

Capítulo 2

Algoritmos

Resultado de aprendizaje

Diseña algoritmos para la resolución de problemas.

Indicador de logro

Utiliza estructuras de control para resolver problemas a través de algoritmos.

2.1. Importancia de la algoritmia en la ingeniería

El análisis de problemas y el diseño de soluciones a través de herramientas y sistemas computacionales trasciende la simple aplicación de un desarrollo de software, sino más bien se constituye, como una disciplina fundamental de la Ciencias de la Computación, en el área informática, que maximiza la capacidad de las computadoras para resolver desafíos complejos. Esta aproximación sistemática y eficiente se logra mediante la aplicación del pensamiento computacional (*Computational Thinking*), una habilidad cognitiva que se ha consolidado como un paradigma esencial en la educación y la práctica profesional en la actualidad [Kolczyk, 2008].

Para materializar estas soluciones en un artefacto tangible, es imprescindible el uso de un lenguaje algorítmico que sea preciso, general y eficiente. Es así como, *la algoritmia* (o *Pensamiento Algorítmico*) es la habilidad de formular problemas de manera que se transformen de una entrada a una salida deseada a través de una secuencia estricta de operaciones, tareas, procesos o secuencia de pasos. De esta manera, *los algoritmos* llegan a ser un concepto fundamental en la informática y representan estrategias formales para resolver problemas específicos a través de una secuencia de pasos bien definidos [Nazareth, 2023].

Más allá de la programación, se trata de un proceso cognitivo que involucra, *descomposición* de problemas complejos en subproblemas manejables, *abstracción* para centrarse en los detalles relevantes y, *evaluación y optimización* de las soluciones. Este enfoque metódico permite plantear las posibles soluciones a cualquier problema, independientemente del campo disciplinar. Su implementación se ha estudiado en áreas no tradicionales como la formación contable, en medicina, o en biología, demostrando que facilita la comprensión y solución procedimental de ejercicios



complejos a través de herramientas informáticas y su debida estructuración en un currículo formal [Llerena-Izquierdo, 2023].

Hoy en día, la necesidad de poseer conocimientos y dominio de la informática es una exigencia transversal para todo profesional, debido a la necesidad de elevar su potencial competitivo en un mercado laboral en aumento enfocado a la digitalización. Para la Ingeniería, el impacto de la informática es trascendental. Se puede evidenciar que ningún otro aspecto de la tecnología tiene un mayor impacto potencial en la ingeniería que la informática [Möller & Collignon, 2023]. Las bases de datos indexadas en áreas de Ciencias Naturales e Ingeniería, como Scopus y WoS, reflejan el volumen de producción científica en esta intersección, confirmando que temas como las ciencias computacionales aplicadas son consistentemente algunos de los más citados en la categoría [Yutong & Jin, 2024].

Los ingenieros no solo utilizan la programación y la simulación, sino que aplican el pensamiento computacional para: *Diseño Asistido por Computadora* (CAD) y *Análisis de Ingeniería Asistido por Computadora* (CAE), el *Modelado y simulación de sistemas complejos* (mecánica, resistencia de materiales, electricidad) para analizar, predecir y optimizar su comportamiento y, para la *implementación de sistemas avanzados como la Inteligencia Artificial (IA) y la Computación en la nube* para enfrentar los desafíos de la Industria 4.0, lo que permite el diseño y el desarrollo de soluciones guiadas y estandarizadas. La habilidad para concebir un plan de trabajo o de solución a un problema, que incluye el análisis y el diseño asistido por computadora de sistemas de producción, es una función directa del dominio algorítmico y el pensamiento computacional [Landman & Kohn, 2024]. Este dominio es lo que permite al ingeniero actual utilizar la herramienta, y también diseñar y automatizar el proceso de solución fundamental.



El avance de la informática ha ampliado exponencialmente el alcance de este bien generado por los primeros investigadores. Hoy, los problemas de ingeniería, medicina, educación o economía se formulan a través de estructuras algorítmicas que permiten simular, analizar y predecir resultados complejos. Estudios recientes en educación en ingeniería demuestran que integrar herramientas computacionales, como Python o MATLAB, mejora de manera significativa la capacidad de análisis y la comprensión conceptual de los estudiantes [Cuevas et al., 2025]. Esto reafirma que la programación es una destreza técnica, y un medio de estructurar el pensamiento, de modelar la realidad con precisión y creatividad [Llerena-Izquierdo et al., 2024b].

Desde los inicios de la historia de la ciencia, la humanidad ha buscado sistematizar los procesos de razonamiento que conducen a la resolución de problemas. Uno de los primeros en lograrlo fue Muhammad ibn Musa al-Khwārizmī, erudito del siglo IX que trabajó en la Casa de la Sabiduría de Bagdad. En su obra *Al-kitāb al-mukhtaṣar fī ḥisāb al-jabr wa'l-muqābala*, Al-Khwārizmī estableció los principios del álgebra, e introdujo una forma de pensar que consistía en descomponer un problema en pasos lógicos y secuenciales para alcanzar una solución. Este método, cuyo nombre derivó en la palabra “*algoritmo*”, se convirtió con el paso del tiempo en el corazón de las Ciencias de la Computación [Sun et al., 2024a]. La esencia de la contribución de Al-Khwārizmī consiste en haber concebido el conocimiento como un proceso de traducción del pensamiento lógico al lenguaje simbólico, algo que hoy se refleja en los lenguajes de programación [Kydyraliev

et al., 2025]. En la actualidad, el acto de programar no es más que la expresión moderna de esa antigua idea: *representar el pensamiento humano de manera precisa, formal y reproducible por una máquina de cómputo* [Fauziah, 2025].

La obra de Al-Khwārizmī, reinterpretada a la luz de la ciencia de la computación contemporánea, nos invita a ver la programación no como una habilidad aislada, sino como una forma de pensamiento universal [Roelens, 2025]. Los algoritmos, lejos de ser simples secuencias de instrucciones, son manifestaciones de una inteligencia estructurada que guía la resolución de problemas en cualquier ámbito del conocimiento [Yıldız, 2025]. Comprenderlos, crearlos y aplicarlos constituye hoy una de las tareas más nobles de la educación moderna: *preparar a los estudiantes para pensar en construir un futuro distinto, como lo hizo Al-Khwārizmī, con rigor lógico, curiosidad científica y sentido humanista*, ver Fig. 2.1.

Representación artificial



Figura 2.1: Imagen generada por la IA Gemini de Muhammad ibn Musa al-Khwārizmī y los algoritmos.

En este contexto, el algoritmo se convierte en el punto de encuentro entre la teoría y la práctica, entre la abstracción y la acción. Las investigaciones actuales coinciden en que el pensamiento algorítmico permite trascender el ámbito técnico para convertirse en una forma de razonamiento aplicable a cualquier disciplina [Sakdiyah et al., 2025].

Programar implica una forma de conocimiento que combina intuición, lógica y estética; una expresión moderna de la filosofía que Al-Khwārizmī anticipó cuando unió las matemáticas con el razonamiento sistemático [Karandikar & Warnekar, 2025].

Otro personaje importante es Augusta Ada King, Condesa de Lovelace (1815-1852), matemática británica e hija del célebre poeta Lord Byron, ella ocupa un lugar singular en la historia de la ciencia computacional como la primera persona en conceptualizar y documentar un algoritmo destinado a ser ejecutado por una máquina [Terroso Sáenz, 2025; Cavendish, 2015]. Su contribución trasciende la simple programación técnica, estableciendo las bases conceptuales del pensamiento algorítmico moderno y anticipando el potencial de las computadoras más allá del cálculo aritmético básico [Grzybowski et al., 2024; Rafiq, 2024], ver Fig. 2.2.

Representación artificial



Figura 2.2: Imagen generada por la IA Gemini de Ada Lovelace y los algoritmos.

Ada Lovelace¹ colaboró estrechamente con Charles Babbage en el desarrollo conceptual de la *Máquina Analítica*, un prototipo de computadora digital pero mecánica y diseñada en la década de 1840 [Martin, 2015; Charman-Anderson, 2020]. Este dispositivo, propulsado por vapor y programable mediante tarjetas perforadas, nunca se construyó completamente durante la vida de Babbage, pero representaba un salto conceptual extraordinario respecto a las calculadoras mecánicas de la época [Callaghan et al., 2020]. La *Máquina Analítica* incorporaba elementos fundamentales de la arquitectura computacional moderna: una *unidad de procesamiento aritmético* (el “Molino”), *memoria para almacenar datos* y resultados intermedios (el “Almacén”), y un *sistema de programación* mediante tarjetas perforadas que especificaban las operaciones determinadas en relación con las direcciones de memoria existentes [Haugtvedt & Abata, 2021; Ferry, 2015; Newton &

¹Biografía de Ada Lovelace

[Maas, 2025]. En sus notas, Ada Lovelace escribió lo que hoy se reconoce como el primer algoritmo destinado a ser procesado por una máquina, lo que la convierte en la primera programadora de la historia [Callaghan et al., 2020; Abbate et al., 2023]. Lo más notable de su trabajo, mucho más que la codificación de instrucciones, fue su visión conceptual de lo que las máquinas pueden llegar a lograr, en aquel presente y en el futuro [Rafiq, 2024]. Ella comprendió que los dispositivos computacionales (o de cómputo o de cálculo) no estaban limitados a cálculos aritméticos, ella tuvo la visión de que estos podían manipular símbolos formales de acuerdo con reglas, anticipando así el concepto de programación general [Charman-Anderson, 2015; Angluin, 2016].

El algoritmo conocido como “*La Nota G*”, fue escrito por Ada Lovelace, y contiene un conjunto de pasos diseñados para calcular los números de Bernoulli² utilizando la hipotética *Máquina Analítica*, y es generalmente reconocida como el primer algoritmo específicamente diseñado para una computadora [Charman-Anderson, 2015; Lovelace, 2015]. Este algoritmo apareció como parte de una serie de notas etiquetadas de la letra A, a la letra G que Lovelace anexó a su traducción al inglés del artículo de Luigi Menabrea sobre la conferencia de Babbage en la Universidad de Turín (1842) [Haugtvedt & Abata, 2021]. Los números de Bernoulli son una secuencia de números racionales con aplicaciones profundas en teoría de números, análisis matemático y combinatoria. El algoritmo de Lovelace para calcular estos números fue revolucionario porque demostró que una máquina podía realizar tareas que iban más allá de la aritmética simple, mostrando el potencial de la computación automatizada [Rice, 2024; Haugtvedt & Abata, 2021; Hénin, 2015].

Ada Lovelace imaginó una visión revolucionaria sobre si la información podía reducirse a símbolos y reglas de manipulación, entonces una máquina podría procesar cualquier tipo de información simbólica, no solo números [Haugtvedt & Abata, 2021]. Esta intuición anticipó conceptos fundamentales de la ciencia de la computación moderna, incluyendo, la computación a base de símbolos. Lovelace consideró que la *Máquina Analítica* podría manipular símbolos de acuerdo con reglas, abriendo la posibilidad de que las máquinas pudieran componer música o producir arte si los elementos del arte pudieran formalizarse adecuadamente [Gonçalves, 2024]. Otro concepto que intuyó fue de “*Programación como disciplina creativa*”, Lovelace identificó que la programación requería habilidades analíticas y creativas, estableciendo la programación como una forma de arte intelectual [Callaghan et al., 2020]. Finalmente, identificó los “*Límites de la computación mecánica*”, contrariamente, Lovelace también anticipó debates contemporáneos sobre inteligencia artificial al afirmar (1843) que la *Máquina Analítica* “*no tiene pretensiones de originar nada. Puede hacer cualquier cosa que sepamos ordenarle que realice*”, planteando tempranamente la cuestión de si las máquinas pueden exhibir creatividad genuina [Blackburn, 2024; Oliveira & Figueiredo, 2024].

Además, su trabajo ha sido una fuente de inspiración para generaciones de científicas y científicos, al demostrar que la programación no es solo una cuestión técnica, sino también una forma de pensamiento creativo [Callaghan et al., 2020; Case & Leggett, 2016]. Sus palabras resaltan la importancia del diseño algorítmico como puente entre la lógica humana y la ejecución mecánica [Füegi & Francis, 2015]. Después de su muerte prematura (36 años), las contribuciones de Ada Lovelace permanecieron en relativa obscuridad [Ferlier, 2025]. Su redescubrimiento en el siglo XX, particularmente tras la publicación de las biografías de Doris Langley Moore (1977) y Betty Alexandra Toole (1992), estableció su posición canónica en la historia de la computación [Haugtvedt & Abata, 2021]. El “*Lenguaje de programación Ada*”, desarrollado por el Departamento de



²Número de Bernoulli - Wikipedia

Defensa de Estados Unidos entre 1977 y 1983, para sistemas críticos, fue nombrado en su honor, reconociendo su contribución pionera, su búsqueda de la precisión y la fiabilidad [Füegi & Francis, 2015; Kyriazis, 2025]. Adicionalmente, el segundo martes de octubre³ se celebra internacionalmente el “Ada Lovelace Day”, dedicado a reconocer los logros de las mujeres en ciencia, tecnología, ingeniería y matemáticas (STEM) [Tamboukou, 2023]. Iniciativas como el Ada Lovelace Day celebran su contribución y promueven la participación de las mujeres en STEM (ciencia, tecnología, ingeniería y matemáticas) [Strawn, 2023; Hollings et al., 2017b].

Lovelace también se adentró en un debate filosófico que sigue vigente hoy: la posibilidad de que la inteligencia artificial posea conciencia [Gonçalves, 2024]. Su afirmación más citada, conocida como la “*Objeción de Lady Lovelace*”, señala una observación, sobre cómo las máquinas reciben órdenes, y que no era una limitación de la máquina, sino una delimitación clara entre la ejecución de instrucciones (el algoritmo) y la capacidad de tener intencionalidad y creatividad autónomas [Rafiq, 2024]. Este argumento estableció, por primera vez, la distinción fundamental entre el software (las instrucciones) y el hardware (la máquina), y continúa siendo un pilar en la ética y filosofía de la IA [Charman-Anderson, 2020; Llerena-Izquierdo & Ayala-Carabajo, 2025a]. La contribución de Ada Lovelace ha cobrado nueva relevancia en la era de la inteligencia artificial [Grzybowski et al., 2024]. Su intuición sobre el poder simbólico de las máquinas se refleja en tecnologías actuales como los modelos de lenguaje, los algoritmos de aprendizaje automático y la programación orientada a objetos [Cavendish, 2015].

En consecuencia, hablar de Ada Lovelace no es solo evocar a la primera programadora, sino reflexionar sobre la dimensión humana del algoritmo [Haugtvedt & Abata, 2021]. Ella comprendió que programar es un acto de traducción entre dos mundos: el de las ideas y el de la materia, el de la mente y el de la máquina [Callaghan et al., 2020]. En esa intersección, el algoritmo deja de ser una simple secuencia de pasos y se convierte en un lenguaje del pensamiento, una manifestación simbólica de la capacidad humana para imaginar sistemas, modelar realidades y transformar el conocimiento en acción con ética [Natale & Henrickson, 2024; Llerena Izquierdo & Ayala Carabajo, 2025]. El impacto de Ada Lovelace trasciende la historia de la computación: es la raíz de una nueva forma de razonar [Haugtvedt & Abata, 2021]. Su trabajo realizado vive en cada código escrito, en cada modelo de inteligencia artificial, y en toda aproximación científica que traduzca la complejidad del mundo en estructuras lógicas comprensibles. Lovelace imaginó el primer algoritmo e imaginó el futuro [Avila et al., 2024]. El trabajo de Ada Lovelace es inconmensurable [Haugtvedt & Abata, 2021]. Al ser reconocida como la primera programadora de la historia, su pensamiento dispuso las bases para conceptualizar el computador y abrir su significado a la computación como ciencia. Su visión conecta directamente con los trabajos de Alan Turing, quien un siglo después retomaría y debatiría sus ideas sobre la inteligencia de las máquinas [Schlombs, 2021].

Hoy, su nombre es sinónimo de innovación en tecnología [Strawn, 2023]. Su figura inspira a nuevas generaciones de científicas, ingenieras y programadoras, recordándonos que la tecnología no es solo una cuestión de ingeniería, sino también de imaginación [Hollings et al., 2017a].

Ada Lovelace estableció fundamentos conceptuales de la algoritmia computacional un siglo antes de que existieran máquinas capaces de ejecutar sus ideas. Su visión trascendió las limitaciones tecnológicas de su época, anticipando el papel transformador que la computación ejercería en la civilización moderna [Ferry, 2015]. Más allá del algoritmo específico que desarrolló, su legado radica en demostrar que las máquinas podrían procesar información simbólica de formas generales, sentando las bases filosóficas de la revolución digital [Newton & Maas, 2025]. Su historia nos recuerda que los avances tecnológicos fundamentales frecuentemente provienen de saltos conceptuales y visión imaginativa tanto como de ingenio técnico, y que la diversidad de perspectivas enriquece fundamentalmente la innovación científica y tecnológica, de alguien que realmente valora, aprecia y se motiva por la ciencia [Charman-Anderson, 2020].

³Ada Lovelace Day

A lo largo de la historia de la computación, varias mujeres han dejado una huella imborrable en el desarrollo de la programación, la algoritmia y la tecnología digital. Entre ellas, **Grace Hopper**, **Hedy Lamarr**, **Margaret Hamilton** y **Katie Bouman**, quienes representan diferentes generaciones que han transformado, desde sus contextos, la forma en que concebimos la relación entre el pensamiento lógico y las máquinas.



Figura 2.3: Grace Murray Hopper ante el teclado UNIVAC I, hacia 1960. completaba su doctorado⁵.

Durante la Segunda Guerra Mundial, se unió a la Reserva Naval (1943) y fue asignada al Proyecto de Computación de la Oficina de Artillería en Harvard, donde trabajó en el equipo que desarrolló la máquina analítica Harvard Mark I, una de las primeras computadoras electromecánicas [Aiken & Hopper, 1946] (ver Fig. 2.3).

En 1949, Hopper se unió a Eckert-Mauchly Computer Corporation, donde diseñó uno de los primeros compiladores, que traducían las instrucciones de un programador en códigos de computadora, y acuñó el término “*compilador*”. Grace Hopper es reconocida como una de las fundadoras de la programación moderna.



Figura 2.5: “*Atrévete a intentarlo. Atrévete a hacer las cosas de forma diferente. Atrévete a ser audaz y simplemente observa qué sucede*”, Grace Hooper

va se convierte en una herramienta que traduce el lenguaje humano en lenguaje máquina, lo que sentó las bases de los lenguajes de alto nivel, democratizando la programación al abstraer la com-

Grace Brewster Murray Hopper⁴ (1906-1992). La Precursora de los Lenguajes de Programación [Candler, 2020]. Nació el 9 de diciembre de 1906 en Nueva York y falleció el 1 de enero de 1992 en Arlington, Virginia (Wikipedia). Se graduó en Vassar College en 1928 con una licenciatura en Matemáticas y Física con honores, obtuvo su maestría en matemáticas de Yale en 1930 y su doctorado en 1934 donde se especializó en matemáticas relativas y física cuántica, simultáneamente, fue profesora de matemáticas en Vassar, mientras com-



Figura 2.4: Grace Hopper desarrolló en 1959, el lenguaje COBOL.

Su contribución más destacada a la algoritmia fue el desarrollo del primer compilador informático, el sistema A-0 en 1952 (el primer compilador para lenguajes informáticos) [McDevitt & McGettigan, 2024], que traducía instrucciones simbólicas matemáticas a código binario legible por máquinas (ver Fig. 2.4).

Esta aportación significati-

⁴Biografía (1) Grace Hopper.

⁵Biografía (2) Grace Hopper.

plejidad del hardware y haciendo que el software fuera accesible más allá de los expertos en ensamblador [Candler, 2020] (ver Fig. 2.5).

Este avance condujo directamente al diseño del lenguaje COBOL -*lenguaje común orientado a negocios*- (1959) [Candler, 2020], que permitió que personas (en las empresas) sin formación altamente especializada, pudieran programar computadoras y escribir programas más comprensibles y eficientes⁶. Como programadora principal, trabajó en el diseño del UNIVAC I y desarrolló aplicaciones navales para COBOL [Hopper, 1952].

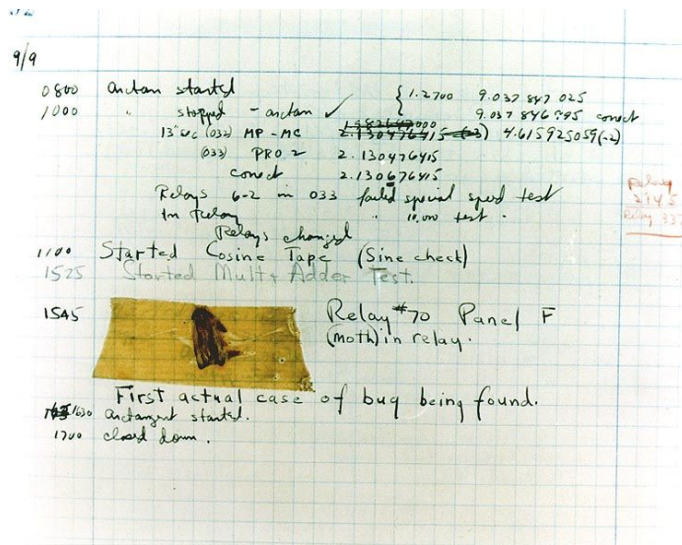


Figura 2.6: Primer caso real de error detectado en un equipo de cómputo.

software fuera escrito en un lenguaje legible por humanos, revolucionando la programación automática. Una de las anécdotas más famosas de la historia de la computación involucra a Hopper. Aunque no acuñó realmente el término, ayudó a popularizarlo cuando una polilla fue encontrada en el Mark II, causando un cortocircuito [Hopper, 1952].

Mientras depuraba la Harvard Mark II, su equipo descubrió una polilla atascada en un relé, causando un mal funcionamiento. Hopper la extrajo y la pegó en el libro de registro con la nota “Primer caso real de un bug encontrado”, acuñando así el término “debugging” (depuración), que aún usamos hoy, para referirse a la corrección de errores en los programas (ver Fig. 2.6). Este incidente dio origen al uso común del término “bug” (error o bicho) en programación. En 1991, el presidente George Bush le otorgó la Medalla Nacional de Tecnología por sus logros pioneros en el desarrollo de lenguajes de programación; fue la primera mujer en recibir individualmente el premio tecnológico más alto de

Este lenguaje permitió que personas sin formación altamente especializada pudieran programar equipos [Aiken & Hopper, 1946]. En 1966, fue llamada nuevamente al servicio activo para ayudar a estandarizar los múltiples lenguajes y programas informáticos de la Marina. Así, Grace Hopper, es conocida como la “Abuela de COBOL” (uno de los primeros lenguajes de alto nivel, 1959) y como la mujer que inventó el primer compilador (A-0 System, 1952) [McDevitt & McGettigan, 2024] y el primer lenguaje de programación que utilizaba comandos similares al inglés (Flow-Matic, 1955 a 1959) [Candler, 2020]. Su visión fue fundamental para que el

Figura 2.7: Sintaxis del lenguaje COBOL.

⁶Biografía (3) Grace Hopper

la nación Yale. En 2016, recibió póstumamente la Medalla Presidencial de la Libertad, el más alto honor civil. Hopper también fue una promotora incansable de la educación en informática, abogando por acercar la programación a un público más amplio. Hoy se la recuerda no solo por su genio técnico, sino por su visión democratizadora del conocimiento computacional. Su legado perdura a través del lenguaje COBOL (aún en uso, ver Fig. 2.7), es recordada con numerosos honores y como almirante de la Marina (retirada en 1986) [McDevitt & McGettigan, 2024], incluyendo ser la primera mujer comodoro de la Marina de los EE. UU. y el destructor USS Hopper (DDG-70) que lleva su nombre, además de su inducción póstuma al Salón de la Fama de Inventores. Inspira generaciones a través de la *Grace Hopper Celebration of Women in Computing*, un evento anual que fomenta la diversidad en STEM. Se retiró de la Marina como contraalmirante a los 79 años, siendo la oficial en servicio de mayor edad en las fuerzas armadas estadounidenses [Hopper, 1952]. Fue enterrada con todos los honores militares en el Cementerio Nacional de Arlington [Beyer, 2012].

Hedwig Eva Maria Kiesler, conocida como Hedy Lamarr⁷, nació en Viena, Austria, el 9 de noviembre de 1914 y falleció en Casselberry, Florida, EE.UU. el 19 de enero de 2000 [Rosengren, 2022] (ver Fig. 2.8). Aunque más conocida como actriz de Hollywood, Hedy Lamarr fue también una inventora extraordinaria [Markstrom, 2020]. Famosa artista de cine, tuvo una formación en ingeniería y fue una autodidacta con una mente extremadamente inquisitiva. Aunque su carrera principal fue el cine, su interés por la invención era profundo. Lamarr⁸ fue hija única de una familia judía acomodada. Su padre era director de un banco y su madre pianista de concierto. Su padre la llevaba a largos paseos donde discutían el funcionamiento interno de diferentes máquinas, como la prensa de impresión o los tranvías, lo que guió su pensamiento. Lamarr abandonó la escuela a los 15 años, para perseguir la actuación, debutando en el controvertido filme *Ecstasy* (1933), aunque su educación informal y autodidacta fue profunda [Griffiths & Farina, 2021].



Figura 2.8: Hedy Lamarr.

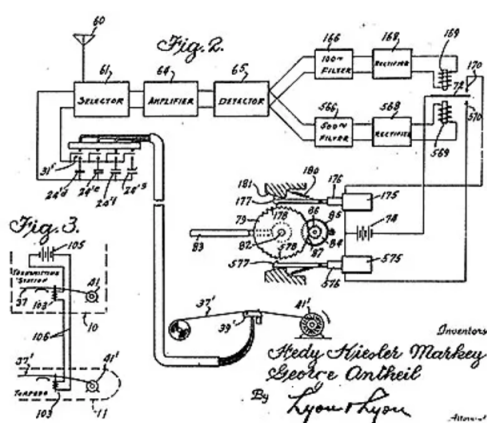


Figura 2.9: Salto de frecuencia de espectro ensanchado.

Tras una breve carrera cinematográfica en Checoslovaquia, huyó de su esposo, un rico fabricante austríaco de municiones, pero este episodio le dio la base de su conocimiento sobre sistemas de armas, incluidos sistemas de control de torpedos. Se trasladó secretamente a París. Viajó a Londres, donde conoció a Louis B. Mayer, quien le ofreció un contrato cinematográfico en Hollywood [Aguayo-Torres et al., 2024].

Su conocimiento técnico fue autodidacta, derivado de lecturas voraces en mecánica e ingeniería mientras filmaba en Hollywood. Lamarr, junto con el compositor George Antheil, inventó el sistema de “Salto de Frecuencia” (*Spread Spectrum*), una técnica que hoy se menciona como un desarrollo importante en el campo de las comunicaciones inalámbricas, desarrollado en 1941 [Burkland,

⁷Biografía (1) Hedy Lamarr

⁸Biografía (2) Hedy Lamarr

2021]. Este no es un algoritmo de programación como tal, sino un principio de comunicaciones que permite que la señal salte entre frecuencias de radio. Esta técnica es el fundamento teórico de la tecnología de las comunicaciones inalámbricas modernas, como el Wi-Fi, el Bluetooth y el GPS [Griffiths & Farina, 2021] (ver Fig. 2.9).



Figura 2.10: Madre del Wi-Fi.

de una manera aparentemente aleatoria que sonaría como mero ruido para cualquiera que escuchara. Este sistema fue diseñado para hacer que los torpedos guiados por radio fueran más difíciles de detectar o interferir [Griffiths & Farina, 2021].

Una anécdota emblemática revela su doble vida: durante una fiesta en su mansión de los años 40, mientras charlaba con aviadores y científicos, esbozó su idea de FHSS en una servilleta, inspirada en el piano de 88 teclas de Antheil⁹. Sin embargo, la Marina de EE.UU. rechazó la patente, argumentando que una “estrella de cine glamorosa” no podía ser seria inventora [Aguayo-Torres et al., 2024] (ver Fig. 2.10).



Figura 2.11: Su genio inventivo.

Durante sus sesiones de lluvia de ideas nocturnas, Hedy y Antheil jugaban un juego musical [Markstrom, 2020]. Se sentaban al piano juntos, una persona comenzaba a tocar una canción popular y la otra veía qué tan rápido podía reconocerla y comenzar a tocar junto. Fue aquí donde Richard Rhodes escribió el libro titulado *Hedy's Folly: The Life and Breakthrough Inventions of Hedy Lamarr, the Most Beautiful Woman in the World*, (La locura de Hedy: La vida y las invenciones revolucionarias de Hedy Lamarr, la mujer más bella del mundo), publicado en 2011, piensa que Hedy y Antheil concibieron por primera vez la idea del salto de frecuencia. Si dos músicos tocan la misma música, pueden saltar por el teclado juntos en perfecta sincronía. Sin embargo, si alguien escuchando no conoce la canción, no tiene idea de qué teclas se presionarán a continuación [Aguayo-Torres et al., 2024] (ver Fig. 2.11).

Lamarr no tuvo formación formal en ingeniería o programación, pero su capacidad para pen-

⁹Biografía (3) Hedy Lamarr

sar algorítmicamente —previendo secuencias de señales codificadas— refleja un entendimiento intuitivo de la lógica de la comunicación digital. Su vida combina el arte y la ciencia, desafiando los estereotipos de género de su época [Griffiths & Farina, 2021].

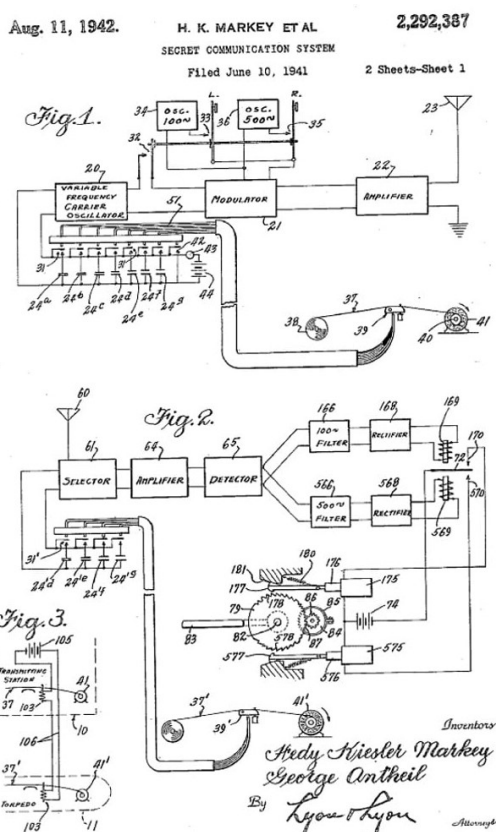


Figura 2.12: Diseño de patente.

[2021]. Se convirtió en la primera mujer en recibir el Premio Bulbie Gnass Spirit of Achievement de la Convención de Invención [Griffiths & Farina, 2021].

El Día del Inventor en su Austria natal se celebra el 9 de noviembre en su honor. Ella es reconocida en el Salón de la Fama de Inventores Nacionales (2014) y en el *National Women's History Museum*. Su legado resalta la interseccionalidad entre arte y ciencia, y ha inspirado biografías y documentales que corrigen su subestimación histórica [Markstrom, 2020].

Margaret Elaine Heafield Hamilton nació el 17 de agosto de 1936 en Paoli, Indiana, actualmente en 2025 sigue activa a sus 89 años [Hamilton & Hackler, 1991], (ver Fig. 2.13). Académicamente, obtuvo una licenciatura en matemáticas de Earlham College en 1958 e inició estudios de posgrado en la Universidad de Michigan, aunque los interrumpió para trabajar en el Proyecto SAGE de defensa aérea [Hamilton, 1986]. Comenzó su carrera en los años sesenta, cuando la programación aún no era considerada una discipli-

Su faceta de inventora fue eclipsada por su fama de estrella de Hollywood durante décadas. Recién en el siglo XXI su contribución tecnológica comenzó a recibir el reconocimiento que merecía, convirtiéndose en símbolo de las mujeres invisibilizadas en la historia de la innovación. La tecnología no fue implementada por los militares hasta la década de 1960, pero desde entonces se ha convertido en la base de tecnologías modernas como Bluetooth y Wi-Fi [Burkland, 2021].

Hedy Lamarr¹⁰ es conocida póstumamente como “la madre del Wi-Fi”, aunque recientemente, los intentos de abordar agravios históricos han caído en otro tipo de historia simplista [Markstrom, 2020]. En las últimas décadas se ha vuelto estándar, incluso de moda, acreditar la invención de Wi-Fi, Bluetooth e incluso teléfonos celulares a la estrella de cine de los años 1940 Hedy Lamarr, cuando en realidad múltiples inventores contribuyeron al desarrollo del salto de frecuencia (ver Fig. 2.12).

En los últimos años de Lamarr¹¹ recibió un premio por su invención. La Electronic Frontier Foundation otorgó conjuntamente a Lamarr y Antheil su Premio Pioneer en 1997 [Burkland,



Figura 2.13: Margaret Hamilton.

¹⁰Biografía (4) Hedy Lamarr

¹¹Biografía (5) Hedy Lamarr

na formal [Hamilton, 2018].

Su carrera despegó en el MIT *Instrumentation Laboratory* [Hamilton & Hackler, 2008], donde dirigió la división de ingeniería de software para NASA¹². Fue la primera programadora contratada para el proyecto Apollo en el MIT y la primera mujer programadora en el proyecto, y posteriormente se convirtió en Directora de la División de Ingeniería de Software (Wikipedia). A principios de la década de 1960, Hamilton se unió al Laboratorio Lincoln del MIT, es allí donde participó en el proyecto *Semi-Automatic Ground Environment* (SAGE), el primer sistema de defensa aérea de Estados Unidos [Hamilton, 1986]. Escribió notablemente software para un programa para identificar aviones enemigos y el rastreo de misiles [Hamilton & Hackler, 1991].



Figura 2.14: Módulo del Apolo 12.

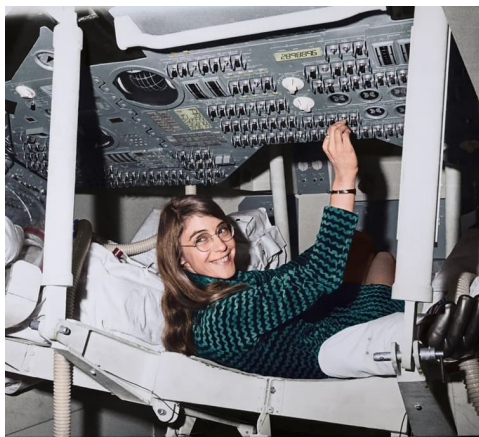


Figura 2.15: En el Módulo Lunar.

tección y recuperación de errores, permitiendo al sistema priorizar tareas críticas [Hamilton & Hackler, 1991], (ver Fig. 2.14).

