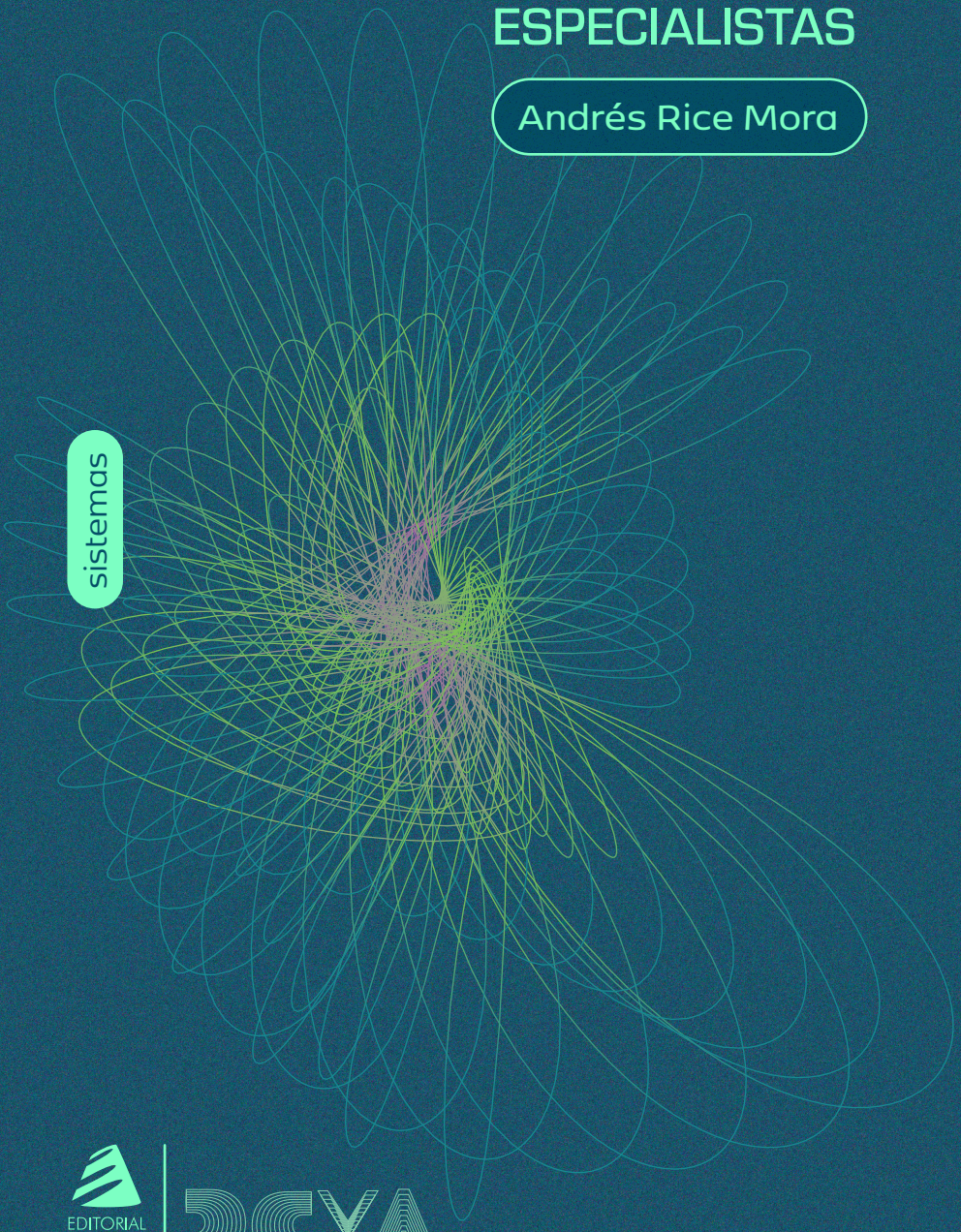
FUNDAMENTOS DE PROGRAMACIÓN PARA NO ESPECIALISTAS

Andrés Rice Mora

sistemas



EDITORIAL USACH

DCYA

FUNDAMENTOS DE PROGRAMACIÓN PARA NO ESPECIALISTAS

Fundamentos de programación para no especialistas
Andrés Rice Mora

Editorial Universidad de Santiago de Chile, 2025
Av. Víctor Jara 3453, Estación Central, Santiago de Chile
Tel.: +56 2 2718 0080
www.editorialusach.cl

ISBN edición digital: 978-956-303-798-2

Director editorial: Jorge Montealegre I.
Edición: Luz María Astudillo U.
Diagramación: Andrea Meza V.
Diseño de portada: Ana Ramírez P.
Revisor académico: Diego Fuentealba

Primera edición, agosto 2025

Impreso en Chile

ANDRÉS RICE MORA

FUNDAMENTOS DE PROGRAMACIÓN PARA NO ESPECIALISTAS

EDITORIAL
USACH

Índice

Prefacio

El Dr. Juan Abello Romero, director del Departamento de Contabilidad y Auditoría de la Universidad de Santiago de Chile, reafirma el compromiso de la institución con una educación de excelencia al encomendar a su equipo docente la creación de material pedagógico que prepare a las y los estudiantes para enfrentar con confianza y solvencia los desafíos del entorno profesional. En un mundo donde la tecnología —y en particular la programación— desempeña un papel esencial en la contabilidad, la auditoría y la gestión de datos, el dominio de estas herramientas es clave para transformar la manera en que procesamos y analizamos la información.

Este material ha sido concebido como un recurso complementario para el aprendizaje de programación, proporcionando un enfoque estructurado y práctico. A través de explicaciones detalladas y ejemplos aplicados, facilita la comprensión de conceptos clave, además de incluir ejercicios basados en situaciones reales que permiten consolidar el conocimiento adquirido. Su propósito principal es vincular la teoría con la práctica, fortaleciendo el desarrollo de habilidades esenciales para el ejercicio profesional.

Agradezco a la Universidad de Santiago de Chile por su respaldo en la realización de este libro, así como al Dr. Juan Abello Romero por su confianza y apoyo. También quiero expresar mi gratitud a mis colegas y estudiantes, cuyas inquietudes y desafíos en el aprendizaje de la programación han sido la inspiración para desarrollar este material.

Esta colaboración refleja un compromiso compartido con la excelencia académica y la convicción de que es posible ofrecer a los estudiantes herramientas que fortalezcan su formación. Estoy seguro de que

este libro enriquecerá el aprendizaje de quienes lo utilicen, fomentará una actitud proactiva hacia la integración de la tecnología y motivará a las y los estudiantes a convertirse en profesionales competentes, versátiles y preparados para prosperar en un entorno dinámico y en constante evolución tecnológica.

Mg. Andrés A. Rice Mora

Público objetivo

Este libro está diseñado para estudiantes de contabilidad y auditoría, así como para profesionales del área que desean adquirir habilidades en programación sin necesidad de experiencia previa en el campo. No se requiere conocimiento previo en lenguajes de programación, ya que el contenido está estructurado para introducir gradualmente los conceptos y su aplicación práctica.

Cómo usar este libro

Cada capítulo incluye explicaciones teóricas, ejemplos prácticos y ejercicios aplicados. Se recomienda seguir la lectura en orden, aplicando los conocimientos adquiridos en cada sección antes de avanzar al siguiente nivel.

Introducción

En un mundo cada vez más digitalizado y dinámico, la contabilidad y la auditoría evolucionan de la mano de los avances tecnológicos. Lo que antes era exclusivo de científicos e ingenieros en computación, hoy se convierte en una habilidad esencial para los profesionales financieros. La programación, lejos de ser una competencia opcional, se ha convertido en un recurso clave para optimizar procesos, mejorar la precisión de los análisis y fortalecer la toma de decisiones en el ámbito contable y de auditoría.

Fundamentos de programación para no especialistas nace con el propósito de dotar a los contadores auditores de las competencias necesarias para enfrentar los desafíos del futuro. La capacidad de programar no solo aumenta la eficiencia al automatizar tareas repetitivas, sino que también abre nuevas oportunidades para el análisis de datos financieros, lo que permite interpretar información con mayor profundidad y estrategia.

Si bien este libro no pretende ser un manual exhaustivo de programación, ha sido diseñado para introducir progresivamente a las y los estudiantes en este campo, comenzando con los fundamentos tecnológicos y avanzando hasta su aplicación práctica. A lo largo de estas páginas, exploraremos los conceptos esenciales de VBA (Visual Basic for Applications), el lenguaje detrás de las Macros en Excel, proporcionando una base que facilitará la comprensión de otros lenguajes como Python, hoy ampliamente utilizado en análisis de datos y automatización de procesos.

Mediante ejemplos prácticos y casos aplicados al mundo contable y de auditoría, este libro busca empoderar a los futuros profesionales para

que no solo se adapten a los cambios tecnológicos, sino que también los lideren. Dominar los fundamentos de la programación permitirá gestionar grandes volúmenes de datos, mejorar la precisión de los informes y ofrecer *insights*[1] más estratégicos y valiosos para sus organizaciones y clientes.

Más que una guía técnica, este libro es una invitación a descubrir la programación como una herramienta poderosa para potenciar el desarrollo profesional. Juntas y juntos podemos redefinir el futuro de la contabilidad y la auditoría, integrando la precisión y el rigor contable con la innovación y la eficiencia tecnológica.

El libro está dividido en cinco capítulos más un apéndice para facilitar la comprensión gradual de la programación.

El capítulo 1 introduce los conceptos básicos de la informática y la arquitectura de los computadores, explicando el sistema binario, los componentes de *hardware* y *software*, y la evolución del *software* en la sociedad contemporánea. Además, se abordan los primeros conceptos de programación y su importancia en la optimización de procesos en el ámbito contable y financiero.

En el capítulo 2 se profundiza en la abstracción, la lógica y los algoritmos, elementos esenciales para el pensamiento computacional. Se presentan estrategias para modelar problemas y diseñar soluciones utilizando diagramas de flujo, con ejemplos prácticos que facilitan la comprensión de la estructuración lógica aplicada a la programación.

El capítulo 3 introduce al lector en el uso del lenguaje VBA, utilizado en Excel para la automatización de tareas. Se detallan aspectos clave como la sintaxis, las estructuras de control, el manejo de variables y operadores, así como el entorno de desarrollo de VBA dentro de Excel. A través de ejemplos y ejercicios, los estudiantes aprenderán a escribir y ejecutar código en este lenguaje.

En el capítulo 4 se exploran las Macros en Excel y su aplicación en la automatización de procesos repetitivos. Se explica cómo grabar, modificar y ejecutar Macros para optimizar tareas dentro de las hojas de

1 El término *insights* proviene del inglés y se traduce como "percepciones", "conocimientos profundos" o "hallazgos clave". En el contexto del análisis de datos y la toma de decisiones, un *insight* se refiere a una conclusión valiosa obtenida a partir del análisis de información, que permite entender mejor una situación y tomar decisiones estratégicas.

cálculo. Además, se diferencian las subrutinas y las funciones en VBA, destacando su utilidad en la programación orientada a mejorar la gestión de datos contables.

El capítulo 5 profundiza en la manipulación avanzada de datos mediante VBA, abordando la importación y exportación de archivos en formatos CSV y XML. También se presentan técnicas para conectar Excel con bases de datos, para permitir gestionar grandes volúmenes de información de manera eficiente y automatizada.

Finalmente, los apéndices incluyen una serie de desafíos prácticos diseñados para reforzar los conocimientos adquiridos. Estos ejercicios abordan diagramas de flujo, subprocesos en VBA y el desarrollo de funciones propias dentro del lenguaje. Además, se proporciona un glosario con términos clave y una tabla de ilustraciones que facilita la consulta de los gráficos utilizados a lo largo del libro.

¡Bienvenidas y bienvenidos a este emocionante viaje donde uniremos la contabilidad y la tecnología para optimizar el trabajo y alcanzar un nuevo nivel de excelencia profesional!

Reconocimientos y recursos para profundizar en VBA

En el desarrollo de este libro, conté con el apoyo de diversas herramientas tecnológicas, entre ellas ChatGPT de OpenAI, una inteligencia artificial que facilitó la generación de ideas, la creación de imágenes y el desarrollo de contenido, para contribuir significativamente a la claridad de los conceptos y la elaboración de ejercicios sobre los temas tratados.

Para los(as) estudiantes interesados(as) en profundizar en VBA (Visual Basic for Applications), las siguientes fuentes ofrecen una cobertura completa y oficial de este lenguaje, especialmente para su uso en aplicaciones como Excel, Access y Word en el entorno de Microsoft Office:

- Microsoft Learn: documentación de VBA – proporciona una introducción a VBA, la referencia de la biblioteca de objetos y conceptos esenciales.
- Microsoft Developer Network (MSDN): documentación avanzada de VBA – ofrece información técnica detallada, ideal para desarrolladores que buscan profundizar en objetos, propiedades y métodos específicos.

Estas fuentes resultan invaluables para quienes deseen explorar VBA y aprovechar sus capacidades al máximo.

Capítulo 1
Introducción a las tecnologías

En la actualidad, las computadoras se han convertido en herramientas indispensables para casi todas las áreas del conocimiento y de la industria. Sin embargo, para entender cómo funcionan estas máquinas es necesario familiarizarse con los conceptos fundamentales de su arquitectura y los principios que rigen su operación. Este capítulo está diseñado para presentar los elementos básicos que componen a un computador y explicar cómo interactúan estos componentes para procesar y gestionar los datos.

Objetivos de aprendizaje

1. Identificar los componentes básicos de un computador (ALU, unidad de control, memoria, dispositivos de E/S, buses).
2. Describir el sistema binario y cómo los *bits* y *bytes* representan la información en los sistemas digitales.
3. Diferenciar entre *hardware* y *software*, y explicar el impacto del *software* en la sociedad moderna.
4. Entender qué es la programación y cómo los lenguajes de programación permiten crear soluciones.

Resultados de aprendizaje

Al finalizar este capítulo, los estudiantes identificarán correctamente los componentes fundamentales de un computador, como la ALU, la unidad de control, la memoria, los dispositivos de E/S y los buses, así como la arquitectura de Von Neumann. Aplicarán el conocimiento del sistema binario para realizar conversiones de magnitudes numéricas simples. Serán capaces de analizar la estructura de distintos tipos de computadores, como los PC, supercomputadoras y *mainframes*, evaluando su idoneidad para diferentes aplicaciones. Además, diferenciarán entre *hardware* y *software* y explicarán cómo el *software* impacta a la sociedad contemporánea. También explicarán qué es la programación, cómo los lenguajes de programación facilitan la creación de soluciones, y serán capaces de analizar los beneficios de aprender a programar.

1.1. Fundamentos de un computador: arquitectura y funcionamiento

Un computador es un dispositivo diseñado para almacenar, procesar y gestionar datos, llevando a cabo tanto operaciones lógicas como matemáticas, y facilitando la interacción con el usuario. Su diseño sigue la arquitectura de Von Neumann, que incluye los siguientes componentes clave:

- Unidad aritmética y lógica (ALU): se encarga de realizar operaciones matemáticas y lógicas.
- Unidad de control: gestiona y coordina las operaciones del computador, controla permanentemente la ejecución de instrucciones.
- Memoria: almacena temporal o permanentemente datos e instrucciones.
- Dispositivos de entrada/salida (E/S): permiten la comunicación entre el computador y el usuario, así como con otros sistemas.

- Buses: son los canales de comunicación que transportan datos entre los diferentes componentes del computador.

La arquitectura de Von Neumann es un modelo conceptual para la estructura de una computadora digital, propuesto por el matemático y físico John Von Neumann en 1945. Este diseño ha sido fundamental para el desarrollo de las computadoras y sigue siendo un principio básico en la mayoría de los sistemas modernos. Aunque presenta algunas limitaciones, sus ventajas en términos de simplicidad y flexibilidad han facilitado una amplia variedad de desarrollos tanto en *hardware* como en *software*.

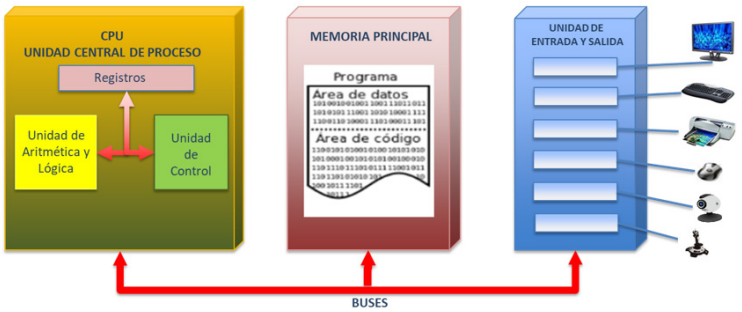


Figura 1. Arquitectura de Von Neumann.

1.2. Tipificación de computadores

Existen varios tipos de computadores, cada uno diseñado para satisfacer diferentes necesidades y usos. Algunos de los tipos más comunes son:

- Supercomputadoras: son los computadores más potentes y rápidos, capaces de realizar millones de cálculos por segundo. Se utilizan en investigaciones científicas complejas, simulaciones climáticas, estudios del espacio, y procesamiento de datos a gran escala.
- Computadoras de *mainframe* (macrocomputadoras): son grandes y potentes, capaces de manejar y procesar miles de millones de transacciones en tiempo real. Se usan principalmente

en grandes empresas, bancos, y agencias gubernamentales para gestionar bases de datos, transacciones y aplicaciones críticas.

- Minicomputadoras (o servidores de rango medio): son más pequeñas y menos potentes que las *mainframes*, pero, aun así, son capaces de manejar múltiples usuarios simultáneamente. Se utilizan en organizaciones medianas para aplicaciones específicas como bases de datos, correo electrónico, y servidores de aplicaciones.
- Computadoras personales (PC): computadoras diseñadas para el uso individual. Hay de tipos escritorio (*desktop*), portátiles (*laptops/notebooks*), tabletas y otros.
- Tabletas: dispositivos portátiles con pantallas táctiles que permiten realizar tareas similares a las de una computadora, aunque con menos capacidad de procesamiento que los PC tradicionales. El uso ideal es para la navegación web, lectura, multimedia y aplicaciones ligeras.
- Computadoras híbridas (2 en 1): son dispositivos que pueden funcionar tanto como tabletas como *laptops*. Suelen tener una pantalla táctil y un teclado desmontable o plegable. Ofrecen la portabilidad de una tableta con la funcionalidad de una computadora portátil, adecuadas para quienes necesitan ambas funciones en un solo dispositivo.
- Estaciones de trabajo (*workstation*): computadoras de alto rendimiento diseñadas para aplicaciones técnicas y científicas. Son más potentes que los PC estándar, utilizadas por profesionales que requieren gran capacidad de procesamiento, gráficos avanzados, y alto rendimiento, como ingenieros, diseñadores gráficos y científicos.
- Computadoras de bolsillo (*handhelds*): dispositivos muy portátiles que caben en la mano, como los teléfonos inteligentes y los asistentes personales digitales (los PDA), utilizados para tareas diarias como la gestión de correos electrónicos, navegación web y el uso de aplicaciones.

1.3. Cómo funciona la computación digital

La computación digital funciona mediante el procesamiento de datos en forma de *bits*, que son la unidad más básica de información en informática. Un *bit* puede tener un valor de 0 o 1 y, mediante combinaciones de estos *bits*, las computadoras digitales pueden representar y manipular información de manera extremadamente rápida y precisa. Aquí explico con más detalle cómo funciona la computación digital.

1.3.1. El *bit*, la unidad elemental

El *sistema binario* es un sistema numérico que utiliza únicamente dos dígitos: 0 y 1. Cada uno de estos dígitos se llama *bit*, una abreviatura de "binary digit" (dígito binario). Un *bit* es la unidad más básica de información en la electrónica digital y representa dos posibles estados: la presencia (1) o la ausencia (0) de voltaje.

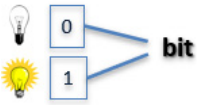


Figura 2. Esquema de un bit.

A diferencia del sistema decimal que utiliza diez dígitos, del 0 al 9, el sistema binario basado en la base 2 (recuerde la operación matemática potencia y sus partes, base y exponente). Esto significa que cada posición en un número binario corresponde a una potencia de base 2 como se muestra a continuación:

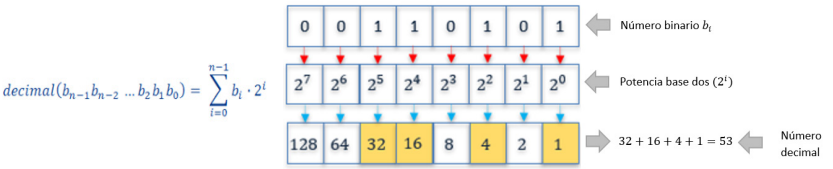


Figura 3. Esquema del Sistema Binario.

1.3.2. El *byte*, unidad elemental de almacenamiento de datos

Un *byte* es la unidad básica de almacenamiento de información en informática y telecomunicaciones. Consiste en 8 *bits*, donde cada *bit* es la unidad más pequeña de datos que puede tener un valor de 0 o 1. Gracias a su estructura de 8 *bits*, un *byte* puede representar hasta 255 combinaciones diferentes de ceros y unos (desde 00000000 hasta 11111111 en binario). Estas combinaciones permiten codificar una amplia variedad de caracteres, símbolos y dígitos. El *byte* se utiliza como una medida estándar para cuantificar la cantidad de datos y es fundamental para el procesamiento y almacenamiento de información digital en computadoras y otros dispositivos electrónicos.

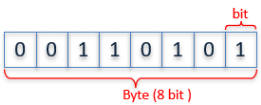


Figura 4. Esquema de un *byte*.

1.3.3. Estándar de codificación de caracteres, símbolos y dígitos

Mediante combinaciones de 0 y 1 en grupos de 8 bits (*byte*), se puede codificar toda la información que los dispositivos electrónicos necesitan para procesar, almacenar y comunicarse electrónicamente, lo que hace posible el procesamiento digital de datos.

Símbolo	Dec	Binario
a	97	01100001
á	225	11100001
à	224	11100000
A	65	01000001
À	192	11000000
Á	193	11000001
Espacio	69	00100000
@	64	01000000
_	95	01011111
-	45	00101101

Figura 5. Extracto de la codificación binaria.

El estándar de codificación de caracteres más utilizado es *Unicode*, especialmente en su formato UTF-8, debido a su eficiencia en el almacenamiento y la transmisión de datos. UTF-8 puede representar prácticamente todos los caracteres de los sistemas de escritura del mundo, incluyendo símbolos técnicos, pictográficos y muchos otros. Esto lo convierte en una herramienta esencial para la comunicación global y la interoperabilidad entre diferentes sistemas informáticos.

Además de UTF-8, Unicode también se puede codificar en otros formatos como UTF-16 y UTF-32, que utilizan 2 y 4 *bytes* por carácter, respectivamente. Sin embargo, estos formatos son menos eficientes en términos de almacenamiento que UTF-8.

Comprender el uso de los estándares de codificación es crucial porque, al desarrollar un programa, algunas conductas inesperadas pueden surgir debido a diferencias entre caracteres como "a" y "A". Aunque ambas representan la misma letra, una está en minúscula y la otra en mayúscula, y su representación binaria es diferente. Lo mismo ocurre con el "espacio", que también tiene su propia representación binaria.

Estandarizar la codificación de caracteres garantiza la compatibilidad en la transferencia de datos, programas e información entre computadoras, independientemente del fabricante, el país de uso u otros factores.

1.3.4. Magnitudes en el sistema binario

En el sistema binario, las magnitudes se refieren a la forma en que se agrupan diferentes cantidades y tamaños de datos. A continuación, se explica cómo funcionan estas magnitudes en el contexto del sistema binario:

Denominación	Símbolo	Equivalencia	Equivalencia en Byte
Byte	B	8 bites	1 Byte
Kilobyte	KB	1024 bytes	1.024 Byte
Megabyte	MB	1024 Kilobyte	1.048.576 Byte
Gigabyte	GB	1024 Megabyte	1.073.741.824 Byte
Terabyte	TB	1024 Gigabyte	1.099.511.627.776 Byte
Petabyte	PB	1024 Terabyte	1.125.899.906.842.620 Byte
Exabyte	EB	1024 Petabyte	1.152.921.504.606.850.000 Byte
Zettabyte	ZB	1024 Exabyte	1.180.591.620.717.410.000.000 Byte
Yottabyte	YB	1024 Zettabyte	1.208.925.819.614.630.000.000.000 Byte
Brontobyte	BB	1024 Yottabyte	1.237.940.039.285.380.000.000.000.000 Byte
Geopbyte	GeB	1024 Brontobyte	1.267.650.600.228.230.000.000.000.000 Byte
Saganbyte	SB	1024 Geopbyte	1.298.074.214.633.710.000.000.000.000.000 Byte
Jotabyte	JB	1024 Saganbyte	1.329.227.995.784.920.000.000.000.000.000.000 Byte

Figura 6. Magnitudes en el sistema binario.

Ahora, haremos unos ejercicios que nos permitirá practicar la conversión entre diferentes magnitudes y analizar la relación costo-beneficio de cada opción.

Desafío (Blitz vs Schenll)
La compañía Blitz ofrece un servicio de internet con una velocidad de transmisión de 46 *megabits* por segundo (Mbps) por $28.000 al mes, mientras que la compañía Schnell ofrece un servicio de internet con una velocidad de transmisión de 5,75 *megabytes* por segundo (MBps) por $26.850 al mes.
Pregunta:
¿Cuál de estas dos opciones de internet ofrece una mejor relación entre la velocidad de transmisión y el costo mensual?

Desarrollo (Blitz vs Schenll)
Lo primero que observamos es que las unidades de magnitud son distintas. Entonces, para desarrollar este ejercicio primero, es necesario convertir las velocidades de transmisión a una misma unidad para poder compararlas directamente.
Sabemos que:
✓ Blitz ofrece 46 *megabits* por segundo (Mbps).
✓ Schnell ofrece 5.75 *megabytes* por segundo (MBps).
Paso 1: homologar las unidades
Para poder comparar las velocidades, primero debemos recordar que: 1 Byte = 8 bits. Ahora, vamos a convertir la velocidad de Schnell de MBps a Mbps:

$$\text{Velocidad de Schnell} = 5.75\,\text{MBps} \times 8 = 46\,\text{Mbps}$$

Por comparación visual, nos damos cuenta que ambas compañías ofrecen una velocidad de transmisión de 46 Mbps.
Paso 2: comparar los costos
Los precios de las compañías son los siguientes:
✓ Blitz cobra $28.000 al mes.
✓ Schnell cobra $26.850 al mes.
Paso 3: conclusión
Como ambas compañías ofrecen la misma velocidad, la que tiene el menor costo mensual es la mejor opción. Por lo tanto, Schnell, con un costo de $26.850 al mes, ofrece una mejor relación entre velocidad de transmisión y costo mensual.

Desafío (Fastenet vs Speedline)
La compañía Fastnet ofrece un plan de internet con una velocidad de 120 *megabits* por segundo (Mbps) por $50.000 al mes, mientras que la compañía SPEEDLINE ofrece un plan con una velocidad de 10 *megabytes* por segundo (MBps) por $52.000 al mes.
Pregunta: ¿cuál de los dos planes de internet proporciona más velocidad por peso? ¿Cuál es la mejor opción en términos de costo y velocidad?

Desarrollo (Fastnet vs Speedline)
1. Convertir las velocidades de internet a la misma unidad de medida:
Recordemos que 1 *megabyte* (MB) es igual a 8 *megabits* (Mb). Por lo tanto, para convertir la velocidad de la compañía Speedline de *megabytes* por segundo (MBps) a *megabits* por segundo (Mbps), multiplicamos por 8:

$$10\ MBps \times 8 = 80\ Mbps$$

Así, la velocidad del plan de Speedline es de 80 Mbps.
2. Calcular el costo por *megabit* por segundo (Mbps) para cada plan:

- Para Fastnet:

$$Costo\ por\ Mbps = \frac{Precio\ mensual}{Velocidad\ en\ Mbps} = \frac{50.000}{120} \approx 416{,}67\ pesos\ por\ Mbps$$

- Para Speedline:

$$Costo\ por\ Mbps = \frac{52.000}{80} = 650\ pesos\ por\ Mbps$$

3. Comparar los costos por Mbps:
- Fastnet: 416,67 pesos por Mbps
- Speedline: 650 pesos por Mbps

4. Determinar la mejor opción:
Fastnet ofrece un menor costo por Mbps (416,67 pesos) en comparación con Speedline (650 pesos), lo que significa que obtienes más velocidad por peso con Fastnet. En conclusión, la mejor opción en términos de costo y velocidad es el plan de internet de Fastnet, ya que proporciona más velocidad (Mbps) por cada peso pagado en comparación con el plan de Speedline.

1.4. La informática

La informática es la ciencia que se encarga del procesamiento automático de información a través de sistemas computacionales. Su objetivo principal es desarrollar herramientas y tecnologías que permitan gestionar, almacenar, transmitir y procesar datos de forma eficiente y segura. Esto abarca tanto la creación de *hardware* como el desarrollo de *software* y la implementación de algoritmos para resolver problemas complejos, optimizando así el manejo de la información.

En auditoría, las computadoras se usan para manejar grandes volúmenes de datos, detectar irregularidades y automatizar tareas repetitivas. La informática no solo facilita estos procesos, sino que también ayuda a implementar sistemas de seguridad y control para proteger la información, lo que es clave en el trabajo de un auditor.

Figura 7. Sistema informático.

1.4.1. El *hardware*

Se refiere a todos los componentes físicos de un sistema informático. Estos son los elementos tangibles que constituyen un computador o dispositivo electrónico y que se pueden ver y tocar. El *hardware* incluye una variedad de componentes que trabajan juntos para realizar tareas específicas. Los componentes de *hardware* más comunes incluyen:

- Unidad Central de Procesamiento (CPU): conocida como el "cerebro" del computador, es responsable de ejecutar instrucciones y procesar datos.
- Memoria (RAM): almacena datos y programas temporalmente mientras el computador está en uso.

- Disco Duro/Unidad de Estado Sólido (HDD/SSD): almacenan datos de forma permanente, incluyendo el sistema operativo, aplicaciones y archivos personales.
- Placa base: es la tarjeta principal que conecta todos los componentes del computador.
- Periféricos: incluyen dispositivos de entrada (como teclado y ratón) y salida (como monitor e impresora).

En resumen, el *hardware* abarca todos los componentes físicos que hacen que un sistema informático funcione.

1.4.2. El *software*

En la vida cotidiana, la palabra *software* se utiliza a menudo como sinónimo de "programa". Sin embargo, más adelante veremos por qué esta analogía no es del todo precisa.

¿Qué es un programa informático? Un programa es un conjunto de instrucciones escritas en un lenguaje de programación que una computadora puede interpretar y ejecutar. Estas instrucciones indican a la computadora qué tareas y cálculos específicos debe realizar, cómo procesar datos, comunicarse con otros dispositivos o llevar a cabo cualquier otra operación necesaria. En términos sencillos, un programa es un conjunto de órdenes que se le dan a una computadora para que las ejecute.

¿Qué es un *software*? Para los profesionales, un *software* se define como el conjunto integral de programas, procesos, reglas y documentación que permite la operación eficiente de un sistema de información. Esta definición no solo abarca el código que constituye los programas ejecutables, sino también los procedimientos necesarios para su correcto funcionamiento, las normativas que guían su desarrollo y uso, y la documentación que respalda tanto la creación como el mantenimiento del *software*.

Figura 8. Definición de un *software*.

El *software* es esencial para la implementación de soluciones informáticas, ya que no solo proporciona las instrucciones que una computadora debe seguir, sino que también asegura que estas instrucciones sean coherentes con los objetivos del usuario y adaptables a diferentes entornos y/o reglas del negocio. La documentación asociada es crucial, ya que ofrece orientación a los desarrolladores y usuarios sobre cómo instalar, utilizar y solucionar problemas en el *software*, garantizando así una operación continua y sin interrupciones. El *software* es más que un simple código, es una combinación de elementos que, en conjunto, facilitan la funcionalidad y la eficiencia de los sistemas de información.

1.4.3. La importancia del *software* en la sociedad contemporánea

La importancia del *software* en la actualidad es inmensa y se extiende a casi todos los aspectos de la vida diaria, el trabajo, la industria y otros. Aquí hay una pequeña muestra de dónde podemos encontrar *software*:

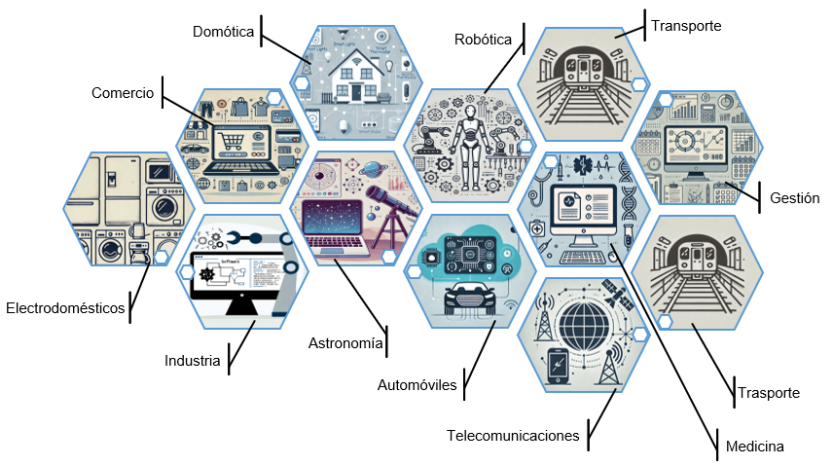


Figura 9. Ámbito del *software* en la sociedad contemporánea.
Imágenes creadas con ChatGPT de OpenAI.

Le preguntamos a la IA qué pasaría si hubiera un "¿apagón de *software*?" y respondió:

Si ocurriera un apagón del *software* en el que todos los programas dejaran de funcionar repentinamente en todo el mundo, las consecuencias serían enormes y afectarían prácticamente todos los aspectos de la vida moderna. Un apagón de este tipo provocaría una paralización masiva de la infraestructura global, la economía, la seguridad y la vida cotidiana. Dado que el *software* es fundamental para casi todos los aspectos de la sociedad actual, su pérdida tendría efectos profundos y duraderos, desencadenando una crisis que exigiría una reconstrucción significativa y una reconfiguración de cómo operamos y dependemos de la tecnología.

Ya que mencionamos a la inteligencia artificial, hagámonos la siguiente pregunta: ¿existe alguna relación entre inteligencia artificial (IA) y *software*? La respuesta es afirmativa, están intrínsecamente relacionados, ya que la IA depende del *software* para funcionar y el *software*, a su vez, se ve potenciado por las capacidades que ofrece la inteligencia artificial. La inteligencia artificial (IA) es una forma de *software*, pero es un tipo de *software* especializado diseñado para imitar algunas capacidades cognitivas humanas, como el aprendizaje, la percepción, el razonamiento, y la toma de decisiones. A continuación, se explica la relación entre ambos.

1.4.4. Inteligencia artificial: la nueva frontera del *software*

La inteligencia artificial (IA) es, en esencia, un tipo de *software* diseñado para realizar tareas que normalmente requieren inteligencia humana, como el reconocimiento de patrones, el aprendizaje, la toma de decisiones y la resolución de problemas. Este *software* utiliza algoritmos avanzados para procesar datos, aprender de la experiencia y adaptarse a nuevos datos sin necesidad de ser reprogramado explícitamente. La IA incluye cualquier técnica que permita a las máquinas imitar o replicar la inteligencia humana.

Machine Learning (aprendizaje automático) es un enfoque específico dentro de la IA que se centra en desarrollar sistemas que pueden aprender y mejorar automáticamente a partir de la experiencia, sin ser programados explícitamente para cada tarea.

El Procesamiento del Lenguaje Natural (NLP, por sus siglas en inglés) es otra subárea de la IA que se enfoca en la interacción entre las computadoras y el lenguaje humano. Se utiliza en asistentes virtuales, *chatbots* y sistemas de traducción automática para permitir que las máquinas comprendan y respondan al lenguaje humano de manera efectiva.

Visión por computadora (Computer Vision) es otro ejemplo de subárea de la IA que permite a las máquinas interpretar y comprender el mundo visual de manera similar a los humanos. Esta tecnología se utiliza en aplicaciones como el reconocimiento facial, la conducción autónoma y el análisis de imágenes médicas.

1.4.5. La inteligencia artificial integrada en *software* común

Muchos programas y aplicaciones de los *softwares* tradicionales están integrando o integran capacidades de IA para mejorar su funcionalidad y ofrecer características avanzadas. Por ejemplo: herramientas como procesadores de texto, hojas de cálculo y plataformas de correo electrónico utilizan IA para funciones como la corrección automática, sugerencias de texto predictivo, y clasificación de correos electrónicos en categorías (importante, *spam*, etc.). También podemos encontrar este tipo de amalgama en herramientas de análisis de datos y visualización, como Tableau o Power BI, que utilizan algoritmos de IA para descubrir patrones, tendencias y relaciones en grandes conjuntos de datos.

La inteligencia artificial (IA) ha revolucionado diversos campos profesionales, incluida la contabilidad, al facilitar la automatización de tareas repetitivas como la conciliación de cuentas y el análisis financiero para predecir ingresos, gastos y flujos de efectivo futuros. Con la continua evolución de esta tecnología, se prevé que la IA desempeñe un papel cada vez más crucial en la transformación de todas las actividades, permitiendo a los profesionales concentrarse en actividades estratégicas y de mayor valor agregado.

La relación entre la IA y el *software* es tanto sinérgica como transformadora. La inteligencia artificial impulsa avances significativos en el desarrollo de *software*, lo que mejora las capacidades de las aplicaciones generando nuevas oportunidades para abordar problemas complejos. A medida que la tecnología sigue avanzando, es probable que la integración de la IA en el *software* continúe ampliándose, conduciendo a soluciones más inteligentes, eficientes y adaptativas en múltiples disciplinas.

1.5. Qué es programar

Programar consiste en comunicarle a la computadora qué tareas debe realizar, utilizando un lenguaje que la máquina es capaz de entender. A través de la programación, los desarrolladores escriben instrucciones precisas que indican a la máquina cómo comportarse en diversas situaciones, desde realizar cálculos complejos hasta controlar dispositivos externos.

La programación no es una ciencia, más bien, es una habilidad técnica y una disciplina práctica que se asemeja a aprender a tocar un instrumento musical o a conducir un vehículo. Al igual que con estas habilidades, se requiere tiempo, práctica constante y dedicación para dominarla.

La práctica permite a los programadores mejorar su capacidad para resolver problemas, optimizar código y adaptarse a nuevas tecnologías, convirtiéndose en expertos a través de la experiencia y el aprendizaje continuo.

1.6. El lenguaje de programación

Un lenguaje de programación es un conjunto de reglas, sintaxis y semántica que permite a los desarrolladores escribir instrucciones que una computadora puede entender y ejecutar para realizar tareas específicas.

Las reglas en un lenguaje de programación se refieren a las normas y directrices que dictan cómo deben estructurarse y organizarse las instrucciones escritas en ese lenguaje para que una computadora las pueda entender y ejecutar correctamente. Estas reglas aseguran que el código sea consistente, comprensible y libre de errores sintácticos, lo cual es esencial para que el *software* funcione como se espera.

Figura 10. Estructura del lenguaje.

- Las reglas de sintaxis son el conjunto de normas que definen cómo deben estructurarse las instrucciones en un lenguaje de programación. Estas reglas abarcan aspectos como el uso co-

rrecto de palabras clave, símbolos, operadores y estructuras de control.
- La regla semántica describe el significado de las instrucciones y cómo deben interpretarse por la computadora. La semántica asegura que las acciones llevadas a cabo por el programa se ajusten a las expectativas del desarrollador.

Volveremos sobre los términos sintaxis y semántica una vez que abordemos el lenguaje de programación.

1.7. Por qué es importante aprender a programar

Aprender a programar es una habilidad fundamental para cualquier estudiante universitario y futuro profesional, sin importar su área de estudio. A continuación, se presentan algunas razones por las cuales deberías considerar aprender a programar:

- Demanda alta en el mercado laboral: en casi todas las industrias, desde la tecnología hasta la salud y las finanzas, hay una alta demanda de profesionales que saben programar. Aprender a programar te da una ventaja competitiva y te abre oportunidades en una variedad de campos, incluyendo trabajos bien remunerados y con estabilidad laboral.
- Habilidad para resolver problemas: la programación enseña habilidades de resolución de problemas que son útiles en cualquier disciplina. Te ayuda a pensar de manera lógica y estructurada, descomponiendo problemas complejos en partes más pequeñas y manejables. Esta habilidad de pensamiento crítico es altamente valorada en el entorno académico y profesional.
- Automatización y eficiencia: la programación te permite automatizar tareas repetitivas y tediosas, ahorrando tiempo y permitiéndote enfocarte en tareas más importantes. Por ejemplo, puedes analizar datos rápidamente o automatizar procesos, lo que aumenta tu productividad y eficiencia.
- Fomenta la creatividad: aprender a programar te da la capacidad de crear tus propias aplicaciones, herramientas, o proyectos

de investigación. Puedes desarrollar *software* personalizado que resuelva problemas específicos o que aporte innovaciones en tu campo de estudio o trabajo futuro. La programación es una forma poderosa de aplicar tu creatividad y ver resultados tangibles.

