Fundamentos de programación con implementaciones en Python y R

orientados a la computación científica, el análisis y la ciencia de datos, entre otros

Camilo José Torres-Jiménez

2024-07-23

Tabla de contenidos

Acerca de

El presente es parte del material que se ha utilizado para las clases de la asignatura **Programación en Lenguajes Estadísticos (PLE) - 2016374**. Dicha asignatura está dirigida a estudiantes de la carrera de Estadística de la Facultad de Ciencias de la Universidad Nacional de Colombia, Sede Bogotá.

i Histórico de este material

Este material se ha venido preparando, construyendo y transformando a través del tiempo. A continuación se mencionan las diferentes herramientas que se han utilizado para producirlo.

Archivos fuente	Formato de salida	Inicio	Fin
Jupyter + Google Colab	`html`	2021-I	2021-I
R Markdown + bookdown + Google Colab	`html`	2021-II	2022-I
Quarto book + Google Colab	`html`	2022-II	

i Acerca del autor (profesor de la asignatura)

Formación:

- **Matemático**. Línea de profundización: *Informática*. Área del trabajo de grado: *Computación en paralelo*.
- **Magister en Ciencias Estadística**, Área general y específica de la tesis: *Procesos estocásticos*; *procesos de ramificación*.

Experiencia en la academia:

- Un semestre como **docente de cátedra** en las *universidades Central y Javeriana*. Cursos a cargo: Álgebra Lineal, Estadística II y Probabilidad y Estadística; dirigidos a: *Ingeniería y Ciencias Económicas*.
- Un (1) año como **docente de planta** en la *Universidad Santo Tomas*. Cursos a cargo: Estadística I, Estadística II e Inferencia Estadística; dirigidos a: *Ciencias Económicas y Estadística*.
- Seis (6) años como **auxiliar docente** en la *Universidad Nacional de Colombia, Sede Bogotá*. Cursos a cargo: Probabilidad y Estadística Fundamental, Estadísti-

ca Descriptiva Multivariada, Programación en Lenguajes Estadísticos, Estadística I, Probabilidad Fundamental, Bioestadística Fundamental; dirigidos a: *Ciencias, Ciencias Económicas e Ingeniería.*
- Dos (2) años como **docente de planta** en la *Universidad Nacional de Colombia, Sede Bogotá.* Cursos a cargo: Probabilidad y Estadística Fundamental, Programación en Lenguajes Estadísticos, Estadística Social Fundamental; dirigidos a: *Estadística, Ciencias Económicas, Ingeniería y Ciencias Humanas, Políticas y Sociales.*

Experiencia fuera de la academia:

- Seis (6) años como **programador**, **administrador de bases de datos (DBA)**, **analista de datos** y **líder de proyecto** en el marco de convenios entre el *Observatorio Colombiano de Ciencia y Tecnología* y *Colciencias* (*Minciencias*).
- Dos (2) años como **gerente** y **representante legal** de una corporación de ciencia y tecnología sin animo de lucro dedicada a proyectos de desarrollo de software, de soporte operativo a conceptual y de análisis e investigación en torno a los datos del Sistema Nacional de Ciencia, Tecnología e Innovación Colombiano.
- Dos (2) años como **Profesional I** en la *Rectoría* de la *Pontificia Universidad Javeriana*, encargado de apoyar el desarrollo y seguimiento de la planeación y el desarrollo de proyectos relacionados con las estadísticas institucionales.
- **Consultor y analista estadístico** para diferentes instituciones, en temas de educación superior.

Parte I

Preliminares

1 Computadores

En esta sección se hará una revisión de algunos temas relacionados con las características de los computadores. En esta revisión se hace mención a la historia de los computadores, a la maquina de Turing, al modelo de von Newmann, a la organización interna de los computadores y al lenguaje de un computador.

> **i** Preparación de clase
>
> - Leer las secciones 1.1 a 1.7 del libro: L. Joyanes Aguilar, *Fundamentos de programación: algoritmos, estructura de datos y objetos*. McGraw Hill, **2020** [Online]. Disponible en http://www.ebooks7-24.com.ezproxy.unal.edu.co/?il=10409&pg=1 o en http://ezproxy.biblored.gov.co:2117/?il=10409&pg=1
> - Ver: **¿Qué es una máquina de Turing?** https://youtu.be/iaXLDz_UeYY
>
> En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó en el texto y vio en el video, incluya su discusión, reflexiones y conclusiones al respecto; exponga lo que no entendió e intente encontrar por su cuenta respuestas a las preguntas que le surgieron, para poder compartirlas en clase.