Gracias a esta lógica, la misión Apollo 11 pudo aterrizar con éxito en la Luna en 1969, a pesar de sobrecargas imprevistas del procesador [Hamilton & Hackler, 2008], (ver Fig. 2.15). Al liderar el equipo de desarrollo del software de vuelo a bordo para el Programa Apolo de la NASA (Apollo 11) [Hamilton & Hackler, 2008]. Su software fue fundamental para evitar un aborto de la misión de alunizaje al priorizar las

Margaret Hamilton fue la directora del equipo de desarrollo de software del MIT *Instrumentation Laboratory* que programó el sistema de navegación de las misiones Apollo de la NASA [Hamilton, 2018]. Su contribución pivotal a la algoritmia fue el diseño del software para la Computadora de Guía de Apollo (AGC) en la década de 1960, incluyendo algoritmos de programación asíncrona y manejo de interrupciones priorizadas [Hamilton & Hackler, 1991]. Estos permitieron multitarea en tiempo real y recuperación de errores, previniendo fallos catastróficos en misiones lunares [Hamilton & Hackler, 2008]. Su aporte a la algoritmia se materializó en los programas de control del módulo lunar, que incorporaban rutinas de de-



Figura 2.16: Apollo Software Code.

¹²Biografía (1) Margaret Hamilton

tareas esenciales del sistema en momentos críticos [Hamilton & Hackler, 1991]. Es conocida por acuñar el término “*Ingeniería de Software*” para legitimar el desarrollo de software como una disciplina de ingeniería [Hamilton, 2018].



Figura 2.17: Ingeniería de Software.

tipo de ingeniería como parte del proceso general de ingeniería de sistemas” [Hamilton, 2018].

A lo largo de su carrera, Hamilton¹⁴ desarrolló los conceptos de software asíncrono, programación de prioridades y capacidad de decisión humana en el circuito, que se convirtieron en la base del diseño de software moderno [Hamilton & Hackler, 1991], (ver Fig. 2.17).

Un error inspirado por su hija pequeña (que jugó con el simulador de vuelo y cargó un programa incorrecto justo antes del lanzamiento) llevó a Hamilton a insistir en la implementación de una sólida detección y recuperación de fallos en el software del Apolo [Hamilton, 1994]. Una anécdota singular data del 20 de julio de 1969, durante el alunizaje de Apollo 11: un error de software activó la alarma 1202, amenazando con abortar la misión [Hamilton, 2018], (ver Fig. 2.18).

Gracias al algoritmo de recuperación de errores de Hamilton¹⁵, el sistema priorizó comandos críticos y salvó el aterrizaje, permitiendo que Neil Armstrong pisara la Luna [Hamilton, 1994]. Tres minutos antes del aterrizaje del Apollo 11, sonaron alarmas cuando la computadora se vio abrumada con datos de un sistema de radar innecesario que había sido activado accidentalmente por la tripulación [Hamilton & Hackler, 1990]. El software de Hamilton, diseñado con detección y recuperación de errores, salvó la misión [Hamilton & Hackler, 2008]. Hamilton recordó estos acontecimientos en una carta de 1971, diciendo “*si la computadora no hubiera reconocido este pro-*

Una imagen icónica muestra a Hamilton junto a una pila de papeles que representa el código que su equipo escribió, símbolo de la magnitud del esfuerzo humano detrás de la exploración espacial¹³, (ver Fig. 2.16).

Hamilton al acuñar el término *software engineering* dio reconocimiento al rigor necesario en la creación de código confiable [Hamilton & Hackler, 1991]. Ella afirmó: “*Comencé a usar el término -ingeniería de software- para distinguirlo del hardware y otros tipos de ingeniería, pero tratando cada*



Figura 2.18: Nasa, Apolo 17.



Figura 2.19: Medalla Presidencial.

¹³Biografía (2) Margaret Hamilton

¹⁴Biografía (3) Margaret Hamilton

¹⁵Biografía (4) Margaret Hamilton

blema y tomado medidas de recuperación, dudo que Apollo 11 hubiera sido el exitoso aterrizaje lunar que fue” [Hamilton, 1986].

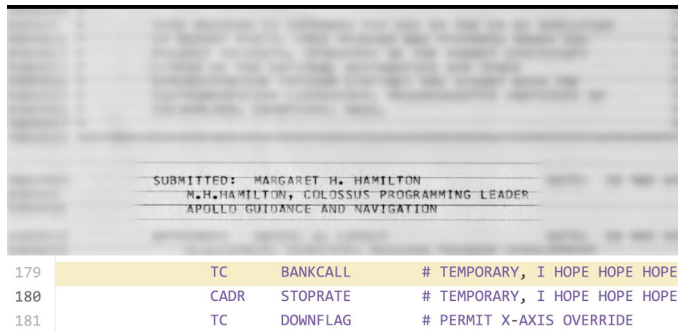


Figura 2.20: Comentarios dentro del código.

de depuración a su trabajo [Hamilton, 1994]. En 2003, recibió el Premio Exceptional Space Act de la NASA por contribuciones científicas y técnicas ¹⁶. El premio incluyó \$37,200, la cantidad más grande otorgada a cualquier individuo en la historia de la NASA [Hamilton & Hackler, 2008]. El 22 de noviembre de 2016, Hamilton recibió la Medalla Presidencial de la Libertad de Barack Obama, el más alto honor civil de Estados Unidos, (ver Fig. 2.19). En 2017, un set de LEGO “Women of NAS” salió a la venta con minifiguras de Hamilton, Mae Jemison, Sally Ride y Nancy Grace Roman [Hamilton, 2018].

En 2019, para celebrar 50 años después del aterrizaje del Apollo, Google decidió hacer un tributo a Hamilton [Hamilton & Hackler, 1990]. Los espejos en la Instalación de Energía Solar Ivanpah se configuraron para crear una imagen de Hamilton y el Apollo 11 bajo la luz de la luna [Hamilton, 1994].

Hamilton sigue siendo una figura de referencia en la ética del software y en la promoción de las mujeres en STEM, además de ser una fuente de inspiración y una voz influyente [Hamilton, 2018]. Fundó dos compañías de software, Higher Order Software en 1976 y su propia empresa, Hamilton Technologies en 1986, ambas en Cambridge, Massachusetts [Hamilton & Hackler, 2008], (ver Fig. 2.20).

En la actualidad ella sigue contribuyendo al desarrollo de metodologías de software. Actualmente, Hamilton es una defensora de la ingeniería de software confiable y sigue impartiendo conferencias sobre ética en IA.

Katherine Louise Bouman¹⁷ nació el 9 de mayo de 1989 en West Lafayette, Indiana, Estados Unidos, de 36 años (2025) [Chanana, 2024], (ver Fig. 2.21).

Su padre, Charles Bouman, es profesor de ingeniería eléctrica e informática y de ingeniería biomédica en la Universidad de Purdue. Como estudiante de secundaria, Bouman realizó una investigación sobre imágenes en la Universidad de Purdue [D’Ignazio & Klein, 2020]. Se graduó cum laude en Ingeniería Eléctrica en la Universidad de Michigan (2011). Obtuvo su Máster (2013) y Doctorado (2017) en Ingeniería Eléctrica y Ciencias de la Computación en el Instituto Tecnológico de Massachusetts (MIT), donde se enfocó en

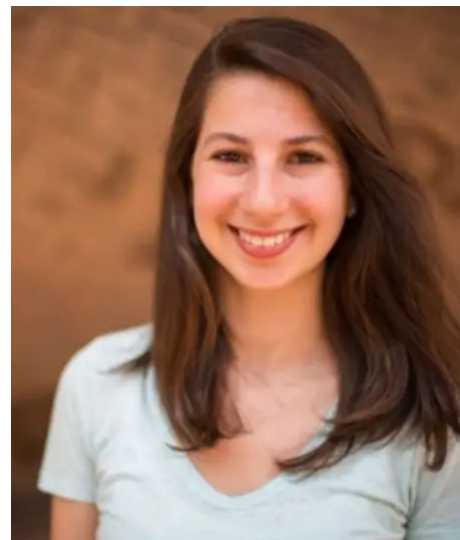


Figura 2.21: Katie Bouman.

¹⁶Biografía (5) Margaret Hamilton

¹⁷Biografía (1) Katie Bouman

imagen computacional [Bouman et al., 2016]. Como postdoc en el Instituto Tecnológico de California (Caltech), lideró el equipo de imagen del EHT¹⁸. En el MIT, fue miembro del Laboratorio de Ciencias de la Computación e Inteligencia Artificial de dicha institución [Bouman et al., 2017].



Figura 2.22: Katie Bouman en TED.

Su tesis de maestría, “*Estimating Material Properties of Fabric through the Observation of Motion*”, recibió el premio Ernst Guillemin a la mejor tesis de máster en ingeniería eléctrica [Bouman et al., 2013]. Su tesis doctoral, “*Extreme imaging via physical model inversion: seeing around corners and imaging black holes*”, fue supervisada por William T. Freeman del MIT [Levis, 2025], (ver Fig. 2.22).

Su avance algorítmico fue el liderar desarrollo del programa CHIRP (Continuous High-resolution Image Reconstruction using Patch priors) en 2016 [Bouman et al., 2016, 2018], un algoritmo de aprendizaje profundo que reconstruye imágenes a partir de datos escasos y ruidosos del Telescopio del Horizonte de Sucesos (EHT) [Bouman et al., 2010, 2017], (ver Fig. 2.23).

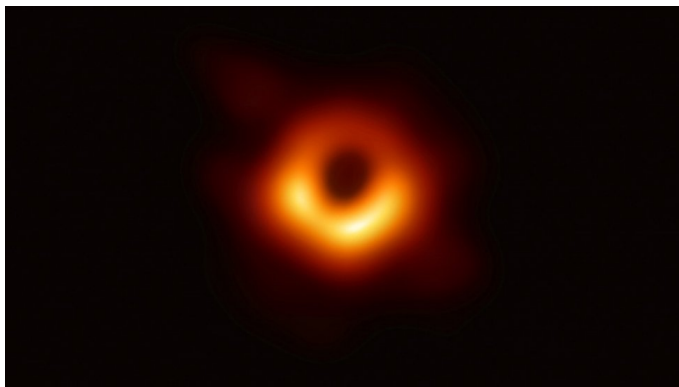


Figura 2.23: Imagen del agujero negro M87.

Este fue uno de los algoritmos de reconstrucción fundamental utilizados por el equipo internacional del Event Horizon Telescope (EHT) para sintetizar los vastos datos de radio y, en 2019 [Bouman et al., 2016, 2018], esto permitió generar la primera fotografía de un agujero negro, una imagen real del M87, fusionando radioastronomía con visión computacional [Bouman, 2020]. Su algoritmo combinaba datos incompletos de múltiples radiotelescopios

y los reconstruía mediante modelos estadísticos [Bouman et al., 2010], una forma avanzada de razonamiento algorítmico que integra física, matemática y computación [Bouman et al., 2010, 2017], (ver Fig. 2.24).

Su trabajo excepcional permitió el desarrollo del algoritmo para la obtención de imágenes de agujeros negros, conocido como Reconstrucción Continua de Imágenes de Alta Resolución utilizando Prioridades de Parche (CHIRP) [Bouman et al., 2018]. Bouman¹⁹ se incorporó al proyecto del Event Horizon Telescope en 2013. CHIRP inspiró los procedimientos de validación de imágenes utilizados en la adquisición de la primera imagen de un agujero negro en abril de 2019 [Levis, 2025], y Bouman desempeñó un papel importante en el proyecto al verificar las imágenes (aproximadamente 5 petabytes de datos) [Bouman et al., 2018],



Figura 2.24: Pilas de discos duros con datos de imágenes de agujeros negros.

¹⁸Biografía (2) Katie Bouman

¹⁹Biografía (3) Katie Bouman

seleccionar los parámetros para filtrar las imágenes tomadas por el Event Horizon y participar en el desarrollo de un marco de imagen robusto que comparaba los resultados de diferentes técnicas de reconstrucción de imágenes [Bouman et al., 2017], (ver Fig. 2.25).



Figura 2.25: Abril de 2019.

Cuando se le preguntó sobre sus inicios, Bouman²⁰ admitió: *“Apenas sabía qué era un agujero negro, pero recuerdo pensar al final de la reunión que realmente quería trabajar en este proyecto”*. Esta honestidad ilustra cómo la pasión y la curiosidad pueden superar la falta inicial de conocimiento especializado [Bouman, 2020]. Su formación era en ciencias de la computación e ingeniería eléctrica, y se involucró en el proyecto mientras cursaba un doctorado en visión computacional [Bouman et al., 2016]. Su pasión es *“idear formas de ver o medir cosas que son invisibles”*, (ver Fig. 2.26).



Figura 2.26: Katie Bouman investigadora en métodos de imagen computacional y visión artificial.

Bouman dirige el Laboratorio de Imagen Computacional [Pasquali, 2020]. Su legado incluye premios como el Breakthrough Prize en Física Fundamental y charlas TED que promueven la colaboración en ciencia, inspirando a jóvenes en IA y astronomía [Levis, 2025].

Finalmente, desde los compiladores de Hopper hasta los algoritmos de imagen de Bouman, estas mujeres han tejido el tapiz algorítmico de la computación moderna. Sus anécdotas —de bugs literales a memes estelares— humanizan su genialidad, recordándonos que la innovación florece en la perseverancia [D’Ignazio & Klein, 2020]. Su legado actual no solo habita en tecnologías cotidianas, sino en un llamado a la inclusión: en un campo aún dominado por hombres, ellas iluminan el camino para futuras pioneras [Bouman et al., 2013].

Una anécdota viral capturó su pasión: al revelarse la imagen del agujero negro M87 en abril de 2019, una foto de Bouman, con ojos desorbitados de emoción, se volvió meme como *“#KatieBug”*, acuñado por un colega [D’Ignazio & Klein, 2020]. Representó no solo su logro, sino el de un equipo global de 200 científicos [Chanana, 2024]. Con esto, ella se convirtió en un símbolo de la joven generación de científicos y programadores [Bouman et al., 2018].

El espacio era pequeño y caliente. En un día fatídico del verano pasado, Katie Bouman y tres compañeros investigadores se encerraron en una pequeña habitación en la Universidad de Harvard, a salvo de miradas indiscretas, para ver una imagen que había tardado años en crearse. En ese día de junio, los datos finalmente habían llegado y el equipo de Bouman presionó *“ir”*, esperando ver si el código que habían escrito realmente podía capturar lo invisible. Es Profesora Asistente de Ciencias de la Computación y Ciencias Matemáticas en el Instituto de Tecnología de California.

²⁰Biografía (4) Katie Bouman

Estas cuatro mujeres, en distintos momentos de la historia, demostraron que la algoritmia no es solo un conjunto de reglas, sino una herramienta creativa con el poder de transformar el mundo, ya sea haciendo los ordenadores más accesibles (Hopper) [Hopper, 1952], protegiendo comunicaciones esenciales (Lamarr) [Markstrom, 2020], salvando misiones espaciales (Hamilton) [Hamilton, 2018], o revelando los secretos del cosmos (Bouman) [Bouman, 2020], (ver Fig. 2.27).



Figura 2.27: Pioneras en la computación.

Ellas en conjunto, trazan una línea de tiempo que recorre casi un siglo de innovación en computación. Desde *la estructuración de los lenguajes de programación por Grace Hopper*, *la visión algorítmica de Hedy Lamarr*, *la ingeniería rigurosa de Margaret Hamilton* y *la creatividad en ciencia de datos de Katie Bouman*, cada una representa un momento importante en la evolución del *pensamiento computacional* ver Fig. 2.28.

Para analizar y profundizar

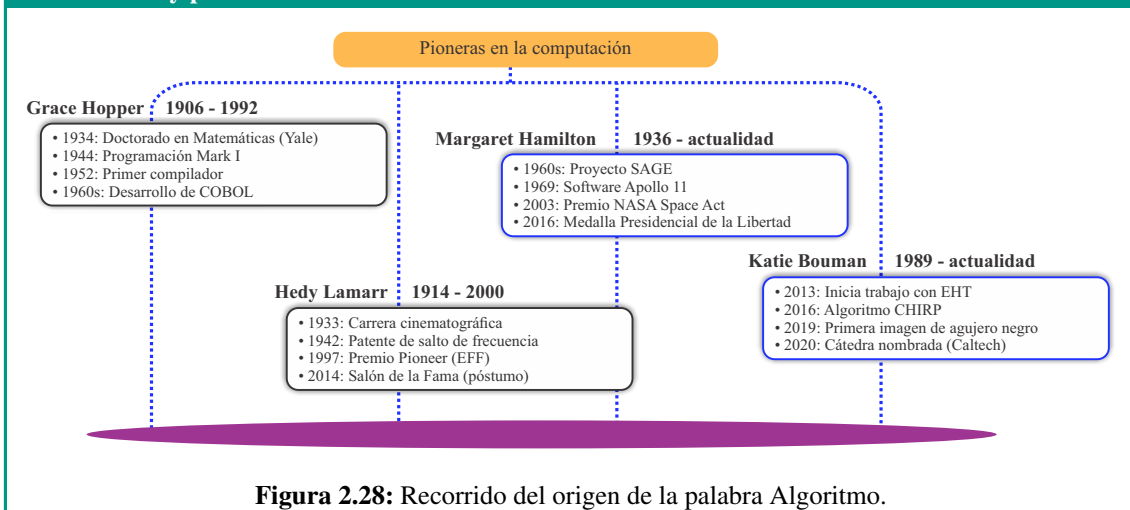


Figura 2.28: Recorrido del origen de la palabra Algoritmo.

Sus trabajos en conjunto demuestran que la programación no solo es un conjunto de códigos, sino una forma de pensar el mundo abstracto, de imaginar soluciones que no existen y de expandir los límites de lo posible desde lo imposible, ver Tabla 2.1.

Comparación entre contribuciones algorítmicas

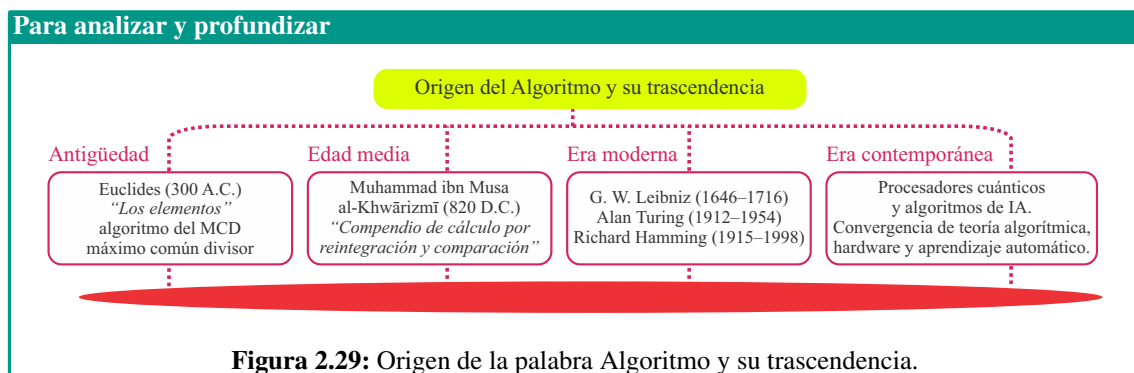
Tabla 2.1: Impacto algorítmico

Pionera	Contribución	Impacto	Referencia
Grace Hopper	Compiladores y COBOL	Traductor de lenguaje natural a código máquina; abstracciones de alto nivel	[Hopper, 1952]
Hedy Lamarr	Salto de frecuencia	Algoritmo de sincronización pseudo-aleatoria para comunicaciones seguras	[Markstrom, 2020]
Margaret Hamilton	Software Apollo	Algoritmos de detección/recuperación de errores; programación asíncrona	[Hamilton, 2018]
Katie Bouman	Algoritmo CHIRP	Reconstrucción de imágenes mediante inversión de modelos físicos	[Bouman, 2020]

2.1.1. Conceptos fundamentales

La búsqueda de métodos para resolver problemas ha sido una constante desde la antigüedad y más aún en la era de la digitalización [Eschenburg, 2020]. Hoy lo que llamamos “*algoritmos*” proviene, como se indicó anteriormente, del nombre del matemático persa Muhammad ibn Musa al-Khwārizmī²¹ (780–850 D.C.), cuyo trabajo en el siglo IX sobre el “*Compendio de cálculo por reintegración y comparación*” puso los inicios para el proceso de cálculo sistemático y, la unión de la aritmética y el álgebra, con ello el origen a lo que llamamos, algoritmo [Tatarchenko, 2023; Nabirahni et al., 2019]. Al-Khwārizmī trabajó en la Casa de la Sabiduría de Bagdad (periodo del Califato Abasí, actualmente Bagdad es capital de Irak), una institución que se convirtió en el centro del conocimiento científico del mundo islámico [Tatarchenko, 2019; Clark, 2012]. Sus métodos matemáticos representaban estructuras algorítmicas, aunque desarrollados hace más de mil años, contenían ya la esencia de lo que hoy entendemos como instrucciones precisas y finitas para resolver problemas [Malet & Panza, 2006]. Entre el año 820 al 825 D.C., publicó en Bagdad su tratado, sentando las bases para el álgebra, procedimientos de cálculo, el pensamiento algorítmico y los métodos iterativos que hoy subyacen en todo software. Así, su trabajo despierta la esencia misma de la computación y el pensamiento lógico [Parshall, 1988]. Para el año del 859, la Universidad de Al Quaraouiyine en la ciudad imperial de Fez, en Marruecos, comenzó a ser un centro de la transferencia de conocimiento, un pilar para el desarrollo posterior de los algoritmos. Este logro transformó las matemáticas en el mundo islámico medieval, que luego de traducciones al latín en la Universidad de Toledo en el siglo XII, influyó en la Revolución Científica europea, demostrando cómo un algoritmo puede trascender fronteras y épocas para alterar el curso de la historia humana [Oaks & Alkhateeb, 2007].

Los algoritmos, son secuencias precisas de instrucciones diseñadas para resolver problemas de manera sistemática, representan el centro de la computación moderna. Este proceso se define como un conjunto finito de instrucciones bien definidas, ordenadas y finitas que permiten realizar una tarea o resolver un problema [Oliveira & Figueiredo, 2024]. La noción de algoritmo estableció procedimientos sistemáticos para resolver ecuaciones, y su concepto trasciende épocas y culturas [Dunlop, 1943; Angluin, 2016]. Desde los cálculos astronómicos de los babilonios alrededor del año 2000 A.C., donde se inscribieron en tablillas cuneiformes los primeros pasos lógicos para predecir eclipses, hasta las complejas simulaciones que impulsan el Observatorio Espacial James Webb, los algoritmos han evolucionado de simples procedimientos manuales a pilares de la inteligencia artificial que procesan datos a escalas inimaginables [Bosworth, 1964; Abbate et al., 2023], ver Fig. 2.29.



Los conceptos fundamentales de los algoritmos se enmarcan en principios básicos para entenderlos, desde cómo se diseñan o representan, así cómo se analizan y aplican en la resolución de problemas reales, logrando abstraer una solución.

²¹ Biografía de Al-Khwārizmī

Definición de un algoritmo

Un algoritmo es un conjunto de pasos que se reproducen en una secuencia finita bien definida, ejecutable y ordenada que, desde un punto de partida (una entrada), alcanza un punto de llegada (o produce una salida) en un tiempo realizable (finito) para conseguir (o resolver) la solución de un problema dado (o muy específico) [Cormen et al., 2022], o como “*Un conjunto finito de reglas precisas que especifican una secuencia de operaciones para resolver un problema específico*”, por Knuth [2022].

Según Knuth [2022], los algoritmos tienen características esenciales. Por ejemplo, los algoritmos deben finalizar luego de un conjunto finito de pasos y en un tiempo determinado. Cada uno de estos pasos deben ser precisos (entendibles e inequívocos). A la vez, deben recibir datos de entrada (valores) y presentar al menos un dato de salida [Jiménez Toledo et al., 2023]. Es así como cada uno de los pasos debe ser realizable y efectivo (en un tiempo finito) ver Fig. 2.30.



Figura 2.30: Imagen desarrollada con napkin.ai para el tema: Características de un Algoritmo.

Las entradas y salida, conocidas como *input* y *output*, son los datos que se ingresan a los algoritmos (los primeros) y el resultado es el que se obtiene luego del procesamiento de los primeros (los segundos), estos se refieren a las variables [Altaher & Ferchichi, 2018], ver Fig. 2.31.

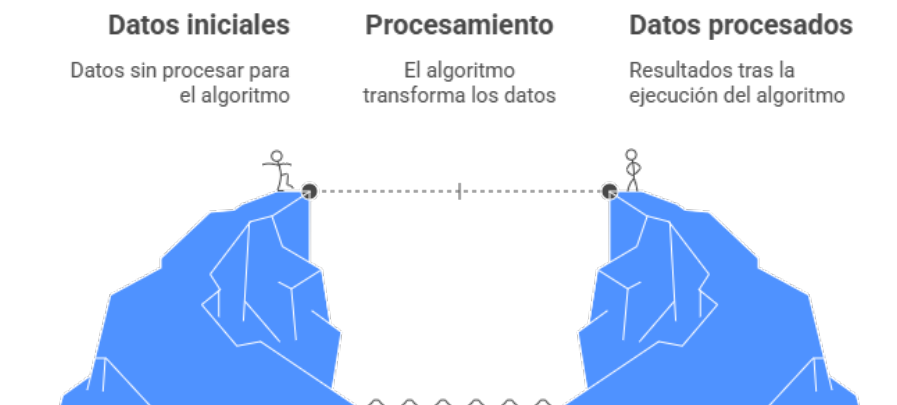


Figura 2.31: Imagen desarrollada con napkin.ai para el tema: Transformación de datos.

Otra característica de los algoritmos son las estructuras de control, bloques de instrucciones (o de código) que abordan un flujo de procesos [Vinayakumar et al., 2018; Stamouli et al., 2004], entre ellas están las estructuras secuenciales, las estructuras selectivas y las estructuras repetitivas, que guardan una jerarquía de uso [Silva-Maceda et al., 2016; Thomas et al., 2002], ver Fig. 2.32.

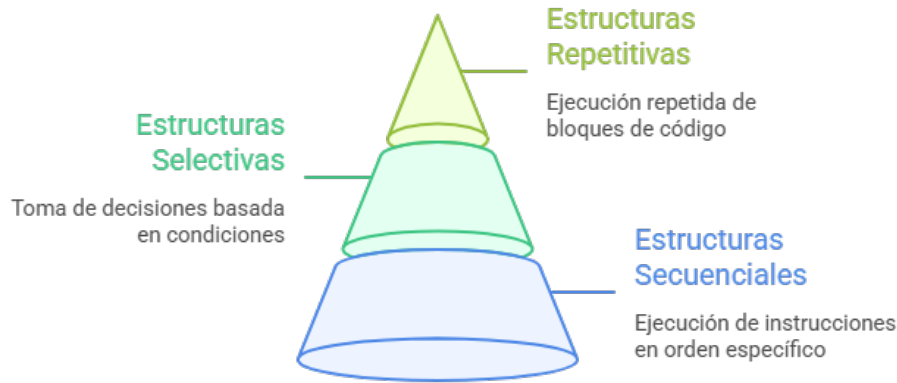


Figura 2.32: Imagen desarrollada con napkin.ai para el tema: Jerarquía de las estructuras.

Las estructuras secuenciales dirigen el flujo del procesamiento en un diseño, muchas veces utilizado una metodología de lectura de arriba hacia abajo (top-down) es decir, el computador (desde el compilador) irá leyendo línea a línea, que haya sido escrita en el archivo fuente (o principal) [Oomori et al., 2019; Watson & Li, 2014], así cada línea escrita (instrucción) será interpretada a medida que esta aparezca (editor), empezando de arriba (desde la primera línea que se escriba) hasta abajo (hasta la última línea o línea final que aparezca) [Futschek, 2006], ver Fig. 2.33.

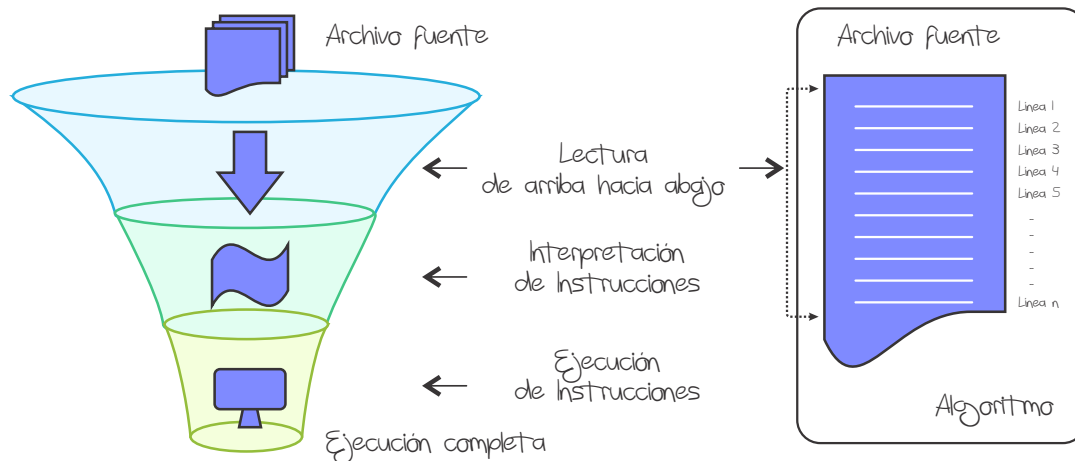


Figura 2.33: Estructura secuencial de la ejecución de un archivo fuente.

Otra característica que guardan los algoritmos es la exactitud, es decir, si es correcto, para todas las entradas válidas (variables iniciales), y que produzcan la salida esperada (variables finales) según la especificación del problema [Passi & Jackson, 2017; Hartmann et al., 2001].

Con esto, además, el análisis de eficiencia que mide los recursos que consume el algoritmo es otra característica tomada en consideración a medida que crezca el algoritmo en líneas de código e invoque recursos internos como externos haciendo uso de todos los enlaces, conexiones o nexo hacia datos, puertos de comunicación, sistemas externos [Thomas et al., 2018; Dourish, 2016].

La complejidad es otra característica llamada análisis de Complejidad de Algoritmos. La complejidad mide los recursos (tiempo y memoria) que un algoritmo requiere para ejecutarse. Es fundamental para predecir su rendimiento en función del tamaño de la entrada. La complejidad temporal ($T(n)$), mide el número de operaciones realizadas en función del tamaño de la entrada (n) [Wu et al., 2024].

Se expresa con la notación Big-O (O), que describe el límite superior del tiempo de ejecución en el peor de los casos [Sefati et al., 2025]. El Orden de Eficiencia sería: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$ ver Fig. 2.34.

- $O(1)$ **(Constante)**: El número de operaciones es fijo, independiente de n [Chen et al., 2025]. *Ejemplo*: Acceso directo a un elemento en un arreglo [Wang et al., 2025a; Kolmogorov, 2025].
- $O(\log n)$ **(Logarítmica)**: El tiempo de ejecución crece muy lentamente [Bender et al., 2025]. El algoritmo descarta grandes porciones de la entrada en cada paso. *Ejemplo*: Búsqueda binaria [Sinnamon & Tarjan, 2025; Bhattacharya et al., 2025].
- $O(n)$ **(Lineal)**: El tiempo de ejecución es directamente proporcional a n [Akitaya et al., 2025; Chernyshev et al., 2025]. *Ejemplo*: Recorrer una lista una vez [Li et al., 2025b].
- $O(n \log n)$ **(Log-Lineal)**: Típica de algoritmos eficientes de “divide y vencerás” [Li et al., 2025b]. Es la complejidad más deseable para la ordenación [Wang et al., 2025c]. *Ejemplo*: Algoritmos de ordenación como *Merge Sort* o *Quick Sort* [Leverrier & Zémor, 2025].
- $O(n^2)$ **(Cuadrática)**: El tiempo de ejecución es proporcional al cuadrado de n (típicamente dos bucles anidados) [Wang et al., 2025b]. *Ejemplo*: Algoritmos de ordenación ineficientes como *Bubble Sort* [Gabow, 2024].
- $O(2^n)$ **(Exponencial)**: El tiempo de ejecución se duplica por cada aumento unitario de n [Malekian et al., 2025]. Solo viable para entradas minúsculas [Chapuy & Perarnau, 2025; Hirobe & Kasai, 2025].



Figura 2.34: La complejidad del algoritmo.

En cambio, la Complejidad Espacial ($S(n)$) mide la cantidad de memoria adicional (espacio auxiliar) que el algoritmo necesita para ejecutarse, sin contar el espacio necesario para la entrada misma [Li et al., 2025c].

- $O(1)$ **(Constante)**: La memoria adicional requerida es fija, sin importar el tamaño de n . *Ejemplo*: Un algoritmo que opera in-place sin estructuras de datos auxiliares [Assadi et al., 2025].
- $O(n)$ **(Lineal)**: La memoria adicional es directamente proporcional a n . *Ejemplo*: Crear una copia auxiliar de la estructura de datos de entrada [Beelen & Neiger, 2025].

La complejidad del algoritmo se refiere a la eficiencia de los recursos para alcanzar el objetivo (rapidez de encontrar la solución y el uso disponible de la memoria para hacerlo) [Nguyen et al., 2024; Xu & Li, 2024].

Algunas estrategias comunes para crear algoritmos y se enmarcan en seis características diferenciadoras. Estas características representan a los paradigmas del diseño, ver Fig. 2.35.

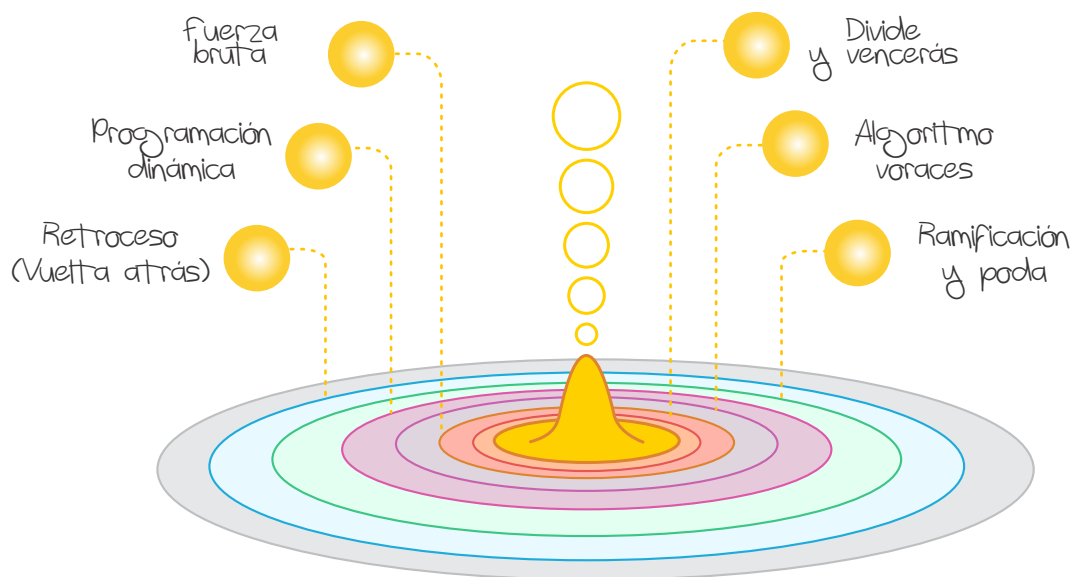


Figura 2.35: Estrategias de diseño de algoritmos basadas en paradigmas.

