluyen datos del estudiante, asignaturas seleccionadas, periodo académico y formas de pago [3]. Cada uno de estos datos debe definirse con tipo, formato y dominio válidos [53].

Asimismo, es necesario considerar la calidad de las entradas en términos de precisión, completitud y consistencia [32]. La literatura de sistemas de información señala que entradas incorrectas o incompletas pueden propagarse a lo largo del procesamiento, generando información final errónea y decisiones equivocadas [12]. Por ello, los algoritmos deben incorporar validaciones que verifiquen rangos, formatos y restricciones antes de procesar los datos [3].

Salidas

Las salidas representan los resultados que el sistema entrega tras procesar las entradas según el algoritmo definido [32]. Pueden adoptar la forma de informes, mensajes, gráficos, archivos generados o actualizaciones en bases de datos, dependiendo de los objetivos del problema computacional [12]. La definición clara de estas salidas permite evaluar si la solución propuesta satisface las necesidades de los usuarios y de la organización [3].

En sistemas de información, las salidas deben diseñarse de forma que sean comprensibles, completas y oportunas para apoyar la toma de decisiones [53]. Por ejemplo, en un sistema contable, las salidas incluyen balances, estados de resultados y reportes de flujo de caja; en un sistema académico, las salidas pueden ser actas de calificaciones, certificados o reportes de rendimiento [3]. Cada salida se deriva de transformaciones lógicas y aritméticas aplicadas a las entradas y a los datos almacenados [12].

Además, la representación de las salidas debe considerar aspectos de formato, presentación y seguridad [55]. La literatura de ingeniería de software recomienda diseñar las

salidas de manera que resuman información relevante sin saturar al usuario, incorporando indicadores, alertas o visualizaciones cuando sea pertinente [12]. La forma en que se diseñan estas salidas influye directamente en la utilidad percibida del sistema [3].

Restricciones y condiciones

Las restricciones y condiciones definen los límites dentro de los cuales la solución computacional debe operar, incluyendo requisitos de tiempo, recursos, normativas legales, políticas internas y condiciones de entorno [12]. Estas restricciones pueden ser explícitas (por ejemplo, "el reporte debe generarse en menos de cinco segundos") o implícitas (capacidad máxima de almacenamiento, número de usuarios concurrentes, etc.) [3]. Ignorar estas condiciones puede dar lugar a soluciones técnicamente correctas pero inviables en la práctica [53].

En sistemas de información, las restricciones suelen abarcar reglas de negocio (por ejemplo, "no se puede matricular una asignatura si el prerrequisito está reprobado"), políticas de seguridad (roles de acceso, confidencialidad de datos) y requisitos de integridad (consistencia entre módulos, transacciones atómicas) [55]. Estas reglas condicionan tanto el diseño de algoritmos como la estructura de los datos y las interfaces de usuario [12].

Asimismo, la literatura enfatiza la importancia de documentar y validar las restricciones con los interesados antes de diseñar la solución [3]. Esta validación asegura que las condiciones reflejen fielmente el contexto operativo y que puedan verificarse mediante pruebas y mecanismos de monitoreo [12]. De este modo, las restricciones se convierten en criterios objetivos para evaluar el correcto funcionamiento del sistema [53].

1.6.2. Ejemplos de problemas en sistemas de información

Los sistemas de información empresariales ofrecen numerosos ejemplos de problemas computacionales que pueden modelarse en términos de entradas, salidas, procesos y restricciones, tales como control de inventario, registro de estudiantes y procesamiento de ventas [3], [12]. Cada uno de estos casos ilustra cómo los conceptos de datos, información y conocimiento se integran para apoyar la toma de decisiones y automatizar operaciones clave en organizaciones modernas [32].

Control de inventario

En un problema de control de inventario, las entradas incluyen datos de productos, niveles de stock, movimientos de entrada y salida, pedidos de compra y ventas realizadas [12]. El sistema debe procesar esta información para calcular existencias actuales, identificar productos por debajo del nivel mínimo y generar alertas o pedidos automáticos a proveedores [3]. Las salidas típicas son reportes de inventario, listados de productos críticos y proyecciones de demanda con base en históricos [53].

Las restricciones pueden incluir capacidad máxima de almacenamiento, fechas de caducidad, políticas de reordenamiento y prioridades de productos [55]. Un algoritmo bien diseñado debe considerar estos factores para evitar desabastecimientos o sobreinventario, fenómenos que tienen impacto directo en costos y niveles de servicio [12]. Este tipo de problema se modela frecuentemente mediante diagramas de flujo de datos y modelos de procesos de negocio en la literatura de sistemas de información [3].

Registro de estudiantes

En un sistema de registro de estudiantes, las entradas comprenden datos personales, información académica previa, asignaturas seleccionadas, horarios y opciones de pago [32]. El sistema debe validar la información de acuerdo con reglas de integridad (por ejemplo, formato de identificación, requisitos de edad) y reglas académicas (cumplimiento de prerrequisitos, límites de créditos, compatibilidad de horarios) [3]. Las salidas incluyen comprobantes de matrícula, horarios personalizados y actualizaciones en el historial académico [12].

Las restricciones abarcan cupos por asignatura, calendarios académicos, límites de créditos por periodo y políticas internas de la institución [55]. Un algoritmo de registro debe manejar condiciones especiales, como reservas de cupo, listas de espera o prioridades según tipo de estudiante, aspectos tratados en la literatura de sistemas de gestión académica [12]. Estas condiciones añaden complejidad al problema computacional y requieren un diseño cuidadoso de estructuras de datos y procesos [3].

Además, el sistema debe garantizar confidencialidad y seguridad de los datos de los estudiantes, cumpliendo normativas legales y políticas institucionales [55]. Esto implica implementar controles de acceso, cifrado y auditoría, elementos que conectan el problema computacional con la gestión de riesgos y la seguridad de la información [53].

Procesamiento de ventas

En el procesamiento de ventas, las entradas incluyen datos de clientes, productos, cantidades, precios, descuentos y métodos de pago [32]. El sistema debe calcular subtotales, impuestos y totales a pagar, actualizando simultáneamente el inventario y los registros contables [12]. Las salidas típicas son facturas, recibos electrónicos, reportes de ventas diarias y registros de transacciones para análisis posterior [3].

Las restricciones incorporan políticas comerciales (descuentos máximos, promociones vigentes), límites de crédito, validación de medios de pago y cumplimiento de normativas fiscales [55]. El algoritmo debe garantizar que cada transacción sea atómica, consistente, aislada y duradera (propiedades ACID), especialmente cuando se utilizan bases de datos transaccionales [53]. Estos requisitos son ampliamente documentados en textos de sistemas de bases de datos y sistemas de información [12].

Finalmente, los datos de ventas se utilizan para generar conocimiento sobre patrones de consumo, productos más vendidos y periodos de mayor demanda [32]. Este conocimiento alimenta procesos de toma de decisiones estratégicas y de planificación, relacionando el problema computacional con la analítica de datos y la inteligencia de negocio [3].

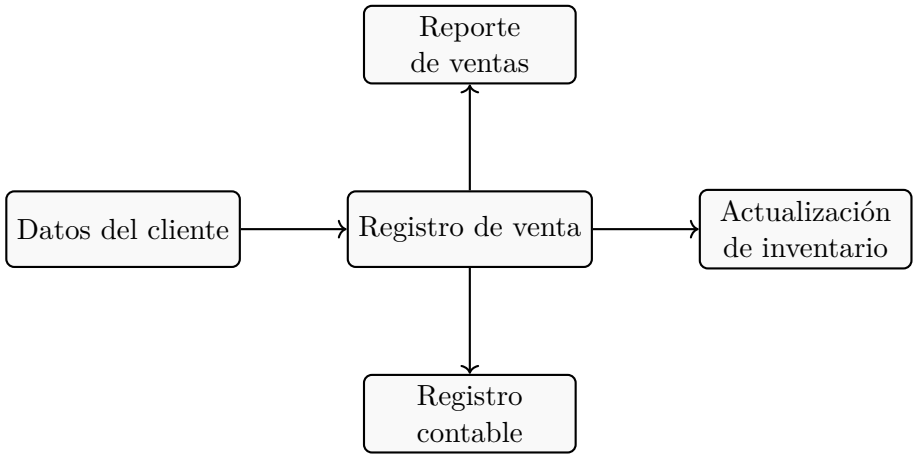

Figura 1.12: Flujo simplificado de información en el procesamiento de ventas [3], [12].

1.7. Proceso de resolución de problemas en programación

El proceso de resolución de problemas en programación se estructura en una secuencia de fases que abarcan desde el análisis del problema hasta el mantenimiento y la mejora continua

del software, integrando actividades de modelado, diseño algorítmico, codificación, pruebas y documentación [3], [7], [12]. Esta organización en etapas permite reducir la complejidad, mejorar la calidad de las soluciones y favorecer la comunicación entre los miembros del equipo de desarrollo y los usuarios finales [32]. La literatura de ingeniería de software y algoritmia insiste en que una solución computacional robusta no surge únicamente de escribir código, sino de seguir un proceso disciplinado que garantice corrección, eficiencia y mantenibilidad [7], [53], [54].

Figura 1.13: Fases generales del proceso de resolución de problemas en programación [3], [7], [12].

1.7.1. Etapas formales

Las etapas formales en la resolución de problemas incluyen análisis, modelado, diseño de algoritmos, codificación, pruebas y documentación, cada una con objetivos y productos claramente definidos [7], [12]. Esta estructura se alinea con los ciclos de desarrollo de software descritos en modelos de proceso clásicos, donde se enfatiza que el avance entre etapas debe basarse en resultados verificables y no solo en la intuición del desarrollador [3]. Al formalizar estas fases, se facilita la planificación, la asignación de recursos y la evaluación de la calidad en cada punto del proceso [7], [53].

Además, las etapas formales permiten establecer mecanismos de retroalimentación, de modo que los errores detectados en fases posteriores puedan corregirse en fases anteriores del proceso [7], [12]. Por ejemplo, defectos encontrados en pruebas pueden revelar problemas de diseño o de análisis de requisitos, lo que impulsa ajustes en los modelos conceptuales o en los algoritmos [3], [7]. Este enfoque iterativo es fundamental para construir soluciones robustas en entornos reales, donde los requisitos pueden evolucionar con el tiempo [32].

Análisis

La fase de análisis se centra en comprender el problema desde la perspectiva del dominio, identificando qué se quiere lograr, quiénes son los usuarios involucrados y qué restricciones condicionan la solución [12]. En esta etapa se recogen requisitos funcionales (servicios que debe prestar el sistema) y no funcionales (rendimiento, seguridad, usabilidad), utilizando

técnicas de elicitación como entrevistas, cuestionarios y análisis de documentos [3]. Una comprensión incompleta del problema en esta fase suele conducir a soluciones inadecuadas o a costosas correcciones posteriores [7], [53].

Durante el análisis se formulan descripciones en lenguaje natural, tablas de entradas y salidas, y escenarios de uso que ilustran cómo interactuarán los usuarios con el futuro sistema [32]. Por ejemplo, en un sistema de matrícula, se describen casos como "registrar estudiante", "agregar asignatura" o "generar comprobante", detallando datos requeridos y resultados esperados [7], [12]. Estos escenarios ayudan a validar el entendimiento del problema con los interesados y a detectar omisiones o ambigüedades [3].

Asimismo, el análisis incluye la identificación de riesgos, dependencias y supuestos que podrían afectar el desarrollo o la operación del sistema [12]. Documentar estos elementos permite priorizar requisitos, negociar alcances y planificar estrategias de mitigación [3]. La fase de análisis, por tanto, constituye la base conceptual sobre la cual se edificarán las etapas posteriores del proceso de programación [7], [53].

Modelado

El modelado traduce los resultados del análisis en representaciones más estructuradas, utilizando diagramas, notaciones formales y herramientas gráficas que simplifican la comprensión del sistema [12]. Entre estas representaciones se incluyen diagramas de flujo de datos, diagramas de casos de uso, diagramas de clases o modelos de procesos, según las metodologías sugeridas en la literatura de ingeniería de software [3]. El objetivo es reducir la complejidad del problema mediante abstracciones que permitan ver el sistema como un conjunto de componentes y relaciones [7], [53].

Por ejemplo, un diagrama de flujo de datos puede mostrar cómo la información se desplaza entre procesos de negocio, bases de datos y entidades externas, clarificando entradas, salidas y transformaciones intermedias [12]. Esta visualización facilita la detección de redundancias, inconsistencias o pasos innecesarios en los procesos [3]. Asimismo, los modelos se convierten en referencia para el diseño algorítmico y la posterior implementación [32].

Los modelos también sirven como documentos de comunicación entre analistas, desarrolladores y usuarios, ya que ofrecen una representación intermedia entre el lenguaje natural y el código fuente [12]. Esta función comunicativa es esencial para alinear expectativas y para validar que el sistema refleje fielmente el dominio del problema [3]. En consecuencia, el modelado contribuye directamente a la calidad global del software [7], [53].

Diseño algorítmico

El diseño algorítmico consiste en definir, a partir de los modelos, una secuencia precisa de pasos que permita transformar las entradas en salidas de acuerdo con las restricciones y objetivos del problema [32]. Esta etapa utiliza pseudocódigo, diagramas de flujo, diagramas de N-S u otras notaciones que describen la lógica de la solución sin depender aún de un lenguaje de programación específico [7], [12]. El propósito es centrarse en la corrección y claridad de la estrategia antes de abordar detalles sintácticos [3].

La literatura de algoritmia y estructuras de datos enfatiza que un buen diseño algorítmico debe considerar eficiencia temporal y espacial, simplicidad y modularidad [53]. Por ejemplo, se evalúa si un problema de búsqueda requiere una solución lineal o si puede beneficiarse de algoritmos más eficientes como búsqueda binaria, dependiendo de la estructura de los datos [54]. Estas decisiones tienen impacto directo en el rendimiento del sistema [12].

Una vez definido el algoritmo, se realizan verificaciones informales o formales para comprobar su corrección, revisando casos típicos, casos límite y situaciones excepcionales [3], [7]. Esta revisión temprana permite detectar errores lógicos y mejorar la estructura antes de la codificación [32]. De este modo, el diseño algorítmico actúa como puente entre los modelos conceptuales y el código ejecutable [12].

Codificación

La codificación consiste en traducir el algoritmo a un lenguaje de programación concreto, respetando su sintaxis, semántica y convenciones de estilo [32]. En esta fase se seleccionan estructuras de control (secuencias, decisiones, bucles), tipos de datos y bibliotecas estándar o especializadas que faciliten la implementación de la lógica diseñada [3]. La calidad del código se mide en términos de legibilidad, modularidad, reusabilidad y ausencia de errores [7], [12].

Los manuales de programación recomiendan aplicar buenas prácticas como nombrar adecuadamente variables y funciones, utilizar comentarios claros, evitar duplicación innecesaria de código y seguir guías de estilo específicas del lenguaje [32]. Estas prácticas simplifican el mantenimiento, la revisión por pares y la evolución del sistema [7], [12]. Asimismo, la codificación se apoya en herramientas como editores, entornos de desarrollo integrado (IDE) y sistemas de control de versiones [3].

Durante la codificación también se introducen pruebas unitarias básicas que verifican el comportamiento de funciones o módulos individuales [12]. Este enfoque, asociado con metodologías como desarrollo guiado por pruebas (TDD), ayuda a detectar errores tempra-

namente y garantiza que el código cumpla con las especificaciones algorítmicas [3], [7]. La estrecha relación entre codificación y pruebas contribuye a la construcción incremental y confiable de sistemas complejos [53].

Pruebas

La fase de pruebas tiene como objetivo detectar defectos y verificar que el software cumple con los requisitos funcionales y no funcionales establecidos durante el análisis [12]. Se distinguen diferentes niveles de prueba: unitarias, de integración, de sistema y de aceptación, cada una enfocada en un nivel de agregación distinto [3]. Las pruebas se diseñan a partir de casos específicos que cubren condiciones normales, límites y situaciones de error [7], [53].

En el contexto de programación básica, las pruebas unitarias y de integración resultan especialmente relevantes, pues permiten comprobar que funciones, procedimientos y módulos colaboran de manera correcta [32]. Los resultados de las pruebas se documentan y analizan para determinar la causa raíz de los errores, lo cual conduce a correcciones en el código o incluso a revisiones en el diseño algorítmico [12]. Este ciclo de prueba–corrección es esencial para alcanzar una calidad aceptable antes de desplegar el software [3].

Además, las pruebas pueden automatizarse mediante marcos especializados que ejecutan casos de prueba de forma repetible y registran resultados de manera sistemática [12]. Esta automatización es fundamental para proyectos que evolucionan continuamente, donde pequeñas modificaciones al código pueden reintroducir errores previos [3]. La literatura destaca la importancia de considerar las pruebas como parte integral del proceso de desarrollo y no como una fase aislada al final [53].

Documentación

La documentación reúne los artefactos que describen el sistema desde distintas perspectivas: requisitos, diseño, código, manuales de usuario y guías de operación [12]. Su propósito es facilitar la comprensión, el mantenimiento y la evolución del software a lo largo del tiempo, permitiendo que otros desarrolladores y usuarios puedan interpretarlo y utilizarlo adecuadamente [3]. En proyectos académicos y profesionales, la documentación bien estructurada es un indicador clave de calidad [7], [53].

Entre los documentos más relevantes se encuentran las especificaciones de requisitos, los diagramas de diseño, los comentarios en el código, los manuales de instalación y los manuales de usuario [12]. Cada uno cumple una función específica: los requisitos describen

qué debe hacer el sistema; los diagramas muestran cómo está organizado; los comentarios aclaran decisiones técnicas; y los manuales explican cómo utilizarlo y administrarlo [3]. Esta diversidad de documentos contribuye a que el sistema sea comprensible en diferentes niveles de detalle [32].

La literatura enfatiza que la documentación debe mantenerse actualizada y coherente con el código fuente [12]. Documentos obsoletos generan confusión y pueden inducir a errores durante el mantenimiento o la ampliación del sistema [3]. Por ello, se recomienda incluir la actualización de la documentación como parte explícita del proceso de desarrollo y de las políticas de control de versiones [53].

1.7.2. Importancia del pensamiento algorítmico

El pensamiento algorítmico se define como la capacidad de descomponer un problema en pasos ordenados, finitos y efectivos que conduzcan a una solución [32]. Esta forma de razonamiento permite abstraer detalles irrelevantes, identificar patrones de solución y diseñar procedimientos generales aplicables a múltiples casos [12]. En el contexto de la programación, el pensamiento algorítmico constituye una competencia fundamental que diferencia el simple uso de herramientas del verdadero diseño de soluciones computacionales [3].

Asimismo, el pensamiento algorítmico promueve la claridad y precisión en la formulación de instrucciones, lo que reduce la ambigüedad y facilita la verificación de la corrección de los programas [53]. La literatura sobre enseñanza de la programación resalta que desarrollar esta competencia es un objetivo central en los cursos introductorios, pues prepara a los estudiantes para abordar problemas complejos en etapas posteriores de su formación [32]. Este enfoque contribuye a una comprensión más profunda de la relación entre problema, algoritmo y programa [12].

Abstracción

La abstracción es la habilidad para identificar los aspectos esenciales de un problema y omitir detalles irrelevantes, construyendo modelos simplificados que facilitan su análisis y solución [3]. En programación, abstraer implica concentrarse en qué debe hacer un módulo o función sin preocuparse inicialmente por cómo se implementará internamente [12]. Esta separación de responsabilidades mejora la claridad y permite dividir el trabajo entre distintos miembros del equipo [53].

Por ejemplo, al diseñar un módulo de *gestión de estudiantes*, se abstrae el concepto de estudiante como una entidad con atributos (nombre, identificación, asignaturas) y operaciones (matricular, dar de baja, generar reporte), sin especificar todavía estructuras de datos o detalles de almacenamiento [32]. Esta abstracción posibilita que diferentes implementaciones cumplan la misma interfaz, lo que facilita la reutilización y el mantenimiento [12].

La abstracción se aplica también a nivel de datos y procesos, permitiendo definir tipos de datos abstractos, clases, interfaces y servicios que encapsulan detalles internos [3]. Esta técnica, ampliamente documentada en la literatura de diseño de software, favorece la construcción de sistemas modulares y extensibles [53].

Decomposición

La decomposición consiste en dividir un problema complejo en subproblemas más pequeños y manejables, permitiendo abordar cada parte de forma independiente [3]. Esta técnica se fundamenta en el principio de "divide y vencerás", ampliamente utilizado en ingeniería de software y en el diseño de algoritmos [12]. La decomposición reduce la complejidad cognitiva y facilita la verificación de cada componente, contribuyendo a la calidad global del sistema [53].

Por ejemplo, un sistema de gestión académica puede dividirse en módulos como gestión de estudiantes, manejo de asignaturas, registro de matrículas, generación de reportes y administración de usuarios [32]. Cada módulo tiene responsabilidades claras y puede implementarse de manera independiente, siempre que respete las interfaces definidas durante el diseño [3]. Esto permite desarrollar, probar y mantener cada parte sin afectar el funcionamiento general del sistema [12].

La decomposición también favorece la reutilización de componentes, ya que subproblemas comunes pueden resolverse mediante funciones o módulos reaprovechables en distintos contextos [53]. Además, facilita la asignación de tareas en equipos de desarrollo, mejorando la organización del proyecto y la eficiencia del trabajo colaborativo [3]. Por estas razones, la decomposición es considerada una habilidad esencial en el pensamiento algorítmico y en la formación de programadores [32].

Patrones de solución

Los patrones de solución representan estrategias recurrentes para resolver problemas que comparten características estructurales, permitiendo reutilizar conocimiento y enfoques

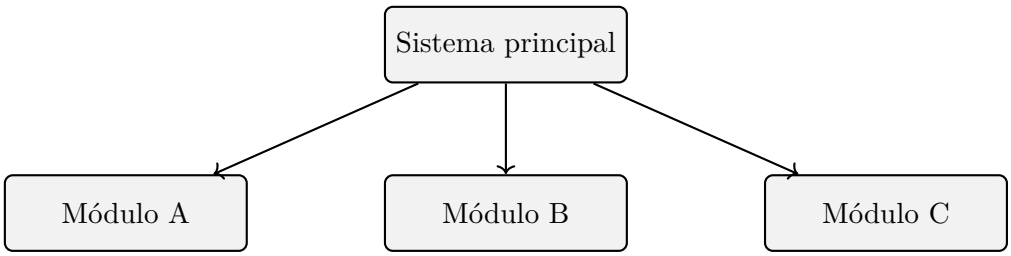

Figura 1.14: Ejemplo de decomposición modular [3], [12].

probados [12]. En el ámbito del diseño de algoritmos, estos patrones incluyen técnicas como búsqueda, ordenamiento, recursión, divide y vencerás, programación dinámica y algoritmos codiciosos [53]. Su estudio proporciona un marco conceptual que ayuda a seleccionar el enfoque más adecuado para cada tipo de problema [3].

Por ejemplo, cuando el problema implica encontrar un elemento en una lista ordenada, la búsqueda binaria constituye un patrón eficiente que reduce la complejidad temporal de $O(n)$ a $O(\log n)$ [54]. Asimismo, problemas como generación de reportes, análisis de datos o manejo de estructuras compuestas se benefician de patrones como iteración controlada, filtrado, transformación y reducción [32]. Estos patrones permiten construir soluciones más claras y predecibles, facilitando el aprendizaje y la implementación [12].

Los patrones de solución no solo mejoran la eficiencia y claridad del código, sino que también contribuyen a la mantenibilidad y escalabilidad del software [3]. La literatura enfatiza que los desarrolladores deben reconocer cuándo un patrón aplica y cuándo no, evitando el uso inapropiado de técnicas que podrían generar complejidad innecesaria [53]. De esta manera, los patrones constituyen un puente entre el pensamiento algorítmico y el diseño de software robusto [12].

1.8. Algoritmos: concepto y propiedades esenciales

Un algoritmo es un conjunto finito y ordenado de pasos que describen cómo transformar entradas en salidas, cumpliendo condiciones de precisión, efectividad, finitud y determinismo [12], [32]. La definición formal de algoritmo, estudiada ampliamente en teoría computacional y programación, establece que cada instrucción debe ser clara e inequívoca, y que el procedimiento debe garantizar resultados correctos en un número finito de pasos [3], [54]. Los algoritmos constituyen la base de toda solución computacional, ya que definen la lógica que ejecutarán los programas [53].

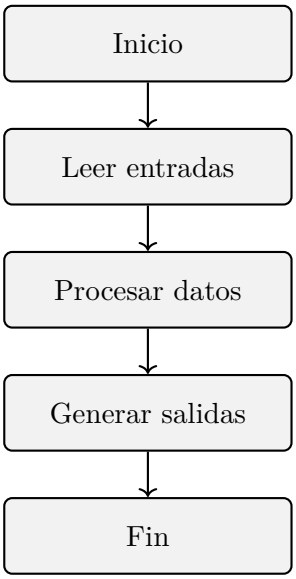

Figura 1.15: Estructura básica de un algoritmo [32], [53].

1.8.1. Definición formal

En su definición formal, un algoritmo debe especificar pasos precisos, ejecutables y no ambiguos, siguiendo la teoría desarrollada en la literatura clásica de programación y sistemas [54]. Cada paso debe representar una instrucción que pueda ser interpretada por un agente computacional, ya sea humano o máquina, garantizando claridad en su ejecución [12]. Este principio de precisión evita interpretaciones múltiples y asegura consistencia en resultados [3].