1.1 Breve historia

- Ábaco. Sumar y restar.
- 1642. Blas Pascal. Primera calculadora mecánica. Sumas y restas.
- 1694. Gottfried Leibniz. Sumar, restar, multiplicar y dividir.
- 1819. Joseph JacQuard. Bases de las tarjetas perforadas.
- 1835. Charles Babbage. Máquina analítica: calculadora que incluía un dispositivo de entrada, dispositivo de almacenamiento de memoria, una unidad de control y dispositivos de salida.
- 1841. Ada Augusta, condesa de Lovelace. Publica los trabajos de Babbage. Primera programadora.
- 1884. Herman Hollerith. Inventó máquina calculadora que funcionaba con electricidad y tarjetas perforadas. 1896. Creó la empresa Tabulating Machine Company (IBM en 1924).

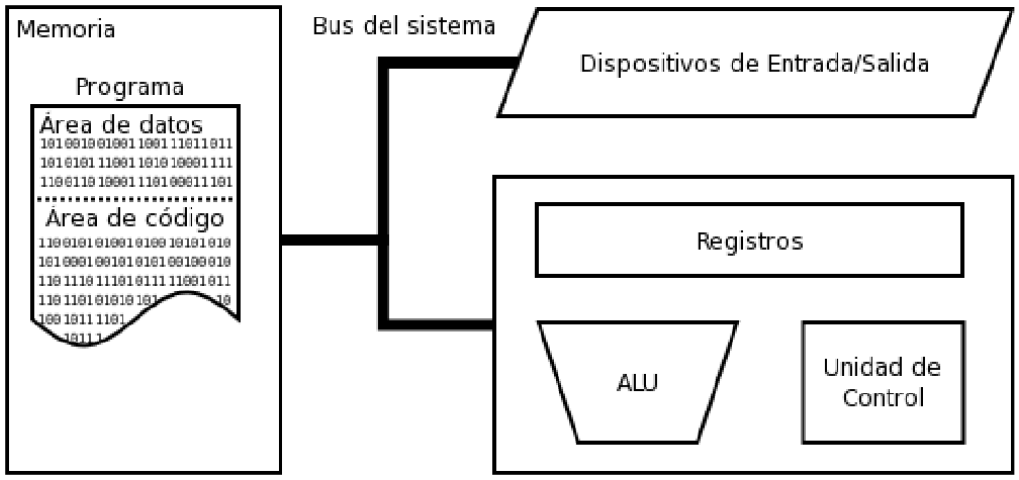
- 1940s. Colossus Mark I: “the world’s first electronic digital programmable computer”. ABC: “the first automatic electronic digital computer”. ENIAC: “the first electronic programmable computer built in the U.S.”.

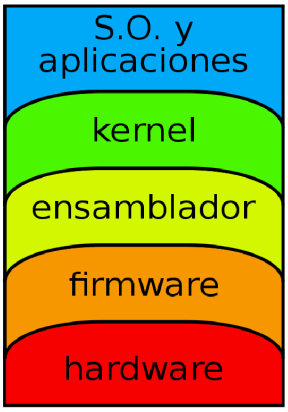
1.2 “Universal Computing machine” o “universal Turing machine”

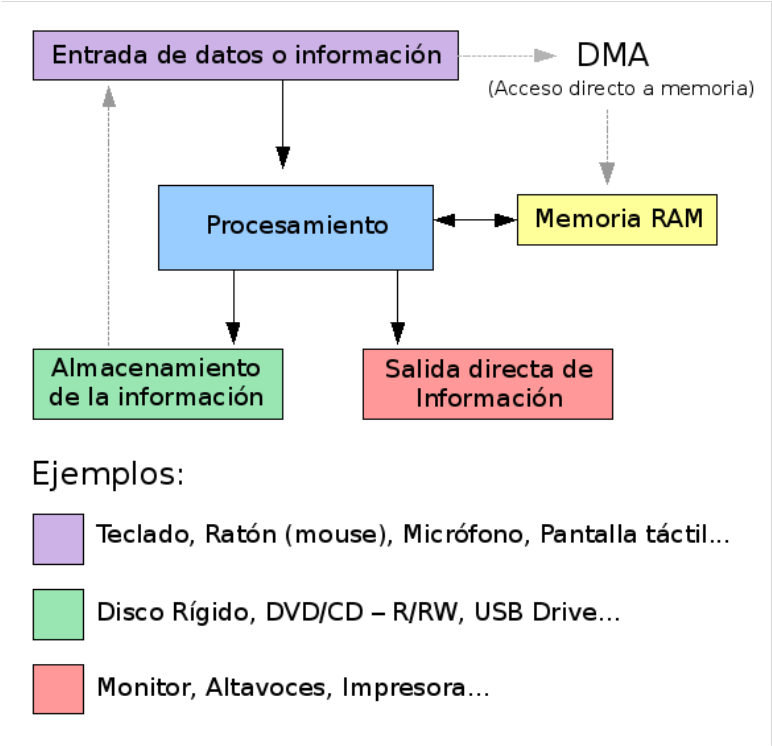
Turing, A. M. (1937). ***“On Computable Numbers, with an Application to the Entscheidungsproblem”***. Proceedings of the London Mathematical Society. 2. 42 (1): 230–265.