El paradigma de la *fuerza bruta* es una estrategia de encontrar todas las posibilidades de solución para un problema dado e ir verificando cuál es la más óptima. El ejemplo más común es adivinar una contraseña probando cada combinación de caracteres posibles [Sato et al., 2025].

El paradigma de *divide y vencerás* es una estrategia que aborda un problema de gran tamaño dividiéndolo en dos o más problemas pequeños, de esa manera se encuentra la solución juntando todas, para formar la solución al problema original. Dos ejemplos son los algoritmos de ordenamiento (Merge Sort y Quicksort), o la búsqueda binaria [Childs et al., 2025].

El paradigma de la *Programación dinámica* es una estrategia que aborda la construcción de soluciones óptimas a partir de soluciones de subproblemas que pueden estar superpuestos [Thengvall et al., 2025]. Es decir, los problemas se resuelven recursivamente, una y otra vez, almacenando las soluciones para construir progresivamente las soluciones de problemas más grandes. Los ejemplos más comunes son la secuencia de Fibonacci o el problema de *Knapsack* (mochila) [Xiao et al., 2025].

El paradigma de los *algoritmos voraces o Greedy* es una estrategia de toma de decisiones basándose en la mejor opción cercana [García-Díaz et al., 2025], por ejemplo, el algoritmo de Kruskal (árbol de expansión mínima) o el algoritmo de Dijkstra (camino o ruta más corta) [Czarnul et al., 2025].

El paradigma de los *algoritmos de retroceso o vuelta atrás* (backtracking) es una estrategia utilizada para ir encontrando soluciones parciales o diferentes, en los problemas de búsqueda,

construyendo soluciones incrementales [Stoddart et al., 2024; Chatterjee et al., 2024]. En cada ejecución, si una solución no conduce a nada “retrocede” al paso anterior y se prueba otra alternativa, a esto se le llama “poda” de la rama de soluciones [Hatami & Hoza, 2024]. Ejemplos de esta característica es, el problema de las N-Reinas, resolver el Sudoku, o encontrar caminos o rutas dentro de un laberinto [Jin et al., 2025].

El paradigma de los *algoritmos de ramificación y poda* (branch and bound) es una estrategia utilizada en problemas de optimización, para encontrar más de una solución óptima [Balcan et al., 2024; Nacar et al., 2024]. Al generar las posibles ramificaciones o soluciones “poda” aquellas soluciones que no pueden superar a la mejor solución actual [dos Santos & de Lima, 2024; Hien et al., 2024].

Cuatro características más que encontramos en el diseño y creación de algoritmos son: la recursión, la invariante de bucle, los casos existentes (peor, promedio y el mejor) y, la optimización y compensaciones.

La estrategia de diseñar algoritmos mediante recursividad es una técnica elegante que permite realizar la invocación a sí mismo de instancias, más pequeñas, del programa. Dos ejemplos son, el caso del factorial de un número o el problema de las torres de Hanoi [Abo Khamis et al., 2024; Qiu et al., 2025]. La estrategia de la invariante del bucle es la técnica de la afirmación que se mantiene verdadera antes y después de cada iteración para probar la exactitud de los bucles o lazos [Redondi et al., 2024; Vincent & Schwager, 2025]. La estrategia del análisis de casos, determina tres comportamientos, el mejor caso donde el algoritmo sea más eficiente o trabaje menos ($\mathcal{O}(n)$ en el algoritmo Quicksort ya ordenado) [Sun et al., 2024b]. El promedio, cuando el comportamiento se da con situaciones aleatorias. Mientras que el peor de los casos se da cuando el algoritmo es menos eficiente o trabaja más ($\mathcal{O}(n^2)$ en el algoritmo Quicksort) [Zhang & Vatedka, 2024], ver Fig. 2.36.

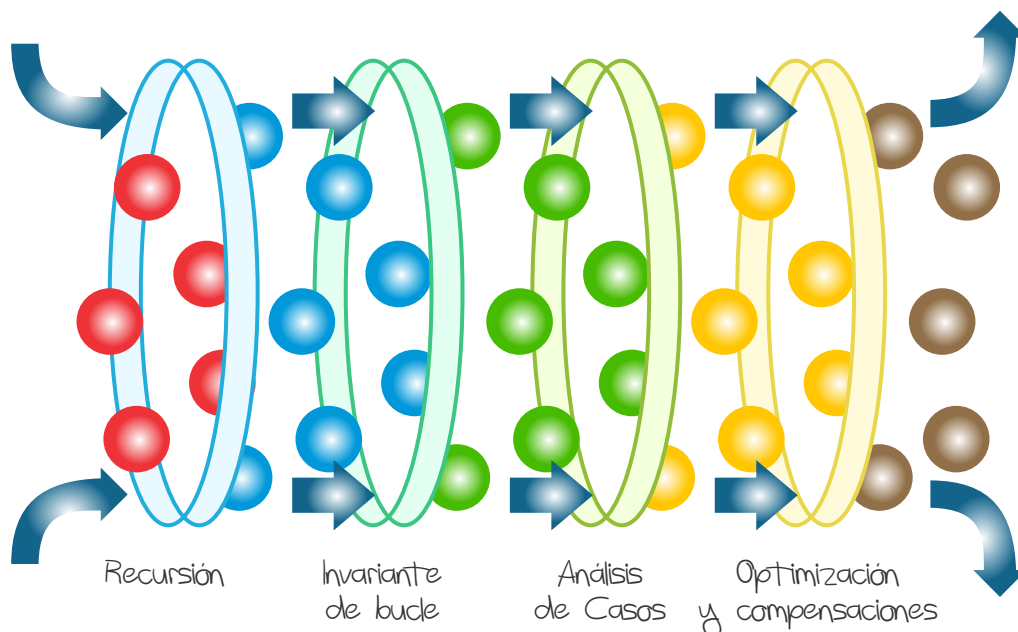


Figura 2.36: Características estratégicas de diseño de algoritmos.

Finalmente, la última estrategia es conocida como optimización y compensación, hay que recordar que todo algoritmo conlleva el compromiso del consumo adecuado del tiempo en relación con el espacio en disco o en memoria [Cuevas et al., 2025; Zhu et al., 2024].

También la necesidad de la simplicidad que se requiere en relación con la velocidad de ejecución total del algoritmo [Kundu et al., 2024; dos Santos & de Lima, 2024]. Así como la exactitud

del algoritmo en relación con la aproximación en situaciones que se requiera (por ejemplo, los algoritmos heurísticos y metaheurísticos) [Rajwar & Deep, 2025].

2.1.2. Formas de representar los algoritmos

Los algoritmos se componen de secuencias de instrucciones bien definidas y pueden ser representadas de diferentes maneras. Estas representaciones son requeridas desde sus diversos enfoques, desde lo textual hasta lo visual acorde a cada etapa del desarrollo del software.

Varios autores han ido promoviendo la claridad y la estructura en el diseño algorítmico. El teorema de estructura de Böhm & Jacopini [1966] estableció que cualquier algoritmo computable se puede expresar con solo tres estructuras básicas: estructuras secuenciales, estructuras de selección y estructuras de repetición, lo que influyó en representaciones posteriores como los diagramas estructurados. Asimismo, Dijkstra [1968a] criticó el uso descontrolado de saltos incondicionales (como GOTO), impulsando representaciones más disciplinadas [Liu et al., 2024], ver Fig. 2.37.



Figura 2.37: Formas de representar algoritmos.

Entonces, uno de los primeros pasos será la identificación del problema y definir su solución. Luego se desarrolla el algoritmo desde la solución secuencial de pasos que se han determinado para resolver el problema [Ensmenger, 2016]. El algoritmo es representado de forma visual a través de la creación de un flujo, utilizando símbolos geométricos (rectángulos, rombos, paralelogramos, etc.) [Wirth, 1973].

A continuación se describe y escribe el algoritmo mediante el uso de un código similar al lenguaje de máquina, conocido como pseudocódigo o seudolenguaje. Opcionalmente, se puede representar el algoritmo mediante los diseños de los autores Nassi & Shneiderman [1973], [Yoder & Schrag, 1978; Dijkstra, 1968b]. Se presenta a continuación un recorrido histórico de la representación de los algoritmos y sus autores, ver Tabla 2.2.

Evolución histórica de las representaciones algorítmicas

Tabla 2.2: Principales fechas en la representación de algoritmos

Año	Representación	Autores/Contribuidor	Contexto
1921	Flujogramas iniciales	Frank Gilbreth	Aplicación industrial para <i>optimización</i> y <i>análisis de procesos de trabajo</i> .
1947	Flujogramas computacionales	John von Neumann y Adele Goldstine	Representación lógica de la <i>secuencia de programas</i> , para máquinas <i>ENIAC</i> .
1966	Teorema de estructura (base para pseudocódigo y NSD)	Corrado Böhm y Giuseppe Jacopini	Comprobación matemática de solo <i>tres estructuras básicas</i> para construir cualquier algoritmo.
1968	Pseudocódigo (pidgin ALGOL)	Comunidad ALGOL (con Tony Hoare)	<i>Notación de alto nivel</i> para el diseño algorítmico, independiente de la sintaxis final del lenguaje.
1973	Diagramas Nassi-Shneiderman (NSD)	Isaac Nassi y Ben Shneiderman	Alternativa gráfica para la <i>programación estructurada</i> que elimina el uso de saltos o GOTO.
1997	Diagramas de actividad UML	Object Management Group (OMG)	<i>Extensión de flujogramas</i> para modelado orientado a objetos y <i>procesos de negocio complejos</i> .

Los flujogramas o *Flowcharts*

Los flujogramas son representaciones gráficas que emplean un conjunto de símbolos estandarizados (como óvalos para inicio/fin, rectángulos para procesos y rombos para decisiones) para ilustrar de manera visual el flujo de control lógico de un algoritmo. Aunque son muy intuitivos y valiosos en la fase de planificación inicial, pueden volverse engorrosos y difíciles de seguir en algoritmos complejos o de gran tamaño. Su origen con [Gilbreth & Gilbreth \[1921\]](#), los introdujo para analizar procesos industriales. Su aplicación a los algoritmos computacionales se consolidó en la década de 1940 gracias a las contribuciones de John von Neumann y Adele Goldstine [[Goldstine, 1977](#)], ver Fig. 2.38.

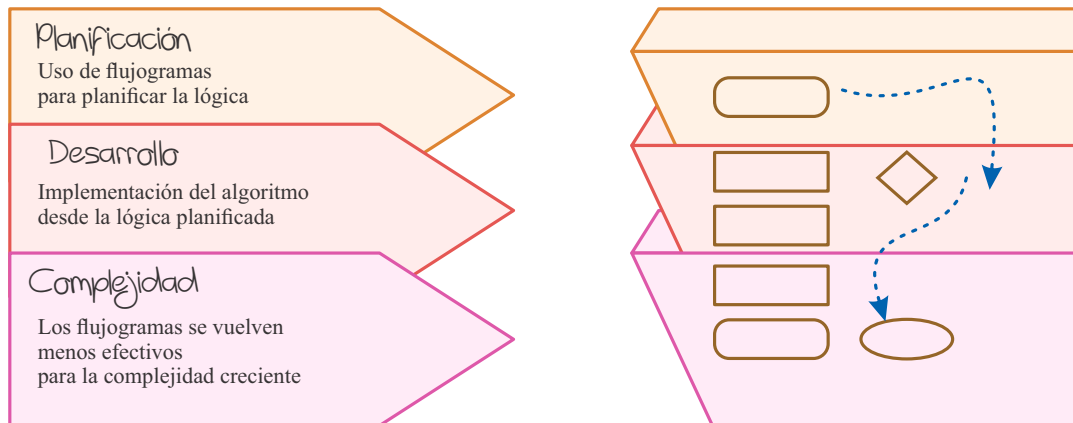


Figura 2.38: Los flujogramas.

El Pseudocódigo

El Pseudocódigo es un enfoque textual y semiformal que describe un algoritmo utilizando el uso del lenguaje natural (como el español o inglés) y construcciones sintácticas similares a las de un lenguaje de programación real, pero sin adherirse a reglas estrictas de compilación o implementación. Esta conformación abstracta lo convierte en una herramienta ideal para el diseño y la revisión formal, ya que facilita la comunicación entre desarrolladores y agiliza la traducción posterior a código fuente funcional. Surgió en la década de 1960 como un “pidgin code” influenciado por lenguajes como ALGOL [[Ponder et al., 1988](#)], ofreciendo una representación legible e independiente de la plataforma final [[Hoare, 2022](#)], ver Fig. 2.39.

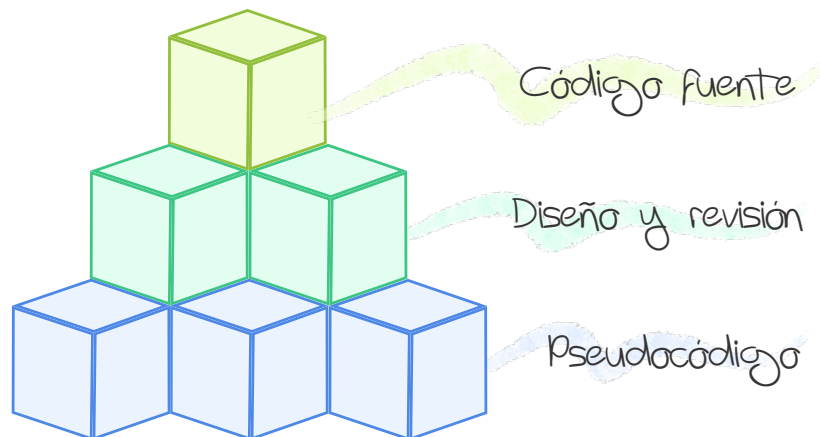


Figura 2.39: El Pseudocódigo.

Diagramas Nassi-Shneiderman (NSD)

Los diagramas Nassi-Shneiderman (NSD) son una alternativa gráfica avanzada a los flujogramas, diseñados específicamente para reforzar la programación estructurada. Estos diagramas utilizan bloques rectangulares anidados y delimitados para representar las estructuras de control fundamentales (secuencia, selección e iteración), eliminando completamente las flechas de flujo y, por lo tanto, previniendo el código “*espaguete*”. Fueron propuestos por [Nassi & Shneiderman \[1973\]](#) (Isaac Nassi y Ben Shneiderman en 1973)²² como una forma más limpia y jerárquica de visualizar la lógica algorítmica. Su estructura se alinea directamente con los principios promovidos por autores clave en la programación estructurada, ver Fig. 2.40.

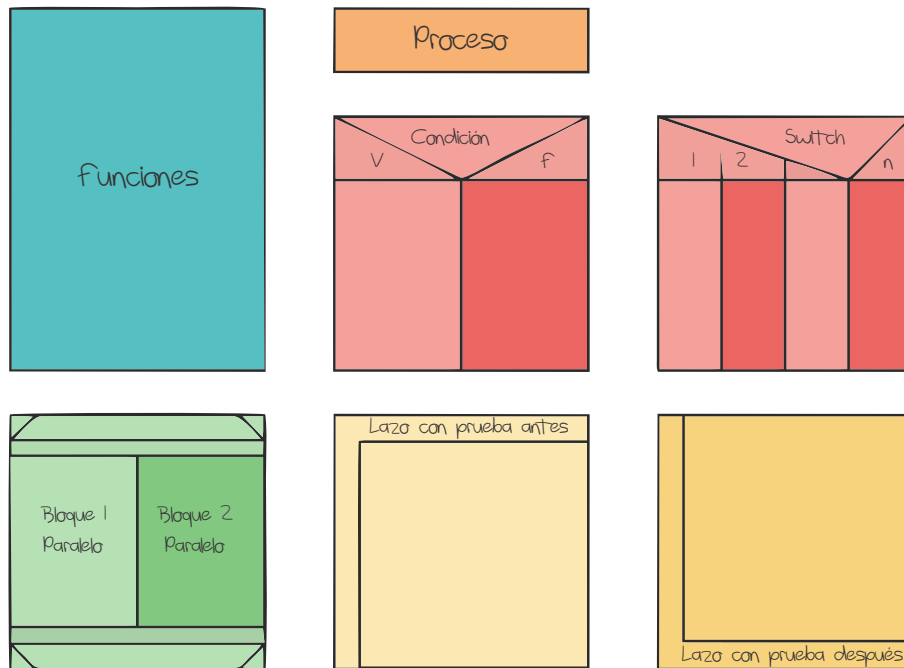


Figura 2.40: Diagramas Nassi-Shneiderman.

Otras representaciones de los algoritmos

Podemos mencionar representaciones que incluyen diagramas de actividad UML, Lenguaje Unificado de Modelado, (desarrollados a partir de 1990 por el Object Management Group para modelado orientado a objetos) y notación [Bachman \[2024\]](#) para bases de datos, pero las tres anteriores son fundamentales en educación y diseño algorítmico [[Liu et al., 2025](#)], ver Fig. 2.41.

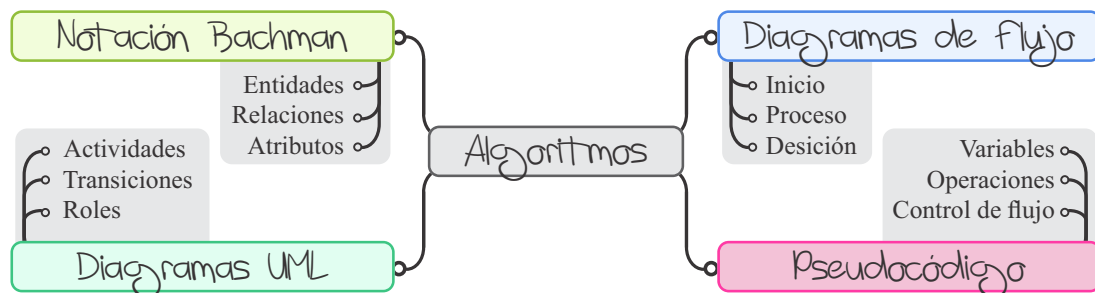


Figura 2.41: Representaciones relevantes y sus características.

²²Disponible en el sitio de <https://www.softwareideas.net/nsd-chart>

De cada representación se destacan ventajas y desventajas. Por ejemplo, los flujogramas son visuales y fáciles de entender, pero escalan mal en complejidad cuando se extienden demasiado. Por el contrario, el pseudocódigo es conciso y modular, pero carece de visualidad para no programadores. Los NSD equilibran ambos, promoviendo legibilidad en entornos educativos, aunque requieren aprendizaje inicial.

Las constantes y las variables

Las *variables* y las *constantes* son los elementos necesarios para construir cualquier desarrollo computacional [Knuth, 1997]. Una *variable* es un espacio en la memoria de la computadora (como una caja de almacenamiento a medida) que guarda un valor (como un número o un texto) que está destinado a *cambiar* o ser modificado durante la ejecución del programa [Van Roy & Haridi, 2004]. Se le asigna un nombre simbólico (como “contador”, “nombre”, “num” o “saldo”) para que el programador pueda referirse a ese espacio y manipular su contenido según lo requiera el flujo del algoritmo (la idea abstracta) [Rajaraman, 1994], permitiendo a los programas responder a condiciones dinámicas del mundo real [Stepanov & McJones, 2009].

Por el contrario, una *constante* es un espacio de memoria cuyo valor es *inmutable*, *invariable*, *que no cambia*; una vez que se define, no puede ser alterado a lo largo de la vida del programa [Apt & Wallace, 2006]. Las constantes se utilizan para almacenar valores críticos que deben ser fijos, como el número π , números atómicos, valores de densidad, el valor de la gravedad (en un planeta), tasas de impuestos o límites de seguridad, garantizando que estos datos esenciales no se modifiquen por error [Sterling & Shapiro, 1994]. Declarar un valor como constante es una buena práctica de diseño que indica a cualquier lector del código que ese valor está blindado y debe permanecer estable y sin cambios [Vagianou, 2006; Develaki, 2019].

El uso estratégico de *constantes* mejora la *legibilidad*, simplifica el *mantenimiento* y *previene errores*, especialmente al centralizar valores fundamentales para que un cambio solo deba hacerse en un único lugar [Chen et al., 2011]. Las *variables*, por el contrario, proporcionan la *flexibilidad* necesaria para implementar procesos dinámicos como bucles y acumuladores [Reas & Fry, 2007; Lin et al., 2021]. Un programador logra un equilibrio preciso entre estos dos conceptos (constantes y variables), separando lo que debe permanecer estable de lo que debe ser dinámico, lo que resulta en un *software* más robusto, elegante y fácil de mantener [Strachey, 2000], ver Fig. 2.42.

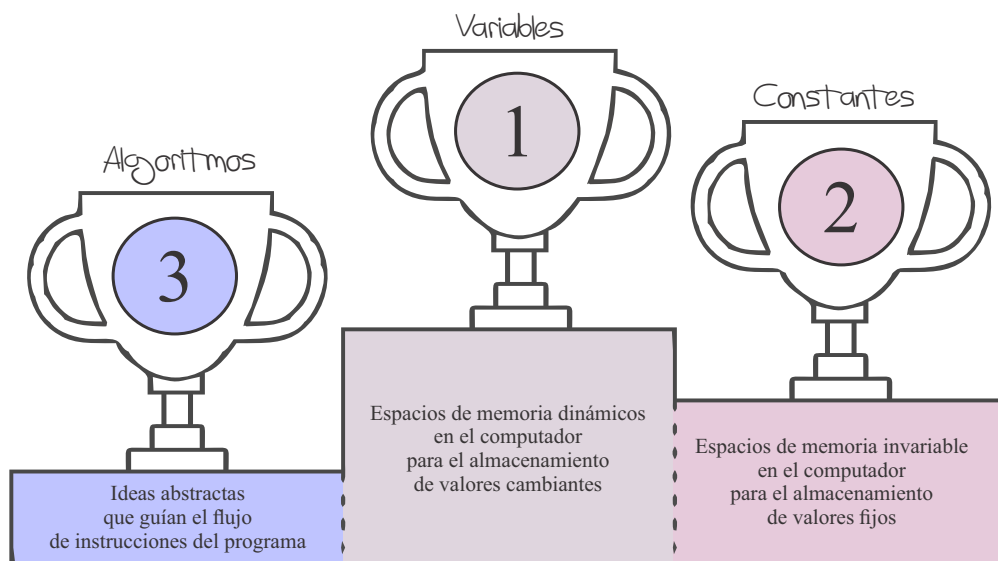


Figura 2.42: Las constantes y las variables.

Ámbito, alcance y los tipos de datos

El ámbito (o *scope*) de una variable o constante define en qué partes del programa se puede acceder a ella y durante cuánto tiempo existe en memoria [Karau et al., 2015]. Es uno de los criterios más importantes que cualquier lenguaje de programación impone para evitar conflictos de nombres y proteger la integridad del código [Lee, 2008]. Existen principalmente cuatro tipos de ámbito: **global**, **local** (o de función), de **bloque** y de **módulo** (o estático en algunos lenguajes) [Casciaro & Mammino, 2016; Singh, 2021]. Una variable global nace cuando se inicia el programa y muere cuando este termina, logrando ser vista desde cualquier función o bloque [Dalmau, 2004]; en cambio, una variable local solo existe dentro de la función o bloque donde se declara y desaparece al salir de él [Voelter et al., 2015]. El ámbito de bloque, popularizado por lenguajes como C (desde C99 con `for(int i=0;...)`), JavaScript (con `let` y `const`) y Python, restringe aún más la visibilidad a un par de llaves [Pym et al., 2019]. El buen uso del ámbito permite encapsular datos, reducir efectos secundarios y hacer que el código sea más predecible y fácil de depurar [Wampler & Payne, 2014; Bayle, 2013].

Los tipos de datos, por su parte, determinan qué clase de valores puede almacenar una variable o constante, qué operaciones son válidas sobre ella y cuánto espacio ocupa en memoria [Ashok, 2010]. Los lenguajes modernos los clasifican en tipos primitivos (o básicos) y tipos compuestos (o derivados) [Gerez, 1999]. Entre los primitivos más comunes están: entero (*int*), *real* o de punto flotante (*float*, *double*), carácter (*char*), booleano (*bool* o *boolean*) y, en algunos lenguajes, tipos como *byte*, *short*, *long* o *decimal* para mayor precisión [Cormen et al., 2022]. Los tipos compuestos incluyen arreglos, cadenas de texto (*string*), estructuras (*struct* o *record*), uniones, enumeraciones y punteros [Lagouvardos et al., 2020]. Además, desde finales del siglo XX, los lenguajes ofrecen tipos genéricos (como *List<T>* en Java y C#, o *templates* en C++) y tipos nulos seguros (*Option<T>* en Rust, *Optional<T>* en Java, o el tipo *nullable* en Kotlin y C#), que permiten expresar explícitamente la posible ausencia de valor y evitar el temido *NullPointerException* [Austin & Chancogne, 1999; Anami et al., 2010], ver Fig. 2.43.

La relación entre ámbito y tipos de datos es profunda y bidireccional [Viswanadham & Narahari, 2015]. Un tipo estático (como en C, C++, Java o Pascal) obliga al compilador a conocer el tipo en tiempo de compilación, lo que permite detectar muchos errores antes de ejecutar el programa y optimizar el uso de memoria [Sasikumar et al., 2014]; en cambio, un tipo dinámico (Python, JavaScript, Ruby, PHP) decide el tipo en tiempo de ejecución, ofreciendo mayor flexibilidad pero trasladando la responsabilidad de la corrección al programador y al entorno de pruebas [Solomon, 1984]. A su vez, el ámbito influye directamente en la vida útil del dato: una variable local de tipo estático se destruye al salir del bloque (liberando memoria automáticamente en lenguajes con recolección de basura o mediante el *stack* en C/C++) [Yevick, 2005], mientras que una variable global o estática mantiene su valor entre llamadas, lo que puede ser tanto una ventaja (para contadores globales o configuraciones) como una fuente de errores difíciles de rastrear [Naramore et al., 2005; Kessler, 2006].

Dominar el ámbito y los tipos de datos, así como conocer la sintaxis de declaración de cada

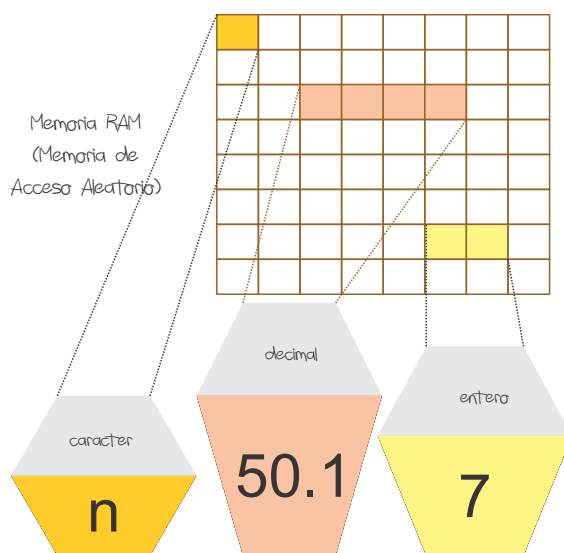


Figura 2.43: Almacenamiento en la memoria de acceso aleatorio por tipo de dato.

lenguaje, es comprender la forma del diseño que busca el equilibrio entre seguridad, rendimiento y expresividad [Kurgalin & Borzunov, 2019]. Un programador que declara conscientemente `const int MAX_INTENTOS = 3`; en ámbito global, `int contador = 0`; en ámbito de función y `let` temporal: `f64 = calcular(); ...` en ámbito de bloque está, en realidad, escribiendo código más robusto, legible y preparado para crecer o escalar [Dale et al., 2000; Bantyukov et al., 2020]. En la práctica, ignorar estas reglas suele traducirse en bugs no esperados, vulnerabilidades de seguridad y costos de mantenimiento elevados; respetar las reglas, es una de las marcas distintivas del desarrollo de un software bien escrito, perdurable y mejorable [Doorenbos, 1995; Dale & Weems, 1996].

Las pruebas de escritorio

Las pruebas de escritorio (conocida como *desk checking* o *dry run*) son la forma más elemental y la más poderosa de verificar la corrección de un algoritmo antes de confiarlo a una máquina. Consisten en ejecutar manualmente, paso a paso, el pseudocódigo, flujograma o código fuente sobre papel o pizarra, siguiendo el flujo de control y actualizando el valor de cada variable en una tabla de seguimiento. Aunque hoy existen procesos de verificación visual dentro de los lenguajes de programación, en los entornos de depuración integrados, su práctica sistemática precede a la era de los computadores electrónicos demostrando repetidamente su capacidad para descubrir errores lógicos que las pruebas automáticas muchas veces pasan por alto, ver Fig. 2.44.

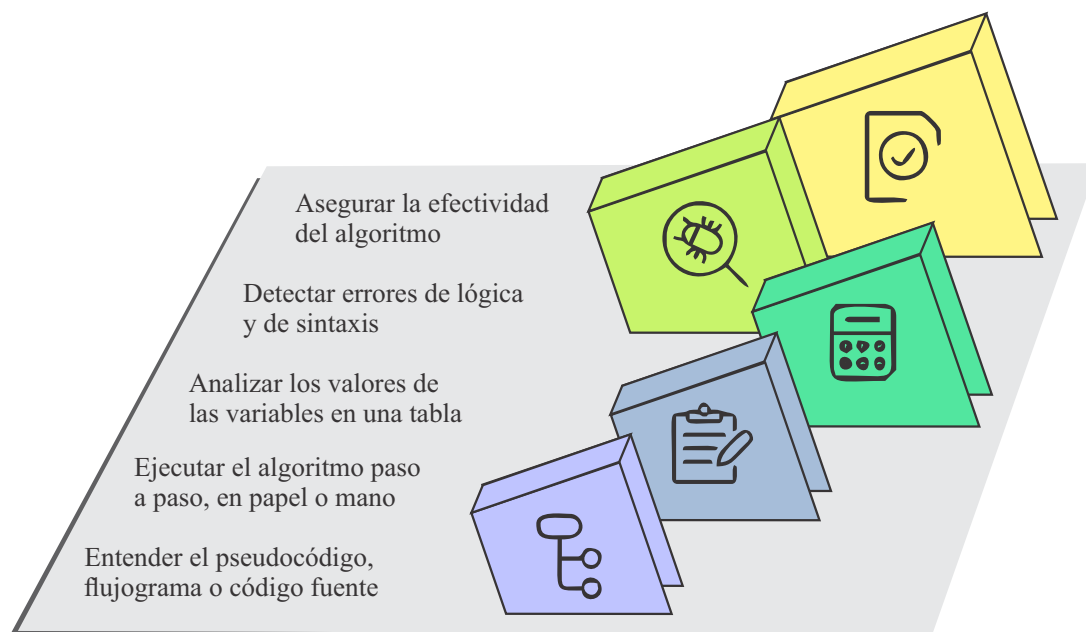


Figura 2.44: Las pruebas de escritorio.

El origen formal de esta técnica se remonta a los años 1940, cuando Goldstine et al. [1947] y Von Neumann & Goldstine [1947], en sus célebres informes sobre el diseño del EDVAC y el IAS, describían detalladamente la necesidad de verificar los diagramas de flujo mediante ejecución manual (lápiz, papel y pizarra) antes de codificarlos en la máquina. En esos documentos ya aparecían tablas de seguimiento idénticas a las que hoy enseñamos en cualquier curso introductorio de programación. Décadas después, a partir de 1970, cuando la programación estructurada luchaba por imponerse, autores como Mills [1972] y Baker [1972] incluyeron explícitamente las pruebas de escritorio dentro de las metodologías de “*inspección de código*” y “*lectura estructurada*”, demostrando empíricamente que un revisor humano siguiendo el algoritmo con lápiz y papel

detectaba entre un 60 % y un 80 % más de defectos que la simple lectura pasiva [Mills, 1976; Linger et al., 1979; Knight & Myers, 1993]. Más tarde, en los años 1980 y 1990, investigadores como Basili & Selby [1987] confirmaron en experimentos controlados que la combinación de pruebas de escritorio con inspecciones formales superaba en relación costo-efectividad a las pruebas funcionales y estructurales tradicionales [Basili, 1997; Selby et al., 1987; Basili & Selby, 1991]. A pesar del auge de los depuradores gráficos, *coverage tools* y *property-based testing*, la literatura científica sigue evidenciando su vigencia. Un estudio de Li et al. [2019] mostró que estudiantes que practicaban pruebas de escritorio sistemáticas antes de escribir código en Python cometían un 43 % menos errores lógicos en exámenes finales que aquellos que dependían exclusivamente del intérprete interactivo.

Otro trabajo de Michaeli & Romeike [2019], publicado en ACM Transactions on Computing Education (indexado en Web of Science), demostró que la habilidad de realizar pruebas de escritorio de forma manual correlaciona fuertemente con la capacidad de comprender y depurar programas ajenos, una competencia fundamental en la industria actual donde la mayoría del tiempo se dedica a mantener código legado [Marwan et al., 2020].

Así, las pruebas de escritorio son una actividad que obliga al programador a razonar con precisión sobre estados, flujo de control y condiciones de borde [Hsu et al., 2018]. Goldstine y von Neumann las necesitaban porque cada ejecución real costaba horas de configuración de cables y tubos de vacío [Goldstine & Von Neumann, 1951]; hoy las necesitamos porque cada desarrollo puede costar millones [Agbo et al., 2019; Tikva & Tambouris, 2021]. El lápiz y la tabla de seguimiento siguen siendo, una de las herramientas más económicas y efectivas para producir software correcto desde la primera versión [Espinal et al., 2024; Buitrago Flórez et al., 2017].