- Entender la tecnología que usas: la tecnología es una parte integral de la vida moderna y la educación universitaria. Entender los conceptos básicos de programación te ayuda a comprender mejor cómo funcionan las herramientas digitales que usas a diario, desde *software* de análisis de datos hasta aplicaciones móviles y plataformas de aprendizaje en línea.
- Desarrollo personal y profesional: aprender a programar es una inversión en tu desarrollo personal y profesional. No solo mejora tu currículum vitae, sino que también te prepara para futuras oportunidades de aprendizaje. Muchas universidades y programas de posgrado valoran la experiencia en programación como una habilidad complementaria importante.
- Flexibilidad y adaptabilidad: la programación es una habilidad transferible que te permite adaptarte rápidamente a nuevas tecnologías y herramientas en tu campo de estudio. A medida que la tecnología avanza, aquellos que pueden programar tienen una mayor capacidad para adaptarse y aprender nuevas técnicas y metodologías.
- Colaboración y comunidad: la programación fomenta la colaboración y te permite trabajar en proyectos con personas de diferentes disciplinas y lugares. Además, la comunidad de programadores es grande y diversa, con muchos recursos y apoyo disponibles en línea para ayudarte a aprender y crecer.

Aprender a programar no solo es una habilidad técnica valiosa, sino que también mejora tu capacidad para resolver problemas, fomenta tu creatividad, y te prepara para un mundo cada vez más digital. Como estudiante, tener habilidades de programación puede enriquecer tu experiencia educativa y te abrirá muchas puertas en el futuro digital.

Capítulo 2
Abstracción, lógica y algoritmos

En el desarrollo de *software*, la abstracción, la lógica y los algoritmos son tres conceptos esenciales. La abstracción ayuda a simplificar problemas complejos dividiéndolos en partes más manejables; la lógica asegura la coherencia y funcionalidad de las soluciones; y los algoritmos establecen los pasos ordenados para resolver problemas. Este capítulo explora la integración de estos tres conceptos en la creación de *software*, a través de ejemplos prácticos, diagramas de flujo y ejercicios.

Objetivos de aprendizaje

1. Analizar la integración de la abstracción, la lógica y los algoritmos en el proceso de desarrollo de *software*.
2. Comparar diferentes tipos de problemas para determinar cuál es el enfoque algorítmico más adecuado.
3. Desarrollar algoritmos, de distinta complejidad, que integren abstracción, lógica y los pasos necesarios para resolver problemas específicos.
4. Diseñar diagramas de flujo que representen el proceso de resolución de un problema utilizando los conceptos de abstracción, lógica y algoritmos.

Resultados de aprendizaje

Al finalizar este capítulo, los estudiantes definirán con precisión los conceptos de abstracción, lógica y algoritmos en el contexto del desarrollo de *software*. Compararán diferentes tipos de problemas para identificar los enfoques algorítmicos más adecuados según el contexto, y diseñarán diagramas de flujo que representen visualmente el proceso de resolución de problemas. Este capítulo establecerá una base sólida para la comprensión de la programación, guiando a los estudiantes en la aplicación de un enfoque estructurado y metódico conocido como lógica de programación para la resolución efectiva de problemas.

2.1. Abstracción, lógica y algoritmo

La abstracción, la lógica y la creación de algoritmos son procesos mentales esenciales del pensamiento computacional y se consideran habilidades cognitivas de alto nivel, cada uno aportando diferentes formas de razonamiento y capacidades analíticas.

En el ámbito de la programación, estos tres procesos mentales trabajan en conjunto para permitir a los desarrolladores diseñar soluciones de *software*. La abstracción simplifica los problemas al dividirlos en partes manejables, eliminando los detalles irrelevantes. La lógica asegura que cada una de estas partes funcione correctamente, siguiendo un razonamiento estructurado y coherente. Por último, los algoritmos definen los pasos específicos necesarios para resolver cada componente del problema.

La integración de la abstracción, la lógica y los algoritmos permite a los desarrolladores crear *software* que se adapten a las necesidades de los usuarios.

2.1.1. Abstracción, lógica y algoritmos en programación

Cuando nos solicitan crear un programa para resolver un problema o satisfacer una necesidad, es fundamental seguir una serie de pasos que

nos garanticen el éxito del desarrollo. Estos pasos, aunque sencillos y familiares, son clave en el proceso de creación:

1. Identificación del problema: reconocemos y definimos claramente el problema que queremos solucionar.
2. Diseño de una solución: pensamos en una solución efectiva que aborde el problema de manera adecuada.
3. Planificación de la estrategia: desarrollamos una estrategia detallada que guiará la implementación de la solución.
4. Ejecución de la estrategia: implementamos la solución de acuerdo con la estrategia diseñada.
5. Evaluación y uso: probamos y ponemos en funcionamiento la solución, asegurándonos de que cumpla con los objetivos establecidos.

2.1.2. Integración de los tres conceptos

La **abstracción** es el proceso de simplificar un problema al centrarse en los aspectos más importantes y relevantes, dejando de lado los detalles innecesarios. En informática, la abstracción permite a los programadores y diseñadores de sistemas manejar la complejidad dividiendo los problemas en niveles más manejables. Los niveles de abstracción pueden variar desde conceptos de alto nivel, como estructuras de datos y arquitecturas de *software*, hasta detalles de bajo nivel, como la implementación de algoritmos específicos. La abstracción de alto nivel se refiere a la simplificación de la representación de un sistema o problema al enfocarse en las ideas y conceptos generales, sin preocuparse por los detalles específicos de la implementación. En esta forma de abstracción, se ocultan los detalles internos y se proporciona una visión más general y accesible del sistema, por ejemplo, una interfaces de usuario: las interfaces gráficas de usuario (GUI) que ofrecen botones, menús y cuadros de diálogo permiten a los usuarios interactuar con el *software* sin necesidad de conocer el código o la lógica interna ni los algoritmos que lo soportan.

Ejemplo de abstracción

¿Qué define a una mesa como tal? Si comparamos diferentes mesas, encontraremos más diferencias que similitudes: pueden variar en color, material, forma, tamaño, grosor y estado de conservación. Sin embargo, a pesar de estas diferencias, somos capaces de reconocer qué objetos son mesas y cuáles no.
Lo que hace que una mesa sea una mesa no depende de su forma (cuadrada, redonda o rectangular), del material con el que esté hecha (madera, mármol, metal) ni de su color (verde, amarillo o rojo). En su lugar, realizamos un proceso de abstracción en el que identificamos sus características esenciales, separándolas de los atributos secundarios o accidentales. Este proceso mental nos permite construir el concepto de mesa, reconociendo su esencia más allá de sus variaciones físicas.

(Platón, s. IV a.C.; Aristóteles, s. IV a.C.).

Si aplicamos el proceso de abstracción al desafío de identificar una fruta con los ojos vendados, podemos reconocer que una manzana no puede ser una naranja debido a sus características esenciales, como la textura, la estructura interna, la cantidad de jugo y la forma en que se consume. Aunque ambas pertenecen al grupo de las frutas, sus diferencias fundamentales las hacen inconfundibles.

En resumen, al eliminar sus características accidentales (como el color o el tamaño) y centrarnos en sus atributos esenciales, seguimos distinguiendo con claridad que una manzana es una manzana y una naranja es una naranja.

Para reforzar la idea de abstracción, podemos decir que la abstracción es el proceso mental mediante el cual identificamos las características esenciales de un objeto o concepto, separándolas de sus atributos secundarios o accidentales. Esto nos permite reconocer, clasificar y comprender distintos elementos sin necesidad de fijarnos en sus variaciones superficiales.

La **lógica** se ocupa del razonamiento válido y de la deducción correcta. El razonamiento lógico soluciona los problemas utilizando la coherencia,

lo que permite organizar las ideas y optimizar la relación entre ellas. Además, emplea la capacidad racional para analizar, comprender y resolver problemas, y utiliza la deducción para establecer conclusiones precisas.

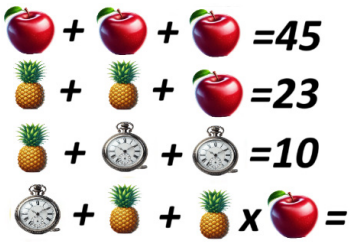


Figura 11. Problema lógico-matemático.
Imágenes generadas con inteligencia artificial.

La resolución de problemas mediante el razonamiento lógico es:

- Deductivo: parte de ideas generales para llegar a conclusiones particulares
- Analítico: desglosan sus ideas para facilitar su entendimiento
- Organizado: ordenar las ideas para analizarlas en orden lógico, una detrás de otras.

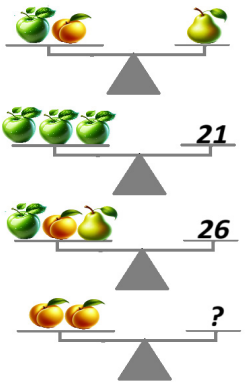


Figura 12. Problema lógico-matemático.
Imágenes generadas con inteligencia artificial.

Un **algoritmo** es una secuencia finita de instrucciones o pasos bien definidos que, ejecutados en un orden específico (lógico), permiten resolver un problema o realizar una tarea.

Los algoritmos son fundamentales en el campo de la informática y las matemáticas, y se utilizan para procesar datos, realizar cálculos y tomar decisiones automatizadas.

Un **algoritmo** tiene las siguientes características:

- *Inicio:* el algoritmo posee un estado inicial
- Finitud: debe tener un número finito de pasos.
- Claridad: cada paso del algoritmo debe ser claro y no ambiguo.
- Entrada(s): requiere de una o más entradas, que son datos iniciales necesarios para ejecutar el algoritmo.
- Estado final: debe tener un estado final que es el resultado de la ejecución del algoritmo.

Los algoritmos son fundamentales en el campo del desarrollo de *software*, inteligencia artificial (IA), ciencia de datos y Big data, logística y transporte, robótica, automatización industrial, finanzas y otros. Los algoritmos son omnipresentes en casi todos los sectores, ayudando a automatizar, optimizar y mejorar la eficiencia de innumerables procesos y tecnologías.

Ejemplo de un algoritmo para calcular la depreciación por línea recta

La depreciación por línea recta es un método contable que asigna el mismo monto de depreciación a cada año durante la vida útil de un activo.

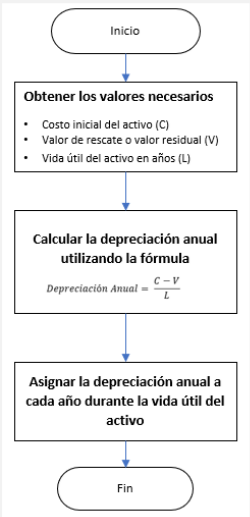
Pasos del Algoritmo	Diagrama del Algoritmo
1. **Obtener los valores necesarios:** • Costo inicial del activo (C). • Valor de rescate o valor residual (V). • Vida útil del activo en años (L). 2. **Calcular la depreciación anual utilizando la fórmula:** $Depreciación\ Anual = \frac{C - V}{L}$ 3. **Asignar la depreciación anual a cada año durante la vida útil del activo.**	

Figura 13. Algoritmo para calcular la depreciación por línea recta.

Ejemplo con datos empíricos para calcular la depreciación en línea recta:

Supongamos que tenemos un equipo de oficina con:
Costo inicial: $10.000
Valor residual: $2.000
Vida útil: 5 años
Cálculo de la depreciación anual:

1. Obtener valores:

- Costo inicial (C): $10.000
- Valor residual (V): $2.000
- Vida útil (L): 5 años

2. Calcular la depreciación anual:

$$Depreciación\ Anual = \frac{10.000 - 2.000}{5} = \frac{8.000}{5} = \$1.600$$

3. Asignar la depreciación anual a cada año:

- Año 1: $1.600
- Año 2: $1.600
- Año 3: $1.600
- Año 4: $1.600
- Año 5: $1.600

El funcionamiento eficiente de las herramientas tecnológicas se debe al uso de la **lógica de programación** con que están programadas y son producto de la habilidad de un programador para resolver problemas por medio del pensamiento estructurado.

2.2. Diagramas de flujo (representación gráfica de algoritmos)

Un *diagrama de flujo* es una representación gráfica de uno o más algoritmos. Utiliza símbolos y flechas para mostrar la secuencia de pasos, decisiones y acciones involucradas en la ejecución de un algoritmo para dar solución a un problema específico. Los diagramas de flujo son utilizados para visualizar, entender y comunicar cómo se desarrolla el proceso del algoritmo, desde el inicio hasta el final.

Para graficar diagramas de flujo se utilizan símbolos estandarizados que indican acciones, decisiones, entradas y salidas. A continuación, se muestra una tabla con los símbolos y su representación asociada:

Símbolo	Acción	Representa
Inicio/Fin	Inicio / Final	El inicio o final de un algoritmo
→	Dirección del flujo	Orden lógico en que se ejecuta el algoritmo
Entrada	Entrada	Entrada de datos
Salida	Salida	Salida de datos o mensajes
Asignación	Asignación	Ejecución de una actividad proceso o tarea
Decisión	Decisión	Bifurcación basada en la respuesta a una evaluación, la cual puede ser "verdadera" o "falsa"

Figura 14. Componentes gráficos que se usan en un diagrama de flujo.

Para explicar de forma práctica cómo se utilizan estos símbolos se muestra un diseño de un proceso algorítmico que resuelve el planteamiento de un problema sencillo. Pensemos en un algoritmo que sume dos valores cualesquiera y que muestre el resultado de la suma.

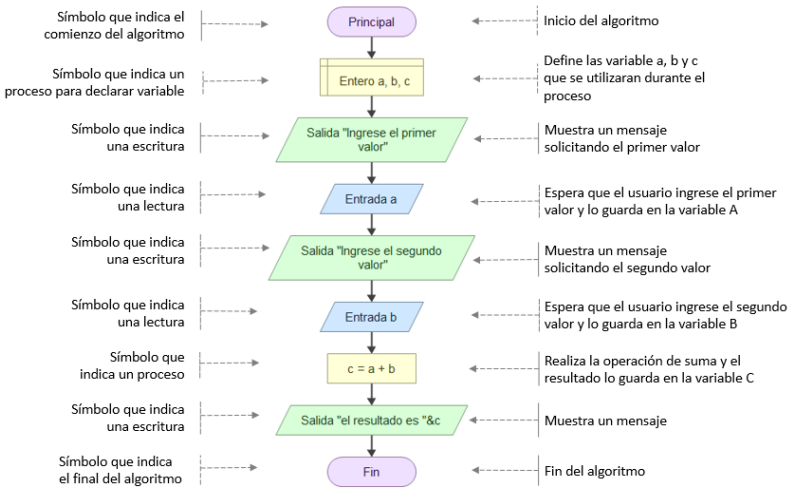


Figura 15. Diagrama de flujo de un algoritmo.

Veamos otro, uno que tome decisiones, que nos diga qué hacer dependiendo de la evaluación de una condición: por ejemplo, construya un algoritmo que tome dos números de entrada, los compare e indique cuál de los dos números es el mayor.

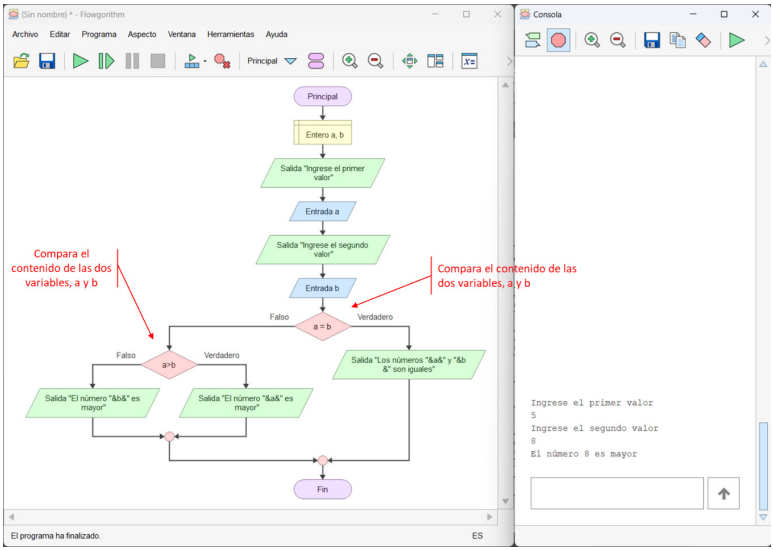


Figura 16. Diagrama de flujo que contiene un control del flujo

El diagrama de flujo es una herramienta sencilla que facilita a los estudiantes el aprendizaje de la programación. Al usar una representación visual, permite entender los procesos de manera clara, haciendo los conceptos más accesibles. Su enfoque gráfico permite a los estudiantes visualizar el flujo de un algoritmo sin sentirse abrumados por el código, lo que reduce el temor a la programación y fomenta una mayor comprensión y confianza en el desarrollo de programas.

Cuando los estudiantes comienzan a programar, generalmente se enfrentan a lenguajes de programación basados en texto. Dependiendo del lenguaje elegido, esta experiencia puede resultar sencilla o, en muchos casos, frustrante.

Con el uso de diagramas de flujo, los estudiantes pueden concentrarse en los principios fundamentales de la programación sin la distracción de los detalles específicos de un lenguaje.

Para nuestra práctica de aprendizaje utilizaremos la herramienta Flowgorithm que nos permitirá dibujar y ejecutar nuestros diagramas de flujo, brindándonos la oportunidad de ver en acción nuestro algoritmo de manera intuitiva, sin necesidad de escribir código complicado. Tal como lo indica en su página principal: "Flowgorithm es un lenguaje de programación gratuito para principiantes que se basa en diagramas de flujo gráficos".

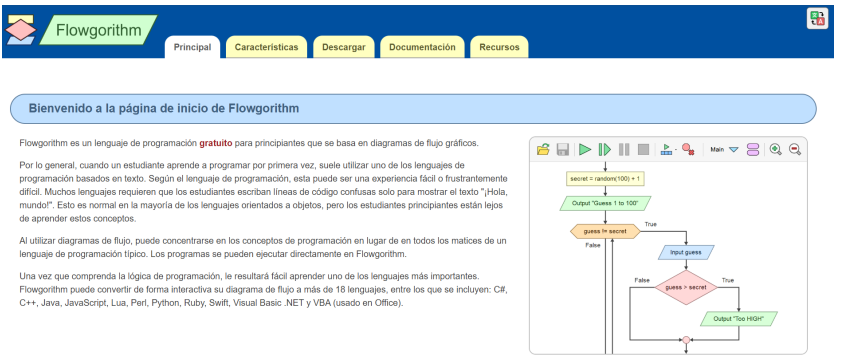


Figura 17. App Flowgorithm para diseñar diagramas de flujo.
Fuente: http://www.flowgorithm.org/

2.3. Modelando algoritmos a través de diagramas de flujos

Los algoritmos se pueden categorizar de diversas maneras, según cómo están estructurados y el tipo de operación que realizan. A continuación, se presentan una serie de ejercicios que los llamaremos "desafíos", categorizados según el tipo de algoritmo en la que se basa su estructura.

2.3.1. Algoritmos lineales

En un algoritmo lineal, las instrucciones se ejecutan en un orden secuencial, una tras otra. No hay bifurcaciones ni repeticiones, y cada paso depende directamente del anterior. Su característica principal es que son simples y fáciles de entender.

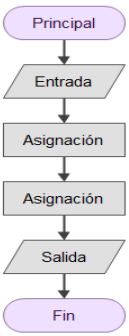


Figura 18. Esquema de un algoritmo lineal.

Para afianzar tu aprendizaje, te invito a resolver los desafíos sobre algoritmos lineales que se encuentran en el Apéndice A. Estos ejercicios te ayudarán a aplicar los conceptos aprendidos y a mejorar tu habilidad en la resolución de problemas.

2.3.2. Algoritmos condicionales (bifurcación)

Estos algoritmos se basan en la evaluación de una condición para tomar decisiones y seguir un flujo determinado. Utilizan estructuras de control que, al evaluar una condición, deciden qué conjunto de instrucciones ejecutar. Su característica principal es que permiten la toma de decisiones dinámicas, ajustando el flujo del programa según el resultado de dicha evaluación.

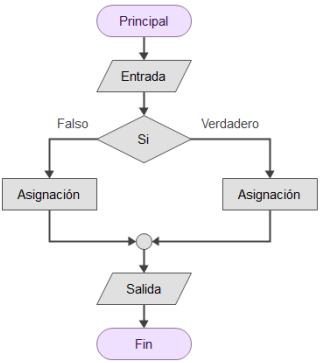


Figura 19. Esquema de un algoritmo condicional.

Para reforzar tu comprensión, te invito a resolver los desafíos sobre algoritmos condicionales que encontrarás en el Apéndice A. Estos ejercicios te ayudarán a aplicar los conceptos aprendidos y a mejorar tu capacidad de análisis y resolución de problemas. ¡Anímate a completarlos!

2.3.3. Algoritmos iterativos (bucles)

Un bucle, en programación, es una secuencia de instrucciones de código que se ejecuta repetidas veces, hasta que la condición de finalización o término del ciclo se cumpla. Se implementan mediante estructuras de bucle como *for* (Para), *while* (Mientras), o *do-while* (Hacer).

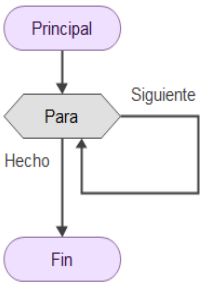
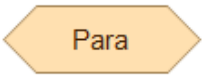
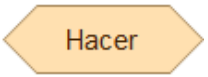
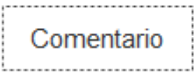


Figura 20. Esquema de un algoritmo iterativo.

En esta parte del libro incluiré cuatro elementos para los diagramas de flujo y que nos permitirá desarrollar los ejercicios:

Tabla 1. Figuras de Flowgorithm

Símbolo	Acción	Representa
	Bucle	Un bucle **Mientras** evalúa una expresión booleana y, si esta es verdadera, ejecuta un bloque de instrucciones. Tras ejecutar dichas instrucciones, el bucle vuelve a verificar la condición booleana. El ciclo continuará repitiéndose mientras la expresión sea verdadera. Una vez que la expresión se evalúe como falsa, el bucle se detendrá.

Símbolo	Acción	Representa
	Bucle	El bucle **Para** es una estructura de control que permite repetir un bloque de código un número específico de veces. A diferencia del ciclo **Mientras**, el ciclo **Para** generalmente se utiliza cuando se conoce de antemano cuántas veces se debe ejecutar el bloque de código.
	Bucle	El bucle **Hacer** es una estructura de control que permite ejecutar un bloque de código repetidamente **al menos una vez**, ya que evalúa la condición de continuación **al final** de cada iteración. Esto lo diferencia de otros bucles, como **Mientras** o **Para**, que evalúan la condición **antes** de ejecutar el bloque de código. El ciclo **Hacer** garantiza que el bloque de código se ejecute al menos una vez, incluso si la condición es falsa desde el principio.
	Comentario	Se utiliza para escribir comentarios que documenten el algoritmo.

Fuente: Flowgorithm.

Para afianzar tu aprendizaje, te animo a enfrentar los desafíos sobre algoritmos con bucles que encontrarás en el Apéndice A. Resolver estos ejercicios te permitirá poner en práctica los conceptos estudiados y perfeccionar tus habilidades en programación. ¡Manos a la obra!

2.3.4. Algoritmos iterativos (bucles) y arreglos

En programación, un arreglo o vector es una estructura de datos que almacena múltiples valores en un bloque de memoria continuo. Cada valor del arreglo se denomina elemento, y se accede a ellos mediante un índice. El primer elemento generalmente tiene el índice 0 (en la mayoría de los lenguajes de programación).

Elemento	A	B	X	I	Z	Y	C	W	G	T
Índice	0	1	2	3	4	5	6	7	8	9

Figura 21. Arreglo o vector.

Ejemplo de aplicabilidad:

Construya un diagrama de flujo que contenga un algoritmo para obtener la desviación estándar de un conjunto de datos.

Consideraciones/restricciones:

1. Utilice arreglos para almacenar los datos (números).
2. Se pueden ingresar n números.
3. Al finalizar el proceso debe mostrar el resultado de la operación.
4. Utilice la siguiente fórmula de desviación estándar para una muestra:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2}$$

Donde:

- n es el número de elementos.
- x_i representa cada valor en el conjunto de datos.
- μ es la media del conjunto de datos.

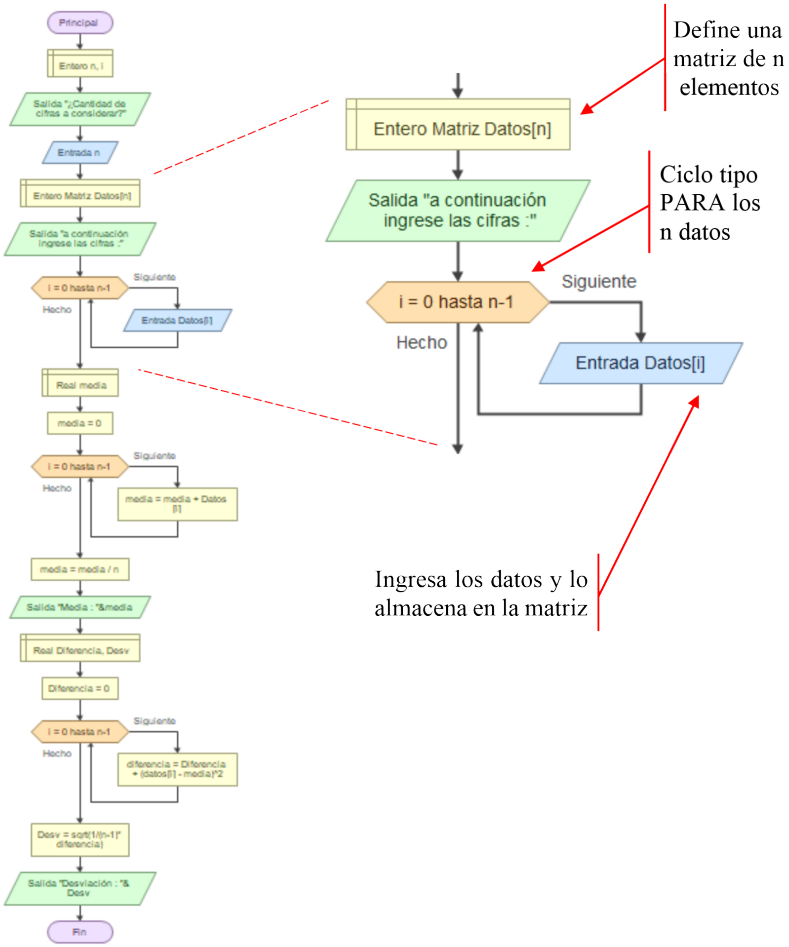


Figura 22. Ejemplo algoritmo desviación estándar.

Tabla 2. Pruebas de algoritmo de desviación estándar

Pruebas a realizar							
Números							Resultado
178	183	170	179	175	186		Desviación estándar: 5,68330889535313
50	60	70	80	90			Desviación estándar: 15,8113883008419
72	68	70	74				Desviación estándar: 2,58198889747161
20	90	35	88	15	24	66	Desviación estándar: 36,0404864932019

Para reforzar tu aprendizaje, te invito a resolver los desafíos sobre bucles con arreglos que se encuentran en el Apéndice A. Estos ejercicios te ayudarán a aplicar los conceptos estudiados y a desarrollar tu lógica de programación. ¡Atrévete a resolverlos!

Capítulo 3
Programación para VBA

Los conceptos básicos de programación son esenciales para el desarrollo de *software*, ya que proporcionan las herramientas necesarias para escribir, leer y mantener código de manera efectiva. Estos fundamentos son aplicables a la mayoría de los lenguajes de programación y resultan clave para comprender la estructura de un programa, manipular datos, controlar el flujo de ejecución, interactuar con los usuarios y realizar una depuración del código.

Objetivos de aprendizaje

1. Identificar y describir las características principales del entorno de desarrollo de VBA, reconociendo sus herramientas y funciones básicas para facilitar el desarrollo de programas.
2. Comprender la relevancia de la sintaxis y la semántica en programación, analizando cómo influyen estos elementos en la correcta ejecución de un programa en VBA.
3. Reconocer y aplicar los conceptos de constantes, variables, tipos de datos y estructuras de datos diferentes contextos de programación en VBA.
4. Distinguir entre estructuras de control condicionales y estructuras de bucle, determinando criterios para su uso adecuado en diferentes procesos y tomar decisiones dinámicas en programación en VBA.
5. Identificar, diagnosticar y corregir errores en el código, empleando herramientas de depuración y técnicas de manejo de errores, para asegurar la integridad y funcionalidad del código.

Resultados de aprendizaje

Al finalizar el capítulo, se espera que las personas logren navegar y utilizar eficazmente el entorno de desarrollo VBA, comprendiendo la importancia de la sintaxis y la semántica para la correcta ejecución de programas. Además, aplicarán conceptos fundamentales como constantes, variables, tipos de datos y estructuras de datos en contextos prácticos, analizarán y seleccionarán estructuras de control condicionales y bucles según las necesidades de programación, y serán capaces de identificar, diagnosticar y corregir errores en el código mediante herramientas de depuración. Finalmente, diseñarán y desarrollarán procedimientos básicos en VBA que integren elementos clave de programación para automatizar tareas específicas en Excel de manera eficiente y funcional.

3.1. Entorno de trabajo de un lenguaje de programación

El entorno de trabajo de un lenguaje de programación, también llamado IDE (Integrated Development Environment, por sus siglas en inglés) es un Entorno de Desarrollo Integrado que proporciona un conjunto de herramientas y recursos que facilitan el desarrollo de *software*. Un IDE agrupa, en una sola aplicación, todas las funcionalidades que un programador necesita para escribir, probar, depurar y ejecutar código.

3.2. IDE para VBA

El entorno de trabajo para VBA (Visual Basic for Applications) en Excel es conocido como el Editor de Visual Basic (VBE) y proporciona todas las herramientas necesarias para escribir, editar, depurar y ejecutar código VBA dentro de Excel. Se puede acceder al VBE directamente desde Excel, seleccionando primero Programado y luego Visual Basic. Si no aparece la opción Programador debes activarla seleccionando **Archivo** + **Opciones** + **Personalizar cinta de opciones** + **Programador**, esta última debe seleccionar la casilla para activarla.

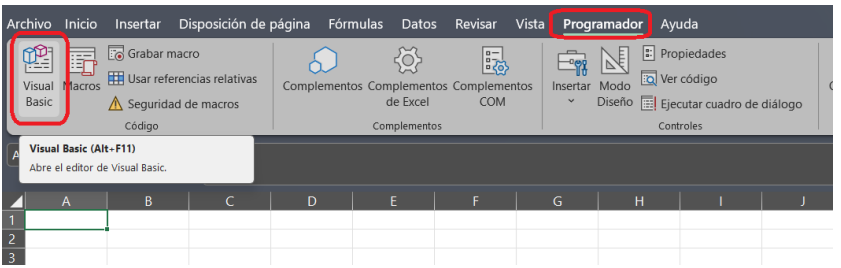


Figura 23. Cinta de opciones de Excel
(Microsoft Office Profesional Plus 2021).

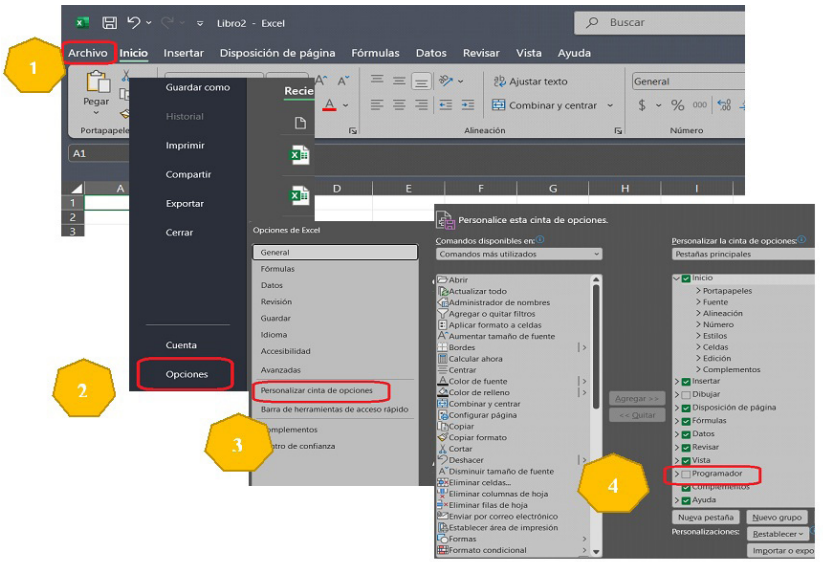


Figura 24. Cómo activar la opción PROGRAMADOR
(Microsoft Office Profesional Plus 2021).

Una vez que esté activada la opción para **Programador**, deberá seguir los pasos para acceder al Editor de Visual Basic de VBA.

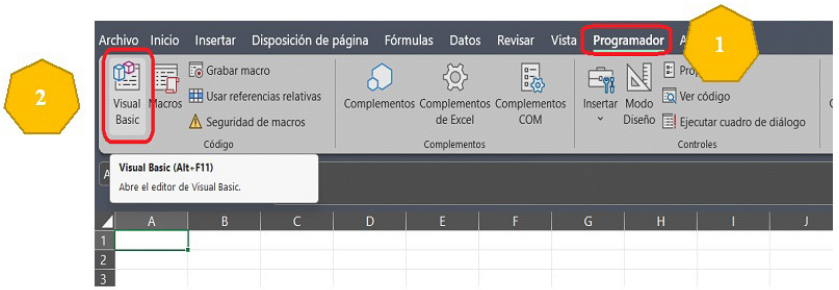


Figura 25. Cómo acceder al Editor de Visual Basic de VBA (Microsoft Office Profesional Plus 2021).

3.3. Cómo interpretar la ventana del editor de código

Esta es la ventana principal donde se escribe y edita el código VBA. Incluye funciones como el resaltado de sintaxis y el autocompletado, que facilitan la escritura de código.

1. Ventana de proyecto (explorador de proyectos):

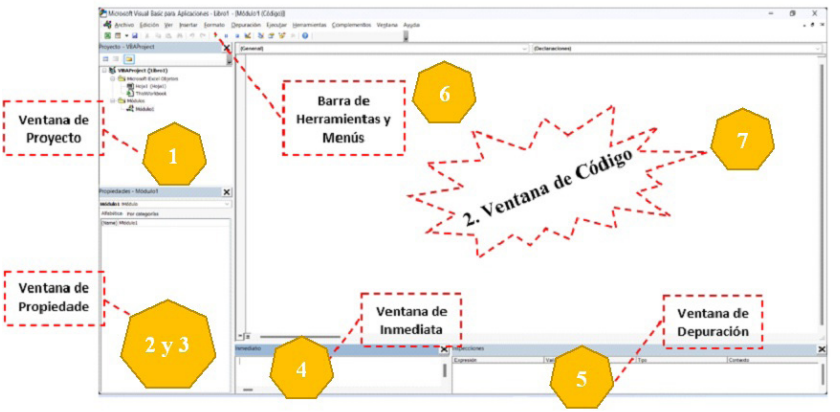


Figura 26. Ventana del editor de Visual Basic (Microsoft Office Profesional Plus 2021).

Esta ventana muestra todos los objetos de VBA que forman parte del proyecto actual, como hojas de cálculo, libros de trabajo y módulos de código. Permite navegar fácilmente entre los diferentes componentes del proyecto VBA, simplemente haciendo doble clic sobre el componente.

2. Ventana de propiedades:

Muestra las propiedades del objeto seleccionado en el Explorador de Proyectos, permitiendo ajustar las propiedades de los objetos VBA, como el nombre de un módulo o las propiedades de un control de formulario.

3. Ventana de propiedades del módulo:

Permite gestionar módulos individuales (unidades de código VBA) dentro del proyecto. Los módulos pueden ser módulos estándar, módulos de clase o módulos de objetos (como hojas de trabajo y libros de trabajo).

4. Barra de herramientas y menús:

Proporciona accesos directos a herramientas comunes como abrir, guardar, ejecutar código, establecer puntos de interrupción, y controlar la ejecución del código durante la depuración.

5. Ventana de código:

Aquí es donde puedes escribir y editar el código VBA.

Algunos consejos de utilidad

Atajos de teclado:
- Usa F5 para ejecutar Macros,
- F8 para ejecutar el código, línea por línea,
- Ctrl + Espacio para la autocompletación.

Ventana inmediata:
- Puedes abrir la ventana inmediata (Ctrl + G) para probar código en tiempo real y ver resultados de depuración.

Depuración:
- Inserta puntos de interrupción haciendo clic en el margen izquierdo de la línea de código o presionando F9.

3.4. Dónde se escribe el código VBA

¡Listo para escribir y probar nuestra primera Macro, vamos!

1) Abrir el Editor de Visual Basic (VBE):

En Excel, ve a la pestaña **Programador + Visual Basic** o utiliza el atajo de teclado **Alt + F11** para abrir el VBE.

2) Crear un módulo de código:

- En el VBE, haz clic derecho en el **Proyecto** de VBA correspondiente a tu archivo de Excel en el Explorador de Proyectos.
- Selecciona **Insertar + Módulo** para añadir un nuevo módulo de código donde puedes empezar a escribir tus Macros o funciones VBA.

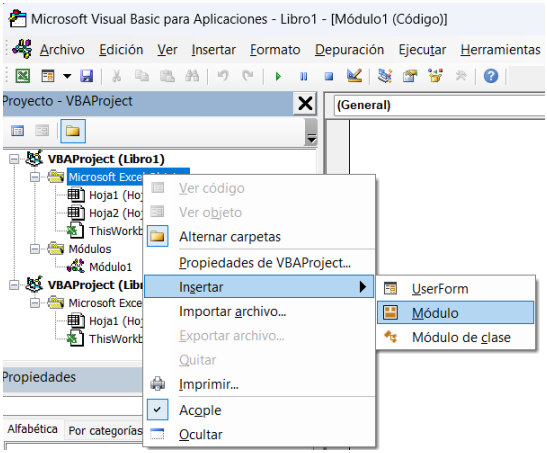


Figura 27. Imagen del VBE para crear un módulo (Microsoft Office Profesional Plus 2021).

3) Escribir código VBA:

Utiliza la Ventana del Editor de Código para escribir y editar tu código VBA. Puedes crear procedimientos (Sub) o funciones (Function) según sea necesario.

- Por ejemplo, copia en la ventana de código lo que se muestra en la siguiente figura. Ese código corresponde a una Macro que nos enviará un mensaje a la pantalla que dirá "¡Hola Mundo!".

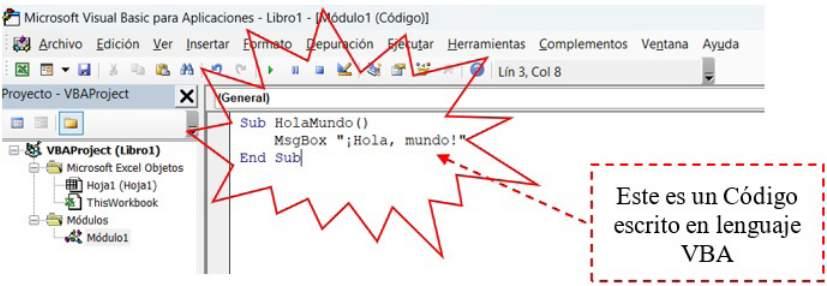


Figura 28. Imagen que muestra el código de una Macro (Microsoft Office Profesional Plus 2021).