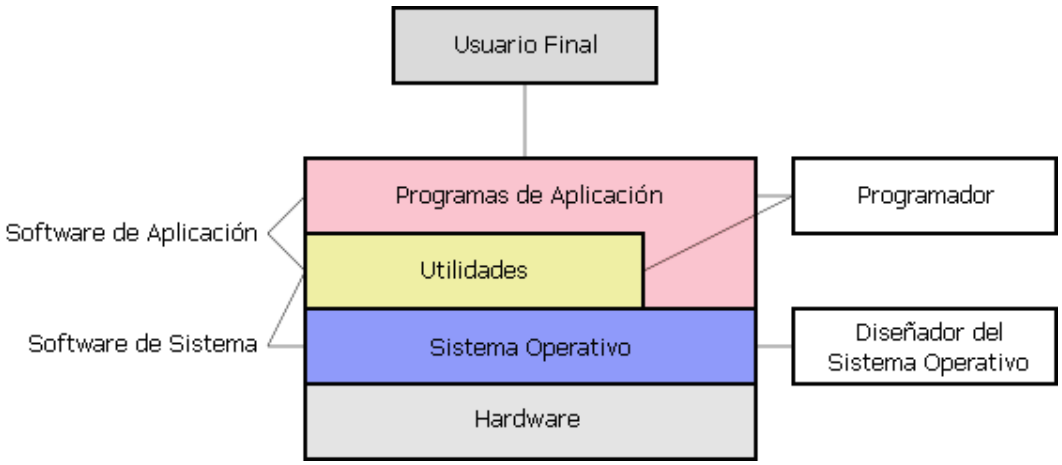
De Castro, R. (2004). ***Teoría de la computación: lenguajes, autómatas, gramáticas***. Editorial UN. enlace

1.3 Modelo von Newmann




S.O. y aplicaciones
kernel
ensamblador
firmware
hardware


Entrada de datos o información
DMA
(Acceso directo a memoria)
Procesamiento
Memoria RAM
Almacenamiento de la información
Salida directa de Información
Ejemplos:
Teclado, Ratón (mouse), Micrófono, Pantalla táctil...
Disco Rígido, DVD/CD – R/RW, USB Drive...
Monitor, Altavoces, Impresora...



1.4 Organización de un computador

1.4.1 Capas de abstracción

1.4.2 Flujo de datos

1.4.3 Programas (Software)

1.5 El lenguaje del computador

bit = **bi**nary dig**it** (dígito binario). Se almacena un cero (0) o un uno (1), apagado o encendido. 8 bits = 1 byte.

ASCII (American Standard Code for Information Interchange) o US-ASCII : 128 caráteres.

Caracter	Código ASCII	Binario
⋮	⋮	⋮
A	65	01000001
B	66	01000010
C	67	01000011
⋮	⋮	⋮

Unicode: 143859 caráteres (Versión 13.0. Marzo 2020). UTF-8 (Unicode Transformation Formats): Codificación que en el 2020 era usada por más del 95% de los sitios web.

2 Lenguajes de programación

En esta sección se hará una revisión de algunos temas relacionados con la programación de los computadores y los lenguajes de programación. En esta revisión se hace mención a los tipos de lenguaje de programación y a los paradigmas de programación.

i Preparación de clase

- Leer las secciones 1.8 a 1.17 del libro: L. Joyanes Aguilar, *Fundamentos de programación: algoritmos, estructura de datos y objetos*. McGraw Hill, 2020 [Online]. Disponible en: http://www.ebooks7-24.com.ezproxy.unal.edu.co/?il=10409&pg=1 o en http://ezproxy.biblored.gov.co:2117/?il=10409&pg=1
- Leer: ***Programming paradigm*** https://en.wikipedia.org/wiki/Programming_paradigm

En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó en el texto, incluya su discusión, reflexiones y conclusiones al respecto; exponga lo que no entendió e intente encontrar por su cuenta respuestas a las preguntas que le surgieron, para poder compartirlas en clase.