2.2. Estructuras de control

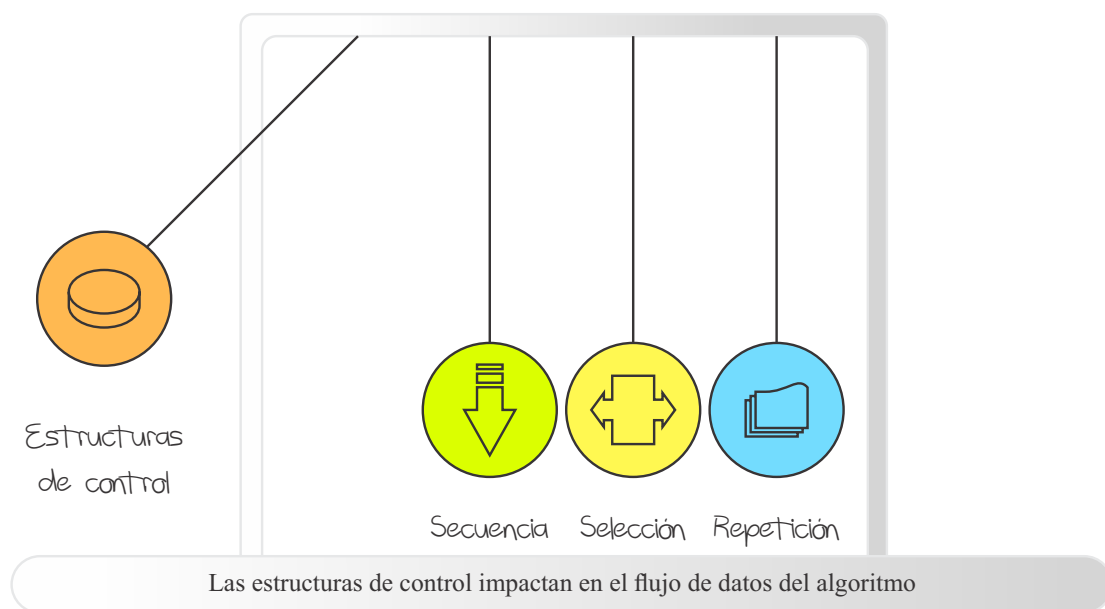


Figura 2.45: Las estructuras de control existentes.

Las estructuras de control son, en sí, aquellas secciones o bloques del algoritmo que permiten dirigir el flujo de datos, que sigue una lógica de solución, ver Fig. 2.45. Böhm & Jacopini [1966] demostraron mediante las matemáticas que todo algoritmo puede elaborarse usando únicamente tres formas de control: la secuencia, la selección y la repetición. Las tres estructuras no son

arbitrarias: reflejan directamente la forma en que los seres humanos razonamos sobre procesos. Ejecutamos pasos uno tras otro (secuencia), tomamos decisiones según condiciones (selección) y repetimos acciones o instrucciones mientras algo sea cierto o hasta alcanzar un objetivo (repetición). Al limitar el flujo de control a estas formas anidadas sin solaparse, se consigue lo que Dijkstra [1968a] llamó, “*programas transparentes*”, donde su comportamiento puede entenderse y demostrarse sin necesidad de simular mentalmente caminos infinitos.

2.2.1. Estructuras secuenciales

La estructura secuencial consiste en ejecutar instrucciones una tras otra, en el orden exacto en que aparecen. No hay bifurcaciones ni bucles: cada instrucción se ejecuta exactamente una vez y el flujo pasa automáticamente a la siguiente. En pseudocódigo se representa simplemente escribiendo las acciones línea por línea. La secuencia de instrucciones es la base de todo algoritmo y sobre la que se construyen las demás estructuras (condicionales y repetitivas).

Sin una ejecución ordenada de pasos, efectiva y predecible, ninguna selección o repetición tendría sentido su uso. A continuación, se presentan un conjunto de ejercicios que demuestran la utilidad de ejecutar las secuencias de instrucciones paso a paso. Además, en los siguientes algoritmos, el uso de las dos barras (//) se los conoce como comentarios, mensajes o anotaciones para comprender la sintaxis de las instrucciones, se irá colocando en comentarios de línea para dejar ideas y buenas prácticas de al momento de programar. La representación gráfica de los algoritmos se los conoce como diagramas de flujo, en ellos se utilizan formas geométricas sencillas para visualizar el flujo o recorrido de los datos, ver Fig. 2.46.

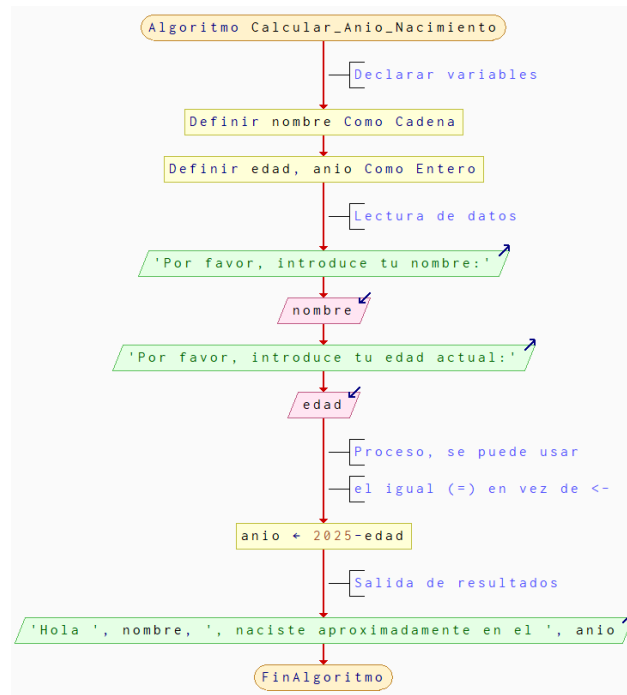


Figura 2.46: Estructura secuencial.

Algoritmo 2.1: Pseudocódigo PSeInt: Cálculo del año de nacimiento

```

1  Algoritmo Calcular_Año_Nacimiento
2  // Declarar variables
3  Definir nombre Como Caracter
4  Definir edad, anio Como Entero
5  // Lectura de datos
6  Escribir "Por favor, introduce tu nombre:"
7  Leer nombre
8  Escribir "Por favor, introduce tu edad actual:"
9  Leer edad
10 // Proceso, se puede usar el igual (=) en vez de <-
11 anio <- 2025 - edad
12 // Salida de resultados
13 Escribir "Hola ", nombre, ", naciste aproximadamente en el ", anio
14 FinAlgoritmo

```

A continuación se presenta un problema para ejemplificar a una estructura secuencial en pseudocódigo y su diagrama de flujo. El área (A) de un círculo se calcula multiplicando el valor de π por el radio (r) elevado al cuadrado.

■ Algoritmo para su revisión en pseudocódigo:

Algoritmo 2.2: Pseudocódigo PSeInt: Área del círculo

```

1 Algoritmo Calcular_Area_Circulo
2   // Declarar variables
3   Definir radio, area Como Real
4   Definir P Como Real
5   // Proceso, en PSeInt PI ya es constante, puedes usar PI en el area
6   P <- 3.14159265
7   Escribir "Introduce el radio:"
8   Leer radio
9   area <- P * (radio * radio)
10  // Salida
11  Escribir "El resultado es: ", area
12 FinAlgoritmo
  
```

Se puede observar con la primera sección (líneas 3 y 4) se definen las variables que intervienen el problema. P es una constante, radio es la variable de entrada y area (sin acento) es la variable de salida, ver Fig. 2.47.

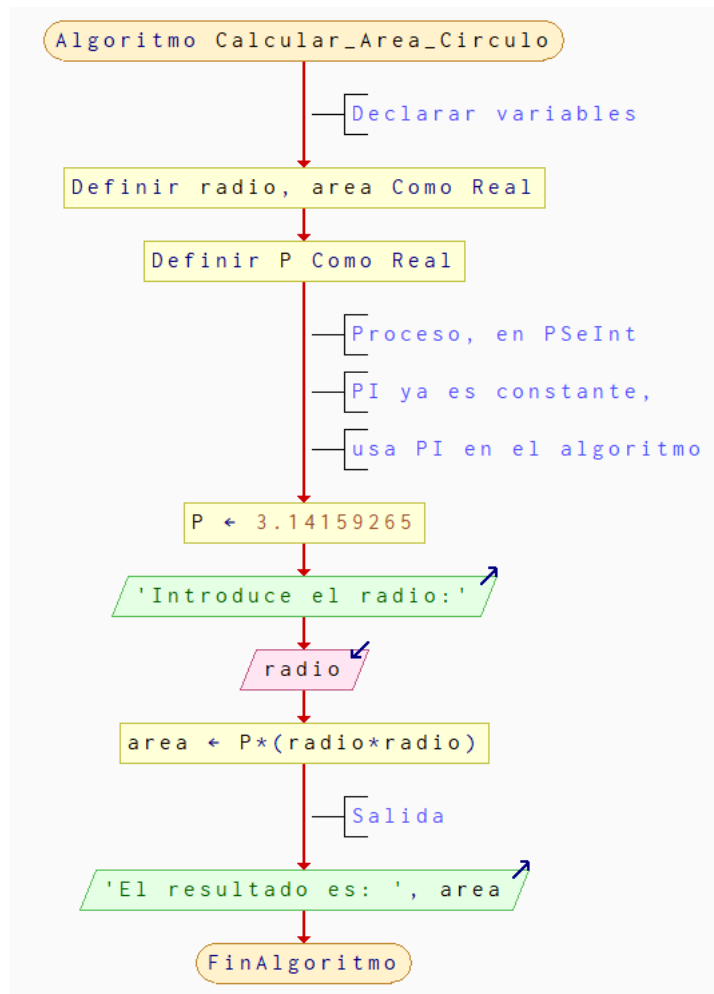


Figura 2.47: Algoritmo que calcula el área del círculo.

Otro ejemplo para representar una estructura secuencial se muestra a continuación. El área de un hexágono usando solo el lado (L). Este algoritmo es práctico entenderlo, ya que solo pide la longitud de un lado como variable de entrada. Utiliza la constante $\sqrt{3} \approx 1,73205$ para evitar el uso de una función raíz o usar la función `raiz(x)` que trae PSeInt.

 Algoritmo para su revisión en pseudocódigo:

Algoritmo 2.3: Pseudocódigo PSeInt: Área de la figura regular usando solo el lado

```

1 Algoritmo Calcular_Area_Hexagono_Lado
2   // Declarar variables. Se usan 'Real' para decimales.
3   Definir lado, area, raiz_tres Como Real
4   // Asignar el valor de la constante Pi o usar PI
5   // Nota: 1.73205 es aproximada del radicando de 3.
6   raiz_tres <- 1.73205
7   // Lectura de datos
8   Escribir "Introduce la longitud del lado (L) de la figura regular:"
9   Leer lado
10  // Proceso: A = (3 * raiz(3) / 2) * L^2
11  // Se traduce: (3 * raiz_tres * (lado * lado)) / 2
12  // Se recomienda calcular (lado * lado)
13  // antes de multiplicar por 3 * raiz_tres
14  area <- (3 * raiz_tres * (lado * lado)) / 2
15  // Salida de resultados
16  Escribir "El resultado es: ", area
17 FinAlgoritmo

```

Este algoritmo define tres nombres para dos variables y una constante. Se observa que las variables llamadas `lado` y `area` (sin tilde) son las variables de entrada y salida, mientras `raiz_tres` es la constante visible en la sección de declarar variables. En la sección de lectura de datos se evidencia el mensaje y la lectura del lado. Seguidamente, se encuentra la sección del proceso donde la línea 14 se presenta la fórmula utilizada en el algoritmo. Finalmente, la sección de salida de resultados se presenta la variable `area` (sin tilde) para mostrar el resultado. A continuación, se presenta el algoritmo en su representación de diagrama de flujo, ver Fig. 2.48.

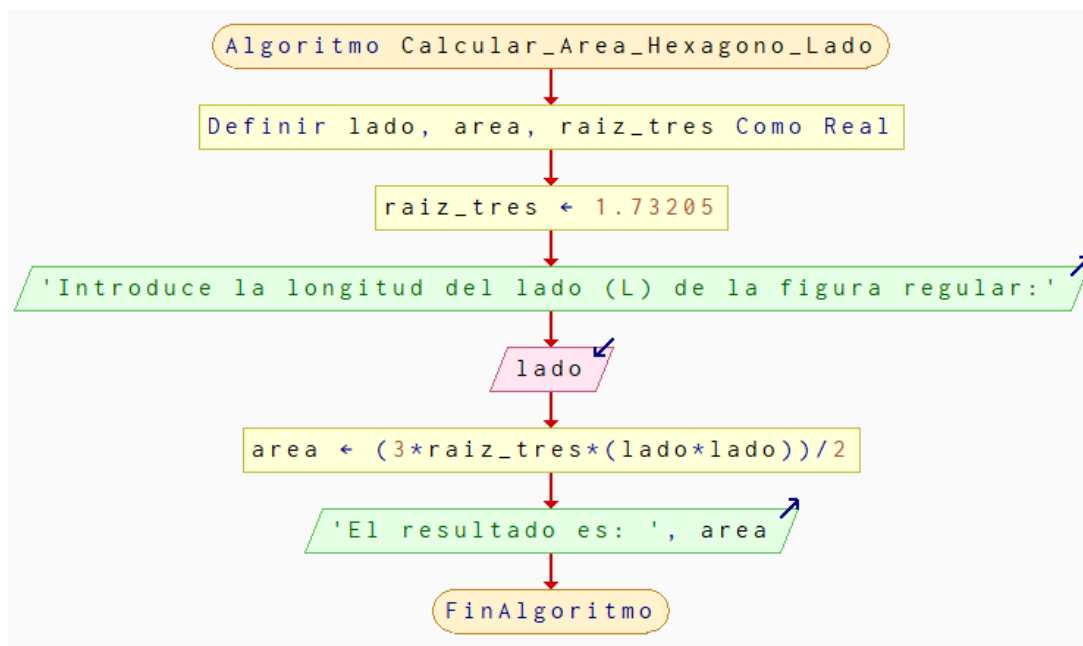


Figura 2.48: Algoritmo que calcula el área del hexágono usando el lado.

Otro ejemplo para representar una estructura secuencial se muestra a continuación. Por ejemplo, si solicitan el área de un Hexágono usando el lado (L) y apotema (a). Este algoritmo sigue la fórmula general para polígonos regulares: $A = (P \cdot a)/2$.

■ Algoritmo para su revisión en pseudocódigo:

Algoritmo 2.4: Pseudocódigo PSeInt: Área de hexágono regular (lado y apotema)

```

1  Algoritmo Calcular_Area_Hexagono_Apotema
2  // Declarar variables. Se usan 'Real' para decimales.
3  Definir lado, apotema, perimetro, area Como Real
4  // Lectura de datos
5  Escribir "Introduce la longitud de la figura regular:"
6  Leer lado
7  Escribir "Introduce la longitud del apotema:"
8  Leer apotema
9  // Proceso:
10 // 1. Calcular la periferia:  $P = 6 * \text{Lado}$ 
11 perimetro  $\leftarrow 6 * \text{lado}$ 
12 // 2. Calcular la superficie:  $A = (P * \text{Apotema}) / 2$ 
13 area  $\leftarrow (\text{perimetro} * \text{apotema}) / 2$ 
14 // Salida de resultados
15 Escribir "La superficie de la figura regular es: ", area
16 FinAlgoritmo

```

Se puede observar que todas las fórmulas utilizadas en la sección del proceso (en los algoritmos) deben escribirse de manera lineal (utilizando los paréntesis y respetando la jerarquía de los operadores). Además, las variables pueden escribirse usando letras y números, pero evita caracteres especiales, letras especiales y el uso de la tilde o acento en las vocales. A continuación, se presenta el algoritmo en su representación de diagrama de flujo, ver Fig. 2.49.

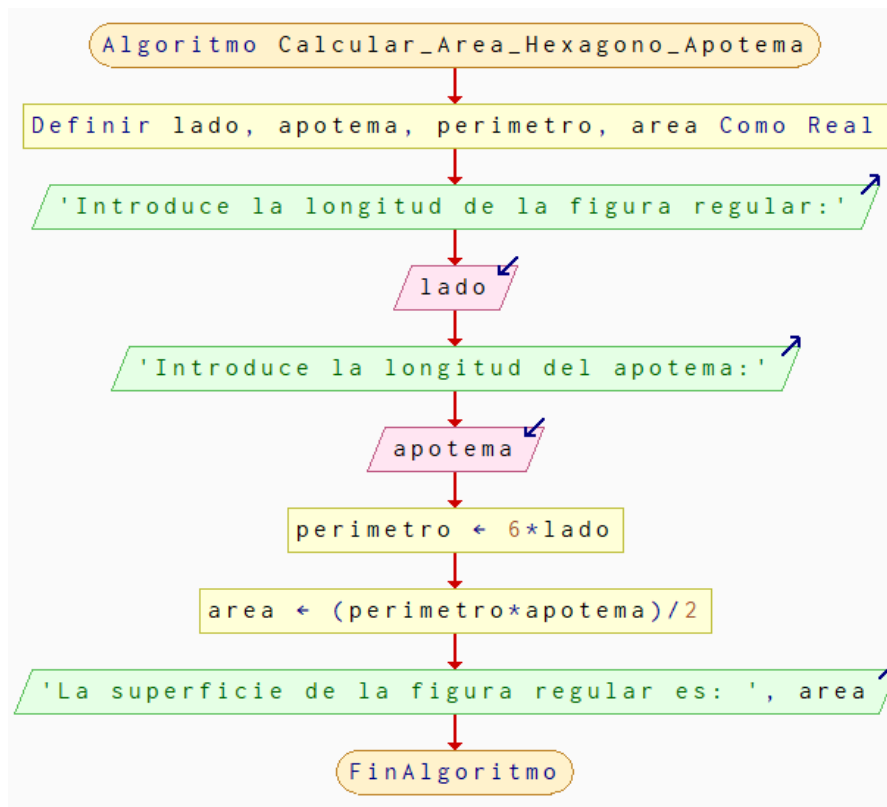


Figura 2.49: Algoritmo que calcula el área del hexágono usando el lado y el apotema.

2.2.2. Estructuras condicionales (Decisión simple)

Las estructuras condicionales permiten que el flujo de datos elija entre uno, dos o más caminos según el valor de una expresión lógica. Las formas existentes son condicional simple, condicional doble y condicional múltiple. El propósito de estas estructuras es modelar decisiones del mundo real, por ejemplo, “si llueve, llevo paraguas; si no, llevo gafas de sol; en cualquier otro caso, voy sin nada”. El teorema de Böhm & Jacopini [1966] demuestra que una sola construcción si-entonces-si_no (if-then-else) basta para expresar cualquier selección, pero los lenguajes modernos ofrecen variantes más legibles y seguras. Un error común entre programadores principiantes es anidar demasiados if-else creando el llamado “infierno de flechas”; la solución pasa por usar estructuras anidadas verticales (elseif), tablas de decisión o, en casos complejos, patrones como polimorfismo.

Algoritmo 2.5: Pseudocódigo PSeInt: Verificación de mayoría de edad

```

1  Algoritmo Verificar_Edad
2      // Declarar variables
3      Definir edad Como Entero
4      // Lectura de datos
5      Escribir "Introduce tu edad actual:"
6      Leer edad
7      // Estructura Selectiva Simple (Si-Entonces)
8      Si edad >= 18 Entonces
9          Escribir "Eres mayor de edad por eso tienes acceso."
10     FinSi
11 FinAlgoritmo

```

Como se observa en el pseudocódigo anterior, la sección de salida se encuentra dentro de la estructura selectiva simple. Además, esta estructura solo tiene habilitado la opción por VERDADERO, es decir, solo interesa enviar un mensaje si es mayor e igual a 18, caso contrario no hay mensaje alguno. A continuación, se presenta el algoritmo en su representación de diagrama de flujo, ver Fig. 2.50.

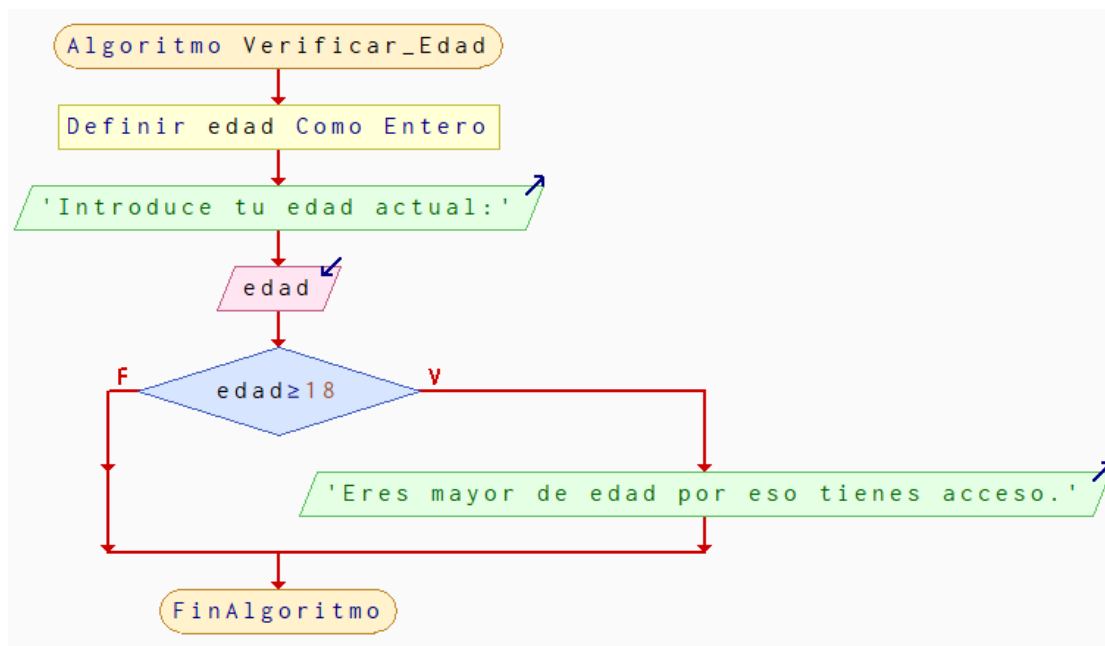


Figura 2.50: Algoritmo que verifica si la edad de una persona es mayor e igual a 18.

2.2.3. Estructuras condicionales (Decisión doble)

Un ejercicio de una estructura doble, puede ser este ejemplo. Un algoritmo que pide la calificación de un estudiante y determina si está Aprobado (nota ≥ 70) o Reprobado (nota < 70).

Algoritmo para su revisión en pseudocódigo:

Algoritmo 2.6: Pseudocódigo PSeInt: Evaluación de calificación

```

1 Algoritmo Evaluar_Aprobacion
2   // Declarar variables
3   Definir calificacion Como Entero
4   // Lectura de datos
5   Escribir "Introduce la nota del estudiante (0-100):"
6   Leer calificacion
7   // Estructura Selectiva Doble (Si-Entonces-Sino)
8   Si calificacion  $\geq 70$  Entonces
9     // Bloque VERDADERO
10    Escribir "Felicidades, el estudiante ha APROBADO."
11  Sino
12    // Bloque FALSO
13    Escribir "El estudiante ha REPROBADO por eso debe repetir."
14  FinSi
15  Escribir "Fin."
16 FinAlgoritmo

```

Es indiscutible que con la llegada de la estructura condicional, en la línea 8, se presenta el uso de operadores de relación. Los bloques de código que podemos obtener dependerán si el resultado de la condición es VERDADERO, o es FALSO. Así, la lógica del flujo de los datos irá dependiendo del resultado de evaluar la decisión. Si es mayor e igual a 70, entonces la respuesta por verdadero realizará el conjunto de instrucciones por verdadero. En caso contrario, si es falso, se ejecutará el conjunto de instrucciones o bloque por falso. Para el diagrama de flujo, la decisión toma una forma de rombo, las entradas y salidas de forma de paralelogramo (con flecha de ingreso o de salida a la figura respectivamente) y los procesos con un rectángulo. A continuación, se presenta el algoritmo en su representación de diagrama de flujo, ver Fig. 2.51.

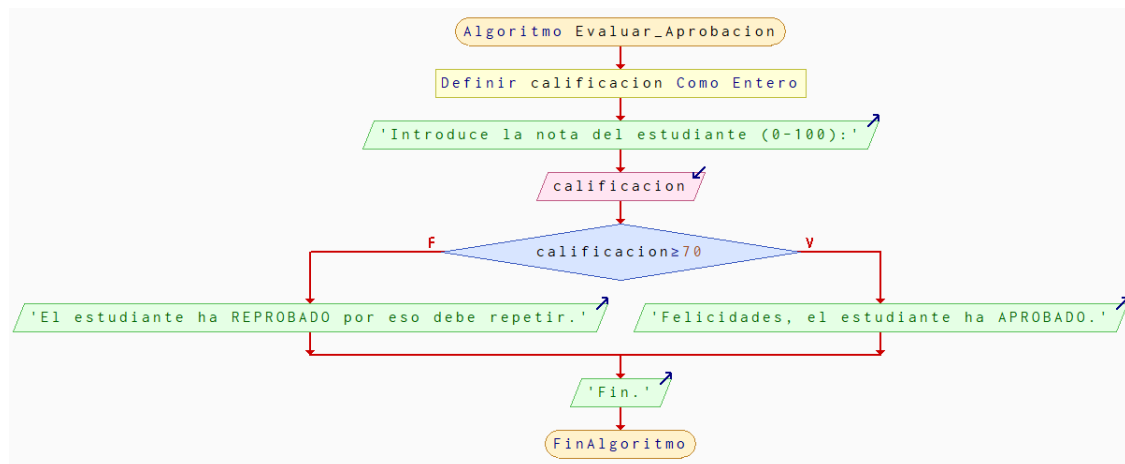


Figura 2.51: Algoritmo que verifica si la calificación es mayor e igual a 70.

Otro ejercicio para ejemplificar un algoritmo con estructura doble se presenta a continuación. Se solicita un algoritmo que verifique si un número es par o impar, para eso ingresamos un número por teclado. Luego utilizamos una operación matemática para determinar el residuo de un número.

Algoritmo para su revisión en pseudocódigo:

Algoritmo 2.7: Pseudocódigo PSeInt: Detección de valor par o impar

```

1 Algoritmo Verificar_Par_Impar
2   // Declarar variables
3   Definir numero Como Entero
4   // Lectura de datos
5   Escribir "Introduce un valor:"
6   Leer numero
7   // Estructura Selectiva Doble (Si-Entonces-Sino)
8   // Usamos el operador MOD (residuo) para obtener
9   // el residuo de dividir por 2.
10  // El residuo al ser 0, el valor es par.
11  Si (numero MOD 2) = 0 Entonces
12    // Bloque VERDADERO: Residuo es 0
13    Escribir "El valor ", numero, " es PAR."
14  Sino
15    // Bloque FALSO: Residuo es 1
16    Escribir "El valor ", numero, " es IMPAR."
17  FinSi
18  Escribir "Verificado."
19 FinAlgoritmo

```

Estructura Selectiva

En el pseudocódigo se puede observar, la estructura selectiva doble, con el uso de la condición. Esta estructura tiene dos bloques, uno por verdadero y otro por falso. En la condición se usa el operador MOD que es el valor del residuo. Si el residuo es cero significa que el dividendo es divisible para el divisor (valor de 2) y por esta razón el número (dividendo) al dividir para 2 y su residuo al ser cero podemos entonces decir que el número es par. A continuación, se presenta el algoritmo en su representación de diagrama de flujo, ver Fig. 2.52.

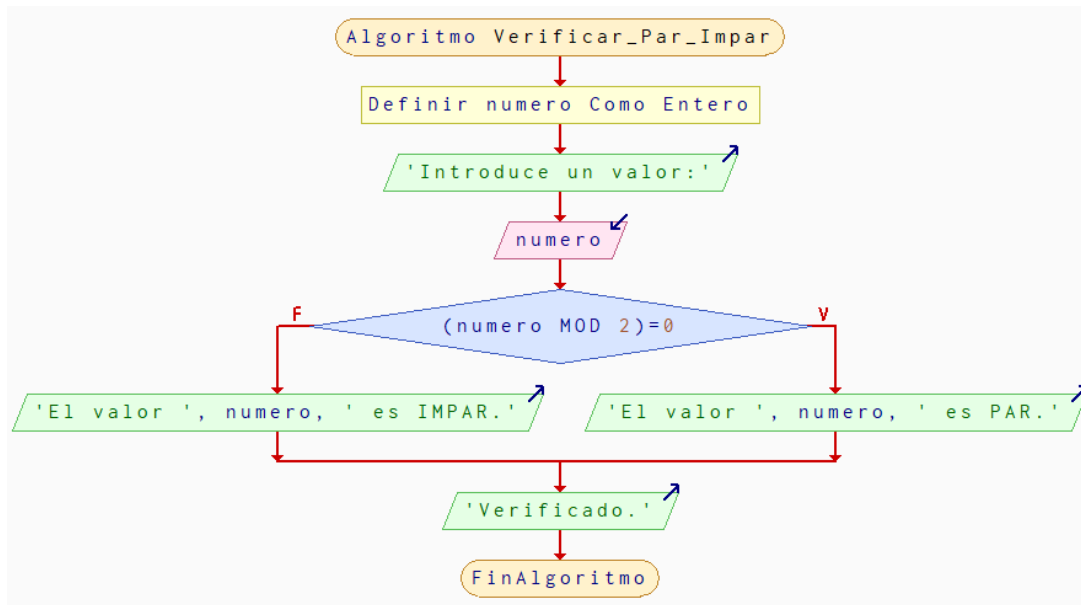


Figura 2.52: Algoritmo que verifica si un número ingresado es par o impar.

A continuación se presenta un problema para ejemplificar a una estructura selectiva anidada. En este ejercicio el uso de rangos es determinante.

Clasificación de edad usando estructuras de control, selectiva y anidada

Elabore un algoritmo que permita el ingreso de un valor entero que represente la edad de una persona. Si se ingresa un valor menor a cero o mayor a 100, se muestra un mensaje de “Incorrecto”. Si se ingresa un valor igual a cero se muestra un mensaje de “Nene” Si se ingresa un valor mayor a cero y menor a doce se muestra un mensaje de “Es infante/a” Si se ingresa un valor mayor igual a doce y menor a dieciocho se muestra un mensaje de “Es joven” Si se ingresa un valor mayor igual a dieciocho y menor a sesenta se muestra un mensaje de “Es adulto” Si se ingresa un valor mayor igual a sesenta y menor igual a cien se muestra un mensaje de “Es adulto mayor”

La primera solución, siguiendo tal como indica el problema, con una situación de no considerar optimización, y posible redundancia, sería:

Algoritmo 2.8: Pseudocódigo PSEInt: Clasificación de edad con anidamiento simple y doble

```

1  Algoritmo Clasificar_Edad_Anidada
2      // Declarar variables
3      Definir edad Como Entero
4      // Lectura de datos
5      Escribir "Introduce un valor que represente la edad:"
6      Leer edad
7      // Estructura anidada principal para manejar todos los casos
8      Si edad < 0 O edad > 100 Entonces
9          // 1. Requisito de Error (Rango principal, condicionas el error)
10         Escribir "ERROR: Valor Incorrecto. La edad debe estar de 0 a 100."
11     Sino
12         // 2. Al ser la edad verdadera, clasificamos
13         // Estructura Selectiva SIMPLE
14         Si edad = 0 Entonces
15             Escribir "Es un nene"
16         Sino
17             // 3. Clasificamos
18             // Estructura Selectiva DOBLE pero puede ser SIMPLE
19             Si edad > 0 Y edad < 12 Entonces
20                 // Implica edad > 0, no es necesario
21                 Escribir "Es infante/a"
22             Sino
23                 // 4. Estructura Selectiva DOBLE
24                 Si edad >= 12 Y edad < 18 Entonces
25                     Escribir "Es joven"
26                 Sino
27                     // 5. Estructura Selectiva DOBLE
28                     Si edad >= 18 Y edad < 60 Entonces
29                         Escribir "Es adulto"
30                     Sino
31                         // 6. Caso final para (60 a 100)
32                         // Estructura Selectiva DOBLE solo con VERDADERO
33                         Si edad >= 60 Y edad <= 100 Entonces
34                             Escribir "Es adulto mayor"
35                         FinSi // Fin del Si 6
36                     FinSi // Fin del Si 5
37                 FinSi // Fin del Si 4
38             FinSi // Fin del Si 3
39         FinSi // Fin del Si 2
40     FinSi // Fin del Si 1 (principal)
41     Escribir "Fin del programa."
42 FinAlgoritmo

```

De acuerdo con el algoritmo anterior es visible como en la línea 33 (el último sí) es notable que solo queda la alternativa por verdadero, además en la línea 19 puede no usarse una condición, ya que se sobreentiende o ya se implica.

A tener en cuenta en la clasificación de edad usando estructuras de control

Problema: Elabore un algoritmo que permita el ingreso de un valor entero que represente la edad de una persona. Si se ingresa un valor menor a cero o mayor a 100, se muestra un mensaje de “Incorrecto”. Si se ingresa un valor igual a cero se muestra un mensaje de “Nene” Si se ingresa un valor mayor a cero y menor a doce se muestra un mensaje de “Es infante/a” Si se ingresa un valor mayor igual a doce y menor a dieciocho se muestra un mensaje de “Es joven” Si se ingresa un valor mayor igual a dieciocho y menor a sesenta se muestra un mensaje de “Es adulto” Si se ingresa un valor mayor igual a sesenta y menor igual a cien se muestra un mensaje de “Es adulto mayor”

Variable de entrada:

- **Edad (E o $edad$):** Variable de valor entero que representa la edad de una persona.

Consideración de rangos a usar ver Tabla 2.3:

Tabla 2.3: Rangos de edad y clasificación

Rango de edad	Condición lógica	Mensaje de salida
Inválido	$E < 0 \vee E > 100$	Incorrecto
Cero	$E = 0$	Bebé / Nene
Infantil	$E > 0 \wedge E < 12$	Es niño/a infante/a
Adolescente	$12 \leq E < 18$	Es joven
Adulto	$18 \leq E < 60$	Es adulto
Adulto Mayor	$60 \leq E \leq 100$	Es adulto mayor

Prueba de escritorio ver Tabla 2.4:

Tabla 2.4: Valores para los rangos de edad y clasificación

Valor a utilizar	$E < 0 \vee E > 100$	$E = 0$	$E > 0 \wedge E < 12$	Mensaje de salida
200	Verdadero			Incorrecto
Cero		Verdadero		Bebé / Nene
5			Verdadero	Es niño/a infante/a

En el diagrama de flujo, se muestra cómo con el uso de varias figuras geométricas sencillas, entre ellas, los paralelogramos, permiten ser útiles para los mensajes y variables de entrada. Por ejemplo, se muestra a continuación, cómo se solicita la variable edad, utilizando un paralelogramo con flecha hacia dentro y de color rosa. Mientras que los paralelogramos, que se utilizan para los mensajes, tienen la flecha hacia afuera y son de color verde. Además, todas las decisiones se concentran en rombos, de color azul. Estos rombos presentan el flujo que deben seguir los datos, si el resultado de la decisión es verdadero (V) o es (F). Es decir, que si una estructura es anidada, significa que en una de las opciones (V o F) se vuelve a colocar una estructura similar a la anterior. Así, se van anidando estructuras hasta llegar a un valor final.