Luego, cuando lo tengas escrito, presiona ejecutar código para ver el resultado:

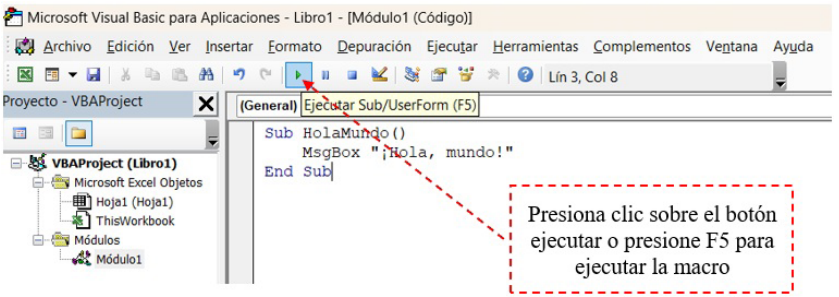


Figura 29. Imagen que muestra el botón de ejecución (Microsoft Office Profesional Plus 2021).

Verá que la ejecución generó una ventana emergente que dice ¡Hola Mundo!

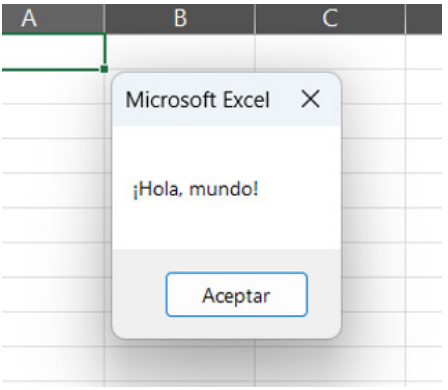


Figura 30. Imagen que muestra el resultado de la ejecución de la Macro.

Ejecutar y depurar código:

Cuando programamos en VBA (Visual Basic for Applications), es esencial asegurarnos de que el código funcione correctamente y, en caso de errores, contar con herramientas para analizarlos y corregirlos. Para ello, es importante conocer cómo ejecutar una Macro y cómo depurarla.

- Ejecutar una Macro en VBA.

Existen varias formas de ejecutar una Macro en VBA, y puedes elegir la más adecuada según el contexto:

1) Desde el Editor de VBA (VBE)
   - Pulsa F5 o haz clic en el botón Ejecutar (icono de "Play").
   - Esto ejecutará todo el código del módulo.

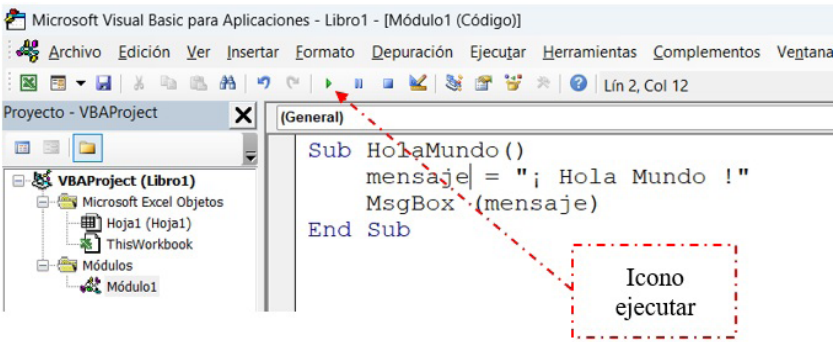


Figura 31. Imagen que muestra el icono "Play" (Microsoft Office Profesional Plus 2021).

2) Desde Excel

- Ve a la pestaña Programador.
- Haz clic en Macros.
- Selecciona la Macro y presiona Ejecutar.

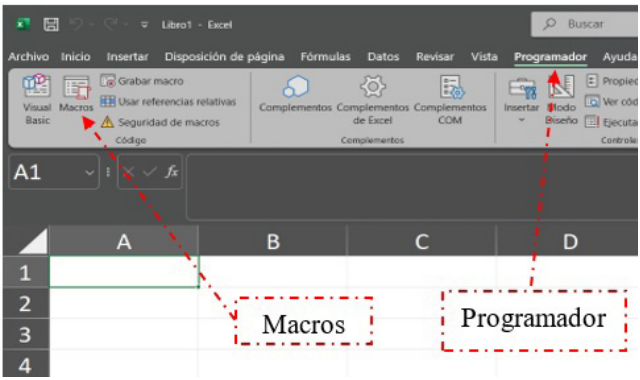


Figura 32. Imagen que muestra la opción Programador y Macros (Microsoft Office Profesional Plus 2021).

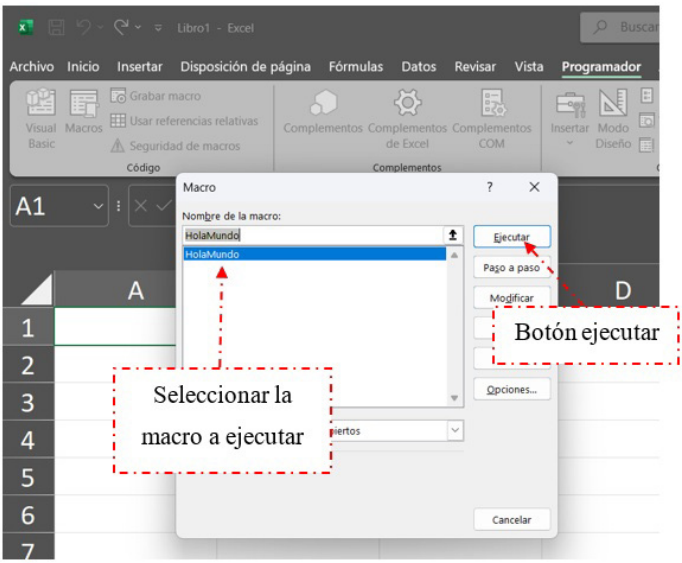


Figura 33. Imagen que muestra el botón ejecutar una Macro (Microsoft Office Profesional Plus 2021).

- Depurar Código VBA

La depuración es el proceso de detectar y corregir errores en el código. VBA ofrece herramientas que permiten analizar su comportamiento del código en tiempo real para encontrar posibles fallos y mejorar su funcionamiento.

1) Ejecutar paso a paso (depuración visual)

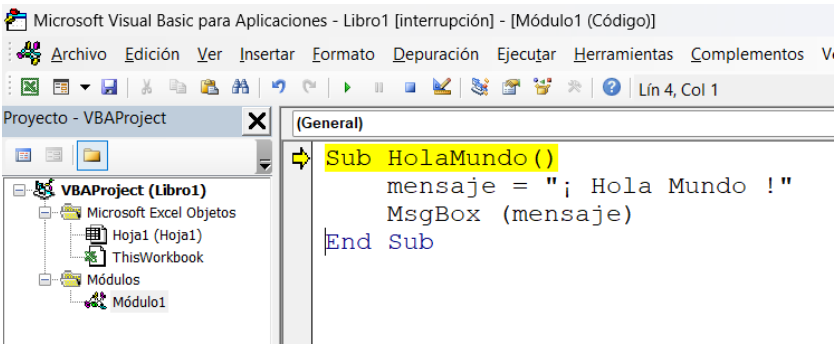


Figura 34. Imagen muestra la ejecución paso a paso (Microsoft Office Profesional Plus 2021).

- Presione F8 para ejecutar el códigolínea por línea.

Ejecutar el código línea por línea (paso a paso), con F8 permite ver cómo se comporta el programa en cada paso. Es útil para detectar en qué momento ocurre un error o evaluar cómo cambian los valores de las variables.

2) Ejecutar con puntos de interrupción (depuración visual)

Un **punto de interrupción** detiene la ejecución en una línea específica del código. Para establecerlo haz clic en el margen izquierdo de la línea de código o presiona F9, como se muestra en la siguiente figura:

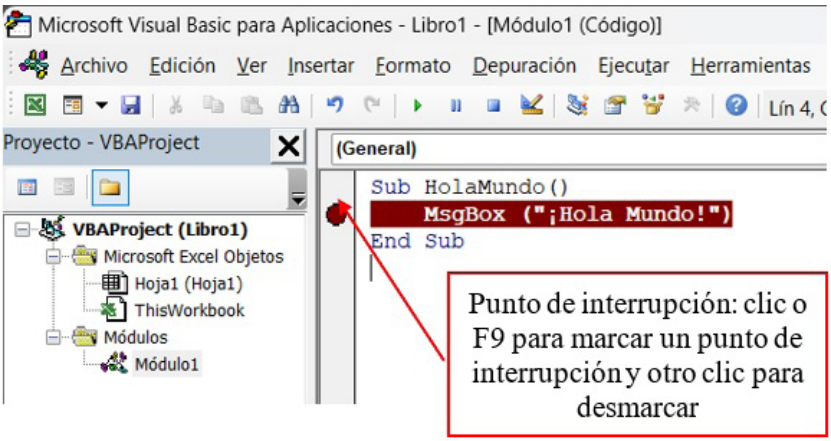


Figura 35. Imagen que muestra dónde marcar el punto de interrupción (Microsoft Office Profesional Plus 2021).

Cuando se ejecute la Macro, se detendrá en esa línea, permitiendo inspeccionar las variables antes de continuar.

3) Ventana de Inmediato (Ctrl + G)

Permite ejecutar comandos de VBA y ver los resultados sin necesidad de ejecutar toda la Macro. Es útil para probar pequeños fragmentos de código y verificar valores en tiempo real.

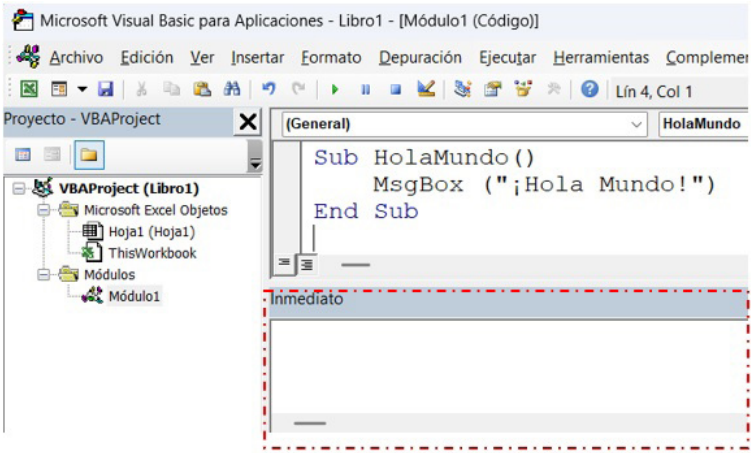


Figura 36. Imagen que muestra la ventana Inmediato de VBA (Microsoft Office Profesional Plus 2021).

4) Watch Window (ventana de inspección)

Supervisa valores de variables a medida que se ejecuta el código, lo que ayuda a detectar cambios inesperados y depurar errores de manera eficiente.

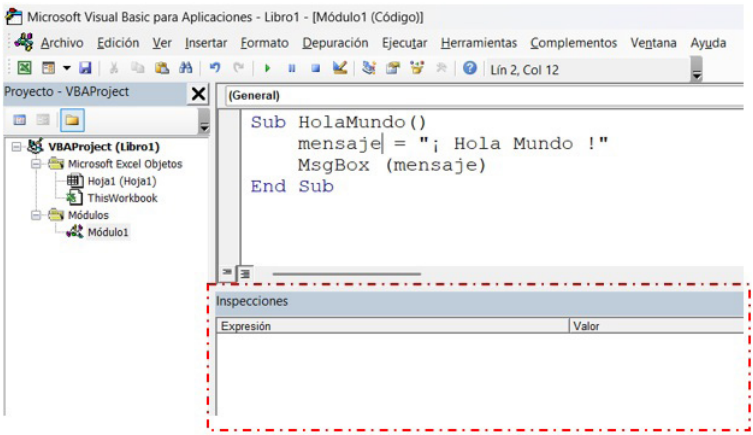


Figura 37. Imagen que muestra la ventana Inspecciones de VBA (Microsoft Office Profesional Plus 2021).

Beneficios de depurar código:

- Facilita la detección de problemas de lógica en el código.
- Permite visualizar valores de variables en tiempo real.
- Facilita el análisis del flujo del programa.

3.5. Fundamentos de un lenguaje de programación

3.5.1. La importancia de la sintaxis en la programación

La sintaxis es el conjunto de reglas que rigen la correcta escritura de un código. Incluye la manera en que se deben declarar variables, escribir instrucciones, utilizar operadores, crear bucles, tomar decisiones condicionales y más. Si el código no sigue estas reglas, se producirá un error de sintaxis y el programa no podrá ejecutarse. La sintaxis considera el uso de palabras claves, asignación de valores, declaración de variables, bloques de código, comentarios, estructura de un procedimiento e indentación (sangría).

La sintaxis en VBA es esencial para asegurar que el código esté escrito correctamente y pueda ser interpretado y ejecutado. Dominar la sintaxis implica conocer las palabras clave, cómo se declaran variables, cómo se escriben los bloques de código, y cómo estructurar correctamente los procedimientos y funciones.

3.5.2. La importancia de la semántica en programación

La semántica es el conjunto de reglas que determinan el significado y comportamiento lógico de las instrucciones en un lenguaje de programación. Su función es garantizar que el código realice las acciones que el programador espera. Un programa puede ser sintácticamente correcto, pero aún contener errores semánticos si no produce el resultado deseado.

Ejemplo de errores semántico:

Supongamos que queremos calcular el área de un rectángulo, la formula dice que

$$\textit{área} = \textit{base} \text{ x } \textit{altura}$$

observemos el siguiente código escrito:

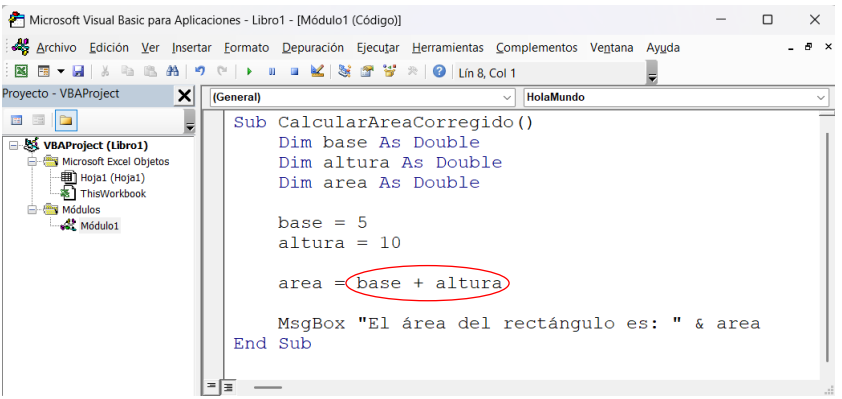


Figura 38. La imagen muestra el error semántico.

Este código no presenta errores de sintaxis, por lo que se ejecuta correctamente y genera un resultado. Sin embargo, el resultado obtenido es incorrecto debido a un error semántico: se ha utilizado el operador de suma (+) en lugar del operador de multiplicación (*), lo que altera por completo el significado del cálculo.

La *semántica* se centra en el *significado y la lógica* de las instrucciones en un programa. Mientras que la *sintaxis* garantiza que el código esté estructurado correctamente, la semántica asegura que el programa realice las acciones esperadas. Ambos aspectos son fundamentales para desarrollar un *software* preciso y funcional.

3.5.3. El lenguaje de programación VBA (Visual Basic for Applications)

VBA es un lenguaje de programación derivado de Visual Basic, desarrollado por Microsoft, que se utiliza para automatizar tareas y crear programas dentro de las aplicaciones de Microsoft Office, como Excel, Word, Access, Outlook y PowerPoint.

¿Por qué aprender VBA y no otro lenguaje?

Aprender VBA (Visual Basic for Applications) tiene múltiples ventajas, especialmente para quienes trabajan con Microsoft Office. Algunas razones clave son:

- Automatización de tareas: permite agilizar procesos repetitivos dentro de Excel, Word y Access, aumentando la productividad.
- Integración nativa: no requiere instalación adicional, ya que está incorporado en Microsoft Office.
- Curva de aprendizaje accesible: es un lenguaje intuitivo y fácil de aprender, ideal para quienes no tienen experiencia en programación.
- Aplicación inmediata en el entorno laboral: los programas creados con VBA pueden utilizarse de inmediato para optimizar tareas en el trabajo.
- Alta demanda en el mercado: las habilidades en VBA son altamente valoradas en sectores que utilizan Microsoft Office, como contabilidad, auditoría y finanzas.
- Excel como herramienta clave: contadores y auditores dependen de Excel para el análisis de datos financieros, y VBA permite potenciar sus capacidades.
- Puente hacia otros lenguajes: una vez dominado VBA, resulta más fácil aprender otros lenguajes de programación como Python o SQL.

Aprender VBA no solo mejora la eficiencia en el uso de Microsoft Office, sino que también abre la puerta a nuevas oportunidades profesionales en automatización y análisis de datos.

3.6. Constantes y variables en VBA

En programación, las constantes y variables son esenciales para almacenar y manipular datos. Ambas representan áreas de memoria, pero su comportamiento es diferente.

- **Constantes: valores fijos durante la ejecución**
  Una *constante* es un espacio en memoria que almacena un valor que no cambia durante toda la ejecución del programa. Se define con un valor inicial que permanece inalterable.
  Por ejemplo, si necesitamos usar el valor de PI (3.14159) en cálculos geométricos, podemos almacenarlo en una constante, ya que su valor es universal y no se modifica.

- **Variables: Espacios de memoria dinámicos**
  En cambio, una variable es como un recipiente en el que podemos almacenar valores que pueden cambiar a lo largo de la ejecución del programa. Podemos agregar, eliminar o modificar su contenido, pero siempre contendrá el último valor asignado.

- **Importancia de las constantes y variables en VBA**
  En cualquier lenguaje de programación, incluidas las Macros en VBA, el uso adecuado de constantes y variables permite organizar mejor el código, reducir errores y hacer que los programas sean más eficientes.

Cada *variable* y *constante* debe tener un *nombre único* para que el programa pueda acceder a sus valores de manera clara y sin confusiones.

- ✓ Constantes → usadas cuando el valor no debe cambiar.
- ✓ Variables → usadas cuando se necesita modificar el valor en el transcurso del programa.

Este concepto es fundamental para programar de manera efectiva y estructurada en VBA.

3.6.1. Declarar una constante

En VBA (Visual Basic for Applications), la sintaxis para declarar una constante sigue esta estructura:

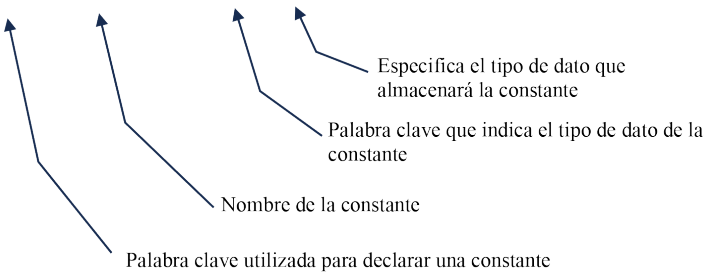
```
const nombreVariable As TipoDeDato
```

Donde:

- *Const:* palabra clave para declarar una constante en VBA.
- *nombreConstante:* nombre asignado a la constante.
- *As:* indica que se va a especificar un tipo de dato.
- *TipoDeDato:* define el tipo de dato que almacenará la constante (por ejemplo, Integer, String, Boolean, etc.).

Ejemplo:

```
Const esMayorDeEdad As Boolean
```



3.6.2. Declarar una variable

En VBA (Visual Basic for Applications), la sintaxis para declarar una variable sigue esta estructura:

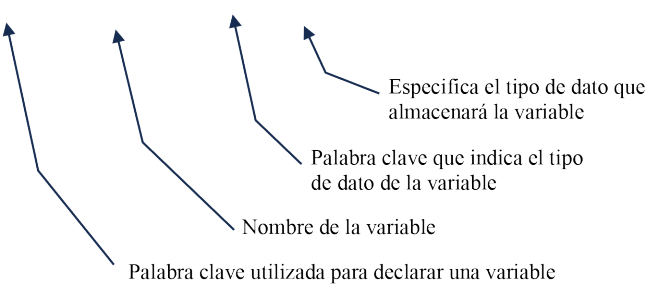
```
Dim nombreVariable As TipoDeDato
```

Donde:

- *Dim:* palabra clave para declarar variables en VBA.
- *nombreVariable:* nombre asignado a la variable.
- *As:* indica que se va a especificar un tipo de dato.
- *TipoDeDato:* define el tipo de dato que almacenará la variable (por ejemplo, Integer, String, Boolean, etc.).

Ejemplo:

```
Dim esMayorDeEdad As Boolean
```



3.6.3. Reglas para definir variables en VBA

Al definir variables es fundamental seguir ciertas reglas que garantizan un código claro, organizado y libre de errores. Estas reglas abarcan aspectos relacionados con la sintaxis, los nombres permitidos y las buenas prácticas de programación.

✓ Declaración de variables
Uso de Dim: en VBA, la declaración de variables es opcional, pero se recomienda utilizar la instrucción Dim antes de emplearlas. Esto mejora la legibilidad del código y evita errores accidentales:

```
Dim edad As Integer
Dim nombre As String
```

✓ Diferentes formas de declarar variables
Se pueden declarar varias variables en una sola línea, separadas por comas:

```
Dim a As Integer, b As Double, c As String
```

✓ Uso de Option Explicit
Para obligar al programa a declarar todas las variables antes de su uso, se puede incluir la instrucción *Option Explicit* al inicio del módulo.

```
Option Explicit

Dim total As Double
total = 100
```

Ventajas de Option Explicit

- Evita errores por nombres mal escritos en las variables.
- Mejora la eficiencia del código al evitar la creación automática de variables no declaradas.

Si una variable no ha sido declarada y Option Explicit está activado, VBA mostrará un error, ayudando a detectar posibles fallos en el código antes de la ejecución.

Seguir estas reglas garantiza un código VBA más estructurado, fácil de depurar y menos propenso a errores.

3.6.4. Reglas para nombrar variables y constantes en VBA

Al definir una variable en VBA, es importante seguir ciertas reglas para garantizar claridad y evitar errores en el código. A continuación, se detallan las principales normas para nombrar variables correctamente:

1) Debe comenzar con una letra:

☑ El primer carácter del nombre de una variable debe ser una letra (A-Z o a-z).

☒ No puede comenzar con un número ni con un símbolo especial.

```
Dim edad As Integer        ' Correcto
Dim 1nombre As String      ' Incorrecto
```

2) No puede contener espacios:

☑ Para separar palabras dentro de una variable, se recomienda utilizar guiones bajos (_) o escribir en notación camelCase.

☒ No se permiten espacios en blanco en el nombre de la variable.

```
Dim miVariable As Integer          ' Correcto
Dim mi_variable As Integer         ' Correcto
Dim mi Variable As String          ' Incorrecto
Dim 1Variable As String            ' Incorrecto
```

3) No debe usar palabras reservadas:

☒ No se pueden utilizar palabras clave propias del lenguaje VBA, como If, For, Next, End, entre otras.

```
Dim For As Integer                 ' Incorrecto
Dim contador As Integer            ' Correcto
```

4) VBA no distingue entre mayúsculas y minúsculas:
En VBA, las variables son insensibles a mayúsculas y minúsculas, por lo que nombre, NOMBRE y Nombre se consideran iguales.

5) Longitud máxima del nombre:
El nombre de una variable puede tener hasta *255 caracteres*, pero *se recomienda usar nombres cortos y descriptivos* para mejorar la legibilidad del código:

```
Casos recomendados:
Dim totalVentas as Double
Dim cantidad_productos as Integer

Caso no recomendado:
Dim variableParaAlmacenarElTotalDeVentasAnualesDeLaEmpresa
```

3.6.5. Tipos de datos y valores predeterminados

Los *tipos de datos* son fundamentales para definir qué clase de información puede ser almacenada en una variable. Cada tipo de dato tiene un propósito específico y permite optimizar el uso de memoria. Cuando se declara una variable, VBA asigna un valor predeterminado según su

tipo. Los tipos de datos permitidos y más utilizados en VBA son los siguientes:

Tabla 3. Tabla con tipos de datos

Tipo de Dato	Descripción	Valor predeterminado
Integer	Almacena valores enteros entre -32,768 y 32,767 (2 bytes). Se utiliza cuando se trabaja con números enteros pequeños que no requieren decimales.	0
Long	Almacena valores enteros grandes entre -2,147,483,648 y 2,147,483,647 (4 bytes).	0
Double	Almacena números con decimales y precisión doble, hasta 15 dígitos de precisión (8 bytes).	0.0
String	Almacena texto. Puede ser de longitud fija (definida por el programador) o de longitud variable (hasta aproximadamente 2 mil millones de caracteres).	“”(Vacío)
Boolean	Almacena valores lógicos True o False (2 bytes).	False
Date	Almacena fechas y horas, con un rango entre 1 de enero del 100 y 31 de diciembre de 9999 (8 bytes).	00:00:00 (30/12/1899)
Currency	Almacena valores monetarios con 4 decimales y maneja rangos de valores entre -922,337,203,685,477.5808 y 922,337,203,685,477.5807 (8 bytes).	0.0000
Variant	Tipo de datos genérico, puede almacenar cualquier tipo de datos, incluidos números, cadenas, fechas y valores nulos (Null). Es útil cuando el tipo de datos no se conoce de antemano, aunque consume más memoria.	Empty

3.6.6. Tipo de dato implícito y el problema de Variant

Si no se especifica un tipo de dato en la declaración, VBA asigna por defecto el tipo *Variant*:

```
Dim totalVentas
```

En esta declaración donde se omite el tipo de datos, VBA lo define como Variant. Esto tiene un inconveniente, consume más memoria y ralentiza la ejecución, por lo que es recomendable evitar el uso de Variant a menos que sea necesario.

3.6.7. Variables locales y globales

- Variables locales
  Son aquellas declaradas dentro de un procedimiento y solo el procedimiento que contiene la declaración puede usarla:

- Variables globales
  Se declaran fuera de los procedimientos y pueden ser utilizadas en todo el módulo o proyecto.
  Para hacer que una variable sea accesible desde cualquier módulo, se usa Public:

3.6.8. Buenas prácticas al nombrar variables y constantes

Para escribir código más claro y fácil de mantener y comprender, es recomendable seguir estas prácticas al nombrar variables:

- Usa nombres descriptivos: el nombre de una variable debe reflejar su propósito en el programa. En lugar de usar nombres genéricos como x o a, es mejor emplear nombres significativos como sumaTotal o edadPersona, que faciliten la comprensión del código.
- Sigue una convención de nombres coherente: mantener un estilo uniforme en la nomenclatura ayuda a mejorar la legibilidad del código. Algunas opciones comunes son:
  ✓ CamelCase → Ejemplo: totalVentasAnuales

✓ Snake_case → Ejemplo: total_ventas_anuales

Adoptar estas prácticas no solo hace que el código sea más claro para ti, sino que también facilita la comprensión a otros programadores.

Ejemplo práctico:

```
Option Explicit ‘ Obliga a declarar todas las variables

Sub CalcularAreaRectangulo()
    ' Declaración de variables
    Dim largo As Double
    Dim ancho As Double
    Dim area As Double

    ' Asignación de valores
    largo = 10.5
    ancho = 5.75

    ' Cálculo del área
    area = largo * ancho

    ' Mostrar el resultado
    MsgBox “El área del rectángulo es “ & area
End Sub
```

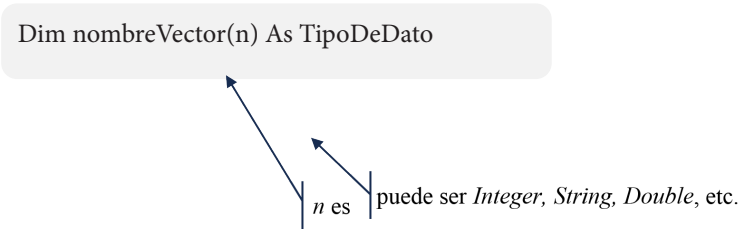
3.7. Estructuras de datos

En VBA, los **vectores** (también llamados **matrices** o **arrays**) son estructuras de datos que permiten almacenar múltiples valores bajo una misma variable. Los vectores en VBA pueden ser de longitud fija o dinámica y pueden tener una o varias dimensiones.

3.7.1. Declaración de vectores de longitud fija

Un vector de longitud fija tiene un tamaño definido cuando se declara y no puede cambiar. Puedes declarar un vector especificando el rango de índices que utilizará.

Sintaxis:



Ejemplo:

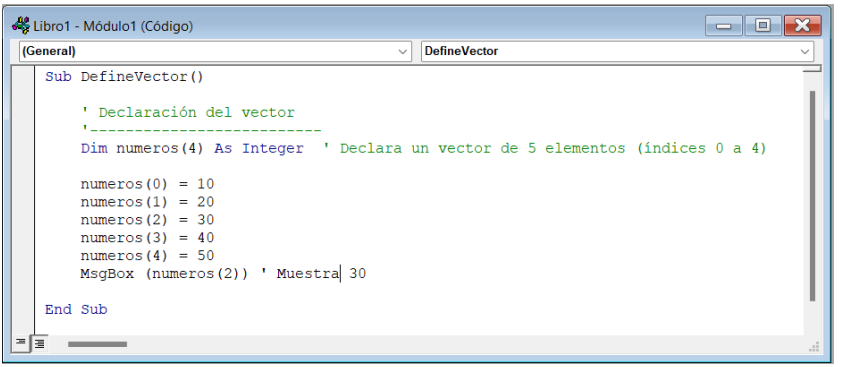


Figura 39. Declarar un vector de longitud fija en VBA.

3.7.2. Vectores de longitud dinámica

Un vector de longitud dinámica permite que el tamaño se ajuste durante la ejecución del programa. Para declarar un vector dinámico, no tienes que especificar el tamaño inicial y lo defines luego usando ReDim (sentencia de VBA).

Sintaxis:

```
Dim nombreVector() As TipoDeDato
ReDim nombreVector(n)      ' Define el tamaño del vector dinámico
```

No se define en índice *n* del vector

Si necesitas conservar los valores actuales contenidos en el vector al redimensionarlo, debes usar ReDim Preserve para preservar los datos.

Ejemplo:

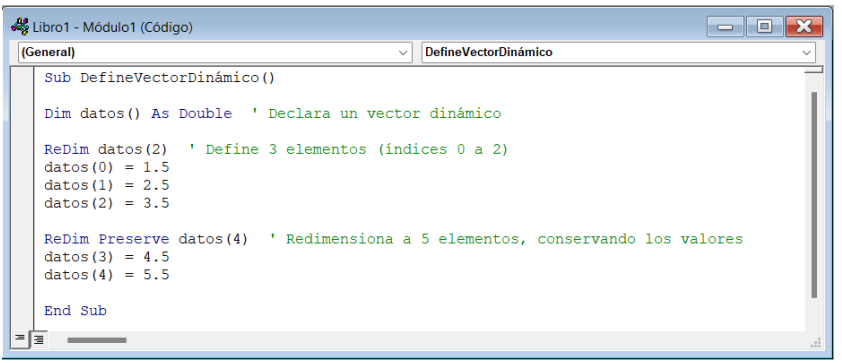
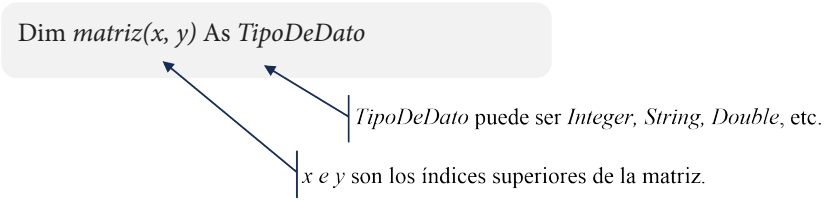


Figura 40. Declara un vector dinámico en VBA.

3.7.3. Vectores multidimensionales

VBA permite vectores de múltiples dimensiones (matrices) para almacenar datos en una estructura de filas y columnas. Puedes declarar una matriz de dos o más dimensiones especificando varios índices.

Sintaxis:



Ejemplo:

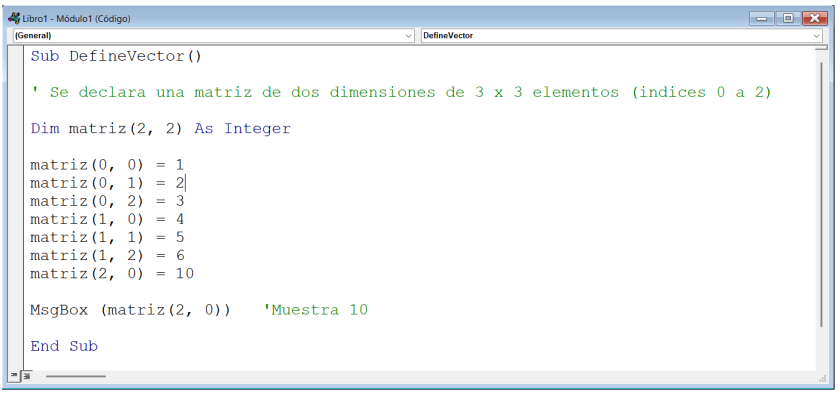


Figura 41. Declara un vector multidimensional en VBA.

3.8. Operadores en VBA

Los *operadores* son símbolos o palabras que se utilizan para realizar operaciones sobre valores o variables. Los operadores permiten realizar cálculos aritméticos, comparaciones lógicas y manipulación de cadenas de texto, entre otras operaciones.

3.8.1. Operadores aritméticos: permiten realizan cálculos

Los operadores se evalúan según un orden de precedencia, lo que determina en qué secuencia se realizarán las operaciones dentro de una expresión que contiene múltiples operadores. La Tabla 4 muestra los operadores aritméticos, junto con su orden de precedencia.

Tabla 4. Operadores aritméticos y precedencia

PRECEDENCIA	SÍMBOLO	DESCRIPCIÓN	EJEMPLO
1	^	Exponenciación (potencia)	2 ^ 4
2	-	Negación	-1
3	* /	Multiplicación y División	2 * 3 | 4 / 2
4	\	División entera	3 \ 2
5	**MOD**	Módulo (devuelve el residual de la división)	3 mod 2
6	+ -	Suma y Resta	4 + 5 | 8 - 3

Veamos un ejemplo: 10 + 5 * 3 ^ 2 - 4 / 2

- Primero se resuelve la operación de potencia, en este caso 3 ^ 2, lo que da como resultado 9.
- Luego, se realiza la multiplicación 5 * 9, que da como resultado 45.
- A continuación, se resuelve la división 4 / 2, lo que da como resultado 2.
- Finalmente, se realiza la suma y la resta, en orden de izquierda a derecha: 10 + 45 - 2, lo que da como resultado 53.

Si deseas cambiar el orden en que se evalúan las operaciones, puedes usar paréntesis. Por ejemplo: (10 + 5) * (3 ^ 2 - 4 / 2)

En este caso, las operaciones dentro de los paréntesis se ejecutan primero, lo que cambia el resultado de la operación, da como resultado 105.

3.8.2. Operadores de concatenación

Permite unir dos o más cadenas de textos.

Tabla 5. Operador de concatenación

OPERADOR	SIGNIFICADO	EJEMPLO	RESULTADO
&	Une dos textos para generar un valor de texto continuo	"Hola" & "Mundo"	"Hola Mundo"

3.8.3. Operadores de comparación

Permiten comparar dos valores y/o expresiones y devuelven un resultado booleano del tipo verdadero o Falso (True or False). Las comparaciones son necesarias para tomar decisiones.

Tabla 6. Operadores de comparación

OPERADOR	SIGNIFICADO	EJEMPLO
=	Igual a	a = b
>	Mayor que	a > b
<	Menor que	a < b
>=	Mayor o igual que ó no menor que	a >= b
<=	Menor o igual que ó no mayor que	a <= b
<>	Distinto de	a <> b

3.8.4. Operadores lógicos

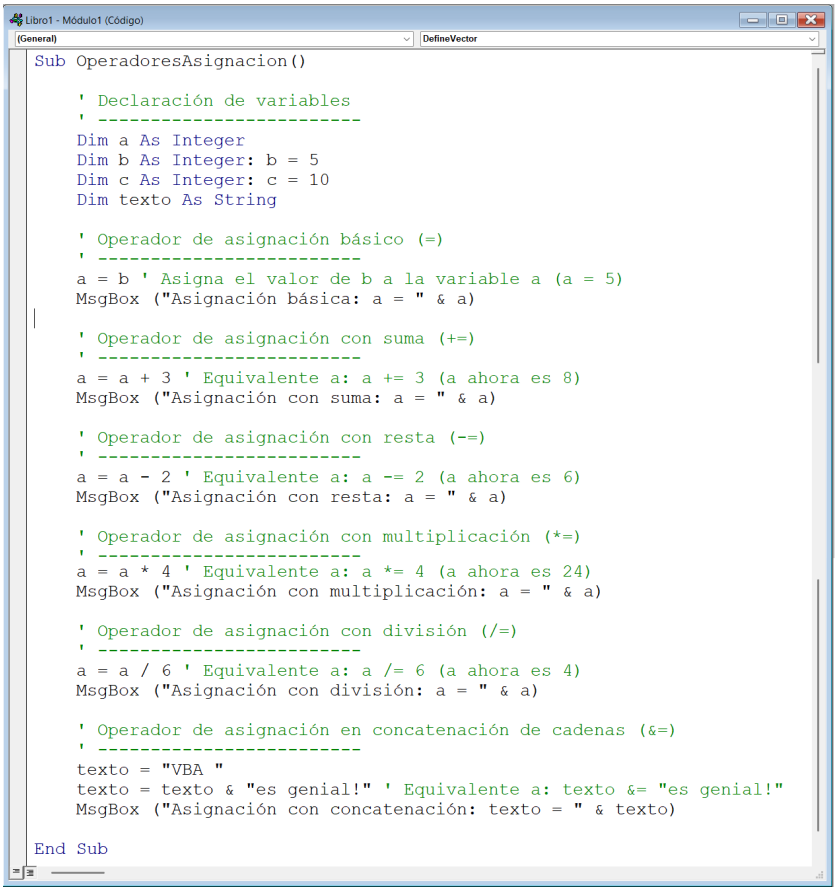
Permiten la combinación de varias expresiones simples para formar una más compleja.

Tabla 7. Operadores lógicos

OPERADOR	UTILIDAD
AND	Combina dos condiciones simples y devuelve un valor verdadero si estas dos combinaciones son verdaderas.
OR	Combina dos condiciones simples y devuelve el valor verdadero cuando una de las dos expresiones es verdadera.
NOT	Se utiliza sobre una sola condición para negar su valor.

3.8.5. Operadores de asignación

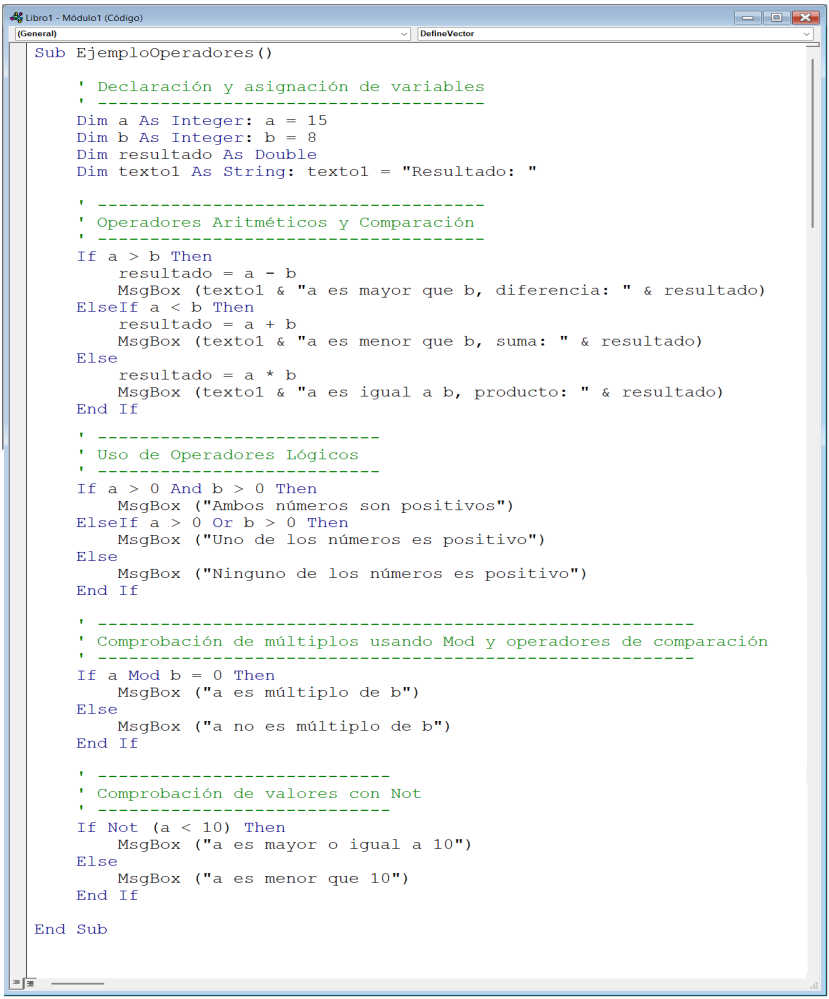
El *operador de asignación* "=" se utilizan para asignar valores a las variables. Ejemplo:




```
Sub OperadoresAsignacion()

    ' Declaración de variables
    ' -------------------------
    Dim a As Integer
    Dim b As Integer: b = 5
    Dim c As Integer: c = 10
    Dim texto As String

    ' Operador de asignación básico (=)
    ' -------------------------
    a = b ' Asigna el valor de b a la variable a (a = 5)
    MsgBox ("Asignación básica: a = " & a)

    ' Operador de asignación con suma (+=)
    ' -------------------------
    a = a + 3 ' Equivalente a: a += 3 (a ahora es 8)
    MsgBox ("Asignación con suma: a = " & a)

    ' Operador de asignación con resta (-=)
    ' -------------------------
    a = a - 2 ' Equivalente a: a -= 2 (a ahora es 6)
    MsgBox ("Asignación con resta: a = " & a)

    ' Operador de asignación con multiplicación (*=)
    ' -------------------------
    a = a * 4 ' Equivalente a: a *= 4 (a ahora es 24)
    MsgBox ("Asignación con multiplicación: a = " & a)

    ' Operador de asignación con división (/=)
    ' -------------------------
    a = a / 6 ' Equivalente a: a /= 6 (a ahora es 4)
    MsgBox ("Asignación con división: a = " & a)

    ' Operador de asignación en concatenación de cadenas (&=)
    ' -------------------------
    texto = "VBA "
    texto = texto & "es genial!" ' Equivalente a: texto &= "es genial!"
    MsgBox ("Asignación con concatenación: texto = " & texto)

End Sub
```

Figura 42. Ejemplo de uso del operador de asignación en VBA.