Además, un algoritmo debe ser finito, es decir, debe concluir su ejecución en un número limitado de pasos [32]. La finitud distingue los algoritmos de procesos indefinidos o recursiones infinitas y permite que las soluciones computacionales sean verificables y medibles [53]. La literatura técnica enfatiza que la ausencia de finitud puede provocar bloqueos, ciclos infinitos o fallas en sistemas críticos [12].

Finalmente, la efectividad del algoritmo implica que cada operación especificada debe ser realizable mediante recursos computacionales disponibles y debe producir el efecto esperado sobre los datos [54]. Esta propiedad garantiza que los algoritmos puedan implementarse en hardware real y se ejecuten dentro de tiempos razonables, conforme a los parámetros de eficiencia establecidos en ingeniería de software [3].

1.8.2. Propiedades esenciales

Los algoritmos poseen propiedades esenciales que permiten evaluarlos: finitud, determinismo, efectividad y correctitud [12]. La finitud garantiza que el procedimiento terminará; el determinismo asegura que cada paso es claro y no admite ambigüedades; y la efectividad indica que cada operación es realizable mediante computación real [3]. Estas propiedades definen la naturaleza formal de los algoritmos y distinguen procedimientos correctos de descripciones vagas o indefinidas [53].

El determinismo implica que un algoritmo, dado un conjunto de entradas, debe producir siempre las mismas salidas siguiendo los mismos pasos, sin depender de factores externos o aleatoriedad [54]. Esta propiedad es fundamental en sistemas críticos, donde decisiones automáticas deben ser reproducibles y auditables [12]. Los modelos de evaluación de algoritmos analizan esta característica mediante pruebas de caja negra y caja blanca [3].

La efectividad está asociada a que cada instrucción del algoritmo corresponda a operaciones simples, realizables y comprensibles, evitando acciones abstractas imposibles de implementar en hardware o software [32]. Esta propiedad asegura la aplicabilidad del algoritmo en sistemas reales y guía el diseño de soluciones computacionales eficientes [53].

Finitud

La finitud indica que un algoritmo debe tener un número limitado de pasos y que su ejecución debe terminar necesariamente [12]. Este principio permite garantizar que los programas no queden atrapados en ciclos infinitos ni consuman recursos indefinidamente, problema recurrente en sistemas mal diseñados [3]. La literatura de algoritmia enfatiza que cualquier algoritmo que no garantice finitud no es computacionalmente válido [53].

Ejemplos clásicos incluyen algoritmos iterativos que usan condiciones de parada explícitas y algoritmos recursivos que definen casos base que permiten terminar la ejecución [54]. Por ejemplo, el cálculo del factorial de un número natural finaliza cuando se alcanza el caso base $0! = 1$, garantizando finitud y correctitud [32]. Esta propiedad debe verificarse durante el diseño para evitar comportamientos anómalos en ejecución [12].

El análisis de finitud también se aplica en validaciones formales mediante métodos matemáticos que permiten demostrar que el algoritmo no continuará indefinidamente, práctica documentada en estudios de verificación y análisis de programas [3]. Esta verificación es crucial en sistemas críticos, como control industrial o aeronáutica [53].

Determinismo

El determinismo establece que cada paso de un algoritmo debe definir una acción única y claramente especificada, de modo que no exista ambigüedad en su interpretación o ejecución [12]. Esta propiedad garantiza que, para un mismo conjunto de entradas, el algoritmo producirá siempre las mismas salidas siguiendo el mismo camino de ejecución, característica fundamental en sistemas predecibles y auditablemente correctos [3]. La ausencia de determinismo puede introducir incertidumbre en el comportamiento del sistema, generando riesgos en aplicaciones críticas como sistemas bancarios, médicos o aeronáuticos [53].

En la práctica, el determinismo implica que las operaciones deben estar definidas con precisión, evitando expresiones ambiguas como "ordenar adecuadamente" o "seleccionar un elemento adecuado", que no especifican una acción concreta [54]. Los lenguajes de programación refuerzan este principio mediante reglas estrictas de sintaxis y semántica que permiten traducir instrucciones a operaciones de máquina sin interpretaciones inconsistentes [32]. La claridad fomentada por el determinismo mejora la mantenibilidad y facilita la verificación del software [12].

Asimismo, el determinismo posibilita la reproducibilidad, condición necesaria para validar algoritmos mediante pruebas y análisis formal [3]. Si un algoritmo no es determinista, los resultados podrían variar entre ejecuciones incluso bajo condiciones idénticas, dificultando su verificación y su uso en contextos que requieren alta confiabilidad [53]. Por estas razones, el determinismo constituye un pilar en la teoría clásica de algoritmos.

Efectividad

La efectividad se refiere a que cada operación descrita en un algoritmo debe ser realizable mediante un conjunto elemental de acciones ejecutables por un agente computacional [12]. Esto implica que las instrucciones deben ser suficientemente simples como para ser implementadas en hardware o software real, y deben garantizar que la acción especificada produzca el efecto esperado sobre los datos [3]. La efectividad evita la presencia de pasos vagos o imposibles de ejecutar, lo cual fortalecería la inconsistencia del algoritmo [53].

Los algoritmos efectivos se basan en operaciones elementales tales como asignación, comparación, aritmética básica, lectura y escritura, todas ellas compatibles con la arquitectura del computador descrita por modelos como la arquitectura de Von Neumann [54]. Por ejemplo, un algoritmo que requiera "adivinar el resultado adecuado" no sería efectivo porque no define una operación mecanizable; en cambio, un algoritmo que busque un

elemento realizando comparaciones secuenciales sí lo es [32]. La efectividad garantiza que un algoritmo pueda ser concretado en un programa real.

Además, la efectividad contribuye al tiempo de ejecución razonable del algoritmo, dado que las operaciones elementales permiten prever el costo computacional de cada paso [12]. Esta característica se vincula estrechamente con el análisis de eficiencia, pues los algoritmos efectivos pueden evaluarse más fácilmente utilizando modelos formales de complejidad temporal y espacial [53]. En consecuencia, la efectividad es indispensable para asegurar que el algoritmo sea útil en la práctica.

1.8.3. Correctitud y eficiencia

La correctitud de un algoritmo implica que produce salidas válidas para todas las entradas válidas y que cumple con la especificación formal del problema [12]. Esta propiedad se fundamenta en la verificación de que cada paso contribuye exactamente al resultado esperado y de que las condiciones previas y posteriores a cada operación son coherentes con los requisitos del sistema [3]. La correctitud es un prerrequisito esencial, especialmente en sistemas donde los errores pueden provocar fallas graves o pérdidas económicas significativas [53].

La eficiencia, por su parte, se relaciona con el uso óptimo de recursos computacionales, principalmente tiempo de ejecución y memoria [54]. Algoritmos que realizan la misma tarea pueden diferir radicalmente en eficiencia; por ejemplo, ordenar una lista mediante selección tiene complejidad $O(n^2)$, mientras que algoritmos como Quicksort o Mergesort logran $O(n \log n)$ [12]. La eficiencia determina la escalabilidad de una solución y su aplicabilidad en escenarios de grandes volúmenes de datos [3].

La combinación de correctitud y eficiencia constituye el fundamento de la calidad algorítmica. Un algoritmo correcto pero ineficiente podría ser impracticable, mientras que uno eficiente pero incorrecto es inútil o peligroso [53]. Por ello, el diseño de algoritmos exige equilibrar claridad, precisión y rendimiento desde las etapas iniciales del proceso de programación [12].

Corrección parcial y total

La corrección parcial establece que, si el algoritmo termina, entonces su resultado es correcto de acuerdo con la especificación formal [3]. Este tipo de corrección verifica la validez del resultado, pero no garantiza que el algoritmo siempre concluya su ejecución

[12]. Un algoritmo con corrección parcial podría presentar ciclos infinitos bajo ciertas condiciones, lo que hace necesario evaluar adicionalmente su finitud [53].

La corrección total combina la corrección parcial con la finitud, exigiendo que el algoritmo siempre termine y que lo haga produciendo el resultado correcto [54]. Esta propiedad es más estricta y se considera el estándar de calidad en sistemas donde la terminación es crítica, como en control industrial, sistemas bancarios y aplicaciones embebidas [3]. La literatura enfatiza que demostrar corrección total requiere analizar tanto la validez lógica de los pasos como las condiciones de terminación [12].

La verificación formal de algoritmos mediante métodos matemáticos, invariantes de bucles y pruebas estructuradas permite establecer corrección parcial y total con rigor científico [53]. Esta verificación es esencial en contextos de alta confiabilidad y se apoya en herramientas computacionales que automatizan parte del proceso [3]. Al aplicar estas técnicas, se garantiza que un algoritmo sea fiable, predecible y adecuado para sistemas complejos [12].

Eficiencia intuitiva

La eficiencia intuitiva se refiere a la capacidad del programador para estimar el rendimiento de un algoritmo basándose en su estructura, sin necesidad inmediata de análisis formal [12]. Esta intuición permite identificar rápidamente soluciones potencialmente ineficientes, como bucles anidados innecesarios o recorridos repetitivos de estructuras de datos [3]. Aunque no reemplaza el análisis riguroso, la intuición sobre eficiencia guía la selección preliminar de técnicas algorítmicas [53].

Por ejemplo, un programador reconoce intuitivamente que ordenar una lista dos veces es menos eficiente que ordenar una sola vez, o que una búsqueda lineal es menos eficiente que una búsqueda binaria en listas ordenadas [54]. Este conocimiento se adquiere mediante experiencia, práctica y estudio de patrones de solución comunes [32]. La eficiencia intuitiva permite mejorar el rendimiento desde etapas tempranas de diseño antes de confirmar mediante análisis formal [12].

Finalmente, la eficiencia intuitiva promueve la toma de decisiones informadas sobre estructuras de datos y técnicas de programación, contribuyendo al desarrollo de algoritmos más robustos y escalables [3]. La literatura enfatiza que los programadores deben combinar intuición con herramientas formales como notación Big-O y análisis empírico para lograr soluciones equilibradas y eficientes [53].

1.9. Lenguajes formales y lenguajes de programación

Los lenguajes formales proporcionan reglas sintácticas y semánticas que definen cómo deben estructurarse las expresiones y cómo debe interpretarse su significado, constituyendo la base teórica de los lenguajes de programación [54]. Estos lenguajes permiten especificar algoritmos de manera precisa y verificable mediante gramáticas, alfabetos, producciones y modelos matemáticos, como autómatas y máquinas de Turing [3]. Su estudio es fundamental para comprender cómo los programas son analizados y traducidos por compiladores e intérpretes [53].

Los lenguajes de programación, construidos sobre los principios de los lenguajes formales, permiten implementar algoritmos en una forma que pueda ejecutarse en una computadora [32]. Cada lenguaje incorpora reglas sintácticas estrictas y modelos semánticos que describen cómo deben ejecutarse las instrucciones, asegurando predictibilidad y consistencia en los resultados [12]. La elección de un lenguaje depende de factores como el dominio del problema, la eficiencia requerida, la portabilidad y la facilidad de mantenimiento [3].

1.9.1. Sintaxis

La sintaxis describe la estructura válida de instrucciones, expresiones y programas dentro de un lenguaje de programación, definiendo cómo deben organizarse los símbolos para formar construcciones correctas [54]. Los compiladores realizan análisis sintáctico para verificar que el código cumpla con estas reglas antes de traducirlo a instrucciones ejecutables, proceso ampliamente documentado en textos de construcción de compiladores [53]. La sintaxis evita ambigüedades y garantiza que el código sea comprensible por la máquina y por otros programadores [32].

Errores sintácticos como falta de paréntesis, comas incorrectas o instrucciones mal formadas son detectados en esta etapa, permitiendo corregirlos antes de la ejecución [12]. La claridad sintáctica contribuye a la mantenibilidad y legibilidad del código, y es un componente esencial en la calidad del software [3]. Por ello, los lenguajes modernos incorporan reglas explícitas y herramientas que ayudan a detectar errores sintácticos de forma temprana [53].

Reglas gramaticales

Las reglas gramaticales de un lenguaje de programación describen, mediante una gramática formal, cómo pueden combinarse los símbolos del lenguaje (palabras clave, iden-

tificadores, operadores, delimitadores) para formar construcciones válidas como sentencias, bloques y programas completos [54]. Estas gramáticas suelen especificarse en notaciones como BNF (Backus–Naur Form) o EBNF (Extended BNF), las cuales permiten expresar producciones del tipo `sentencia` ::= `if expresion then sentencia` [53]. El uso de gramáticas formales facilita el análisis automático del código por parte de compiladores e intérpretes [3].

Los analizadores sintácticos (*parsers*) utilizan estas reglas gramaticales para construir árboles de derivación o árboles sintácticos que representan la estructura jerárquica del programa [12]. Por ejemplo, una expresión aritmética como `a + b * c` puede representarse como un árbol donde la multiplicación tiene mayor precedencia que la suma, lo cual refleja las reglas gramaticales del lenguaje [54]. La correcta definición de estas reglas evita ambigüedades y garantiza interpretaciones consistentes del código [53].

En la literatura se enfatiza que las gramáticas de los lenguajes modernos se diseñan para ser no ambiguas o, al menos, para que la ambigüedad se resuelva mediante reglas adicionales de precedencia y asociatividad [3]. Esto permite que el mismo programa sea interpretado de igual manera por diferentes compiladores conformes al estándar del lenguaje [12]. La precisión gramatical es por tanto un requisito para la portabilidad y la confiabilidad del software [53].

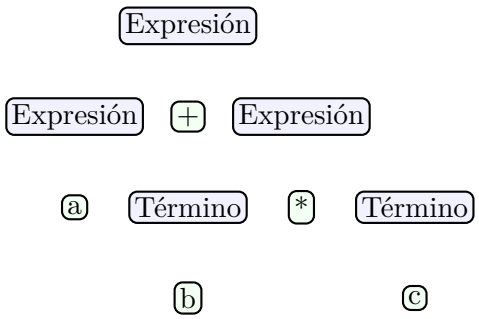

Figura 1.16: Árbol sintáctico para la expresión `a + b * c` [53], [54].

Formación de expresiones

La formación de expresiones en un lenguaje de programación se rige por reglas que especifican cómo combinar operandos y operadores para producir valores [32]. Estas reglas definen qué operadores son válidos para cada tipo de dato, cuál es su precedencia y cómo se asocian en ausencia de paréntesis explícitos [54]. Por ejemplo, en la mayoría de lenguajes, la multiplicación tiene mayor precedencia que la suma, y operadores binarios como `+` o `-`

suelen ser asociativos por la izquierda [53].

Las expresiones no se limitan a operaciones aritméticas; incluyen también expresiones lógicas, relacionales y de concatenación de cadenas, cada una con semánticas específicas [3]. Así, una expresión como `(edad > 18) && (pais == "ECUADOR")` combina operadores relacionales y lógicos para producir un valor booleano que controla el flujo de ejecución [32]. La correcta comprensión de estas reglas es esencial para evitar errores sutiles en condiciones y decisiones [12].

Además, muchos lenguajes permiten sobrecarga de operadores, lo cual significa que el mismo símbolo puede tener significados distintos según los tipos de sus operandos (por ejemplo, `+` como suma numérica o concatenación de cadenas) [54]. Esta característica, aunque poderosa, puede introducir confusión si no se documenta y utiliza adecuadamente [3]. La literatura recomienda emplear la sobrecarga con moderación y claridad para preservar la legibilidad [12].

Las expresiones también interactúan con conversiones implícitas y explícitas de tipos ("casting"), por lo que una formación incorrecta puede producir pérdida de información o resultados inesperados [53]. Por ejemplo, dividir dos enteros puede generar un resultado entero truncado, mientras que dividir números de punto flotante conserva los decimales [32]. Estas diferencias deben considerarse cuidadosamente al diseñar algoritmos y escribir código [12].

1.9.2. Semántica

La semántica de un lenguaje de programación describe el significado de las construcciones sintácticamente válidas, indicando qué efecto tienen sobre el estado del programa y del sistema al ejecutarse [54]. Mientras la sintaxis responde a la pregunta "¿está bien formada esta instrucción?", la semántica responde "¿qué hace esta instrucción?", tal como se discute en la literatura de lenguajes formales y compiladores [53]. La semántica puede describirse de manera informal (texto natural), operacional (mediante máquinas abstractas) o denotacional (mediante funciones matemáticas) [3].

Comprender la semántica es fundamental para razonar sobre la corrección y el comportamiento del programa, especialmente en presencia de estructuras de control, llamadas a funciones, recursión y manejo de memoria [12]. Por ejemplo, dos fragmentos de código pueden ser sintácticamente válidos pero semánticamente diferentes si modifican variables de formas distintas o si manejan errores de manera divergente [32]. La semántica precisa permite predecir el efecto de las instrucciones y verificar la adherencia a los requisitos del

sistema [3].

Significado operativo

El significado operativo describe la semántica de un programa en términos de los pasos que realiza una máquina abstracta al ejecutar sus instrucciones [54]. Esta perspectiva se formaliza mediante la definición de estados del programa (valores de variables, posición en el código, contenido de memoria) y reglas de transición que indican cómo cada instrucción transforma dicho estado [53]. De este modo, es posible modelar la ejecución como una secuencia de estados que evoluciona siguiendo la lógica del programa [3].

Por ejemplo, una sentencia de asignación `x = x + 1;` puede interpretarse como una transición desde un estado en el que `x = n` a otro en el que `x = n + 1`, manteniendo constantes el resto de variables [12]. De manera similar, una instrucción condicional define bifurcaciones en el espacio de estados según el valor de la condición evaluada [54]. Esta visión facilita el análisis de flujos de control, la detección de caminos no alcanzables y la verificación de propiedades como invariantes de bucle [53].

El significado operativo se utiliza tanto en la especificación formal de lenguajes como en herramientas de verificación y depuración, donde se simulan estados y transiciones para detectar errores [3]. La comprensión de esta semántica permite a los programadores anticipar cómo se comportará el código en diferentes escenarios de ejecución, contribuyendo a un diseño más robusto y seguro [12].

Errores semánticos

Los errores semánticos ocurren cuando el programa es sintácticamente correcto pero su comportamiento no coincide con la intención del programador o con la especificación del problema [32]. Por ejemplo, utilizar el operador de asignación en lugar del operador de comparación dentro de una condición, o aplicar la fórmula errónea para un cálculo, produce resultados incorrectos sin que el compilador detecte necesariamente un fallo [12]. Estos errores son especialmente peligrosos porque pueden pasar inadvertidos hasta que se observan resultados inconsistentes en producción [3].

La detección de errores semánticos requiere análisis cuidadoso, pruebas exhaustivas y, en algunos casos, revisión por pares [53]. Las técnicas de pruebas de caja negra y caja blanca, combinadas con revisiones de código y análisis estático, ayudan a identificar discrepancias entre el comportamiento observado y el comportamiento esperado [12]. La comprensión profunda de la semántica del lenguaje y del dominio del problema es un factor clave para

prevenir y corregir estos errores [3].

1.9.3. Errores comunes en principiantes

Los estudiantes que se inician en la programación suelen cometer errores recurrentes derivados de dificultades para comprender la sintaxis, la semántica y la lógica de los programas [32]. Entre los más frecuentes se encuentran omitir delimitadores, confundir operadores, utilizar tipos de datos inadecuados, diseñar condiciones incorrectas y olvidar casos especiales en estructuras de control [3]. La literatura sobre enseñanza de la programación resalta la importancia de abordar explícitamente estos errores para favorecer el desarrollo de competencias sólidas [12].

Además, muchos principiantes tienden a centrarse en "hacer que el programa funcione" sin analizar si la solución es correcta, eficiente o mantenible [32]. Esta actitud puede conducir a código difícil de entender, duplicación de lógica y errores que reaparecen al introducir cambios [3]. Por ello, se recomienda enfatizar buenas prácticas desde los primeros cursos, incluyendo trazado de código, diseño previo, pruebas sistemáticas y reflexión sobre la solución [12].

Errores sintácticos

Los errores sintácticos se producen cuando el código viola las reglas de formación del lenguaje, como falta de paréntesis, llaves desbalanceadas, palabras clave mal escritas o uso incorrecto de signos de puntuación [32]. Estos errores impiden que el compilador o intérprete genere código ejecutable y suelen acompañarse de mensajes de error que indican la ubicación aproximada del problema [53]. En general, los errores sintácticos se detectan en fases tempranas del proceso de compilación [54].

Los principiantes a menudo encuentran dificultades para interpretar los mensajes de error del compilador, que pueden referirse a la línea detectada aunque el problema se origine en líneas previas [32]. La literatura didáctica recomienda desarrollar el hábito de revisar cuidadosamente el código cercano a la posición indicada, buscando patrones comunes como delimitadores faltantes o mal ubicados [3]. Comprender la relación entre gramática del lenguaje y mensajes de error facilita la corrección [12].

Para reducir errores sintácticos, se aconseja utilizar entornos de desarrollo que proporcionen resaltado de sintaxis, autocompletado y verificación incremental del código [53]. Estas herramientas ayudan a detectar problemas mientras se escribe el programa, evitando

acumulación de errores y mejorando la productividad [12]. No obstante, la responsabilidad última de comprender la sintaxis recae en el programador [32].

Errores lógicos

Los errores lógicos ocurren cuando el programa se ejecuta sin generar errores sintácticos o de tiempo de ejecución, pero produce resultados incorrectos debido a fallos en el razonamiento o en el diseño de la solución [32]. Estos errores pueden surgir por condiciones mal formuladas, bucles que no cubren todos los casos, cálculos erróneos o uso inadecuado de variables [12]. A diferencia de los errores sintácticos, los errores lógicos no son detectados directamente por el compilador [3].

Un ejemplo típico de error lógico es utilizar un operador relacional equivocado en una condición, como `>` en lugar de `>=`, lo que excluye un caso límite importante [32]. Otro caso frecuente es inicializar mal una variable de acumulación o contador, lo que afecta el resultado final de un cálculo iterativo [3]. Estos errores pueden permanecer ocultos si las pruebas no cubren adecuadamente los casos de frontera [12].

La literatura sobre buenas prácticas recomienda utilizar técnicas de trazado de código (*dry run*), depuración paso a paso y registro de valores intermedios para identificar el punto exacto en el que la lógica se desvía de lo esperado [32]. Estas estrategias permiten comparar el comportamiento real del programa con el comportamiento previsto en el diseño algorítmico [12]. En entornos educativos, el trazado manual es una herramienta clave para desarrollar comprensión profunda de la ejecución [3].

Además, el uso de pruebas sistemáticas, incluyendo casos típicos, extremos y casos erróneos, es esencial para detectar errores lógicos [12]. Las pruebas de caja negra se centran en la relación entrada–salida, mientras que las pruebas de caja blanca analizan rutas internas del código [3]. La combinación de ambas perspectivas incrementa la probabilidad de descubrir errores lógicos antes del despliegue [53].

Finalmente, la prevención de errores lógicos se apoya en un buen diseño algorítmico, en la claridad del código y en la revisión por pares [12]. Documentar decisiones, añadir comentarios significativos y mantener un estilo coherente facilitan la detección temprana de inconsistencias [3]. De este modo, la formación en pensamiento crítico y en metodologías de prueba resulta tan importante como el aprendizaje de la sintaxis del lenguaje [32].

1.10. Buenas prácticas para aprender programación

Aprender programación requiere no solo dominar la sintaxis de un lenguaje, sino también adoptar estrategias de estudio y prácticas sistemáticas que favorezcan el desarrollo del pensamiento algorítmico, la comprensión profunda del código y la capacidad de resolver problemas de forma estructurada [32]. La literatura en educación en computación destaca que los estudiantes que aplican métodos activos como trazado manual, depuración reflexiva y análisis de errores alcanzan niveles superiores de entendimiento y retención [3]. Asimismo, el aprendizaje colaborativo, el uso de herramientas de apoyo y la aceptación del error como parte natural del proceso son aspectos fundamentales para consolidar la competencia programadora [12], [53].

1.10.1. Métodos de estudio

Los métodos de estudio en programación deben orientarse hacia la comprensión de los conceptos fundamentales que subyacen al código, evitando la memorización superficial que conduce a errores y dificultades para resolver problemas novedosos [32]. La investigación educativa sugiere que las estrategias efectivas incluyen leer y analizar código antes de ejecutarlo, trazar el estado de las variables, realizar ejercicios progresivos y reflexionar sobre los resultados [3]. Estas prácticas fomentan el razonamiento lógico y la interpretación de algoritmos, elementos centrales en la formación de programadores [12].

Asimismo, se recomienda alternar entre estudio teórico y práctica deliberada, resolviendo ejercicios de complejidad creciente y comparando distintas aproximaciones a un mismo problema [53]. Esta combinación permite integrar conocimientos y desarrollar intuición algorítmica, habilidades que fortalecen la autonomía del programador en formación y su capacidad para enfrentar desafíos más avanzados [3]. El uso de técnicas reflexivas, como explicar en voz alta el funcionamiento de un programa, ha mostrado efectos positivos en la retención y comprensión [32].