2.1 La programación de computadores

- "Un programa de computador es un conjunto detallado de instrucciones paso a paso que indican al dispositivo las acciones a realizar con exactitud. Si cambiamos el programa, entonces la computadora realiza una secuencia diferente de acciones y, por consiguiente, ejecutará una tarea diferente".
- "El software (los programas) regulan el hardware (la parte física)". "El proceso de creación de software se llama **programación**".

2.2 ¿Qué es un lenguaje de programación?

Como un programa es una secuencia de instrucciones que le dicen a un computador lo que debe hacer, naturalmente esas instrucciones se le deben dar en un lenguaje que pueda "entender".

El lenguaje "natural" humano, en el idioma que sea, sigue siendo entre difícil e imposible de "comprender" por un computador.

El conjunto de palabras o símbolos (*código* de escritura) que se usan para que un computador reciba instrucciones es lo que se llama un *lenguaje de programación*. Cada lenguaje de programación tiene un formato preciso (su *sintaxis*) y un significado preciso (su *semántica*).

Una característica relevante de los lenguajes de programación es precisamente que más de un programador puedan tener un conjunto común de instrucciones que puedan ser comprendidas entre ellos para realizar la construcción del programa de forma colaborativa.

Existen numerosos lenguajes de programación (por ejemplo, ver los que se listan y comparan en https://en.wikipedia.org/wiki/Comparison_of_programming_languages).

Cada lenguaje de programación tiene sus características propias, y por ende, tiene sus propias ventajas y desventajas dependiendo de cómo se le mire. En el gráfico de este enlace se dan unas ciertas ventajas y desventajas de los "principales lenguajes de programación para ciencias de datos" (en la opinión del que hizo el gráfico). Adicionalmente, tenemos la comparación que por ejemplo se hace aquí

Por otro lado, es importante tener claro que **los lenguajes de marcado no son lenguajes de programación**. En el Apéndice A encontrarán una pequeña revisión de estos otros tipos de lenguaje. Dicha revisión está enfocada en lo que suelen necesitar estadísticos y afines, de los lenguajes de marcado, para la obtención de informes, reportes, presentaciones y documentos en general.

2.3 Tipos de lenguajes de programación

- **Lenguajes de máquina**: Instrucciones en cadenas binarias.
- **Lenguajes de bajo nivel**: Código de maquina y lenguaje ensamblador, se trabaja con los registros de memoria de forma directa.
- **Lenguajes de medio nivel**: Se acerca a los lenguajes de bajo nivel pero teniendo al mismo tiempo algunas cualidades de lenguaje humano.
- **Lenguajes de alto nivel**: Están formados por elementos lo más cercanos que se pueda al lenguaje natural.