Ejemplo de uso de operadores:




```
Sub EjemploOperadores()

    ' Declaración y asignación de variables
    ' --------------------------------------
    Dim a As Integer: a = 15
    Dim b As Integer: b = 8
    Dim resultado As Double
    Dim texto1 As String: texto1 = "Resultado: "

    ' --------------------------------------
    ' Operadores Aritméticos y Comparación
    ' --------------------------------------
    If a > b Then
        resultado = a - b
        MsgBox (texto1 & "a es mayor que b, diferencia: " & resultado)
    ElseIf a < b Then
        resultado = a + b
        MsgBox (texto1 & "a es menor que b, suma: " & resultado)
    Else
        resultado = a * b
        MsgBox (texto1 & "a es igual a b, producto: " & resultado)
    End If

    ' ---------------------------
    ' Uso de Operadores Lógicos
    ' ---------------------------
    If a > 0 And b > 0 Then
        MsgBox ("Ambos números son positivos")
    ElseIf a > 0 Or b > 0 Then
        MsgBox ("Uno de los números es positivo")
    Else
        MsgBox ("Ninguno de los números es positivo")
    End If

    ' ------------------------------------------------------------
    ' Comprobación de múltiplos usando Mod y operadores de comparación
    ' ------------------------------------------------------------
    If a Mod b = 0 Then
        MsgBox ("a es múltiplo de b")
    Else
        MsgBox ("a no es múltiplo de b")
    End If

    ' ---------------------------
    ' Comprobación de valores con Not
    ' ---------------------------
    If Not (a < 10) Then
        MsgBox ("a es mayor o igual a 10")
    Else
        MsgBox ("a es menor que 10")
    End If

End Sub
```

Figura 43. Ejemplo de uso de operadores en VBA.

3.9. Sentencia MsgBox

La sentencia MsgBox en VBA se utiliza para mostrar cuadros de mensaje al usuario. Puedes usarla para mostrar información, advertencias o recoger una respuesta.

```
Sintaxis

    MsgBox(mensaje[, botones][, Título][, ayuda, contexto])

    Nota: Lo escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

1. **Mensaje** (obligatorio): el texto que se mostrará en el cuadro de mensaje.
1. **Botones** (Opcional): especifica el tipo de botones y el ícono a mostrar en el cuadro de mensaje.

Tabla 8. Tipos de botones e iconos en el MsgBox

Botones y Estilos Comunes		
Constante	**Valor**	**Descripción**
vbOKOnly	0	Solo botón Aceptar
vbOKCancel	1	Botones Aceptar y Cancelar
vbAbortRetryIgnore	2	Botones Anular, Reintentar, Ignorar
vbYesNoCancel	3	Botones Sí, No y Cancelar
vbYesNo	4	Botones Sí y No
vbRetryCancel	5	Botones Reintentar y Cancelar
vbCritical	16	Ícono de crítica
vbQuestion	32	Ícono de pregunta
vbExclamation	48	Ícono de advertencia
vbInformation	64	Ícono de información

1. **Título** (opcional): el texto que aparecerá en la barra de título del cuadro de mensaje.
1. **Ayuda y contexto** (opcionales): se usan para especificar ayuda contextual (poco común).

La sentencia MsgBox puede devolver valores según el botón presionado por el usuario, como vbYes, vbNo, vbOK, vbCancel, etc.

3.9.1. Ejemplo de uso simple, sin respuesta

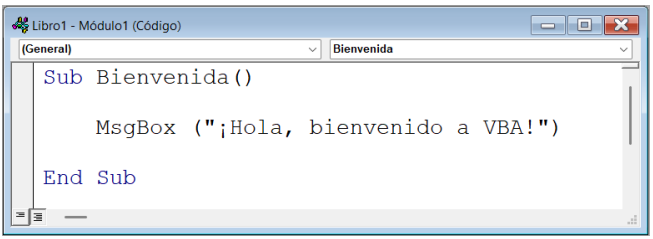


Figura 44. Uso de MsgBox simple.

3.9.2. Ejemplo con títulos y botones

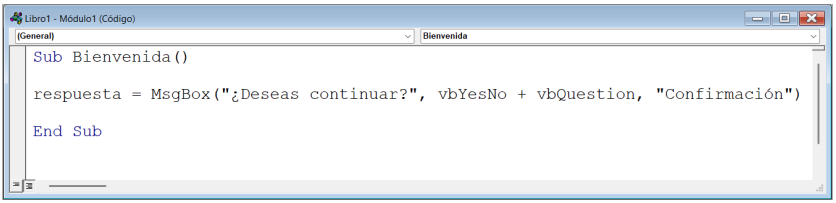


Figura 45. Uso de MsgBox con títulos y botones.

3.9.3. Ejemplo de uso con respuesta del usuario

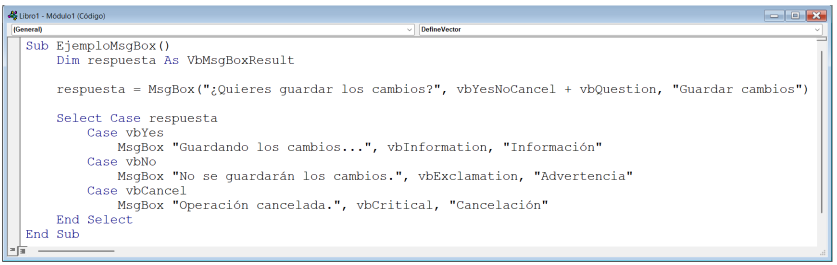


Figura 46. Ejemplo de uso de MsgBox.

3.10. Estructuras de control

Las estructuras de control son fundamentales para controlar el flujo de ejecución del código, permitiendo tomar decisiones, repetir acciones y realizar evaluaciones condicionales. Las principales estructuras de control en VBA se dividen en dos tipos: estructuras condicionales y estructuras de bucle (repetitivas).

3.10.1. Estructuras condicionales

Instrucción If…Next

La instrucción If permite evaluar una condición y la respuesta a esa evaluación puede ser Verdadera o Falsa, en VBA se utiliza para tomar decisiones en el código, es decir, permite ejecutar diferentes bloques de código en función de si una condición es verdadera o falsa. Es una estructura condicional que verifica una expresión lógica, y dependiendo del resultado (verdadero o falso), ejecuta un bloque de código.

```
Sintaxis

        If condición Then
                [instrucciones]
        [ElseIf condición-n Then
                [instrucciones del ElseIf]]
        [Else
                [instrucciones del Else]]
        End If
Nota. Lo escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

- Condición: es de uso obligatorio y representa una expresión numérica o expresión de cadena que se evalúa **Verdadera** o **Falsa**. Si condición es **Null**, la condición es considerada Falsa.
- Instrucciones: es de uso opcional y representa una o más instrucciones; se ejecutan si condición es **Verdadero**.
- condición-n: es de uso opcional y se utiliza igual a condición.
- ElseIf: es de uso opcional y representa una o más instrucciones ejecutadas si el valor de condición-n resulta ser **Verdadera**.
- Else: es de uso opcional y representa una o más instrucciones que se ejecutan si expresión de condición y condición-n resulta no ser **Verdaderas**.
- Los bloques If pueden anidarse, lo que significa que un bloque If puede contener a otros IF dentro de sí.
- Palabras reservadas: **IF**, **Then**, **Else** y **EndIF**

¿Cómo funciona If…EndIf?

La siguiente estructura de la instrucción If es la forma más sencilla de construir:

```
IF condición Then
      [Instrucción(s)
      para Verdadera]
Endif
```

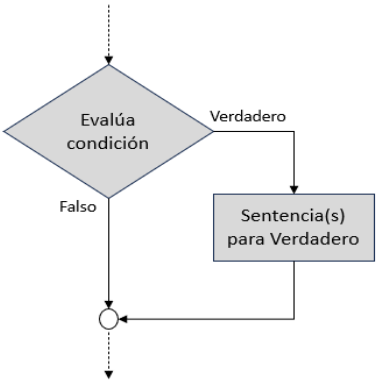


Figura 47. Esquema de un IF.

El flujo de ejecución se desvía si al evaluar la condición la respuesta es verdadera, en ese caso ejecuta la(s) instrucción(s) que están al interior del proceso y luego continúa con la secuencia normal.

Una estructura más completa se muestra a continuación. Esta estructura incluye instrucciones tanto para cuando la condición es verdadera como para cuando es falsa:

```
IF condición Then
     [Instrucción(s)
     para Verdadera]
[Else
      [Instrucción(s)
      para Falsa]]
Endif
```

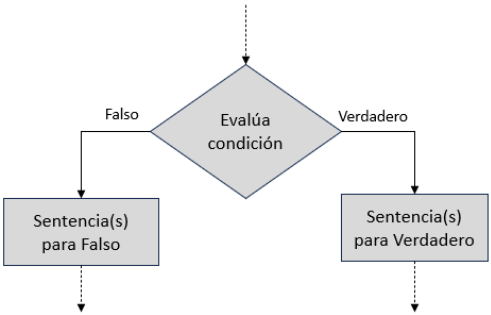


Figura 48. Esquema de un IF con Else.

Al ejecutar un bloque If …Then…Else…Endif (segunda sintaxis), se evalúa la condición, si la condición es **Verdadera**, se ejecutan las instrucciones que siguen a **Then**. Si la condición es **Falsa**, se ejecutan las instrucciones que siguen a **Else**.

Una vez ejecutadas las instrucciones de **Then** o **Else**, la ejecución continúa con la instrucción que sigue a **End If.**

Una tercera estructura es más completa que las anteriores, ya que permite evaluar varias condiciones e incluye instrucciones para el caso en que ninguna de ellas sea verdadera.

```
IF condición-1 Then
      [Instrucciones-1]
[ElseIF condición-2 Then
      [Instrucciones-2]
      [ElseIf condición-3 Then
      [Instrucciones-3]
[Else
      [Instrucciones-4]]
Endif
```

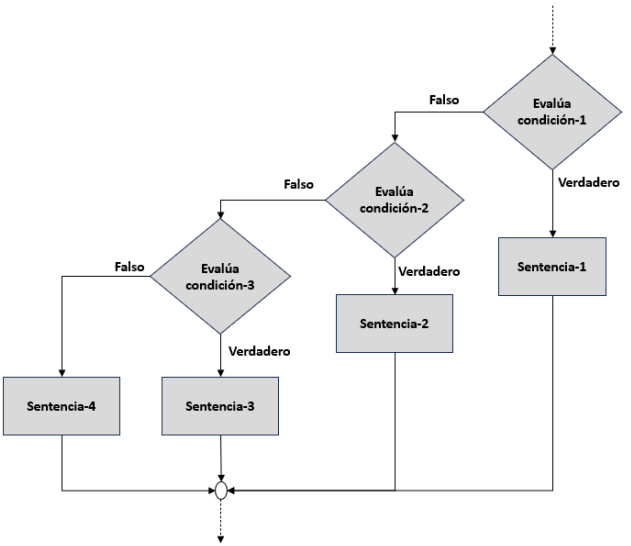


Figura 49. Esquema de un If con ElseIF y Else.

Primero, se evalúa la condición-1. Si esta es **verdadera**, se ejecuta la sentencia-1. Si resulta ser **falsa**, se evalúa la condición-2 ubicada después de la cláusula ElseIf. Si la condición-2 es **verdadera**, se ejecutan la sentencia-2 si no, se evalúa la condición-3 ubicada después de la cláusula ElseIf. Si la condición-3 es **verdadera**, se ejecuta la sentencia-3. Si ninguna de las condiciones anteriores es **verdadera**, se ejecuta la sentencia-4 que siguen a la cláusula Else.

Se pueden incluir tantas cláusulas ElseIf como se desee en un bloque If, pero ninguna debe aparecer después de una cláusula Else.

Sentencia Select Case

La instrucción Select Case se utiliza como una alternativa más organizada y legible al uso de múltiples bloques ElseIf en una estructura If. Cada bloque ElseIf es equivalente a un bloque Case, esto facilita la comparación de una variable o expresión con varios valores. La estructura de la instrucción *Select Case* permite manejar múltiples condiciones de manera más clara.

```
Sintaxis

        Select Case expresión-a-evaluar
                [Case Lista-expresión-n [Sentencia-n]]
                [Case Else [Sentencia-Else]]
        End Select

Nota. Lo escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

- expresión-a-evaluar: es de uso obligatorio y representa cualquier expresión numérica o expresión de cadena a evaluar.
- Lista-expresión-n: es de uso opcional y es necesario cuando aparece un Case. La lista de expresiones debe estar delimitada de alguna de las siguientes maneras: expresión, expresión To expresión, o una expresión de comparación con Is.
- La palabra clave To indica un rango de valores de menor a mayor, mientras que la palabra clave Is se utiliza con operadores de comparación para definir una comparación.
- Sentencia: es de uso opcional y representa una o más instrucciones que se ejecutan si expresión-*a-evaluar* coincide con cualquier parte de Lista-*expresión-n*.
- Las instrucciones **Select Case** pueden anidarse. Cada instrucción **Select Case** anidada debe tener una correspondiente instrucción **End Select**.
- Palabras reservadas: **Select**, **Case, Else**.

¿Cómo funciona Select Case?

Si la expresión a evaluar coincide con cualquier expresión en la lista de expresiones de una cláusula **Case**, se ejecutan las instrucciones correspondientes hasta la siguiente cláusula **Case** o, en el caso de la última cláusula, hasta **End Select**.

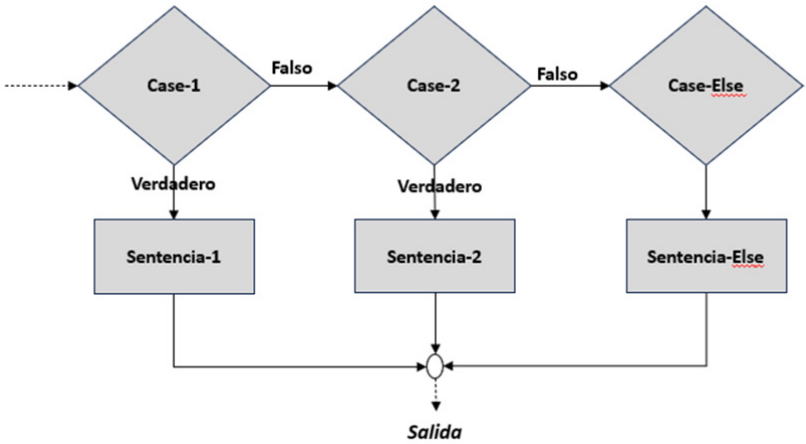


Figura 50. Esquema de un Select Case.

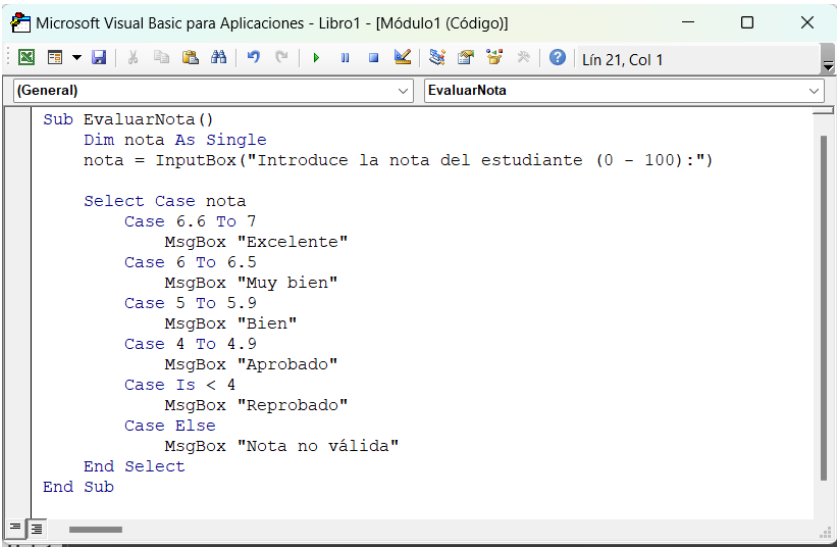


Figura 51. Ejemplo de uso de un Select Case.

Explicación:

1. Entrada de datos: se solicita al usuario que ingrese la nota del estudiante.
2. La nota ingresada se evalúa usando la instrucción Select Case.
3. Si la nota estás entre 6.6 y 7, muestra "Excelente".
4. Si la nota está entre 6 y 6.5, muestra "Muy bien".
5. Si la nota está entre 5 y 5.9, muestra "Bien".
6. Si la nota está entre 4 y 4.9, muestra "Aprobado".
7. Si la nota es inferior a 4, muestra "Reprobado".
8. Case Else se utiliza para manejar cualquier entrada que no corresponda a las condiciones anteriores (como una nota fuera del rango 0-7).

3.10.2. Estructuras de bucle

Las estructuras de bucle en VBA (y en los lenguajes de programación) se utilizan para ejecutar un bloque de código de forma repetida, ya sea un número específico de veces o hasta que se cumpla una condición. Este tipo de bucle pueden contener una o varias tareas a realizar y son muy útiles para automatizar procesos repetitivos o procesar grandes volúmenes de datos.

Figura 52. Izquierda: actividad repetitiva hasta cumplir una condición. Derecha: varias actividades repetitivas hasta cumplir una condición.

En general, los procesos artesanales son únicos, mientras que los procesos industriales tienden a ser repetitivos, ya que están diseñados para mantener la calidad y minimizar los costos de los procesos productivos.

Por ejemplo, la Figura 53 muestra el proceso de embotellado de una bebida, que incluye el llenado, sellado y etiquetado en una línea de producción automatizada. Este proceso se repite de manera constante para cada botella, lo que puede compararse con un bucle en programación, donde una secuencia de acciones se ejecuta múltiples veces de forma continua.

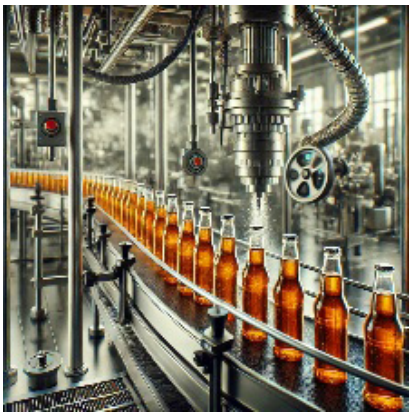


Figura 53. Representación de un proceso repetitivo de una planta de embotellado de bebidas. Fuente: propia desarrollado con Open AI.

Todos los procesos tipo bucle tiene características comunes:

- Inicio claro: el proceso debe comenzar siempre desde un estado inicial definido antes de entrar en el ciclo de repetición.
- Repetible: el proceso debe poder ejecutarse de manera repetida y consistente, con los mismos resultados cada vez. Esto significa que las acciones dentro del bucle deben estar bien definidas y ser ejecutadas en un orden específico.
- Consistencia: las mismas reglas o instrucciones deben aplicarse en cada iteración del bucle. Si hay variaciones en la ejecución del proceso, debe ser predefinido cuándo y cómo ocurren.
- Incremento o cambio por iteración: cada vez que se repite el bucle, debe haber algún tipo de progreso o modificación en las variables del proceso, que acerque al sistema a la condición de salida. Si no hay cambios, el bucle puede volverse infinito.
- Control de errores: durante la ejecución del bucle, es importante monitorear posibles errores o situaciones imprevistas que puedan ocurrir. El proceso debe incluir mecanismos para manejar errores y corregir el flujo si algo sale mal.

Ejemplo. En un proceso de embotellado automatizado:

- **Condición de inicio**: el proceso comienza cuando se presiona el botón START.
- **Incremento**: en cada iteración, se procesa una nueva botella.
- **Repetible**: cada botella se llena y se sella de manera similar en cada ciclo del proceso.
- **Condición de salida**: el proceso finaliza una vez que se han llenado y sellado todas las botellas.
- **Control de errores**: existe un botón EXIT que permite detener el proceso en caso de error o accidente.

Sentencia For…Next (Bucle)

El bucle For...Next en VBA se utiliza para ejecutar un bloque de código un número determinado de veces. Es una de las estructuras de control más utilizadas, pero necesita que se conozca de antemano, cuántas veces se necesita repetir una acción.

Su sintaxis es la siguiente:

```
Sintaxis
        For contador [As tipoDeDato] = inicio To fin [Step incremento]
            [instrucciones]
            [Continue For]
            [Exit For]
        Next [Contador]

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

- Contador: es de uso obligatorio e identifica la variable numérica para el control del bucle.
- TipoDeDato: es de uso opcional y representa el tipo de datos del contador.
- Inicio: es de uso obligatorio y representa una expresión numérica que corresponde al valor inicial del contador.
- Fin: es de uso obligatorio y representa una expresión numérica correspondiente al valor final del contador.

- Incremento: es de uso opcional y representa una expresión numérica que corresponde a la cantidad en que contador se incrementa en cada iteración del bucle.
- Instrucciones: es de uso opcional y corresponde a una o más instrucciones que se ejecutan en cada iteración del bucle FOR…Next
- Continue for: es de uso opcional e indica que cuando se ejecute esa instrucción se transfiere el control a la siguiente iteración del bucle.
- Exit for: es de uso opcional e indica que el control se transfiere a la siguiente instrucción fuera del bucle, finaliza el bucle.
- Next: es de uso obligatorio y marca el final del bucle FOR.

¿Cómo funciona For…Next?

Cuando se inicia un bucle For...Next, se evalúan los valores de inicio, fin e incremento. Luego, el valor de inicio se asigna a contador. Antes de que se ejecute el bloque de instrucciones, se compara el valor de contador con el valor de fin. Si contador es mayor que el valor de fin (o menor, en caso de que incremento sea negativo), el bucle For finaliza y el control pasa a la instrucción siguiente al FOR… Next. De lo contrario, se ejecuta una siguiente iteración del bloque de instrucciones. Cada vez que se alcanza la instrucción Next, el contador se incrementa según el valor definido en incremento. El control regresa a la instrucción FOR y la siguiente iteración continúa.

En cada iteración, el contador se vuelve a comparar con el valor de fin, y según el resultado, se ejecuta el bloque de instrucciones o sale del bucle. Este proceso continúa hasta que contador supera el valor límite de fin o se encuentra una instrucción Exit For. El bucle no se detendrá hasta que contador haya superado dicho el límite.

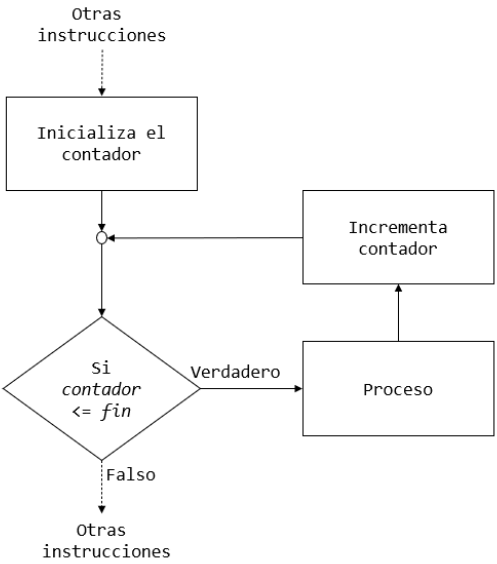


Figura 54. Esquema del funcionamiento de una sentencia FOR…Next.

La condición para determinar si debe ejecutarse la siguiente iteración depende del valor del incremento: si es positivo, la comparación será contador <= fin, y si es negativo, será contador >= fin.

Modificar el valor del contador dentro del bucle puede dificultar la lectura y depuración del código, por lo que es recomendable evitarlo para mantener la claridad y previsibilidad del bucle.

Ejemplos:

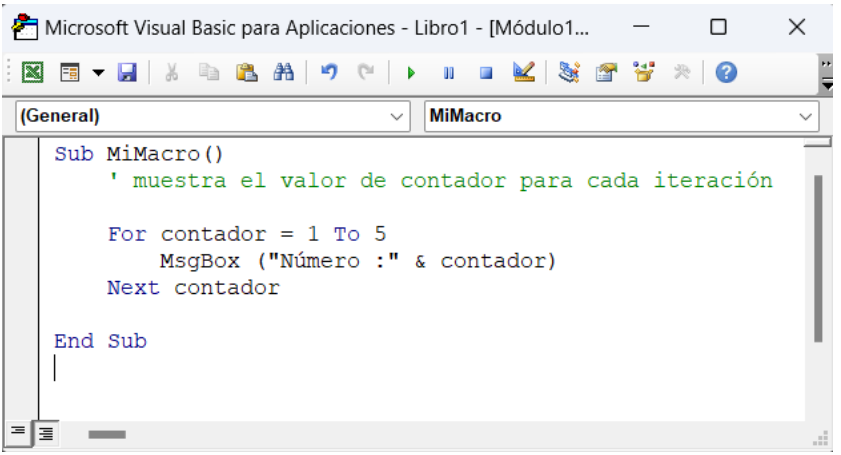


Figura 55. Ejemplo del uso de una sentencia FOR...NEXT.

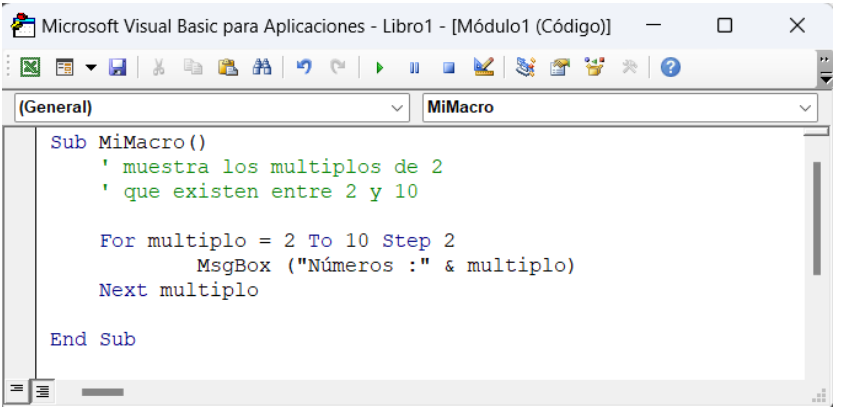


Figura 56. Ejemplo del uso de una sentencia FOR...NEXT.

Sentencia For Each…Next

La estructura For Each se utiliza para recorrer o iterar a través de todos los elementos de una colección u objeto, como rangos de celdas, hojas de cálculo, libros de trabajo, o cualquier otra colección de objetos. Es especialmente útil cuando no se conoce de antemano la cantidad de elementos que contiene la colección.

Descripción de la sintaxis:

```
Sintaxis
        For Each elemento [As tipoDeDato] In Colección
            [instrucciones]
            [Continue For]
            [Exit For]
        Next [elemento]

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

- Elemento: es de uso obligatorio y corresponde a una variable y se usa para iterar por los elementos de la colección.
- TipoDeDato: es de uso opcional y representa el Tipo de datos de Elemento.
- Colección: es de uso obligatorio y representa la variable iterable tipo objeto o tipo de colección.
- Instrucciones: es de uso opcional y representa una o varias instrucciones entre For Each y Next que se ejecutan para cada elemento del grupo.
- Continue For: es de uso opcional y transfiere el control al comienzo del bucle For Each.
- Exit For: es de uso opciona y permite transferir el control fuera del bucle For Each, a la siguiente instrucción fuera del bucle.
- Next: es de uso obligatorio y marca el final de la definición del bucle For Each.

¿Cómo funciona For Each…Next?

Cuando se ejecuta una instrucción For Each...Next, Visual Basic evalúa la colección antes de iniciar el bucle. Si la colección contiene elementos, el bucle comienza; de lo contrario, el control pasa directamente a la siguiente instrucción fuera del bucle.

En cada iteración, el bucle toma un elemento de la colección y lo asigna a la variable elemento, para que este sea procesado. Una vez que todos los elementos de la colección han sido asignados y procesados, el bucle For Each finaliza, y el control pasa a la instrucción que sigue después de Next.

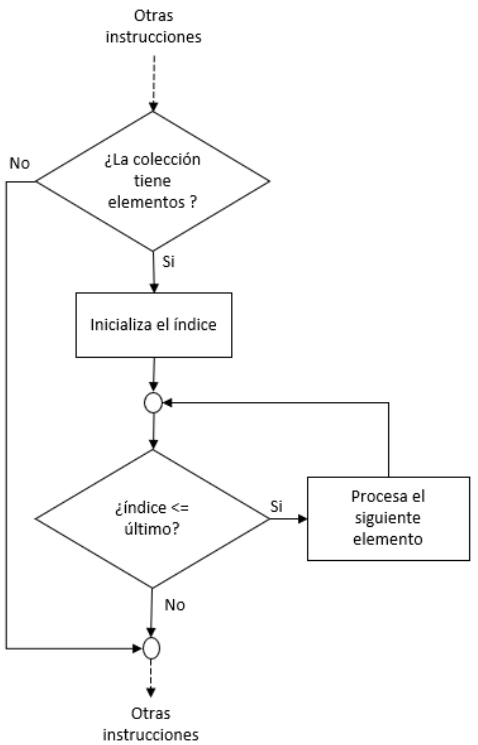


Figura 57. Esquema del funcionamiento de la sentencia For Each…Next.

Ejemplos:

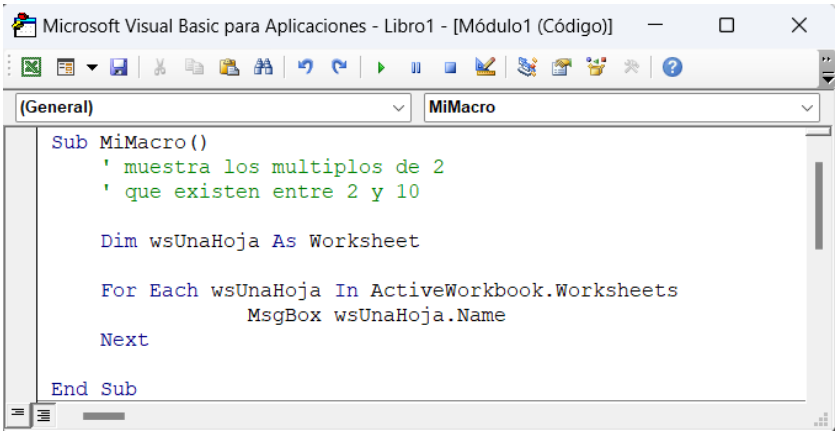


Figura 58. Ejemplo del uso de la sentencia For Each…Next.

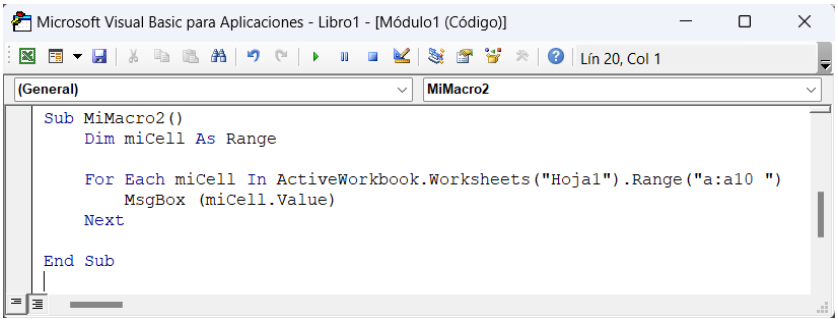


Figura 59. Ejemplo de uso de la sentencia For Each...Next.

Sentencia While…Wend

El bucle While...Wend ejecuta un conjunto de instrucciones mientras la condición sea verdadera. Cuando la condición se evalúa como falsa, el bucle finaliza. En otras palabras, repite las instrucciones mientras la condición sea verdadera.

```
SINTAXIS
        While Condición
            [instrucciones]
        Wend

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

- Condición: es de uso obligatorio y corresponde a una expresión numérica o expresión de cadena que sea **Verdadera** o **Falsa**. Si condición es Null, se considera **Falsa**.
- Instrucciones: es de uso opcional y representa una o más instrucciones que se ejecutan mientras la condición sea **Verdadera**.

¿Cómo funciona While…Wend?

Mientras la condición sea verdadera, se ejecutan todas las instrucciones situadas entre While y Wend. Este proceso se repite hasta que la condición se vuelva falsa, momento en el cual la ejecución continúa con la instrucción que sigue inmediatamente a Wend.

Los bucles While...Wend pueden anidarse en cualquier nivel y cada instrucción Wend corresponderá al While más cercano.

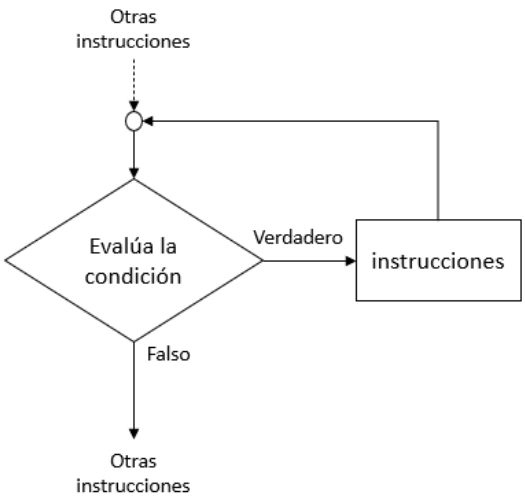


Figura 60. Esquema del funcionamiento de la sentencia While ... Wend.

Ejemplo:

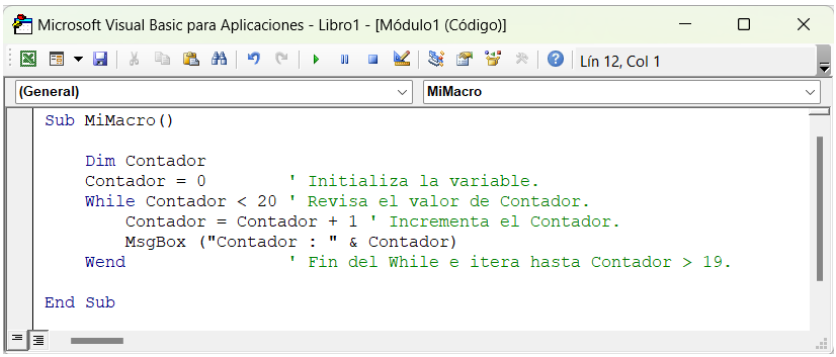


Figura 61. Ejemplo de uso de la sentencia While...Wend.

Sentencia Do While…Loop

La lógica de este tipo de bucles es exactamente igual a la estructura While… Wend.

Descripción de la sintaxis:

```
Sintaxis
        Do While Condición
            [instrucciones]
        Loop

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

- Condición: es de uso obligatorio y corresponde a una expresión numérica o expresión de cadena que sea **Verdadera** o **Falsa**. Si condición es Null, se considera **Falsa**.
- Instrucciones: es de uso opcional y representa una o más instrucciones que se ejecutan mientras la condición sea **Verdadera**.

¿Cómo funciona Do While…Loop?

Mientras la condición sea verdadera, se ejecutan todas las instrucciones situadas entre Do While y Loop. Este proceso se repite hasta que la condición se vuelva falsa, momento en el cual la ejecución continúa con la instrucción que sigue inmediatamente a Loop.

Los bucles Do While...Loop pueden anidarse en cualquier nivel, y cada instrucción Loop corresponderá al Do While más cercano.

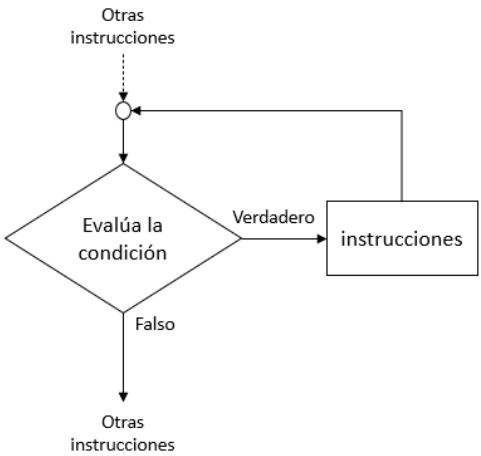


Figura 62. Esquema de funcionamiento del bucle Do While…Loop.

Ejemplo:

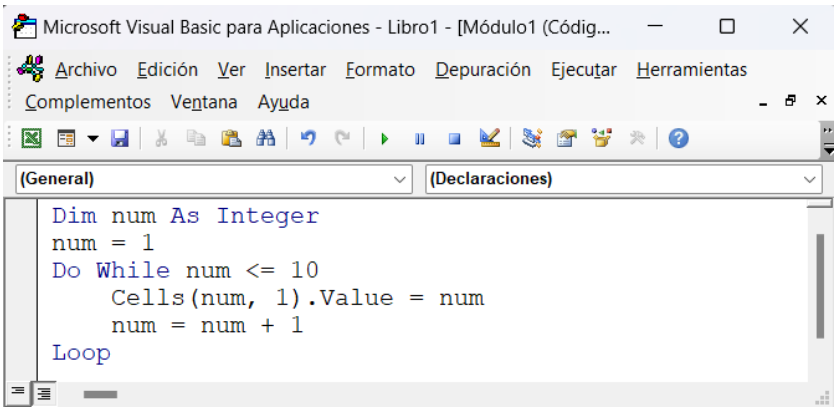


Figura 63. Ejemplo de uso de la sentencia Do While ... Loop.

Sentencia Do …Loop While

El bucle Do …Loop While ejecuta un conjunto de instrucciones mientras la condición sea verdadera. Cuando la condición se evalúa como falsa, el bucle finaliza. En otras palabras, repite las instrucciones mientras la condición sea verdadera.

```
Sintaxis
        Do
            [instrucciones]
        Loop While condición

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

- Condición: es de uso obligatorio y corresponde a una expresión numérica o expresión de cadena que sea **Verdadera** o **Falsa**. Si condición es Null, se considera **Falsa**.
- Instrucciones: es de uso opcional y representa una o más instrucciones que se ejecutan mientras la condición sea **Verdadera**.

¿Cómo funciona Do … Loop While?

En este caso la condición se evalúa después de ejecutar todas las instrucciones que están al interior de Do…Loop While. Se asegura que las instrucciones al interior de bucle se ejecuten, al menos, una vez. Al final de la ejecución se evalúa la condición y si resulta ser verdadera se vuelve a ejecutar el conjungto de instrucciones del bucle.

Este proceso se repite hasta que la condición se vuelva falsa, momento en el cual la ejecución continúa con la instrucción que sigue inmediatamente a Loop.

Los bucles Do Loop While pueden anidarse en cualquier nivel, y cada instrucción Loop corresponderá al Do más cercano.

Figura 64. Esquema de funcionamiento del bucle Do...Loop While.