Trazar código

Trazar código consiste en simular mental o manualmente la ejecución de un programa, registrando los cambios en las variables y el flujo de control paso a paso [32]. Esta técnica es fundamental para detectar errores lógicos, comprender estructuras de control y verificar la coherencia entre el algoritmo y su implementación [3]. La literatura destaca que el trazado permite visualizar la dinámica del programa, facilitando la anticipación de comportamientos

imprevistos [12].

El uso sistemático del trazado ayuda a comprender cómo se modifican los valores, cómo se recorren estructuras de datos y cómo se activan y desactivan diferentes ramas de ejecución [53]. Por ejemplo, en un bucle que calcula la suma de los primeros n números, trazar el estado de las variables `i` y `suma` permite verificar si las actualizaciones se realizan correctamente en cada iteración [32]. Este análisis detallado contribuye a desarrollar habilidades de depuración y razonamiento algorítmico [3].

El trazado también es útil para comparar la ejecución real del programa con la esperada según el diseño algorítmico [12]. Esta comparación revela discrepancias entre implementación e intención, lo que facilita corregir errores y mejorar la calidad del código [53]. Por ello, se considera una de las prácticas centrales en los cursos iniciales de programación [32].

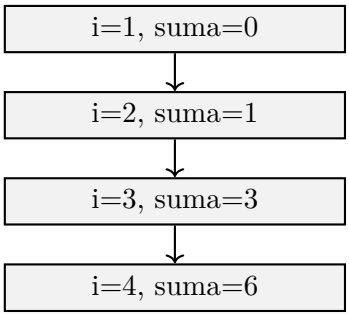

Figura 1.17: Trazado de un algoritmo que suma los primeros n números naturales [3], [32].

Escribir antes que ejecutar

La práctica de escribir el algoritmo o el pseudocódigo antes de ejecutar el programa fomenta claridad conceptual y reduce errores comunes asociados a improvisación o dependencia excesiva del entorno de ejecución [12]. La literatura subraya que esta estrategia permite a los estudiantes organizar sus ideas, prever la estructura del programa y anticipar la interacción entre variables y estructuras de control [3]. De este modo, escribir antes de ejecutar fortalece la planificación lógica previa a la codificación [53].

Cuando los principiantes ejecutan el código sin un diseño previo, tienden a introducir cambios impulsivos que complican la detección de errores y deterioran la estructura del programa [32]. Al escribir primero, se puede analizar la solución con calma, aplicar principios de diseño modular y simplificar pasos innecesarios [12]. Este enfoque incrementa la calidad del código desde sus primeras versiones, disminuyendo la cantidad de errores arrastrados a etapas posteriores [3].

Además, escribir antes de ejecutar promueve la reflexión sobre alternativas algorítmicas, permitiendo comparar diferentes formas de resolver un mismo problema [53]. Por ejemplo, antes de implementar una búsqueda, el programador en formación puede evaluar si utilizará búsqueda lineal o binaria, dependiendo del tamaño y la estructura de los datos [54]. Esta reflexión previa contribuye al aprendizaje de patrones de solución y a la mejora de la eficiencia [12].

Finalmente, esta práctica facilita la comunicación y la revisión por pares, ya que el pseudocódigo es más accesible y menos dependiente de detalles del lenguaje que el código fuente completo [3]. Las revisiones tempranas basadas en pseudocódigo permiten detectar errores conceptuales sin el ruido adicional de la sintaxis [32]. Por estas razones, escribir antes de ejecutar es considerada una buena práctica tanto en contextos educativos como profesionales [12].

1.10.2. La importancia del error

El error desempeña un papel fundamental en el aprendizaje de la programación, ya que permite identificar malentendidos, explorar límites conceptuales y desarrollar habilidades de depuración [32]. La literatura educativa establece que los estudiantes que interpretan el error como una oportunidad de aprendizaje, y no como un fracaso, adquieren competencias más sólidas y persistentes en pensamiento computacional [3]. Las técnicas de depuración, análisis y revisión de código se apoyan directamente en esta disposición positiva frente al error [12].

Además, el error proporciona retroalimentación valiosa que guía la reestructuración de conceptos y corrige modelos mentales incorrectos [53]. Al equivocarse, el programador en formación debe revisar su razonamiento, contrastarlo con el comportamiento real del programa y ajustar su comprensión de la lógica computacional [12]. Este ciclo de autoevaluación y mejora continua es esencial tanto en entornos académicos como en el desarrollo profesional de software [3].

1.10.3. Trabajo colaborativo

El trabajo colaborativo favorece la construcción conjunta de conocimiento y mejora la capacidad de resolver problemas mediante discusión, análisis compartido y revisión mutua [3]. En programación, la colaboración permite que los estudiantes contrasten sus soluciones, detecten errores que individualmente pasarían desapercibidos y aprendan diferentes estilos

y estrategias de razonamiento computacional [12]. Las metodologías ágiles y el desarrollo en pareja (*pair programming*) se apoyan en estos principios colaborativos [53].

Asimismo, la colaboración desarrolla competencias transversales como comunicación efectiva, negociación de ideas, organización de tareas y gestión conjunta del tiempo [3]. Estas habilidades resultan esenciales en entornos profesionales de desarrollo de software, donde la construcción de sistemas complejos requiere coordinación entre múltiples roles y equipos [12]. La literatura señala que el trabajo colaborativo incrementa significativamente la calidad del código y la motivación de los estudiantes [32].

1.10.4. Herramientas de apoyo

Las herramientas de apoyo facilitan el aprendizaje de la programación mediante asistencia sintáctica, ejecución controlada, visualización de estructuras y automatización de pruebas [32]. Entre las más utilizadas se encuentran los entornos de desarrollo integrado (IDE), que ofrecen resaltado de sintaxis, autocompletado, depuración paso a paso y monitoreo del estado de variables [12]. Estas funcionalidades permiten detectar errores más rápidamente y comprender el flujo de ejecución del programa [3].

Además, herramientas especializadas como simuladores de memoria, visualizadores de algoritmos y plataformas de aprendizaje interactivo ayudan a los estudiantes a desarrollar una comprensión más profunda de conceptos internos como pilas, colas, memoria dinámica o recursión [53]. Estas herramientas permiten observar visualmente lo que ocurre durante la ejecución, facilitando la relación entre teoría y práctica [32]. La literatura destaca que las visualizaciones incrementan la retención y la comprensión conceptual [3].

Los sistemas de control de versiones, como Git, también constituyen herramientas esenciales que permiten gestionar cambios, colaborar en proyectos, explorar alternativas y revertir errores [12]. Su uso temprano en la formación ayuda a fomentar disciplina técnica, trazabilidad y orden en el desarrollo [3]. De este modo, las herramientas de apoyo complementan el aprendizaje teórico y fortalecen las competencias prácticas en programación [53].

Capítulo 2

Algoritmos y Representación Estructurada

Los algoritmos constituyen la base operativa de todo sistema computacional, pues describen de forma precisa y finita la secuencia de pasos necesaria para transformar datos de entrada en resultados válidos. Su estudio permite comprender cómo los sistemas realizan procesos de decisión, cálculo y control, integrándose con los fundamentos conceptuales presentados en el capítulo anterior, donde se estableció el papel central de la información y su procesamiento en los sistemas computacionales [3], [12], [55]. La formalización algorítmica, además, posibilita la automatización fiable de tareas y asegura que las soluciones obtenidas sean reproducibles, verificables y coherentes bajo distintos escenarios de ejecución [49], [54].

2.1. Terminología básica de algoritmos

El análisis formal de algoritmos requiere un conjunto de conceptos fundamentales que permiten describir, con precisión, los elementos que intervienen en la resolución automática de problemas. Estas nociones son compartidas en la literatura moderna de ingeniería de software, ciencias de la computación y diseño de sistemas, donde la claridad terminológica es esencial para garantizar la consistencia y verificabilidad de los procesos [3], [12]. A través de esta sección se definen los componentes básicos que sustentan la representación estructurada de algoritmos y que habilitan su implementación posterior en lenguajes de programación.

En un nivel conceptual, los algoritmos se consideran mecanismos de transformación que actúan sobre representaciones simbólicas de datos, siguiendo reglas bien definidas

y ordenadas de forma explícita [54], [55]. Este enfoque coincide con los principios de formalización empleados en teoría de autómatas, modelos computacionales y metodologías de diseño, donde se enfatiza la importancia de describir cada paso como una operación elemental e inequívoca. La existencia de una terminología estandarizada permite establecer equivalencias entre descripciones informales, pseudocódigo y lenguajes estructurados.

Finalmente, estos conceptos fundamentales se conectan con la estructura lógica que compone todo algoritmo, la cual se desarrolla en secciones posteriores. Dicha estructura—secuencia, selección e iteración—ha sido ampliamente discutida en la literatura y constituye el fundamento de los lenguajes imperativos modernos [12], [23]. Con ello se establece un marco conceptual coherente para interpretar, analizar y diseñar algoritmos robustos, eficientes y verificables.

2.1.1. Instancia del problema

Una *instancia del problema* se define como un conjunto específico de valores de entrada para los cuales se debe ejecutar un algoritmo con el fin de producir un resultado concreto. Mientras que el problema describe la tarea de forma general, la instancia delimita un caso particular, permitiendo evaluar el comportamiento del algoritmo bajo condiciones específicas [3]. Esta distinción es esencial para analizar la corrección y el rendimiento, ya que un mismo algoritmo puede comportarse de manera distinta dependiendo de los datos sobre los que opera.

La noción de instancia es ampliamente utilizada en complejidad algorítmica, donde se estudia cómo varía el tiempo de ejecución o el uso de memoria conforme se modifican los tamaños de entrada [54]. También es fundamental en pruebas de software, pues permite seleccionar casos representativos —incluyendo valores normales, extremos y fronteras— para validar que el algoritmo se comporte de forma consistente y segura [12]. En este sentido, una instancia no es solo un conjunto de datos, sino un elemento esencial del proceso de verificación.

Ejemplo 1: Consideremos el algoritmo que determina si una persona puede acceder a un recinto según su edad. El problema general es «verificar si la edad cumple el requisito mínimo». Una instancia concreta sería: *edad = 17 años*. El algoritmo evaluará esta entrada y producirá como salida «acceso denegado». El algoritmo formal puede ser como el que se muestra en el listado de código 2.1

Listing 2.1: Algoritmo para determinar si una persona puede acceder al recinto

```
Algoritmo VerificarAcceso
Entrada: edad
Salida: mensaje de acceso

1. Si edad >= 18 entonces
2.     mensaje <- "Acceso permitido"
3. SiNo
4.     mensaje <- "Acceso denegado"
5. FinSi
6. Devolver mensaje
FinAlgoritmo
```

Ejemplo 2: Para el problema general «calcular el máximo común divisor entre dos enteros», una instancia sería el par de valores $(84, 36)$. El algoritmo lista para ser implementado en el lenguaje de programación de la preferencia del lector se muestra en el listado de código 2.2. Al ejecutar el algoritmo de Euclides sobre esta instancia, el resultado obtenido será 12.

Listing 2.2: Cálculo del máximo común divisor entre dos números

```
Algoritmo CalcularMCD
Entrada: a, b
Salida: mcd

1. Mientras b != 0 hacer
2.     r <- a mod b
3.     a <- b
4.     b <- r
5. FinMientras
6. mcd <- a
7. Devolver mcd
FinAlgoritmo
```

Ejemplo 3: En el problema de «determinar si un usuario puede autenticar en un sistema», una instancia concreta se compone del usuario `juan.perez` y la contraseña ingresada. El algoritmo de autenticación procesará estos datos y decidirá si conceder o no el acceso(ver listado de código 2.3), siguiendo reglas formales definidas [55].

Listing 2.3: Algoritmo simplificado de autenticación de usuario

```
Algoritmo AutenticarUsuario
Entrada: usuarioIngresado, contraseniaIngresada
Salida: resultado

1. Recuperar contraseniaReal del sistema para usuarioIngresado
2. Si usuarioIngresado no existe entonces
3.     resultado <- "Usuario no encontrado"
4.     Devolver resultado
5. FinSi
6. Si contraseniaIngresada = contraseniaReal entonces
7.     resultado <- "Acceso concedido"
8. SiNo
9.     resultado <- "Acceso denegado"
10. FinSi
11. Devolver resultado
FinAlgoritmo
```

2.1.2. Estado y transición

El *estado* de un algoritmo se define como la configuración actual de todas las variables, estructuras de datos y parámetros relevantes en un momento determinado de su ejecución. Esta noción permite modelar el comportamiento computacional como una evolución ordenada, donde cada paso modifica el estado anterior produciendo uno nuevo, siguiendo reglas bien definidas [3]. La representación explícita del estado facilita el análisis de corrección, especialmente en problemas donde intervienen condiciones, ciclos o estructuras de control complejas.

Una *transición* describe el cambio de un estado a otro como consecuencia de ejecutar una operación elemental del algoritmo. Estas transiciones, que constituyen la dinámica interna del proceso computacional, deben ser determinísticas y reproducibles para asegurar que el algoritmo produzca siempre los mismos resultados ante la misma instancia del problema [12]. La teoría de sistemas y los modelos de autómatas formales utilizan el concepto de transición para caracterizar comportamientos verificables, lo que resulta fundamental para garantizar consistencia en el diseño de software.

En el contexto de los sistemas computacionales, el seguimiento del estado permite comprender cómo los datos fluyen y se transforman a lo largo del algoritmo, lo cual

coincide con los principios de representación y manipulación del conocimiento señalados por Silberschatz et al. [55] y por Spivak y Kent [49]. Esta visión es especialmente útil en la depuración, ya que los errores suelen manifestarse como estados inesperados o inconsistentes que no cumplen las condiciones definidas.

Ejemplo 1: Para un algoritmo que califica el desempeño de un estudiante según su nota, el estado inicial consiste en el valor ingresado para la nota. Tras evaluar si la nota es mayor o igual que 90, el estado cambia para reflejar la asignación de la categoría “Excelente”, tal como se muestra en el listado de código 2.4. Cada comparación y asignación representa una transición entre estados.

Listing 2.4: Algoritmo para clasificar el desempeño de un estudiante

```
Algoritmo ClasificarDesempeno
Entrada: nota
Salida: categoria

1. Si nota >= 90 entonces
2.     categoria <- "Excelente"
3. SiNo
4.     Si nota >= 80 entonces
5.         categoria <- "Muy bueno"
6.     SiNo
7.         Si nota >= 70 entonces
8.             categoria <- "Bueno"
9.         SiNo
10.            categoria <- "Insuficiente"
11.        FinSi
12.    FinSi
13. FinSi

14. Devolver categoria
FinAlgoritmo
```

Ejemplo 2: En el cálculo iterativo del factorial de un número, el estado está compuesto por la variable acumuladora y la variable de control del ciclo. Cada iteración produce una transición al multiplicar el acumulador por el valor actual del contador, reflejando la evolución ordenada del cálculo.

Listing 2.5: Algoritmo iterativo para calcular el factorial

```
Algoritmo CalcularFactorial
Entrada: n
Salida: resultado

1. resultado <- 1
2. contador <- 1

3. Mientras contador <= n hacer
4.     resultado <- resultado * contador
5.     contador <- contador + 1
6. FinMientras

7. Devolver resultado
FinAlgoritmo
```

Ejemplo 3: En un proceso de autenticación, el estado inicial contiene las credenciales ingresadas. Tras consultar la base de datos, se genera un nuevo estado que indica si el usuario existe. Posteriormente, otra transición determina si la contraseña coincide, produciendo un estado final que representa "Acceso concedido" o "Acceso denegado" [55]. En el listado de código 2.6 se muestra una de las opciones de un algoritmo formal para este problema.

Listing 2.6: Algoritmo de autenticación basado en cambios de estado

```
Algoritmo Autenticar
Entrada: usuarioIngresado, contraseniaIngresada
Salida: estadoFinal

1. estado <- "Credenciales recibidas"

2. Si usuarioIngresado no existe en la base de datos entonces
3.     estado <- "Usuario no encontrado"
4.     Devolver estado
5. FinSi

6. estado <- "Usuario encontrado"
7. Recuperar contraseniaReal del sistema

```

```
8. Si contraseniaIngresada = contraseniaReal entonces
9.     estado <- "Acceso concedido"
10. SiNo
11.    estado <- "Acceso denegado"
12. FinSi

13. Devolver estado
FinAlgoritmo
```

2.1.3. Operaciones básicas

Las operaciones básicas constituyen las unidades mínimas de acción de un algoritmo y representan los pasos elementales que modifican el estado del sistema de forma controlada y verificable. La literatura clásica en sistemas operativos y organización computacional describe estas operaciones como acciones atómicas que deben poder ejecutarse de manera determinista y sin ambigüedad [54], [55]. Entre ellas se encuentran la asignación, la comparación, el acceso a memoria, la lectura de entrada y la escritura de salida, todas fundamentales para la construcción de procesos más complejos.

En ingeniería de software, estas operaciones permiten descomponer algoritmos en pasos que pueden analizarse individualmente en términos de corrección, complejidad y efectos sobre las estructuras de datos [12]. Esta descomposición facilita el razonamiento estructurado y la verificación formal, ya que cada operación puede considerarse como una transición entre estados claramente definidos. Además, conocer el conjunto de operaciones básicas resulta esencial para identificar cuellos de botella y optimizaciones potenciales.

Desde la perspectiva del diseño algorítmico, la correcta identificación de las operaciones elementales permite modelar el comportamiento computacional de manera uniforme, manteniendo coherencia entre pseudocódigo, implementación y pruebas [3]. En particular, muchas técnicas de análisis —como el conteo de operaciones en complejidad temporal— dependen de reconocer qué acciones se consideran básicas y cuántas veces se ejecutan para una instancia del problema.

Estas operaciones también cumplen un rol fundamental en la interacción con el entorno, pues mediante la entrada y salida se establece la comunicación entre los datos iniciales y los resultados obtenidos. La claridad en su especificación previene inconsistencias y asegura que el algoritmo responda adecuadamente a diferentes tipos de entrada, incluidas aquellas consideradas casos límite o valores no previstos [55]. Este enfoque robusto contribuye al

diseño de soluciones confiables que puedan integrarse en sistemas mayores.

Finalmente, las operaciones básicas permiten establecer analogías entre algoritmos de distintos dominios, dado que comparten una estructura común basada en manipular estados y producir transiciones ordenadas. Esto facilita la enseñanza, el análisis comparativo y la reutilización conceptual en distintas áreas de la computación, reforzando la universalidad y aplicabilidad de los principios algorítmicos [12].

Asignación simple

La asignación simple es la operación mediante la cual se establece un valor específico a una variable, representando una transición directa del estado del algoritmo. Esta operación es fundamental porque constituye el mecanismo primario de almacenamiento y actualización de información durante la ejecución [3], [12]. En términos computacionales, la asignación simple se considera una instrucción elemental que suele ejecutarse en tiempo constante, razón por la cual desempeña un rol clave en el razonamiento algorítmico [53], [54]. Para principiantes, comprender esta operación implica reconocer que el símbolo de asignación no representa igualdad matemática, sino transferencia de valor, lo que establece una diferencia conceptual necesaria en lenguajes de programación.

Ejemplo: Si una variable `precio` almacena el costo de un producto, entonces la instrucción `precio <- 12.50` establece el estado inicial del sistema respecto al valor monetario. Posteriormente, todas las operaciones que utilicen `precio` dependerán de esta asignación.

Ejercicio: Escriba un algoritmo que defina las variables `nombre`, `edad` y `ciudad`, asignando a cada una un valor inicial. El algoritmo debe imprimir los valores asignados.

Listing 2.7: Algoritmo que define variables y muestra sus valores iniciales

```
Algoritmo DefinirVariables
Entrada: nombre, edad, ciudad
Salida: impresión de los valores

1. nombre <- "Juan"
2. edad <- 25
3. ciudad <- "Quito"

4. Escribir "Nombre: ", nombre
5. Escribir "Edad: ", edad
```

```
11  6. Escribir "Ciudad: ", ciudad
12
13  FinAlgoritmo
```

Asignación compuesta

La asignación compuesta combina una operación aritmética con la actualización de una variable. Esta forma de asignación es relevante porque reduce la complejidad sintáctica y permite expresar de manera directa transformaciones del estado interno del algoritmo [12]. Silberschatz et al. [55] destacan que estas operaciones encapsulan el patrón general de leer, operar y escribir sobre la misma ubicación de memoria, facilitando la comprensión del flujo de cómputo. Para estudiantes principiantes, su uso permite familiarizarse con la noción de actualización incremental o acumulativa, frecuente en algoritmos iterativos.

Ejemplo 1: La instrucción `promedio <- suma / n` le asigna a la variable promedio se le asigna (almacena) el valor resultado de la división del valor almacenado en la variable `suma` para el valor almacenado en la variable `n`. Esta operación es común en algoritmos estadísticos como es cálculo del promedio. Para ello, se debió calcular la `suma` de los `n` valores que intervienen en este proceso.

Ejemplo 2: La instrucción `total <- total + 5` incrementa el valor almacenado en `total`, siendo equivalente a recuperar su valor previo, sumarle 5 y almacenar el resultado. Esta operación es común en cálculos financieros y en algoritmos de conteo.

Ejercicio: Diseñe un algoritmo que reciba un valor inicial `x` y lo incremente sucesivamente mediante asignaciones compuestas para representar el crecimiento acumulado de un ahorro.

Listing 2.8: Algoritmo que incrementa un valor x mediante asignaciones compuestas para modelar ahorro acumulado

```
1  Algoritmo AhorroAcumulado
2  Entrada: x, incremento, periodos
3  Salida: valor final del ahorro
4
5  1. Escribir "Ingrese el valor inicial del ahorro (x):"
6  2. Leer x
7
8  3. Escribir "Ingrese el valor del incremento periódico:"
9  4. Leer incremento
```

```
10
11  5. Escribir "Ingrese el número de periodos:"
12  6. Leer periodos
13
14  7. contador <- 1
15
16  8. Mientras contador <= periodos hacer
17  9.      x = x + incremento        // Asignación compuesta
18  10.    Escribir "Periodo ", contador, ": ", x
19  11.    contador <- contador + 1
20  12. FinMientras
21
22  13. Escribir "El valor final del ahorro es: ", x
23
24  FinAlgoritmo
```

Asignación en cascada

La asignación en cascada consiste en efectuar múltiples asignaciones dentro de una misma instrucción, permitiendo que varios identificadores reciban el mismo valor. Aunque su sintaxis depende del lenguaje, su comprensión conceptual favorece la construcción de estados iniciales consistentes [3]. Desde la perspectiva de eficiencia, Stallings [53] explica que esta operación puede optimizar el número de instrucciones ejecutadas, aunque su uso debe acompañarse de claridad semántica para evitar ambigüedades.

Ejemplo: Si se desea inicializar tres variables con el mismo valor, una instrucción como `a <- b <- c <- 0` establece un estado uniforme para todas ellas (cuando el lenguaje lo permite), lo cual es útil en algoritmos que requieren sincronización inicial.

Ejercicio 1: Construya un algoritmo que inicialice tres variables `nota1`, `nota2` y `notaFinal` en cero utilizando asignación en cascada.

Listing 2.9: Algoritmo que inicializa variables usando asignación en cascada

```
1  Algoritmo InicializarNotas
2  Entrada: nota1, nota2, notaFinal
3  Salida: variables inicializadas
4
5  1. nota1 <- nota2 <- notaFinal <- 0
```

```
6
7   2. Escribir "Valores inicializados:"
8   3. Escribir "nota1 = ", nota1
9   4. Escribir "nota2 = ", nota2
10  5. Escribir "notaFinal = ", notaFinal
11
12  FinAlgoritmo
```

Ejercicio 2: Diseñe un algoritmo que solicite al usuario una temperatura expresada en grados Celsius mediante una operación de entrada. El valor ingresado deberá asignarse en cascada a varias variables con el propósito de preservar intacto el dato original y, simultáneamente, disponer de copias que permitan calcular su equivalente en otras escalas de temperatura. A partir de dicho valor inicial, el algoritmo deberá obtener las conversiones correspondientes en grados Fahrenheit, Kelvin y Rankine, y finalmente mostrar en pantalla todas las temperaturas calculadas, manteniendo siempre el valor original como referencia.

Listing 2.10: Asignación en cascada aplicada a la conversión de temperaturas

```
1   Algoritmo ConversionTemperaturas
2   Entrada: celsiusOriginal, fahrenheit, kelvin, rankine
3   Salida: conversiones
4
5   1. Escribir "Ingrese la temperatura en grados Celsius:"
6   2. Leer tempC
7
8   // Asignación en cascada: todas las variables reciben el valor inicial leído
9   3. celsiusOriginal <- fahrenheit <- kelvin <- rankine <- tempC
10
11  // Uso posterior del valor original
12  4. fahrenheit <- (fahrenheit * 9/5) + 32
13  5. kelvin <- kelvin + 273.15
14  6. rankine <- (rankie + 273.15) * 9/5
15
16  7. Escribir "Temperatura original en Celsius: ", celsiusOriginal
17  8. Escribir "En Fahrenheit: ", fahrenheit
18  9. Escribir "En Kelvin: ", kelvin
19  10. Escribir "En Rankine: ", rankine
20
```

```
21  FinAlgoritmo
```

Asignación con actualización

Las operaciones de actualización, como incrementos y decrementos, constituyen patrones esenciales en construcción de ciclos, acumuladores y contadores. Estas operaciones expresan de manera compacta la modificación recurrente de un valor, lo cual es particularmente importante para algoritmos iterativos [12]. Tanenbaum y Austin [54] resaltan que estas operaciones representan instrucciones eficientes a nivel de hardware y reflejan la estrecha relación entre los modelos lógicos y los modelos físicos de ejecución. Para principiantes, dominar estas operaciones permite comprender el comportamiento evolutivo del estado en ciclos `Mientras` y `Para`.