El uso de estructuras anidadas debe usarse con cuidado, tratando de seguir un patrón de lógica para colocar los operadores de decisión. Si se observa la siguiente imagen, podrás notar el uso de los operadores de relación mayores y menores, así como el uso del igual en ellos y sin ellos. Además, puedes notar que se adjuntan con ellos los operadores lógicos, tanto el Y como el O, y que su uso mejora la comprensión de los rangos. El uso práctico de las decisiones, permiten mantener una lógica comprensiva de clasificar un número, yendo de un número menor a otro mayor, así como usar la recta numérica para establecer límites, y si estos se incluyen o no. Finalmente, es recomendable dibujar, estableciendo un color a los límites, el límite con un punto oscuro (si se incluye el número) y un punto sin color (si no se incluye), de acuerdo con esto podrás saber si se usa un operador de relación u otro.

A continuación, se presenta el algoritmo en su representación de diagrama de flujo, clasifica la edad según el rango y la ubica donde pertenece, a continuación presenta el mensaje correspondiente, ver Fig. 2.53.

Algoritmo para su revisión en pseudocódigo:

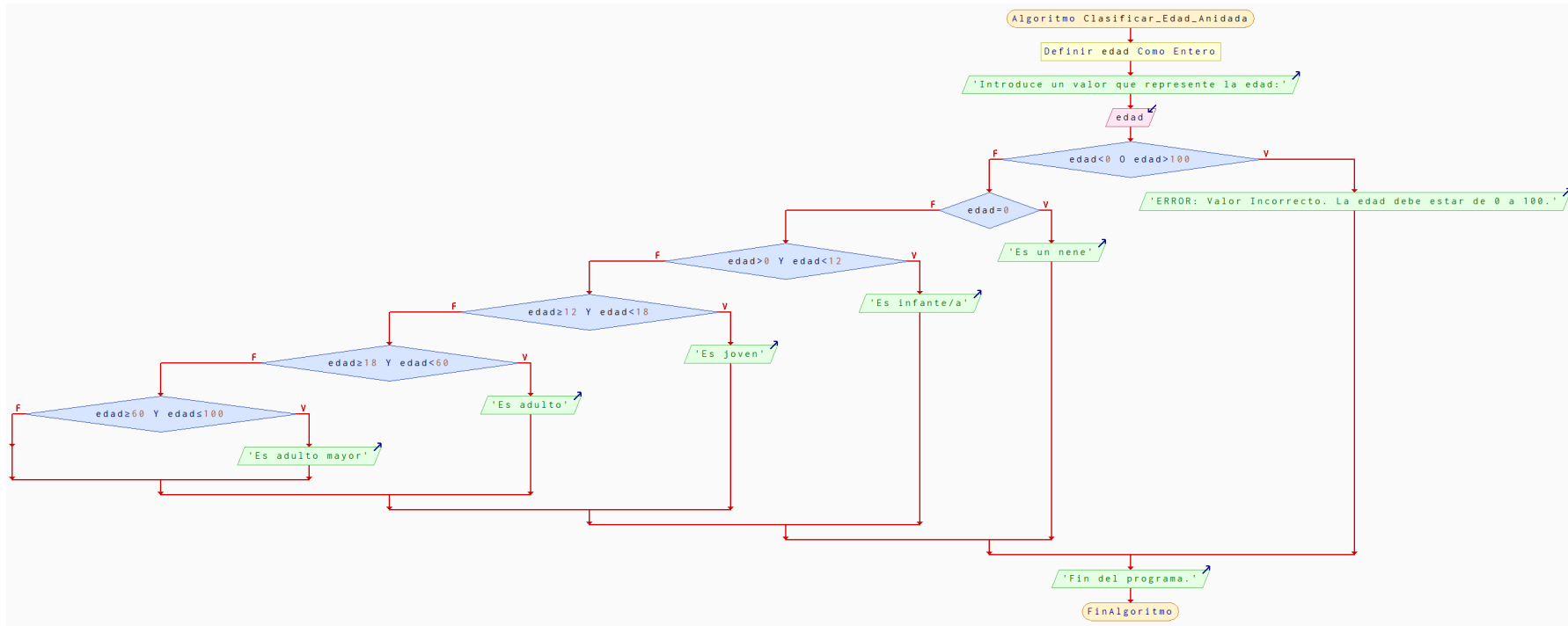


Figura 2.53: Algoritmo que clasifica la edad según el rango.

En la imagen anterior debe observarse que el autor ha decidido anidar las estructuras por el valor falso de la decisión. Y emitir el mensaje por verdadero. También es visible que en las decisiones, las expresiones de la derecha del operador lógico, son aquellos que se encuentran de lado izquierdo en la recta numérica, mientras los que están de lado derecho, son aquellos valores de mayor valor.

Una segunda versión mejorando su escritura.

I Algoritmo para su análisis, revisión y comparación:

Algoritmo 2.9: Pseudocódigo PSeInt: Clasificación de etapas de la vida (Selectiva Múltiple)

```

1 Algoritmo Clasificar_Edad_Anidada
2   // Declarar variables
3   Definir edad Como Entero
4   // Lectura de datos
5   Escribir "Introduce un valor que represente la edad:"
6   Leer edad
7   // Estructura Selectiva Multi (Si-SinoSi)
8   Si edad < 0 O edad > 100 Entonces
9     // 1. Error
10    Escribir "ERROR: Valor Incorrecto. La edad debe estar de 0 a 100."
11  Sino Si edad = 0 Entonces
12    // 2. Nene
13    Escribir "Nene"
14  Sino Si edad > 0 Y edad < 12 Entonces
15    // 3. Infante/a
16    Escribir "Es infante/a"
17  Sino Si edad >= 12 Y edad < 18 Entonces
18    // 4. Joven
19    Escribir "Es joven"
20  Sino Si edad >= 18 Y edad < 60 Entonces
21    // 5. Adulto
22    Escribir "Es adulto"
23  Sino // Esto cubre el rango final identificado (>= 60 y <= 100)
24    // 6. Adulto Mayor
25    Escribir "Es adulto mayor"
26  FinSi
27  Escribir "Fin del programa."
28 FinAlgoritmo

```

Ahora te presentamos una tercera versión mejorada optimizando las condiciones implicando o sobreentendiendo rangos.

I Algoritmo para su análisis, revisión y comparación:

Algoritmo 2.10: Pseudocódigo PSeInt: Clasificación de etapas con Si-Sino Si (Mejor para rangos)

```

1 Algoritmo Clasificar_Edad_SiSinoSi
2   // Declarar variables
3   Definir edad Como Entero
4   // Lectura de datos
5   Escribir "Introduce un valor que represente la edad:"
6   Leer edad
7   // Selectiva Multi de Rango (Evaluar varias condiciones)
8   Si edad < 0 O edad > 100 Entonces
9     // 1. Se verifica el Error (Invar)
10    Escribir "ERROR: Valor Incorrecto. La edad debe estar entre 0 y 100."
11  Sino Si edad = 0 Entonces
12    // 2. Caso especial
13    Escribir "Nene"
14  Sino Si edad < 12 Entonces // Implica: edad > 0
15    // 3. Rango
16    Escribir "Es infante/a"
17  Sino Si edad < 18 Entonces // Implica: edad >= 12
18    // 4. Rango

```

```

19     Escribir "Es joven"
20 Sino Si edad < 60 Entonces // Implica: edad >= 18
21     // 5. Rango
22     Escribir "Es adulto"
23 Sino // Implica: edad >= 60 Y edad <= 100
24     // 6. Rango Final
25     Escribir "Es adulto mayor"
26 FinSi
27 Escribir "Fin del programa."
28 FinAlgoritmo

```

Finalmente, la estructura selectiva múltiple con “Según” es menos recomendada para rangos. El comando Según (switch) en PSeInt está diseñado para manejar un número limitado de valores exactos (discretos), no rangos. Para clasificar por rangos, tendríamos que usar un arreglo o funciones complejas o, como se mostró, anidar el “Según” dentro de un Si-Sino Si, lo que lo vuelve menos eficiente que la solución anterior. Por lo tanto, la forma didácticamente correcta de mostrar “Según” es clasificando previamente la edad en una variable que represente la etapa, o usándolo para un problema distinto. Para el problema de la edad, la única forma de implementarlo directamente es listar cada valor discreto dentro de cada rango, lo cual sería impracticable (listando edades hasta 100 casos: 12, 13, 14, ..., 57, 58, ...).

Vamos profundizando

Utiliza Si-Sino Si para la mayoría de los problemas que involucren rangos numéricos, como edades, notas o precios. Utiliza “Según” cuando la variable tiene un conjunto limitado y conocido de valores exactos (ej: día de la semana, opción de menú, código de estado).

Un ejemplo usando “Según” donde se presentan los días de la semana según el ingreso numérico, valor entero, correspondiente al día.

Algoritmo 2.11: Pseudocódigo PSeInt: Uso de la estructura Según

```

1 Algoritmo Mostrar_Dia_Semana
2     // Declarar variables
3     Definir dia Como Entero
4     Escribir "Introduce un valor del 1 al 7:"
5     Leer dia
6     // Selectiva Multi por valor discreto
7     Segun dia Hacer
8         1: Escribir "Monday"
9         2: Escribir "Tuesday"
10        3: Escribir "Wednesday"
11        4: Escribir "Thursday"
12        5: Escribir "Friday"
13        6: Escribir "Saturday"
14        7: Escribir "Sunday"
15        De Otro Modo:
16            Escribir "Valor incorrecto."
17 FinSegun
18 FinAlgoritmo

```

Ahora se presenta un algoritmo que simula un cajero automático simple, donde la acción (depositar, retirar, consultar) depende de la opción exacta que ingrese el usuario, utilizando para ellos un número entero del 1 al 4.

Algoritmo 2.12: Pseudocódigo PSeInt: Menú de opciones con la estructura Según

```

1 Algoritmo Cajero_Automatico

```

```

2 // Declarar variables
3 Definir opcion Como Entero
4 Definir saldo Como Real
5 // Inicializar saldo
6 saldo <- 1000.00
7 // Mostrar servicios, elegir una
8 Escribir "--- CAJERO AUTOMATIC ---"
9 Escribir "1. Consultar Saldo"
10 Escribir "2. Depositar"
11 Escribir "3. Retirar"
12 Escribir "4. Salir"
13 Escribir "-----"
14 Escribir "Seleccione una (1-4):"
15 Leer opcion
16 // Estructura Selectiva Multi:
17 Segun opcion Hacer
18 1:
19     Escribir "Su saldo actual es: ", saldo
20 2:
21     Escribir "Elegiste 'Depositar' seleccionada."
22     // Se ejecuta la idea de depositar
23 3:
24     Escribir "Elegiste 'Retirar' seleccionada."
25     // Se ejecutar la idea de retiro
26 4:
27     Escribir "Gracias por usar nuestro cajero. Vuelva pronto"
28 De Otro Modo:
29     // Caso por defecto (cualquier valor)
30     Escribir "ERROR. Intente nuevamente."
31 FinSegun
32 FinAlgoritmo

```

Se presenta la tabla de los operadores matemáticos, relacionales y lógicos, comunes y útiles, para el uso en el desarrollo de los algoritmos. Se presenta en el contenido de la tabla, el operador, el nombre, un ejemplo y su resultado o interpretación, ver Tabla 2.5.

Tabla 2.5: Operadores aritméticos, de relación y lógicos

Tipo	Operador	Nombre	Ejemplo	Resultado / Interpretación
Aritmético	+	Suma	8 + 5	13
	-	Resta	20 - 7	13
	*	Multiplicación	6 * 9	54
	/	División (real)	15 / 4	3.75
	% ó MOD	Módulo (resto)	15 mod 4	3
	^	Potencia	5^3	125
Relacional	==	Igual que	edad == 18	Verdadero
	<>	Diferente de	nota <> 0	Verdadero
	<	Menor que	x < 10	Verdadero si x es 5
	>	Mayor que	y > 100	Verdadero si y es 150
	<=	Menor o igual	puntaje <= 60	Verdadero si puntaje es 60
	>=	Mayor o igual	edad >= 18	Si es mayor e igual a 18
Lógico	& ó Y	Y lógico (AND)	edad >= 18 Y edad <= 65	Solo si ambas son ciertas
	ó O	O lógico (OR)	día = 6 O día = 7	Verdadero si es fin de semana
	~ ó NO	Negación (NOT)	no(x < 0)	Verdad si x es positivo o cero

Cada uno de los operadores matemáticos, cumple con su función, es decir, matemáticamente

el resultado a obtener será el que dependiendo del tipo de dato (entero o real) que se ingrese y con esto, el valor se presentará en pantalla. Asimismo, los operadores relacionales, evalúan el valor o las expresiones que se encuentran de lado izquierdo y derecho de ellos. Finalmente, los operadores lógicos, o conocidos como booleanos, permiten determinar el estado lógico de verdadero o falso según el resultado final de la expresión. Dependiendo del programa o entorno de escritura, del algoritmo o ambiente de desarrollo, se utiliza el operador de acuerdo con su sintaxis.

Gracias a estos operadores, un bloque de código o una parte del algoritmo puede escribirse de diversas maneras para alcanzar el mismo resultado. La combinación de valores límite, operadores de relación y operadores lógicos, junto con la aplicación de las leyes de la teoría de proposiciones y el álgebra de Boole, nos permite una extensa combinación de estructuras lógicas equivalentes, ver Tabla 2.6.

Soluciones lógicas equivalentes en el rango [0, 24]

La misma verificación puede ser escrita de diez maneras diferentes:

Tabla 2.6: Todos los códigos son lógicamente equivalentes y producen el mismo resultado.

Soluciones directas (validación ✓)		Soluciones con negación (Lógica inversa)	
1.	Si (hora ≥ 0) y (hora ≤ 24) Escribir ("Buen ingreso") Sino Escribir ("Mal ingreso")	6.	Si (hora < 0) ó (hora > 24) Escribir ("Mal ingreso") Sino Escribir ("Buen ingreso")
2.	Si (hora > -1) y (hora < 25) Escribir ("Buen ingreso") Sino Escribir ("Mal ingreso")	7.	Si negado ((hora ≥ 0) y (hora ≤ 24)) Escribir ("Mal ingreso") Sino Escribir ("Buen ingreso")
3.	Si negado ((hora > -1) y (hora < 25)) Escribir ("Mal ingreso") Sino Escribir ("Buen ingreso")	8.	Si negado ((hora < 0) ó (hora > 24)) Escribir ("Buen ingreso") Sino Escribir ("Mal ingreso")
4.	Si (negado (hora > -1) ó negado (hora < 25)) Escribir ("Mal ingreso") Sino Escribir ("Buen ingreso")	9.	Si (negado (hora < 0) y negado (hora > 24)) Escribir ("Buen ingreso") Sino Escribir ("Mal ingreso")
5.	Si ((hora ≤ -1) ó (hora ≥ 25)) Escribir ("Mal ingreso") Sino Escribir ("Buen ingreso")	10.	Si (hora > 0) y (hora ≤ 24) ó (hora = 0) Escribir ("Buen ingreso") Sino Escribir ("Mal ingreso")

Podemos observar que del conjunto de diez bloques de pseudocódigo, se demuestra la flexibilidad lógica para validar el rango de la variable hora entre [0, 24], siendo todos lógicamente equivalentes. La validación directa, que se muestran en los bloques 1 y 2, representa la forma más legible comprobando si $0 \leq \text{hora} \leq 24$. El bloque 10 (la versión extendida $(\text{hora} > 0 \wedge \text{hora} \leq 24) \vee (\text{hora} = 0)$) es una alternativa prolija que produce el mismo resultado. La validación inversa, representada en los bloques 5 y 6, comprueba si la hora es inválida ($\text{hora} < 0 \vee \text{hora} > 24$), invirtiendo la lógica de salida para que el bloque "Sino" ejecute el "Buen ingreso". Finalmente, los bloques 3, 4, 7, 8, y 9 ilustran la aplicación de las "Leyes de De Morgan", mostrando cómo la palabra clave negado y la inversión de los operadores ($y \leftrightarrow \text{ó}$) permiten expresar la misma condición de múltiples maneras, como se ve en el bloque 7 en $\neg((\text{hora} \geq 0) \wedge (\text{hora} \leq 24))$.

2.2.4. Estructuras repetitivas (Bucles)

Las estructuras repetitivas, de repetición o, también llamadas bucles o ciclos, son instrucciones que permiten ejecutar un bloque de código varias veces de forma automática, sin tener que escribirlo, varias veces lo mismo, o repetidamente. Son fundamentales para evitar código duplicado y para procesar listas, hacer conteos, validaciones, en otras palabras automatizar, ver Fig. 2.54.

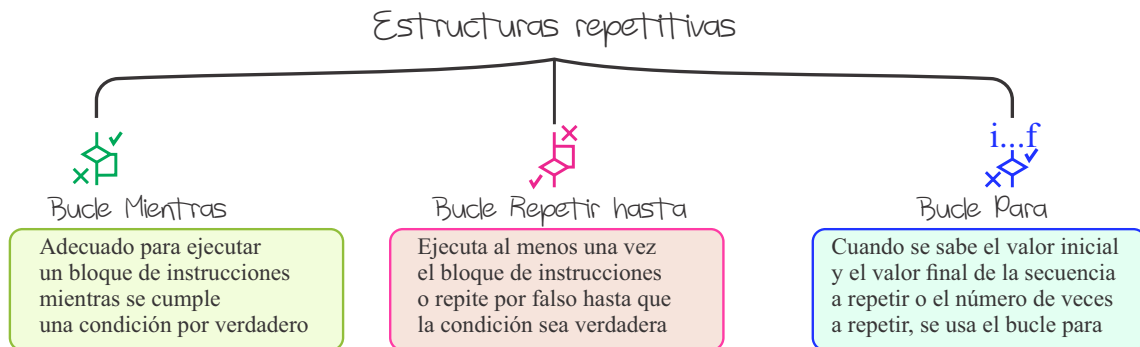


Figura 2.54: Estructuras repetitivas iniciales.

Bucle Mientras

Esta estructura ejecuta el bloque de instrucciones, mientras una condición sea verdadera. La condición se evalúa al principio del ciclo. Si la condición es falsa desde el inicio, las instrucciones no se ejecutan ni una sola vez. Se presenta a continuación el algoritmo para generar la tabla de multiplicar. Este algoritmo permite comprender la estructura de repetición “*Mientras*”.

Algoritmo 2.13: Pseudocódigo PSeInt: Generar tabla de multiplicar

```

1  Algoritmo Tabla_Multiplicar
2    // Declarar variables
3    Definir num, i Como Entero
4    // Entrada de datos
5    Escribir "Ingresa un valor:"
6    Leer num
7    // Inicializar el contador
8    i <- 1
9    // Ciclo se repite 10 veces
10   Mientras i <= 10 Hacer
11     // Muestra la secuencia de la tabla: num * i = resultado
12     Escribir num, " x ", i, " = ", num * i
13     // Incremento del contador (Control de ciclo)
14     i <- i + 1
15   FinMientras
16   Escribir "Fin de la tabla."
17 FinAlgoritmo
  
```

La estructura mientras tiene dos variantes, una de ellas es el mientras controlado por un contador y otra variante es el mientras controlado por un centinela. La variable contador permite controlar las repeticiones, mientras van incrementando o decrementando, adicionan las iteraciones y llegan al valor esperado. En el diagrama de flujo siguiente, la variable i, es el contador, que es inicializado en 1 y llega al valor de 10 inclusive, por el menor e igual en la condición (figura del rombo), además se visualizan las tres partes de toda estructura repetitiva, la variable inicializada, la condición y el incremento (que se encuentra dentro del bloque de repetición, ver Fig. 2.55).

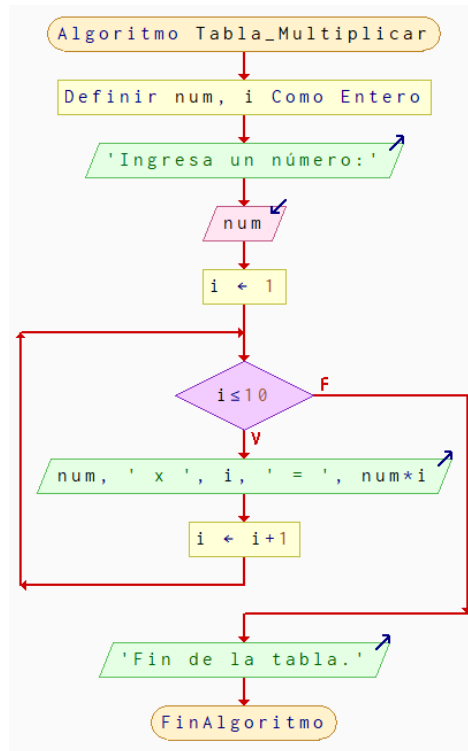


Figura 2.55: Algoritmo que genera una tabla de multiplicar.

Un ciclo Mientras controlado por centinela (o *sentinel-controlled loop*) es aquel que continúa ejecutándose hasta que se lee un valor específico, llamado el valor centinela, que indica el final de la entrada de datos. La idea de esta estructura es que debe la variable tener un valor que permita acceder a la estructura (lectura inicial o primario) esto se llama “*priming read*”. Luego las mismas líneas se colocan al final del bloque de repetición (lectura secundaria o del bloque), para la siguiente iteración.

Algoritmo 2.14: Pseudocódigo PSeInt: Suma controlada por centinela

```

1  Algoritmo Suma_Centinela
2  // Declarar variables. Usamos REAL para permitir decimales.
3  Definir numero, suma Como Real
4  // El centinela es el valor quien detiene el ciclo.
5  Definir CENTINELA Como Entero
6  CENTINELA <- -1
7  // Inicializar el acumulador de la suma
8  suma <- 0
9  // Lectura Primaria (Apertura): Lee el primer valor
10 Escribir "Ingrese un valor (", CENTINELA, " para terminar):"
11 Leer numero
12 // Repite seguidamente, en tanto el valor NO sea el centinela
13 Mientras numero <> CENTINELA Hacer
14     // 1. Proceso: Acumular el valor ingresado
15     suma <- suma + numero
16     // 2. Lectura Secundaria (Actualizar):
17     // Lee el siguiente valor para controlar, condicionar e iterar
18     Escribir "Ingrese nuevo valor (", CENTINELA, " para terminar):"
19     Leer numero
20 FinMientras
21 // Muestra el resultado final (se ejecuta al ingresar el centinela)
  
```

```

22  Escribir "---"
23  Escribir "Se ha ingresado el centinela."
24  Escribir "La suma total es: ", suma
25  FinAlgoritmo

```

En el diagrama de flujo, se muestra el valor centinela de -1 (se detiene el algoritmo si se ingresa el número -1) mientras se ingresan números en la variable “numero”. Mientras no se ingrese el valor -1 (como centinela o valor diferente a todos los que se ingresan) continúa el ingreso de valores, ver Fig. 2.56.

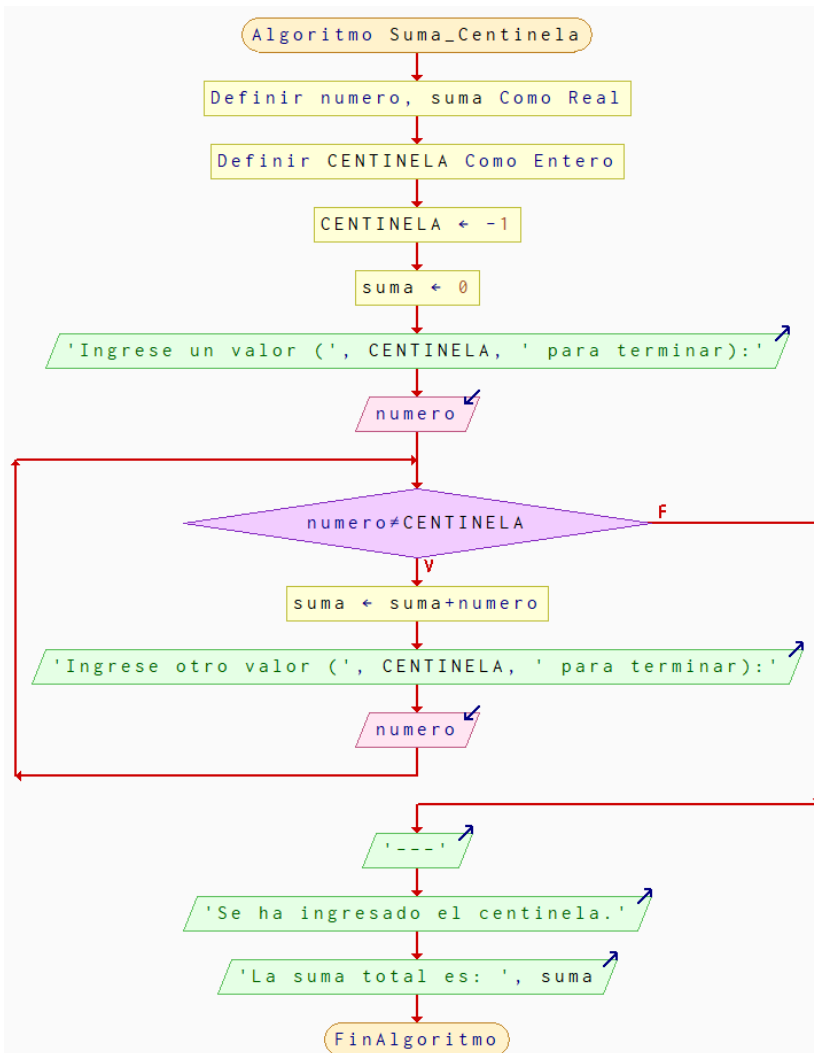


Figura 2.56: Algoritmo que genera una suma controlada por centinela.

Bucle Repetir-Hasta

El ciclo Repetir-Hasta (Repeat-Until) es otra estructura fundamental en pseudocódigo. A diferencia del ciclo Mientras, donde la condición se evalúa al inicio, el ciclo Repetir-Hasta siempre ejecuta el bloque de código al menos una vez antes de evaluar la condición al final. El ciclo continúa mientras la condición sea falsa, y se detiene hasta que la condición es verdadera. Esta estructura, en el lenguaje de programación en Java, por ejemplo, se la conoce como “Hacer-mientras” y repite por verdadero, cambiando su repetición por verdad (en vez de repetir por falso), en cambio, ya en el lenguaje de programación en Python, desaparece esta estructura.

Algoritmo 2.15: Pseudocódigo PSeInt: Validar un número en un rango (Repetir-Hasta)

```

1  Algoritmo Validacion_Rango
2      // Declarar variables
3      Definir numero Como Entero
4      // El ciclo se ejecuta al menos una vez
5      Repetir
6          // Lectura de datos
7          Escribir "Introduce un valor entre 1 - 10:"
8          Leer numero
9          // Verificar rango (Se muestra error cuando es incorrecto)
10         Si (numero < 1) o (numero > 10) Entonces
11             Escribir "ERROR El valor fuera del rango [1-10]."

```

En el diagrama de flujo se puede observar el uso de la estructura repetir-hasta (rombo morado) que repite por falso. Además, se utiliza la estructura de selección simple (si simple), para validar el valor de ingreso. Es interesante el uso combinado de las estructuras de selección y repetición como buenas prácticas en programación, observa además cómo el flujo, valor verdadero o falso se ubican en las estructuras, por ejemplo en la estructura de selección simple (rombo azul), el valor por falso está a la izquierda mientras que el valor por verdadero (y el que nos interesa) está a la derecha. En cambio, en la estructura de repetición el valor falso está a la izquierda (regresando hacia arriba) y el valor verdadero está abajo, siguiendo el flujo de arriba-abajo, ver Fig. 2.57.

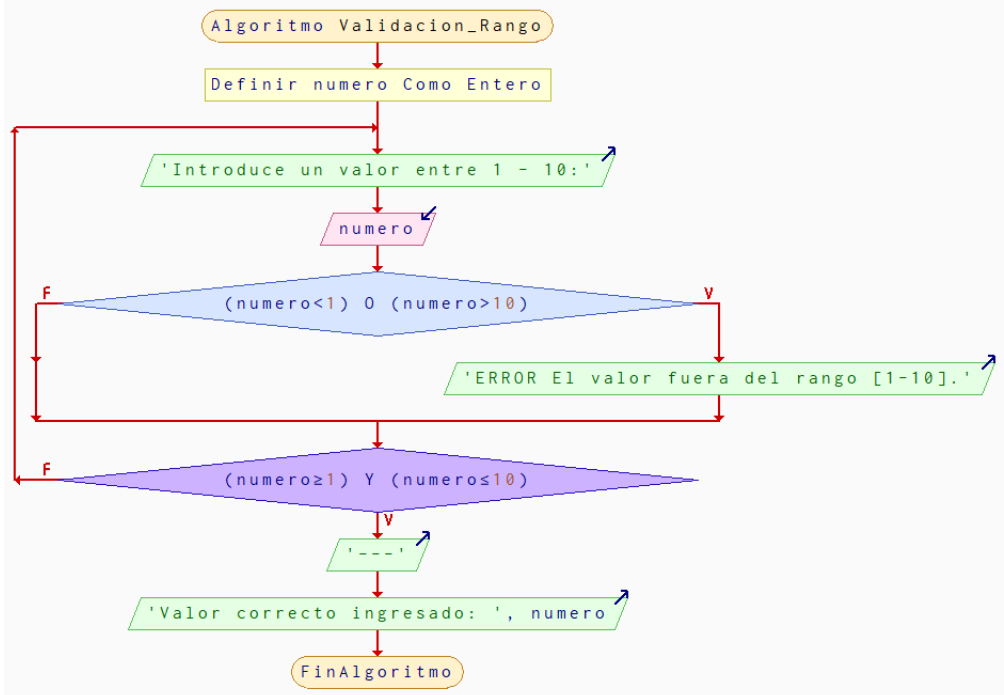


Figura 2.57: Algoritmo que utiliza la estructura mientras con un valor centinela.

Una práctica en programación es el uso de menús. Así como los menús de un local o restaurante, permiten primero leer las opciones existentes y luego escoger una de ellas. Se presenta un algoritmo donde se debe escoger una opción, a partir de un menú de opciones. Para esto el uso de la estructura Repetir-Hasta es de gran utilidad.

Algoritmo 2.16: Pseudocódigo PSeInt: Menú interactivo con Repetir-Hasta

```

1  Algoritmo Menu_Interactivo
2      // Declarar de variables
3      Definir opcion Como Entero
4      // Inicializar la variable opcion para asegurar la entrada inicial
5      // (aunque Repetir lo garantiza)
6      opcion <- 0
7      // Ciclo principal del algoritmo
8      Repetir
9          // 1. Mostrar opciones
10         Escribir "---- LISTA DE OPERACIONES ----"
11         Escribir "1. Sumar dos valores"
12         Escribir "2. Restar dos valores"
13         Escribir "3. Ver datos del programa"
14         Escribir "4. Salir"
15         Escribir "-----"
16         // 2. Leer lo escogido del usuario
17         Escribir "Elige una alternativa (1-4):"
18         Leer opcion
19         // 3. Procesar la alternativa elegida
20         // (Estructura Selectiva Diversa)
21         Segun opcion Hacer
22             1:
23                 Escribir "Has elegido Sumar."
24                 // Algoritmo para sumar.
25             2:
26                 Escribir "Has elegido Restar."
27                 // Algoritmo para restar.
28             3:
29                 Escribir "Programa de ejemplo usando Repetir-Hasta."
30             4:
31                 Escribir "Saliendo del programa. Hasta luego"
32         De Otro Modo:
33             Escribir "Alternativa incorrecta Prueba de nuevo."
34         FinSegun
35         Escribir "" // Salto siguiente para limpiar la vista
36         // La alternativa se estima o analiza al final.
37         // el ciclo termina siempre y cuando
38         // la alternativa sea 4.
39         Hasta Que opcion = 4
40 FinAlgoritmo

```

Se puede observar que las opciones pueden escogerse entre los números del 1 al 4, donde 4 es salir. También es necesario comprender que las líneas 24 y 27, es el lugar apropiado para colocar el subprograma o todo el bloque necesario para otras pequeñas soluciones que, en los lenguajes de programación, pueden llamarse módulos. Así vamos observando el inicio de la modularidad que empieza con esta estructura que permite al programador seguir escalando y extendiendo posibilidades.

En el diagrama de flujo se muestran las figuras geométricas que son apropiadas para representar el problema, por ejemplo se observa cómo la estructura Repetir-Hasta (rombo morado) se encuentra al final, permitiendo que se muestre o presente, al menos una vez, todo el bloque interior y superior al rombo, ver Fig. 2.58.

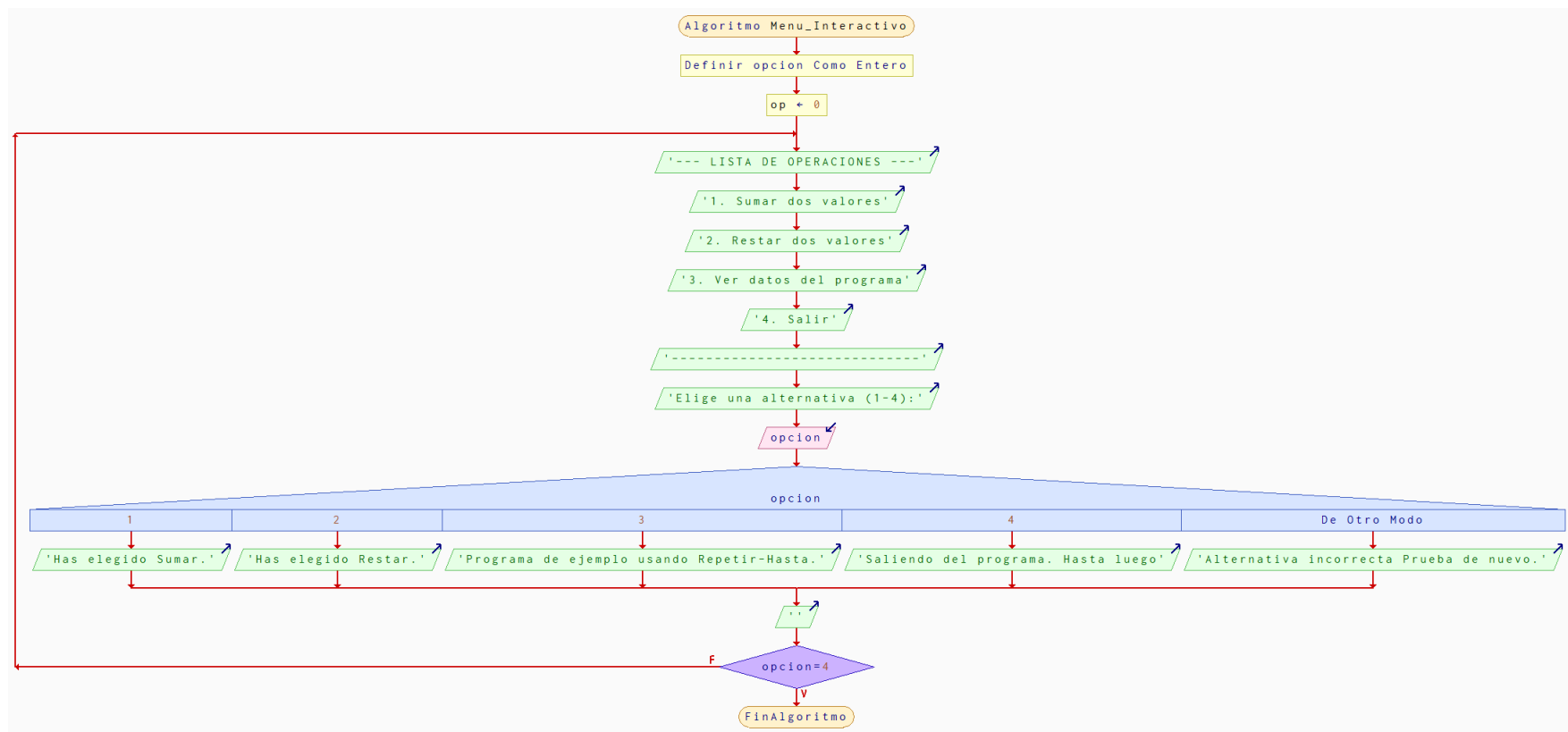


Figura 2.58: Diagrama de flujo que utiliza la estructura mientras con un valor centinela para opciones de menú.