Ejemplo:

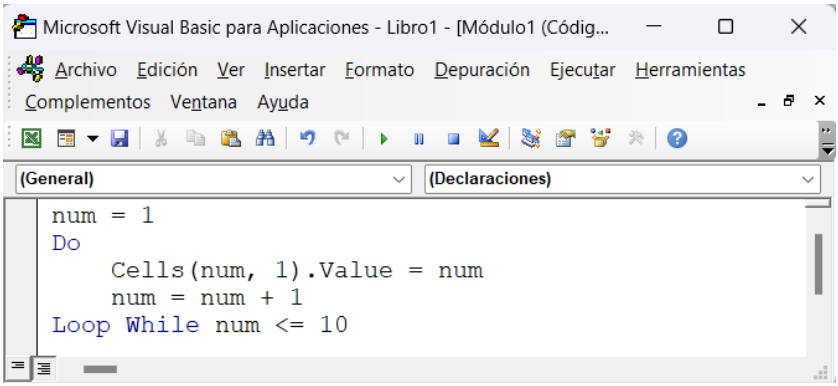


Figura 65. Ejemplo de uso de la sentencia Do …Loop While.

Sentencia Do Until…Loop

Se ejecutará el conjunto de instrucciones mientras la condición especificada sea Falsa. Deja de ejecutarse cuando la condición es verdadera.

```
SINTAXIS
        Do Until Condición
            [instrucciones]
        Loop

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

- Condición: es de uso obligatorio y corresponde a una expresión numérica o expresión de cadena que sea **Verdadera** o **Falsa**. Si *condición* es Null, se considera **Falsa**.
- Instrucciones: es de uso opcional y representa una o más instrucciones que se ejecutan mientras la condición sea **Falsa**.

¿Cómo funciona Do Until … Loop?

Mientras la condición sea Falso, se ejecutan todas las instrucciones situadas entre Do Until y Loop. Este proceso se repite hasta que la condición se vuelva Verdadera, momento en el cual la ejecución continúa con la instrucción que sigue inmediatamente a Loop.

Los bucles Do Until …Loop pueden anidarse en cualquier nivel, y cada instrucción Loop corresponderá al Do Until más cercano.

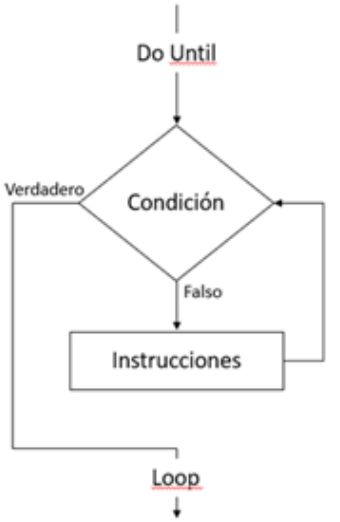


Figura 66. Esquema de funcionamiento del bucle Do Until ...Loop.

Ejemplo:

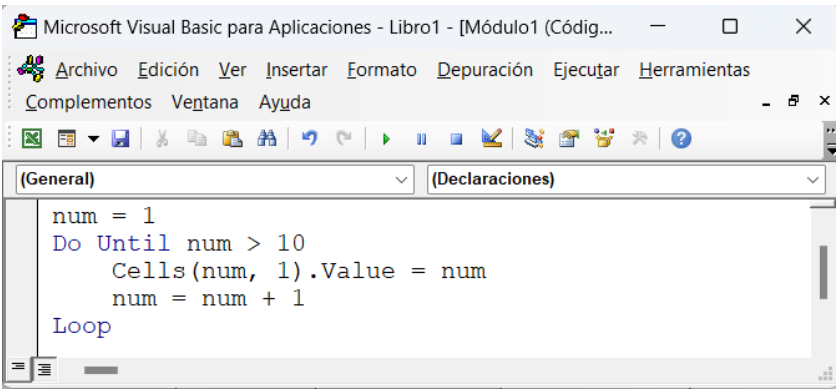


Figura 67. Ejemplo de uso de la sentencia Do Until …Loop.

Sentencia Do…Loop Until

En este caso, el bloque de código se ejecutará al menos una vez antes de evaluar la condición.

```
Sintaxis
        Do Loop
            [instrucciones]
        Until Condición

Nota. Todo lo que está escrito entre paréntesis de corchete es opcional.
```

Descripción de la sintaxis:

- Instrucciones*:* es de uso opcional y representa una o más instrucciones que se ejecutan mientras la condición sea **Falsa**. Las instrucciones se ejecutan a lo menos una vez.
- Condición: es de uso obligatorio y corresponde a una expresión numérica o expresión de cadena que sea **Verdadera** o **Falsa**. Si condición no existe el ciclo se ejecuta infinitamente porque no tiene una condición que se evalúe como **Verdadera** y si la condición es NULL, el ciclo continuará ejecutándose, ya que NULL no cumple con el criterio de ser "verdadero" para salir del ciclo.

¿Cómo funciona Do Loop …Until?

Mientras la condición sea Falso, se ejecutan todas las instrucciones situadas entre Do Loop … Until. Este proceso se repite hasta que la condición se vuelva Verdadera, momento en el cual la ejecución continúa con la instrucción que sigue inmediatamente a Until.

Los bucles Do Loop … Until pueden anidarse en cualquier nivel, y cada instrucción Until corresponderá al Do Loop más cercano.

Figura 68. Esquema del funcionamiento del bucle Do Until ...Loop.

Ejemplo de uso:

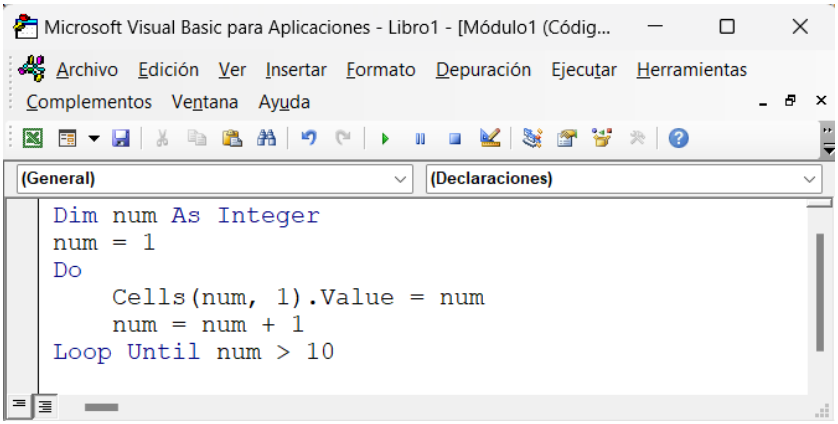


Figura 69. Ejemplo de uso de la sentencia Do Until …Loop.

3.10.3. Sentencia Exit Do

La sentencia Exit Do se utiliza para salir de un ciclo inmediatamente, sin evaluar la condición del ciclo. Esta sentencia es útil cuando se quiere terminar el ciclo de manera anticipada, por ejemplo, al cumplir con una condición específica dentro del bloque de código. Una vez que se ejecuta Exit Do, el control del programa se transfiere a la línea de código inmediatamente después del ciclo.

Aunque algunos estiman que este es una sentencia que ensucia la lógica porque el bucle se aborta de forma forzada y ellos estiman que el bucle debería salir por medio de la condición, otras estiman que es válido usar esta instrucción porque para ello existe, para ser usada. Esto es una discusión filosófica y que no ahondaremos en este libro.

¿Cuándo se justifica usar la sentencia Exit Do?

- Para evitar iteraciones innecesarias cuando se ha alcanzado una meta.
- Para manejar situaciones excepcionales o errores específicos dentro del ciclo.
- Para optimizar el código, saliendo del ciclo en cuanto se obtenga el resultado deseado

Ejemplo de uso:

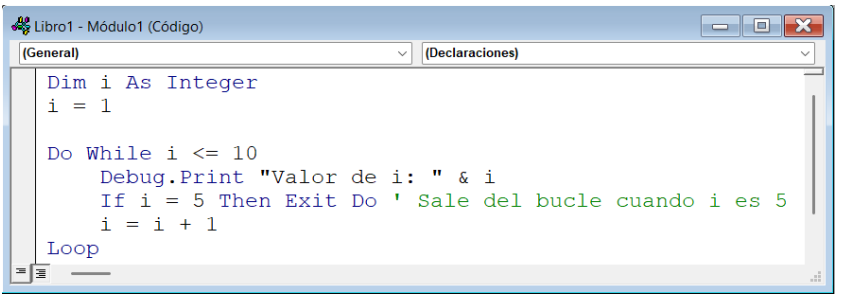


Figura 70. Funcionamiento de la sentencia Exit Do.

3.10.4. Manejo de errores (estructura condicional)

Tener la posibilidad de administrar los errores es fundamental para gestionar situaciones inesperadas durante la ejecución de un programa, como divisiones por cero, acceso a archivos inexistentes o errores de tipo de datos. Aunque no es una estructura de control propiamente tal, permite controlar el flujo del código ante errores, evitando que el programa se detenga abruptamente.

En VBA, la instrucción más utilizada para la gestión de errores es On Error, la cual permite detectar y manejar errores de manera controlada. Su uso facilita la ejecución del código en presencia de fallos, redirigiendo el flujo del programa a un bloque específico en lugar de interrumpir la ejecución con un mensaje de error predeterminado. Esto resulta especialmente útil para evitar bloqueos, proporcionar soluciones alternativas y generar mensajes personalizados que mejoren la experiencia del usuario.

Existen diferentes formas de gestionar errores en VBA:

Sentencia On Error GoTo [Etiqueta]

```
SINTAXIS
        On error GoTo etiquetaError
```

La sentencia On Error GoTo [EtiquetaError] permite redirigir el flujo del programa a una sección específica cuando se produce un error en tiempo de ejecución. Su principal utilidad es detectar y gestionar errores de manera controlada, evitando interrupciones inesperadas y permitiendo definir una respuesta adecuada para cada situación.

Ejemplo de uso de la sentencia On Error:

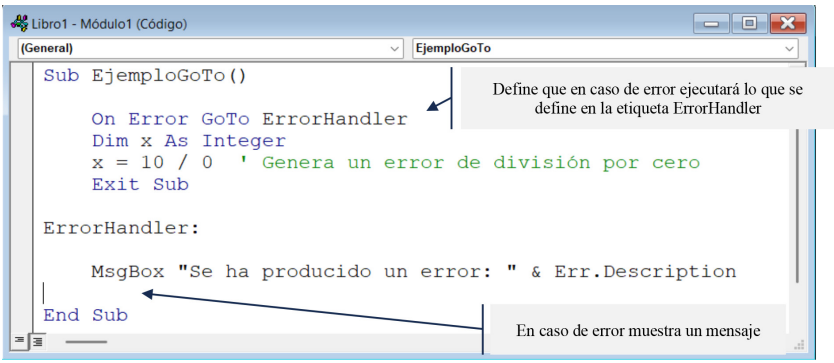


Figura 71. Ejemplo de uso de la sentencia On Error.

Sentencia On Error Resume Next

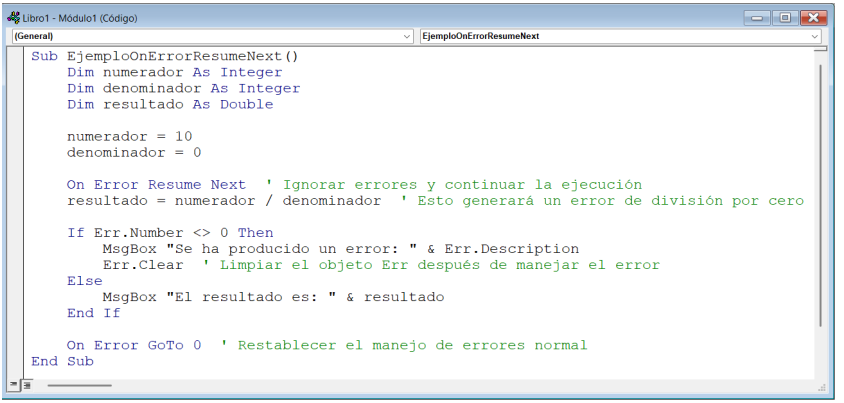
```
Sintaxis
        On error Resume Next
```

¿Cómo funciona la sentencia On Error Resume Next?

La instrucción On Error Resume Next permite que el programa continúe ejecutándose tras un error, avanzando a la siguiente línea de código sin interrumpirse. En lugar de desviar el flujo a una rutina de manejo de errores, el error se puede gestionar en el mismo lugar donde ocurre. Sin embargo, esta instrucción deja de estar activa al llamar a otro procedimiento, por lo que es necesario incluirla en cada procedimiento donde se desee aplicar el control de errores.

- On Error Resume Next no resuelve el error, solo ignora el mensaje de error y permite que el programa continúe a pesar del error que se ha generado.

Ejemplo de uso de la sentencia On Error:




```
Sub EjemploOnErrorResumeNext()
    Dim numerador As Integer
    Dim denominador As Integer
    Dim resultado As Double

    numerador = 10
    denominador = 0

    On Error Resume Next  ' Ignorar errores y continuar la ejecución
    resultado = numerador / denominador  ' Esto generará un error de división por cero

    If Err.Number <> 0 Then
        MsgBox "Se ha producido un error: " & Err.Description
        Err.Clear  ' Limpiar el objeto Err después de manejar el error
    Else
        MsgBox "El resultado es: " & resultado
    End If

    On Error GoTo 0  ' Restablecer el manejo de errores normal
End Sub
```

Figura 72. Ejemplo de uso de la sentencia On Error Resume.

Explicación detallada del código

Declaración de variables:

- Numerador y denominador son variables enteras (Integer).
- Resultado es una variable de tipo Double para almacenar el resultado de la división.

Definición de valores iniciales:

- Numerador se establece en 10.
- Denominador se establece en 0 (esto causará un error cuando intentemos dividir).

Activación del manejo de errores:

- On Error Resume Next le indica a VBA que continúe con la ejecución del código incluso si se produce un error.
- La siguiente línea intenta calcular resultado = numerador / denominador. Como denominador = 0, esto genera un error de "división por cero", pero gracias a On Error Resume Next, el programa no se detiene.

Detección y manejo del error:
- If Err.Number <> 0 Then verifica si se ha producido un error.
- Si hay un error, se muestra un mensaje con MsgBox "Se ha producido un error: " & Err.Description, que indica qué tipo de error ocurrió.
- Err.Clear se usa para limpiar el estado de error después de haberlo gestionado.

Manejo del resultado:
- Si no hubo error, se muestra el resultado con MsgBox "El resultado es: " & resultado.

Restablecimiento del manejo de errores:
- On Error GoTo 0 desactiva On Error Resume Next, restaurando el comportamiento normal de VBA.
- A partir de aquí, cualquier error detendrá el programa nuevamente.

Sentencia On Error GoTo 0

```
Sintaxis
        On error Goto 0
```

Esta sentencia desactiva todo manejo de errores establecido anteriormente. Cuando se encuentra, VBA vuelve a su comportamiento predeterminado, es decir, si ocurre un error se muestra un mensaje de error y se detiene la ejecución.

¿Cómo funciona la sentencia On Error GoTo 0?

La instrucción On Error GoTo 0 en VBA desactiva cualquier manejo personalizado de errores, restaurando el comportamiento predeterminado. Esto significa que, si ocurre un error, el programa se detendrá y mostrará un mensaje de error, sin omitirlo ni tratarlo de forma especial. En resumen, restablece el manejo estándar de errores en VBA.

Ejemplo:

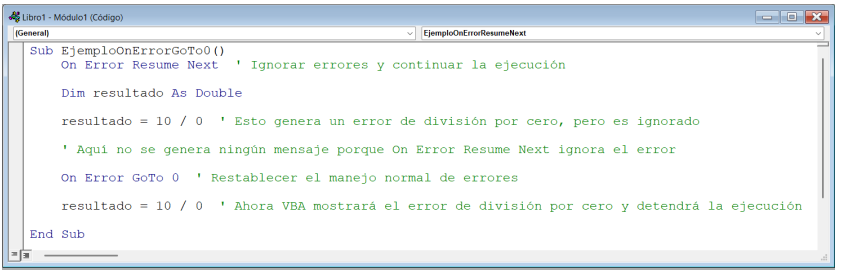


Figura 73. Ejemplo de uso de la sentencia On Error Goto 0.

Explicación del código

Activación de On Error Resume Next:

- Permite que el código continúe ejecutándose incluso si ocurre un error.
- Se genera una división por cero (10 / 0), lo que normalmente causaría que VBA detuviera la ejecución.
- Sin embargo, debido a On Error Resume Next, el error es ignorado y el programa sigue funcionando sin interrupciones.

Desactivación del manejo de errores con On Error GoTo 0:

- Se restablece el comportamiento predeterminado de VBA.
- Ahora, si ocurre otro error (como la división por cero nuevamente), el programa se detendrá y mostrará un mensaje de error.

Ejemplo práctico:

- La primera división por cero no genera ninguna alerta debido a On Error Resume Next.
- La segunda división por cero, después de On Error GoTo 0, sí detendrá la ejecución mostrando el error.

Objeto Err

El objeto **Err** es una herramienta fundamental en VBA para la gestión de errores en tiempo de ejecución. Permite identificar el tipo de error ocurrido, obtener información detallada sobre su causa y aplicar estrategias adecuadas para manejarlo.

A través de sus propiedades y métodos, **Err** proporciona datos como el número del error, su descripción y la fuente que lo generó, lo que facilita la depuración y el diseño de respuestas específicas dentro del código.

Propiedades principales del objeto Err

- Número de error (Err.Number): indica el código numérico del error ocurrido. Si no hay error, devuelve 0.
- Descripción (Err.Description): proporciona un mensaje explicativo sobre la causa del error.
- Origen (Err.Source): identifica la fuente del error, como el nombre del módulo o el objeto que lo generó.
- Método (Err.Clear): permite limpiar la información del error después de haberlo manejado, asegurando que Err esté listo para detectar futuros errores.
- Método (Err.Raise): permite generar errores de manera intencional dentro del código, útil para pruebas o para simular condiciones específicas.

¿Cómo funciona el Objeto Err?

Cuando ocurre un error en el código, VBA almacena información detallada en el objeto **Err**. Este objeto contiene datos clave, como el número de error, su descripción y la fuente que lo generó. Su uso es fundamental para gestionar errores de manera controlada, permitiendo al programador identificar la causa del problema y tomar medidas adecuadas, en lugar de que el programa se detenga inesperadamente.

Ejemplo: el siguiente código muestra cómo capturar y mostrar detalles de un error

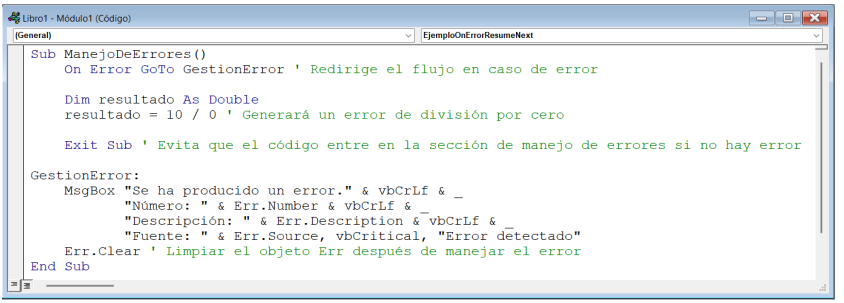


Figura 74. Ejemplo de uso del objeto Err.

Explicación del código

- On Error GoTo GestionError: activa el control de errores y redirige la ejecución a la sección GestionError si ocurre un problema.
- Intento de división por cero (resultado = 10 / 0): esto generará un error, ya que no es posible dividir entre cero.
- Exit Sub: si no ocurre un error, el código sale antes de entrar en la sección de manejo de errores.
- GestionError: muestra un mensaje con la información almacenada en el objeto Err.
- Err.Clear: borra los detalles del error para evitar conflictos con futuras ejecuciones.

Capítulo 4
Macros

Una Macro es una secuencia de instrucciones o acciones programadas que automatizan tareas repetitivas en aplicaciones como Excel, Word y otras del entorno de Microsoft Office. Las Macros se crean utilizando el Editor de Visual Basic (VBE) y el lenguaje Visual Basic for Applications (VBA). Son especialmente útiles para ahorrar tiempo, reducir errores y mejorar la eficiencia en procesos manuales.

Objetivos de aprendizaje

1. Reconocer los usos de Macros en la automatización de tareas contables y en el manejo de grandes volúmenes de datos.
2. Diseñar y programar Macros en VBA que automaticen procesos contables complejos, como el cálculo de amortizaciones, la consolidación de datos financieros o la generación de reportes personalizados, con estructuras de control dinámicas que se adapten a las necesidades de un proceso contable específico.
3. Elaborar soluciones personalizadas en VBA que combinen múltiples funciones de Excel para satisfacer necesidades específicas del flujo de trabajo contable, integrando cálculos financieros avanzados y generando reportes automáticos.

Resultados de aprendizaje

Identificar cómo las Macros automatizan tareas contables y el manejo de grandes volúmenes de datos.
Crear Macros en VBA para automatizar procesos contables complejos, tales como cálculos de amortizaciones, consolidación de datos financieros y generación de reportes personalizados, usando estructuras de control adaptativas.
Crear Macros que combinen múltiples funciones de Excel para cubrir necesidades específicas del flujo de trabajo contable, incluyendo cálculos financieros avanzados y generación automática de reportes.

4.1. Introducción a las Macros

¿Para qué sirve una Macro?

- Automatización de tareas repetitivas: las Macros permiten ejecutar, con un solo clic, tareas que suelen requerir mucho tiempo y esfuerzo cuando se realizan manualmente. Esto resulta especialmente útil para tareas contables repetitivas, como la limpieza y organización de datos financieros.
- Procesamiento de grandes volúmenes de datos: las Macros facilitan el análisis y procesamiento rápido de grandes cantidades de datos, lo cual es ideal para realizar cálculos contables complejos, generar reportes financieros detallados y resumir datos. En contabilidad, esto puede incluir tareas como la consolidación de balances o la actualización de datos de ventas.
- Personalización de procesos: con VBA, es posible crear Macros que adapten Excel y otras aplicaciones de Office a necesidades contables específicas, añadiendo funciones personalizadas que Excel no incluye de forma predeterminada. Esto puede ser útil para crear reportes financieros ajustados a la normativa contable o para implementar cálculos personalizados, como tasas de interés o amortización.
- Reducción de errores humanos: automatizar procesos contables manuales ayuda a minimizar los errores que suelen ocurrir al reali-

zar tareas repetitivas, como ingresar transacciones o aplicar fórmulas en celdas. Las Macros siguen instrucciones precisas cada vez que se ejecutan, lo que mejora la confiabilidad de los datos.
- Ahorro de tiempo y aumento de productividad: al reducir el tiempo que los profesionales dedican a tareas repetitivas y al simplificar procesos contables complejos, las Macros permiten que los usuarios se concentren en actividades estratégicas, como el análisis financiero y la toma de decisiones.

En resumen, las principales ventajas de utilizar macros en contabilidad son:

- Eficiencia: ejecutan tareas de forma rápida, ahorrando tiempo valioso.
- Precisión: reducen los errores humanos en tareas repetitivas, mejorando la confiabilidad de los datos.
- Flexibilidad: permiten adaptar las aplicaciones de Office a necesidades contables específicas.
- Productividad: liberan tiempo para actividades de mayor valor, como la planificación y el análisis financiero.

Las Macros son herramientas esenciales en el ámbito contable, donde el manejo preciso de datos y la optimización del tiempo son fundamentales.

4.2. Subrutinas o procedimientos

Las subrutinas o procedimientos (Sub) son un conjunto de instrucciones escritas en VBA y que ejecutan tareas específicas. Sus principales características son:

- **Las subrutinas (Sub)**: no devuelve un valor.
- **Utiliza parámetros** para permitir que las subrutinas reciban datos externos para una operación que se adapte al entorno.

- **Pueden ser colaborativas**, una subrutina puede llamar a otras subrutinas par que colabore en el proceso; eso se realiza invocándola con su nombre y opcionalmente con parámetros.

Las subrutinas en VBA son fundamentales para escribir código más organizado, estructurado y fácil de mantener.

```
Sintaxis
Sub NombreSubrutina()
        ‘ Código que realiza alguna tarea
End Sub
```

Descripción de la sintaxis:

- Sub: palabra clave que indica el inicio de una subrutina.
- NombreSubrutina: el nombre que identifica a la subrutina.
- End Sub: palabra clave que marca el final de la subrutina.

Manos a la obra, tratemos de crear nuestra primera subrutina.

Ahora que hemos explorado la sintaxis y estructura de una subrutina en VBA, pongamos manos a la obra y creemos nuestra primera subrutina paso a paso para aprender de forma empírica.

1) Abrir el Editor de Visual Basic (VBE)

Para comenzar, sigue estos pasos:
- En Excel, ve a la pestaña *Programador* y haz clic en *Visual Basic*.
- También puedes utilizar el atajo de teclado *Alt + F11* para abrir el Editor de VBA.

2) Crear un nuevo módulo
Dentro del Editor de VBA:
- Haz clic en *Insertar → Módulo*.
- Se abrirá una nueva ventana donde escribiremos nuestro código.

3) Escribir la subrutina
Ahora, copia el siguiente código en el módulo:

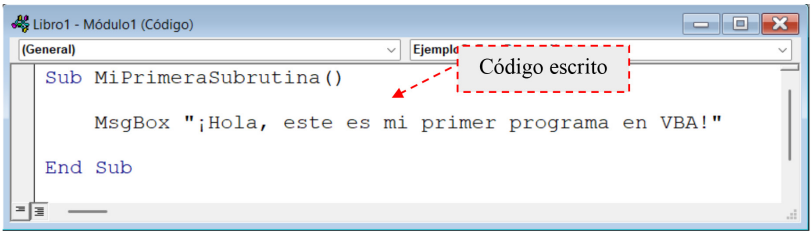


Figura 75. Ejemplo de una subrutina en VBA.

4) Explicación del código línea por línea

- Se declara una *subrutina* llamada *MiPrimeraSubrutina*.

```
Sub MiPrimeraSubrutina()
```

- Se usa la función MsgBox, que muestra un **cuadro de mensaje** con el texto "¡Hola, este es mi primer programa en VBA!".
- Cuando se ejecute la Macro, aparecerá una ventana emergente con este mensaje y un botón **"Aceptar"**.

```
MsgBox "¡Hola, este es mi primer programa en VBA!"
```

- Indica el **fin de la subrutina**.

```
End Sub
```

5) Guardar la Macro

Para evitar perder nuestro código, guardemos el archivo correctamente:

- Presiona Archivo → Guardar como.
- En Tipo de archivo, selecciona Libro de Excel habilitado para Macros (*.xlsm).
- Asigna un nombre al archivo y haz clic en Guardar.

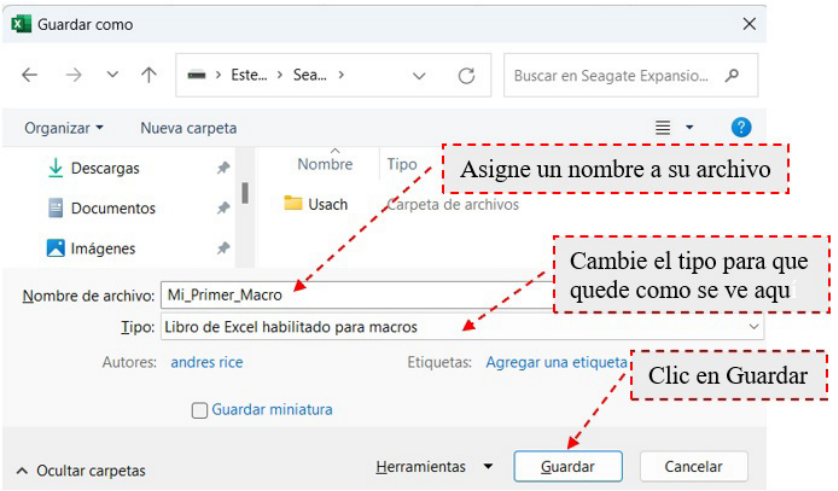


Figura 76. Cómo guardar una Macro.

Si no seleccionas la opción que indica habilitado para Macros, perderás la Macro al guardar.

6) Ejecutar la subrutina
Para probar nuestra subrutina:

- Dentro del Editor de VBA, haz clic en la flecha Ejecutar (o presiona F5).

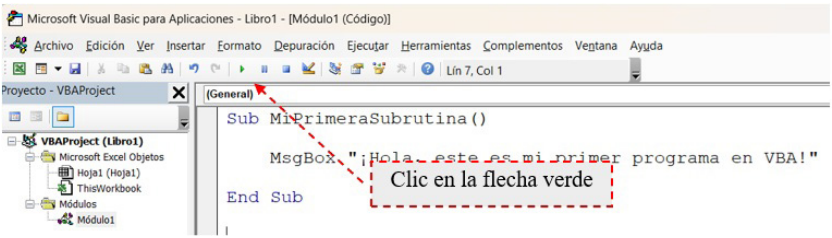


Figura 77. Cómo ejecutar una macro.

- Aparecerá una ventana emergente con el mensaje: ¡Hola, este es mi primer programa en VBA!

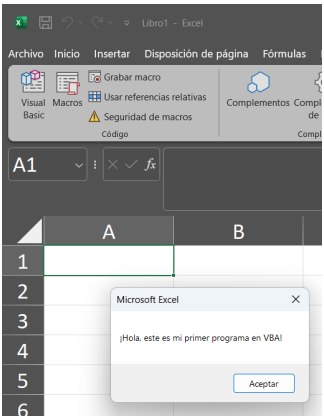


Figura 78. Ventana emergente después de la ejecución.

¡Felicidades! Has creado y ejecutado tu primera subrutina en VBA.

A partir de aquí, puedes explorar más funciones, agregar parámetros a las subrutinas o combinarlas con otras Macros para automatizar tareas aún más complejas. ¡El límite es tu creatividad!

4.2.1. ¿Cómo llamar a una subrutina desde otra subrutina?

En VBA, es común dividir el código en pequeñas secciones llamadas subrutinas para mejorar la organización y reutilización del código. En muchas ocasiones, una subrutina necesita ejecutar otra para completar una tarea específica. Esto se hace llamando a una subrutina desde otra.

1) Llamando a una subrutina sin parámetros
Para llamar a una subrutina dentro de otra, simplemente escribimos su nombre dentro del código.

Ejemplo:

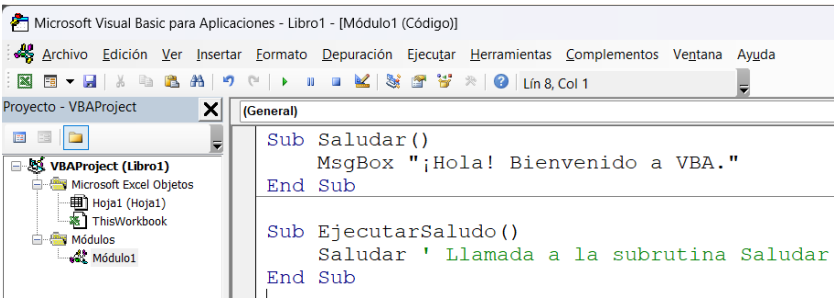


Figura 79. Subrutinas en VBA.

Explicación:

- Sub Saludar(): esta subrutina muestra un mensaje en pantalla.
- Sub EjecutarSaludo(): al ejecutarse, invoca la subrutina Saludar, lo que genera la ventana emergente con el mensaje.

2) Llamando a una subrutina con parámetros
Si queremos pasar información a una subrutina, podemos definirla con parámetros.

```
Sintaxis
Sub NombreSubrutina(parámetro1 As Tipo, parámetro2 As Tipo)
        ‘ Código que usa los parámetros
End Sub
```

Ejemplo:

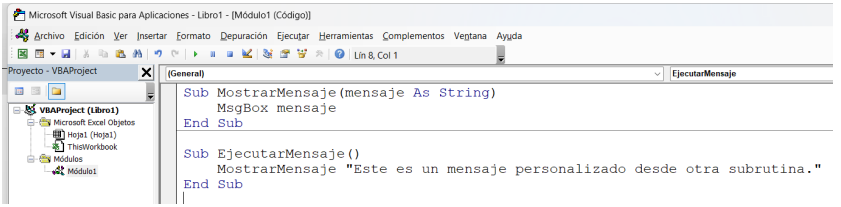


Figura 80. Ejemplo del llamando a una subrutina con parámetros.

Análisis línea por línea del código MostrarMensaje

- Define una subrutina llamada MostrarMensaje.
- Recibe un parámetro mensaje de tipo String, lo que significa que se debe proporcionar un texto al llamar a la subrutina.

```
Sub MostrarMensaje(mensaje As String)
```

- Usa la función MsgBox para mostrar el contenido de la variable mensaje en un cuadro de mensaje.

```
MsgBox mensaje
```

- Indica el fin de la subrutina.

```
End Sub
```

Subrutina EjecutarMensaje

- Define una segunda subrutina llamada EjecutarMensaje.
- No recibe parámetros, simplemente ejecuta la subrutina MostrarMensaje.

```
Sub EjecutarMensaje()
```

- Llama a la subrutina MostrarMensaje y le pasa un texto como argumento.

```
MostrarMensaje "Este es un mensaje personalizado desde otra subrutina".
```

- En este caso, el mensaje que se mostrará en el MsgBox será:

```
Este es un mensaje personalizado desde otra subrutina.
```

- Indica el fin de la subrutina.

```
End Sub
```

3) Buenas prácticas al llamar subrutinas

- Modularización: divide el código en subrutinas pequeñas y específicas para mejorar su mantenimiento.
- Uso de parámetros: permite reutilizar subrutinas sin modificar su código interno.

Evitar código duplicado: si una tarea se repite en varias partes del código, encapsúlala en una subrutina.

4.3. Funciones

Al igual que una subrutina Sub(), una función es un bloque de código diseñado para ejecutar una tarea específica. Sin embargo, a diferencia de una subrutina, una función siempre devuelve un resultado al finalizar su ejecución. Este resultado puede utilizarse en otras partes del programa o directamente en una hoja de cálculo de Excel, lo que facilita la integración de cálculos y procesos personalizados de manera flexible y reutilizable.

¿Para qué sirve una función en VBA?

Una función en VBA sirve para encapsular una serie de instrucciones que realizan una tarea específica y retornar un resultado de ese proceso. Esto permite reutilizar el código, ordenar el proceso, mejorar la claridad del programa y simplificar procesos complejos al dividir el trabajo en bloques más manejables. Aquí tienes algunas aplicaciones clave de las funciones:

1. **Reutilizar código y evitar repetición**
   En lugar de escribir las mismas líneas de código cada vez que se requiere un mismo cálculo o proceso, se puede crear una función que encapsule estas instrucciones y llamarla en cualquier parte del programa. Esto no solo simplifica el código y mejora su organización, sino que también facilita su mantenimiento. Cualquier modificación en la función se reflejará automáticamente en todos los lugares donde se invoque, reduciendo la posibilidad de errores y mejorando la eficiencia del desarrollo.

2. **Mejorar la modularización y mantenibilidad del código**
   Dividir un problema en partes más pequeñas facilita la construcción de una solución eficiente y organizada. Al estructurar el código en funciones, cada una se encarga de una tarea específica, lo que mejora la claridad y comprensión del programa. Además, esta segmentación permite aplicar mejoras o corregir errores de manera más focalizada, ya que cada función puede probarse y depurarse de forma independiente. Esto no solo optimiza el mantenimiento del código, sino que también reduce el riesgo de afectar otras partes del programa al realizar modificaciones.

3. **Estandarizar procesos complejos**
   Las funciones permiten estructurar y estandarizar procesos que deben ejecutarse de manera uniforme en distintas partes del código. Al encapsular una lógica de cálculo dentro de una función, se garantiza que este proceso se aplique siempre de forma consistente y sin errores.

Por ejemplo, en una tabla de amortización, el cálculo del interés de cada periodo sigue una fórmula específica que depende del saldo pendiente y la tasa de interés. En lugar de repetir este cálculo en varias partes del código, se puede definir una función que tome como entrada el saldo pendiente y la tasa de interés, y devuelva el monto del interés calculado para el periodo correspondiente. Esto asegura que el cálculo se realice de manera uniforme en toda la tabla, evitando errores y facilitando su actualización si la fórmula debe ajustarse en el futuro.

De este modo, al centralizar la lógica en una función, cualquier cambio solo requiere modificar una única sección del código, lo que simplifica la actualización y el mantenimiento del programa.

4. **Realizar cálculos personalizados**

   Las funciones son útiles para realizar cálculos específicos que se repiten en distintas partes del código. Puedes usarlas para calcular valores financieros, convertir unidades o ejecutar operaciones matemáticas avanzadas, entre otras tareas. Dependiendo del propósito, una función puede devolver valores numéricos, cadenas de texto o fechas, adaptándose a las necesidades del programa.

5. **Crear funciones personalizadas en Excel**

   Las funciones pueden utilizarse directamente en las celdas de Excel como si fueran funciones nativas, lo cual es muy útil para cálculos específicos en hojas de cálculo. Esto permite diseñar fórmulas personalizadas para realizar tareas que no están cubiertas por las funciones estándar de Excel.

Clasificación de funciones

Las funciones las podemos dividir en dos grandes grupos:

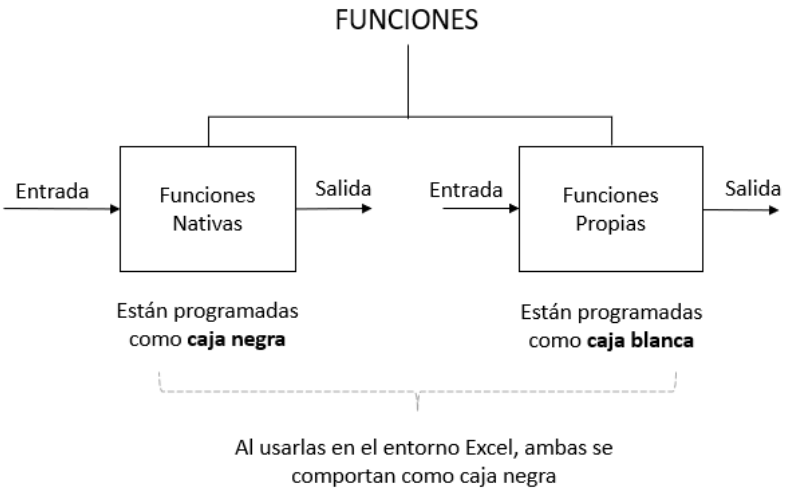


Figura 81. Tipos de funciones en VBA.

4.3.1. Funciones nativas

VBA cuenta con una amplia variedad de **funciones nativas** que permiten realizar operaciones de manera sencilla, sin necesidad de escribir código adicional. Estas funciones están organizadas en categorías que cubren distintas áreas:

- **matemáticas** (para cálculos y operaciones numéricas),
- **texto** (para manipulación de cadenas),
- **fecha y hora** (para trabajar con datos temporales),
- **conversión** (para cambiar entre tipos de datos) y
- **manejo de errores** (para controlar y validar situaciones inesperadas en el código).

En el Apéndice B de este libro ("Apéndice B: Listado de funciones nativas de VBA") encontrará un extenso listado de funciones nativas disponibles en VBA.

Utilizar una función nativa en VBA es sencillo: basta con llamar a la función por su nombre, proporcionar los argumentos requeridos (si los hay) y asignar el resultado a una variable o utilizarlo directamente en una operación.

Ejemplo:

El siguiente código es un procedimiento en VBA que suma los valores de un rango de celdas en Excel y muestra el resultado en un cuadro de mensaje.

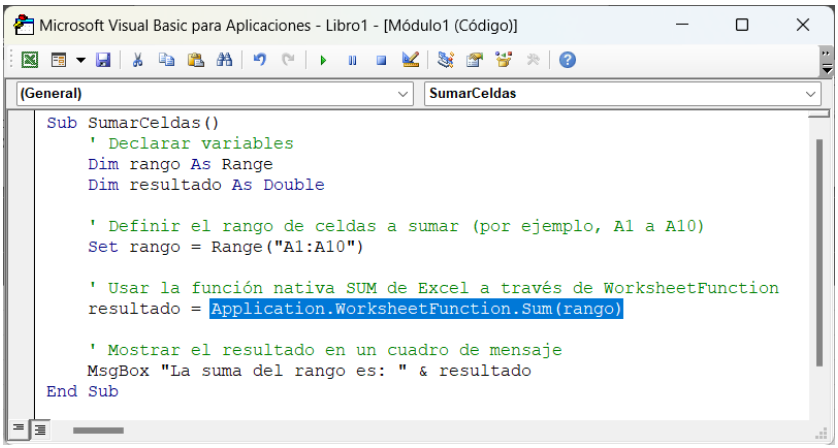


Figura 82. Ejemplo de uso de una función nativa.

A continuación, se explica su funcionamiento línea por línea:

- Declara una *subrutina* llamada SumarCeldas().

No devuelve valores directamente, ya que es una **Sub**, no una **Function**.