Ejemplo: La instrucción `i <- i + 1` incrementa el valor del contador `i`, lo cual habilita el avance de iteraciones en algoritmos de recorrido.

Ejercicio: Escriba un algoritmo que, dado un número natural `n`, utilice una variable contadora que se incremente desde 1 hasta `n`, imprimiendo cada valor.

Listing 2.11: Algoritmo que imprime valores desde 1 hasta n utilizando un contador

```
1   Algoritmo ContadorHastaN
2   Entrada: n
3   Salida: impresión de valores desde 1 hasta n
4
5   1. Escribir "Ingrese un número natural n:"
6   2. Leer n
7
8   3. contador <- 1
9
10  4. Mientras contador <= n hacer
11  5.      Escribir contador
12  6.      contador <- contador + 1
13  7. FinMientras
14
15  FinAlgoritmo
```

Comparación por igualdad

La comparación por igualdad permite verificar si dos valores representan el mismo estado o condición, constituyendo la base de numerosas estructuras de control en algoritmos. Esta operación es esencial para la toma de decisiones, así como para validar entradas y detectar inconsistencias [3], [12]. En términos de diseño algorítmico, su correcta interpretación evita errores comunes al diferenciar asignación de comparación, una confusión típica entre estudiantes principiantes.

Ejercicio. Diseñe un algoritmo que determine si dos números ingresados por el usuario son iguales.

Listing 2.12: Algoritmo que determina si dos números ingresados son iguales

```
Algoritmo DeterminarIgualdad
Entrada: num1, num2
Salida: mensaje

1. Escribir "Ingrese el primer número:"
2. Leer num1

3. Escribir "Ingrese el segundo número:"
4. Leer num2

5. Si num1 = num2 entonces
6.     mensaje <- "Los números son iguales"
7. SiNo
8.     mensaje <- "Los números son diferentes"
9. FinSi

10. Escribir mensaje

FinAlgoritmo
```

El algoritmo que muestra el listado de código 2.12 permite ilustrar la operación de comparación por igualdad, que es una de las operaciones lógicas más utilizadas en programación. Tras solicitar dos valores numéricos al usuario, el algoritmo compara ambos mediante el operador = y genera un mensaje que indica si representan o no el mismo valor. Es un ejercicio fundamental para comprender el funcionamiento de las estructuras

condicionales.

Listing 2.13: Algoritmo que determina si dos números son divisibles el uno para el otro

```
Algoritmo DivisibilidadMutua
Entrada: num1, num2
Salida: mensaje

1. Escribir "Ingrese el primer número:"
2. Leer num1

3. Escribir "Ingrese el segundo número:"
4. Leer num2

// Determinar divisibilidad en ambos sentidos
5. Si (num2 <> 0) AND (num1 mod num2 = 0) entonces
6.     mensaje <- "num1 es divisible para num2."
7. SiNo
8.     Si (num1 <> 0) AND (num2 mod num1 = 0) entonces
9.         mensaje <- "num2 es divisible para num1."
10.    SiNo
11.        mensaje <- "No existe divisibilidad entre ellos."
12.    FinSi
13. FinSi

14. Escribir mensaje

FinAlgoritmo
```

El algoritmo que se muestra en el listado de código 2.13 evalúa la divisibilidad en ambos sentidos, ya que para dos números cualesquiera:

1. num1 es divisible para num2
2. num2 es divisible para num1
3. No existe divisibilidad entre ellos.

La comprobación se realiza mediante el operador módulo (`mod`), donde si `a mod b = 0`, entonces `b` divide para `a` sin residuo. Además, el algoritmo incluye una verificación simple para evitar división entre cero.

Comparación por desigualdad

La comparación por desigualdad evalúa si dos valores difieren, habilitando decisiones relevantes cuando el algoritmo debe reaccionar ante cambios o discrepancias [12]. Esta operación es particularmente utilizada en validación de contraseñas, identificación de errores o detección de casos especiales en algoritmos matemáticos.

Ejercicio. Construya un algoritmo que reciba un número y verifique si es distinto de cero. Uno de los conjuntos de pasos que dan solución a este problema se muestra en el listado de código 2.14.

Listing 2.14: Algoritmo que verifica si un número es distinto de cero

```
Algoritmo VerificarDistintoDeCero
Entrada: numero
Salida: mensaje

1. Escribir "Ingrese un número:"
2. Leer numero

3. Si numero <> 0 entonces
4.     mensaje <- "El número es distinto de cero"
5. SiNo
6.     mensaje <- "El número es igual a cero"
7. FinSi

8. Escribir mensaje

FinAlgoritmo
```

Comparaciones relacionales

Las comparaciones relacionales permiten analizar magnitudes entre valores, lo cual es fundamental en algoritmos que requieren ordenamiento, selección o evaluación de límites [7], [53]. Estas operaciones dan soporte a decisiones complejas y habilitan mecanismos de control adaptativos, especialmente en estructuras como ciclos y búsqueda lineal.

Ejercicio. Escriba un algoritmo que determine si un número ingresado está dentro del rango [10, 50].

Listing 2.15:]Algoritmo para determinar si un número está en el rango [10, 50]

```
Algoritmo VerificarRango
Entrada: numero
Salida: mensaje

1. Si numero >= 10 Y numero <= 50 entonces
2.     mensaje <- "El número está dentro del rango [10, 50]"
3. SiNo
4.     mensaje <- "El número está fuera del rango [10, 50]"
5. FinSi

6. Devolver mensaje
FinAlgoritmo
```

Operación de entrada

La operación de entrada permite que el algoritmo adquiera información desde el entorno, ya sea proporcionada por un usuario o por un dispositivo externo. Silberschatz et al. [55] destacan que la entrada constituye un punto crítico de interacción, ya que condiciona la validez de las operaciones posteriores. Desde la ingeniería del software, Pressman y Maxim [12] señalan la importancia de validar las entradas para prevenir inconsistencias o errores derivados de datos inesperados o mal formados.

Además, Sommerville [3] subraya la necesidad de diseñar mecanismos de entrada robustos y claros, especialmente en sistemas de información donde los datos constituyen el recurso principal. Para estudiantes principiantes, comprender esta operación implica reconocer que no se modifica el estado interno hasta que el valor ingresado se asigna a una variable, reforzando la separación entre adquisición y procesamiento.

Ejercicio: Diseñe un algoritmo que solicite al usuario su nombre y edad, verificando que la edad sea mayor a cero.

Listing 2.16: Algoritmo para solicitar nombre y edad, verificando que la edad sea mayor a cero

```
    Algoritmo SolicitarNombreEdad
    Entrada: nombre, edad
    Salida: mensaje

```

```
    1. Escribir "Ingrese su nombre:"
    2. Leer nombre

    3. Escribir "Ingrese su edad:"
    4. Leer edad

    5. Si edad > 0 entonces
    6.     mensaje <- "Datos válidos: " + nombre + ", " + edad + " años"
    7. SiNo
    8.     mensaje <- "Error: la edad debe ser mayor que cero"
    9. FinSi

    10. Devolver mensaje
    FinAlgoritmo
```

Operación de salida

La operación de salida permite comunicar resultados al usuario o enviar información a otro proceso, cumpliendo la función esencial de presentar el efecto computacional de las transformaciones realizadas. Silberschatz et al. [55] explican que las salidas deben ser comprensibles, oportunas y adecuadas al contexto, especialmente en sistemas interactivos. Pressman y Maxim [12] destacan que una salida mal diseñada puede inducir a errores de interpretación, incluso si el algoritmo interno es correcto.

Sommerville [3] recalca que la salida es un componente crítico en la experiencia del usuario, formando parte de los requisitos de usabilidad en ingeniería del software. Para estudiantes, esta operación ayuda a visualizar el efecto de cada instrucción en el algoritmo, facilitando la depuración y el análisis del flujo.

Ejercicio: Escriba un algoritmo que reciba dos números y muestre su suma, diferencia y producto.

Listing 2.17: Algoritmo que calcula suma, diferencia y producto de dos números

```
Algoritmo OperacionesBasicas
Entrada: num1, num2
Salida: suma, diferencia, producto

1. Escribir "Ingrese el primer número:"
```

```
  2. Leer num1

  3. Escribir "Ingrese el segundo número:"
  4. Leer num2

  5. suma <- num1 + num2
  6. diferencia <- num1 - num2
  7. producto <- num1 * num2

  8. Escribir "La suma es: ", suma
  9. Escribir "La diferencia es: ", diferencia
  10. Escribir "El producto es: ", producto

  FinAlgoritmo
```

Operación de suma

La suma es una de las operaciones aritméticas fundamentales y se utiliza ampliamente para acumulación de valores, cálculos financieros, contadores y algoritmos iterativos. Según Stallings [53], la suma constituye una instrucción muy eficiente a nivel de hardware. Desde la perspectiva de ingeniería de software, se considera un mecanismo de transformación de estado ampliamente utilizado [12].

La suma es útil para representar cantidades crecientes, totales acumulados y progresiones matemáticas. Sommerville [3] destaca su importancia en el modelado de procesos que implican crecimiento incremental.

Ejemplo. `total <- total + precio`

Ejercicio. Diseñe un algoritmo que sume los primeros `n` números naturales.

Listing 2.18: Algoritmo para sumar los primeros n números naturales

```
  Algoritmo SumarPrimerosN
  Entrada: n
  Salida: suma

  1. Escribir "Ingrese un número natural n:"
  2. Leer n

```

```
  3. suma <- 0
  4. contador <- 1

  5. Mientras contador <= n hacer
  6.     suma <- suma + contador
  7.     contador <- contador + 1
  8. FinMientras

  9. Escribir "La suma de los primeros ", n, " números naturales es: ", suma

  FinAlgoritmo
```

La suma de los primeros n números naturales puede realizarse de forma iterativa, tal como se presenta en el algoritmo anterior; sin embargo, este enfoque tiene un costo temporal proporcional a n, lo que implica ejecutar un ciclo completo para producir el resultado. Una optimización clásica consiste en emplear la fórmula cerrada $S = \frac{n(n+1)}{2}$, atribuida históricamente a Gauss, la cual permite calcular la sumatoria en tiempo constante $O(1)$ sin utilizar estructuras repetitivas. Tal como señalan Pressman y Maxim [12], el uso de expresiones matemáticas directas reduce la complejidad algorítmica y mejora la eficiencia computacional, especialmente cuando el valor de n es grande. De forma complementaria, Sommerville [3] destaca que la sustitución de bucles por expresiones determinísticas contribuye a la simplicidad y mantenibilidad del código, beneficiando tanto la verificación como las pruebas del algoritmo. Esta optimización representa un ejemplo claro de cómo el análisis matemático y la ingeniería de software convergen para producir soluciones más eficientes y elegantes.

Operación de resta

La resta permite modelar decrementos, diferencias y reducciones de cantidades, siendo esencial en algoritmos que gestionan inventarios, distancias o diferencias temporales. Silberschatz et al. [55] explican que esta operación se utiliza frecuentemente para controlar condiciones de agotamiento o para evaluar progresos inversos.

Desde la perspectiva educativa, la resta ayuda a comprender el flujo descendente de un algoritmo, preparando al programador en formación para ciclos decrecientes y estructuras de control más complejas.

Ejemplo. `stock <- stock - 1`

Ejercicio. Construya un algoritmo que reste 1 unidad a un inventario hasta llegar a cero.

Listing 2.19: Algoritmo para decrementar el inventario de un producto hasta llegar a cero

```
Algoritmo DecrementarInventario
Entrada: producto, inventario
Salida: inventario final

1. Escribir "Ingrese el nombre del producto:"
2. Leer producto

3. Escribir "Ingrese la cantidad inicial de inventario:"
4. Leer inventario

5. Mientras inventario > 0 hacer
6.     inventario <- inventario - 1
7.     Escribir "Inventario de ", producto, ": ", inventario
8. FinMientras

9. Escribir "El inventario de ", producto, " ha llegado a cero."

FinAlgoritmo
```

Operación de multiplicación

La multiplicación permite modelar escalamiento, crecimiento proporcional y cálculos más complejos que requieren multiplicar constantes o variables. Stallings [53] señala que esta operación, aunque más costosa que la suma, sigue siendo altamente optimizada en arquitecturas modernas.

Desde un punto de vista algorítmico, la multiplicación se utiliza para cálculos de áreas, volúmenes, conversiones de unidades y cualquier transformación que dependa de un factor constante.

Ejemplo: `area <- base * altura`

Ejercicio: Diseñe un algoritmo que calcule el área de un rectángulo.

Listing 2.20: Algoritmo para calcular el área de un rectángulo

```
Algoritmo CalcularAreaRectangulo
```

```
2  Entrada: base, altura
3  Salida: area
4
5  1. Escribir "Ingrese la base del rectángulo:"
6  2. Leer base
7
8  3. Escribir "Ingrese la altura del rectángulo:"
9  4. Leer altura
10
11 5. area <- base * altura
12
13 6. Escribir "El área del rectángulo es: ", area
14
15 FinAlgoritmo
```

Operación de división

La división permite representar particiones equitativas, tasas, promedios y proporciones. Silberschatz et al. [55] destacan que esta operación requiere validación adicional para evitar divisiones por cero, un error común entre principiantes. En ingeniería del software, su tratamiento adecuado forma parte de las buenas prácticas de gestión de errores [12].

La división también se emplea para distribuir recursos, calcular promedios y evaluar rendimientos o eficiencias.

Ejemplo. `promedio <- sumaTotal / cantidad`

Ejercicio 1: Escriba un algoritmo que calcule el promedio de tres notas.

Listing 2.21: Algoritmo para calcular el promedio de tres notas

```
1  Algoritmo PromedioTresNotas
2  Entrada: nota1, nota2, nota3
3  Salida: promedio
4
5  1. Escribir "Ingrese la primera nota:"
6  2. Leer nota1
7
8  3. Escribir "Ingrese la segunda nota:"
9  4. Leer nota2
10
```

```
11  5. Escribir "Ingrese la tercera nota:"
12  6. Leer nota3
13
14  7. promedio <- (nota1 + nota2 + nota3) / 3
15
16  8. Escribir "El promedio de las tres notas es: ", promedio
17
18  FinAlgoritmo
```

Ejercicio 2: Escriba un algoritmo que calcule el promedio de n notas, n debe ser ingresado mediante una operación de entrada.

Listing 2.22: Algoritmo para calcular el promedio de n notas

```
1   Algoritmo PromedioNNotas
2   Entrada: n, nota, suma, promedio
3   Salida: promedio
4
5   1. Escribir "Ingrese la cantidad de notas:"
6   2. Leer n
7
8   3. suma <- 0
9   4. contador <- 1
10
11  5. Mientras contador <= n hacer
12  6.      Escribir "Ingrese la nota ", contador, ":"
13  7.      Leer nota
14  8.      suma <- suma + nota
15  9.      contador <- contador + 1
16  10. FinMientras
17
18  11. promedio <- suma / n
19
20  12. Escribir "El promedio de las ", n, " notas es: ", promedio
21
22  FinAlgoritmo
```

Operación de módulo

El módulo (`mod`) devuelve el residuo de una división (ver listado de código 2.13), siendo fundamental para algoritmos relacionados con divisibilidad, ciclos periódicos y reconocimiento de patrones. Sommerville [3] destaca su importancia en validación de entradas, calendarios, control de repeticiones y detección de valores pares o impares.

Desde el análisis algorítmico, el módulo permite construir condiciones eficientes y expresar comportamientos cíclicos, como verificar cada cierto número de iteraciones.

Ejemplo: Si se requiere verificar que un número `n` es par, entonces: `si (n mod 2 = 0) entonces imprimir ``par''`

Ejercicio: Diseñe un algoritmo que determine si un número es múltiplo de 5.

Un número entero a se considera múltiplo de otro número entero b cuando existe un entero k tal que se cumple la igualdad $a = b \cdot k$. Esta relación implica que a contiene exactamente k veces a b, sin dejar residuo alguno al efectuar la división entre ellos. Desde la teoría de números, esta propiedad es fundamental para caracterizar divisibilidad, diseñar algoritmos aritméticos y establecer patrones cíclicos en procesos computacionales [57]. En el ámbito algorítmico y educativo, identificar múltiplos permite construir condiciones lógicas eficientes basadas en la operación módulo, ampliamente utilizada en algoritmos de clasificación, verificación y recorridos periódicos [58]. La comprensión de esta noción facilita el razonamiento sobre estructuras recurrentes y operaciones matemáticas comunes en programación básica y estructuras de control.

Listing 2.23: Algoritmo para determinar si un número es múltiplo de 5

```
1  Algoritmo EsMultiploDe5
2  Entrada: numero
3  Salida: mensaje
4
5  1. Escribir "Ingrese un número:"
6  2. Leer numero
7
8  3. Si numero mod 5 = 0 entonces
9  4.     mensaje <- "El número es múltiplo de 5"
10 5. SiNo
11 6.     mensaje <- "El número no es múltiplo de 5"
12 7. FinSi
13
```

```
14  8. Escribir mensaje
15
16  FinAlgoritmo
```

Operación lógica AND

La operación lógica AND produce verdadero solo cuando todas las condiciones evaluadas son verdaderas. Esta operación permite construir decisiones compuestas que integran múltiples restricciones simultáneas, tal como destaca Sommerville [3]. En sistemas de información, AND es ampliamente utilizada para validar formularios, políticas de acceso y reglas de negocio, donde deben cumplirse varias condiciones al mismo tiempo [12].

Desde el punto de vista educativo, AND ayuda a los principiantes a comprender cómo se combinan condiciones lógicas para tomar decisiones más precisas. Además, Tanenbaum y Austin [54] mencionan que esta operación tiene un correlato directo en la lógica de circuitos, reforzando su importancia fundamental.

Ejemplo. `si edad >= 18 AND documento = true entonces permitirAcceso`

Ejercicio. Diseñe un algoritmo que verifique si un estudiante aprueba cuando `nota >= 70 AND asistencia >= 80`. Una de sus soluciones se muestra en el listado de código 2.24

Listing 2.24: Algoritmo para verificar si un estudiante aprueba según nota y asistencia

```
1   Algoritmo VerificarAprobacion
2   Entrada: nota, asistencia
3   Salida: mensaje
4
5   1. Escribir "Ingrese la nota del estudiante:"
6   2. Leer nota
7
8   3. Escribir "Ingrese el porcentaje de asistencia:"
9   4. Leer asistencia
10
11  5. Si nota >= 70 AND asistencia >= 80 entonces
12  6.     mensaje <- "El estudiante aprueba"
13  7. SiNo
14  8.     mensaje <- "El estudiante no aprueba"
15  9. FinSi
16
```

```
17  10. Escribir mensaje
18
19  FinAlgoritmo
```

Operación lógica OR

La operación lógica OR devuelve verdadero cuando al menos una de las condiciones se cumple. Es esencial cuando el algoritmo debe habilitar caminos alternativos o comportamientos flexibles [3]. En sistemas de información se usa para validar opciones múltiples, permisos condicionales y flujos con tolerancia a incompletitud [12].

Para estudiantes, OR ofrece una comprensión más amplia de los mecanismos de decisión, mostrando cómo distintos criterios pueden habilitar una misma acción. En hardware, OR corresponde a un componente fundamental en puertas digitales, lo que explica su eficiencia y universalidad [53].

Ejemplo: `si rol = "admin" OR rol = "supervisor" entonces accesoEspecial`

Ejercicio: Construya un algoritmo que permita acceder a un sistema si el usuario es estudiante **OR** profesor. Una posible solución es la que se muestra en el listado de código 2.25

Listing 2.25: Algoritmo para permitir acceso si el usuario es estudiante OR profesor

```
1   Algoritmo VerificarAcceso
2   Entrada: rol
3   Salida: mensaje
4
5   1. Escribir "Ingrese el rol del usuario (estudiante/profesor):"
6   2. Leer rol
7
8   3. Si rol = "estudiante" OR rol = "profesor" entonces
9   4.     mensaje <- "Acceso permitido"
10  5. SiNo
11  6.     mensaje <- "Acceso denegado"
12  7. FinSi
13
14  8. Escribir mensaje
15
16  FinAlgoritmo
```

Operación lógica NOT

La operación lógica NOT invierte el valor lógico de una condición, transformando verdadero en falso y viceversa. Silberschatz et al. [55] señalan que esta operación es crucial para detectar excepciones, valores nulos o estados ausentes en sistemas de información. Desde la ingeniería del software, el uso adecuado de NOT permite crear condiciones de control que advierten sobre comportamientos anómalos [12].

Para principiantes, comprender NOT implica adquirir la capacidad de construir condiciones negativas o exclusiones explícitas, lo cual es indispensable para validar entradas y evitar estados no deseados.

Ejemplo. `si NOT (usuarioRegistrado) entonces solicitarRegistro`

Ejercicio. Existen aplicaciones que no pueden continuar si se dio algún error, por lo tanto verificar si existe un error previo es necesario para que la aplicación continúe con algún proceso, o simplemente continúe en funcionamiento. Diseñe un algoritmo que permita continuar solo si NO existe un error previo. Aunque en una aplicación real, no se daría el ingreso de la condición, una solución se muestra en el listado de código 2.26.

Listing 2.26: Algoritmo que permite continuar solo si NO existe un error previo

```
1  Algoritmo ContinuarSiNoError
2  Entrada: errorPrevio   // valor booleano: true o false
3  Salida: mensaje
4
5  1. Escribir "¿Existe un error previo? (true/false):"
6  2. Leer errorPrevio
7
8  3. Si NOT errorPrevio entonces
9  4.     mensaje <- "Puede continuar el proceso"
10 5. SiNo
11 6.     mensaje <- "No puede continuar: existe un error previo"
12 7. FinSi
13
14 8. Escribir mensaje
15
16 FinAlgoritmo
```

Lectura de memoria

La lectura consiste en recuperar el valor almacenado en una variable o posición de memoria. Esta operación se considera fundamental en todos los algoritmos, ya que permite acceder al estado actual del sistema para realizar transformaciones posteriores [55]. Sommerville [3] enfatiza que comprender la lectura es esencial para razonar sobre estados, dependencias y comportamiento algorítmico.

Durante la ejecución de un algoritmo, las lecturas permiten obtener valores intermedios que condicionan decisiones o cálculos. Stallings [53] explica que, aunque la lectura puede parecer transparente, su eficiencia depende de la jerarquía de memoria del sistema. Para estudiantes, visualizar la lectura como un paso explícito mejora su capacidad para depurar programas.

La lectura también se relaciona con problemas comunes como el uso de variables no inicializadas, lo cual es una fuente frecuente de errores en programación inicial [12].

Ejercicio. Diseñe un algoritmo que muestre en pantalla el valor de varias variables previamente almacenadas.

Listing 2.27: Algoritmo para mostrar el valor de variables previamente almacenadas

```
1  Algoritmo MostrarVariables
2  Entrada: a, b, c    // Variables ya almacenadas en memoria
3  Salida: mensaje en pantalla
4
5  1. Escribir "Los valores almacenados son:"
6  2. Escribir "Variable a: ", a
7  3. Escribir "Variable b: ", b
8  4. Escribir "Variable c: ", c
9
10 FinAlgoritmo
```

Este algoritmo ilustra la operación de *lectura de memoria:* el programa accede a los valores actuales de las variables sin modificarlos y los muestra en pantalla. Es una operación fundamental en programación porque permite al programador en formación visualizar el estado interno del algoritmo, reforzando el modelo mental de cómo se almacenan y recuperan datos.

Escritura en memoria

La escritura consiste en almacenar un valor en una variable, modificando el estado del algoritmo. Silberschatz et al. [55] indican que la escritura es una operación crítica porque implica persistencia temporal del valor, lo cual condiciona la ejecución futura. Desde la ingeniería de software, Pressman y Maxim [12] subrayan que escribir correctamente contribuye a la claridad estructural del algoritmo.

Stallings [53] menciona que la escritura puede requerir más recursos computacionales que la lectura, especialmente en niveles bajos de memoria. Para estudiantes principiantes, comprender la escritura como una acción explícita ayuda a visualizar cómo se transforma el estado del programa.

La escritura también está asociada a la consistencia interna del algoritmo, pues valores incorrectos pueden producir fallas lógicas o comportamientos no deseados [3].

Ejercicio. Escriba un algoritmo que almacene en variables los datos ingresados en un formulario (nombre, edad, correo).

Listing 2.28: Algoritmo para almacenar en variables los datos ingresados en un formulario

```
Algoritmo AlmacenarDatosFormulario
Entrada: nombre, edad, correo
Salida: mensaje

1. Escribir "Ingrese su nombre:"
2. Leer nombre

3. Escribir "Ingrese su edad:"
4. Leer edad

5. Escribir "Ingrese su correo electrónico:"
6. Leer correo

7. Escribir "Los datos almacenados son:"
8. Escribir "Nombre: ", nombre
9. Escribir "Edad: ", edad
10. Escribir "Correo: ", correo

FinAlgoritmo
```

Actualización de memoria

La actualización combina lectura, operación y escritura, constituyendo una secuencia completa de transformación del estado interno del algoritmo [53]. Este patrón es esencial en ciclos, acumuladores, contadores y estructuras iterativas. Desde la teoría de sistemas, la actualización representa un mecanismo evolutivo del estado, permitiendo que el algoritmo progrese [3].