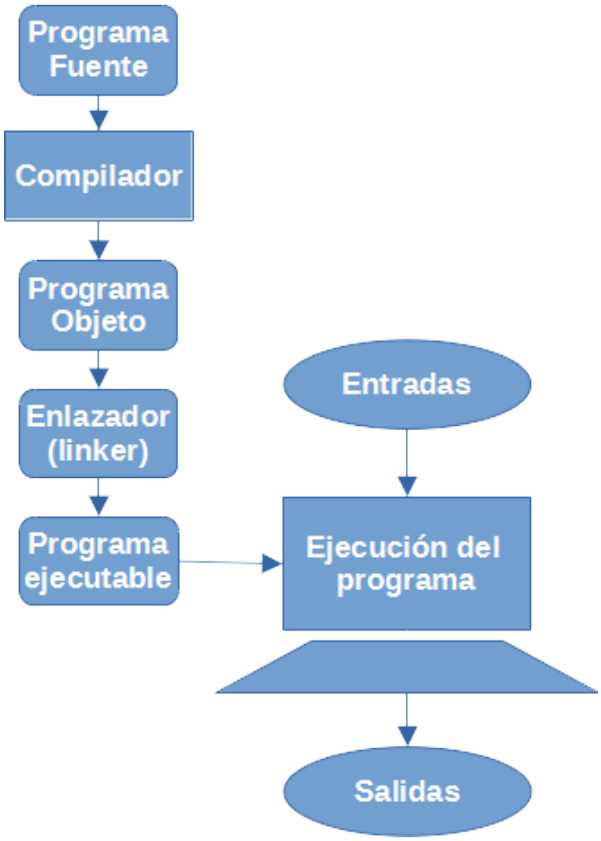
2.4 Traducción de los lenguajes

Un computador únicamente "entiende" instrucciones en cadenas binarias, así que todos los demás lenguajes de programación deberán traducirse a lenguaje de máquina para que el computador pueda seguir las instrucciones que ahí se le dan. Es evidente que se requiere un mecanismo que traduzca en lenguaje de máquina, las instrucciones dadas en el lenguaje de programación

de nuestra elección. Dicho mecanismo es básicamente un programa que se encarga de leer el lenguaje que estemos utilizando (programa fuente) y lo traduce a un programa equivalente en el lenguaje de máquina (programa objeto). Como parte importante de este proceso de traducción, el traductor informa a su usuario de la presencia de errores en el archivo fuente.

La traducción se hace usualmente de una de dos maneras:

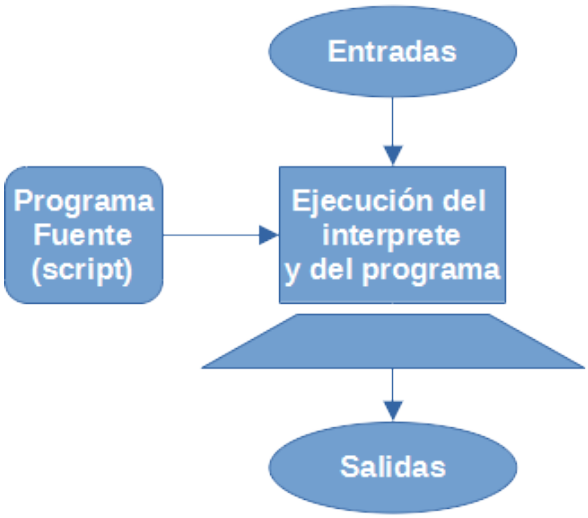
- Se traducen por completo todas las instrucciones que están dadas en un archivo, produciendo otro archivo con las instrucciones traducidas. A ese proceso se lo llama *compilar* y al programa traductor se le denomina *compilador*.



- Las instrucciones se van traduciendo conforme son encontradas o dadas. A este proceso se lo llama *interpretar* y a los programas que lo hacen se los conoce como *interpretes*.

2.5 Paradigmas de programación

"Un paradigma de programación indica un método de realizar cómputos y la manera en que se deben estructurar y organizar las tareas que debe llevar a cabo un programa"



Los paradigmas fundamentales están asociados a determinados modelos de cómputo y estilos de programación.

Los lenguajes de programación pueden incorporar varios paradigmas (lenguajes multiparadigma).

Programación imperativa:

> "In computer science, **imperative programming** is a programming paradigm of software that uses statements that change a program's state. In much the same way that the imperative mood in natural languages expresses commands, an imperative program consists of commands for the computer to perform. Imperative programming focuses on describing how a program operates step by step, rather than on high-level descriptions of its expected results."

Programación estructurada:

> "**Structured programming** is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection and repetition , block structures, and subroutines."

Programación procedimental:

> "**Procedural programming** is a programming paradigm, derived from imperative programming, based on the concept of the procedure call. Procedures (a type of routine or subroutine) simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself."

Programación orientada a arreglos:

> "In computer science, **array programming** refers to solutions which allow the application of operations to an entire set of values at once… Modern programming languages that support array programming (also known as vector or multidimensional languages) have been engineered specifically to generalize operations on scalars to apply transparently to vectors, matrices, and higher-dimensional arrays… In these languages, an operation that operates on entire arrays can be called a vectorized operation…"

Programación orientada a objetos:

> "**Object-oriented programming** is a programming paradigm based on the concept of <>, which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods)."

Programación funcional:

> "In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program."

En el curso trabajaremos:

- El paradigma de programación imperativa estructurada procedimental (PIEP), en donde un conjunto de instrucciones se ejecutan de una en una, de principio a fin, de modo secuencial excepto cuando intervienen estructuras de control o de salto de secuencia.
- El paradigma de programación orientado a objetos (POO), el cual se basa en el diseño y construcción de objetos que se componen de datos (atributos) y operaciones sobre esos datos (métodos). El programador define en primer lugar los objetos del problema junto con sus atributos y operaciones.
- De manera un tanto indirecta, se tocarán algunos aspectos o temas que se podrían considerar parte de los paradigmas de programación funcional y orientado a arreglos.