```
Sub SumarCeldas()
```

Se declaran dos variables:

- Rango de tipo Range, que almacenará el conjunto de celdas a sumar.
- Resultado de tipo Double, que almacenará el total de la suma.

```
Dim rango As Range
Dim resultado As Double
```

- Se asigna el rango de celdas A1:A10 a la variable rango utilizando Set (porque es un objeto).

```
Set rango = Range("A1:A10")
```

- El resultado se almacena en la variable resultado.
- Se usa la función nativa **SUM** de Excel a través del objeto Application.WorksheetFunction.
- Application.WorksheetFunction.Sum(rango) calcula la suma de los valores dentro del rango A1:A10.

El resultado se almacena en la variable resultado.

```
resultado = Application.WorksheetFunction.Sum(rango)
```

Muestra un cuadro de mensaje (MsgBox) con el resultado de la suma.

```
MsgBox "La suma del rango es":  & resultado
```

Finaliza la subrutina.

```
End Sub
```

- Resumen del funcionamiento
  1. Define un *rango de celdas* (A1:A10).
  2. Usa la función nativa *SUM* de Excel en VBA para sumar los valores de ese rango.
  3. Muestra el resultado en un cuadro de mensaje.

A continuación, se presentan otros ejemplos con uso de funciones nativas dentro de un procedimiento:

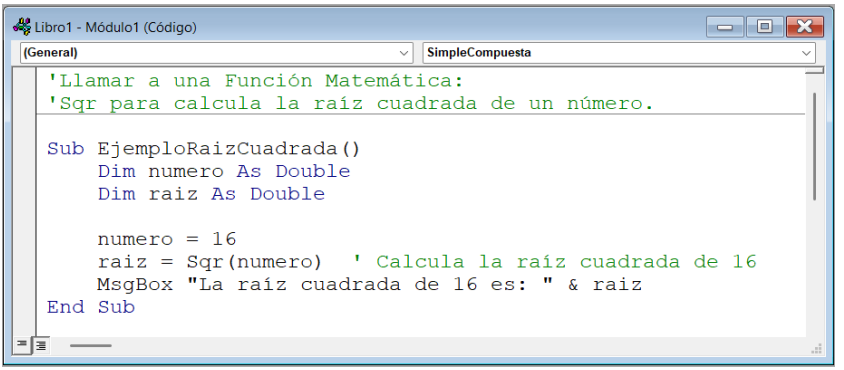


Figura 83. Ejemplo de cálculo de la raíz cuadrada de un número.

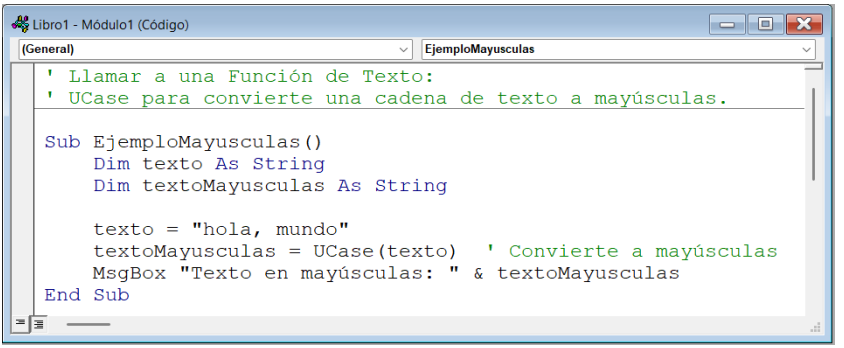


Figura 84. Ejemplo para convertir texto a mayúsculas.

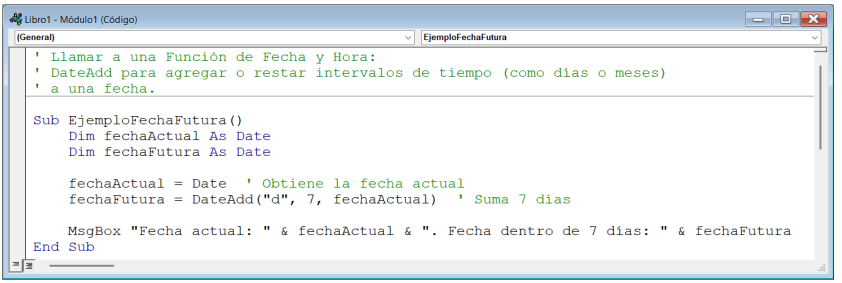


Figura 85. Ejemplos con funciones para fechas.

Estos son ejemplos sencillos pero muy útil sobre cómo utilizar funciones nativas de Excel dentro de una Macro. Te animo a modificar este código para que explore otras funciones nativas, como Max(), Min(), Average() y muchas más. Prueba adaptarlo para encontrar el valor máximo o mínimo de un rango, calcular promedios o realizar diferentes operaciones con funciones nativas.

4.3.2. Funciones propias

En VBA, también es posible crear funciones personalizadas, diseñadas por el usuario para realizar tareas específicas y reutilizables dentro del código. Estas funciones pueden recibir parámetros y devolver un valor, lo que permite encapsular cálculos, lógica y operaciones repetitivas en una sola instrucción. A diferencia de las funciones nativas, las funciones personalizadas se definen directamente en el código y se adaptan a las necesidades específicas del proyecto. Esto no solo mejora la organización del código, sino que también lo hace más eficiente y flexible para diferentes situaciones.

```
Sintaxis

Function NombreFunción([argumentos]) As TipoDeDato
                ‘ Cuerpo de la función
                NombreFuncion = ValorDeRetorno
End Function
```

4.3.3. Explicación de la sintaxis

- Function → Define que se trata de una función.
- NombreFunción → Nombre de la función (debe ser único en el módulo).
- ([argumentos]) → Parámetros opcionales que recibe la función (puede haber ninguno o varios).
- As TipoDeDato → Define el tipo de dato que devuelve la función (Integer, Double, String, etc.).
- Cuerpo de la función → Código que realiza la operación.
- NombreFunción = ValorDeRetorno → La función asigna un valor de retorno con el mismo nombre de la función.
- End Function → Finaliza la función.

Ejemplo:

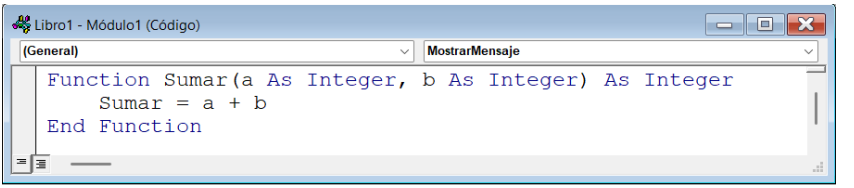


Figura 86. Ejemplo de una función escrita en VBA.

Análisis del código línea por línea:
- Declara una función llamada Sumar.
- Recibe dos parámetros (a y b), ambos de tipo Integer.
- La función devolverá un valor de tipo Integer (indicado por `As` Integer).

4.3.4. Cómo trabajar con un rango de celdas en una función

Para comprender cómo trabajar con rangos de celdas en las funciones tratemos de replicar la función nativa sumar. Para sumar un rango de celdas utilizaremos el siguiente ejemplo de una función:

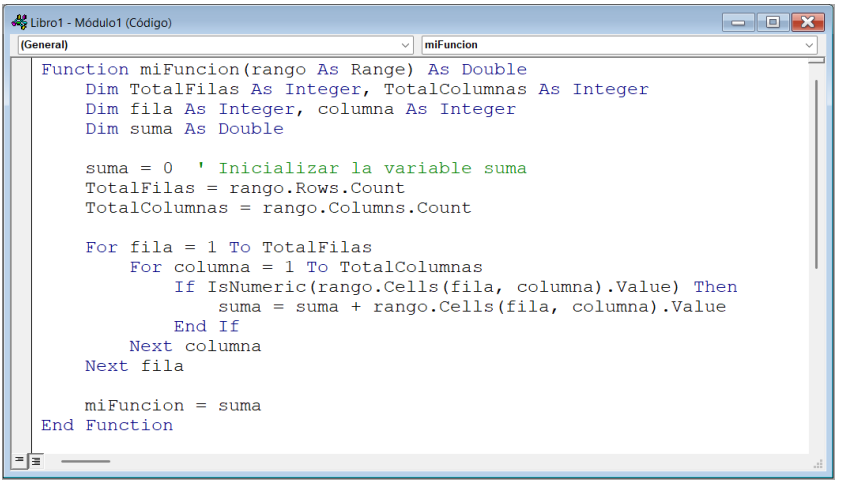


Figura 87. Ejemplo de una función con rango de datos.

Análisis del código línea por línea:

Esta función en VBA se llama miFuncion y su propósito es recibir un rango de celdas en Excel y devolver la suma de todos los valores dentro de ese rango.

- Declara una función llamada miFuncion.

```
Function miFuncion(rango As Range) As Variant
```

- Recibe un parámetro denominado rango de tipo Range, lo que significa que acepta un rango de celdas como argumento.
- La función devuelve el contenido de una variable tipo Variant, lo que le permite manejar distintos tipos de datos.

Obtener dimensiones del rango para saber cuántas filas y cuántas columnas (total de celdas) son las que hay que sumar

```
TotalFilas = rango.Rows.Count
TotalColumnas = rango.Columns.Count
```

- rango.Rows.Count obtiene la cantidad total de filas dentro del rango.
- rango.Columns.Count obtiene la cantidad total de columnas dentro del rango.

Inicialización de variables y bucle anidado

```
For fila = 1 To TotalFilas
        For columna = 1 To TotalColumnas
                suma = suma + rango(i)
        Next columna
Next fila
```

- Se usa un bucle anidado para recorrer todas las filas y columnas dentro del rango.
- Bucle externo (For fila = 1 To TotalFilas)
  - Recorre las filas del rango.
- Bucle interno (For columna = 1 To TotalColumnas)
  - Dentro de cada fila, recorre todas las columnas.
- Suma de valores:
  suma = suma + rango(i) →Agrega el valor de la celda actual a la variable suma.

Devolver el resultado

```
miFuncion = suma
```

- La función asigna el resultado de la suma a miFuncion, lo que significa que este valor será devuelto cuando se finalice la ejecución de la función.

Finaliza el contexto de la función

```
End Function
```

Podemos verificar si el resultado devuelto por la función miFuncion(B2:H5) es correcto sumando manualmente los valores dentro del rango B2:H5. La matriz de datos en el rango B2:H5 es:

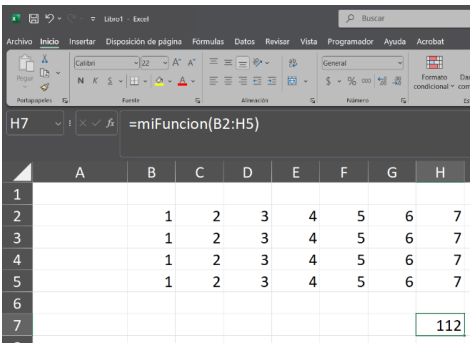


Figura 88. Matriz con datos a sumar.

Paso 1: sumar los valores en el rango B2:H5
Fila 2: 1+2+3+4+5+6+7=28
Fila 3: 1+2+3+4+5+6+7=28
Fila 4: 1+2+3+4+5+6+7=28
Fila 5: 1+2+3+4+5+6+7=28
Total: 28+28+28+28=112

Paso 2: comparar con el resultado de la función
En la celda *H7*, escriba *"=miFuncion(B2:H5)"*
Al tipear Enter, devolverá el valor *112*, lo cual es correcto.

Conclusión
La función miFuncion en VBA está funcionando correctamente y devuelve la suma esperada de los valores en el rango dado.
Usted puede realizar más pruebas con otros datos y verificar que la función opera correctamente.

4.3.5. Comparación entre Sub y Function en VBA

En VBA (Visual Basic for Applications) existen dos estructuras principales para organizar y ejecutar código: Subrutinas (Sub) y Funciones (Function). Ambas permiten automatizar tareas, pero se diferencian en su propósito y funcionamiento.

- Subrutinas (Sub): ejecutan una serie de instrucciones sin devolver un valor. Son útiles para realizar acciones como mostrar mensajes, modificar celdas, interactuar con el usuario o ejecutar procesos por lotes.
- Funciones (Function): procesan datos y devuelven un valor, lo que permite reutilizar cálculos dentro de VBA o directamente en celdas de Excel.

A continuación, se presenta una tabla comparativa que destaca sus diferencias clave, acompañada de ejemplos prácticos para ilustrar su aplicación en distintos escenarios.

Tabla 9. Comparativa entre function y Sub

Característica	Subrutina (Sub)	Función (Function)
Propósito	Ejecuta un bloque de código sin devolver un valor.	Ejecuta un bloque de código y devuelve un valor.
Sintaxis	Sub NombreSubrutina() End Sub	Function NombreFuncion() As TipoDeDato End Function
Retorno de valores	No devuelve valores directamente.	Devuelve un valor utilizando el mismo nombre de la función.
Uso en Excel	No puede ser usada directamente en celdas de Excel.	Puede ser llamada desde celdas de Excel como una función personalizada.

Capítulo 5

Importación y exportación de datos con Macros

En la actualidad, la gestión eficiente de datos es clave para la toma de decisiones fundamentadas en información confiable. La importación y exportación de datos es una tarea recurrente, especialmente en entornos donde Excel se utiliza para consolidar información proveniente de múltiples fuentes, como bases de datos, sistemas ERP y archivos en formatos CSV, XML, entre otros.

Empresas y profesionales trabajan con datos que requieren actualizaciones constantes. Ya sea para cargar registros de ventas, analizar tendencias del mercado, generar reportes financieros o actualizar bases de datos, la automatización de estos procesos mediante Macros en VBA no solo reduce el trabajo manual y minimiza errores, sino que también agiliza el análisis de los datos.

Dado que la importación de datos es una tarea cotidiana en muchas organizaciones, depender de procesos manuales no solo ralentiza el flujo de trabajo, sino que también incrementa el riesgo de inconsistencias y errores humanos. Por ello, el uso de Macros VBA permite estandarizar, optimizar y acelerar la extracción y transferencia de datos, asegurando que la data esté siempre actualizada y lista para su análisis.

A continuación, exploraremos cómo automatizar la importación y exportación de datos en Excel mediante VBA, con ejemplos prácticos que permitirán optimizar estos procesos de manera eficiente, precisa y segura.

5.1. Tipos de archivos posibles de gestionar

La variedad de formatos de archivos que pueden ser gestionados en Excel mediante VBA permite una integración eficiente con diferentes sistemas y fuentes de datos. Dependiendo de la estructura y propósito de los archivos, estos pueden clasificarse en estructurados, como CSV, XML y JSON, que organizan la información de manera jerárquica o tabular; y no estructurados, como archivos de texto plano (TXT), que requieren procesamiento adicional para su interpretación.

Cada formato tiene ventajas específicas según el contexto en el que se utilice. Los archivos tipo CSV (Comma-Separated Values o Valores Separados por Comas) son ampliamente empleados para intercambiar datos debido a su simplicidad y compatibilidad universal. En un archivo CSV, los datos están organizados en filas y columnas, donde los valores dentro de cada fila están separados por comas (,), aunque en algunos casos pueden utilizarse otros delimitadores, como punto y coma (;) o tabulación. Este formato permite almacenar grandes volúmenes de datos en un formato ligero y fácilmente accesible.

Por otro lado, los archivos XML (Extensible Markup Language) y JSON (JavaScript Object Notation) se utilizan especialmente en la comunicación con APIs, donde es necesario manejar datos estructurados en formatos estandarizados.

Los archivos TXT (texto plano) pueden contener registros de información en líneas separadas o formatos delimitados, facilitando su uso en reportes automatizados o almacenamiento de logs (registros de eventos, procesos o actividades que ocurren en un sistema).

Asimismo, los archivos XML (Extensible Markup Language) y JSON (JavaScript Object Notation) se utilizan en aplicaciones modernas, especialmente en la comunicación con APIs, donde es necesario manejar datos estructurados en formatos estandarizados. Los archivos TXT (texto plano) pueden contener registros de información en líneas separadas o formatos delimitados, facilitando su uso en reportes automatizados o almacenamiento de logs.

Además de la importación de datos, la exportación en estos formatos también es clave en la automatización de procesos, permitiendo que Excel sirva como una herramienta de generación de reportes dinámicos o como puente de comunicación con otros sistemas. A través de Macros

en VBA, es posible gestionar la importación y exportación de estos archivos de manera eficiente, asegurando la integridad de la información y optimizando los tiempos de procesamiento.

La capacidad de importar datos desde diversas fuentes es fundamental para el análisis de datos que servirá como insumo en la toma de decisiones informada. Excel, combinado con VBA, permite automatizar la importación de datos desde archivos tipo CSV, TXT, XML, JSON y bases de datos, lo que facilitó su procesamiento y análisis eficiente.

Al importar datos de manera automatizada no solo ahorra tiempo, sino que también reduce errores humanos y mejora la consistencia de la información. Empresas y analistas utilizan esta técnica para consolidar grandes volúmenes de datos provenientes de distintas plataformas, tales como sistemas contables, ERPs, bases de datos SQL, o APIs web.

Al estructurar correctamente la importación de datos, se pueden aplicar filtros, generar reportes dinámicos y realizar análisis predictivos con herramientas avanzadas como tablas dinámicas y modelos estadísticos. A continuación, exploraremos cómo VBA facilita este proceso y cómo se pueden implementar Macros para importar datos de manera eficiente.

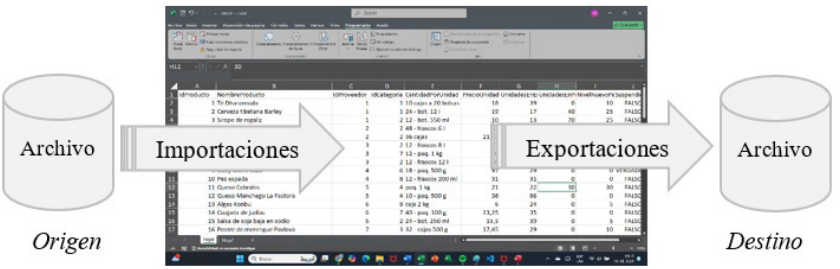


Figura 89. Importación y Exportación en Excel.

5.2. Importar o exportar archivos CSV con Macros

Los archivos de texto son uno de los formatos más utilizados para intercambiar datos entre sistemas. Excel puede importar y exportar este tipo de archivos de manera sencilla con Macros.

CSV (Comma-Separated Values)

Formato de datos estructurados donde cada campo está separado por comas (,) o punto y coma (;). Se puede abrir en Excel directamente, pero VBA permite automatizar su importación.

5.2.1. Macro para importar archivos tipo CSV

Ejemplo:

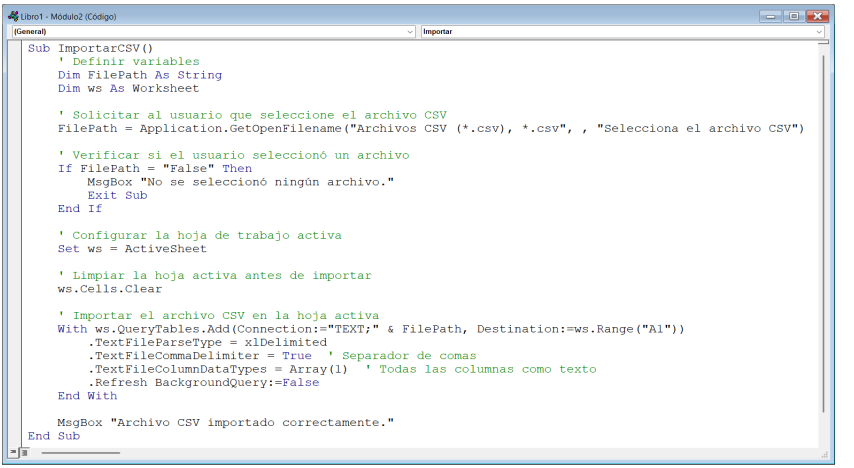


Figura 90. Macro para importar archivo CSV.

Análisis línea a línea de la Macro Importar CSV()

1) Definición de variables

```
Dim FilePath As String
Dim ws As Worksheet
```

- FilePath: variable de tipo String que almacenará la ruta del archivo CSV seleccionado.
- ws: variable que representa la hoja de cálculo activa donde se importará el archivo.

2) Selección del archivo CSV

```
FilePath = Application.GetOpenFilename("Archivos CSV (*.csv), *.csv", ,
"Selecciona el archivo CSV")
```

- Se abre una ventana emergente para que el usuario seleccione un archivo CSV.
- Solo se pueden seleccionar archivos con extensión *.csv*.

3) Validación de la selección del archivo

```
If FilePath = "False" Then MsgBox "No se seleccionó ningún archivo." Exit
Sub End If
```

- Si el usuario **cancela la selección**, FilePath toma el valor "False", por lo que la Macro finaliza mostrando un mensaje de advertencia.

4) Configurar la hoja activa

```
Set ws = ActiveSheet
```

- La variable ws se asigna a la hoja activa, indicando que el archivo CSV se importará en esta hoja.

5) Limpiar la hoja activa antes de importar

```
ws.Cells.Clear
```

- Se elimina todo el contenido de la hoja antes de importar el archivo CSV.
- Esta acción evita que queden datos previos en la hoja.

6) Importar el archivo CSV en la hoja activa

```
With ws.QueryTables.Add(Connection:="TEXT;" & FilePath,
Destination:=ws.Range("A1"))
    .TextFileParseType = xlDelimited .TextFileCommaDelimiter = True '
    Separador de comas
    .TextFileColumnDataTypes = Array(1) ' Todas las columnas como texto
    .Refresh BackgroundQuery:=False
End With
```

- QueryTables.Add: crea una conexión de datos desde el archivo CSV seleccionado.
- Connection:="TEXT;" & FilePath: especifica que la fuente de datos es un archivo de texto (CSV).
- Destination:=ws.Range("A1"): indica que los datos se importarán desde la celda A1.
- TextFileParseType = xlDelimited: define que los datos están separados por un delimitador.

- TextFileCommaDelimiter = True: especifica que el delimitador es una coma (,).
- TextFileColumnDataTypes = Array(1): establece que todas las columnas serán importadas como texto.
- .Refresh BackgroundQuery:=False: ejecuta la importación de datos sin retrasos.

7) Confirmación de importación

```
MsgBox "Archivo CSV importado correctamente."
```

- Muestra un mensaje de confirmación al usuario cuando la importación se ha completado con éxito.

Esta Macro en VBA facilita la importación de archivos CSV en Excel, evitando errores manuales y automatizando el proceso de carga de datos.

5.2.2. Macro para exportar archivos tipo CSV

Ejemplo:

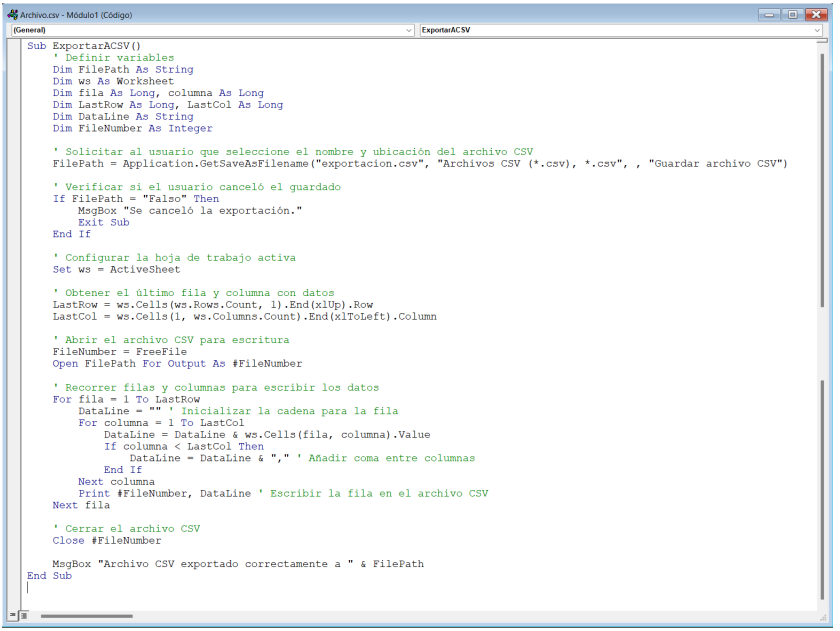


Figura 91. Macro para exportar archivo CSV.

Análisis línea por línea de la Macro exportar ACSV()

1) Definición de variables

```
Dim FilePath As String Dim ws As Worksheet
Dim fila As Long, columna As Long
Dim LastRow As Long, LastCol As Long
Dim DataLine As String
Dim FileNumber As Integer
```

- FilePath: almacena la ruta donde se guardará el archivo CSV.
- ws: representa la hoja activa de Excel.
- Fila, columna: variables para recorrer las celdas de la hoja.

- LastRow, LastCol: guardan la última fila y columna con datos en la hoja.
- DataLine: variable donde se construye cada línea del archivo CSV.
- FileNumber: número del archivo para manipulación en VBA.

2) Pedir al usuario la ubicación del archivo

```
FilePath = Application.GetSaveAsFilename("exportacion.csv", "Archivos
CSV (*.csv), *.csv", , "Guardar archivo CSV")
```

- Application.GetSaveAsFilename: abre un cuadro de diálogo para que el usuario seleccione dónde guardar el archivo CSV y con qué nombre.
- "exportacion.csv": nombre sugerido por defecto.
- "Archivos CSV (*.csv), *.csv": filtro para que solo se pueda guardar en formato .csv.
- "Guardar archivo CSV": título de la ventana emergente.

3) Verificar si el usuario canceló el guardado

```
If FilePath = "False" Then
   MsgBox "Se canceló la exportación."
   Exit Sub
End If
```

- Si el usuario presiona "Cancelar" en la ventana de selección de archivo, FilePath tomará el valor "False", por lo que la Macro muestra un mensaje y finaliza sin realizar la exportación.

4) Definir la hoja de trabajo activa

```
Set ws = ActiveSheet
```

- Se asigna la hoja activa a la variable ws, lo que significa que los datos a exportar provendrán de la hoja en la que el usuario esté trabajando.

5) Obtener el último rango de datos en la hoja

```
LastRow = ws.Cells(ws.Rows.Count, 1).End(xlUp).Row
LastCol = ws.Cells(1, ws.Columns.Count).End(xlToLeft).Column
```

- LastRow: encuentra la última fila con datos en la columna A.
- LastCol: encuentra la última columna con datos en la fila 1.

Este paso es fundamental para recorrer dinámicamente la tabla sin importar su tamaño.

6) Abrir el archivo CSV para escritura

```
FileNumber = FreeFile
Open FilePath For Output As #FileNumber
```

- FreeFile: obtiene un número de archivo disponible en VBA.
- Open FilePath For Output As #FileNumber: abre el archivo en modo escritura (Output), lo que permite guardar datos en él.

7) Recorrer la hoja y guardar los datos en el archivo CSV

```
For fila = 1 To LastRow
   DataLine = "" ' Inicializar la cadena para la fila
      For columna = 1 To LastCol
      DataLine = DataLine & ws.Cells(fila, columna).Value
      If columna < LastCol Then
         DataLine = DataLine & "," ' Añadir coma como delimitador
      End If
   Next columna

   Print #FileNumber, DataLine ' Escribir la línea en el archivo CSV
Next fila
```

- Bucle For fila = 1 To LastRow:
  - Recorre todas las filas desde la 1 hasta la última con datos.
- Bucle For columna = 1 To LastCol:
  - Recorre todas las columnas dentro de la fila actual.
  - Extrae el contenido de la celda y lo concatena en DataLine.
  - Si la celda no es la última de la fila, agrega una coma (,) para separar los valores.

Resultado: cada fila de la hoja se transforma en una línea de texto en el archivo CSV.

8) Cerrar el archivo CSV

```
Close FileNumber
```

- Cierra el archivo después de escribir los datos, asegurando que toda la información quede guardada correctamente.

9) Confirmar que el archivo CSV fue exportado

```
MsgBox "Archivo CSV exportado correctamente a " & FilePath
```

- Muestra un mensaje con la ubicación donde se guardó el archivo.

5.3. Importar o exportar archivos XML con Macros

5.3.1. Macro para importar archivos tipo XML

Ejemplo:

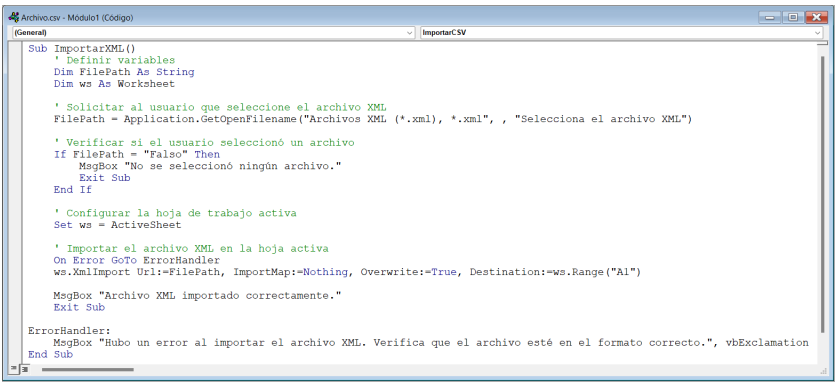


Figura 92. Macro para exportar archivo XML.

Análisis línea por línea de la Macro ImportarXML()

1) Definición de variables

```
Dim FilePath As String
Dim ws As Worksheet
```

- FilePath: variable de tipo String que almacenará la ruta del archivo XML seleccionado por el usuario.

- ws: variable que representará la hoja activa donde se importarán los datos del XML.

2) Solicitar al usuario la selección del archivo XML

```
FilePath = Application.GetOpenFilename("Archivos XML (*.xml), *.xml", ,
"Selecciona el archivo XML")
```

- Application.GetOpenFilename: abre un cuadro de diálogo para que el usuario seleccione el archivo XML.
- "Archivos XML (*.xml), *.xml": filtro para que solo se puedan seleccionar archivos con la extensión .xml.
- "Selecciona el archivo XML": mensaje de la ventana emergente para guiar al usuario.

3) Verificar si el usuario canceló la selección

```
If FilePath = "False" Then
   MsgBox "No se seleccionó ningún archivo."
   Exit Sub
End If
```

- Si el usuario presiona "Cancelar", FilePath tomará el valor "False", lo que indica que no se seleccionó un archivo.
- En este caso, se muestra un mensaje de advertencia y la Macro finaliza sin importar datos.

4) Configurar la hoja activa

```
Set ws = ActiveSheet
```

- La variable ws se asigna a la hoja activa, indicando que el archivo XML será importado en esta hoja.

5) Importar el archivo XML en la hoja activa

```
On Error GoTo ErrorHandler
ws.XmlImport Url:=FilePath, ImportMap:=Nothing, Overwrite:=True,
Destination:=ws.Range("A1")
```

- On Error GoTo ErrorHandler: si ocurre un error durante la importación, el código saltará a la etiqueta ErrorHandler para manejarlo.
- ws.XmlImport: método que importa el archivo XML.
- Url:=FilePath: indica la ruta del archivo XML seleccionado.
- ImportMap:=Nothing: no se utiliza un mapa de importación (se toma la estructura predeterminada del XML).
- Overwrite:=True: si hay datos previos en la hoja, los sobrescribe.
- Destination:=ws.Range("A1"): los datos importados comenzarán desde la celda A1.

Este comando carga el archivo XML en la hoja activa y distribuye sus datos en las columnas y filas correspondientes.

6) Confirmar la importación exitosa

```
MsgBox "Archivo XML importado correctamente."
Exit Sub
```

- Si el archivo se exporta sin problemas, aparece un mensaje de confirmación y la Macro finaliza correctamente.

7) Manejo de errores

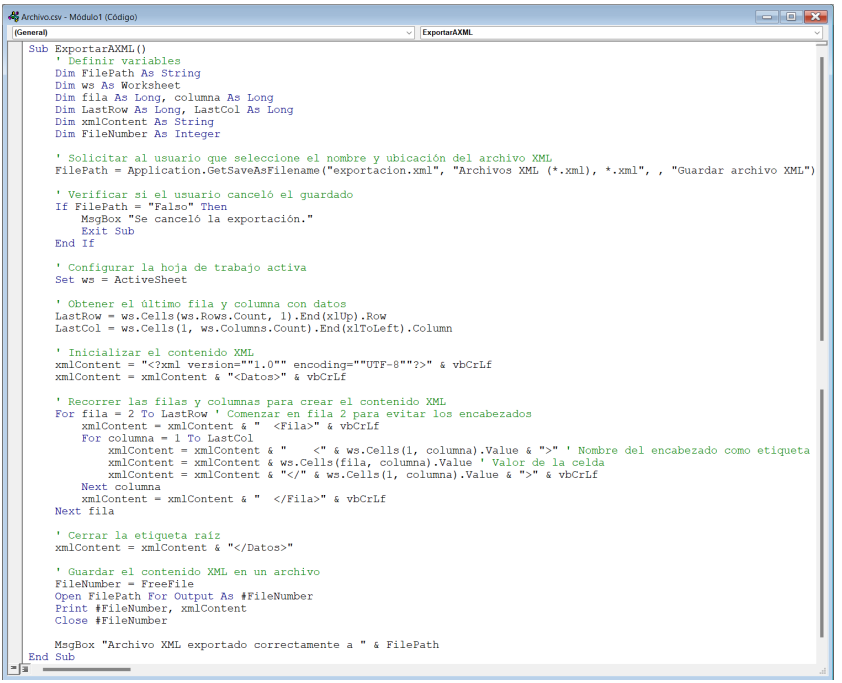
```
ErrorHandler:
MsgBox "Hubo un error al importar el archivo XML. Verifica que el
archivo esté en el formato correcto.", vbExclamation
```

- Si ocurre un error (por ejemplo, si el archivo XML está mal formado o no es válido), se ejecuta este bloque de código.

- Se muestra un mensaje de advertencia (vbExclamation) indicando que el archivo XML puede no estar en el formato correcto.

5.3.2. Macro para exportar archivos tipo XML

Ejemplo:




```
Sub ExportarAXML()
    ' Definir variables
    Dim FilePath As String
    Dim ws As Worksheet
    Dim fila As Long, columna As Long
    Dim LastRow As Long, LastCol As Long
    Dim xmlContent As String
    Dim FileNumber As Integer

    ' Solicitar al usuario que seleccione el nombre y ubicación del archivo XML
    FilePath = Application.GetSaveAsFilename("exportacion.xml", "Archivos XML (*.xml), *.xml", , "Guardar archivo XML")

    ' Verificar si el usuario canceló el guardado
    If FilePath = "Falso" Then
        MsgBox "Se canceló la exportación."
        Exit Sub
    End If

    ' Configurar la hoja de trabajo activa
    Set ws = ActiveSheet

    ' Obtener el último fila y columna con datos
    LastRow = ws.Cells(ws.Rows.Count, 1).End(xlUp).Row
    LastCol = ws.Cells(1, ws.Columns.Count).End(xlToLeft).Column

    ' Inicializar el contenido XML
    xmlContent = "<?xml version=""1.0"" encoding=""UTF-8""?>" & vbCrLf
    xmlContent = xmlContent & "<Datos>" & vbCrLf

    ' Recorrer las filas y columnas para crear el contenido XML
    For fila = 2 To LastRow ' Comenzar en fila 2 para evitar los encabezados
        xmlContent = xmlContent & "  <Fila>" & vbCrLf
        For columna = 1 To LastCol
            xmlContent = xmlContent & "    <" & ws.Cells(1, columna).Value & ">" ' Nombre del encabezado como etiqueta
            xmlContent = xmlContent & ws.Cells(fila, columna).Value ' Valor de la celda
            xmlContent = xmlContent & "</" & ws.Cells(1, columna).Value & ">" & vbCrLf
        Next columna
        xmlContent = xmlContent & "  </Fila>" & vbCrLf
    Next fila

    ' Cerrar la etiqueta raíz
    xmlContent = xmlContent & "</Datos>"

    ' Guardar el contenido XML en un archivo
    FileNumber = FreeFile
    Open FilePath For Output As #FileNumber
    Print #FileNumber, xmlContent
    Close #FileNumber

    MsgBox "Archivo XML exportado correctamente a " & FilePath
End Sub
```

Figura 93. Macro para exportar datos a XML.

Análisis línea por línea de la Macro ExportarAXML ()

1) Definición de variables

```
Dim FilePath As String
Dim ws As Worksheet
Dim fila As Long, columna As Long
Dim LastRow As Long, LastCol As Long
Dim xmlContent As String
Dim FileNumber As Integer
```

- FilePath: almacena la ruta donde se guardará el archivo XML.
- ws: variable que representa la hoja de trabajo activa.
- Fila, columna: variables para recorrer las filas y columnas de la hoja.
- LastRow, LastCol: última fila y columna con datos en la hoja, necesarias para recorrer dinámicamente la tabla.
- xmlContent: cadena de texto donde se almacenará el contenido del archivo XML.
- FileNumber: número de archivo asignado por VBA para escribir en el archivo XML.

2) Solicitar al usuario la ubicación y nombre del archivo XML

```
FilePath = Application.GetSaveAsFilename("exportacion.xml", "Archivos
XML (*.xml), *.xml", , "Guardar archivo XML")
```

- Application.GetSaveAsFilename: muestra una ventana para que el usuario elija dónde guardar el archivo XML.
- "exportacion.xml": nombre sugerido por defecto.
- "Archivos XML (*.xml), *.xml": filtro para permitir solo archivos XML.
- "Guardar archivo XML": título de la ventana emergente.

3) Verificar si el usuario canceló la exportación

```
If FilePath = "False" Then
   MsgBox "Se canceló la exportación."
   Exit Sub
End If
```

- Si el usuario presiona "Cancelar", la variable FilePath tomará el valor "False".
- Se muestra un mensaje de advertencia y la Macro finaliza sin realizar la exportación.

4) Definir la hoja de trabajo activa

```
Set ws = ActiveSheet
```

- Se asigna la hoja activa a la variable ws, lo que indica que los datos a exportar provienen de esta hoja.

5) Determinar el último rango de datos en la hoja

```
LastRow = ws.Cells(ws.Rows.Count, 1).End(xlUp).Row
LastCol = ws.Cells(1, ws.Columns.Count).End(xlToLeft).Column
```

- LastRow: encuentra la última fila con datos en la columna A.
- LastCol: encuentra la última columna con datos en la fila 1.

Este paso es fundamental para recorrer dinámicamente la tabla sin importar su tamaño.

6) Inicializar el contenido XML

```
xmlContent = "<?xml version=""1.0"" encoding=""UTF-8""?>" & vbCrLf
xmlContent = xmlContent & "<Datos>" & vbCrLf
```

Se define la estructura inicial del archivo XML:

- <?xml version="1.0" encoding="UTF-8"?>: Declaración estándar de XML.
- <Datos>: Nodo raíz del archivo XML, que contendrá todos los registros.

7) Recorrer la hoja y crear el contenido XML

```
For fila = 2 To LastRow ' Comienza en 2 para evitar los encabezados
   xmlContent = xmlContent & "  <Fila>" & vbCrLf

   For columna = 1 To LastCol
xmlContent = xmlContent & "    <" & ws.Cells(1, columna).Value &
">" ' Nombre del encabezado como etiqueta
            xmlContent = xmlContent & ws.Cells(fila, columna).Value '
Valor de la celda
xmlContent = xmlContent & "</" & ws.Cells(1, columna).Value & ">"
& vbCrLf
   Next columna
```

- Bucle For fila = 2 To LastRow:
  - Comienza en la segunda fila (2 en lugar de 1) porque la primera fila contiene los encabezados de las columnas que se usarán como etiquetas en XML.
- Bucle For columna = 1 To LastCol:
  - Recorre todas las columnas de cada fila.
  - <Encabezado>Valor</Encabezado>:
    - El nombre del encabezado de la columna (ws.Cells(1, columna).Value) se usa como etiqueta XML.
    - El contenido de la celda (ws.Cells(fila, columna).Value) se coloca dentro de la etiqueta.

Ejemplo de salida generada en XML:
<Fila>
<Nombre>Juan</Nombre>
<Edad>30</Edad>
<Ciudad>Madrid</Ciudad>
</Fila>

Este paso convierte cada fila de la hoja en un nodo <Fila> dentro del archivo XML.

8) Cerrar la etiqueta raíz

```
xmlContent = xmlContent & "</Datos>"
```

- Se agrega el cierre del nodo raíz <Datos>, asegurando que el XML tenga una estructura válida.

9) Guardar el contenido XML en un archivo