Bucle Para

El bucle Para (o For en otros lenguajes) en pseudocódigo es una estructura de control repetitiva diseñada para ejecutar un bloque de instrucciones un número definido de veces. Es ideal usar esta estructura, cuando sabes de antemano cuántas repeticiones necesitas realizar. Se basa en una variable de control o contador que maneja el ciclo de la repetición. Además, es posible determinar el paso o el incremento (o decremento) adecuado.

Este ejemplo utiliza el bucle Para, para calcular e imprimir la tabla de multiplicar del 5, visualizando el proceso de multiplicación, desde el 1 hasta el 10.

Algoritmo 2.17: Pseudocódigo PSeInt: Tabla de multiplicar con el ciclo Para

```

1  Algoritmo TablaDelCinco
2  // Definimos las variables a utilizar
3  Definir num, resultado, i Como Entero;
4  // El valor cuya tabla queremos calcular
5  num ← 5
6  Escribir "--- Tabla de Multiplicar del ", num, " ---"
7  // Bucle Para:
8  // i es el contador, inicia en 1, y va a 10, con paso de 1 (por defecto)
9  Para i ← 1 Hasta 10 Hacer
10     // Calculamos el resultado cuando itera
11     resultado ← num * i
12     // Imprimimos el proceso y el resultado
13     Escribir num, " x ", i, " = ", resultado
14 FinPara
15 FinAlgoritmo

```

A continuación, se muestra el diagrama de flujo que presenta la solución del problema. Es necesario observar que el dibujo de la estructura tiene forma de un círculo con tres áreas, donde se muestra la variable de control i, el valor inicial a la izquierda y el valor final a la derecha. Se sobreentiende que la repetición se la realiza por verdadero, y el bloque que está dentro de las líneas del dibujo circular son las instrucciones que se iteran, ver Fig. 2.59.

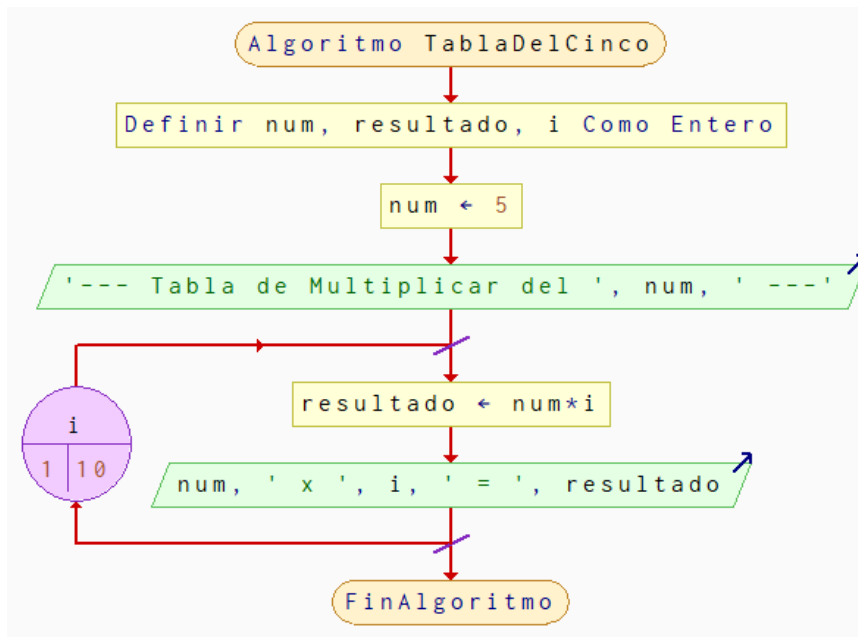


Figura 2.59: Diagrama de flujo que utiliza la estructura Para.

2.2.5. Tipos de variables (Contadores y acumuladores)

Cuando usamos estructuras repetitivas (bucles), siempre necesitamos variables especiales que nos ayuden a controlar el ciclo o a guardar resultados. Hay que poner nombres claros y apropiados, especialmente para compartir el código a otros programadores. Así el algoritmo puede sobrevivir en el tiempo. Entre las variables por su tipo de función encontramos los contadores, acumuladores, sumadores, iteradores de control, incrementadores, decrementadores y las banderas, todos ellos cumpliendo una acción de adicionar valores, en sumas, restas o acumular productos o cocientes con el mismo fin de avanzar o retroceder valores que hacen funcionar al algoritmo, ver Fig. 2.60.



Figura 2.60: Las variables nos ayudan a controlar el ciclo o a guardar resultados.

Los contadores sirven para llevar la cuenta de cuántas veces ha pasado algo. Suelen empezar en 0 o en 1. Y aumentan de 1 en 1 (o de cualquier número). El acumulador (o sumador) sirve para ir sumando valores poco a poco (como una alcancía). Siempre se inicia en 0, cuando se acumulan sumas o restas, mientras se inicializan en 1, cuando se acumulan productos o cocientes.

Un iterador o variable de control de un ciclo, es la variable que usa el bucle (Para) automáticamente (casi siempre se llama i, j, k). Los incrementadores o decrementadores son instrucciones que aumentan o disminuyen el valor de una variable. Estas acciones pueden incrementar un valor de 1 en 1, de otro número o decrementar propiamente. En cambio, una variable bandera es una variable lógica (Verdadero o Falso) que sirve para recordar si algo pasó.

A continuación, se presenta un algoritmo donde se utilizan variables que de acuerdo con su función permiten, acumular, iterar, contar, guardar un valor mayor, ser un valor bandera, que hacen de ellas útiles para la ejecución de una solución o tarea.

Algoritmo 2.18: Pseudocódigo PSeInt: Ejemplo de todas las variables útiles en un algoritmo

```

1 Algoritmo EjemploCompleto
2   Definir numero, suma, mayor Como Entero
3   Definir contador, i Como Entero
4   Definir salir Como Logico
5   suma <- 0 // acumulador

```

```

6  contador <- 0 // contador
7  mayor <- -999999 // a guardar el mayor
8  salir <- Falso // bandera
9  Repetir
10     Escribir "Dame un valor (0 destinado a terminar): "
11     Leer numero
12     Si numero <> 0 Entonces
13         contador <- contador + 1 // contador
14         suma <- suma + numero // acumulador
15         Si numero > mayor Entonces
16             mayor <- numero // guardamos el mayor
17         FinSi
18     Sino
19         salir <- Verdadero // activamos la bandera
20     FinSi
21 Hasta Que salir = Verdadero
22 Escribir "Ingresaste ", contador, " valores"
23 Escribir "La suma total es: ", suma
24 Escribir "El promedio es: ", suma/contador
25 Escribir "El mayor valor fue: ", mayor
26 FinAlgoritmo

```

Las variables de tipo contador y los acumuladores siempre deben inicializarse en un valor conocido (generalmente cero cuando van a incrementar o decrementar, pero en 1 si acumulan productos o cocientes) antes de ingresar a un bucle. Estas variables de acuerdo con su función, su ubicación, su almacenamiento y tipo de valor a guardar son fundamentales para gestionar las repeticiones o resultados lógicos que requiere el algoritmo, ver Tabla 2.7.

Tipos de variables de control de acuerdo a su función

Estas variables son fundamentales para gestionar bucles, totales y estados lógicos.

Tabla 2.7: Clasificación y función de las variables de control.

Nombre	¿Para qué sirve?	Valor inicial	Ejemplo
Contador	Contar cuántas veces ocurre un evento o se ejecuta un bloque.	0 o 1	conta <- conta + 1
Acumulador	Sumar o totalizar una serie de valores (montos, notas, etc.).	0 ó 1	suma <- suma + nota acum <- acum * 3
Iterador (i, j, k)	Variable de control que lleva la cuenta en un ciclo (Para).	Lo define el ciclo.	i toma valores 1, 2, 3...
Incrementador	Aumentar una variable por una cantidad fija o variable.	-	edad <- edad + 1
Decrementador	Disminuir una variable por una cantidad fija o variable.	-	vidas <- vidas - 1
Bandera	Recordar si una condición o evento ocurrió (estado Sí o No).	Falso	ubica <- Verdadero

El diagrama de flujo permite la representación del algoritmo y el uso importante de las variables, utilizando adecuadamente nombres apropiados y claros para su comprensión, ver Fig. 2.61.

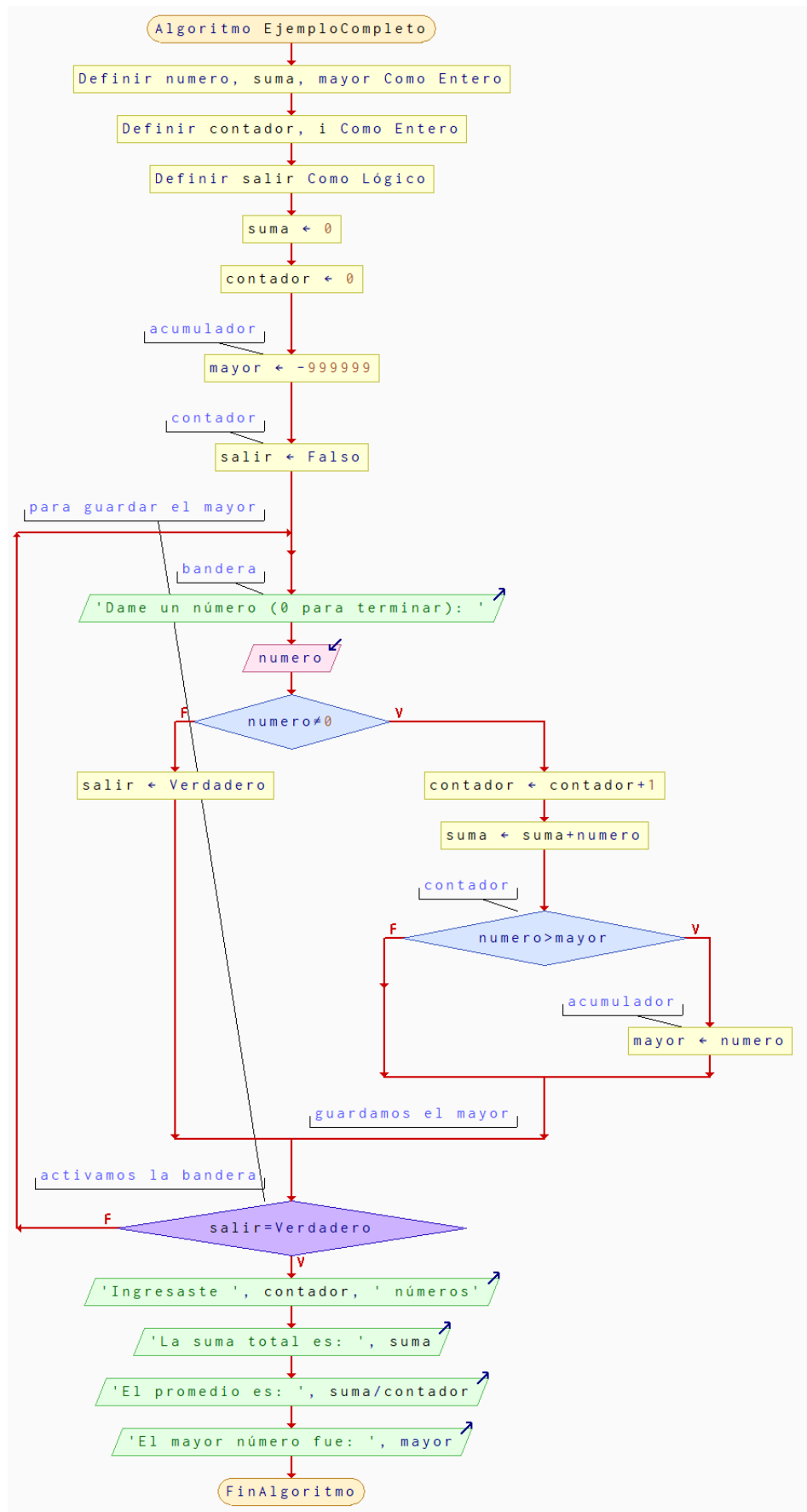


Figura 2.61: Uso de diferentes tipos de variables para controlar el ciclo y guardar resultados.



Laboratorio 3

Capítulo 2

2.3. Ejercicios del Laboratorio 3

Ejercicio 1

Escribe un algoritmo que nos diga cuál es el volumen de un cono con un radio de la base de 14 y una altura de 26. La fórmula que debes usar es: $V = \frac{1}{3} \cdot \pi \cdot r^2 \cdot h$. Recuerda que el valor (aproximado) de PI es 3.141592. Para este ejercicio utilizaremos los valores fijos proporcionados, asignando al radio con nombre $r = 14$, altura $h = 26$ y $\pi \approx 3.141592$.

Algoritmo 2.19: Pseudocódigo PSeInt: Cálculo del volumen de un cono

```

1 Algoritmo Calcular_Volumen_Cono
2   // Declarar variables
3   Definir radio, altura Como Entero
4   Definir pi, volumen Como Real
5   // 1. Asignar valores fijos (Datos del problema)
6   radio <- 14
7   altura <- 26
8   pi <- 3.141592
9   // 2. Proceso:  $V = (1/3) \cdot \pi \cdot \text{radio}^2 \cdot \text{altura}$ 
10  // El 1/3 se implementa mediante el cociente por 3 al final.
11  // Calcular radio al cuadrado (radio * radio)
12  volumen <- pi * (radio * radio) * altura
13  // Dividir el resultado por 3 con el fin de obtener el volumen final
14  volumen <- volumen / 3
15  // 3. Salida de resultados
16  Escribir "--- Volumen de un Cono ---"
17  Escribir "Radio (r): ", radio
18  Escribir "Altura (h): ", altura
19  Escribir "El volumen (V) del cono es: ", volumen
20 FinAlgoritmo

```

Ejercicio 2

Modifica el algoritmo anterior para que use tres variables, todas de tipo real: radio, altura y volumen. Las dos primeras se inicializarán a 14.5 y 26.79 respectivamente. La tercera obtendrá el resultado de la fórmula.

Algoritmo 2.20: Pseudocódigo PSeInt: Cálculo de volumen de un cono con inicialización de variables

```

1 Algoritmo Calcular_Volumen_Cono_Modificado
2   // Declarar las variables de tipo Real (equivalente a double)
3   Definir radio, altura, volumen, PI Como Real
4   // 1. Inicializar variables (Datos de entrada fijos)
5   radio <- 14.5
6   altura <- 26.79
7   PI <- 3.141592
8   // 2. Proceso:  $V = (1/3) \cdot \pi \cdot \text{radio}^2 \cdot \text{altura}$ 
9   // Calculamos el volumen sin pedir datos al usuario,
10  // solo usando las variables inicializadas.
11  // Procedimiento:  $\pi \cdot (\text{radio} \cdot \text{radio}) \cdot \text{altura} / 3$ 
12  volumen <- (PI * radio * radio * altura) / 3
13  // 3. Salida de resultados
14  Escribir "--- Volumen de un Cono ---"
15  Escribir "Radio (r) inicializado: ", radio
16  Escribir "Altura (h) inicializada: ", altura
17  Escribir "El volumen (V) del cono es: ", volumen
18 FinAlgoritmo

```

*** Ejercicio 3**

Escribe un algoritmo que lea del teclado un número (real) de grados Fahrenheit y lo convierta a Celsius mostrando el resultado en la pantalla. La fórmula para convertir grados Fahrenheit (F) a grados Celsius (C) es:

$$C = (F - 32) \cdot \frac{5}{9}$$

Algoritmo 2.21: Pseudocódigo PSeInt: Conversión de temperatura

```

1 Algoritmo Fahrenheit_a_Celsius
2   // Declarar variables de tipo Real (para manejar decimales)
3   Definir fahrenheit, celsius Como Real
4   // Lectura de datos
5   Escribir "Introduce la temperatura en grados Fahrenheit (F):"
6   Leer fahrenheit
7   // Proceso:  $C = (F - 32) \cdot (5/9)$ 
8   // Nota: Es fundamental usar 5.0/9.0 o (5/9.0)
9   // para asegurar el cociente real
10  // y evitar que PSeInt realice un cociente entero (5 div 9 = 0).
11  celsius <- (fahrenheit - 32) * (5.0 / 9.0)
12  // Salida de resultados
13  Escribir "-----"
14  Escribir "La temperatura en grados Fahrenheit es: ", fahrenheit, " F"
15  Escribir "Equivale a ", celsius, " C"
16 FinAlgoritmo

```

*** Ejercicio 4**

Escribe un algoritmo que lea del teclado un tiempo transcurrido en segundos y muestre en la pantalla las horas, los minutos y los segundos equivalentes.

Algoritmo 2.22: Pseudocódigo PSeInt: Segundos a horas y minutos

```

1 Algoritmo Conversor_Tiempo
2   // Declarar variables.
3   Definir total_segundos, segundos, minutos, horas Como Entero
4   // Lectura de datos
5   Escribir "Introduce el tiempo transcurrido en segundos:"
6   Leer total_segundos
7   // 1. Encontrar Horas. Hay 3600 segundos en una hora (60 * 60).
8   // El cociente nos da la cantidad de horas completas.
9   horas <- total_segundos / 3600
10  // 2. Minutos, Segundos restantes
11  // El operador (%) nos da los segundos sobrantes luego de las horas.
12  segundos_restantes_despues_horas <- total_segundos MOD 3600
13  // 3. Minutos. Dividimos los segundos restantes entre 60
14  // luego obtenemos los minutos completos.
15  minutos <- segundos_restantes_despues_horas / 60
16  // 4. Segundos Finales
17  // El operador del cociente anterior nos da los segundos sobrantes
18  // luego de los minutos.
19  segundos <- segundos_restantes_despues_horas MOD 60
20  // Salida de resultados
21  Escribir "-----"
22  Escribir "Tiempo original en segundos: ", total_segundos
23  Escribir "Tiempo equivalente: ", horas, " horas, " Sin Saltar
24  Escribir "minutos, " minutos y ", segundos, " segundos."
25 FinAlgoritmo

```

* Ejercicio 5

Escriba un programa que calcule la nota final de la asignatura (sobre 100 puntos) a partir de las calificaciones obtenidas en los dos ciclos de evaluación (Interciclo y Final). El programa deberá solicitar al usuario las cuatro calificaciones: Aprovechamiento del Interciclo (Máx. 40 puntos). Examen del Interciclo (Máx. 10 puntos). Aprovechamiento Final (Máx. 40 puntos). Examen Final (Máx. 10 puntos). La Nota Total Final se obtendrá sumando las puntuaciones de los cuatro componentes.

Algoritmo 2.23: Pseudocódigo PSeInt: Cálculo de nota final)

```

1 Algoritmo Calcular_Nota_Final_100
2   // Declarar variables.
3   Definir ap_interciclo, ex_interciclo, ap_final, ex_final Como Real
4   Definir nota_total Como Real
5   // --- 1. Lectura de Calificaciones ---
6   Escribir "--- INGRESO DE CALIFICACIONES (100 Puntos) ---"
7   // Ciclo Interciclo (50 puntos)
8   Escribir "1. Ingrese Nota de Aprovechamiento del INTERCICLO (40 ptos):"
9   Leer ap_interciclo
10  Escribir "2. Ingrese Nota de Examen del INTERCICLO (10 ptos):"
11  Leer ex_interciclo
12  // Ciclo Final (50 puntos)
13  Escribir "3. Ingrese Nota de Aprovechamiento FINAL (40 ptos):"
14  Leer ap_final
15  Escribir "4. Ingrese Nota de Examen FINAL (10 ptos):"
16  Leer ex_final
17  // --- 2. Proceso: Nota Total Final ---
18  // Nota Total = (AP Interciclo + EX Interciclo) + (AP Final + EX Final)
19  nota_total <- ap_interciclo + ex_interciclo + ap_final + ex_final
20  // --- 3. Salida de resultados ---
21  Escribir "-----"
22  Escribir "RESUMEN DE NOTAS OBTENIDAS:"
23  Escribir "Total Interciclo (50): ", ap_interciclo + ex_interciclo
24  Escribir "Total Final (50): ", ap_final + ex_final
25  Escribir "NOTA FINAL TOTAL: ", nota_total, " / 100"
26 FinAlgoritmo

```

* Ejercicio 6

Escriba un algoritmo que evalúe la siguiente expresión en términos de una sola línea (usa los paréntesis que sean estrictamente necesarios):

$$\frac{4}{3 \cdot (r + 34)} - 9 \cdot (a + b \cdot c) + \frac{3 + d \cdot (2 + a)}{a + b \cdot d}$$

Algoritmo 2.24: Pseudocódigo PSeInt: Evaluación de expresión matemática compleja

```

1 Algoritmo Evaluar_Expresion
2   // Declarar variables para evaluar con decimales
3   Definir r, a, b, c, d, resultado, t1, t2, t3 Como Real
4   // Asignar valores de ejemplo para probar
5   // (Estos valores son arbitrarios, se pueden cambiar)
6   r <- 5
7   a <- 2
8   b <- 3
9   c <- 4
10  d <- 1
11  // Proceso: Evaluar en una sola lista

```

```

12 // 4 / (3*(r+34)) - 9*(a+b*c) + (3+d*(2+a))/(a+b*d)
13 t1 <- 4 / (3 * (r + 34))
14 t2 <- - (9 * (a + b * c))
15 t3 <- (3 + d * (2 + a)) / (a + b * d)
16 resultado <- t1 + t2 + t3
17 // Salida de resultados
18 Escribir "--- EVALUAR ---"
19 Escribir "El resultado es: ", resultado
20 FinAlgoritmo

```

Ejercicio 7

Escriba un programa que calcule el área de un triángulo utilizando la Ley del Seno. El programa debe solicitar al usuario la longitud de dos lados (Lado1 y Lado2) y el ángulo incluido (α) entre ellos, medido en grados. La fórmula a implementar es:

$$\text{Área} = \frac{1}{2} \cdot \text{Lado1} \cdot \text{Lado2} \cdot \sin(\alpha_{\text{rad}})$$

Nota Importante: El programa debe convertir el ángulo ingresado de grados a radianes antes de usar la función seno, usando la fórmula:

$$\alpha_{\text{rad}} = \alpha_{\text{grados}} \cdot \frac{\pi}{180}$$

Algoritmo 2.25: Pseudocódigo PSeInt: Cálculo del área de triángulo por Ley del Seno

```

1 Algoritmo Area_Triangulo_Ley_Seno
2 // Declarar variables.
3 Definir lado1, lado2, angulo_grados, angulo_radianes Como Real
4 Definir area, Pi Como Real
5 // 1. Asignar constante Pi
6 Pi <- 3.141592
7 // 2. Lectura de datos
8 Escribir "--- Hallar valor con Ley del Seno ---"
9 Escribir "Ingrese la longitud del primer lado (Lado1):"
10 Leer lado1
11 Escribir "Ingrese la longitud del segundo lado (Lado2):"
12 Leer lado2
13 Escribir "Ingrese el valor en grados entre estos dos lados:"
14 Leer angulo_grados
15 // 3. Proceso: Convertir grados a radianes
16 // angulo_radianes = angulo_grados * (Pi / 180)
17 angulo_radianes <- angulo_grados * (Pi / 180.0)
18 // 4. Proceso: Calcular
19 // Area = 0.5 * lado1 * lado2 * sen(angulo_radianes)
20 area <- 0.5 * lado1 * lado2 * sen(angulo_radianes)
21 // 5. Salida de resultados
22 Escribir "-----"
23 Escribir "Convertido a radianes: ", angulo_radianes
24 Escribir "La superficie es: ", area
25 FinAlgoritmo

```

* Ejercicio 8

Escribe un algoritmo que pida al usuario el valor de dos variables reales x e y , y a continuación muestre el resultado de aplicarles la siguiente fórmula. Revisa en el programa de PSeInt todas aquellas expresiones válidas y útiles para desarrollar el ejercicio, por ejemplo, en la ayuda de PseInt, en funciones predefinidas, se indica que la raíz puede ser $RC()$ o $Raiz()$. (Declare constantes para los valores fijos):

$$f(x,y) = \sqrt{1,531^{(x+y)} + \frac{e^x - e^y \cdot (\sin(x) - \tan(y))}{\ln(y) \cdot 3,141592^x}}$$

Algoritmo 2.26: Pseudocódigo PSeInt: Evaluación de fórmula con constantes (Final)

```

1  Algoritmo Evaluar_Formula_XY_V2
2      // Declarar constantes (valores fijos)
3      Definir BASE_POTENCIA Como Real; BASE_POTENCIA <- 1.531
4      Definir PII Como Real; PII <- 3.141592
5      Definir x, ye, resultado, numerador, denominador Como Real
6      // 1. Lectura de datos
7      Escribir "Introduce el valor de la variable x:"
8      Leer x
9      Escribir "Introduce el valor de la variable ye:"
10     Leer ye
11     // 2. Procesamiento (Numerador entre Denominador)
12     // Numerador: e^x - e^ye * (sen(x) - tan(ye))
13     // Nota: sen() como tan() esperan radianes.
14     numerador <- exp(x) - exp(ye) * (sen(x) - tan(ye))
15     // Denominador: ln(ye) * PII^x
16     // Nota: ln() es el logaritmo natural.
17     // La potencia usa el circunflejo ^
18     denominador <- ln(ye) * (PII ^ x)
19     // 3. Calcular el valor radical (f(x,ye))
20     // El denominador no sea cero antes del cociente
21     Si denominador <> 0 Entonces
22         // Calculo del valor interno del radicando
23         resultado <- (BASE_POTENCIA^(x+ye)) + (numerador/denominador)
24         // Obtener el radical con rc(z) (radicando)
25         Si resultado >= 0 Entonces
26             Escribir "El resultado de f(x,ye) es: ", rc(resultado)
27         Sino
28             Escribir "ERROR: El valor radical es negativo."
29         FinSi
30     Sino
31         Escribir "ERROR: Cociente cero."
32     FinSi
33 FinAlgoritmo

```



Laboratorio 4

Capítulo 2

2.4. Ejercicios del Laboratorio 4

* Ejercicio 1

Realiza un algoritmo que muestre solo el siguiente mensaje: Hola a todos.

Algoritmo 2.27: Pseudocódigo PSeInt: Mensaje de saludo simple

```
1 Algoritmo Mensaje_Hola
2     // Proceso: mostrar el texto requerido
3     Escribir "Hola a todos."
4 FinAlgoritmo
```

* Ejercicio 2

Realiza un algoritmo que, dado un lado, obtenga y muestre el área del cuadrado. Para efectuar con potencias, en algoritmos se utiliza el símbolo ****** o el circunfijo **^**, se muestran a continuación las fórmulas a utilizar: $area = lado^2$ o utilizar $area = lado * lado$.

Algoritmo 2.28: Pseudocódigo PSeInt: Cálculo del área de un cuadrado

```
1 Algoritmo Area_Cuadrado
2     // Definir variables
3     Definir lado, area Como Real
4     // 1. Lectura de datos
5     Escribir "Introduce la medida del lado del cuadrado:"
6     Leer lado
7     // 2. Procesamiento
8     // Se usa el circunflejo ^ para la potencia (lado^2)
9     area <- lado ^ 2
10    // 3. Salida de resultado
11    Escribir "La superficie del cuadrado con lado ", lado, " es: ", area
12 FinAlgoritmo
```

* Ejercicio 3

Realiza un algoritmo que, dado el valor de un producto, determine, el subtotal a pagar, que depende del valor del producto por unidad y la cantidad de productos a comprar. Luego determine el valor del IVA 12 % del subtotal y sume dicha cantidad con el subtotal para hallar el total a pagar (revise una factura de compras que tenga IVA para su comprensión).

Algoritmo 2.29: Pseudocódigo PSeInt: Computo de una compra con impuesto al valor agregado

```
1 Algoritmo Proceso_Factura
2     // Declarar constantes
3     Definir IVA_TASA Como Real; IVA_TASA <- 0.12
4     // Representa el 12 por ciento
5     // Declaracion de variables
6     Definir valor_unitario, cantidad, subtotal Como Real
7     Definir iva_computado, total_a_pagar Como Real
8     // 1. Lectura de datos
9     Escribir "Ingresa el valor unitario del producto:"
10    Leer valor_unitario
11    Escribir "Ingresa la cantidad de productos a comprar:"
12    Leer cantidad
13    // 2. Proceso
14    // Determinar el subtotal (valor * cantidad)
15    subtotal <- valor_unitario * cantidad
16    // Determinar el valor del IVA (subtotal * tasa)
17    iva_computado <- subtotal * IVA_TASA
18    // Determinar el total a pagar (subtotal + IVA)
```

```

19     total_a_pagar <- subtotal + iva_computado
20     // 3. Salida de resultados
21     Escribir "--- Detalle de la factura ---"
22     Escribir "Subtotal a pagar: ", subtotal
23     Escribir "Valor del IVA (12 por ciento): ", iva_computado
24     Escribir "Total a pagar: ", total_a_pagar
25 FinAlgoritmo

```

* Ejercicio 4

Realiza un algoritmo que obtenga y muestre en pantalla el valor de y a partir de la ecuación:

$$y = 3 * x^2 + 7x - 15$$

Algoritmo 2.30: Pseudocódigo PSeInt: Computo del valor de la variable

```

1 Algoritmo Obtener_Valores_YE
2     // Declarar variables
3     Definir x, ye Como Real
4     // 1. Lectura de datos
5     // Se requiere el valor de x para calcular ye
6     Escribir "Ingrese el valor de la variable x:"
7     Leer x
8     // 2. Proceso
9     // Aplicar: ye = 3*x^2 + 7x - 15
10    // La potencia se indica con el circunflejo ^
11    ye <- (3 * (x ^ 2)) + (7 * x) - 15
12    // 3. Salida de resultado
13    Escribir "El valor de ye para x = ", x, " es: ", ye
14 FinAlgoritmo

```

* Ejercicio 5

Realiza un algoritmo que obtenga y muestre el área del círculo y el perímetro de la circunferencia: $area = \pi r^2$, $perimetro = 2\pi r$. La salida será de la siguiente forma: El radio ingresado es, el área es, el perímetro es Para efectuar potencias, en algoritmos se utiliza el símbolo de intercalación ^denominado caret, sombrero o circunflejo.

Algoritmo 2.31: Pseudocódigo PSeInt: Cómputo del área y perímetro de un círculo

```

1 Algoritmo Area_y_Perimetro_Circulo
2     // Declarar constantes
3     Definir PII Como Real; PII <- 3.141592
4     Definir radio, area, perimetro Como Real
5     // 1. Lectura de datos
6     // Se solicita el valor del radio
7     Escribir "Ingrese el valor del radio de la circunferencia:"
8     Leer radio
9     // 2. Proceso
10    // Aplicar area = PII * radio^2
11    area <- PII * (radio ^ 2)
12    // Aplicar perimetro: perimetro = 2 * PII * radio
13    perimetro <- 2 * PII * radio
14    // 3. Salida de resultado
15    // Mostrar el resultado en el formato solicitado
16    Escribir "El radio ingresado es ", radio Sin Saltar
17    Escribir ", el area es ", area, ", el perimetro es ", perimetro
18 FinAlgoritmo

```

*** Ejercicio 6**

Realiza un algoritmo que al ingresar dos números, obtenga y muestre la suma de ambos. La salida debe mostrarse de la siguiente manera: Los números ingresados son y, la suma es

Algoritmo 2.32: Pseudocódigo PSeInt: Suma de dos números

```

1 Algoritmo Suma_Dos_Numeros
2     // Declarar variables de proceso
3     Definir num1, num2, suma Como Real
4     // 1. Lectura de datos
5     Escribir "Ingrese el primer valor:"
6     Leer num1
7     Escribir "Ingrese el segundo valor:"
8     Leer num2
9     // 2. Proceso
10    // Aplicar la suma
11    suma <- num1 + num2
12    // 3. Salida de resultado
13    // Mostrar el resultado en el formato solicitado
14    Escribir "Los valores ingresados son ", num1 Sin Saltar
15    Escribir " y ", num2, ", la suma es ", suma
16 FinAlgoritmo

```

*** Ejercicio 7**

Realiza un algoritmo que al ingresar dos números, obtener y mostrar la resta de ambos. La salida debe mostrarse de la siguiente manera: Los números ingresados son y, la resta es

Algoritmo 2.33: Pseudocódigo PSeInt: Resta de dos números

```

1 Algoritmo Resta_Dos_Numeros
2     // Declarar variables de proceso
3     Definir num1, num2, diferencia Como Real
4     // 1. Lectura de datos
5     Escribir "Ingrese el primer numero (minuyendo):"
6     Leer num1
7     Escribir "Ingrese el segundo numero (sustraendo):"
8     Leer num2
9     // 2. Proceso de resta
10    diferencia <- num1 - num2
11    // 3. Salida de resultado
12    // Mostrar el resultado en el formato solicitado
13    Escribir "Los valores ingresados son ", num1, " y " Sin Saltar
14    Escribir num2, ", la resta es ", diferencia
15 FinAlgoritmo

```

*** Ejercicio 8**

Realiza un algoritmo que al ingresar dos números, obtener y mostrar la división de ambos. La salida debe mostrarse de la siguiente manera: Los números ingresados son y, la división es (explique qué ocurriría si se ingresan los números en orden inverso).

Algoritmo 2.34: Pseudocódigo PSeInt: Cociente de dos números

```

1 Algoritmo Division_Dos_Numeros
2     // Declarar variables de proceso
3     Definir num1, num2, cociente Como Real
4     // 1. Lectura de datos
5     Escribir "Ingrese el primer valor (dividendo):"
6     Leer num1

```

```

7      Escribir "Ingrese el segundo valor (divisor):"
8      Leer num2
9      // 2. Proceso
10     // El cociente solo es posible si el divisor no es cero
11     Si num2 <> 0 Entonces
12         cociente <- num1 / num2
13         // 3. Salida de resultado
14         // Mostrar el resultado en el formato solicitado
15         Escribir "Los valores ingresados son ", num1 Sin Saltar
16         Escribir " y ", num2, ", la division es ", cociente
17     Sino
18         // Mensaje de error al ser el divisor cero
19         Escribir "ERROR: No es posible dividir por cero."
20     FinSi
21 FinAlgoritmo

```

La división es una operación no conmutativa. Esto significa que el orden en que se ingresan los números afecta directamente el resultado. Caso A (Orden Ingresado): Se calcula $num1/num2$. Caso B (Orden Inverso): Si se invierte el orden de los números en la operación, se calcularía $num2/num1$. Si el usuario ingresa los números 10 y 5, ver Tabla 2.8.