Silberschatz et al. [55] destacan que las actualizaciones deben diseñarse con claridad para evitar sobreescrituras involuntarias o pérdida de información. En enseñanza inicial de programación, esta operación ayuda a desarrollar un modelo mental adecuado del estado interno del programa.

El manejo adecuado de actualizaciones mejora la capacidad de depuración y evita errores como la modificación incorrecta de contadores o índices.

Ejercicio. Construya un algoritmo que incremente un contador hasta un valor dado y muestre los resultados.

Listing 2.29: Algoritmo que incrementa un contador hasta un valor dado y muestra los resultados

```
Algoritmo IncrementarContador
Entrada: limite, contador
Salida: valores del contador

1. Escribir "Ingrese el valor límite:"
2. Leer limite

3. contador <- 1

4. Mientras contador <= limite hacer
5.     Escribir "Contador: ", contador
6.     contador <- contador + 1
7. FinMientras

8. Escribir "El contador ha alcanzado el valor límite."

FinAlgoritmo
```

Direcciones conceptuales

Las direcciones conceptuales representan una abstracción de cómo los sistemas organizan y localizan datos en memoria. Stallings [53] explica que, aunque los lenguajes de alto nivel abstraen estos detalles, su comprensión mejora la capacidad de interpretar modelos computacionales. Sommerville [3] menciona que esta abstracción contribuye a comprender cómo los datos se relacionan entre sí.

Desde la perspectiva educativa, enseñar direcciones conceptuales prepara al programador en formación para temas avanzados como estructuras de datos, punteros y paso por referencia.

Ejercicio. Modele mediante un diagrama conceptual cómo se almacenan tres variables consecutivas en memoria.

Modelo conceptual de memoria

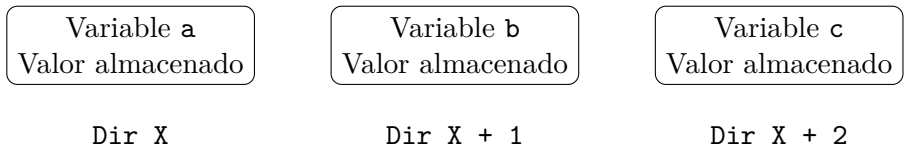

Figura 2.1: Representación conceptual de tres variables consecutivas en memoria.

Variables vs. constantes

Las variables representan espacios de memoria cuyo contenido puede cambiar, mientras que las constantes almacenan valores fijos. Pressman [12] señala que distinguir ambas es clave para garantizar claridad semántica en los algoritmos. Silberschatz et al. [55] indican que el uso adecuado de constantes evita errores relacionados con modificaciones no deseadas.

Desde un enfoque pedagógico, esta distinción ayuda a estructurar algoritmos más robustos y coherentes.

Ejercicio. Diseñe un algoritmo que utilice una constante para almacenar la tasa de interés y una variable para calcular montos futuros. El algoritmo que se muestra en el listado de código 2.30 representan los pasos que se deben implementar en un programa de computadora para obtener los cálculos que soluciona el requisito.

Listing 2.30: Algoritmo que utiliza una constante de interés para calcular montos futuros

```
1  Algoritmo CalcularMontoFuturo
2  Constante TASA_INTERES <- 0.05   // 5% de interés anual
```

```
3  Entrada: capital, montoFuturo
4  Salida: montoFuturo
5
6  1. Escribir "Ingrese el capital inicial:"
7  2. Leer capital
8
9  3. montoFuturo <- capital + (capital * TASA_INTERES)
10
11 4. Escribir "La tasa de interés aplicada es: ", TASA_INTERES
12 5. Escribir "El monto futuro es: ", montoFuturo
13
14 FinAlgoritmo
```

Representación conceptual del estado

La representación conceptual del estado consiste en visualizar las variables del algoritmo y sus valores en un instante específico. Sommerville [3] destaca que esta representación es esencial para depurar y verificar la lógica del algoritmo. Spivak y Kent [49] indican que un modelo claro del estado permite razonar formalmente sobre la transición entre estados.

Desde una perspectiva formativa, esta representación facilita al programador en formación comprender cómo las operaciones afectan el sistema.

Ejercicio: Dibuje una tabla que represente el estado de un algoritmo antes y después de ejecutar una operación de actualización.

Cuadro 2.1: Estado de una variable antes y después de una operación de actualización

Variable	Estado Antes	Actualización	Estado Después
contador	1	contador <- contador + 1	2
acumulador	1	acumulador <- acumulador + contador	3

La Figura 2.2 muestra una representación conceptual de cómo dos variables enteras, `contador` y `acumulador`, se almacenan en memoria principal. Cada entero ocupa cuatro bytes consecutivos; en este ejemplo, `contador` reside desde la dirección hexadecimal 0x1200 hasta 0x1203, mientras que `acumulador` se ubica desde 0x1204 hasta 0x1207. La columna de la derecha ilustra de forma didáctica el bloque de cuatro bytes reservado para cada variable, representado por un rectángulo con el valor lógico almacenado en su interior. Este modelo ayuda a los estudiantes a visualizar que una variable de alto nivel no corresponde a

una única posición física, sino a un conjunto de direcciones contiguas, y constituye la base para comprender cómo las operaciones de actualización actúan sobre el estado almacenado en memoria.

Variable	Dirección	Espacio de memoria (4 bytes)
`contador`	0x1200	1
	0x1201	
	0x1202	
	0x1203	
`acumulador`	0x1204	1
	0x1205	
	0x1206	
	0x1207	

Figura 2.2: Representación conceptual del espacio de memoria ocupado por dos variables enteras consecutivas.

La Figura 2.3 complementa la representación de memoria presentada anteriormente al mostrar explícitamente cómo se modifica el estado lógico de las variables durante un proceso de actualización. En este ejemplo, `contador` incrementa su valor de 1 a 2 mediante la operación `contador <- contador + 1`, mientras que `acumulador` pasa de 1 a 3 a través de la instrucción `acumulador <- acumulador + 2`. Estas transformaciones ilustran que el estado de una variable constituye un elemento dinámico dentro del algoritmo: cada actualización produce un nuevo valor que sustituye al anterior, generando una transición de estado que será utilizada en operaciones posteriores. El diagrama enfatiza visualmente cómo las variables cambian y cómo dichas modificaciones responden a instrucciones específicas dentro del flujo del algoritmo.

2.2. Estructura lógica de un algoritmo

La estructura lógica de un algoritmo constituye el fundamento esencial para comprender cómo se organiza la ejecución de un proceso computacional. Un algoritmo es, en términos formales, una secuencia finita y ordenada de instrucciones que transforman un estado inicial en un estado final mediante operaciones definidas y no ambiguas. La claridad en la organización de estas instrucciones permite que el algoritmo sea comprensible, verificable y

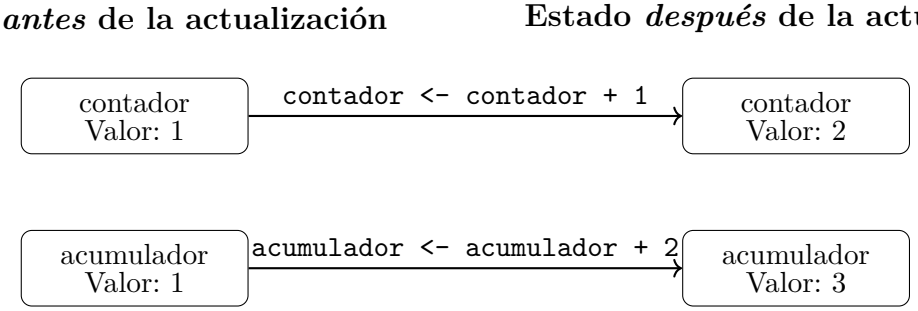

Figura 2.3: Actualización explícita del estado de variables: el contador pasa de 1 a 2, mientras que el acumulador pasa de 1 a 3.

ejecutable tanto por humanos como por máquinas [7], [58]. Desde la teoría de la computación, esta estructura representa un modelo abstracto que describe el comportamiento del sistema en función de sus transiciones de estado.

Diversos autores coinciden en que la lógica algorítmica se compone de tres patrones fundamentales: la secuencia, la selección y la iteración [7], [57]. Estos patrones describen las formas esenciales mediante las cuales se controla el flujo de ejecución y permiten expresar la mayoría de los procesos computacionales. El uso adecuado de estas estructuras contribuye a la creación de algoritmos correctos, eficientes y fáciles de mantener, constituyendo una base indispensable para desarrollos más complejos como el diseño modular, la programación orientada a objetos y la construcción de sistemas de información.

La estructuración lógica también permite definir las condiciones en las que un algoritmo interactúa con su entorno. Este intercambio se expresa mediante el modelo Entrada–Proceso–Salida (EPS), ampliamente aceptado en la literatura de ingeniería del software como un mecanismo formal para describir cómo un sistema recibe datos, los transforma y devuelve resultados [3], [12]. Dicho modelo facilita la comprensión del flujo de información y establece un marco conceptual que sirve de guía para el análisis, diseño y posterior implementación en un lenguaje de programación.

Finalmente, comprender la estructura lógica de un algoritmo fortalece la capacidad de abstracción del programador en formación y desarrolla habilidades para descomponer problemas en pasos claros y manejables. Este proceso de razonamiento algorítmico resulta esencial en la formación en ingeniería, pues permite construir soluciones reproducibles, formalmente correctas y adaptables a distintos dominios [59]. El dominio de estas estructuras no solo facilita el aprendizaje de la programación, sino que constituye un paso fundamental para el desarrollo de competencias relacionadas con la resolución sistemática de problemas.

2.2.1. Secuencia

La estructura secuencial constituye la forma más elemental de control del flujo en un algoritmo y se caracteriza por la ejecución ordenada de instrucciones, una tras otra, sin bifurcaciones ni repeticiones. Este modelo refleja el principio fundamental de que todo algoritmo debe especificar una serie de pasos inequívocos que transformen un estado de entrada en un estado de salida, siguiendo una lógica lineal y predecible. La literatura clásica en ciencias de la computación enfatiza que la secuencialidad es la base sobre la cual se construyen las estructuras más complejas, al proveer un mecanismo de ordenamiento rígido y determinista [60].

Autores como Knuth destacan que la secuencia es indispensable para garantizar la trazabilidad y la corrección de un algoritmo, ya que permite que cada instrucción dependa del resultado de la anterior, construyendo así un flujo de información coherente y verificable [61]. La claridad en la secuencia evita ambigüedades y facilita el análisis formal del algoritmo, así como su implementación en lenguajes de programación, donde la ejecución secuencial constituye el comportamiento por defecto de la mayoría de las instrucciones.

En el ámbito pedagógico, la secuencia es el primer patrón de control que los estudiantes deben dominar, pues proporciona un modelo intuitivo para comprender cómo se ejecutan los programas y cómo fluye la información a través de las variables. De acuerdo con Joyanes Aguilar, la estructura secuencial permite introducir conceptos esenciales como operaciones de asignación, entrada y salida, preparando al programador en formación para estructuras de mayor complejidad [7]. Su simplicidad la convierte en un recurso didáctico primordial para explicar la lógica algorítmica de forma gradual y progresiva.

Ejemplo 1: Para preparar una taza de té se siguen pasos estrictamente secuenciales: (1) hervir agua, (2) colocar la bolsa de té en la taza, (3) verter el agua caliente, y (4) esperar a que infusione. Cada paso debe completarse antes de continuar con el siguiente.

Ejemplo 2: Para calcular el área de un triángulo: (1) leer la base, (2) leer la altura, (3) multiplicar base por altura, y (4) dividir el resultado entre dos.

Ejercicio 1: Diseñe un algoritmo secuencial para cepillarse los dientes. Este ejercicio refuerza cómo tareas cotidianas pueden modelarse mediante secuencias estrictas sin decisiones ni ciclos.

Listing 2.31: Algoritmo secuencial para cepillarse los dientes

```
1 Algoritmo CepilladoDientes
2 1. Tomar el cepillo.
3 2. Aplicar pasta dental al cepillo.
```

```
4  3. Cepillar los dientes durante 2 minutos.
5  4. Enjuagar la boca.
6  5. Lavar el cepillo.
7  FinAlgoritmo
```

La secuencia de pasos de un algoritmo se ejecuta exactamente en el orden especificado (ver listado de código 2.31); cambiar el orden comprometería la efectividad o coherencia de la tarea.

Ejercicio 2: Elabore un algoritmo secuencial para enviar un correo electrónico. Este ejercicio (ver listado de código 2.32) muestra cómo un proceso cotidiano digital también sigue una estructura secuencial.

Listing 2.32: Algoritmo secuencial para enviar un correo electrónico

```
1  Algoritmo EnviarCorreo
2  1. Abrir el programa de correo.
3  2. Crear un nuevo mensaje.
4  3. Escribir destinatario, asunto y contenido.
5  4. Adjuntar archivos si es necesario.
6  5. Presionar enviar.
7  FinAlgoritmo
```

Cada paso depende del anterior; no se puede enviar un mensaje sin haberlo escrito, ni escribir sin haber creado el mensaje antes.

Ejercicio 3: Diseñe un algoritmo secuencial que calcule el doble y el triple de un número ingresado. La solución (ver el algoritmo listado de código 2.33) introduce operaciones matemáticas elementales siguiendo el flujo secuencial.

Listing 2.33: Cálculo del doble y triple de un número

```
1  Algoritmo DobleTriple
2  1. Escribir "Ingrese un número:"
3  2. Leer n
4  3. doble <- 2 * n
5  4. triple <- 3 * n
6  5. Escribir doble, triple
7  FinAlgoritmo
```

El valor de doble y triple depende directamente del valor leído en el paso 2.

Ejercicio 4: Construya un algoritmo que convierta metros a centímetros y milímetros. Su solución (ver listado de código 2.34) requiere de una secuencia estricta de pasos: primero se lee el valor, luego se realizan las conversiones y finalmente se muestran los resultados.

Listing 2.34: Conversión de metros a cm y mm

```
Algoritmo ConversionLongitud
1. Leer metros
2. centimetros <- metros * 100
3. milimetros <- metros * 1000
4. Escribir centimetros, milimetros
FinAlgoritmo
```

Ejercicio 5: Diseñe un algoritmo secuencial que solicite un nombre y una edad, y los imprima en pantalla. La estructura secuencial que requiere el algoritmo solución (ver listado de código 2.35)permite recibir primero los datos y luego mostrarlos; invertir el orden haría que el algoritmo falle conceptualmente.

Listing 2.35: Impresión secuencial de datos

```
Algoritmo DatosUsuario
1. Leer nombre
2. Leer edad
3. Escribir "Nombre:", nombre
4. Escribir "Edad:", edad
FinAlgoritmo
```

2.2.2. Selección

La estructura de selección permite que un algoritmo tome decisiones en función de condiciones lógicas que determinan qué camino de ejecución seguir. A diferencia de la secuencia, que ejecuta instrucciones de manera lineal, la selección introduce bifurcaciones controladas, haciendo posible que el algoritmo responda de forma diferente ante distintos datos de entrada o situaciones del problema. La literatura clásica de algoritmia establece que la selección es un mecanismo fundamental en la resolución de problemas debido a su capacidad para modelar comportamientos condicionales esenciales en los procesos computacionales [58].

Existen dos formas principales de selección: la selección simple y la selección doble, complementadas por estructuras más complejas como la selección múltiple. La selección

simple ejecuta un bloque de instrucciones únicamente si la condición especificada se cumple; la selección doble agrega un bloque alternativo que se ejecuta cuando la condición no se satisface. En cambio, la selección múltiple permite evaluar múltiples alternativas, ofreciendo mayor flexibilidad para modelar situaciones con más de dos resultados posibles. Este conjunto de estructuras constituye la base del control condicional en la mayoría de los lenguajes de programación modernos [62].

Desde el punto de vista conceptual, la selección exige que las expresiones utilizadas en la condición sean precisas y evaluables, evitando ambigüedades que puedan conducir a resultados incorrectos. La teoría de los predicados, ampliamente utilizada en matemáticas y ciencias de la computación, aporta el marco formal para definir las condiciones lógicas que permiten decidir entre diferentes caminos de ejecución [57]. Los predicados permiten representar afirmaciones que pueden ser verdaderas o falsas, lo cual constituye el fundamento lógico de las estructuras condicionales.

La selección también se relaciona con el diseño de algoritmos eficientes. Al utilizar condiciones adecuadas, es posible reducir la cantidad de operaciones necesarias para llegar a una solución, lo que incrementa la eficiencia temporal y espacial del algoritmo. Por ejemplo, una selección bien construida puede evitar cálculos innecesarios o podar caminos de ejecución que no contribuyen al resultado final. Knuth señala que el control condicional, cuando se emplea de manera correcta, contribuye directamente al diseño de algoritmos óptimos [61].

Por lo tanto, la estructura de selección posee un alto valor pedagógico porque facilita que el programador en formación comprenda cómo los algoritmos "toman decisiones". Tal como explican Joyanes Aguilar y otros autores, el uso didáctico de estructuras condicionales permite vincular problemas cotidianos con soluciones computacionales formales, favoreciendo la adquisición progresiva del pensamiento algorítmico [7], [58], [63]. La capacidad de modelar decisiones en un algoritmo constituye uno de los pilares fundamentales de la programación imperativa.

Ejemplo 1: Un caso cotidiano de selección es determinar si una persona es mayor de edad. La decisión depende de una condición: si la edad es mayor o igual a 18 años. Esto se modela mediante la estructura `if-else` en el listado de código 2.36:

Listing 2.36: Determinación de mayoría de edad

```
1 Algoritmo DeterminarMayoriaEdad
2 1. Leer edad
```

```
2. Si edad >= 18 entonces
3.     Escribir "Es mayor de edad"
4. SiNo
5.     Escribir "Es menor de edad"
6. FinSi
FinAlgoritmo
```

Este algoritmo refleja cómo una única condición puede modificar el flujo del proceso para producir una salida coherente con los datos ingresados.

Ejemplo 2: Otro ejemplo común de selección es verificar si un número es par utilizando la operación módulo. Si el residuo de dividir entre 2 es cero, el número es par. El algoritmo se muestra en el listado de código 2.37.

Listing 2.37: Verificación de número par

```
Algoritmo VerificarPar
1. Leer n
2. Si n mod 2 = 0 entonces
3.     Escribir "El número es par"
4. SiNo
5.     Escribir "El número es impar"
6. FinSi
FinAlgoritmo
```

Este ejemplo muestra el uso de condiciones basadas en operaciones aritméticas, fundamentales para muchos algoritmos numéricos.

Ejercicio 1: Diseñe un algoritmo que determine si una persona puede ingresar a una función de cine según su edad mínima permitida (>= 15 años). Una de las soluciones en la que se muestra en el listado de código 2.38.

Listing 2.38: Algoritmo para control de acceso al cine

```
Algoritmo ControlCine
1. Leer edad
2. Si edad >= 15 entonces
3.     Escribir "Acceso permitido"
4. SiNo
5.     Escribir "Acceso denegado"
```

```
7  6. FinSi
8  FinAlgoritmo
```

En el algoritmo (ver listado de código 2.38) se evalúa una sola condición y se bifurca el proceso en dos respuestas posibles. Es un caso claro de selección simple.

Ejercicio 2: Diseñe un algoritmo que determine si un alimento puede consumirse según su fecha de caducidad.

Listing 2.39: Verificación de caducidad de alimento

```
1  Algoritmo VerificarCaducidad
2  1. Leer fechaActual
3  2. Leer fechaCaducidad
4  3. Si fechaActual <= fechaCaducidad entonces
5  4.     Escribir "El alimento está en buen estado"
6  5. SiNo
7  6.     Escribir "El alimento está caducado"
8  7. FinSi
9  FinAlgoritmo
```

El algoritmo que muestra el listado de código 2.39 refleja la comparación entre dos valores temporales, un caso típico de selección compuesta.

Ejercicio 3: Diseñe un algoritmo que determine el mayor de tres números usando selección múltiple.

Listing 2.40: Mayor de tres números

```
1  Algoritmo MayorDeTres
2  1. Leer a, b, c
3  2. Si (a >= b) Y (a >= c) entonces
4  3.     Escribir a
5  4. SiNo
6  5.     Si (b >= a) Y (b >= c) entonces
7  6.         Escribir b
8  7.     SiNo
9  8.         Escribir c
10 9.     FinSi
11 10. FinSi
```

```
12  FinAlgoritmo
```

Se evalúan condiciones anidadas para identificar el mayor valor entre tres números (ver listado de código 2.40).

Ejercicio 5: Diseñe un algoritmo que valide un usuario y contraseña simples.

Listing 2.41: Algoritmo de validación de usuario

```
1  Algoritmo ValidarLogin
2  1. Leer usuario
3  2. Leer clave
4  3. Si (usuario = "admin") Y (clave = "1234") entonces
5  4.     Escribir "Acceso permitido"
6  5. SiNo
7  6.     Escribir "Acceso denegado"
8  7. FinSi
9  FinAlgoritmo
```

El algoritmo utiliza una condición compuesta para verificar dos criterios simultáneos: usuario y contraseña correctos.

Ejercicio (venta con descuento). Enunciado: Diseñe un algoritmo que calcule el total a pagar por la compra de un artículo. Si la cantidad adquirida es mayor a 3 unidades, se debe aplicar un descuento del 25 % sobre el subtotal.

Listing 2.42: Algoritmo de venta con descuento por cantidad

```
1  Algoritmo VentaConDescuento
2  Entrada: precioUnitario, cantidad
3  Salida: totalPagar
4
5  1. Escribir "Ingrese el precio unitario del artículo:"
6  2. Leer precioUnitario
7
8  3. Escribir "Ingrese la cantidad solicitada:"
9  4. Leer cantidad
10
11 5. subtotal <- precioUnitario * cantidad
12
```

```
13  6. Si cantidad > 3 entonces
14  7.     descuento <- subtotal * 0.25
15  8.     totalPagar <- subtotal - descuento
16  9. SiNo
17  10.    totalPagar <- subtotal
18  11. FinSi
19
20  12. Escribir "Subtotal: ", subtotal
21  13. Escribir "Total a pagar: ", totalPagar
22
23  FinAlgoritmo
```

El algoritmo del listado de código 2.42 inicia solicitando el precio unitario del artículo y la cantidad deseada. A partir de estos valores se calcula el `subtotal`. Luego se emplea una estructura de selección para determinar si la cantidad adquirida supera las tres unidades. Si la condición es verdadera, se aplica un descuento del 25 % y se obtiene el valor final a pagar. En caso contrario, el total corresponde simplemente al subtotal. Este tipo de decisión es común en sistemas comerciales que implementan promociones basadas en volumen de compra.

2.2.3. Iteración

La iteración constituye uno de los mecanismos esenciales de control en el diseño algorítmico, ya que permite repetir un conjunto de instrucciones mientras se cumpla una condición específica o durante un número determinado de veces. Esta estructura es fundamental para resolver problemas que requieren procesamiento repetitivo, tales como recorridos de datos, acumulaciones, cálculos secuenciales y simulaciones. Según Cormen et al. [58], la iteración es un elemento central en la computación porque reduce la complejidad del diseño, evita la duplicación de instrucciones y proporciona soluciones escalables para problemas de cualquier tamaño.

Knuth [61] señala que las estructuras iterativas como `while`, `for` y `repeat-until` representan patrones recurrentes en el análisis de algoritmos y desempeñan un papel crucial en la eficiencia computacional. Asimismo, el uso adecuado de estas estructuras permite controlar la complejidad temporal y espacial del algoritmo, especialmente en tareas que involucran grandes colecciones de datos. La iteración no sólo automatiza procesos repetitivos, sino que también facilita la sistematización de cálculos que serían inviables de

realizar de manera manual.

Desde la perspectiva pedagógica, la iteración contribuye al desarrollo del pensamiento computacional al introducir el concepto de *estado* cambiante durante la ejecución del algoritmo. Wing [64] enfatiza que las estructuras repetitivas ayudan a los estudiantes a comprender cómo los sistemas evolucionan paso a paso mediante actualizaciones sucesivas de variables. De forma complementaria, Joyanes Aguilar [7] destaca que la comprensión de contadores, acumuladores y condiciones de parada sienta las bases para la construcción de programas correctos y robustos.

La iteración también es indispensable para el tratamiento de datos en estructuras como listas, arreglos, matrices o flujos de entrada. Sedgewick y Wayne [65] explican que gran parte de los algoritmos clásicos, incluyendo búsquedas, ordenamientos y recorridos estructurados, se fundamentan en patrones iterativos bien definidos. Por ello, dominar la iteración permite abordar problemas más avanzados dentro de la ingeniería del software, la ciencia de datos y la inteligencia artificial.

En conclusión, el correcto uso de la iteración exige considerar aspectos como la inicialización adecuada, la actualización coherente del estado y la condición de salida que garantiza la finalización del proceso. Como señalan Dasgupta, Papadimitriou y Vazirani [66], una condición incorrecta puede conducir a bucles infinitos o comportamientos no deseados. La iteración, cuando se diseña apropiadamente, no solo resuelve problemas de manera eficiente, sino que permite construir software confiable y verificable.