```
FileNumber = FreeFile
Open FilePath For Output As #FileNumber
Print #FileNumber, xmlContent
Close #FileNumber
```

- FreeFile: obtiene un número de archivo disponible.
- Open FilePath For Output As #FileNumber: abre el archivo en modo escritura (Output).
- Print #FileNumber, xmlContent: escribe todo el contenido almacenado en xmlContent en el archivo XML.
- Close #FileNumber: cierra el archivo, asegurando que se guarde correctamente.

10) Mensaje de confirmación

```
MsgBox "Archivo XML exportado correctamente a " & FilePath
```

- Una vez completada la exportación, muestra un mensaje de confirmación con la ubicación del archivo XML.

Esta Macro en VBA convierte datos de una hoja de Excel en un archivo XML estructurado. Es ideal para integrar Excel con sistemas que utilizan XML para el intercambio de información.

5.4. Uso de las Macros para acceder a las bases de datos

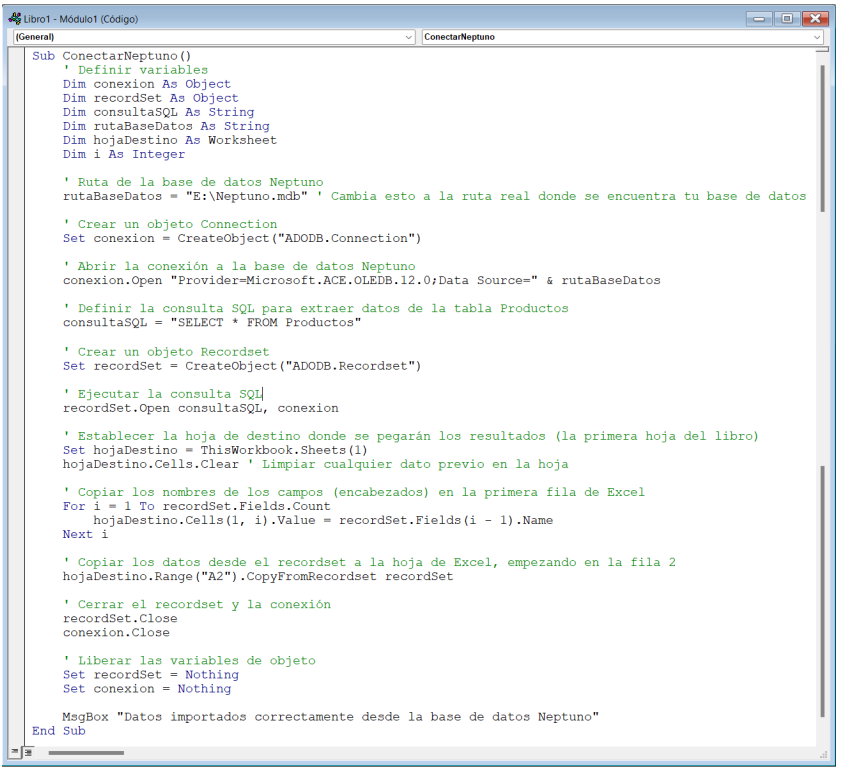
Conectar Excel con una base de datos es esencial para **profesionales de contabilidad y auditoría**, ya que combina la **facilidad de análisis** que posee Excel con la capacidad de los sistemas de bases de datos (como Access o SQL Server) para gestionar grandes volúmenes de datos de forma segura. Esta conexión puede permitir automatizar la extracción, actualización de datos o realizar consultas SQL y generar informes financieros precisos en un tiempo breve.

La integración bidireccional entre Excel y la base de datos es clave para **auditar datos**, actualizar registros y crear informes contables sin errores manuales, para segurar la confiabilidad de la información. Dominar esta técnica optimiza la gestión de datos, automatiza tareas repetitivas y asegura la precisión en la elaboración de informes financieros.

¿Cómo conectar con una base de datos?

Para establecer una conexión con una base de datos es fundamental asegurarse de contar con los permisos adecuados para acceder a ella y realizar consultas. Si trabajas con una base de datos sensible, es importante manejar las credenciales de acceso de forma segura y aplicar buenas prácticas de seguridad, como el uso de conexiones cifradas o la encriptación de datos, para garantizar la protección de la información

durante la interacción con la base de datos. A continuación, un ejemplo de una Macro que se conecte con una base de datos para extraer los registros de una de sus tablas:




```
Sub ConectarNeptuno()
    ' Definir variables
    Dim conexion As Object
    Dim recordSet As Object
    Dim consultaSQL As String
    Dim rutaBaseDatos As String
    Dim hojaDestino As Worksheet
    Dim i As Integer

    ' Ruta de la base de datos Neptuno
    rutaBaseDatos = "E:\Neptuno.mdb" ' Cambia esto a la ruta real donde se encuentra tu base de datos

    ' Crear un objeto Connection
    Set conexion = CreateObject("ADODB.Connection")

    ' Abrir la conexión a la base de datos Neptuno
    conexion.Open "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" & rutaBaseDatos

    ' Definir la consulta SQL para extraer datos de la tabla Productos
    consultaSQL = "SELECT * FROM Productos"

    ' Crear un objeto Recordset
    Set recordSet = CreateObject("ADODB.Recordset")

    ' Ejecutar la consulta SQL
    recordSet.Open consultaSQL, conexion

    ' Establecer la hoja de destino donde se pegarán los resultados (la primera hoja del libro)
    Set hojaDestino = ThisWorkbook.Sheets(1)
    hojaDestino.Cells.Clear ' Limpiar cualquier dato previo en la hoja

    ' Copiar los nombres de los campos (encabezados) en la primera fila de Excel
    For i = 1 To recordSet.Fields.Count
        hojaDestino.Cells(1, i).Value = recordSet.Fields(i - 1).Name
    Next i

    ' Copiar los datos desde el recordset a la hoja de Excel, empezando en la fila 2
    hojaDestino.Range("A2").CopyFromRecordset recordSet

    ' Cerrar el recordset y la conexión
    recordSet.Close
    conexion.Close

    ' Liberar las variables de objeto
    Set recordSet = Nothing
    Set conexion = Nothing

    MsgBox "Datos importados correctamente desde la base de datos Neptuno"
End Sub
```

Figura 94. Macro para conectarse con una base de datos y extraer registro.

Análisis línea por línea de la Macro ConectarNeptuno()

Esta Macro en VBA permite conectar una base de datos Access (Neptuno.mdb) con Excel, importar datos desde una tabla y pegarlos en una hoja de cálculo.

1) Definir variables

```
Dim conexion As Object
Dim recordSet As Object
Dim consultaSQL As String
Dim rutaBaseDatos As String
Dim hojaDestino As Worksheet
Dim i As Integer
```

- conexion: objeto que representa la conexión con la base de datos.
- recordSet: objeto que almacena los resultados de la consulta SQL.
- consultaSQL: variable que contiene la consulta SQL a ejecutar.
- rutaBaseDatos: ruta donde está almacenada la base de datos Access.
- hojaDestino: hoja de Excel donde se pegarán los datos.
- i: contador para recorrer los campos de la consulta.

2) Definir la ruta de la base de datos

```
rutaBaseDatos = "E:\Neptuno.mdb"
```

- Especifica la ubicación del archivo de base de datos Access (.mdb).

IMPORTANTE: esta ruta debe ajustarse según la ubicación real del archivo en el sistema.

3) Crear un objeto de conexión

```
Set conexion = CreateObject("ADODB.Connection")
```

- CreateObject("ADODB.Connection"): crea una conexión OLE DB con la base de datos Access.

ADO (ActiveX Data Objects) es un método para interactuar con bases de datos desde VBA.

4) Abrir la conexión con la base de datos

```
conexion.Open "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" &
rutaBaseDatos
```

- Provider=Microsoft.ACE.OLEDB.12.0: indica que se está usando el motor OLEDB de Access.
- Data Source=" & rutaBaseDatos: define la ruta de la base de datos.

Si la base de datos es .accdb, este proveedor sigue funcionando, pero para bases muy antiguas (.mdb), podría ser necesario usar "Provider=-Microsoft.Jet.OLEDB.4.0".

5) Definir la consulta SQL

```
consultaSQL = "SELECT * FROM Productos"
```

- Consulta SQL que selecciona todos los registros (*) de la Tabla Productos.

Se puede modificar para filtrar datos, por ejemplo:

```
consultaSQL = "SELECT NombreProducto, Precio FROM Productos
WHERE Precio > 20"
```

6) Crear un objeto Recordset

```
Set recordSet = CreateObject("ADODB.Recordset")
```

- recordSet almacena los resultados de la consulta SQL.

7) Ejecutar la consulta SQL

```
recordSet.Open consultaSQL, conexion
```

- Ejecuta la consulta SQL y guarda los resultados dentro de *recordSet*.

8) Definir la hoja destino y limpiar datos anteriores

```
Set hojaDestino = ThisWorkbook.Sheets(1)
hojaDestino.Cells.Clear
```

- ThisWorkbook.Sheets(1): usa la primera hoja del libro activo.
- hojaDestino.Cells.Clear: borra cualquier contenido previo en la hoja.

Se puede cambiar para elegir otra hoja específica, por ejemplo:

```
Set hojaDestino = ThisWorkbook.Sheets("Resultados")
```

9) Copiar encabezados (nombres de campos) a la primera fila

```
For i = 1 To recordSet.Fields.Count
   hojaDestino.Cells(1, i).Value = recordSet.Fields(i - 1).Name
Next i
```

- Recorre todos los campos de la consulta y copia sus nombres en la fila 1 de la hoja de Excel.

- recordSet.Fields(i - 1).Name: obtiene el nombre de cada columna de la tabla SQL.

Ejemplo de salida en Excel después de este paso:

A	B	C
ID	Nombre	Precio

Figura 95. Ejemplo salida ciclo que recorre encabezados.

10) Copiar datos a la hoja de Excel

```
hojaDestino.Range("A2").CopyFromRecordset recordSet
```

- Copia los datos obtenidos desde el recordSet a Excel, comenzando desde la celda A2.
- CopyFromRecordset es un método eficiente para pegar grandes volúmenes de datos.

Ejemplo de salida después de este paso:

A	B	C
ID	Nombre	Precio
1	Manzana	10.00
2	Pera	15.50
3	Naranja	20.00

Figura 96. Ejemplo de copia de datos desde base de datos a Excel.

11) Cerrar el Recordset y la conexión

```
recordSet.Close
conexion.Close
```

- Libera los recursos utilizados por la consulta SQL y cierra la conexión con Access.

12) Liberar memoria de los objetos

```
Set recordSet = Nothing
Set conexion = Nothing
```

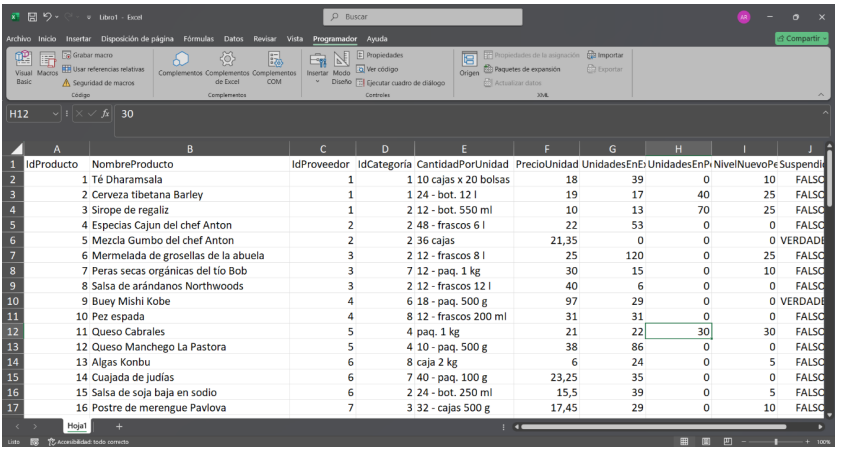
- Se liberan las variables del objeto.

13) Mensaje de confirmación