Tabla 2.8: Ejemplos de operaciones de división

Orden de Ingreso	Operación	Resultado
Normal (10, 5)	10 / 5	2
Inverso (5, 10)	5 / 10	0.5

El cociente resultante sería diferente. La única excepción a esta diferencia ocurriría si $num1$ y $num2$ fueran iguales (por ejemplo, $10/10 = 1$) o si el resultado de la división original fuera -1 (por ejemplo, $-5/5 = -1$ y $5/-5 = -1$). Además, si el primer número ($num1$) fuera cero, y se invirtiera el orden, podría ocurrir un error de división por cero si $num2$ no fuera cero. Por ejemplo: $0/5 = 0$, $5/0 = \text{Error}$, (División por cero).

* Ejercicio 9

Realiza un algoritmo que obtenga y muestre el área e hipotenusa de un triángulo rectángulo. Ingresar b (base), a (lado), $area = (b \cdot a)/2$, e $hipotenusa = \sqrt{(a^2 + b^2)}$. Utilice el símbolo $^$, para obtener en bloques de proceso en algoritmos $H^2 = (a^2 + b^2)$, donde $H = \sqrt{(H^2)}$. También hay funciones para elevar potencia que se trabajarán posteriormente. La salida será de la siguiente forma: El lado ingresado es, la base ingresada es, el área es, la hipotenusa es

Algoritmo 2.35: Pseudocódigo PSeInt: Cómputo del área e hipotenusa del triángulo rectángulo

```

1 Algoritmo Area_e_Hipotenusa_Triangulo
2     // Declarar variables
3     Definir lado_a, base, area, hipotenusa Como Real
4     Escribir "Ingrese el valor del lado a (altura):"
5     Leer lado_a
6     Escribir "Ingrese el valor de la base b:"
7     Leer base
8     area <- (base * lado_a) / 2
9     hipotenusa <- ( (lado_a ^ 2) + (base ^ 2) ) ^ (1/2)
10    Escribir "El lado ingresado es ", lado_a Sin Saltar
11    Escribir ", la base ingresada es ", base Sin Saltar
12    Escribir ", el area es ", area, ", la hipotenusa es ", hipotenusa
13 FinAlgoritmo

```

* Ejercicio 10

Realiza un algoritmo que permita ingresar dos números, obtener y mostrar la suma de ambos, la multiplicación de ambos y la división de ambos. La salida debe mostrar de forma concatenada de la siguiente manera: Los números ingresados son y, la suma es, la multiplicación es la división es

Algoritmo 2.36: Pseudocódigo PSeInt: Operaciones combinadas

```

1 Algoritmo Operaciones_Basicas
2     // Declarar variables de proceso
3     Definir num1, num2, suma, producto, cociente Como Real
4     // 1. Lectura de datos
5     Escribir "Ingrese el primer valor:"
6     Leer num1
7     Escribir "Ingrese el segundo valor:"
8     Leer num2
9     // 2. Proceso
10    suma <- num1 + num2
11    producto <- num1 * num2
12    Si num2 <> 0 Entonces
13        cociente <- num1 / num2
14        Escribir "Los valores ingresados son ", num1 Sin Saltar
15        Escribir " y ", num2, ", la suma es ", suma Sin Saltar
16        Escribir ", la multiplicacion es ", producto Sin Saltar
17        Escribir ", la division es ", cociente
18    Sino
19        Escribir "ERROR. El segundo valor no puede ser cero."
20        Escribir "Los valores ingresados son ", num1 Sin Saltar
21        Escribir " y ", num2, ", la suma es ", suma Sin Saltar
22        Escribir ", el producto es ", producto Sin Saltar
23        Escribir ", el cociente es: INDETERMINADO"
24    FinSi
25 FinAlgoritmo

```

* Ejercicio 11

Realiza un algoritmo que, dada una resistencia de 4 ohmios y el voltaje de 2 voltios y, usando las fórmulas $V = I \cdot R$, donde (V) es voltaje, I es corriente en amperios y resistencia R es ohmios, se calcule y muestre el valor de la corriente, $I = V/R$. Posteriormente, calcular y mostrar la potencia $P = RI^2$, o $P = VI$. Finalmente, mostrar en una sola salida, tanto los datos ingresados, resistencia, voltaje como corriente y la potencia.

Algoritmo 2.37: Pseudocódigo PSeInt: Ley de Ohm (Voltaje y Potencia)

```

1 Algoritmo Calculo_Electrico
2     // Declarar variables
3     Definir resistencia, corriente, voltaje, potencia Como Real
4     Escribir "Ingrese el valor de la resistencia (R) en Ohmios:"
5     Leer resistencia
6     Escribir "Ingrese el valor del voltaje (V) en Voltios:"
7     Leer voltaje
8     corriente = voltaje / resistencia
9     potencia = voltaje * corriente
10    Escribir "La resistencia (R) es ", resistencia, " Ohmios."
11    Escribir "La corriente (I) es ", corriente, " Amperios."
12    Escribir "El voltaje (V) es ", voltaje, " Voltios"
13    Escribir ", y la Potencia (P) es ", potencia, " Watts."
14 FinAlgoritmo

```

* Ejercicio 12

Realiza un algoritmo que obtenga y muestre el área y perímetro de un rectángulo. Ingresar b (base) y h (altura). Muestre el resultado concatenando las variables de salida y las variables de entradas, $area = b \cdot a$ y $perimetro = 2 \cdot (b + a)$.

Algoritmo 2.38: Pseudocódigo PSeInt: Computo del área y perímetro de un rectángulo

```

1 Algoritmo Area_y_Perimetro_Rectangulo
2   // Declarar variables de proceso
3   Definir base, altura, area, perimetro Como Real
4   Escribir "Ingrese el valor de la base (b):"
5   Leer base
6   Escribir "Ingrese el valor de la altura (h):"
7   Leer altura
8   area <- base * altura
9   perimetro <- 2 * (base + altura)
10  Escribir "La base ingresada es ", base Sin Saltar
11  Escribir ", la altura es ", altura Sin Saltar
12  Escribir ", el area es ", area Sin Saltar
13  Escribir " y el perimetro es ", perimetro
14 FinAlgoritmo

```

* Ejercicio 13

Realiza un algoritmo que obtenga y muestre el área y perímetro de un rombo. Ingresar D , d , a (lado), $area = 4 \cdot a$, o $perimetro = (D \cdot d)/2$. Muestre el resultado concatenando las variables de salida y las variables de entradas.

Algoritmo 2.39: Pseudocódigo PSeInt: Cómputo del área y perímetro de un rombo

```

1 Algoritmo Area_y_Perimetro_Rombo
2   // Declarar variables de proceso
3   Definir diagonal_mayor, diagonal_menor, lado_a Como Real
4   Definir area, perimetro Como Real
5   Escribir "Ingrese el valor de la Diagonal Mayor (D):"
6   Leer diagonal_mayor
7   Escribir "Ingrese el valor de la Diagonal Menor (d):"
8   Leer diagonal_menor
9   Escribir "Ingrese el valor del lado (a):"
10  Leer lado_a
11  area <- (diagonal_mayor * diagonal_menor) / 2
12  perimetro <- 4 * lado_a
13  Escribir "La Diagonal Mayor (D) es ", diagonal_mayor Sin Saltar
14  Escribir ", la Diagonal Menor (d) es ", diagonal_menor Sin Saltar
15  Escribir ", el lado (a) es ", lado_a Sin Saltar
16  Escribir ", el area es ", area Sin Saltar
17  Escribir " y el perimetro es ", perimetro
18 FinAlgoritmo

```

* Ejercicio 14

Realiza un algoritmo que obtenga y muestre el área y perímetro de un paralelogramo. Ingresar b , h , a , y las fórmulas $area = b \cdot h$ y $perimetro = 2 \cdot (a + b)$. Muestre el resultado concatenando las variables de salidas y las variables de entradas.

Algoritmo 2.40: Pseudocódigo PSeInt: Cómputo de área y perímetro de un paralelogramo

```

1 Algoritmo Area_y_Perimetro_Paralelogramo
2   // Declarar variables de proceso
3   Definir base, altura, lado_a, area, perimetro Como Real

```

```

4      // 1. Lectura de datos
5      // Ingreso de valores, base, la altura y el lado adyacente
6      Escribir "Ingrese el valor de la base (b):"
7      Leer base
8      Escribir "Ingrese el valor de la altura (h):"
9      Leer altura
10     Escribir "Ingrese el valor del lado adyacente (a):"
11     Leer lado_a
12     // 2. Proceso
13     area <- base * altura
14     perimetro <- 2 * (base + lado_a)
15     // 3. Salida de resultado
16     Escribir "La base (b) es ", base Sin Saltar
17     Escribir ", la altura (h) es ", altura Sin Saltar
18     Escribir ", el lado (a) es ", lado_a Sin Saltar
19     Escribir ", el area es ", area Sin Saltar
20     Escribir " y el perimetro es ", perimetro
21 FinAlgoritmo

```

Ejercicio 15

Elabore un algoritmo para calcular el promedio final de la materia de Programación. Tome como referencia los porcentajes de evaluación proporcionados por el docente de la materia. La suma de los puntos máximos es: $10 + 10 + 10 + 20 = 50$ puntos. El promedio final se calcula sobre la base de esos 50 puntos. Para obtener el promedio final sobre 10 (o 100 %, dependiendo de cómo se interprete el “promedio final”), primero sumaremos los puntajes obtenidos y luego aplicaremos la conversión necesaria.

Algoritmo 2.41: Pseudocódigo PSeInt: Calculo del promedio final de Programación

```

1  Algoritmo Calculo_Promedio_Programacion
2      // Declarar variables
3      Definir PUNTAJE_MAXIMO Como Real; PUNTAJE_MAXIMO <- 50
4      Definir colab, experi, autonomo, examen Como Real
5      Definir total_obtenido, promedio_final Como Real
6      // 1. Lectura de datos
7      // Pedir al usuario los puntajes obtenidos en cada actividad.
8      Escribir "Ingrese puntaje de Actividad Colaborativa (Max 10):"
9      Leer colab
10     Escribir "Ingrese puntaje de Actividad Experimental (Max 10):"
11     Leer experi
12     Escribir "Ingrese puntaje de Actividad Autonomo (Max 10):"
13     Leer autonomo
14     Escribir "Ingrese puntaje del Examen (Max 20):"
15     Leer examen
16     // 2. Proceso
17     // A. Sumar todos los puntajes obtenidos
18     total_obtenido <- colab + experi + autonomo + examen
19     // B. Calcular el promedio final sobre una base de 10 puntos
20     // Formula: (Total Obtenido / 50) * 10
21     promedio_final <- (total_obtenido / PUNTAJE_MAXIMO) * 10
22     // 3. Salida de resultado
23     Escribir "El total obtenido es: ", total_obtenido Sin Saltar
24     Escribir " de ", PUNTAJE_MAXIMO, " puntos."
25     Escribir "El promedio es: ", promedio_final Sin Saltar
26     Escribir " / 10 puntos."
27 FinAlgoritmo

```



Laboratorio 5

Capítulo 2

2.5. Ejercicios del Laboratorio 5

* Ejercicio 1

Realizar un algoritmo que, ingresando una edad, determine si es mayor o menor de edad.

Algoritmo 2.42: Pseudocódigo PSeInt: Determinación de Mayor o Menor de Edad

```

1 Algoritmo Determinar_Edad
2   // Declarar variables
3   Definir EDAD_LEGAL Como Entero; EDAD_LEGAL <- 18
4   Definir edad Como Entero
5   // 1. Lectura de datos
6   Escribir "Ingrese la edad de la persona:"
7   Leer edad
8   // 2. Proceso
9   Si edad >= EDAD_LEGAL Entonces
10      Escribir "La persona tiene ", edad
11      Escribir " años. Es MAYOR de edad."
12  Sino
13      // En cualquier otro caso, es menor de edad
14      Escribir "La persona tiene ", edad
15      Escribir " años. Es MENOR de edad."
16  FinSi
17 FinAlgoritmo

```

* Ejercicio 2

Realizar un algoritmo que ingrese un nombre, ingrese una edad, determine si es mayor o menor de edad. Muestre los datos concatenados. Ej.: Juan con 19 años es mayor de edad. Operación: Edad >= 18.

Algoritmo 2.43: Pseudocódigo PSeInt: Determinación de Mayor o Menor de Edad con salida concatenada

```

1 Algoritmo Determinar_Edad_Concatenado
2   // Declarar variables
3   Definir EDAD_LEGAL Como Entero; EDAD_LEGAL <- 18
4   Definir nombre Como Caracter
5   Definir edad Como Entero
6   // 1. Lectura de datos
7   Escribir "Ingrese el nombre de la persona:"
8   Leer nombre
9   Escribir "Ingrese la edad de la persona:"
10  Leer edad
11  // 2. Proceso
12  Si edad >= EDAD_LEGAL Entonces
13      Escribir nombre, " con ", edad Sin Saltar
14      Escribir " años es mayor de edad."
15  Sino
16      Escribir nombre, " con ", edad Sin Saltar
17      Escribir " años es menor de edad."
18  FinSi
19 FinAlgoritmo

```

*** Ejercicio 3**

Realizar un algoritmo que ingrese un valor real X, luego, determine que pertenece al intervalo $x \leq 22$.

Algoritmo 2.44: Pseudocódigo PSeInt: Determinación de pertenencia a un rango simple

```

1 Algoritmo Verificar_Rango_Menor_Igual
2   // Declarar el limite superior del rango
3   Definir LIMITE_SUPERIOR Como Real; LIMITE_SUPERIOR <- 22
4   Definir X Como Real
5   // 1. Lectura de datos
6   Escribir "Ingrese un valor real para X:"
7   Leer X
8   // 2. Proceso
9   // Condicion:  $X \leq 22$ 
10  Si  $X \leq \text{LIMITE\_SUPERIOR}$  Entonces
11    Escribir "El valor X = ", X, " SI pertenece al rango."
12  Sino
13    Escribir "El valor X = ", X, " NO pertenece al rango."
14  FinSi
15 FinAlgoritmo

```

*** Ejercicio 4**

Realizar un algoritmo que ingrese un valor real X, determine que pertenece al intervalo $x < 22$

Algoritmo 2.45: Pseudocódigo PSeInt: Determinación de pertenencia a un rango menor estricto

```

1 Algoritmo Verificar_Rango_Menor_22
2   // Declarar el limite superior del rango (exclusivo)
3   Definir LIMITE_SUPERIOR Como Real; LIMITE_SUPERIOR <- 22
4   // Definir la variable de entrada
5   Definir X Como Real
6   // 1. Lectura de datos
7   Escribir "Ingrese un valor real para X:"
8   Leer X
9   // 2. Proceso
10  Si  $X < \text{LIMITE\_SUPERIOR}$  Entonces
11    Escribir "El valor X = ", X
12    Escribir " SI pertenece al rango (es menor a 22)."
13  Sino
14    // En este caso, X es mayor o igual a 22
15    Escribir "El valor X = ", X
16    Escribir " NO pertenece al rango (es mayor o igual a 22)."
17  FinSi
18 FinAlgoritmo

```

*** Ejercicio 5**

Realizar un algoritmo que ingrese un valor real X, determine que pertenece al intervalo $x > -13$

Algoritmo 2.46: Pseudocódigo PSeInt: Determinación de pertenencia a un rango mayor estricto

```

1 Algoritmo Verificar_Rango_Mayor_Menos13
2   // Declarar el limite inferior del rango (exclusivo)
3   Definir LIMITE_INFERIOR Como Real; LIMITE_INFERIOR <- -13
4   Definir X Como Real
5   // 1. Lectura de datos
6   Escribir "Ingrese un valor real para X:"
7   Leer X
8   // 2. Proceso

```

```

9      Si X > LIMITE_INFERIOR Entonces
10         Escribir "El valor X = ", X
11     Escribir " SI pertenece al rango (es mayor a -13)."
12     Sino
13         Escribir "El valor X = ", X
14         Escribir " NO pertenece al rango (es menor o igual a -13)."
15     FinSi
16 FinAlgoritmo

```

* Ejercicio 6

Realizar un algoritmo que ingrese un valor real X, luego determine que pertenece al intervalo $x \geq -13$

Algoritmo 2.47: Pseudocódigo PSeInt: Determinación de pertenencia a un rango mayor o igual

```

1 Algoritmo Verificar_Rango_Mayor_Igual_Menos13
2     // Declarar
3     Definir LIMITE_INFERIOR Como Real; LIMITE_INFERIOR <- -13
4     Definir X Como Real
5     // 1. Lectura de datos
6     Escribir "Ingrese un valor real para X:"
7     Leer X
8     // 2. Proceso
9     Si X >= LIMITE_INFERIOR Entonces
10        Escribir "El valor X = ", X
11        Escribir " SI pertenece al rango (es mayor o igual a -13)."
12    Sino
13        // En este caso, X es estrictamente menor a -13
14        Escribir "El valor X = ", X
15        Escribir " NO pertenece al rango (es menor estricto a -13)."
16    FinSi
17 FinAlgoritmo

```

* Ejercicio 7

Realizar un algoritmo que ingrese un valor real X, determine que pertenece a UNO DE LOS DOS intervalos. Y luego, determine a cuál de los intervalos pertenece, $(x \geq 18) \cup (x < 11)$. Hay que recordar que la unión es la expresión OR.

Algoritmo 2.48: Pseudocódigo PSeInt: Determinación de pertenencia a una unión de intervalos (OR)

```

1 Algoritmo Verificar_Union_Intervalos
2     // Declarar los limites de los intervalos
3     Definir LIMITE_SUPERIOR Como Real; LIMITE_SUPERIOR <- 18
4     Definir LIMITE_INFERIOR Como Real; LIMITE_INFERIOR <- 11
5     Definir X Como Real
6     // 1. Lectura de datos
7     Escribir "Ingrese un valor real para X:"
8     Leer X
9     // 2. Proceso
10    Si X >= LIMITE_SUPERIOR Entonces
11        Escribir "El valor X = ", X
12        Escribir " SI pertenece al intervalo [18, infinito)."
13    Sino Si X < LIMITE_INFERIOR Entonces
14        Escribir "El valor X = ", X
15        Escribir " SI pertenece al intervalo (-infinito, 11)."
16    Sino
17        Escribir "El valor X = ", X
18        Escribir " NO pertenece a la union de intervalos" Sin Saltar

```

```

19         Escribir ", esta en el rango [11, 18)."
```

```

20     FinSi
```

```

21 FinAlgoritmo
```

* Ejercicio 8

Realizar un algoritmo que ingrese un valor real X, determine si pertenece al intervalo: $x \geq -10 \cap x < 5$. Hay que recordar que la intersección es la expresión AND.

Algoritmo 2.49: Pseudocódigo PSeInt: Determinación de pertenencia a una intersección de intervalos (AND)

```

1 Algoritmo Verificar_Interseccion_Intervalo
2     // Declarar
3     Definir LIMITE_INFERIOR Como Real; LIMITE_INFERIOR <- -10
4     Definir LIMITE_SUPERIOR Como Real; LIMITE_SUPERIOR <- 5
5     Definir X Como Real
6     // 1. Lectura de datos
7     Escribir "Ingrese un valor real para X:"
8     Leer X
9     // 2. Proceso
10    Si (X >= LIMITE_INFERIOR) Y (X < LIMITE_SUPERIOR) Entonces
11        Escribir "El valor X = ", X
12        Escribir " SI pertenece al intervalo [-10, 5)."
```

```

13    Sino
```

```

14        Escribir "El valor X = ", X
```

```

15        Escribir " NO pertenece al intervalo [-10, 5)."
```

```

16    FinSi
```

```

17 FinAlgoritmo
```

* Ejercicio 9

Realizar un algoritmo que ingrese un valor real X, determine si pertenece al intervalo: $x < 13 \cap x \geq 9$. Hay que recordar que la intersección es la expresión AND.

Algoritmo 2.50: Pseudocódigo PSeInt: Determinación de pertenencia a una intersección (AND) de intervalos

```

1 Algoritmo Verificar_Interseccion_9_13
2     // Declarar intervalo cerrado [9, 13)
3     Definir LIMITE_INFERIOR Como Real; LIMITE_INFERIOR <- 9
4     Definir LIMITE_SUPERIOR Como Real; LIMITE_SUPERIOR <- 13
5     Definir X Como Real
6     // 1. Lectura de datos
7     Escribir "Ingrese un valor real para X:"
8     Leer X
9     // 2. Proceso
10    Si (X >= LIMITE_INFERIOR) Y (X < LIMITE_SUPERIOR) Entonces
11        Escribir "El valor X = ", X Sin Saltar
12        Escribir " SI pertenece al intervalo [9, 13) (interseccion)."
```

```

13    Sino
```

```

14        Escribir "El valor X = ", X Sin Saltar
```

```

15        Escribir " NO pertenece al intervalo [9, 13)."
```

```

16    FinSi
```

```

17 FinAlgoritmo
```

*** Ejercicio 10**

Realizar un algoritmo que ingrese un valor real X, determine que pertenece a UNO DE LOS DOS intervalos. Y luego, determine a cuál de los intervalos pertenece. $x < 0 \cup x \geq 3$. Hay que recordar que la unión es la expresión OR.

Algoritmo 2.51: Pseudocódigo PSeInt: Determinación de pertenencia a una unión de intervalos (OR)

```

1  Algoritmo Verificar_Union_Intervalos_0_3
2      // Declarar los intervalos
3      Definir LIMITE_SUPERIOR Como Real; LIMITE_SUPERIOR <- 3
4      Definir LIMITE_INFERIOR Como Real; LIMITE_INFERIOR <- 0
5      Definir X Como Real
6      // 1. Lectura de datos
7      Escribir "Ingrese un valor X:"
8      Leer X
9      // 2. Proceso
10     // Evaluar: Intervalo (-infinito, 0)
11     Si X < LIMITE_INFERIOR Entonces
12         Escribir "El valor X = ", X
13         Escribir " SI pertenece al intervalo (-infinito, 0)."
14     // Evaluar: Intervalo [3, infinito)
15     Sino Si X >= LIMITE_SUPERIOR Entonces
16         Escribir "El valor X = ", X
17         Escribir " SI pertenece al intervalo [3, infinito)."
18     // Si no cumple ninguna de las anteriores,
19     // se encuentra en el intervalo central [0, 3)
20     Sino
21         Escribir "El valor X = ", X
22         Escribir " NO pertenece a la union de intervalos, " Sin Saltar
23         Escribir "se encuentra en el rango [0, 3)."
24     FinSi
25 FinAlgoritmo

```

*** Ejercicio 11**

Realice un algoritmo que permita el ingreso de un valor de temperatura (el programa debe indicar qué debe ingresar / o qué desea el usuario ingresar, grados Kelvin, Fahrenheit o Centígrados) luego devuelva su valor correspondiente en los dos grados faltantes.

Algoritmo 2.52: Pseudocódigo PSeInt: Conversor de unidades de temperatura

```

1  Algoritmo Conversor_Temperatura
2      // Declarar variables de proceso
3      Definir temp_original, temp_C, temp_F, temp_K Como Real
4      Definir opcion Como Entero
5      // 1. Lectura de datos
6      Escribir "--- Conversor de Temperatura ---"
7      Escribir "Ingrese la unidad de temperatura que desea convertir:"
8      Escribir "1: Grados Centígrados (Celsius)"
9      Escribir "2: Grados Fahrenheit"
10     Escribir "3: Grados Kelvin"
11     Leer opcion
12     Escribir "Ingrese el valor de la temperatura:"
13     Leer temp_original
14     // 2. Proceso
15     Segun opcion Hacer
16         // Caso 1: Entrada en Centígrados (C)
17         1:

```

```

18         temp_C <- temp_original
19         // Comuto a Fahrenheit: F = C * 1.8 + 32
20         temp_F <- (temp_C * 1.8) + 32
21         // Comuto a Kelvin: K = C + 273.15
22         temp_K <- temp_C + 273.15
23     // Caso 2: Entrada en Fahrenheit (F)
24     2:
25         temp_F <- temp_original
26         // Comuto a Centigrados: C = (F - 32) / 1.8
27         temp_C <- (temp_F - 32) / 1.8
28         // Comuto a Kelvin: K = C + 273.15
29         temp_K <- temp_C + 273.15
30     // Caso 3: Entrada en Kelvin (K)
31     3:
32         temp_K <- temp_original
33         // Comuto a Centigrados: C = K - 273.15
34         temp_C <- temp_K - 273.15
35         // Comuto a Fahrenheit: F = C * 1.8 + 32
36         temp_F <- (temp_C * 1.8) + 32
37     De Otro Modo:
38         Escribir "ERROR: valor incorrecto."
39     FinSegun
40     // 3. Salida de resultado
41     // Solo mostrar resultados si la opcion fue correcta
42     Si (opcion >= 1) Y (opcion <= 3) Entonces
43         Escribir "--- Resultado de la conversion ---"
44         Escribir "Temperatura en Centigrados: ", temp_C, " C"
45         Escribir "Temperatura en Fahrenheit: ", temp_F, " F"
46         Escribir "Temperatura en Kelvin: ", temp_K, " K"
47     FinSi
48 FinAlgoritmo

```

* Ejercicio 12

Se solicita un número por teclado y verificar que el número se encuentre entre el rango, mayor a 1000 (sin incluir) y 1999 (incluido), (utilice la tabla de verdad AND y OR para su análisis).

Análisis: El rango solicitado es $X \in (1000, 1999]$. Esto significa que el número X debe cumplir dos condiciones simultáneamente: Condición 1 (Límite Inferior): X debe ser mayor a 1000 ($X > 1000$). Condición 2 (Límite Superior): X debe ser menor o igual a 1999 ($X \leq 1999$). Para que el número pertenezca al rango, la Condición 1 y la Condición 2 deben ser Verdaderas.

El operador AND (Y) garantiza que solo cuando ambas condiciones son ciertas, el resultado final es cierto:

Tabla 2.9: Tabla de verdad (Operador AND para rango)

Condición 1 ($X > 1000$)	Condición 2 ($X \leq 1999$)	Resultado (Pertenece al Rango)
Falso	Falso	Falso
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Verdadero	Verdadero	Verdadero

Algoritmo 2.53: Pseudocódigo PSeInt: Verificación de rango con AND

```

1 Algoritmo Verificar_Rango_Mil
2     // Declarar los limites del rango (1000, 1999]

```

```

3 Definir LIMITE_INF Como Entero; LIMITE_INF <- 1000
4 Definir LIMITE_SUP Como Entero; LIMITE_SUP <- 1999
5 Definir numero Como Entero
6 // 1. Lectura de datos
7 Escribir "Ingrese un valor entero:"
8 Leer numero
9 // 2. Proceso
10 // Condicionar: (numero > 1000) and (numero <= 1999)
11 Si (numero > LIMITE_INF) Y (numero <= LIMITE_SUP) Entonces
12     Escribir "El numero ", numero
13     Escribir " SI se encuentra entre 1000" Sin Saltar
14     Escribir "(sin incluir) y 1999 (incluido).".
15 Sino
16     Escribir "El numero ", numero
17     Escribir " NO se encuentra en el rango especificado."
18 FinSi
19 FinAlgoritmo

```

* Ejercicio 13

13. Se solicita un número por teclado y verificar que el número se encuentre entre el rango, menor a 1000 (incluido) o mayor 1999 (sin incluir), utilice la tabla de verdad AND y OR para su análisis.

Análisis: El rango solicitado es $X \in (-\infty, 1000] \cup (1999, \infty)$. Esto significa que el número X debe cumplir al menos una de las dos condiciones para pertenecer al conjunto: Condición 1 (Límite Superior): X debe ser menor o igual a 1000 ($X \leq 1000$). Condición 2 (Límite Inferior): X debe ser mayor a 1999 ($X > 1999$). Para que el número pertenezca al rango, la Condición 1 o la Condición 2 deben ser Verdaderas.

El operador OR (O) garantiza que si al menos una de las condiciones es cierta, el resultado final es cierto:

Tabla 2.10: Tabla de Verdad (Operador OR para rangos disjuntos)

Condición 1 ($X \leq 1000$)	Condición 2 ($X > 1999$)	Resultado (Pertenece al Rango)
Falso	Falso	Falso
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Verdadero	Verdadero	Verdadero

Algoritmo 2.54: Pseudocódigo PSeInt: Verificación de rango con OR

```

1 Algoritmo Verificar_Rango_Disjunto
2 // Declarar los limites del rango disjunto
3 Definir LIMITE_SUP_IN Como Entero
4 LIMITE_SUP_IN <- 1000
5 Definir LIMITE_INF_EX Como Entero
6 LIMITE_INF_EX <- 1999
7 Definir numero Como Entero
8 // 1. Lectura de datos
9 Escribir "Ingrese un valor:"
10 Leer numero
11 // 2. Proceso
12 // Condicionar: (numero <= 1000) or (numero > 1999)
13 // Se usa el operador logico O para verificar
14 // si pertenece a la union

```

```

15  Si (numero <= LIMITE_SUP_IN) O (numero > LIMITE_INF_EX) Entonces
16      Escribir "El valor ", numero
17      Escribir " SI pertenece al rango disjunto " Sin Saltar
18      Escribir "(menor o igual a 1000 O mayor a 1999).".
19  Sino
20      // El valor que no pertenece se encuentra
21      // en el rango central: (1000, 1999]
22      Escribir "El valor ", numero
23      Escribir " NO pertenece al rango disjunto " Sin Saltar
24      Escribir "esta entre 1000 y 1999."
25  FinSi
26  FinAlgoritmo

```

* Ejercicio 14

Elabore un algoritmo que permita el ingreso de un valor de tres cifras. Luego determine si el número es un valor Armstrong de tres cifras. Para lograrlo, el número ingresado debe descomponerse en unidad, decena, centena. Luego cada uno de ellos se eleva al cubo (por ser de tres cifras) y se suman, para reconstruir el número. Si dicho número reconstruido y el valor ingresado son iguales, entonces es un número Armstrong.

Análisis: El algoritmo debe determinar si un número de tres cifras es un número de Armstrong. La clave está en la descomposición usando los operadores MOD (módulo o residuo) y TRUNC (truncar o parte entera). Un Número Armstrong de tres cifras es aquel cuya suma de los cubos de sus dígitos es igual al número original. Por ejemplo, 153 es Armstrong porque $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

Algoritmo 2.55: Pseudocódigo PSeInt: Verificación de valor Armstrong

```

1  Algoritmo Verificar_Armstrong
2      // Declarar variables de proceso
3      Definir num_entrada, unidad, decena, centena Como Entero
4      Definir resultado_cubos Como Entero
5      // 1. Lectura de datos
6      Escribir "Ingrese un numero de tres cifras:"
7      Leer num_entrada
8      // 2. Proceso
9      // La descomposicion se basa en la aritmetica modular (MOD)
10     // La Unidad: residuo de dividir por 10
11     unidad <- num_entrada MOD 10
12     // La Decena: Truncar la division por 10, luego residuo por 10
13     decena <- TRUNC(num_entrada / 10) MOD 10
14     // La Centena: Truncar la division por 100
15     centena <- TRUNC(num_entrada / 100)
16     // 3. Proceso: Reconstruir el valor sumando los cubos
17     // Puede ser u*u*u + d*d*d + c*c*c
18     resultado_cubos <- (unidad ^ 3) + (decena ^ 3) + (centena ^ 3)
19     // 4. Verificar y salir
20     // Si el valor de entrada es igual al resultado
21     // de la suma de los cubos
22     Si num_entrada == resultado_cubos Entonces
23         Escribir num_entrada, " SI es un valor ARMSTRONG."
24     SiNo
25         Escribir num_entrada, " NO es un valor ARMSTRONG."
26     Fin Si
27     // Se muestra el resultado
28     Escribir "Resultado de la suma de cubos es: ", resultado_cubos
29  FinAlgoritmo

```

*** Ejercicio 15**

Elabore un algoritmo que permita recorrer todos los números de tres cifras. Luego determine todos los valores Armstrong de tres cifras. Para lograrlo, el número ingresado debe descomponerse en unidad, decena, centena. Luego cada uno de ellos se eleva al cubo (por ser de tres cifras) y se suman, para reconstruir el número. Si dicho número reconstruido y el valor recorrido son iguales, entonces es un número Armstrong.

Algoritmo 2.56: Pseudocódigo PSeInt: Búsqueda de números Armstrong de tres cifras

```

1 Algoritmo Encontrar_Armstrongs
2   // Declarar variables de proceso
3   // x es el contador que va de 100 a 999
4   Definir x, num_actual, u, d, c Como Entero
5   Definir resultado_cubos Como Entero
6   Escribir "--- Valores ARMSTRONG de 3 Cifras ---"
7   // 1. Proceso
8   // Recorrer todos los valores desde el 100 hasta el 999
9   Para x = 100 Hasta 999 Con Paso 1 Hacer
10      num_actual <- x // 'num_actual' toma el valor a verificar
11      // A. Descomponer el valor
12      u <- num_actual MOD 10
13      d <- TRUNC(num_actual / 10) MOD 10
14      c <- TRUNC(num_actual / 100)
15      // B. Reconstruir el valor sumando los cubos
16      resultado_cubos <- (u*u*u) + (d*d*d) + (c*c*c)
17      // C. Verificar y Salir
18      // Si el valor actual es igual al resultado
19      // de la suma de los cubos
20      Si num_actual == resultado_cubos Entonces
21          // 2. Salida de resultado (solo si es Armstrong)
22          Escribir num_actual, " SI es un numero ARMSTRONG."
23      Fin Si
24  Fin Para
25  Escribir "--- Proceso finalizado ---"
26 FinAlgoritmo

```

*** Ejercicio 16**

Elabore un algoritmo que permita el ingreso de un valor de cuatro cifras. Luego determine si el número es un valor Armstrong de cuatro cifras. Para lograrlo, el número ingresado debe descomponerse en unidad, decena, centena y unidad de mil (m). Luego cada uno de ellos se eleva a la cuarta (por ser de cuatro cifras) y se suman, para reconstruir el número. Si dicho número reconstruido y el valor ingresado son iguales, entonces es un número Armstrong.