Ejemplo 1: Un ejemplo clásico de iteración consiste en sumar los primeros n números naturales:

Listing 2.43: Suma iterativa de los primeros n números

```
Algoritmo SumarPrimerosN
1. Leer n
2. suma <- 0
3. contador <- 1
4. Mientras contador <= n hacer
5.     suma <- suma + contador
6.     contador <- contador + 1
7. FinMientras
8. Escribir suma
FinAlgoritmo
```

Este ejemplo (ver listado de código 2.43) muestra cómo un contador se inicializa, se actualiza y determina la condición de parada.

Ejemplo 2: Otra aplicación común es calcular a^b utilizando multiplicación iterativa:

Listing 2.44: Potencia mediante iteración

```
Algoritmo CalcularPotencia
1. Leer a
2. Leer b
3. resultado <- 1
4. contador <- 1
5. Mientras contador <= b hacer
6.     resultado <- resultado * a
7.     contador <- contador + 1
8. FinMientras
9. Escribir resultado
FinAlgoritmo
```

Se observa en el listado de código 2.44 cómo el valor intermedio evoluciona en cada iteración, representando el efecto acumulado.

Ejercicio 1: Diseñe un algoritmo que simule contar los pasos que una persona da para llegar a un destino, incrementando un contador hasta alcanzar el total deseado.

Listing 2.45: Conteo de pasos

```
Algoritmo ContarPasos
1. Leer totalPasos
2. contador <- 1
3. Mientras contador <= totalPasos hacer
4.     Escribir "Paso ", contador
5.     contador <- contador + 1
6. FinMientras
FinAlgoritmo
```

La iteración permite simular un proceso repetitivo donde cada paso depende del anterior como se muestra en los pasos del 3 al 6 del listado de código 2.45.

Ejercicio 2: Diseñe un algoritmo que simule llenar un recipiente utilizando vasos de agua, incrementando el nivel hasta alcanzar la capacidad máxima.

Listing 2.46: Llenado de un recipiente en iteración

```
Algoritmo LlenarRecipiente
1. Leer capacidad
2. nivel <- 0
3. Mientras nivel < capacidad hacer
4.     nivel <- nivel + 1
5.     Escribir "Nivel actual: ", nivel
6. FinMientras
FinAlgoritmo
```

El algoritmo del listado de código 2.46 ilustra una condición estricta de parada basada en un límite físico (capacidad del recipiente).

Ejercicio 3: Diseñe un algoritmo que cuente cuántos números entre 1 y `n` son pares. El procedimiento descrito en el Algoritmo 2.47 permite recorrer secuencialmente cada valor dentro del rango y verificar su paridad mediante la operación módulo.

Listing 2.47: Conteo de números pares

```
Algoritmo ContarPares
1. Leer n
2. contador <- 1
3. totalPares <- 0
4. Mientras contador <= n hacer
5.     Si contador mod 2 = 0 entonces
6.         totalPares <- totalPares + 1
7.     FinSi
8.     contador <- contador + 1
9. FinMientras
10. Escribir totalPares
FinAlgoritmo
```

Explicación: El Algoritmo 2.47 utiliza un contador que recorre uno a uno los números desde 1 hasta `n`. En cada paso se evalúa si el número actual es par usando la expresión `contador mod 2 = 0`. Si la condición se cumple, se incrementa la variable `totalPares`.

Este enfoque permite comprender el funcionamiento básico de la iteración, ya que el programador en formación observa cómo el valor de las variables cambia en cada ciclo.

Desde una perspectiva introductoria, una optimización sencilla consiste en notar que no es necesario analizar todos los números uno por uno. Dado que los números pares aparecen cada dos unidades, podría iniciarse el contador en 2 e incrementar de 2 en 2, reduciendo a la mitad el número de iteraciones. Esta optimización mantiene la claridad del algoritmo y conserva la lógica original, pero permite al programador en formación apreciar que existen distintas formas de resolver un mismo problema con un uso más eficiente de operaciones repetitivas.

Además de recorrer los números de manera iterativa, como se muestra en el Algoritmo 2.47, y como se menciona en el párrafo anterior, es posible optimizar completamente el cálculo utilizando una expresión matemática directa. Para determinar cuántos números pares existen entre 1 y `n`, basta observar que los números pares se obtienen multiplicando 2 por cada número natural: $2, 4, 6, \ldots, 2k$.

El último número par menor o igual que `n` es $2k = n$ (si `n` es par) o $2k = n - 1$ (si `n` es impar). En ambos casos, el valor de k se obtiene aplicando la división entera:

$$k = n \operatorname{div} 2$$

Esto significa que el total de números pares entre 1 y `n` es exactamente la parte entera de $n/2$. Por ejemplo, si $n = 10$, entonces $n \operatorname{div} 2 = 5$, por lo que hay cinco números pares; si $n = 11$, entonces $11/2 = 5.5$, cuya parte entera es 5, que corresponde a los pares $2, 4, 6, 8, 10$.

Este método elimina por completo la necesidad de iterar, ya que la respuesta puede calcularse con una sola operación. Para estudiantes principiantes, este resultado muestra que, aunque la iteración es útil para comprender la lógica paso a paso, muchos problemas pueden resolverse de manera más eficiente usando propiedades matemáticas básicas.

Ejercicio 4: Diseñe un algoritmo que incremente un ahorro inicial sumando un valor fijo en cada iteración durante un número de periodos. En el listado de código del Algoritmo 2.48 se presentan los pasos necesarios para incrementar el valor de `ahorro` mediante sumas sucesivas del valor `incremento` a lo largo de los `periodos` especificados por el usuario. Este enfoque permite modelar de manera sencilla un crecimiento acumulado, donde en cada iteración el ahorro se actualiza con el incremento definido, mostrando cómo una variable evoluciona progresivamente en un ciclo controlado por una condición de parada.

Listing 2.48: Ahorro acumulado en iteración

```
Algoritmo AhorroAcumulado
1. Leer ahorroInicial
2. Leer incremento
3. Leer periodos
4. ahorro <- ahorroInicial
5. contador <- 1
6. Mientras contador <= periodos hacer
7.     ahorro <- ahorro + incremento
8.     contador <- contador + 1
9. FinMientras
10. Escribir ahorro
FinAlgoritmo
```

El valor del ahorro evoluciona en cada iteración, reflejando un crecimiento lineal.

Ejercicio 5: Diseñe un algoritmo que cuente desde un número n hasta 0.

Listing 2.49: Temporizador regresivo

```
Algoritmo Temporizador
1. Leer n
2. Mientras n >= 0 hacer
3.     Escribir n
4.     n <- n - 1
5. FinMientras
FinAlgoritmo
```

La iteración decreciente es útil para simulaciones temporales o control de eventos.

2.2.4. Estructura Entrada–Proceso–Salida

La estructura Entrada–Proceso–Salida (EPS) constituye uno de los modelos conceptuales más utilizados para describir el funcionamiento de un algoritmo o sistema computacional. Este modelo establece que cualquier transformación de información puede entenderse mediante tres componentes fundamentales: los datos que ingresan al sistema (entrada), las operaciones que se aplican sobre ellos (proceso) y los resultados generados (salida). Pressman y Maxim [12] señalan que esta organización es esencial no solo para comprender el

comportamiento de un algoritmo, sino también para estructurar correctamente los sistemas de información en etapas tempranas del análisis.

Desde la perspectiva pedagógica, el modelo EPS permite a los estudiantes distinguir claramente el rol de cada parte del algoritmo, lo cual es especialmente importante cuando se introducen los conceptos de variables, instrucciones y operaciones de transformación. Tal como plantea Sommerville [3], la claridad en la separación entre datos y procesamiento contribuye a reducir errores conceptuales y facilita el proceso de diseño. La estructura EPS actúa como un puente entre el razonamiento cotidiano y el pensamiento computacional, permitiendo que el programador en formación identifique qué información es necesaria y cómo esta debe manipularse para resolver un problema.

Además, el modelo EPS es ampliamente utilizado en metodologías de diseño algorítmico porque permite representar problemas complejos como composiciones de tareas más simples. Cormen et al. [58] explican que en la mayoría de los algoritmos, independientemente de su complejidad, puede reconocerse esta estructura básica, ya que todo procedimiento computacional implica recibir información, procesarla mediante reglas definidas y generar un resultado. Esta universalidad convierte al modelo EPS en una herramienta de análisis compatible con algoritmos secuenciales, iterativos, recursivos y modulares.

En resumen, la estructura EPS facilita la documentación y validación del diseño algorítmico, pues permite representar de forma gráfica o textual cómo fluye la información a través del sistema. Como indican Dennis, Wixom y Tegarden [67], los modelos de entrada, proceso y salida se utilizan ampliamente en ingeniería de software y análisis de requerimientos para identificar funciones, establecer límites del sistema y garantizar consistencia en el flujo de datos. De esta manera, el modelo EPS no solo constituye una guía conceptual, sino que también es una herramienta práctica para el desarrollo disciplinado de soluciones computacionales.

Figura 2.4: Modelo conceptual Entrada–Proceso–Salida utilizado en el diseño de algoritmos.

2.3. Representación en lenguaje natural estructurado

La representación en lenguaje natural estructurado constituye un punto intermedio entre la descripción informal de un problema y su especificación algorítmica formal. A diferencia

del lenguaje natural convencional, este enfoque restringe las expresiones utilizadas, organiza la información en pasos claramente delimitados y prioriza la ausencia de ambigüedades. Su objetivo es facilitar el razonamiento secuencial, manteniendo la expresividad adecuada para que el algoritmo pueda ser comprendido por personas sin conocimientos avanzados de programación, lo cual es coherente con el propósito formativo de esta etapa del libro.

En concordancia con la estructura Entrada–Proceso–Salida (EPS) previamente desarrollada, el lenguaje natural estructurado permite describir la transformación de datos de forma ordenada y verificable antes de traducirla a pseudocódigo. Esta aproximación ha sido utilizada históricamente en metodologías de diseño algorítmico, pues permite identificar errores conceptuales, inconsistencias y omisiones antes de avanzar hacia representaciones más formales. Tal como ocurre en el análisis de algoritmos descrito por Cormen et al. [19], la claridad en las descripciones previas constituye un requisito para cualquier especificación computacional rigurosa.

En el ejemplo que se muestra en el listado de código 2.50, que funciona como referencia central para las subsecciones posteriores, presenta un algoritmo expresado íntegramente en lenguaje natural estructurado. Su extensión permite ilustrar decisiones secuenciales, condiciones y validaciones básicas, las mismas que se examinarán en cada apartado.

Listing 2.50: Algoritmo en lenguaje natural estructurado para calcular el índice de masa corporal (IMC)

```
1  Algoritmo CalcularIMC
2
3  Entrada:
4  - peso (kilogramos)
5  - estatura (metros)
6
7  Proceso:
8  1. Verificar que peso sea mayor que 0.
9  2. Verificar que estatura sea mayor que 0.
10 3. Calcular imc como peso entre estatura al cuadrado: imc <- peso / (estatura
     ^2)
11 4. Si imc < 18.5, entonces categoría <- "Bajo peso".
12 5. Si imc está entre 18.5 y 24.9, entonces categoría <- "Normal".
13 6. Si imc está entre 25 y 29.9, entonces categoría <- "Sobrepeso".
14 7. Si imc >= 30, entonces categoría <- "Obesidad".
15
```

```
16  Salida:
17  - Mostrar imc.
18  - Mostrar categoría.
19
20  FinAlgoritmo
```

2.3.1. Pasos numerados

El empleo de pasos numerados, como se muestra en el algoritmo del listado 2.50, permite organizar el razonamiento del programador en formación mediante una secuencia explícita y ordenada de acciones. La numeración evita interpretaciones múltiples acerca del orden de ejecución y facilita la correlación posterior con las estructuras secuenciales del pseudocódigo. Además, constituye una práctica alineada con la descripción operacional de algoritmos presentada en secciones anteriores, donde cada acción representa una transición clara del estado del sistema.

Otra ventaja de la numeración consiste en su capacidad para descomponer problemas complejos en unidades más simples y verificables. En el ejemplo del IMC, cada validación y cada cálculo aparecen como pasos independientes, permitiendo identificar fácilmente qué ocurre si un dato de entrada es incorrecto o si una condición no se cumple. La literatura sobre diseño algorítmico destaca esta modularidad como un elemento clave para el análisis y depuración temprana de procedimientos [12], [53].

Finalmente, la numeración establece una relación directa con las estructuras de control estudiadas previamente, en particular la secuencia. Cada número indica un avance lineal en el flujo de ejecución, lo que permite a los estudiantes establecer paralelos entre el lenguaje natural estructurado y los bloques de código que implementarán más adelante. Esta transición gradual reduce la dificultad cognitiva y fortalece la comprensión del proceso algorítmico.

2.3.2. Precisión y claridad

La claridad en la redacción es un componente esencial del lenguaje natural estructurado. Tal como se observa en el listado 2.50, cada paso debe contener únicamente la información necesaria, evitando descripciones redundantes o informales. Una instrucción precisa permite que distintos lectores interpreten la misma acción de manera idéntica, reduciendo la ambigüedad y aumentando la reproducibilidad del algoritmo, principio central en la

ingeniería del software según Pressman y Maxim [12].

La precisión también implica nombrar explícitamente las variables, operaciones y comparaciones involucradas. Por ejemplo, en el cálculo del IMC se indican claramente las restricciones de entrada y los rangos asociados a cada categoría. Estas expresiones eliminan posibles confusiones respecto a cómo se realiza la operación o qué intervalos determinan la clasificación. Como se enfatiza en Sommerville [53], esta especificación rigurosa es indispensable para garantizar la coherencia entre el diseño conceptual y la futura implementación.

Otro aspecto relevante es el uso consistente de conectores y palabras clave. Expresiones como “si”, “entonces”, “verificar” o “calcular” deben emplearse siempre con el mismo significado y en contextos controlados. La estandarización terminológica permite que los estudiantes desarrollen un vocabulario compartido, facilitando el trabajo colaborativo y la revisión entre pares. Además, esta práctica es coherente con los estándares utilizados en notaciones semiformales de requisitos.

Finalmente, la claridad del algoritmo también depende de la estructura visual. Separar entradas, procesos y salidas; emplear listas; y organizar las condiciones de manera explícita, como se aprecia en el ejemplo del IMC, mejora la legibilidad y hace posible identificar errores lógicos antes de convertir el algoritmo en pseudocódigo o en un lenguaje de programación.

2.3.3. Evitar ambigüedades

Uno de los propósitos fundamentales del lenguaje natural estructurado es eliminar la ambigüedad inherente al lenguaje cotidiano. Expresiones informales como “aproximadamente”, “más o menos”, “grande”, “pequeño” o “rápido” deben evitarse por completo, pues no tienen un significado operativo dentro del diseño algorítmico. El ejemplo del listado 2.50 ilustra esta precisión al utilizar comparaciones numéricas claramente definidas para cada rango de clasificación.

Asimismo, las decisiones condicionales deben diferenciarse explícitamente. Frases como “si es necesario” o “en caso contrario” solo deben utilizarse cuando el contexto está plenamente determinado. La falta de precisión puede conducir a implementaciones inconsistentes o incompatibles con el objetivo del algoritmo. La claridad en los límites del problema es un requisito destacado por Cormen et al. [58], quienes enfatizan la necesidad de definir comportamientos específicos en cada situación posible.

2.4. Pseudocódigo

El pseudocódigo constituye una representación formalizada del algoritmo que, sin pertenecer a un lenguaje de programación real, adopta convenciones precisas que permiten describir con exactitud la lógica del procedimiento. A diferencia del lenguaje natural estructurado, el pseudocódigo utiliza palabras reservadas, bloques anidados y notaciones estandarizadas que favorecen la posterior traducción a un lenguaje de programación. Su función es, por tanto, actuar como puente entre el diseño conceptual y la implementación concreta.

La figura 2.5 muestra la estructura general de un algoritmo en pseudocódigo, destacando los bloques fundamentales: encabezado, entrada, proceso y salida. Esta estructura es coherente con el modelo EPS desarrollado previamente y se vincula directamente con la construcción de programas reales.

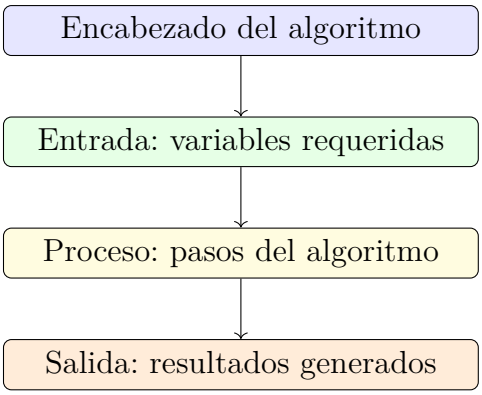

Figura 2.5: Estructura básica de un algoritmo expresado en pseudocódigo.

2.4.1. Convenciones de escritura

El pseudocódigo utiliza un conjunto de convenciones que permiten representar algoritmos de forma independiente del lenguaje. Entre estas se encuentran las palabras reservadas (p. ej., `Algoritmo`, `FinAlgoritmo`, `Si`, `Entonces`, `Mientras`), la indentación para marcar bloques y el uso explícito de operaciones básicas. El objetivo es garantizar que cualquier lector pueda comprender el algoritmo sin conocer un lenguaje específico.

Por ejemplo, la instrucción que se muestra en el listado de código 2.51, representa de manera clara la estructura condicional, independientemente del lenguaje de implementación.

Listing 2.51: Estructura de Selección/Condicional como Parte de un Algoritmo

```
Si x > 0 Entonces
  escribir "Número positivo"
FinSi
```

Palabras reservadas

Las palabras reservadas constituyen el vocabulario fundamental del pseudocódigo, ya que cada una representa una operación o estructura de control específica. Su uso permite establecer una correspondencia directa con los lenguajes reales. Entre las más utilizadas se encuentran:

Algoritmo / FinAlgoritmo — delimitan el inicio y fin del procedimiento.

Si / Entonces / SiNo / FinSi — expresan decisiones.

Mientras (hacer) / FinMientras — definen ciclos condicionales.

Para / FinPara — definen ciclos contados.

Escribir, Leer — representan operaciones de entrada/salida.

<- — operador de asignación.

Estas palabras se emplean cuando el algoritmo requiere expresar control del flujo, iteraciones o decisiones lógicas, y su propósito es eliminar variaciones posibles del lenguaje natural. Al mantener este conjunto reducido, el pseudocódigo preserva claridad y evita interpretaciones ambiguas, alineándose con las recomendaciones formales de representación descritas en la sección 2.2.

Indentación

La indentación es una convención visual esencial para identificar bloques lógicos, anidamientos y alcances dentro del algoritmo. Su propósito es facilitar la lectura y permitir que el lector distinga rápidamente qué instrucciones pertenecen a qué estructura de control. Se utiliza siempre que se inicia una estructura como `Si`, `Mientras`, `Para`, entre otras.

Desde la perspectiva pedagógica, la indentación ayuda al programador en formación a internalizar los niveles jerárquicos dentro de un algoritmo, lo cual será fundamental cuando trabaje con lenguajes reales donde la organización del código influye en la mantenibilidad y comprensión general del programa.

2.4.2. Construcción de sentencias

Las sentencias en pseudocódigo permiten expresar operaciones básicas tales como asignación, entrada/salida, decisiones y ciclos. Su finalidad es representar el comportamiento del algoritmo de manera explícita, manteniendo correspondencia con los lenguajes reales de programación. A continuación se presentan ejemplos de cada tipo, vinculados lógicamente con el algoritmo del listado de código 2.50.

Asignación

La asignación permite establecer o actualizar el valor de una variable. En pseudocódigo se escribe mediante el operador `<-`. Existen asignaciones simples, matemáticas como la asignación del paso 3, y dependientes de condiciones previas como las que se muestran en los pasos 4, 5, 6 y 7 del listado de código 2.50. Según el mencionado algoritmo, en el segmento de código 2.52 se muestran las dos asignaciones, sin y con restricción:

Listing 2.52: Algoritmo en lenguaje natural estructurado para calcular el índice de masa corporal (IMC)

```
3. imc <- peso / (estatura^2)
.....
5. categoria <- "Normal"
```

En algoritmos más complejos, la asignación puede expresar actualizaciones dentro de ciclos o acumulaciones progresivas. A continuación se presentan dos ejercicios matemáticos.

Ejercicio 1: El objetivo de este ejercicio es calcular la suma de los primeros diez números enteros positivos utilizando un acumulador. Esta actividad permite comprender cómo una variable puede modificarse progresivamente dentro de un ciclo y muestra el rol fundamental de la asignación en procesos iterativos. El programador en formación debe identificar cómo el valor inicial del acumulador evoluciona en función del índice del ciclo, reforzando la noción de actualización de estado.

La solución implementa un ciclo contado que recorre los valores del 1 al 10 y acumula cada uno de ellos en la variable `total`. El algoritmo 2.53 ejemplifica adecuadamente el uso combinado de asignación inicial, actualización progresiva y salida final, permitiendo visualizar cómo la estructura de control dirige la transformación paso a paso del valor acumulado.

Listing 2.53: Cálculo de una suma acumulada

```
Algoritmo SumaAcumulada
1. total <- 0
2. Para i desde 1 hasta 10 hacer
total <- total + i
FinPara
3. Escribir total
FinAlgoritmo
```

Ejercicio 2: En este ejercicio se requiere evaluar una expresión polinómica para un valor específico de `x`. El propósito es mostrar cómo las asignaciones pueden utilizar operaciones aritméticas más complejas, incorporando exponentes, productos y sumas. Este tipo de cálculo es común en problemas físicos, económicos y matemáticos donde se necesita evaluar funciones de forma programática.

El algoritmo 2.54 presenta una asignación inicial para la variable `x` y posteriormente calcula el valor del polinomio mediante una expresión matemática directa. La solución refuerza el uso de la asignación como mecanismo para almacenar tanto valores iniciales como resultados intermedios y finales.

Listing 2.54: Asignación con potencias y productos

```
Algoritmo OperacionesMatematicas
1. x <- 3
2. y <- 2*x^2 + 5*x - 8
3. Escribir y
FinAlgoritmo
```

Entrada/Salida

Las sentencias de entrada y salida permiten la comunicación entre el usuario y el algoritmo. Se expresan mediante `Leer` y `Escribir`. Su diseño en pseudocódigo debe ser explícito, indicando qué datos se solicitan y qué valores se presentan.

El propósito de este ejercicio es solicitar al usuario un número entero y posteriormente mostrarlo por pantalla (ver listado de código 2.55). Esta actividad constituye el caso más elemental del flujo de entrada y salida, y permite comprender cómo un algoritmo interactúa con el usuario final. También puede introducir la necesidad de validar que el dato ingresado sea adecuado para su posterior procesamiento.

Listing 2.55: Instrucciónes de: lectura t escritura

```
Leer peso
Escribir "El IMC es:", imc
```

Condicionales

Las estructuras condicionales permiten que un algoritmo seleccione un camino de ejecución dependiendo del resultado de una evaluación lógica. Su función es controlar el flujo del programa ante situaciones donde el comportamiento no puede ser lineal, lo cual es esencial para modelar decisiones. La literatura coincide en que la selección es uno de los tres pilares fundamentales de la computación algorítmica [58], [68].

El listado 2.56 evidencia cómo un conjunto de decisiones encadenadas permite clasificar un valor numérico dentro de rangos bien definidos. Este tipo de estructura se fundamenta en dividir el dominio de entrada en subconjuntos lógicos, asignando un comportamiento explícito a cada uno. Esta técnica es ampliamente utilizada en sistemas de decisión expertos y en validación de datos en aplicaciones reales [3], [12].

Listing 2.56: Clasificación de un número

```
Algoritmo ClasificarNumero
1. Leer n
2. Si n > 0 Entonces
3.  Escribir "Positivo"
4. SiNo
5.    Si n = 0 Entonces
6.      Escribir "Cero"
7.    SiNo
8.      Escribir "Negativo"
9.    FinSi
10. FinSi
FinAlgoritmo
```

Las condicionales pueden ser simples, dobles o anidadas. Una condicional simple ejecuta una acción únicamente si se cumple la condición. En la condicional doble se introduce una ruta alternativa para cuando esta no se cumple, y en las condicionales anidadas se permite evaluar condiciones adicionales dentro de otras evaluaciones. Esta clasificación ha sido utilizada ampliamente en la enseñanza inicial de programación por su progresión cognitiva clara [7], [69].

La claridad en la definición de cada condición es imprescindible para reducir ambigüedades. Las buenas prácticas recomiendan evitar expresiones vagas, rangos indefinidos o comparaciones incompletas, pues dificultan la futura implementación. Estudios en ingeniería del software señalan que gran parte de los errores en el código fuente provienen de especificaciones ambiguas o mal definidas en fases tempranas [70], [71].

Llas estructuras condicionales constituyen un puente conceptual que facilita al programador junior la transición hacia lenguajes como Python, Java o C, los cuales emplean operadores y sintaxis específicas pero basadas en los mismos principios. Este vínculo conceptual es clave en el desarrollo de habilidades de pensamiento computacional [64], [72], [73]

Ejercicio 1: Se desea diseñar un algoritmo que determine si un número ingresado por el usuario es positivo, negativo o cero. Este tipo de clasificación numérica permite introducir el uso de condicionales anidadas, las cuales son necesarias cuando un problema requiere múltiples comparaciones secuenciales dentro de un mismo proceso de decisión.