```
MsgBox "Datos importados correctamente desde la base de datos Neptuno"
```

- Muestra un mensaje indicando que los datos fueron importados correctamente.

Esta Macro en VBA permite conectarse a una base de datos Access, ejecutar una consulta SQL y exportar los datos a una hoja de Excel de manera eficiente.



	A	B	C	D	E	F	G	H	I	J
1	IdProducto	NombreProducto	IdProveedor	IdCategoría	CantidadPorUnidad	PrecioUnidad	UnidadesEnEs	UnidadesEnPe	NivelNuevoPe	Suspendid
2	1	Té Dharamsala	1	1	10 cajas x 20 bolsas	18	39	0	10	FALSO
3	2	Cerveza tibetana Barley	1	1	24 - bot. 12 l	19	17	40	25	FALSO
4	3	Sirope de regaliz	1	2	12 - bot. 550 ml	10	13	70	25	FALSO
5	4	Especias Cajun del chef Anton	2	2	48 - frascos 6 l	22	53	0	0	FALSO
6	5	Mezcla Gumbo del chef Anton	2	2	36 cajas	21,35	0	0	0	VERDADE
7	6	Mermelada de grosellas de la abuela	3	2	12 - frascos 8 l	25	120	0	25	FALSO
8	7	Peras secas orgánicas del tío Bob	3	7	12 - paq. 1 kg	30	15	0	10	FALSO
9	8	Salsa de arándanos Northwoods	3	2	12 - frascos 12 l	40	6	0	0	FALSO
10	9	Buey Mishi Kobe	4	6	18 - paq. 500 g	97	29	0	0	VERDADE
11	10	Pez espada	4	8	12 - frascos 200 ml	31	31	0	0	FALSO
12	11	Queso Cabrales	5	4	paq. 1 kg	21	22	30	30	FALSO
13	12	Queso Manchego La Pastora	5	4	10 - paq. 500 g	38	86	0	0	FALSO
14	13	Algas Konbu	6	8	caja 2 kg	6	24	0	5	FALSO
15	14	Cuajada de judías	6	7	40 - paq. 100 g	23,25	35	0	0	FALSO
16	15	Salsa de soja baja en sodio	6	2	24 - bot. 250 ml	15,5	39	0	5	FALSO
17	16	Postre de merengue Pavlova	7	3	32 - cajas 500 g	17,45	29	0	10	FALSO

Figura 97. Resultado de la ejecución de la Macro que consulta la tabla Productos.

Apéndice A

Desafíos para desarrollar con diagramas de flujo

Desafíos con algoritmos lineales

Desafío A1

Mediante el uso de un diagrama de flujo, construya un algoritmo que permita leer dos valores, sumarlos e indicar el resultado de la operación.

PRUEBAS A REALIZAR		
Valor 1	**Valor 2**	**Resultado**
3	4	La suma de 3 más 4 es 7
-11	-3	La suma de -11 más -3 es -14
8	3	La suma de 8 más 3 es 11
-1	4	La suma de -1 más 4 es 3

Desafío A2

Mediante el uso de un diagrama de flujo, construya un algoritmo que permita calcular el área de una circunferencia e indicar el resultado de la operación.

Consideraciones/restricciones:

- Considere PI con el valor 3,1416 o utilice la constante PI de Flowgorithm

PRUEBAS A REALIZAR	
Radio	**Resultado**
4	El área de una circunferencia de radio 4 es 50.2656
2	El área de una circunferencia de radio 2 es 12.5664

Desafío A3

Construya un algoritmo que permita convertir de grados Celsius a Fahrenheit utilizando la siguiente fórmula:

$$Fahrenheit = \frac{Celsius \times 9}{5} + 32$$

PRUEBAS A REALIZAR	
Grados Celsius	**Grados Fahrenheit**
32	89,6
0	32
12	53,6
-10	14
-15	5
-17,7777	0,00014
50	122

Desafío A4

Construya un algoritmo que cálculo la edad a partir del año de nacimiento y el año actual.

PRUEBAS A REALIZAR		
Año nacimiento	**Año actual**	**Resultado**
2000	2024	Edad: 24 años
2005	2024	Edad: 19 años
1999	2024	Edad: 25 años
1988	2024	Edad: 36 años

Desafío A5

Construya un algoritmo que permita calcular el área de un rectángulo.

$$\textit{Área Rectángulo} = \textit{Langitud} \times \textit{Ancho}$$

PRUEBAS A REALIZAR		
Longitud	**Ancho**	**Área**
Centímetros	**Centímetros**	**Centímetros**
13	2	26
12	5	60
13	4	52
23	8	184
25	2	50
25	2	50
2	6	48

Desafío A6

Construya un algoritmo que calcule el promedio de tres números.

PRUEBAS A REALIZAR			
Número 1	**Número 2**	**Número 3**	**Promedio**
2	1	4	2,33333333
6	7	3	5,33333333
6	3	2	3,66666667
5	6	4	5
6	6	5	5,66666667
3	7	7	5,66666667
6	1	1	2,66666667

Desafío A7

Construya un algoritmo que permita calcular el perímetro de un triángulo.

PRUEBAS A REALIZAR			
Lado a	**Lado b**	**Lado c**	**Perímetro**
4	6	2	12
10	6	8	24
2	9	11	22
6	1	4	11
2	13	13	28
15	1	10	26
11	5	12	28

Desafío A8

Construya un algoritmo que convierta una cantidad de horas en segundos.

PRUEBAS A REALIZAR	
Horas	**Segundos**
14	50.400
1	3.600
6	21.600
2	7.200
2	7.200
4	14.400
12	43.200

Desafío 9

Construya un algoritmo que, a partir de una cantidad y precio, calcule el precio total de una compra. El resultado debe mostrar valor neto, IVA y total.

PRUEBAS A REALIZAR		
Cantidad	**Precio**	**Resultado**
1	1.103	Neto:1103 IVA: 209 Total:1312
5	408	Neto:2040 IVA: 387 Total:2427
1	999	Neto: 999 IVA: 189 Total: 1188
13	1.185	Neto: 15405 IVA: 2926 Total: 18331
15	215	Neto: 3225 IVA: 612 Total: 3837
12	85	Neto: 1020 IVA: 193 Total: 1213
15	1.040	Neto: 15600 IVA: 2964 Total: 18564

Desafío A10

Construya un algoritmo que, a partir de una cantidad, un precio y un porcentaje de descuento, calcule el neto, IVA y total de la compra. Todos los valores deben calcularse sin decimales.

PRUEBAS A REALIZAR			
Cantidad	**Precio**	**Descuento**	**Resultado**
15	1.191	10%	Neto: 17865 Descuento: 1786 IVA: 3055 Total: 19134
10	937	13%	Neto: 9370 Descuento: 1218 IVA: 1548 Total: 9700
4	190	11%	Neto: 760 Descuento: 83 IVA: 128 Total: 805
3	436	4%	Neto: 1308 Descuento: 52 IVA: 238 Total: 1494
2	143	6%	Neto: 286 Descuento: 17 IVA: 51 Total: 320
2	704	1%	Neto: 1408 Descuento: 14 IVA: 264 Total: 1658
4	832	2%	Neto: 3328 Descuento: 66 IVA: 619 Total: 3881

Desafío A11

Construya un algoritmo que convierte una distancia en millas a kilómetros.

PRUEBAS A REALIZAR	
Miles	**Kilómetros**
359	577,75306
1626	2616,78684
515	828,8101
701	1128,14734
746	1200,56764
1790	2880,7186
1211	1948,91074

Desafío A12

Construya un algoritmo que utilice el teorema de Pitágoras para calcular la hipotenusa de un triángulo rectángulo dado los valores de los catetos.

PRUEBAS A REALIZAR		
Cateto a	**Cateto b**	**Hipotenusa c**
17	17	24,04163
14	14	19,79899
3	5	5,83095
15	17	22,67157
15	9	17,49286
10	6	11,66190
8	9	12,04159

Desafíos con algoritmos condicionales (bifurcación)

Desafío A13

Construya un algoritmo que lea dos valores y los compare. Si ambos valores son iguales indicar "los números son iguales". Si son distintos indicar, por ejemplo, "4 es mayor que 3".

PRUEBAS A REALIZAR		
Valor 1	**Valor 2**	**Resultado**
3	4	4 es mayor que 3
3	2	3 es mayor que 2
8	3	8 es mayor que 3
5	7	5 es mayor que 7
0	3	0 es mayor que 3
8	-3	8 es mayor que -3
6	6	Los números son iguales

Desafío A14

Desarrolle un diagrama de flujo que refleje un algoritmo que permita leer tres valores y compararlos. Si los tres valores son iguales indicar "Los tres valores son iguales", si existe más de un valor distinto entre ellos, indicar cuál es el mayor, por ejemplo, "El valor 6 es el mayor".

PRUEBAS A REALIZAR			
Valor 1	**Valor 2**	**Valor 3**	**Resultado**
8	3	9	El valor 9 es el mayor
1	2	3	El valor 3 es el mayor
5	4	2	El valor 5 es el mayor
8	8	8	Los tres valores son iguales

5	5	4	El valor 5 es el mayor
9	4	9	El valor 9 es el mayor
4	6	6	El valor 6 es el mayor

Desafío A15

Desarrolle un algoritmo que permita leer tres valores desordenados. Si los tres valores son iguales indicar "Todos valores son iguales" y si son distintos, ordenarlos de menor a mayor y mostrar la secuencia ordenada.

PRUEBAS A REALIZAR			
Valor 1	**Valor 2**	**Valor 3**	**Resultado**
1	2	3	El valor del resultado es 1 2 3
8	7	6	El valor del resultado es 6 7 8
6	4	5	El valor del resultado es 4 5 6
3	8	4	El valor del resultado es 3 4 8
5	5	5	Todos los números son iguales
6	2	2	El valor del resultado es 2 2 6
3	9	3	El valor del resultado es 3 3 9

Desafío A16

Construya un algoritmo que lea las longitudes de los tres lados de un triángulo y determinar si es equilátero, isósceles o escaleno.

PRUEBAS A REALIZAR			
Lado a	**Lado b**	**Lado c**	**Resultado**
14	10	25	Escaleno
12	12	12	Equilátero
106	17	57	Escaleno
8	8	25	Isóceles
40	61	22	Escaleno

11	21	33	Escaleno
15	25	15	Isóceles
60	30	30	Isóceles

Desafío A17

Desarrolle un algoritmo que, ocupando el teorema de Pitágoras, calcule uno de los lados (hipotenusa o uno de los catetos) de un triángulo rectángulo cualesquiera. El resultado debe indicar el lado (hipotenusa, cateto opuesto o cateto adyacente) y valor calculado. Si existe más de un valor en cero debe indicar un error. Si existe uno o más valores negativos, debe indicar un error.

Consideraciones/restricciones

- Si existe más de un valor en cero debe indicar un error.
- Si existe uno o más valores negativos, debe indicar un error.
- Utilice la función SQRT() para calcular la raíz cuadrada de un número.
- La operación de exponenciación se indica por el símbolo ^, ejemplo 2^4 que indica dos elevado a cuatro.
- La raíz cuadrada de un número negativo no tiene solución real.
- Ingrese un valor 0 para indicar el lado que desea calcular.

PRUEBAS A REALIZAR			
a	**b**	**h**	**Resultado**
3	4	0	El cateto opuesto mide 3, el cateto adyacente mide 4 y la hipotenusa mide 5
5	0	7	El cateto opuesto mide 5, el cateto adyacente mide 4.898979486 y la hipotenusa mide 7
9	13	0	El cateto opuesto mide 9, el cateto adyacente mide 13 y la hipotenusa mide 15.8113883
0	9	18	El cateto opuesto mide 15.58845727, el cateto adyacente mide 9 y la hipotenusa mide 18

0	0	0	Los tres lados del triángulo son cero, no se puede realizar el cálculo
12	4	18	Debe indicar un lado del triángulo en cero para realizar el cálculo
0	0	18	Error, existen dos lados del triángulo con valor cero
5	0	0	Error, existen dos lados del triángulo con valor cero
0	4	0	Error, existen dos lados del triángulo con valor cero
4	2	0	El cateto opuesto mide 4, el cateto adyacente mide 2 y la hipotenusa mide 4.472135955
13	0	4	Error, no se puede calcular el cateto adyacente
0	12	4	Error, no se puede calcular el cateto opuesto

Desafío A18

Desarrolle un algoritmo que haga las cuatro operaciones aritméticas básicas (suma, resta, multiplicación y división), para dos términos numéricos ingresados por el usuario.
Los términos numéricos pueden ser números reales y la operación se debe ingresar como: + (suma), - (sustracción), / (división) o * (multiplicación), otro símbolo debe ser indicado como un error.

Desafío A19

Construya un algoritmo que muestre el cociente y residuo de una división entera. Utilice la función MÓDULO para obtener el residuo. Recuerde, la división por cero es indeterminada, ese caso es un error.

PRUEBAS A REALIZAR		
Numerador	**Denominador**	**Resultado**
10	3	Cociente: 3 Residuo: 1
55	7	Cociente: 7 Residuo: 6
20	2	Cociente: 10 Residuo: 0

11	5	Cociente: 2 Residuo: 1
12	0	Error, división por cero

Desafío A20

Construya un algoritmo que acepte un año y verifique si ese año es bisiesto o no.

Un año es bisiesto si:

a) Es divisible por 4.
b) No es divisible por 100, a menos que también sea divisible por 400.

PRUEBAS A REALIZAR	
Años	**Resultado**
2024	Bisiesto
1900	No bisiesto
2020	Bisiesto
2012	Bisiesto
2000	Bisiesto
2023	No bisiesto
2200	No bisiesto
2400	Bisiesto
2022	No bisiesto
2040	Bisiesto
2028	Bisiesto
1600	Bisiesto
2300	No bisiesto
2100	No bisiesto
2016	Bisiesto
2021	No bisiesto

PRUEBAS A REALIZAR	
Años	**Resultado**
1800	No bisiesto

Desafío A21

Construya un algoritmo que muestre si un número es par o impar. Un número es par si es divisible por 2, de lo contrario es impar.

PRUEBAS A REALIZAR	
Número	**Resultado**
20	El número 20 es par
3	El número 3 es impar
20	El número 20 es par
17	El número 17 es impar
8	El número 8 es par
11	El número 11 es impar
9	El número 9 es impar

Desafío A22

Construya un algoritmo que muestre el valor futuro de un crédito calculado con tasa de interés compuesta. Considere que: el valor presente, ni el plazo, ni la tasa de interés pueden tomar valores cero y/o negativos. La tasa se expresa como un número, ejemplo: 1.5 y está en correspondencia al plazo.

PRUEBAS A REALIZAR			
VP	**n**	**i**	**Resultado**
10	3	10	Para un crédito de: 10 a una tasa de interés de 10% y un plazo de 3 periodos, el valor futuro es 13,31

3.500.000	24	0,16	Para un crédito de: 3.500.000 a una tasa de interés de 0,16% y un plazo de 24 periodos, el valor futuro es 3.636.902,22
1.850.000	12	0,15	Para un crédito de: 1.850.000 a una tasa de interés de 0,15% y un plazo de 12 periodos, el valor futuro es 1.883.576,1
500.000	18	0,2	Para un crédito de: 500.000 a una tasa de interés de 0,2% y un plazo de 18 periodos, el valor futuro es 518.309,289
0	6	0,1	Error en el valor presente, no puede ser <=0
456.356	0	0,3	Error en el plazo, no puede ser <=0
346.321	8	0	Error en el interés, no puede ser <=0

Desafío A23

Construya un algoritmo que permita obtener el monto a pagar por estacionamiento según los minutos consumidos.

Consideraciones/restricciones

- Los minutos entre 1 y menores a 60 tienen un costo de $23,5 /min
- Los minutos entre 60 y menores a 120 tienen un costo de 22,0 /min
- Los minutos entre 120 y más tienen un costo de $20,5 /min
- Se debe cobrar los minutos entre 1 y 480. Los minutos sobre 480 queda exento de pago

PRUEBAS A REALIZAR	
Min.	**Resultado**
60	Total de minutos: 60, monto a pagar $1.408,5
180	Total de minutos: 180, monto a pagar $3.957
540	Total de minutos: 540, monto a pagar $10.107
45	Total de minutos: 45, monto a pagar $1.057,5

Desafío A24

Construya un algoritmo que resuelva una ecuación cuadrática e indique cuántas soluciones tiene:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Consideraciones/restricciones

La salida del proceso debe indicar:

- La ecuación tiene dos soluciones reales distintas si su discriminante es positiva.
- La ecuación tiene una solución real si su discriminante es cero.
- La ecuación no tiene soluciones en los reales si su discriminante es negativa.
- El discriminante es la parte de la fórmula cuadrática bajo la raíz cuadrada.
- Si a = 0 la ecuación no es cuadrática, sino lineal.
- Si a = 0 y b = 0, la ecuación no tiene solución (a menos que c = 0).

$$b^2 - 4ac$$

PRUEBAS A REALIZAR			
a	**b**	**c**	**Resultado**
6	10	-1	La ecuación cuadrática tiene dos soluciones reales distintas
3	24	48	La ecuación cuadrática tiene una solución real
1	12	-64	La ecuación cuadrática tiene dos soluciones reales distintas
2	-25	144	La ecuación cuadrática no tiene solución en los reales
0	12	24	La ecuación no es cuadrática, sino lineal
0	0	12	La ecuación cuadrática no tiene solución

Desafíos con algoritmos iterativos (bucles)

Un bucle o ciclo, en programación, es una secuencia de instrucciones de código que se ejecuta repetidas veces, hasta que la condición de finalización o término del ciclo se cumpla. Utilizando las funciones CICLO PARA y/o CICLO MIENTRAS:

Desafío A25

Construya un algoritmo que permita sumar uno o más números hasta que se ingrese un 0, para el desarrollo de este ejercicio ocupe la figura "ciclo mientras".

Consideraciones/restricciones

- Se pueden ingresar n números al proceso siendo n, un valor indeterminado al inicio del proceso.
- El proceso debe operar correctamente sobre valores positivos o negativos.
- Para finalizar el proceso debe ingresar un valor cero.
- Al finalizar el proceso debe indicar cuál es el resultado de la suma.

PRUEBAS A REALIZAR	
Valores	**Resultado**
1, 2, 3, 4, 5, 0	------> El resultado es: 15 <--------
-10, -9, -8, -7, -6, 0	-------> El resultado es: -40 <--------
8, 9, 10, 11, 12, 13, 14, 15, 0	-------> El resultado es: 92 <--------
-2, -1, 1, 2, 3, 4, 5, 0	-------> El resultado es: 12 <--------
0	-------> El resultado es: 0 <--------
5, 4, 3, 2, ,1, -1, -2, 0	-------> El resultado es: 12 <--------

Desafío A26

Construya un algoritmo que permita sumar los números enteros (Z) que existen en un rango. Para el proceso utilice el *ciclo para*.

Consideraciones/restricciones

- El rango puede darse en cualquier orden, de menor a mayor o de mayor a menor.
- El proceso debe operar correctamente sobre valores positivos y/o negativos.
- Al finalizar el proceso debe indicar cuál es el resultado de la suma.

PRUEBAS A REALIZAR	
Rango	**Resultado**
1 → 5	Suma del rango es: 15
-10 → - 6	Suma del rango es: -40
8 → 15	Suma del rango es: 92
-2 → 5	Suma del rango es: 12
0 → 1	Suma del rango es: 1
5 → -2	Suma del rango es: 12

Desafío A27

Construya un algoritmo que permita calcular el factorial de un número, para el desarrollo del ejercicio utilice la figura "*ciclo para*". El proceso debe respetar las restricciones para el cálculo del factorial.

Consideraciones/restricciones

- El factorial se define sobre el conjunto de los números naturales.
- El factorial de 0 o 1 es 1.

PRUEBAS A REALIZAR	
Factorial	**Resultado**
1	Factorial de 1 es 1
0	Factorial de 0 es 1
5	Factorial de 5 es 120
7	Factorial de 7 es 5040
10	Factorial de 10 es 3628800

Desafío A28

Construya un algoritmo que permita sumar los múltiplos de un número, que existen en un rango de números naturales, por ejemplo: los múltiplos de 3 en un rango de 1 a 10 son 3, 6, 9 y suman 18.

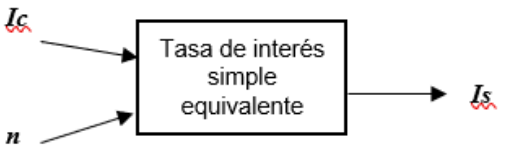
Consideraciones/restricciones

- El rango y múltiplo deben ser números naturales.
- El rango puede comenzar en cualquier número.
- Utilice la función MOD (Módulo de un número) para determinar si es múltiplo o no.

PRUEBAS A REALIZAR			
Inicio rango	**Termino rango**	**Múltiplo**	**Resultado**
1	10	3	La suma de los múltiplos de 3 en el rango de 1 a 10 es: 18
5	1	2	La suma de los múltiplos de 2 en el rango de 1 a 5 es: 6
12	36	4	La suma de los múltiplos de 4 en el rango de 12 a 36 es: 168

Desafío A29

Desarrollar un diagrama de flujo que contenga un algoritmo para calcular la tasa de interés simple equivalente, a partir de una tasa de interés compuesta y un plazo dado.



Antecedentes

- ***Ic***: tasa de interés compuesto equivalente
- ***n***: número de flujos (plazo)
- ***Is***: tasa de interés simple

Consideraciones/restricciones

- La tasa de interés compuesta debe ingresar como un número, sin incluir el símbolo porcentaje.
- Los valores de entrada debe ser números >= 0. Si no cumple esa condición se debe mostrar un mensaje de error.
- La tasa simple equivalente debe estar expresada con un decimal.
- La ecuación que debe utilizar para el cálculo es la siguiente:

$$i_c = \left((1 + n \times i_s)^{\frac{1}{n}} - 1\right) \times 100$$

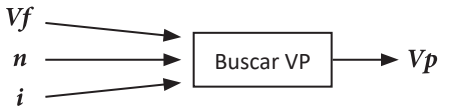
- No puede alterar o despejar la ecuación, debe iterar sobre la ecuación descrita.
- La función debe contener a lo menos un bucle para realizar el cálculo.

Existe una tasa simple (***Is***, desconocido) que es equivalente para una tasa de interés compuesta (***Ic***), en un plazo de ***n*** periodos:

PRUEBAS A REALIZAR		
Ic	**n**	**Is**
5,9	6	6,9
3,7	12	4,6
2,5	24	3,4
3,1	18	4,1
2,8	10	3,2

Desafío A30

Desarrollar un diagrama de flujo que represente el algoritmo para determinar el Valor presente (***Vp***) que invertido a una tasa de interés (***i***) durante un plazo (***n***) alcanza un Valor futuro (***Vf***).



Antecedentes

- ***Vp***: corresponde al valor presente que se debe encontrar.
- ***Vf***: es el valor futuro que se quiere alcanzar (conocido), por ejemplo $3.287.856.
- ***n***: es el plazo al que se quiere invertir (conocido), por ejemplo 24 meses.
- ***i***: corresponde a la tasa de interés que se ha conseguido (conocida), por ejemplo 1,5% mensual.

Consideraciones/restricciones

- La tasa de interés debe ser un número sin símbolo porcentual.
- Los valores que se utilizaran para la prueba van desde $1 hasta $100.000.000.
- Ninguno de los valores de entrada debe ser <= 0, si eso ocurre debe indicar un error.
- La expresión que debe utilizar para el cálculo es la siguiente:

$$Vf = Vp \times \left(1 + \frac{i}{100}\right)^{n}$$

- No puede alterar o despejar la expresión, debe iterar sobre la expresión descrita y para ello, debe utilizar bucle.

Existe un valor presente (***Vp***, desconocido) que invertido a una tasa de interés ***i***, en un plazo ***n***, genera un retorno ***Vf***:

PRUEBAS A REALIZAR			
i	**n**	**vf**	**Resultado**
1,5	24	365.508	255689 depositado al 1,5% durante 24 meses, genera un valor futuro de 365508
0,9	12	210.825	189334 depositado al 0,9% durante 12 meses, genera un valor futuro de 210825
1,2	18	482.690	389421 depositado al 1,2% durante 18 meses, genera un valor futuro de 482690
0,6	12	81.536	75889 depositado al 0,6% durante 12 meses, genera un valor futuro de 81536

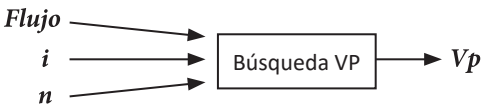
Desafío A31

Supongamos que enfrentamos una **restricción de nuestro flujo de caja** disponible para el pago de un crédito que necesitamos solicitar. El banco nos ofrece una tasa de interés mensual ***i*** y un plazo máximo de ***n*** periodos.

Es crucial evaluar las diferentes combinaciones de montos de crédito que se ajusten a nuestras restricciones de flujo de caja, las cuales están afectadas por la demora en los pagos de nuestros clientes.

Dado que cuentas con habilidades para generar algoritmos, te solicitamos ayuda para desarrollar un diagrama de flujo que contenga un algoritmo para calcular el valor presente (***VP***) del préstamo que podríamos solicitar, considerando la tasa de interés (***i***), el plazo (***n***) ofrecidos por el banco y nuestras **limitaciones de flujo de caja**.

Este algoritmo nos permitirá determinar el monto máximo del préstamo que podemos asumir sin comprometer nuestra capacidad de pago mensual, dada la incertidumbre en los cobros de nuestros clientes. Este algoritmo facilita la evaluación de nuestras opciones de financiamiento, asegurando que el crédito solicitado sea viable dentro de nuestras restricciones financieras.



Antecedentes

- ***VP*** → valor presente
- ***i*** → tasa de interés mensual
- ***n*** → números de periodos de pago
- ***Flujo*** → valor del flujo

Consideraciones/restricciones

- La tasa de interés debe ser un número sin símbolo porcentual.
- Ningún valor de entrada debe ser <= 0, si eso ocurre debe indicar un error.
- La ecuación que debe utilizar para el cálculo es la siguiente ecuación:

$$Flujo = \frac{Vp \times i}{1 - (1 + i)^{-n}}$$

- No está permitido alterar o despejar la ecuación, debe iterar sobre la expresión descrita. Para ello debe considerar el uso de bucle.

PRUEBAS A REALIZAR			
i	**n**	**Flujo**	**Resultado**
1,55	18	32.046	Un crédito de 499999 a una tasa de interés 1,55% durante 18 meses, exige un flujo mensual de 32046
0,5	12	42.919	Un crédito de 498673 a una tasa de interés 0,5% durante 12 meses, exige un flujo mensual de 42919
1,8	36	45.580	Un crédito de 1199985 a una tasa de interés 1,8% durante 36 meses, exige un flujo mensual de 45580
1,23	24	43.535	Un crédito de 8999992 a una tasa de interés 1,23% durante 24 meses, exige un flujo mensual de 43535

Desafío A32

Desafíos con algoritmos iterativos (bucles) y arreglos

La moda es una medida de tendencia central que representa el valor o los valores que aparecen con mayor frecuencia en un conjunto de datos. Es especialmente útil para datos categóricos y discretos, ya que identifica el elemento más común en el conjunto.
Existen varios "tipos" de moda:

- Unimodal: si solo hay un valor que se repite con mayor frecuencia, se dice que el conjunto de datos es unimodal.
- Bimodal: si hay dos valores que tienen la misma frecuencia más alta, el conjunto se considera bimodal.
- Multimodal: si más de dos valores tienen la misma frecuencia más alta, el conjunto se clasifica como multimodal.
- Sin moda: si todos los valores aparecen con la misma frecuencia, el conjunto se considera sin moda.

Problema: encontrar la moda de un conjunto de n números.

Dado un conjunto de *n* números enteros, construya algoritmo que, utilizando bucle, encuentre la moda.

Salida esperada: debe indicar la moda y el tipo de moda que representa, ejemplos:

Para los datos 3, 5, 5, 8, 9, la moda es el valor 5, ya que es el único valor que se repite con mayor frecuencia (dos veces). En este caso, el conjunto de datos es unimodal, ya que tiene una sola moda.

- Salida: **la moda es 5. Es de tipo unimodal**
- Para los datos: 3, 5, 5, 8, 8, 9
- Salida: **la moda es 5 y 8. Es de tipo bimodal**
- Para los datos: 3, 5, 5, 7, 7, 8, 8, 9
- Salida: **la moda es 5,7 y 8. Es de tipo multimodal**
- Para los datos: 2, 4, 6, 8, 10
- Salida: **no tiene moda. Es de tipo sin moda**

Requerimientos

- Los números de entrada deben ser naturales. Si no cumple esa condición debe indicar un error.

Sugerencias

- Utilice arreglos (vectores/matrices) para almacenar los datos.
- Utilice bucle para almacenar o recorrer los arreglos.
- Al comienzo del proceso, pida la cantidad de números que conforman el conjunto de números, eso le permitirá conformar el arreglo.

Desafío A33

En estadística, existen varios tipos de frecuencia que se utilizan para describir la distribución de un conjunto de datos. La frecuencia absoluta (*fi*) es el número de veces que un valor específico aparece en un conjun-

to de datos. Por ejemplo, si el número 5 aparece 3 veces en un conjunto de datos, su frecuencia absoluta es 3.

Problema: calcular la frecuencia absoluta de un conjunto de números.

Supongamos que tienes un conjunto de n números, por ejemplo: 4, 6, 4, 7, 4, 6. Usted debe encontrar la frecuencia absoluta de cada número. Recuerde que la frecuencia absoluta representa la cantidad de veces que aparece cada número en el conjunto.

Ejemplo: conjunto de datos: [4, 6, 4, 7, 4, 6]

- Salida esperada: frecuencias: [4:3, 6:2, 7:1]. La salida puede ser horizontal, como se muestra, o puede ser vertical como una lista de pares, elija usted.
- La salida debe indicar el número y su frecuencia.

Requerimientos

- Los números de entrada deben ser naturales. Si no cumple esa condición debe indicar un error.

Sugerencias

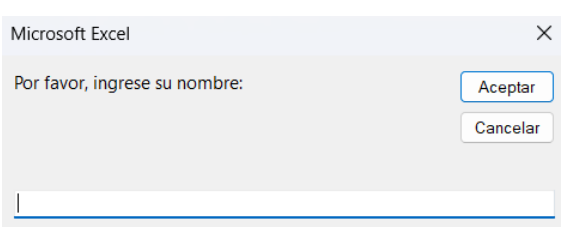
- Utilice arreglos (vectores/matrices) para guardar los datos.
- Utilice bucle para almacenar o recorrer los elementos del arreglo.
- Al comienzo del proceso, pida la cantidad de números que conforman el conjunto de números, eso le permitirá conformar el arreglo.

Apéndice B

Desafíos de subprocesos en VBA

Desafío B1

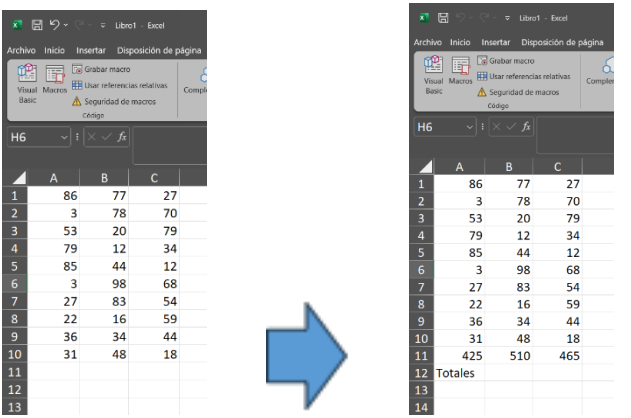
Desarrollar una Macro que solicite al usuario su nombre usando un InputBox y luego le dé la bienvenida con un mensaje personalizado en un MsgBox.



Solicitud de nombre Resultado

Desafío B2

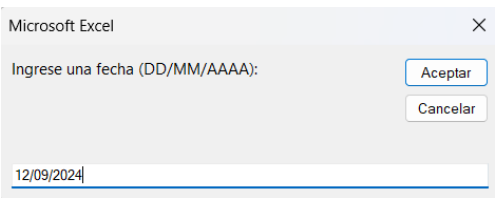
Desarrollar una Macro que cree un informe de totales. La Macro debe sumar las filas 1 a 10 de las columnas A, B, y C y colocar los totales en las celdas A11, B11, y C11 respectivamente.



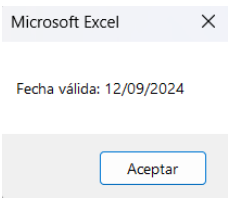
	A	B	C
1	86	77	27
2	3	78	70
3	53	20	79
4	79	12	34
5	85	44	12
6	3	98	68
7	27	83	54
8	22	16	59
9	36	34	44
10	31	48	18
11	425	510	465
12	Totales		

Desafío B3

Desarrolle una Macro que solicite al usuario ingresar una fecha a través de un InputBox y luego valide si la fecha es válida. Si la fecha es válida, debe mostrar un mensaje de confirmación. Si no lo es, debe mostrar un mensaje de error y solicitar nuevamente la fecha.



Solicitud de fecha



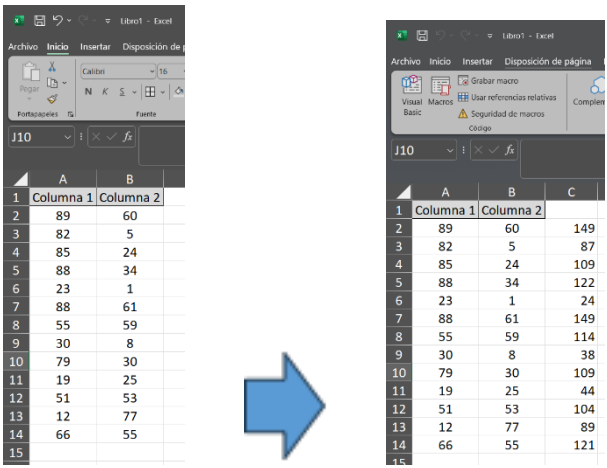
Resultado

Este ejercicio permite practicar:

- InputBox para solicitar la fecha.
- Validar la fecha utilizando la función nativa IsDate().
- Utilizar MsgBox para indicar si la fecha es válida o inválida.
- Bucle mientras la fecha sea incorrecta.

Desafío B4

Desarrolle una Macro en VBA que sume los valores de las columnas A y B fila por fila, y coloque el resultado en la columna C. La Macro debe comenzar en la fila 2 y continuar sumando hasta la última fila con datos en las columnas A y B; verificar si las columnas contienen valores numéricos y manejar adecuadamente las celdas vacías o con datos no numéricos colocando un mensaje en la columna C.



	A	B
1	Columna 1	Columna 2
2	89	60
3	82	5
4	85	24
5	88	34
6	23	1
7	88	61
8	55	59
9	30	8
10	79	30
11	19	25
12	51	53
13	12	77
14	66	55
15		

	A	B	C
1	Columna 1	Columna 2	
2	89	60	149
3	82	5	87
4	85	24	109
5	88	34	122
6	23	1	24
7	88	61	149
8	55	59	114
9	30	8	38
10	79	30	109
11	19	25	44
12	51	53	104
13	12	77	89
14	66	55	121
15			

Este ejercicio permite practicar:

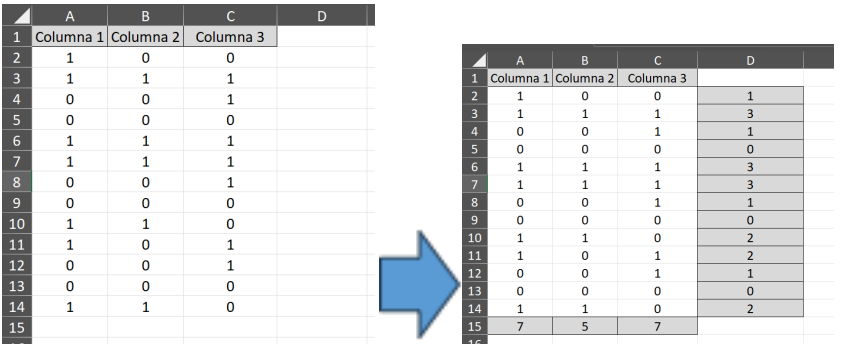
- Application.WorksheetFunction.Max() es una función de Excel que nos servirá para localizar la última fila con datos.
- Bucle For para recorrer filas.
- Manipulación de celdas con la función Cells de VBA.
- Validación de datos usando la función IsNumeric.
- Automatización de tareas repetitivas en Excel VBA.

Desafío adicional: permitir que la Macro maneje celdas vacías, considerando que, si una celda está vacía, se debe tratar como un valor 0 en la suma.

Desafío B5

Desarrolle una Macro en VBA que cuente la frecuencia del número 1 en cada fila y en cada columna dentro del rango que abarca desde la columna A hasta la columna C y desde la fila 2 hasta la fila 14. Los resultados deben mostrarse en la columna D para las filas y en la fila 15 para las columnas.

- Por filas: contar cuántas veces aparece el número 1 en las celdas de las columnas A a C en cada fila y colocar el resultado en la columna D.
- Por columnas: contar cuántas veces aparece el número 1 en las celdas de las filas 2 a 14 para cada columna y colocar el resultado en la fila 15.



	A	B	C	D
1	Columna 1	Columna 2	Columna 3	
2	1	0	0	
3	1	1	1	
4	0	0	1	
5	0	0	0	
6	1	1	1	
7	1	1	1	
8	0	0	1	
9	0	0	0	
10	1	1	0	
11	1	0	1	
12	0	0	1	
13	0	0	0	
14	1	1	0	
15				

	A	B	C	D
1	Columna 1	Columna 2	Columna 3	
2	1	0	0	1
3	1	1	1	3
4	0	0	1	1
5	0	0	0	0
6	1	1	1	3
7	1	1	1	3
8	0	0	1	1
9	0	0	0	0
10	1	1	0	2
11	1	0	1	2
12	0	0	1	1
13	0	0	0	0
14	1	1	0	2
15	7	5	7	

Este ejercicio permite practicar:

- Recorrer rangos específicos en Excel usando bucles For.
- Contar valores específicos en celdas de Excel.
- Mostrar resultados en otras celdas (como en una columna o fila adicional).

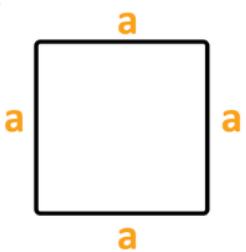
Desafío B6

Diseñe un proceso para desarrollar una tabla de amortización a partir de los datos que deberías ingresar, como el valor del préstamo, el plazo y la tasa de interés. Este ejercicio requiere el uso de Macros en VBA para automatizar el cálculo de la amortización del préstamo y generar la tabla en función de los datos ingresados.

Apéndice C
Desafíos con funciones propias en VBA

Desafío C1

Cree una función propia que permita calcular el área de un cuadrado perfecto.

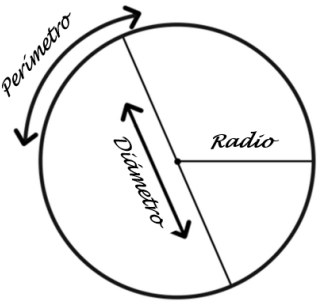


Desafío C2

Construya una función propia en VBA que permita realizar la conversión de grados a radianes, ejemplo: 180° -> 3.14159 radianes (aproximado).

Desafío C3

Cree una función propia que permita calcular el perímetro de una circunferencia de radio ***r***.



Desafío C4

Cree una función propia que permita calcular el valor futuro ***VF*** de una inversión ***VP*** que se deposita a una tasa de interés ***i*** en un plazo ***n***.

Desafío C5

Cree una función propia que permita calcular la varianza de un conjunto de tres números cualquiera.

$$Varianza = \frac{(a-\bar{x})^2 + (a-\bar{x})^2 + (a-\bar{x})^2}{3}$$

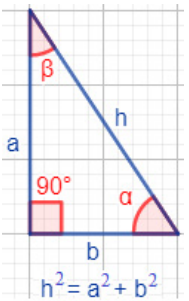
Desafío C6

Cree una función propia que permita calcular la desviación estándar de tres números cualquiera.

NOTA: desde el desafío 7 en adelante necesitará aplicar bifurcaciones.

Desafío C7

Construya una función propia en VB que determine la hipotenusa de un triángulo rectángulo ocupando la siguiente ecuación:



Observaciones/restricciones

- Para asegurar que la función ejecute correctamente y sin errores, debe validar que los catetos sean distintos de cero y positivos. En caso de que algún valor no cumpla las condiciones debe entregar el mensaje "Error en parámetros" y detener la ejecución.

Desafío C8. Cálculo de un lado de un triángulo rectángulo utilizando el teorema de Pitágoras

Basándonos en el ejercicio anterior, construya una función propia en VB que determine el valor del cateto ***a***, valor del cateto ***b*** o valor de la hipotenusa. El valor a calcular se reconoce porque contiene un valor cero, pero ejemplo:

a	b	h
0	5	6
4	6	0
7	0	8

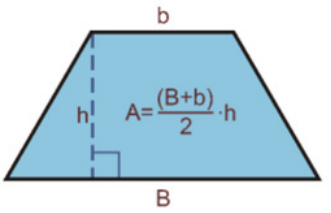
Resultado		
a	b	h
3,317		
		7,211
	3,873	

Observaciones/restricciones

- El único valor que puede ser cero es el lado que se desea calcular.
- Para asegurar que la función ejecute correctamente y sin errores, debe validar que los solo un lado del triángulo sea cero, el resto deben ser mayores a cero. En caso de que algún valor no cumpla las condiciones debe entregar el mensaje "Error en parámetros" y detener la ejecución.
- La salida debe ser solo el valor calculado, sin texto.

Desafío C9. Cálculo del área de un trapecio

Construya una función propia que permita calcular el ÁREA de cualquier TRAPECIO. Considerando los datos descritos a continuación, deduzca la(s) entrada(s), el proceso y la salida de la función propia.



$$\text{Área} = \frac{B + b}{2} \times h$$

$$\text{Perímetro} = B + b + L + L$$

$$\text{Área} = \frac{8 + 6}{2} \times 4 = 28\ \text{cm}^2$$

$$\text{Perímetro} = 8 + 6 + 5 + 5 = 24\ \text{cm}$$

Desafío C10. Cálculo del cuadrado de un binomio

Construya una función propia que permita calcular el cuadrado de un binomio.

$$(a \pm b)^2 = a^2 \pm 2ab + b^2$$

Desafío C11. Determinar el mayor entre dos valores

Construya una función propia, en VBA, que permita determinar el número mayor entre dos valores.

Observaciones/restricciones

- No está permitidos utilizar funciones nativas.

Desafío C12

Construya una función propia, en VBA, que permita determinar el número mayor entre tres valores.

Observaciones/restricciones:

- No está permitidos utilizar funciones nativas.

Desafío C13

Construya una función propia, en VBA, que permita determinar si un número es PAR o no.

Observaciones/restricciones

- No está permitidos utilizar funciones nativas.
- La salida debe ser "Es Par" o "No es par".

Desafío C14

Construya una función propia que determine el Monto Final M, de una inversión C, a una tasa de interés nominal i, durante k periodos. Para alcanzar el objetivo utilice la siguiente ecuación:

$$M = C\,(1 + i)^k$$

Observaciones/restricciones

- Para asegurar que la función se ejecute correctamente y sin errores, debe validar que todos los valores de la ecuación sean distintos de cero y positivos. En caso de que algún valor no cumpla las condiciones debe entregar el mensaje "Error en parámetros".

Desafío C15. Determinación del valor de una amortización

Construya una función propia que determine el **valor de la amortización** de un periodo ***k*** de un préstamo ***VA*** otorgado a un plazo ***n*** y tasa de interés ***i*** conocidos, utilizando la siguiente ecuación:

$$Amort = \frac{VA \times i \times (1 + i)^{k-1}}{(1 + i)^n - 1}$$

Donde:

- *Amort.* : valor de la amortización del flujo
- *VA* : valor actual o valor presente
- *i* : tasa de interés periódica
- *n* : número total de periodos o flujos
- *k* : número del flujo actual (flujo a calcular)

Observaciones/restricciones

- Validar que los valores sean pertinentes al cálculo, por ejemplo, no pueden existir periodos o cantidad total de flujos que no sean números naturales.
- Divisor cero es un error.
- ¿Puede existir una tasa de interés cero o negativa? Cuestiónese eso.

Desafío C16. Determinación del valor del interés

Construya una función propia que determine el **valor del interés** del flujo ***k*** (periodo) de un préstamo ***VA*** otorgado a un plazo ***n*** y tasa de interés ***i*** conocidos, utilizando la siguiente ecuación:

$$Interés = VA \times i \times \left(\frac{(1+i)^n - (1+i)^{k-1}}{(1+i)^n - 1} \right)$$

Donde:

- Interés : valor del interés del flujo
- VA : valor actual o valor presente
- i : tasa de interés periódica
- n : número total de periodos o flujos
- k : número del flujo actual (flujo a calcular)

Observaciones/restricciones

- Validar que los valores sean pertinentes al cálculo, por ejemplo, no pueden existir periodos negativos o cantidad total de flujos menores a 1.
- Divisor cero es un error.
- ¿Puede existir una tasa de interés cero o negativa? Cuestiónese eso.

Desafío C17. Determinar el valor del flujo

Construya una función propia que determine el **valor del flujo** (cuota) de un préstamo ***VA*** otorgado a un plazo ***n*** y tasa de interés ***i*** conocidos, utilizando la siguiente ecuación:

$$Cuota = \frac{VA \times i \times (1+i)^n}{(1+i)^n - 1}$$

Donde:

- Cuota. : valor constante de la cuota o del flujo
- VA : valor actual o valor presente
- i : tasa de interés periódica
- n : número total de periodos o flujos

Observaciones/restricciones

- Validar que los valores sean pertinentes al cálculo, por ejemplo, no pueden existir periodos negativos o cantidad total de flujos menores a 1.
- Divisor cero es un error.
- ¿Puede existir una tasa de interés cero o negativa? Cuestiónese eso.

Desafío C18. Determinación del valor residual

Construya una función propia que determine el **valor residual** de una deuda en el flujo ***k*** de un préstamo otorgado a un plazo ***n,*** tasa de interés ***i*** y valor de la ***cuota*** conocida, utilizando la siguiente ecuación:

$$Residual = Cuota \times \left(\frac{(1+i)^n - (1+i)^k}{i \times (1+i)^n} \right)$$

Donde:

- Residual : valor residual al flujo
- Cuota. : valor constante de la cuota o del flujo
- i : tasa de interés periódica
- n : número total de periodos o flujos
- k : número del flujo actual (flujo a calcular)

Observaciones/restricciones

- Validar que los valores sean pertinentes al cálculo, por ejemplo, no pueden existir periodos negativos o cantidad total de flujos menores a 1.
- Divisor cero es un error.
- ¿Puede existir una tasa de interés cero o negativa? Cuestiónese eso.

Desafío C19. Determinación de la deuda extinguida

Construya una función propia que determine el **valor de la deuda extinguida** de una deuda en el flujo ***k*** de un préstamo otorgado a un plazo ***n,*** tasa de interés ***i*** y valor de la ***cuota*** conocida, utilizando la siguiente ecuación:

$$Deuda\_Extinguida = Cuota \times \left(\frac{(1+i)^k - 1}{i \times (1+i)^n} \right)$$

Donde:

- Residual : valor residual al flujo
- Cuota : valor constante de la cuota o del flujo
- i : tasa de interés periódica
- n : número total de periodos o flujos
- k : número del flujo actual

Observaciones/restricciones

- Validar que los valores sean pertinentes al cálculo, por ejemplo, no pueden existir periodos negativos o cantidad total de flujos menores a 1.
- Divisor cero es un error.
- ¿Puede existir una tasa de interés cero o negativa? Cuestiónese eso.

Desafío C20. Calcular la tasa de interés compuesta equivalente

Construya una función propia en VBA que permita calcular la tasa de interés compuesta equivalente a partir de una tasa de interés simple y un plazo determinado.

- Datos conocidos: la tasa de interés simple () y el plazo (*n*).
- Objetivo: determinar la tasa de interés compuesta () que sea equivalente a la tasa de interés compuesta dada para el periodo especificado.

Para el cálculo utilice la siguiente ecuación:

$$\boldsymbol{i_s = \frac{(1 + i_c)^n - 1}{n}}$$

Observaciones/restricciones

- Para este ejercicio usted necesita usar bucle.
- No está permitido despejar la ecuación, debe utilizarla tal como se presenta.

- Para asegurar que la función se ejecute correctamente y sin errores, debe validar que todos los valores de la ecuación sean distintos de cero y positivos. En caso de que algún valor no cumpla las condiciones debe entregar el mensaje "Error en parámetros".
- El plazo y/o la tasa puede ser cualquier valor que el usuario determine, lo que se muestra a continuación son solo ejemplos.

		Ejemplos	
i_s	Tasa de interés simple	9,39%	5%
n	Periodos (Flujos)	5	12
i_c	Tasa de interés compuesta	8%	4%

Desafío C21. Calcular la tasa de interés simple equivalente

Construya una función propia en VBA que permita calcular la tasa de interés simple equivalente a partir de una tasa de interés compuesta y un plazo determinado.

- Datos conocidos: la tasa de interés compuesta (i_c) y el plazo (n).
- Objetivo: determinar la tasa de interés simple (i_s) que sea equivalente a la tasa de interés compuesta dada para el periodo especificado.

Para el cálculo utilice la siguiente ecuación:

$$i_c = \sqrt[n]{1 + n \times i_s} - 1$$

Observaciones/restricciones

- Para este ejercicio usted necesita usar bucle.
- No está permitido despejar la ecuación, debe utilizarla tal como se presenta.
- Para asegurar que la función ejecute correctamente y sin errores, debe validar que todos los valores de la ecuación sean distintos de

cero y positivos. En caso de que algún valor no cumpla las condiciones debe entregar el mensaje "Error en parámetros".
- El plazo y/o la tasa puede ser cualquier valor que el usuario determine, lo que se muestra a continuación son solo ejemplos.

		Ejemplos	
i_c	Tasa de interés compuesta	4,47%	6,69%
n	Periodos (Flujos)	6	5
i_s	**Tasa de interés simple**	**5,0%**	**7,64%**

Desafío 22. Obtener el resultado de una exponenciación

Construya una función propia que permita obtener el resultado de una exponenciación.

Observaciones/restricciones

- Se recomienda el uso de bucle.
- El exponente puede ser cualquier número entero.
- No puede utilizar la función nativa POTENCIA.
- No puede utilizar la operación matemática exponenciación.
- Recuerde las propiedades de las potencias, debe considerarlas.
- Tiene que pensar cómo resolver este problema respetando las restricciones.

Desafío C23. Calcular la suma de una serie numérica

Cree una función propia que permita calcular la suma de una **serie numérica** que comienza en un número ***x***, con incremento de 1 en 1, y termina en un número ***y***.

Ejemplo:
- Serie que comienza en 3 y termina en 9
- 3 + 4 + 5 + 6 + 7 + 8 + 9 = 42
- Resultado 42

Las series numéricas es un grupo de números ordenados, que guardan relación consecutiva entre sí.

Observaciones/restricciones

- Se recomienda el uso de bucle.
- Debe validar que los números desde y hasta tengan una relación, de lo contrario no podrá generar la serie.

Desafío C24. Valor de una renta

Construya una función propia (en VBA) que permita obtener el valor de una Renta (R) utilizando la siguiente ecuación:

$$R = \frac{K * i * (1+i)^{n-1}}{(1+i)^n - 1}$$

Donde:
- *K*: capital inicial
- *i*: interés anual
- *n*: número de flujos anuales

Ejemplo:
- K = US$ 700.000
- i = 22%
- n = 5
- R = US$ 200.364,06

Observaciones/restricciones:

- Para asegurar que la función ejecute correctamente y sin errores, debe validar que los datos ingresados por el usuario sean positivos distintos de cero. Si no se cumple con esas condiciones debe entregar el mensaje "Error en parámetros".

Desafío C25

Cree una función propia que permita calcular la suma de una **serie numérica** que comienza en un número ***x***, con incremento de ***n*** en ***y***, para termina en un número ***z***.

Ejemplo:
- Serie que comienza en 3 y termina en 9
- 3 + 4 + 5 + 6 + 7 + 8 + 9 = 42
- Resultado 42

Las series numéricas es un grupo de números ordenados, que guardan relación consecutiva entre sí.

Observaciones/restricciones

- Se recomienda el uso de bucle.
- Debe validar que los números desde y hasta tengan una relación, de lo contrario no podrá generar la serie.

Desafío C26

En matemática, una serie es la generalización de la noción de suma, aplicada a los infinitos términos de una sucesión {*an*}= {*a*1,*a*2,…}, lo que suele escribirse con el símbolo de sumatorio:

$$S = a_1 + a_2 + a_3 + \cdots + a_n = \sum_{k=1}^{n} a_k$$

Donde {*an*} es el «término general» de la sucesión, que usualmente se expresa por medio de una regla, o se obtiene a partir de un **algoritmo**.

Problemas para resolver

Construya una función propia que permita obtener la resta algebraica entre la sumatoria de los numero pares y la sumatoria de los números impares de una sucesión.

Observaciones/restricciones

- La sucesión se genera a partir de un número inicial dado en una celda y se incrementa de uno en uno hasta alcanzar el número final indicado.
- En el proceso los términos inicial y final deben estar considerados en la sumatorias.
- El resultado a entregar debe ser la resta algebraica entre ambas sumatorias.
- La sucesión la genera usted en el proceso a partir de los extremos indicados por el usuario.
- Ejemplo, entre el número 1 y 10 la sucesión es 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10. La sumatoria de los números pares es 30 y de los impares es 25 por tanto el resultado es 30 - 25 = 5

Desafío C27

Construya una función propia que permita obtener la sumatoria, al Enésimo término, de la siguiente serie alterna:

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \cdots = \sum_{n=1}^{N} (-1)^{n+1} \frac{1}{n}$$

Donde:

- ***N,*** es un dato que se entrega a la función para que haga el proceso hasta ese término enésimo.
- ***n***, siempre parte en 1.
- La sucesión siempre es **alterna en sus signos.**

Por ejemplo:

N = 3

$$\mathbf{s = 1 - \frac{1}{2} + \frac{1}{3} = 0,8\overline{33}}$$

Desafío C28

Construya una función propia que permita transformar un número arábigo a número romano.

Observaciones/restricciones

- El/la estudiante deberá analizar el problema, relacionarlo con elementos aprendidos en su trayectoria como estudiante.
- No está permitido el uso de funciones nativas.
- No se permite usar ciclos, esto obliga al estudiante a utilizar sentencias con flujos de control (bifurcaciones).
- No se permite usar arreglos, solo está permitido el uso de operaciones con string para conformar el número romano resultante.

- Los valores de entrada permitidos son números naturales entre 1 y 3999 porque del cuatro mil en adelante los números romanos requieren de otros símbolos para su representación numérica.
- La salida debe ser el número romano equivalente al número arábigo ingresado.

Pruebas posibles de realizar para corroborar si el algoritmo funciona correctamente:

Número arábigo	Número romano
1879	MDCCCLXXIX
2896	MMDCCCXCVI
980	CMLXXX
3333	MMMCCCXXXIII
-555	Número erróneo
0	Número erróneo
999	CMXCIX

Desafío C29

Desarrolle una función propia en VBA que, utilizando un proceso iterativo, permita determinar la Tasa Efectiva Anual (TEA) que corresponde a una Tasa Efectiva Mensual (TEM) especificada por el usuario, en un plazo n dado.

Datos conocidos
- Tasa Efectiva Mensual () y plazo (n), proporcionada por el usuario.
- Objetivo: calcular la Tasa Efectiva Anual () que, al ser aplicada en la siguiente ecuación, iguala la TEM especificada:

$$TEM = (1 + TEA)^{\frac{1}{n}} - 1$$

Donde:

- ***TEA*** : representa la *Tasa Efectiva Anual*, que será un valor entre 0 y 100 (por ejemplo, una tasa del 10% anual).
- ***n*** : es el número de periodos en un año (para la tasa mensual, n =12).

Observaciones/restricciones

- El cálculo debe realizarse mediante iteraciones sobre la TEA hasta alcanzar la TEM especificada.
- No está permitido despejar la ecuación, debe utilizarla tal como se presenta.
- La función debe recibir como parámetros la TEA y el número de periodos (n).
- La función debe validar que la TEA sea ≥ 0 y que n sea > 0. En caso de valores inválidos, la función debe devolver un mensaje de error.
- La función debe retornar la TEM con al menos cuatro decimales de precisión.

Ejemplo de uso**:** si la **TEA** es 12% y el periodo es mensual (**n** = 12), al ejecutar la función se debe obtener la **TEM** que corresponda a ese valor anual.
TEM = 1,55; n = 8; TEA= 13,094

Desafío C30

Construya una función propia que permita calcular la **FACTORIAL DE UN NÚMERO**.

$$n! = n \cdot (n - 1) \cdot (n - 2) \ldots 1$$

Ejemplo:

$$\mathbf{6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720}$$

Observaciones/restricciones

- No se permite el uso de la función nativa factorial
- Debe respetar las propiedades de factorial
- **n** debe ser un número entero ≥ 0

Desafío C31

En matemáticas, un **número primo** es un número natural mayor que 1 que tiene únicamente dos divisores positivos distintos: él mismo y el 1. Por el contrario, los números compuestos son los números naturales que tienen algún divisor natural aparte de sí mismos y del 1, y, por lo tanto, pueden factorizarse. El número 1, por convenio, no se considera ni primo ni compuesto.

Construya una función propia que contenga un algoritmo para determinar si un número *N* cualquiera, es número primo o no lo es.

Entre 1 y 100 existen los siguientes números primos: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89 y 97. El primer número primo a partir del número mil es el 1009, después de diez mil es el 10.007, a partir de cien mil es el 100.003 e inmediatamente tras un millón es el 1.000.003.

El algoritmo no debe utilizar constantes para determinar si un número es primo o no, lo debe hacer por medio de un proceso lógico, haciendo evaluaciones lógicas simples y/o compuestas según sea necesario para el propio algoritmo definido.

N es un valor que se entrega a la función para que evalúe y responda si es primo o no lo es.

Según lo dicho más arriba, 4 no es un número primo, 5 si es un número primo, 8 y 9 no son números primos, etc.

Desafío C32

¿Cómo se calcula el dígito verificador?

El dígito verificador se calcula en función del número, esta operación se utiliza para evaluar la validez del RUT completo.

Se procede a tomar el número de RUT de derecha a izquierda, multiplicando cada dígito por los números que componen la serie numérica 2, 3, 4, 5, 6, y 7; y sumando el resultado de estos productos. Si se ha aplicado la serie hasta el final y quedan dígitos por multiplicar. Al número obtenido por la suma del producto de cada dígito se obtiene el valor del módulo 11, que es lo mismo que dividir por once y guardar el resto de la división entera.

El resultado final, se convierte a un número o a la letra K siguiendo estas reglas:

- Si el resultado es 11, el dígito verificador será 0 (cero).
- Si el resultado es 10, el dígito verificador será K.
- En otro caso, el resultado será el propio dígito verificador.

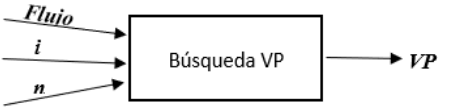
Ejemplo el RUT 5.544.332, para calcular el dígito verificador:

Primera parte	Segunda parte
2 * 2 = 4 3 * 3 = 9 3 * 4 = 12 4 * 5 = 20 4 * 6 = 24 5 * 7 = 35 5 * 2 = 10	Suma de los productos = 4 + 9 + 12 + 20 + 24 + 35 + 10 = 114 Se divide 114: 11 = 10 (se toma la parte entera del número) Luego, a 11 se le resta el residuo de 114:11 que es 4 y el resultado es 7 Siguiendo las reglas mencionadas el rut completo sería 5.544.332-7

Desafío C33

Supongamos que enfrentamos una **restricción de flujo de caja** disponible para el pago de un crédito que necesitamos solicitar. El banco nos ofrece una tasa de interés ***i*** mensual y un plazo máximo de ***n*** periodos. Debemos evaluar las posibles combinaciones de crédito que se ajusten a nuestras restricciones de flujo de caja, las cuales están influenciadas por la lentitud con la que nos pagan nuestros clientes.

Dado que tienes habilidades de programación, ayúdanos a construir una función propia que calcule el valor presente (***VP***) del préstamo que podríamos solicitar, teniendo en cuenta la tasa de interés (***i***), el plazo (***n***) ofrecidos por el banco, y nuestras **restricciones de flujo de caja**.



Antecedentes

- *VP*: valor presente
- *i*: tasa de interés anual
- *n*: números de periodos de pago
- *Flujo:* valor del flujo

Restricciones

- La tasa de interés se debe ingresar como un número, sin símbolo porcentual.
- Ninguno de los valores de entrada debe ser menor o igual a cero, si eso ocurre debe indicar un error.
- El resultado debe estar redondeado al peso (utilice la función ROUND() para redondear).
- La ecuación que debe utilizar para el cálculo es la siguiente:

$$\boldsymbol{Flujo = \frac{VP\ x\ i}{1 - (1 + i)^{-n}}}$$

- No puede alterar o despejar la ecuación, **debe iterar sobre la ecuación descrita**.
- La función debe contener a lo menos un bucle para realizar el cálculo.
- La tolerancia permitida (diferencia), para el resultado es de ± 5.

Ejemplos para sus pruebas:

Flujo	vp	i	n
115.365,89	1.800.000	1,55	18
86.066,43	1.000.000	0,5	12
94.959,58	2.500.000	1,8	36
169.304,44	3.500.000	1,23	24

Apéndice D

Listado de funciones nativas de VBA

Para obtener un listado completo y detallado de las funciones de VBA, lo más recomendable es consultar la documentación oficial de Microsoft o utilizar el Explorador de Objetos en el editor de VBA. Estas son las fuentes más confiables y actualizadas.

A continuación, se presenta un listado de las funciones nativas más utilizadas, categorizadas según su área de aplicación, junto con una breve descripción de cada una (una lista completa y actualizada puede encontrar en funciones VBA).

Funciones nativas para manipulación de cadenas	
Función	**Descripción**
Len	Devuelve la longitud de una cadena de texto.
Left	Extrae un número determinado de caracteres desde el inicio de una cadena.
Right	Extrae un número determinado de caracteres desde el final de una cadena.
Mid	Extrae una subcadena desde una posición específica de una cadena.
InStr	Devuelve la posición de la primera aparición de una subcadena en una cadena.
LCase	Convierte todos los caracteres de una cadena a minúsculas.
UCase	Convierte todos los caracteres de una cadena a mayúsculas.
Trim	Elimina los espacios en blanco al inicio y al final de una cadena.
Replace	Reemplaza una subcadena dentro de una cadena con otra subcadena.

StrReverse	Invierte el orden de los caracteres de una cadena.
Funciones nativas para fecha y hora	
Date	Devuelve la fecha actual del sistema.
Now	Devuelve la fecha y hora actuales.
Time	Devuelve la hora actual del sistema.
Day	Devuelve el día de una fecha.
Month	Devuelve el mes de una fecha.
Year	Devuelve el año de una fecha.
Hour	Devuelve la hora de una hora específica.
Minute	Devuelve los minutos de una hora específica.
Second	Devuelve los segundos de una hora específica.
DateAdd	Suma un intervalo de tiempo a una fecha.
DateDiff	Calcula la diferencia entre dos fechas en unidades específicas (días, meses, años, etc.
DateSerial	Devuelve una fecha a partir de los valores de año, mes y día.
TimeSerial	Devuelve una hora a partir de los valores de hora, minutos y segundos.
Weekday	Devuelve el día de la semana de una fecha.
MonthName	Devuelve el nombre del mes correspondiente a un número de mes.
WeekdayName	Devuelve el nombre del día de la semana correspondiente a un número de día.
Funciones nativas matemáticas	
Abs	Devuelve el valor absoluto de un número.
Atn	Devuelve el arcotangente de un número.
Cos	Devuelve el coseno de un ángulo en radianes.
Exp	Devuelve el número e elevado a la potencia especificada.
Fix	Similar a Int, pero para números negativos devuelve el primer entero más cercano hacia 0.
Int	Devuelve la parte entera de un número, truncando los decimales.
Log	Devuelve el logaritmo natural (base e) de un número.
Rnd	Devuelve un número aleatorio entre 0 y 1.

Round	Redondea un número a un número especificado de decimales.
Sgn	Devuelve el signo de un número (1 si es positivo, 0 si es 0, -1 si es negativo).
Sin	Devuelve el seno de un ángulo en radianes.
Sqr	Devuelve la raíz cuadrada de un número.
Tan	Devuelve la tangente de un ángulo en radianes.
Derived Math	Lista de funciones matemáticas nointrinsicas que se pueden derivar de las funciones matemáticas intrínsecas.
Funciones nativas financieras	
PV	Calcula el valor presente de una inversión o préstamo basado en una serie de pagos futuros y una tasa de interés constante.
FV	Calcula el valor futuro de una inversión o préstamo, teniendo en cuenta pagos periódicos, una tasa de interés constante, y un valor presente.
PMT	Calcula el pago periódico de un préstamo o inversión, considerando una tasa de interés constante y un número determinado de periodos.
NPV	Calcula el valor presente neto (NPV) de una serie de flujos de efectivo en función de una tasa de descuento.
IRR	Calcula la tasa interna de retorno (IRR) para una serie de flujos de efectivo periódicos.
RATE	Calcula la tasa de interés por periodo de un préstamo o inversión basándose en pagos periódicos, número total de periodos y valor presente.
IPMT	Calcula el pago de intereses de un periodo específico para una inversión o préstamo con pagos periódicos y una tasa de interés constante.
PPMT	Calcula el pago de capital en un periodo específico de un préstamo o inversión con pagos periódicos.
SLN	Calcula la depreciación lineal de un activo por periodo, asumiendo una depreciación constante.
SYD	Calcula la depreciación acelerada de un activo por periodo usando el método de suma de dígitos de los años (SYD).
DDB	Calcula la depreciación de un activo por el método del saldo decreciente doble o por otro método de saldo decreciente especificado.
CUMIPMT	Calcula el interés acumulado pagado entre dos periodos específicos de un préstamo o inversión.

CUMPRINC	Calcula el capital acumulado pagado en un préstamo entre dos periodos específicos.
MIRR	Calcula la tasa interna de retorno modificada (MIRR) para una serie de flujos de efectivo considerando costos de inversión y tasas de reinversión.
Funciones nativas para conversión de tipos	
CInt	Convierte una expresión en un número entero.
CLng	Convierte una expresión en un número entero largo.
CDbl	Convierte una expresión en un número de doble precisión (punto flotante).
CStr	Convierte una expresión en una cadena de texto.
CDate	Convierte una expresión en una fecha.
CSng	Convierte una expresión en un número de precisión simple.
CByte	Convierte una expresión en un byte.
CBool	Convierte una expresión en un valor booleano (True o False).
CVar	Convierte una expresión en un tipo de dato Variant.
Funciones nativas de control de flujo y ejecución	
MsgBox	Muestra un cuadro de mensaje con un texto personalizado.
InputBox	Solicita al usuario que ingrese un valor mediante un cuadro de diálogo.
DoEvents	Permite que el sistema procese otros eventos mientras se ejecuta el código.
Switch	Evalúa una lista de expresiones y devuelve el valor correspondiente a la primera expresión que es verdadera.
IIf	Evalúa una expresión y devuelve un valor si es verdadera y otro valor si es falsa.
Funciones nativas para información y evaluación	
IsNull	Verifica si una expresión es Null.
IsEmpty	Verifica si una variable no ha sido inicializada.
IsNumeric	Verifica si una expresión es numérica.
IsDate	Verifica si una expresión puede ser interpretada como una fecha.
TypeName	Devuelve el nombre del tipo de una variable.

VarType	Devuelve un valor que indica el tipo de una variable.
IsArray	Devuelve Verdadero si la variable es una matriz; en caso contrario, devuelve Falso. IsArray es especialmente útil con variant que contienen matrices.
Funciones nativas para trabajar con archivos	
Dir	Devuelve el nombre de un archivo o directorio que coincida con una ruta específica.
FileLen	Devuelve el tamaño de un archivo en bytes.
GetAttr	Devuelve los atributos de un archivo o carpeta.
SetAttr	Establece los atributos de un archivo o carpeta.
MkDir	Crea un nuevo directorio.
RmDir	Elimina un directorio vacío.
Kill	Elimina uno o varios archivos.
FreeFile	Devuelve un número de archivo libre que se puede usar para abrir archivos.
Funciones nativas para evaluación de errores	
IsError	Determina si una expresión representa un valor de error.
CVErr	Convierte un número en un valor de error.

Glosario

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W

Director
Jorge Montealegre

Equipo editorial
Luz María Astudillo
Galo Ghigliotto
Daniella Gutiérrez
Katherine Hoch
Consuelo Olguín

Equipo diseño
Andrea Estefanía
Andrea Meza
Ana Ramírez

Equipo administrativo
Martín Angulo
Daisy Farías
Claudia Gamboa

Equipo comercial
Darío Núñez
Javier Solís

Esta primera edición de
Fundamentos de programación para no especialistas se terminó de editar en
agosto de 2025.

Para los textos de portada se utilizó la
tipografía Eurostile; para el interior se utilizó
la tipografía Minion Pro.