Parte II

PIEP

3 Algoritmos y la resolución de problemas

En esta sección se hará una revisión del proceso a seguir para la resolución de problemas mediante el uso de un computador, incluyendo el concepto de algoritmo.

En esta revisión se hace mención a las características, medios de expresión, elementos básicos y estructura general de los algoritmos.

> **i** Preparación de clase
>
> - Leer las secciones 2.1 a 2.3 y 2.5 a 2.8 del libro: L. Joyanes Aguilar, *Fundamentos de programación: algoritmos, estructura de datos y objetos*. McGraw Hill, **2020** [Online]. Disponible en http://www.ebooks7-24.com.ezproxy.unal.edu.co/?il=10409 o en http://www.ebooks7-24.com.ezproxy.biblored.gov.co/?il=10409
>
> En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó en el texto, incluya su discusión, reflexiones y conclusiones al respecto; exponga lo que no entendió e intente encontrar por su cuenta respuestas a las preguntas que le surgieron, para poder compartirlas en clase.

3.1 Fases en la resolución de problemas

Las personas aprenden lenguajes de programación para usar un computador en la resolución de problemas. Varias fases pueden ser identificadas en el proceso de resolución de problemas mediante el uso de un computador.

- **Análisis**. El primer paso para encontrar la solución a un problema es el análisis del mismo. Se debe examinar cuidadosamente el problema a fin de obtener una idea clara sobre lo que solicita y determinar lo que se necesita para conseguirlo. El problema se analiza teniendo en presente la especificación de los requisitos dados por la persona que requiere el programa (el usuario).
- **Diseño**. Una vez analizado el problema, se diseña una solución que conducirá a un *algoritmo* que resuelva el problema. Se plantean, la forma en que se reciben los datos de entrada, el proceso por el cual se produce el resultado y la forma en que se muestra la salida. Una vez que se ha terminado de escribir un algoritmo es necesario comprobar que realiza la tarea para la cual fue diseñado y produce el resultado correcto y esperado.

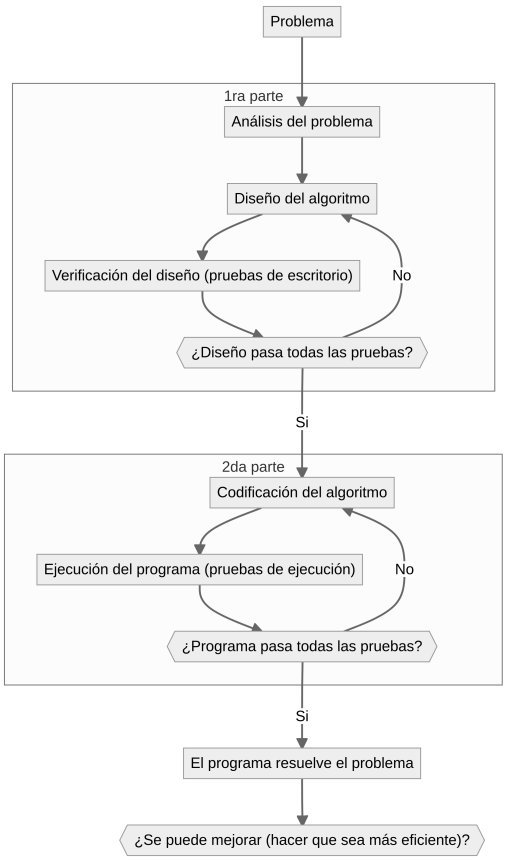


Figura 3.1: Fases en la elaboración de un programa de computador que resuelva un problema dado.

- **Codificación (implementación)**. El algoritmo diseñado se escribe en la sintaxis del lenguaje de programación seleccionado, obteniendo así el programa fuente.
- **Ejecución, verificación y depuración**. El programa se ejecuta, se comprueba rigurosamente y se eliminan todos los errores que puedan aparecer (eliminar *bugs*). Asumiendo una competente codificación, entre mejor se haga todo en las fases de análisis y diseño menos tiempo se gastará buscando, identificando y solucionando errores. Cuando se ejecuta un programa pueden producirse tres tipos de errores: *de traducción*, *de ejecución* y *lógicos*.
- **Documentación**. Documentos que soportan todo lo realizado con respecto al programa. Incluye diseño, codificación, manuales, normas, etc. Se recomienda ir documentando cada cosa que se hace, en vez de esperar hasta el final y tener que documentar todo de principio a fin. Existe documentación interna, incluida dentro del código fuente mediante *comentarios*, y documentación externa. La documentación es vital para que los usuarios puedan usar el programa y para que los programadores entiendan y modifiquen el código fuente.
- **Mantenimiento**. El programa se actualiza y modifica cada vez que sea necesario, de modo que se cumplan todas las necesidades adicionales de los usuarios.