El algoritmo mostrado en el listado 2.56 aplica una estructura condicional doble con una anidación interna para evaluar los tres casos posibles. Esta solución muestra cómo las rutas alternativas dependen jerárquicamente de la condición original, reforzando la idea de bifurcación controlada del flujo.

Ejercicio 2: Se requiere determinar si un número entero es divisible por 3. Este ejercicio permite comprender la relación entre operaciones aritméticas y decisiones lógicas, mostrando cómo el operador módulo (`mod`) facilita verificar residuales en problemas de divisibilidad, ampliamente utilizados en análisis numérico y validación de datos.

El algoritmo del listado 2.57 evalúa si el residuo de dividir el número entre 3 es igual a cero. Esta verificación determina qué mensaje se imprime, lo cual ejemplifica el uso de condicionales simples y fortalece la comprensión del vínculo entre evaluación numérica y ramificación lógica.

Listing 2.57: Verificación de divisibilidad

```
1 Algoritmo VerificarDivisibilidad
2 1. Leer n
3 2. Si n mod 3 = 0 Entonces
4 3.    Escribir "Es divisible por 3"
5 4. SiNo
6 5.    Escribir "No es divisible por 3"
7 6. FinSi
8 FinAlgoritmo
```

Ciclos

Los ciclos permiten repetir un conjunto de instrucciones mientras una condición sea verdadera o durante un número determinado de iteraciones. Esta estructura resulta esencial para resolver problemas que requieren procesamiento repetitivo o acumulativo, como sumas parciales, recorridos de listas o simulaciones computacionales [7], [58].

En su forma general, un ciclo puede controlarse mediante un contador (ciclo determinado) o mediante una condición lógica (ciclo indeterminado). Esta distinción responde a necesidades distintas: cuando se conoce el número de iteraciones, se emplea un ciclo contado; cuando se desconoce y depende del estado del sistema, se usa un ciclo condicionado [3], [12].

El uso adecuado de ciclos favorece la eficiencia algorítmica al evitar repeticiones innecesarias. De hecho, una gestión incorrecta de las condiciones de parada es responsable de errores comunes como ciclos infinitos, lo cual constituye una de las fallas típicas en la programación inicial [69], [70].

Los ciclos también permiten estructurar cálculos iterativos que dependen de valores ingresados por el usuario. Esto facilita desarrollar algoritmos interactivos que ajustan su comportamiento dinámicamente según las entradas recibidas, un aspecto clave en aplicaciones prácticas y simulaciones computacionales [64], [72], [73].

Los ciclos constituyen una herramienta expresiva indispensable para representar procesos repetitivos de la vida real, tales como sumatorias, barridos, comprobaciones sucesivas o acumulaciones. Su comprensión prepara al programador en formación para estructuras avanzadas como iteradores, generadores o recorridos de colecciones en lenguajes modernos [60], [74].

Ejercicio 1: El objetivo es imprimir los números del 1 al 10 utilizando un ciclo contado. Este tipo de estructura permite que el programador en formación aprenda cómo el algoritmo controla el avance del índice, asegurando que el número de iteraciones sea conocido desde el inicio.

El algoritmo 2.58 utiliza un ciclo `Para` que incrementa automáticamente la variable de control `i`. Cada iteración imprime su valor, lo que permite observar el comportamiento secuencial del ciclo contado.

Listing 2.58: Ciclo contado para imprimir números

```
Algoritmo ImprimirNumeros
1. Para i desde 1 hasta 10 hacer
2.    Escribir i
3. FinPara
FinAlgoritmo
```

Ejercicio 2: Se requiere construir un algoritmo que permita sumar números ingresados por el usuario hasta que la suma total alcance o supere el valor límite de 50. Este ejercicio enfatiza el uso de ciclos controlados por condición, donde la cantidad de iteraciones no es conocida de antemano.

El algoritmo 2.59 emplea un ciclo `Mientras`, repitiendo la lectura y el proceso de acumulación hasta cumplir la condición. Este enfoque refleja el carácter dinámico de los ciclos condicionados.

Listing 2.59: Ciclo condicional para sumar hasta superar un límite

```
Algoritmo SumaHastaLimite
1. total <- 0
2. Mientras total < 50 hacer
3.    Escribir "Ingrese un número:"
4.    Leer x
5.    total <- total + x
6. FinMientras
7. Escribir "Suma final:", total
FinAlgoritmo
```

Ejercicio 3: Se desea diseñar un algoritmo que permita calcular el promedio de n valores ingresados por el usuario. Para ello, el programa debe solicitar primero la cantidad total de datos y luego leer cada uno de los valores dentro de un ciclo contado. Este ejercicio permite comprender el uso de acumuladores y el control del flujo mediante un ciclo `Para`, reforzando la importancia de la estructura iterativa en tareas de agregación numérica.

El algoritmo presentado en el listado de código 2.60 utiliza un acumulador para sumar los valores ingresados y, posteriormente, divide la suma total entre la cantidad de datos registrados. El uso del ciclo `Para` garantiza que el algoritmo ejecute exactamente n iteraciones, lo cual facilita el control del proceso y evita condiciones indeterminadas.

Listing 2.60: Cálculo del promedio de n valores

```
Algoritmo PromedioNValores
```

```
1.  Escribir "Ingrese la cantidad de valores:"
2.  Leer n
3.  suma <- 0
4.  Para i desde 1 hasta n hacer
5.    Escribir "Ingrese el valor ", i, ":"
6.    Leer x
7.    suma <- suma + x
8.  FinPara
0.  promedio <- suma / n
10. Escribir "El promedio es:", promedio
FinAlgoritmo
```

Ejercicio 4: Se desea diseñar un algoritmo que simule un cronómetro de cuenta regresiva. El usuario ingresa un número entero n que representa los segundos iniciales. El algoritmo debe mostrar cada segundo restante y, adicionalmente, emitir mensajes especiales cuando falten exactamente 10, 5 y 1 segundo. Este problema **solo puede resolverse adecuadamente con un ciclo decremental**, ya que los eventos dependen de la cercanía al final de la cuenta y no del progreso ascendente. Usar un ciclo ascendente generaría una lógica antinatural y obligaría a cálculos inversos innecesarios.

Una de las soluciones que se puede dar al problema del ejercicio 4 es la que muestra el algoritmo del listado 2.61. Esa solución utiliza un ciclo `Para` que inicia en n y decrementa hasta 0. En cada iteración se evalúan los segundos restantes y, cuando coinciden con los puntos críticos definidos, se muestra la advertencia correspondiente. Gracias al recorrido descendente, la lógica se expresa de manera directa, clara y coherente con el funcionamiento real de un temporizador.

Listing 2.61: Cuenta regresiva con avisos en tiempos críticos

```
Algoritmo TemporizadorConEventos
1.  Escribir "Ingrese el número de segundos iniciales:"
2.  Leer n
3.  Para i desde n hasta 0 hacer paso -1
4.    Escribir "Tiempo restante:", i, " segundos"
5.    Si i = 10 Entonces
6.      Escribir "¡Advertencia: quedan 10 segundos!"
7.    FinSi
8.    Si i = 5 Entonces
9.      Escribir "¡Atención: solo 5 segundos restantes!"
```

```
  10.   FinSi
  11.   Si i = 1 Entonces
  12.     Escribir "¡Último segundo!"
  13.   FinSi
  14. FinPara
  15. Escribir "Fin de la cuenta regresiva."
FinAlgoritmo
```

2.5. Diagramas de flujo

Los diagramas de flujo son representaciones gráficas del comportamiento lógico de un algoritmo mediante símbolos estandarizados y conectores que permiten visualizar el flujo de control. Su finalidad es clarificar la estructura del procedimiento, facilitar la comprensión del proceso y establecer una transición natural entre el análisis del problema y la implementación formal. Diversas fuentes coinciden en que los diagramas de flujo constituyen una herramienta pedagógica fundamental para reforzar el pensamiento computacional y el razonamiento algorítmico [7], [75].

La utilidad de los diagramas de flujo reside en su capacidad para eliminar ambigüedades al representar operaciones, decisiones y secuencias de forma visual. Esta característica facilita la revisión entre pares, el análisis de errores y la validación de la lógica antes de escribir pseudocódigo o código real en un lenguaje de programación. Además, permiten identificar estructuras como ciclos y condicionales de manera inmediata, lo cual favorece la comprensión del flujo de ejecución.

En consecuencia, los diagramas de flujo sirven como documentación técnica, facilitando la comunicación entre equipos de desarrollo o entre docentes y estudiantes. Su estandarización ha permitido que se mantengan como un recurso esencial en el aprendizaje inicial de algoritmos, complementando otras representaciones formales como el pseudocódigo [3], [12].

2.5.1. Símbolos básicos

Los símbolos básicos constituyen el vocabulario visual fundamental de los diagramas de flujo, permitiendo representar las acciones, decisiones y operaciones que conforman un algoritmo. Cada símbolo posee una semántica particular y una forma estandarizada, lo cual contribuye a que la interpretación del diagrama sea clara e inequívoca, independientemente

del lector o del contexto. La estandarización de estos símbolos facilita la enseñanza inicial de la programación y reduce la ambigüedad durante el análisis de sistemas [7], [75].

El uso correcto de los símbolos garantiza que el flujo del algoritmo sea comprensible tanto para estudiantes como para profesionales, actuando como una herramienta intermedia entre el lenguaje natural estructurado y el pseudocódigo. Asimismo, los diagramas de flujo permiten identificar de forma visual estructuras como secuencias, decisiones y ciclos, lo que resulta útil al validar la lógica del procedimiento antes de su codificación [3], [12].

Además, los símbolos básicos facilitan la construcción modular de algoritmos más complejos, debido a que cada componente puede combinarse con otros para representar procesos iterativos o ramificados. Esta modularidad gráfica favorece la depuración conceptual en etapas tempranas del diseño, previniendo errores de interpretación o inconsistencias lógicas [69].

La correcta disposición de los símbolos promueve la legibilidad del diagrama, permitiendo un flujo natural generalmente descendente y de izquierda a derecha. Seguir las convenciones visuales internacionales asegura que los diagramas puedan ser interpretados de manera uniforme, incluso por lectores externos al entorno académico o profesional del autor [70].

Inicio/fin

El inicio y el fin de un diagrama de flujo se representan mediante un óvalo o un rectángulo con bordes redondeados. En su interior se escribe la palabra correspondiente a la función del símbolo, ya sea *Inicio* o *Fin*. Este elemento indica los límites del algoritmo y asegura que el flujo esté correctamente definido desde su punto de entrada hasta su terminación. Su presencia es obligatoria en cualquier diagrama y constituye la base estructural sobre la cual se desarrolla el resto del flujo [75].

La Figura 2.6 ilustra el símbolo estándar para inicio/fin utilizado en diagramas de flujo, siguiendo la notación más empleada en textos de ingeniería y ciencias de la computación.

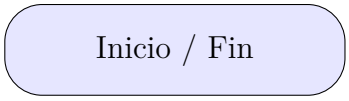

Figura 2.6: Símbolo estándar de inicio y fin en un diagrama de flujo.

Proceso

El símbolo de proceso se representa mediante un rectángulo de bordes rectos y se utiliza para indicar una operación que el algoritmo debe ejecutar de forma determinista. Estas

operaciones pueden incluir asignaciones, cálculos aritméticos, transformaciones o cualquier instrucción interna que no implique bifurcación del flujo. Su uso es esencial para expresar la secuencia lógica del procedimiento y constituye la base de la ejecución operacional del algoritmo [60], [74].

Como ejemplo, en la Figura 2.7, se muestra el símbolo de proceso ejecutando una operación de incremento:

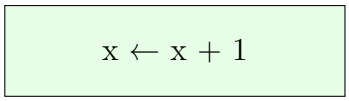

Figura 2.7: Símbolo de proceso para representar una operación interna.

Decisión

El símbolo de decisión se representa mediante un rombo y permite expresar bifurcaciones en el flujo del algoritmo. En su interior se escribe una condición lógica cuya evaluación determinará qué camino seguirá el flujo. Este símbolo es fundamental para modelar estructuras selectivas y tomar decisiones basadas en valores de entrada o estados intermedios. Los estudios sobre pensamiento computacional destacan que la bifurcación es un concepto clave para comprender la lógica condicional [64], [72], [73].

Caso 1: Rama verdadera. Cuando la condición se evalúa como verdadera, el flujo continúa por la rama etiquetada como *Sí*. Esta ruta contiene las acciones que deben ejecutarse únicamente cuando la condición se cumple. La Figura 2.8 ilustra un ejemplo simple.

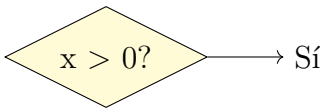

Figura 2.8: Rama verdadera del símbolo de decisión.

Caso 2: Rama falsa. Si la condición se evalúa como falsa, el flujo continúa por la rama *No*. Esto permite definir acciones alternativas que solo deben ejecutarse en ausencia del cumplimiento de la condición. La Figura 2.9 representa este comportamiento.

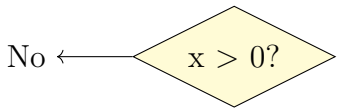

Figura 2.9: Rama falsa del símbolo de decisión.

Unión de ambas ramas y continuación del flujo. En muchos algoritmos, independientemente de si la condición evaluada resulta verdadera o falsa, ambas ramas deben converger posteriormente para continuar con un mismo conjunto de instrucciones. Esta convergencia representa el cierre lógico de una estructura condicional del tipo *Si–SiNo* y permite retomar el flujo secuencial del algoritmo. Desde el punto de vista conceptual, esta unión garantiza que el proceso no quede fragmentado y que todas las posibles rutas de ejecución conduzcan a un estado común, lo cual es esencial para la corrección y completitud del algoritmo [72], [73].

La Figura 2.10 muestra un ejemplo completo de un símbolo de decisión con sus dos ramas —verdadera y falsa— que, tras ejecutar acciones distintas, se unen nuevamente para continuar con el flujo normal del diagrama. Este patrón es uno de los más utilizados en la construcción de diagramas de flujo y tiene una correspondencia directa con las estructuras condicionales utilizadas en pseudocódigo y lenguajes de programación.

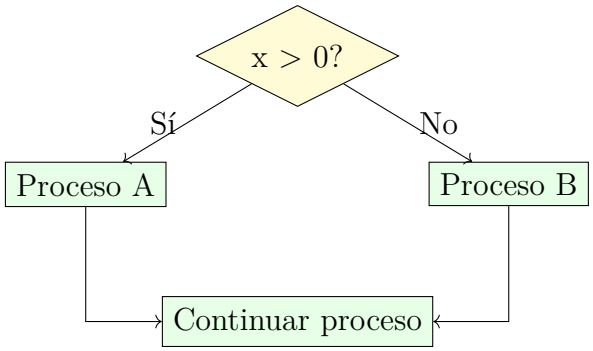

Figura 2.10: Decisión con ramas verdadera y falsa que convergen para continuar el flujo del algoritmo.

Entrada/salida

El símbolo de entrada/salida se representa mediante un paralelogramo y se utiliza para indicar tanto la captura de datos desde el usuario como la presentación de resultados. Este símbolo permite incorporar la interacción dentro del algoritmo y es esencial en aplicaciones que requieren comunicación con el entorno. La teoría de interacción humano–computadora

destaca su importancia para representar interfaces básicas en modelos algorítmicos [76].

Caso 1: Lectura de datos. Cuando el símbolo representa una operación de entrada, se especifica dentro de él la acción de lectura, indicando el dato que debe ser proporcionado por el usuario. La Figura 2.11 muestra un ejemplo de lectura.

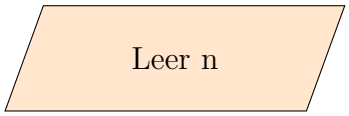

Figura 2.11: Símbolo de entrada para captura de datos.

Caso 2: Escritura de datos. Cuando representa una salida, el símbolo contiene el mensaje o dato que se desea mostrar al usuario. La Figura 2.12 ilustra este uso.

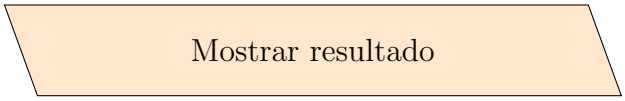

Figura 2.12: Símbolo de salida para presentación de información.

Entradas especializadas según el origen de los datos. En los diagramas de flujo, además de indicar la lectura de datos de forma genérica, es posible especificar el *origen* desde el cual se obtienen dichos datos. Esta diferenciación resulta especialmente relevante en algoritmos que interactúan con distintos medios de entrada, como dispositivos físicos o fuentes de almacenamiento. Representar explícitamente el origen de los datos mejora la comprensión del algoritmo y facilita su posterior implementación en sistemas reales [12], [77].

Entrada manual (teclado). La entrada manual representa la introducción de datos realizada directamente por una persona, generalmente mediante el uso del teclado u otro dispositivo de entrada manual. En los diagramas de flujo, este tipo de entrada se representa mediante el símbolo de *entrada manual*, caracterizado por tener la forma de un polígono similar a un rectángulo, pero con el borde superior inclinado y de menor longitud que el inferior. Esta forma distingue claramente la entrada manual de otras fuentes de datos, como archivos o dispositivos automáticos. La Figura 2.13 muestra el símbolo estándar utilizado para representar la entrada manual en diagramas de flujo, de acuerdo con la simbología ampliamente aceptada.

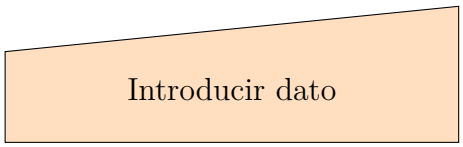

Figura 2.13: Símbolo de entrada manual (teclado) en un diagrama de flujo.

Salida por pantalla (visualización).

Salida por pantalla (visualización). La salida por pantalla representa una operación mediante la cual el sistema muestra información directamente al usuario a través de un dispositivo de visualización, como un monitor o una interfaz gráfica. En los diagramas de flujo, este tipo de salida no se representa con el símbolo genérico de entrada/salida, sino mediante el *símbolo de visualización* (*display*), el cual permite distinguir la presentación visual de información de otros tipos de salida, como la impresión de documentos o el almacenamiento en archivos. Esta diferenciación resulta fundamental para una correcta interpretación del flujo de información dentro del sistema.

El símbolo de visualización se caracteriza por una forma similar a un rectángulo con uno de sus lados curvado, lo que indica que la información es mostrada al usuario de manera directa y no almacenada ni transmitida a otro medio. De acuerdo con la simbología empleada en herramientas de diagramación ampliamente aceptadas, como SmartDraw, este símbolo se utiliza específicamente para representar operaciones de salida visual, tales como mostrar resultados, mensajes o datos procesados en pantalla [78].

Desde el punto de vista del análisis y diseño de sistemas, la separación entre salidas visuales y otros tipos de salida contribuye a una mayor claridad del modelo y facilita la transición del diagrama de flujo hacia su implementación en software. Autores clásicos en el área de sistemas de información y programación estructurada recomiendan el uso explícito del símbolo de visualización cuando la interacción con el usuario se produce a través de la pantalla, ya que esto mejora la legibilidad del diagrama y reduce ambigüedades durante el desarrollo y mantenimiento del sistema [77], [79].

Figura 2.14: Símbolo de Documento impreso.

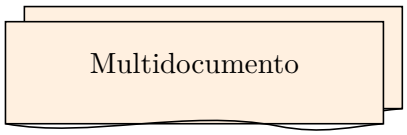

Figura 2.15: Símbolo de Multidocumento.

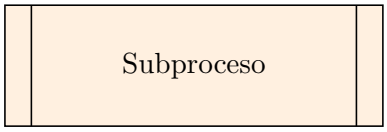

Figura 2.16: Símbolo de Subproceso (proceso predefinido).

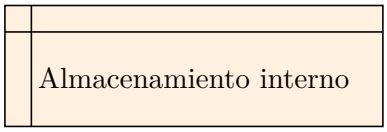

Figura 2.17: Símbolo de Almacenamiento interno.

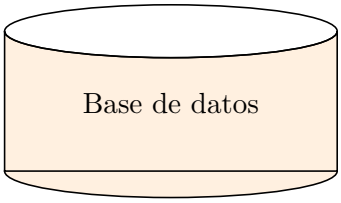

Figura 2.18: Símbolo de Base de datos (almacenamiento persistente).

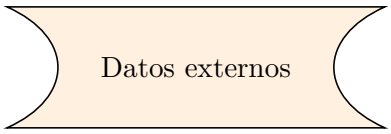

Figura 2.19: Símbolo de Datos externos.

Figura 2.20: Símbolo de Tira (paper tape).

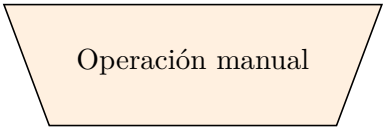

Figura 2.21: Símbolo de Operación manual.

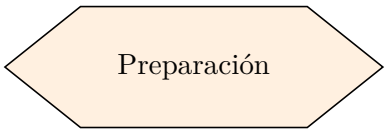

Figura 2.22: Símbolo de Preparación.

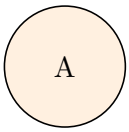

Figura 2.23: Símbolo de Conector (continuidad del flujo).

2.5.2. Construcción de diagramas

2.5.3. Correspondencia con pseudocódigo

2 PÁRRAFOS O LOS QUE ESTIMES CONVENIENTE PARA DAR GRAN DETALLE. Un problema, y los pasos para dar la solución, el pseudocódigo o algoritmo y el diagrama de flujo.

2.6. Refinamiento sucesivo y descomposición

3 PÁRRAFOS - AGREGAR FIGURA

2.6.1. De un problema grande a subproblemas

3 PÁRRAFOS - AGREGAR FIGURA

2.6.2. Construcción de un árbol de descomposición

5 PÁRRAFOS - AGREGAR FIGURA

2.6.3. Ventajas del diseño modular

LOS PÁRRAFOS QUE SEAN NECESARIOS. UNO POR CADA VENTAJA - AGREGAR LAS IMÁGENES NECESARIAS PARA EXPLICAR MEJOR SUS VENTAJAS.

2.7. Errores típicos en el diseño algorítmico

4 PÁRRAFOS

2.7.1. Casos no contemplados

1 PÁRRAFO PARA CADA CASO.

2.7.2. Condiciones incompletas

3 PÁRRAFOS

2.7.3. Ciclos sin salida

1 PÁRRAFO PARA CADA CASO QUE SE PUEDE PRESENTAR. AGREGAR LA FIGURA CORRESPONDIENTE

2.7.4. Inconsistencias con el problema

1 PÁRRAFO PARA CADA TIPO DE INCONSISTENCIA QUE SE PUEDE PRESENTAR. AGREGAR LA FIGURA CORRESPONDIENTE

Parte II

Programación Estructurada: Datos, Control y Estructuras

Capítulo 3

Lenguajes de Programación e Introducción al Código

3.1. Lenguajes de programación en el contexto de los sistemas de información

3.1.1. Lenguajes de bajo nivel

3.1.2. Lenguajes de alto nivel

3.1.3. Paradigmas de programación

Imperativo

Funcional

Orientado a objetos

3.2. Compiladores, intérpretes y máquinas virtuales

3.2.1. Etapas del compilador

Análisis léxico

Análisis sintáctico

Generación de código

3.2.2. Ejecución interpretada

3.2.3. Bytecode y máquinas virtuales

3.3. Estructura mínima de un programa de compu-

Capítulo 4

Tipos de Datos, Operadores, Variables y Expresiones

4.1. Correspondencia entre tipos lógicos y tipos del lenguaje

4.1.1. Enteros

4.1.2. Reales

4.1.3. Caracteres

4.1.4. Booleanos

4.2. Declaración, inicialización y ámbito de variables

4.2.1. Declaración

4.2.2. Inicialización

4.2.3. Ámbito

Variables locales

Variables globales

4.3. Operadores aritméticos, relacionales y lógicos

4.3.1. Aritméticos

4.3.2. Relacionales

Capítulo 5

Estructuras de Control: Selección y Repetición

5.1. Estructuras condicionales

5.1.1. If simple

5.1.2. If–else

5.1.3. If anidado

5.1.4. Switch/case

5.2. Estructuras de repetición

5.2.1. While

5.2.2. Do–while

5.2.3. For

5.2.4. Control de iteración (break/continue)

5.3. Diseño de algoritmos condicionales frecuentes

5.3.1. Decisiones por rangos

5.3.2. Validación de datos

5.3.3. Selección múltiple

5.4. Diseño de algoritmos iterativos frecuentes

Capítulo 6

Arreglos y Datos Estructurados

6.1. Arreglos unidimensionales

6.1.1. Definición

6.1.2. Declaración

6.1.3. Inicialización

6.1.4. Recorrido

6.1.5. Modificación

6.2. Arreglos bidimensionales

6.2.1. Tablas

6.2.2. Matrices

6.2.3. Procesamiento fila–columna

6.3. Operaciones típicas sobre colecciones

6.3.1. Búsqueda lineal

6.3.2. Conteo

6.3.3. Modificación simple

6.4. Introducción intuitiva a ordenamiento y búsqueda

6.4.1. Intercambio de valores entre variables

Parte III

Modularidad, Abstracción y Proyecto Integrador

Capítulo 7

Funciones, Procedimientos y Modularidad

7.1. Motivación de la modularidad

7.1.1. Limitaciones de los programas monolíticos

7.1.2. Ventajas de dividir el programa en módulos

Separación de responsabilidades

Aislamiento de errores y facilidad de depuración

Facilidad de actualización y mantenimiento

7.1.3. Relación entre modularidad y calidad del software

Legibilidad del código

Reutilización en proyectos futuros

Favorecer el trabajo colaborativo

7.1.4. Modularidad en distintos paradigmas de programación

Modularidad en programación estructurada

Modularidad en programación orientada a objetos

Modularidad en programación funcional

7.2. Definición y sintaxis de funciones y procedimientos

7.2.1. Conceptos fundamentales

Capítulo 8

Proyecto Integrador de Programación Básica

8.1. Planteamiento del proyecto

8.1.1. Contexto del proyecto

Importancia de resolver problemas reales

Relación con áreas típicas de sistemas de información

Dominios recomendados: académico, comercial, administrativo

8.1.2. Justificación del proyecto

Necesidad de integrar todo lo aprendido

Desarrollo de competencias profesionales

Simulación del trabajo real del programador

8.1.3. Propuesta de escenarios de proyecto

Sistema simple de inventarios

Gestor de estudiantes o calificaciones

Agenda o sistema de turnos

Simulaciones interactivas por consola

8.1.4. Alcance esperado del proyecto

Nivel de complejidad apropiado para programación básica

Requisitos obligatorios y opcionales

Apéndice A

Introducción al control de versiones con Git

Apéndice B

Recomendaciones para el trabajo autónomo en programación

Apéndice C

Glosario de términos fundamentales

Apéndice D

Bibliografía recomendada

Bibliografía

[1] F. M. González-Longatt, "Introducción a los Sistemas de Información: Fundamentos," *Manuscrito (Universidad Experimental Politécnica de la Fuerza Armada, Venezuela)*, pág. 7, 2007, pdf disponible en línea. dirección: `https://www.uv.mx/personal/artulopez/files/2012/08/fundamentossistemasinformacion.pdf`.