Algoritmo 2.57: Pseudocódigo PSeInt: Verificación de número Armstrong de cuatro cifras

```

1 Algoritmo Verificar_Armstrong_Cuatro
2   // Declarar variables de proceso
3   Definir num_entrada, u, d, c, m Como Entero
4   // u:unidad, d:decena, c:centena, m:mil
5   Definir resultado_potencias Como Entero
6   // 1. Lectura de datos
7   Escribir "Ingrese un valor de cuatro cifras (entre 1000 y 9999):"
8   Leer num_entrada
9   // 2. Proceso: Descomponer el valor en sus cifras
10  // A. Obtener la Unidad (u): residuo de dividir por 10
11  u <- num_entrada MOD 10
12  // B. Obtener la Decena (d): Truncar (/10), luego residuo por 10

```

```

13     d <- TRUNC(num_entrada / 10) MOD 10
14     // C. Obtener la Centena (c): Truncar (/100), luego residuo por 10
15     c <- TRUNC(num_entrada / 100) MOD 10
16     // D. Obtener la Unidad de mil (m): Truncar la division por 1000
17     m <- TRUNC(num_entrada / 1000)
18     // 3. Proceso: Reconstruir el valor sumando las cuartas potencias
19     // (potencia ^ 4)
20     resultado_potencias <- (u ^ 4) + (d ^ 4) + (c ^ 4) + (m ^ 4)
21     // 4. Verificar y Salir
22     // Valor de entrada es igual al resultado de suma y potencias
23     Si num_entrada == resultado_potencias Entonces
24         Escribir num_entrada
25     Escribir " SI es un numero ARMSTRONG de cuatro cifras."
26     SiNo
27         Escribir num_entrada
28     Escribir " NO es un numero ARMSTRONG de cuatro cifras."
29     Fin Si
30     // Opcional: Mostrar el resultado del computo
31     Escribir "El resultado es: ", resultado_potencias
32 FinAlgoritmo

```

* Ejercicio 17

Elabore un algoritmo que permita recorrer todos los números de cuatro cifras. Luego determine todos los valores Armstrong de cuatro cifras. Para lograrlo, el número ingresado debe descomponerse en unidad, decena, centena y unidades de mil. Luego cada uno de ellos se eleva a la cuarta (por ser de cuatro cifras) y se suman, para reconstruir el número. Si dicho número reconstruido y el valor recorrido son iguales, entonces es un número Armstrong.

Algoritmo 2.58: Pseudocódigo PSeInt: Búsqueda de números Armstrong de cuatro cifras

```

1  Algoritmo Encontrar_Armstrongs_Cuatro
2      // Declarar variables de proceso
3      // x es el contador que va de 1000 a 9999
4      Definir x, num_actual, u, d, c, m Como Entero
5      // u, d, c, m son los digitos
6      Definir resultado_potencias Como Entero
7      Escribir "--- Búsqueda de valores ARMSTRONG de 4 Cifras ---"
8      // 1. Proceso (Uso de estructura Para)
9      // Recorrer todos los valores desde el 1000 hasta el 9999
10     Para x = 1000 Hasta 9999 Con Paso 1 Hacer
11         num_actual <- x // 'num_actual' toma el valor a verificar
12         // A. Descomponer el valor en sus digitos
13         // u: Unidad, d: Decena, c: Centena, m: Unidad de Mil
14         u <- num_actual MOD 10
15         d <- TRUNC(num_actual / 10) MOD 10
16         c <- TRUNC(num_actual / 100) MOD 10
17         m <- TRUNC(num_actual / 1000)
18         // B. Reconstruir el valor sumando las cuartas potencias
19         resultado_potencias <- (u^4) + (d^4) + (c^4) + (m^4)
20         // C. Verificar y Mostrar
21         Si num_actual == resultado_potencias Entonces
22             // 2. Salida de resultado (solo si es Armstrong)
23             Escribir num_actual, " SI es un numero ARMSTRONG."
24         Fin Si
25     Fin Para
26     Escribir "--- Proceso finalizado ---"
27 FinAlgoritmo

```

*** Ejercicio 18**

Elabore un algoritmo que permita recorrer todos los números de cinco cifras. Luego determine todos los valores Armstrong de cinco cifras. Para lograrlo, el número ingresado debe descomponerse en unidad, decena, centena y unidades de mil, decena de miles. Luego cada uno de ellos se eleva a la quinta (por ser de cinco cifras) y se suman, para reconstruir el número. Si dicho número reconstruido y el valor recorrido son iguales, entonces es un número Armstrong.

Algoritmo 2.59: Pseudocódigo PSeInt: Búsqueda de números Armstrong de cinco cifras

```

1  Algoritmo Encontrar_Armstrongs_Cinco
2      // Declarar variables de proceso
3      // x es el contador que va de 10000 a 99999
4      Definir x, num_actual, u, d, c, m, dm Como Entero
5      // dm: decena de mil
6      Definir resultado_potencias Como Entero
7      Escribir "--- Valores ARMSTRONG de 5 Cifras ---"
8      // 1. Proceso (Uso de estructura Para)
9      // Recorrer todos los valores desde el 10000 hasta el 99999
10     Para x = 10000 Hasta 99999 Con Paso 1 Hacer
11         num_actual <- x // 'num_actual' toma el valor a verificar
12         // A. Descomponer el valor en sus digitos
13         // u: Unidad, d: Decena, c: Centena,
14         // m: Mil, dm: Decena de Mil
15         u <- num_actual MOD 10
16         d <- TRUNC(num_actual / 10) MOD 10
17         c <- TRUNC(num_actual / 100) MOD 10
18         m <- TRUNC(num_actual / 1000) MOD 10
19         dm <- TRUNC(num_actual / 10000)
20         // B. Reconstruir el valor
21         // sumando las quintas potencias (digito ^ 5)
22         resultado_potencias <- (u^5)+(d^5)+(c^5)+(m^5)+(dm^5)
23         // C. Verificar y Mostrar
24         // Si el valor actual es igual al resultado
25         // de la suma de las potencias
26         Si num_actual == resultado_potencias Entonces
27             // 2. Salida de resultado (solo si es Armstrong)
28             Escribir num_actual, " SI es un numero ARMSTRONG."
29         Fin Si
30     Fin Para
31     Escribir "--- Proceso finalizado ---"
32 FinAlgoritmo

```



Laboratorio 6

Capítulo 2

2.6. Ejercicios del Laboratorio 6

* Ejercicio 1

Diseñar un algoritmo que imprima la serie de números: 3 7 11 15 19 23 27 31 35 39 (usando el bucle Para)

Algoritmo 2.60: Pseudocódigo PSeInt: Imprimir serie numérica con Para (Incremento de 4)

```

1 Algoritmo Imprimir_Serie
2     // Declarar variables de proceso
3     // 'numero' es quien almacena el valor actual de la serie
4     Definir numero Como Entero
5     Escribir "--- Serie de valores: 3, 7, 11, ..., 39 ---"
6     // Recorrer los valores de 3 hasta 39 con un incremento de 4
7     Para numero = 3 Hasta 39 Con Paso 4 Hacer
8         Escribir Sin Saltar numero, " "
9     Fin Para
10 FinAlgoritmo

```

* Ejercicio 2

Diseñar un algoritmo que imprima la serie de números: 3 7 11 15 19 23 27 31 35 39 (usando el bucle Mientras)

Algoritmo 2.61: Pseudocódigo PSeInt: Imprimir serie numérica con Mientras (Incremento de 4)

```

1 Algoritmo Imprimir_Serie_Mientras
2     // Declarar variables de proceso
3     Definir numero Como Entero
4     Escribir "--- Serie de valores: 3, 7, 11, ..., 39 ---"
5     // 1. Inicializar el primer valor de la serie
6     numero <- 3
7     // 2. Permanece en el bucle siempre que el valor no exceda 39
8     Mientras numero <= 39 Hacer
9         Escribir Sin Saltar numero, " "
10        numero <- numero + 4
11    Fin Mientras
12 FinAlgoritmo

```

* Ejercicio 3

Diseñar un algoritmo que imprima la serie de números: 3 7 11 15 19 23 27 31 35 39 (usando el bucle Repetir-Hasta)

Algoritmo 2.62: Pseudocódigo PSeInt: Imprimir serie numérica con Repetir-Hasta (Incremento de 4)

```

1 Algoritmo Imprimir_Serie_Repetir
2     // Declarar variables de proceso
3     Definir numero Como Entero
4     Escribir "--- Serie de valores: 3, 7, 11, ..., 39 ---"
5     // 1. Inicializar el primer valor de la serie
6     numero <- 3
7     // 2. Se detiene cuando es > 39
8     Repetir
9         Escribir Sin Saltar numero, " "
10        numero <- numero + 4
11    // 3. Condicion de Salida
12    Hasta Que numero > 39
13 FinAlgoritmo

```

*** Ejercicio 4**

Diseñar un algoritmo que imprima la siguiente serie de números de forma descendente: 3 7 11 15 19 23 27 31 35 39, se mostrarían así: 39 35 31 27 23 19 15 11 7 3, usando el bucle Para.

Algoritmo 2.63: Pseudocódigo PSeInt: Imprimir serie numérica descendente con Para (Decremento de 4)

```

1 Algoritmo Imprimir_Serie_Descendente
2   // 'numero' es quien almacena el valor actual de la serie
3   Definir numero Como Entero
4   Escribir "--- Serie de valores descendente: 39, 35, ..., 3 ---"
5   // Recorrer los valores de 39 hasta 3 con un decremento de -4
6   // El 'Paso' es un valor negativo en el orden descendente
7   Para numero = 39 Hasta 3 Con Paso -4 Hacer
8       Escribir Sin Saltar numero, " "
9   Fin Para
10 FinAlgoritmo

```

*** Ejercicio 5**

Diseñar un algoritmo que imprima la siguiente serie de números de forma descendente: 3 7 11 15 19 23 27 31 35 39, se mostrarían así: 39 35 31 27 23 19 15 11 7 3, usando el bucle Mientras.

Algoritmo 2.64: Pseudocódigo PSeInt: Imprimir serie numérica descendente con Mientras (Decremento de 4)

```

1 Algoritmo Imprimir_Serie_Descendente_Mientras
2   Definir numero Como Entero
3   // 'numero' es quien almacena el valor actual
4   Escribir "--- Serie de valores descendente: 39, 35, ..., 3 ---"
5   // 1. Inicializar el primer valor de la serie
6   numero <- 39
7   // 2. Permanece en el bucle siempre que
8   // el valor sea mayor o igual a 3
9   Mientras numero >= 3 Hacer
10       Escribir Sin Saltar numero, " "
11       // Decrementar el valor para el siguiente termino
12       numero <- numero - 4
13   Fin Mientras
14 FinAlgoritmo

```

*** Ejercicio 6**

Diseñar un algoritmo que imprima la siguiente serie de números de forma descendente: 3 7 11 15 19 23 27 31 35 39, se mostrarían así: 39 35 31 27 23 19 15 11 7 3, usando el bucle Repetir-Hasta.

Algoritmo 2.65: Pseudocódigo PSeInt: Imprimir serie numérica descendente con Repetir-Hasta (Decremento de 4)

```

1 Algoritmo Imprimir_Serie_Descendente_Repetir
2   Definir numero Como Entero
3   // 'numero' es quien almacena el valor actual
4   Escribir "--- Serie de valores descendente: 39, 35, ..., 3 ---"
5   // 1. Inicializar el primer valor de la serie
6   numero <- 39
7   // 2. Se detiene cuando el valor de 'numero' es menor que 3
8   Repetir
9       Escribir Sin Saltar numero, " "
10      // Decrementar el valor para el siguiente termino
11      numero <- numero - 4
12      // 3. Condicion de Salida
13      Hasta Que numero < 3
14 FinAlgoritmo

```

* Ejercicio 7

Diseñar un algoritmo que imprima la serie de números 8 11 14 17 20 23 26 29 Mostrar la salida indicando en la parte izquierda el número de iteración ejemplo: número 1 = 8, número 2 = 11 (Primera forma).

Algoritmo 2.66: Pseudocódigo PSeInt: Imprimir serie numérica (Incremento de 3) con Contador

```

1 Algoritmo Imprimir_Serie_Contador
2   // Declarar variables de proceso
3   // 'termino' almacena el valor actual de la serie (8, 11, ...)
4   Definir termino Como Entero
5   // 'iteracion' almacena el numero de la posicion (1, 2, ...)
6   Definir iteracion Como Entero
7   Escribir "--- Serie de valores: 8, 11, ..., 29 ---"
8   // 1. Inicializar el contador de iteraciones
9   iteracion <- 1
10  // 2. Proceso
11  // Recorrer los valores de 8 hasta 29 con un incremento de 3
12  Para termino = 8 Hasta 29 Con Paso 3 Hacer
13    Escribir "Numero ", iteracion, " = ", termino
14    iteracion <- iteracion + 1
15  Fin Para
16 FinAlgoritmo

```

* Ejercicio 8

Diseñar un algoritmo que imprima la serie de números 8 11 14 17 20 23 26 29 Mostrar la salida indicando en la parte izquierda el número de iteración ejemplo: número 1 = 8, número 2 = 11 (Segunda forma).

Algoritmo 2.67: Pseudocódigo PSeInt: Imprimir serie numérica con Contador (Usando Mientras)

```

1 Algoritmo Imprimir_Serie_Contador_Mientras
2   // Declarar variables de proceso
3   Definir termino, iteracion Como Entero
4   Escribir "--- Serie de valores: 8, 11, ..., 29 ---"
5   // 1. Inicializar variables
6   termino <- 8
7   iteracion <- 1
8   // 2. Permanece en el bucle mientras el valor no exceda 29
9   Mientras termino <= 29 Hacer
10    Escribir "Numero ", iteracion, " = ", termino
11    // Actualizar el valor y la iteracion
12    termino <- termino + 3
13    iteracion <- iteracion + 1
14  Fin Mientras
15  Escribir "--- Proceso finalizado ---"
16 FinAlgoritmo

```

* Ejercicio 9

Diseñar un algoritmo que muestre la serie de números: 2 7 12 17 22 27 32. Mostrar la salida indicando en la parte izquierda la cantidad de repetición.

Algoritmo 2.68: Pseudocódigo PSeInt: Imprimir serie numérica (Incremento de 5) con Contador

```

1 Algoritmo Imprimir_Serie_Contador_5
2   // Declarar variables de proceso
3   // 'termino' almacena el valor actual de la serie (2, 7, ...)
4   Definir termino Como Entero

```

```

5      // 'iteracion' almacena el numero de la repeticion (1, 2, ...)
6      Definir iteracion Como Entero
7      Escribir "--- Serie de valores: 2, 7, 12, ..., 32 ---"
8      // 1. Inicializar el contador de repeticiones
9      iteracion <- 1
10     // 2. Proceso (Usando el bucle Para para generar la serie)
11     // Recorrer los valores de 2 hasta 32 con un incremento de 5
12     Para termino = 2 Hasta 32 Con Paso 5 Hacer
13         // A. Salida de resultado
14         Escribir "Repeticion ", iteracion, " = ", termino
15         // B. Incrementar el contador
16         iteracion <- iteracion + 1
17     Fin Para
18 FinAlgoritmo

```

* Ejercicio 10

Diseñar un algoritmo que dado un número ingresado por teclado, determine la tabla de multiplicar de dicho número desde 0 hasta 12.

Algoritmo 2.69: Pseudocódigo PSeInt: Generar tabla de multiplicar

```

1 Algoritmo Tabla_Multiplicar
2     // Declarar variables de proceso
3     Definir num_tabla, contador, resultado Como Entero
4     Escribir "--- Generar Tabla de Multiplicar (0 a 12) ---"
5     // 1. Lectura de datos
6     Escribir "Ingrese el numero del cual desea la tabla:"
7     Leer num_tabla
8     // 2. Proceso (Uso de estructura Para)
9     // Recorrer los valores del multiplicador desde 0 hasta 12
10    Para contador = 0 Hasta 12 Con Paso 1 Hacer
11        // A. Calcular el resultado
12        resultado <- num_tabla * contador
13        // B. Salida de resultado: Mostrar la operacion y el resultado
14        Escribir num_tabla, " x ", contador, " = ", resultado
15    Fin Para
16 FinAlgoritmo

```

* Ejercicio 11

La empresa eléctrica desea calcular el consumo generado por sus abonados. Por cada uno de ellos se ingresará el código del abonado y el consumo de electricidad en Kw/h. Se pide: a) Ingresar abonados o clientes hasta que se ingrese una marca de fin de datos, en la cual el código del abonado sea – 99. b) Mostrar el listado de los abonados, el consumo en Kw/h, y también mostrar el consumo en dólares, pero teniendo en cuenta que, si el consumo ingresado es mayor a 125 Kw/h cobrarlo a 6 centavos, en otro caso cobrarlo a 4 centavos. c) Mostrar el número de abonados ingresados. d) Mostrar el consumo de Kw/h de todos los abonados. e) Mostrar el total de cobro que ha realizado la empresa eléctrica.

Algoritmo 2.70: Pseudocódigo PSeInt: Cálculo de consumo eléctrico y cobro

```

1 Algoritmo Consumo_Electrico
2     // Declarar variables de control y acumuladores
3     Definir codigo_abonado, num_abonados Como Entero
4     Definir consumo_kwh, cobro_abonado Como Real
5     Definir total_cobro, total_kwh Como Real
6     // Declarar constantes para tarifas
7     Definir TARIFA_ALTA Como Real

```

```

8     TARIFA_ALTA <- 0.06 // $0.06 por kWh
9     Definir TARIFA_BAJA Como Real
10    TARIFA_BAJA <- 0.04 // $0.04 por kWh
11    Definir LIMITE_TARIFA Como Real; LIMITE_TARIFA <- 125
12    // 1. Inicializar contadores y acumuladores
13    num_abonados <- 0
14    total_cobro <- 0.0
15    total_kwh <- 0.0
16    Escribir "---- INICIO DE INGRESO DE ABONADOS ----"
17    Escribir "Ingrese el codigo del abonado (-99 para finalizar):"
18    Leer codigo_abonado
19    // 2. Proceso (Uso de estructura Mientras para ingreso de datos)
20    // Permanecer en el bucle mientras
21    // el codigo NO sea la marca de fin (-99)
22    Mientras codigo_abonado <> -99 Hacer
23        // A. Leer los datos del abonado actual
24        Escribir "Ingrese el consumo de electricidad en Kw/h:"
25        Leer consumo_kwh
26        // B. Determinar la tarifa
27        // y Calcular el cobro (Usando Si-Sino)
28        Si consumo_kwh > LIMITE_TARIFA Entonces
29            // Consumo mayor a 125 kWh -> Tarifa alta ($0.06)
30            cobro_abonado <- consumo_kwh * TARIFA_ALTA
31        SiNo
32            // Consumo menor o igual a
33            // 125 kWh -> Tarifa baja ($0.04)
34            cobro_abonado <- consumo_kwh * TARIFA_BAJA
35        Fin Si
36        // C. Acumular y contar
37        num_abonados <- num_abonados + 1
38        // Contar el abonado ingresado
39        total_cobro <- total_cobro + cobro_abonado
40        // Acumular el cobro total
41        total_kwh <- total_kwh + consumo_kwh
42        // Acumular el consumo total
43        // D. Mostrar el listado del abonado procesado (Punto b)
44        Escribir "-----"
45        Escribir "Abonado: ", codigo_abonado
46        Escribir "Consumo (Kw/h): ", consumo_kwh
47        Escribir "Cobro (Dolares): $", cobro_abonado
48        Escribir "-----"
49        // E. Leer el siguiente codigo para continuar o salir
50        Escribir "Ingrese el codigo del siguiente abonado" Sin Saltar
51        Escribir "(-99 para finalizar):"
52        Leer codigo_abonado
53    Fin Mientras
54    // 3. Salida de resultados finales (Puntos c, d, e)
55    Escribir "--- RESUMEN DE LA EMPRESA ELECTRICA ---"
56    Escribir "c) Numero de abonados ingresados: ", num_abonados
57    Escribir "d) Consumo total de Kw/h de todos los abonados: "
58    Escribir total_kwh, " Kw/h"
59    Escribir "e) Total de cobro realizado por la empresa: $"
60    Escribir total_cobro
61 FinAlgoritmo

```

* Ejercicio 12

Diseñar un algoritmo que presente la serie de Fibonacci, permitiendo solicitar al usuario un valor que representa la cantidad límite n . Por ejemplo, si se ingresa 10, se mostrará: 0 1 1 2 3 5 8 13 21 34. Para esto las variables iniciales serán $a = -1$, $b = 1$, con la fórmula $c = a + b$, n repetidamente.

Algoritmo 2.71: Pseudocódigo PSeInt: Generar Serie de Fibonacci hasta N terminos

```

1  Algoritmo Serie_Fibonacci
2      // Declarar variables
3      Definir a, b, c, n, contador Como Entero
4      Escribir "--- Generar Serie de Fibonacci ---"
5      // 1. Lectura de datos
6      Escribir "Ingrese la cantidad limite de terminos (N):"
7      Leer n
8      // 2. Inicializar los valores base: a=-1, b=1
9      a <- -1
10     b <- 1
11     // 3. Imprimir la cabecera
12     Escribir "Serie de Fibonacci (", n, " terminos):"
13     // 4. Proceso
14     // Recorrer el bucle N veces para generar los N terminos
15     Para contador = 1 Hasta n Con Paso 1 Hacer
16         // A. Calcular el siguiente termino de la serie
17         c <- a + b
18         // B. Imprimir el termino actual
19         Escribir Sin Saltar c, " "
20         // C. Actualizar los valores base
21         a <- b // El valor antiguo de b pasa a ser el nuevo a
22         b <- c // El nuevo termino c pasa a ser el nuevo b
23     Fin Para
24 FinAlgoritmo

```

* Ejercicio 13

Diseñar un algoritmo que presente la serie de Ulam, para esto, se solicita al usuario un número n . Y la generación de la serie empieza analizando el número, si es par se lo divide para 2 y si es impar se multiplica por 3 y luego se suma 1. Todo repetidamente hasta llegar a 1.

Análisis: La serie sigue dos reglas condicionales: Si N es par: $N \leftarrow N/2$. Si N es impar: $N \leftarrow 3 \times N + 1$

Algoritmo 2.72: Pseudocódigo PSeInt: Generar serie de Ulam o Collatz

```

1  Algoritmo Serie_Ulam
2      // Declarar variables
3      Definir n, contador Como Entero
4      // n es el numero actual en la serie
5      Escribir "--- Generar Serie de Ulam o Collatz ---"
6      // 1. Lectura de datos
7      Escribir "Ingrese un numero entero positivo (n)"
8      Escribir " para iniciar la serie:"
9      Leer n
10     // 2. Imprimir el valor inicial
11     Escribir "Serie generada: "
12     Escribir Sin Saltar n, " "
13     // 3. Proceso (Uso de estructura Mientras)
14     // Permanecer en el bucle mientras
15     // el valor actual sea mayor que 1

```

```

16  Mientras n > 1 Hacer
17      // A. Verificar la paridad (Uso de Si-Sino)
18      Si n MOD 2 == 0 Entonces
19          // El valor es par: Dividir para 2
20          n <- n / 2
21      SiNo
22          // El valor es impar: Multiplicar por 3 y sumar 1
23          n <- (3 * n) + 1
24      Fin Si
25      // B. Imprimir el nuevo termino
26      Escribir Sin Saltar n, " "
27  Fin Mientras
28  FinAlgoritmo

```

* Ejercicio 14

Diseñar un algoritmo que presente la serie de números primos. Para esto el usuario ingresa un valor n que representa la cantidad de números a generar. Un número primo es aquel que es divisible para sí mismo y para la unidad.

Algoritmo 2.73: Pseudocódigo PSeInt: Generar N números Primos

```

1  Algoritmo Generar_Primos
2      // Declarar variables
3      Definir n_limite, contador_primos, numero_actual Como Entero
4      Definir divisor, residuo, iter Como Entero
5      Definir es_primo Como Logico
6      Escribir "---- Generar N Numeros Primos ----"
7      // 1. Lectura de datos
8      Escribir "Ingrese la cantidad (N) de numeros primos a generar:"
9      Leer n_limite
10     // 2. Inicializar contadores y el numero a evaluar
11     contador_primos <- 0
12     numero_actual <- 1 // Empezar a buscar desde 1 o 2
13     Escribir "Los primeros ", n_limite, " numeros primos son:"
14     // 3. Proceso (Uso de estructura para contar primos)
15     // Permanecer en el bucle mientras
16     // no se haya alcanzado el limite N
17     Mientras contador_primos < n_limite Hacer
18         // A. Incrementar el numero a evaluar
19         numero_actual <- numero_actual + 1
20         // B. Verificar Primalidad: Asumir que el numero es primo
21         es_primo <- Verdadero
22         // C. Los numeros menores o iguales a 1 no son primos
23         Si numero_actual <= 1 Entonces
24             es_primo <- Falso
25         SiNo
26             // D. Bucle anidado para verificar divisores
27             // (desde 2 hasta raíz de N)
28             // La verificacion solo necesita llegar hasta
29             // la raíz cuadrada (o hasta numero_actual - 1
30             // si no se usa la raíz)
31             iter = TRUNC(RAIZ(numero_actual))
32             Para divisor = 2 Hasta iter Con Paso 1 Hacer
33                 residuo <- numero_actual MOD divisor
34                 // Si el residuo es 0, el numero tiene
35                 // otro divisor (no es primo)
36                 Si residuo == 0 Entonces

```

```

37         es_primo <- Falso
38         Romper
39     Fin Si
40 Fin Para
41 Fin Si
42 // E. Si es primo, contabilizar e imprimir
43 Si es_primo Entonces
44     contador_primos <- contador_primos + 1
45     Escribir Sin Saltar numero_actual, " "
46 Fin Si
47 Fin Mientras
48 FinAlgoritmo

```

* Ejercicio 15

Elaborar un algoritmo para verificar si un conjunto de cuatro números enteros ingresados por el usuario cumple con la propiedad de ser un Conjunto Diofántico con una constante fija de 1.

Análisis: Un conjunto $\{a, b, c, d\}$ es Diofántico si, para cualquier par distinto (x, y) del conjunto, se cumple:

$$x \cdot y + 1 = k^2$$

donde k es un entero.

Algoritmo 2.74: Pseudocódigo PSeInt: Verificación de conjunto Diofántico

```

1  Algoritmo Verificar_Conjunto_Diofantico
2      // Declarar funcion para verificar
3      // si un numero es cuadrado perfecto
4      Funcion es_cuadrado_perfecto(num_entrada)
5          Definir raiz Como Real
6          Definir es_perfecto Como Logico
7          // Si el numero es negativo,
8          // no es un cuadrado perfecto real
9          Si num_entrada < 0 Entonces
10             es_perfecto <- Falso
11         SiNo
12             // 1. Calcular la raiz cuadrada
13             raiz <- RAIZ(num_entrada)
14             // 2. Verificar si la parte truncada
15             // es igual al valor real
16             // Si son iguales, la raiz es entera
17             // y es un cuadrado perfecto
18             Si TRUNC(raiz) == raiz Entonces
19                 es_perfecto <- Verdadero
20             SiNo
21                 es_perfecto <- Falso
22         Fin Si
23     Fin Si
24     Retornar es_perfecto
25 Fin Funcion
26 // -----
27 // Declarar variables principales
28 Definir a, b, c, d Como Entero
29 Definir producto_1 Como Entero
30 Definir es_diofantico Como Logico
31 Escribir "--- Conjunto Diofantico (Constante +1) ---"
32 // 1. Lectura de datos: Ingresar los cuatro numeros
33 Escribir "Ingrese el primer numero (a):"

```

```

34 Leer a
35 Escribir "Ingrese el segundo numero (b):"
36 Leer b
37 Escribir "Ingrese el tercer numero (c):"
38 Leer c
39 Escribir "Ingrese el cuarto numero (d):"
40 Leer d
41 // 2. Proceso: Inicializar la bandera de verificacion
42 es_diofantico <- Verdadero
43 // 3. Evaluar las 6 combinaciones de pares (Uso del operador Y)
44 // Par 1: a * b + 1
45 producto_mas_uno <- a * b + 1
46 es_diofantico <- es_diofantico Y es_cuadrado_perfecto(producto_1)
47 // Par 2: a * c + 1
48 producto_mas_uno <- a * c + 1
49 es_diofantico <- es_diofantico Y es_cuadrado_perfecto(producto_1)
50 // Par 3: a * d + 1
51 producto_mas_uno <- a * d + 1
52 es_diofantico <- es_diofantico Y es_cuadrado_perfecto(producto_1)
53 // Par 4: b * c + 1
54 producto_mas_uno <- b * c + 1
55 es_diofantico <- es_diofantico Y es_cuadrado_perfecto(producto_1)
56 // Par 5: b * d + 1
57 producto_mas_uno <- b * d + 1
58 es_diofantico <- es_diofantico Y es_cuadrado_perfecto(producto_1)
59 // Par 6: c * d + 1
60 producto_mas_uno <- c * d + 1
61 es_diofantico <- es_diofantico Y es_cuadrado_perfecto(producto_1)
62 // 4. Salida de resultados
63 Escribir "--- Resultado de la Verificacion ---"
64 Si es_diofantico Entonces
65     Escribir "El conjunto {", a, ", ", b, ", ", c, ", ", d, "}"
66     Escribir "SI es un Conjunto Diofantico."
67 SiNo
68     Escribir "El conjunto {", a, ", ", b, ", ", c, ", ", d, "}"
69     Escribir "NO es un Conjunto Diofantico."
70 Fin Si
71 FinAlgoritmo

```

* Ejercicio 16

Diseñar un algoritmo que le permita al usuario ingresar un valor entero N, el cual representa la cantidad de términos que desea generar de la sucesión de números cuadrados perfectos (1, 4, 9, 16, 25, ...) por medio de la suma acumulativa de los números impares consecutivos.

Algoritmo 2.75: Pseudocódigo PSeInt: Serie de sumas de impares (Números cuadrados)

```

1 Algoritmo Generar_Cuadrados_Impares
2     // Declarar variables de proceso
3     Definir n_limite, contador Como Entero
4     Definir impar_actual, suma_acumulada Como Entero
5     Escribir "--- Generar Numeros Cuadrados Perfectos ---"
6     // 1. Lectura de datos
7     Escribir "Ingrese la cantidad (N) de numeros cuadrados a generar:"
8     Leer n_limite
9     // 2. Inicializar acumuladores y generadores
10    suma_acumulada <- 0 // Acumular la suma de impares
11    impar_actual <- 1 // El primer impar a sumar

```

```

12  Escribir "Resultados de la suma:"
13  // 3. Proceso (Uso de estructura Para para generar N terminos)
14  // Recorrer el bucle N veces
15  Para contador = 1 Hasta n_limite Con Paso 1 Hacer
16      // A. Sumar el impar actual al acumulador
17      // (genera el numero cuadrado)
18      suma_acumulada <- suma_acumulada + impar_actual
19      // B. Salida de resultado
20      Escribir "Suma de los primeros ", contador Sin Saltar
21  Escribir " impares = ", suma_acumulada, " (", contador, "^2)"
22      // C. Actualizar el impar para la siguiente iteracion
23      // (incremento de 2)
24      impar_actual <- impar_actual + 2
25  Fin Para
26  FinAlgoritmo

```

* Ejercicio 17

Elaborar un algoritmo que solicite al usuario dos valores enteros, uno que será la base y otro que será el exponente. Luego genere la potencia del número.

Algoritmo 2.76: Pseudocódigo PSeInt: Cálculo de Potencia

```

1  Algoritmo Calcular_Potencia
2      // Declarar variables de proceso
3      Definir base, exponente, contador Como Entero
4      Definir resultado Como Real
5      // Usar Real para manejar posibles resultados grandes
6      Escribir "--- Calculo de la Potencia (Base ^ Exponente) ---"
7      // 1. Lectura de datos
8      Escribir "Ingrese la base (valor entero):"
9      Leer base
10     Escribir "Ingrese el exponente (valor entero):"
11     Leer exponente
12     // 2. Inicializar el resultado a 1
13     // La multiplicacion debe comenzar en 1
14     resultado <- 1
15     // 3. Proceso (Simulacion de la potencia mediante un bucle Para)
16     // Multiplicar la 'base' por si misma tantas veces
17     // como indique el 'exponente'
18     // Manejar caso especial: Exponente negativo
19     Si exponente < 0 Entonces
20         // Para manejar potencias negativas
21         // Usamos el valor absoluto del exponente
22         Para contador = 1 Hasta ABS(exponente) Con Paso 1 Hacer
23             resultado <- resultado * base
24         Fin Para
25         // Invertir el resultado
26         resultado <- 1 / resultado
27     // Manejar caso especial: Exponente cero
28     Sino Si exponente == 0 Entonces
29         // Cualquier numero elevado a 0 es 1 (si base no es 0)
30         Si base == 0 Entonces
31             Escribir "ERROR: Indeterminacion (0^0)."
32             resultado <- 0 // 0 asignar valor de error
33         SiNo
34             resultado <- 1
35     Fin Si

```

```

36 // Manejar caso estandar: Exponente positivo
37 Sino
38     Para contador = 1 Hasta exponente Con Paso 1 Hacer
39         resultado <- resultado * base
40     Fin Para
41 Fin Si
42 // 4. Salida de resultado
43 Si (exponente <> 0) O (base <> 0) Entonces
44     Escribir "El resultado de ", base, " elevado a ",
45     Escribir "exponente, " es: ", resultado
46 Fin Si
47 Escribir "--- Proceso finalizado ---"
48 FinAlgoritmo

```

* Ejercicio 18

Elaborar un algoritmo que solicite al usuario un valor entero. Luego genere el factorial del número.

Algoritmo 2.77: Pseudocódigo PSeInt: Cálculo del Factorial (N!)

```

1 Algoritmo Calcular_Factorial
2     // Declarar variables de proceso
3     Definir n, contador Como Entero
4     Definir factorial Como Real
5     // Usar Real para manejar factoriales grandes
6     Escribir "--- Factorial (N!) ---"
7     // 1. Lectura de datos
8     Escribir "Ingrese un numero entero positivo (N):"
9     Leer n
10    // 2. Inicializar el factorial en 1
11    // El factorial de 0 es 1, y la multiplicacion debe comenzar en 1
12    factorial <- 1
13    // 3. Proceso y Verificacion (Usando Si y Para)
14    Si n < 0 Entonces
15        Escribir "ERROR: El factorial solo esta definido"
16        Escribir "para numeros positivos o cero."
17    SiNo Si n == 0 Entonces
18        // Caso base: 0! = 1
19        factorial <- 1
20    SiNo
21        // Recorrer el bucle desde 1 hasta N para multiplicar
22        Para contador = 1 Hasta n Con Paso 1 Hacer
23            // Multiplicar el acumulador por el valor del contador
24            factorial <- factorial * contador
25        Fin Para
26    Fin Si
27    // 4. Salida de resultado
28    Si n >= 0 Entonces
29        Escribir "El factorial de ", n, " (", n, "!) es: ", factorial
30    Fin Si
31    Escribir "--- Proceso finalizado ---"
32 FinAlgoritmo

```



Al-Khwārizmī, Ada Lovelace y Alan Turing

¡Buen trabajo!

Finalización del capítulo 2