3.2 Algoritmos

3.2.1 Características

- **Carácter finito**. Un algoritmo siempre debe terminar después de un número finito de pasos.
- **Precisión**. Cada paso de un algoritmo debe estar precisamente definido; las operaciones a llevar a cabo deben ser especificadas de manera rigurosa y no ambigua para cada caso.
- **Entrada**. Un algoritmo tiene cero o más entradas: cantidades que le son dadas antes de que el algoritmo comience, o dinámicamente mientras el algoritmo corre. Estas entradas son tomadas de conjuntos específicos de objetos.
- **Salida**. Un algoritmo tiene una o más salidas: cantidades que tienen una relación específica con las entradas.
- **Eficacia**. También se espera que un algoritmo sea eficaz, en el sentido de que todas las operaciones a realizar en un algoritmo deben ser suficientemente básicas como para que en principio puedan ser hechas de manera exacta y en un tiempo finito por un hombre usando lápiz y papel.

Los problemas complejos se pueden resolver de manera más eficaz cuando se dividen en subproblemas que sean más fáciles de solucionar que el inicial (diseño descendente).

3.2.2 Medios de expresión

- **Máquina de Turing**: Es un modelo matemático diseñado por Alan Turing que formaliza el concepto de algoritmo, es una descripción que se puede considerar de abstracción de bajo nivel.
- **Pseudocódigo**: Es la descripción, de tal forma que se asemeja a un lenguaje de programación pero con algunas convenciones del lenguaje natural.
- **Diagrama de Flujo**: Los diagramas de flujo son descripciones gráficas de algoritmos, usando símbolos conectados con flechas para indicar la estructura del algoritmo.
- **Descripción de Alto Nivel**: Se establece el problema, se selecciona el modelo matemático y se explica el algoritmo de manera verbal.

3.2.3 Elementos básicos

3.2.3.1 Comentarios

Los comentarios sirven para documentar el algoritmo y en ellos se escriben anotaciones generalmente sobre su funcionamiento. En pseudocódigo, cuando se coloque un comentario de una sola línea se escribirá precedido de `//`. Si el comentario es multilínea, lo pondremos entre `{}`.

3.2.3.2 Palabras reservadas

Las palabras reservadas o palabras clave (Keywords) son palabras que tienen un significado especial, como: `inicio` y `fin`, que marcan el principio y fin del algoritmo.

3.2.3.3 Identificadores

Identificadores son los nombres que se dan a las constantes simbólicas, variables, funciones, procedimientos, u otros objetos que manipula el algoritmo. La regla para construir un identificador establece que:

- Debe resultar significativo, sugiriendo lo que representa.
- No podrá coincidir con palabras reservadas, propias del lenguaje algorítmico. Como se verá más adelante, la representación de algoritmos mediante pseudocódigo va a requerir la utilización de palabras reservadas.
- Se recomienda un máximo de 50 caracteres.
- Comenzará siempre por un carácter alfabético y los siguientes podrán ser letras, dígitos o el símbolo de subrayado.
- Podrá ser utilizado indistintamente escrito en mayúsculas o en minúsculas

3.2.3.4 Datos

Dato es la expresión general que describe los objetos con los cuales opera el algoritmo. El tipo de un dato determina su forma de almacenamiento en memoria y las operaciones que van a poder ser efectuadas con él. En principio hay que tener en cuenta que, prácticamente en cualquier lenguaje y por tanto en cualquier algoritmo, se podrán usar datos de los siguientes tipos:

- **entero**. Subconjunto finito de los números enteros, cuyo rango o tamaño dependerá del lenguaje en el que posteriormente codifiquemos el algoritmo y de la computadora utilizada.
- **real**. Subconjunto de los números reales limitado no sólo en cuanto al tamaño, sino también en cuanto a la precisión.
- **lógico**. Conjunto formado por los valores verdad y falso.
- **carácter**. Conjunto finito y ordenado de los caracteres que la computadora reconoce.
- **cadena**. Los datos (objetos) de este tipo contendrán una serie finita de caracteres, que podrán ser directamente traídos o enviados a/desde la consola.