[2] J. M. Rodríguez Rodríguez y M. J. Daureo Campillo, *Sistemas de Información: Aspectos Técnicos y Legales*. Almería, España: Universidad de Almería, 2003. dirección: `http://biblioteca.udgvirtual.udg.mx/jspui/handle/123456789/973`.

[3] I. Sommerville, *Software Engineering*, 10.ª ed. Harlow, England: Pearson, 2015, pág. 816, obra clásica que define el rol integral del ingeniero de software, desde requisitos hasta mantenimiento, ISBN: 978-0133943030.

[4] B. Ekmekci, C. E. McAnany y C. Mura, "An Introduction to Programming for Bioscientists: A Python-based Primer," *PLOS: Computational Biology*, vol. 12, n.º 6, e1004867, 2016, Documento introductorio que resalta la importancia de la programación para automatización y procesamiento de datos. DOI: `10.1371/journal.pcbi.1004867`.

[5] R. Lopez-Pablos, "Elementos de ingeniería de explotación de la información: réplica y algunos trazos sobre teoría informática," *Asociación Argentina de Economía Política*, n.º XLVIII Reunión Anual, págs. 1-4, 2013, 4 páginas, 1 figura. Publicado en castellano., ISSN: 1852-0022. DOI: `10.48550/arXiv.1312.4076`.

[6] X. Molero, C. Juiz y M. J. Rodeno, "Evaluación y modelado del rendimiento de los sistemas informáticos," *arXiv preprint arXiv:2508.16996*, 2025, Libro/monografía sobre modelado y evaluación de sistemas informáticos. DOI: `10.48550/arXiv.2508.16996`. dirección: `https://arxiv.org/abs/2508.16996`.

[7] L. J. Aguilar, *Fundamentos de Programación: Algoritmos, Estructuras de Datos y Objetos*, 3.ª ed. Madrid: McGraw-Hill, 2020, pág. 744, ISBN: 978-6071514684.

[8] Central Washington University (CWU) Faculty, *Information Systems Engineering vs. Computer Engineering*, Online article, Diferencia entre Ingeniería en Sistemas de Información (IS) y la Ingeniería de Computación / Informática, 2025. dirección: https://www.cwu.edu.tr/information-systems-engineering-vs-computer-engineering/.

[9] M. Alenezi, "Software and Security Engineering in Digital Transformation," *arXiv preprint arXiv:2201.01359*, 2021, discusión sobre el rol crítico de la ingeniería de software en la transformación digital organizacional. DOI: 10.48550/arXiv.2201.01359. dirección: https://arxiv.org/abs/2201.01359.

[10] W. Naufal, N. Dimas, N. Tauhid y H. Sulistiani, "The Role of Software Engineering in Digital Transformation of Industry 4.0," en *5th International Conference on Information Technology and Security (IC-ITECHS 2024)*, vol. 5, Indonesia: Universitas Bina Nusantara, 2024. DOI: 10.32664/ic-itechs.v5i1.1623.

[11] J. Miquel y J. Raya Martos, *Introducción a la ingeniería del software*. Barcelona, España: Fundació per a la Universitat Oberta de Catalunya (FUOC), 2013, $PID_0$0198149, Material docente. Licencia CC BY-NC-ND 3.0 ES. dirección: https://openaccess.uoc.edu/server/api/core/bitstreams/fab2269c-d5ae-4e93-934a-b0d91601d0f7/content.

[12] R. S. Pressman y B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 9.ª ed. New York: McGraw-Hill Education, 2019, pág. 704, ISBN: 978-1259872976.

[13] A.-A. Vărzaru y otros, "Digital Transformation and Innovation: The Influence of Digital Technologies on Turnover from Innovation Activities and Types of Innovation," *Systems*, vol. 12, n.º 9, 359, 2024, analiza cómo las tecnologías digitales, habilitadas por software, impulsan innovación y ventajas competitivas. DOI: 10.3390/systems12090359.

[14] M. Fadhlurrahman y otros, "An Academic Analysis of Digital Transformation: A Comprehensive Review of Literature and Business Strategies," *Scientific Information Technology Journal*, vol. 1, n.º 1, págs. 22-40, 2024, revisión del impacto de la transformación digital en modelos de negocio, procesos operativos y eficiencia organizacional. dirección: https://sitjournal.com/sitj/article/download/4/3/36.

[15] A. Bitkowska, B. Detyna y J. Detyna, "Importance of IT systems in integration of knowledge and business process management," *Information Systems International Journal*, vol. 23, n.º 1, págs. 117-130, 2022, Estudio empírico que muestra

relación positiva entre uso de sistemas de TI, BPM y calidad/efectividad de procesos organizacionales. DOI: 10.48009/1_iis_2022_109.

[16] A. Fazlollahi, U. Franke y J. Ullberg, "Benefits of Enterprise Integration: Review, Classification, and Suggestions for Future Research," en *Enterprise Interoperability. IWEI 2012*, ép. Lecture Notes in Business Information Processing, M. van Sinderen, P. Johnson, X. Xu y G. Doumeingts, eds., vol. 122, Berlin, Heidelberg: Springer, 2012, págs. 34-45. DOI: 10.1007/978-3-642-33068-1_5.

[17] A. Martín-Navarro, M. P. Lechuga Sancho y J. A. Medina-Garrido, "BPMS for management: a systematic literature review," *Revista Espanola de Documentacion Cientifica*, vol. 41, n.º 3, e213, 2018, revisión sistemática reciente sobre los sistemas de gestión de procesos de negocio (BPMS) como tecnología que automatiza procesos organizacionales. DOI: 10.3989/redc.2018.3.1532.

[18] M. Blahušiaková, "Business process automation: New challenges to increasing the efficiency and competitiveness of companies," *Strategic Management*, vol. 28, n.º 3, págs. 18-33, 2023, análisis del impacto de la automatización mediante software en procesos de negocio y contabilidad. DOI: 10.5937/StraMan2300038B.

[19] P. E. L. Zahir Irani Marinos Themistocleous, "The impact of enterprise application integration on information system lifecycles," *Information & Management*, vol. 41, n.º 2, págs. 177-187, 2003, muestra cómo la integración mediante software permite coordinar procesos organizacionales previamente aislados. DOI: 10.1016/S0378-7206(03)00046-6.

[20] A. Anand, S. Fosso Wamba y D. Gnanzou, "A Literature Review on Business Process Management, Business Process Reengineering and Business Process Innovation," en *9th International Workshop on Enterprise & Organizational Modeling and Simulation (EOMAS 2013)*, revisión sobre BPM, reingeniería de procesos y su relación con sistemas de información, Valencia, Spain, 2013, págs. 1-16.

[21] M. Watson, *Roles and Responsibilities of a Software Developer*, Online article, Describe cómo los desarrolladores traducen requerimientos de negocio en código, realizan pruebas, mantenimiento y asegura calidad, 2025. dirección: https://fullscale.io/blog/roles-and-responsibilities-of-a-software-developer/.

[22] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou y J. C. Carver, "Software Engineering Practices for Scientific Software Development: A Systematic Mapping Study," *arXiv preprint arXiv:2010.09914*, arXiv:2010.09914, 2020, estudio sobre

buenas prácticas de ingeniería de software — diseño, pruebas, mantenimiento — y su impacto en calidad y sostenibilidad del software. DOI: 10.48550/arXiv.2010.09914. dirección: https://arxiv.org/abs/2010.09914.

[23] C. Abid, V. Alizadeh, M. Kessentini, T. d. N. Ferreira y D. Dig, "30 Years of Software Refactoring Research: A Systematic Literature Review," *arXiv preprint arXiv:2007.02194*, arXiv:2007.02194v1, 2020, revisión sistemática sobre refactorización de código — práctica esencial para mantener calidad, legibilidad y robustez en sistemas de software complejos. DOI: 10.48550/arXiv.2007.02194. dirección: https://arxiv.org/abs/2007.02194.

[24] D. L. Lima, R. de Souza Santos, G. Pires Garcia, S. S. da Silva, C. Franca y L. F. Capretz, "Software Testing and Code Refactoring: A Survey with Practitioners," *arXiv preprint arXiv:2310.01719*, 2023, informe reciente sobre la práctica de pruebas automáticas y refactorización en entornos reales, destaca su importancia en mantenimiento y calidad del software. DOI: 10.48550/arXiv.2310.01719. dirección: https://arxiv.org/abs/2310.01719.

[25] T. Balić y H. Ebrahimi, "Automation and Digital Transformation," Master's thesis, University of Gothenburg, Gothenburg, Sweden, 2017. dirección: https://files.core.ac.uk/download/pdf/130026849.pdf.

[26] A. A. Atieh, A. A. Hussein, S. Al-Jaghoub, A. F. Alheet y M. Attiany, "The Impact of Digital Technology, Automation, and Data Integration on Supply Chain Performance: The Moderating Role of Digital Transformation," *Logistics*, vol. 9, n.º 1, 11, 2025, estudio empírico que evidencia que la automatización y la integración de datos mejoran el rendimiento organizacional cuando van acompañadas de iniciativas de transformación digital. DOI: 10.3390/logistics9010011.

[27] P. C. Verhoef, T. Broekhuizen, Y. Bart et al., "Digital transformation: A multidisciplinary reflection and research agenda," *Journal of Business Research*, vol. 122, págs. 889-901, 2021, revisión conceptual que distingue claramente los tres niveles: digitization, digitalization y digital transformation, y sus implicaciones organizacionales. DOI: 10.1016/j.jbusres.2019.09.022.

[28] M. J. Medina, A. D. Santo, P. Oswald y M. Sokhn, "Digital Business Transformation for SMEs: Maturity Model for Systematic Roadmap," en *Information Systems and Technologies*, ép. Lecture Notes in Networks and Systems, Á. Rocha, H. Adeli, L. P. Reis y S. Costanzo, eds., vol. 801, Springer, Cham, 2024, págs. 229-240. DOI:

10.1007/978-3-031-45648-0_23. dirección: https://doi.org/10.1007/978-3-031-45648-0_23.

[29] A. I. Ribas y otros, "Information Systems in Digital Transformation: Practical Case and Critical Success Factors," en *Proceedings of the International Conference on Industrial Engineering and Operations Management (IEOM) 2021*, análisis de factores críticos de éxito para implementar sistemas de información en proyectos de transformación digital, incluyendo integración, involucramiento de usuarios y alineamiento estratégico, Monterrey, Mexico, 2021, págs. 1075-1086. dirección: https://ieomsociety.org/proceedings/2021monterrey/183.pdf.

[30] J. M. González-Varona, A. López-Paredes, D. Poza y F. Acebes, "Building and development of an organizational competence for digital transformation in SMEs," *Journal of Industrial Engineering and Management*, vol. 14, n.º 1, págs. 15-24, 2024, modelo de competencias organizacionales para avanzar en madurez digital en pymes, destacando la importancia del desarrollo de software, integración y cultura organizacional. DOI: 10.3926/jiem.3279.

[31] E. J. Omol, "Organizational digital transformation: from evolution to future trends," *Emerging Markets Journal*, vol. 3, n.º 3, págs. 240-256, 2024, reflexión sobre cómo la transformación digital habilita a las organizaciones a adaptarse a entornos cambiantes mediante integración tecnológica, cambio cultural y rediseño de procesos, ISSN: 2755-0761. DOI: 10.1108/DTS-08-2023-0061.

[32] T. Gaddis, *Starting Out with Programming Logic and Design*, 6th. Boston, MA: Pearson, 2022, ISBN: 978-0137602148.

[33] L. Hackman, "Behind every good research there are data. What are they and their importance to forensic science," *Forensic Science International: Synergy*, vol. 8, 100456, 2024, discute cómo los datos son hechos u observaciones en bruto que requieren procesamiento para convertirse en información útil. DOI: 10.1016/j.fsisyn.2024.100456.

[34] J. Reich, "Data," *arXiv preprint arXiv:1801.04992*, 2017, presenta una definición formal de "dato" como un tipo de información tipada, útil para fundamentar conceptualmente datos vs información. DOI: 10.48550/arXiv.1801.04992. dirección: https://arxiv.org/abs/1801.04992.

[35] O. Dammann, "Data, Information, Evidence and Knowledge: A Proposal for Health Informatics and Data Science," *PMC*, vol. 10, n.º 3, e224, 2019, argumenta que la información se obtiene al contextualizar y procesar datos, transformando observaciones crudas en datos significativos para la toma de decisiones. dirección: https://pmc.ncbi.nlm.nih.gov/articles/PMC6435353/.

[36] C. Zins, "Conceptual approaches for defining data, information, and knowledge," *Journal of the American Society for Information Science and Technology*, vol. 58, n.º 4, págs. 479-493, 2007, revisión clásica que documenta decenas de definiciones disciplinares de datos, información y conocimiento, útil para fundamentar los conceptos en el campo de la ciencia de la información. DOI: 10.1002/asi.20508.

[37] G. Marchionini, "Information and data sciences: Context, units of analysis, meaning, and human impact," *Data and Information Management*, vol. 7, n.º 1, 100031, 2023, ISSN: 2543-9251. DOI: 10.1016/j.dim.2023.100031.

[38] B. Hjørland, "Data (with Big Data and Database Semantics)," *Knowledge Organization*, vol. 45, n.º 8, págs. 685-692, 2018, discusión crítica sobre la definición de "dato" en ciencias de la información y su rol como representación simbólica sin significado inherente. DOI: 10.5771/0943-7444-2018-8-685.

[39] V. B. Hans, "Data Representation and Abstraction in Computer Systems: An Interactive Study," *International Journal of Data Structure Studies*, vol. 2, n.º 2, págs. 22-31, 2024, explica cómo los datos se representan internamente en sistemas computacionales — bits, bytes, tipos de datos — y cómo la abstracción permite manipular datos sin exponer su representación física.

[40] Anonymous, "Data Representation in Computer Science," *ScienceDirect Topics (encyclopedic entry)*, 2025, descripción general de cómo los sistemas informáticos codifican, almacenan y manipulan datos como secuencias binarias, tipos de datos, estructuras de datos, etc. dirección: https://www.sciencedirect.com/topics/computer-science/data-representation.

[41] L. Floridi, "From Data to Semantic Information," *Entropy*, vol. 5, n.º 2, págs. 125-140, 2003, presenta un marco filosófico-científico clásico sobre cómo los datos se transforman en información mediante semántica, contexto y procesamiento. DOI: 10.3390/e5020125.

[42] R. Gellert, "Comparing Definitions of Data and Information in Information Science," *Journal of the Association for Information Science and Technology*, vol. 73, n.º 5, págs. 657-672, 2022, análisis empírico de cómo distintas disciplinas definen "dato" e "información", útil para fundamentar conceptualmente la distinción data–information. DOI: 10.1002/asi.24724.

[43] S. Baškarada y A. Koronios, "Data, Information, Knowledge, Wisdom (DIKW): A Semiotic Theoretical and Empirical Exploration of the Hierarchy and Its Quality Dimension," *Australasian Journal of Information Systems*, vol. 18, n.º 1, pág. 5, 2013, artículo influyente que analiza definiciones de dato, información y conocimiento, y examina cómo los datos se transforman en información significativa. DOI: 10.3127/ajis.v18i1.748.

[44] S. Calzati, "An Ecosystemic View on Information, Data, and Knowledge: Insights on Agential AI and Relational Ethics," *AI and Ethics*, vol. 5, págs. 3763-3776, 2025. DOI: 10.1007/s43681-025-00665-0.

[45] E. Horvitz y T. Mitchell, "From Data to Knowledge to Action: A Global Enabler for the 21st Century," *arXiv preprint arXiv:2008.00045*, 2020, documenta cómo los datos masivos, correctamente procesados y analizados, permiten generar información útil para la toma de decisiones en diferentes dominios, desde la ciencia hasta políticas públicas.

[46] M. Frické, "The Knowledge Pyramid: the DIKW Hierarchy," *Knowledge Organization*, vol. 46, n.º 1, págs. 33-47, 2019, revisión crítica del modelo DIKW y de la distinción entre datos e información en ciencias de la información. DOI: 10.5771/0943-7444-2019-1-33.

[47] T. Silwattananusarn y K. Tuamsuk, "Data Mining and Its Applications for Knowledge Management: A Literature Review from 2007 to 2012," *arXiv preprint arXiv:1210.2872*, 2012, revisión sobre cómo técnicas de minería de datos transforman datos en información y conocimiento valiosos para organizaciones.

[48] K. D. Cato, K. McGrow y S. C. Rossetti, "Transforming Clinical Data Into Wisdom: Artificial Intelligence Implications for Nurse Leaders," *Nursing Management (Springhouse)*, vol. 51, n.º 11, págs. 24-30, 2020. DOI: 10.1097/01.NUMA.0000719396.83518.d6.

[49] D. I. Spivak y R. E. Kent, "Ologs: a categorical framework for knowledge representation," *arXiv preprint arXiv:1102.1889*, 2011, propone un marco formal (teoría de categorías) para representación y estructuración de conocimiento en sistemas computacionales, útil cuando la información necesita transformarse en conocimiento manipulable y fiable. dirección: https://arxiv.org/abs/1102.1889.

[50] J. H. Bernstein, "The Data–Information–Knowledge–Wisdom Hierarchy and Its Antithesis," *Journal of Management and Information Systems*, vol. 26, n.º 2, págs. 187-220, 2009, análisis crítico del modelo tradicional DIKW, discutiendo sus límites y proporcionando una visión más realista sobre cómo la información puede (o no) transformarse en conocimiento. DOI: 10.2753/MIS0742-1222260206.

[51] D. Williams y otros, "Models, metaphors and symbols for information and knowledge systems," *Journal of Enterprise Information Management*, vol. 27, n.º 6, págs. 796-815, 2014, critica las representaciones simplistas del conocimiento (como la pirámide DIKW), argumentando la necesidad de modelos más complejos que reflejen la naturaleza dinámica del conocimiento en sistemas de información. DOI: 10.1108/JEIM-03-2014-0026.

[52] M. Bhatt, *Computer Fundamental. UNIT: 1 Introduction to Computer*, Apuntes de curso, preparados por Meghna Bhatt. Consultado el 1 de diciembre de 2025. dirección: https://ashishmodi.weebly.com/uploads/1/8/9/7/18970467/computer_fundamental.pdf.

[53] W. Stallings, *Computer Organization and Architecture: Designing for Performance*, 11.ª ed. London: Pearson, 2021, pág. 896, ISBN: 978-1292420103.

[54] A. S. Tanenbaum y T. Austin, *Structured Computer Organization*, 6.ª ed. Boston, MA: Pearson, 2021, pág. 777, ISBN: 978-0137618446.

[55] A. Silberschatz, P. B. Galvin y G. Gagne, *Operating System Concepts*, 10.ª ed. Hoboken, NJ: Wiley, 2018, pág. 1040, ISBN: 978-1119320913.

[56] A. S. Tanenbaum y H. Bos, *Modern Operating Systems*, 5.ª ed. Hoboken, NJ: Pearson, 2024, pág. 1879, ISBN: 978-9361595479.

[57] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8.ª ed. McGraw-Hill Education, 2018.

[58] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein, *Introduction to Algorithms*, 4.ª ed. Cambridge, MA: MIT Press, 2022, ISBN: 978-0262046305.

[59] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen y K. A. Houston, *Object-Oriented Analysis and Design with Applications*, 3.ª ed. Addison-Wesley, 2007, pág. 720, ISBN: 978-0201895513.

[60] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2 update. Addison-Wesley, 2012, ISBN: 978-0133002140.

[61] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3.ª ed. Addison-Wesley, 1997, pág. 650, ISBN: 978-0201896831.

[62] R. W. Sebesta, *Concepts of Programming Languages*, 12.ª ed. Pearson, 2018, Update 2018, ISBN: 978-0134997186.

[63] J. Farrell, *Programming Logic and Design*, 9th. Boston, MA: Cengage Learning, 2018, ISBN: 9781337102070.

[64] J. M. Wing, "Computational Thinking," *Communications of the ACM*, vol. 49, n.º 3, págs. 33-35, 2006. DOI: 10.1145/1118178.1118215.

[65] R. Sedgewick y K. Wayne, *Algorithms*, 4.ª ed. Addison-Wesley, 2011, pág. 976, ISBN: 978-0321573513.

[66] S. Dasgupta, C. Papadimitriou y U. Vazirani, *Algorithms*. McGraw-Hill, 2023, pág. 336, ISBN: 978-9355325525.

[67] A. Dennis, B. H. Wixom y D. Tegarden, *Systems Analysis and Design: An Object-Oriented Approach with UML*, 7.ª ed. Wiley, 2025, ISBN: 978-1394331765.

[68] J. Farrell, *Programming Logic and Design*, 8.ª ed. Cengage Learning, 2015, pág. 704, ISBN: 978-1285776712.

[69] C. Ghezzi, M. Jazayeri y D. Mandrioli, *Fundamentals of Software Engineering*, 2.ª ed. Prentice Hall, 2003, ISBN: 978-0133056990.

[70] R. Fitzpatrick, "Software quality revisited," English, en *Proceedings of the Software Measurement European Forum (SMEF)*, ép. Proceedings of the Software Measurement European Forum (SMEF), Milan, Italy: Instituto di Ricerca Internazionale, 2004, págs. 305-315. DOI: 10.21427/e2gf-0035.

[71] M. Li, W. Fang, Q. Zhang y Z. Xie, "SpecLLM: Exploring Generation and Review of VLSI Design Specification with Large Language Model," en *2025 International Symposium of Electronics Design Automation (ISEDA)*, 2025, págs. 749-755. DOI: 10.1109/iseda65950.2025.11100410.

[72] L. Perković, A. Settle, S. Hwang y J. Jones, "A framework for computational thinking across the curriculum," en *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ép. ITiCSE '10, Bilkent, Ankara, Turkey: Association for Computing Machinery, 2010, págs. 123-127, ISBN: 978-1605588209. DOI: 10.1145/1822090.1822126.

[73] D. Duckworth y J. Fraillon, "Computational Thinking Framework," *Educational Research Review*, vol. 22, J. Fraillon y M. Rožman, eds., págs. 35-43, 2025. DOI: 10.1007/978-3-031-61194-0_3.

[74] B. Kernighan y D. Ritchie, *The C Programming Language*, 2.ª ed. Pearson/Prentice Hall, 1988, pág. 272, ISBN: 978-0131103627.

[75] N. Wirth, *Algorithms + Data Structures = Programs*, 1.ª ed. Prentice Hall, 1976, pág. 366, Última Actualización Moscow, 2 de febrero 2012-02-22 por Fyodor Tkachov, ISBN: 978-0130224187.

[76] A. Dix, J. Finlay, G. Abowd y R. Beale, *Human-Computer Interaction*, 3.ª ed. Pearson/Prentice-Hall, 2004, pág. 834, ISBN: 978-0130461094.

[77] J. L. Manzano, *Programación: Algoritmos y estructuras de datos*. Paraninfo, 2017.

[78] SmartDraw. "Símbolos de diagramas de flujo." Consultado en diciembre de 2025. (2024), dirección: https://www.smartdraw.com/flowchart/simbolos-de-diagramas-de-flujo.htm.

[79] K. E. Kendall y J. E. Kendall, *Systems Analysis and Design*, 11.ª ed. Pearson, 2024, ISBN: 9780137947850.