En los algoritmos para indicar que un dato es de uno de estos tipos se declarará utilizando directamente el identificador o nombre del tipo.

Los datos pueden venir expresados como constantes, variables, expresiones o funciones.

Las **constantes** son datos cuyo valor no cambia durante todo el desarrollo del algoritmo.

Una **variable** es un objeto cuyo valor puede cambiar durante el desarrollo del algoritmo. Se identifica por su nombre y por su tipo, que podrá ser cualquiera, y es el que determina el conjunto de valores que podrá tomar la variable. En los algoritmos se deben declarar las variables que se van a usar, especificando su tipo.

Una **expresión** es una combinación de operadores y operandos. Los operandos podrán ser constantes, variables u otras expresiones y los operadores de cadena, aritméticos, relacionales o lógicos. Las expresiones se clasifican, según el resultado que producen, en:

- **Numéricas**. Los operandos que intervienen en ellas son numéricos, el resultado es también de tipo numérico y se construyen mediante los operadores aritméticos. Se pueden considerar análogas a las fórmulas matemáticas. Debido a que son los que se encuentran en la mayor parte de los lenguajes de programación, los algoritmos utilizarán los siguientes operadores aritméticos: inverso aditivo (`-`), suma (`+`), resta (`-`), multiplicación (`*`), división real (`/`), residuo/módulo de la división entera (`mod`) y cociente de la división entera (`div`). Tenga en cuenta que la división real siempre dará un resultado real y que los operadores `mod` y `div` sólo operan con enteros y el resultado es entero.
- **Alfanuméricas**. Los operandos son de tipo alfanumérico y producen resultados también de dicho tipo. Se construyen mediante el operador de concatenación, representado por el operador ampersand (`&`) o con el mismo símbolo utilizado en las expresiones aritméticas para la suma.

- **Booleanas**. Su resultado podrá ser verdad o falso. Se construyen mediante los operadores relacionales y lógicos. Los operadores de relación son: igual (`=`), distinto (`<>`), menor que (`<`), mayor que (`>`), mayor o igual (`>=`), menor o igual (`<=`). Actúan sobre operandos del mismo tipo y siempre devuelven un resultado de tipo lógico. Los operadores lógicos básicos son: negación lógica (`no`), multiplicación lógica (`y`), suma lógica (`o`). Actúan sobre operandos de tipo lógico y devuelven resultados del mismo tipo, determinados por las tablas de verdad correspondientes a cada uno de ellos.

En los lenguajes de programación existen ciertas **funciones** predefinidas o internas que aceptan unos argumentos y producen un valor denominado resultado.

3.2.3.5 La operación de asignación

Esta operación se utiliza para dar valor a una variable en el interior de un algoritmo y permite almacenar en una variable el resultado de evaluar una expresión, perdiéndose cualquier otro valor previo que la variable pudiera tener.

Se supone que se efectúan conversiones automáticas de tipo cuando el tipo del valor a asignar a una variable es compatible con el de la variable y de tamaño menor. En estos casos se considera que el valor se convierte automáticamente al tipo de la variable. También se interpreta que se efectúa este tipo de conversiones cuando aparecen expresiones en las que intervienen operandos de diferentes tipos.

3.2.4 Estructura general y partes

El pseudocódigo es la herramienta más adecuada para la representación de algoritmos. El algoritmo en pseudocódigo debe tener una estructura muy clara y similar a un programa, de modo que se facilite al máximo su posterior codificación. Interesa por tanto conocer las secciones en las que se divide un programa, que habitualmente son:

- La cabecera.
- El cuerpo del programa:
  - Bloque de declaraciones.
  - Bloque de instrucciones.

La **cabecera** contiene el nombre del programa.

El **cuerpo del programa** contiene a su vez otras dos partes: el bloque de declaraciones y el bloque de instrucciones.

En el **bloque de declaraciones** se definen o declaran las constantes con nombre, los tipos de datos definidos por el usuario y también las variables. Suele ser conveniente seguir este orden. La declaración de tipos suele realizarse en base a los tipos estándar o a otros definidos

previamente, aunque también hay que considerar el método directo de enumeración de los valores constituyentes.

El **bloque de instrucciones** contiene las acciones a ejecutar para la obtención de los resultados. Las instrucciones o acciones básicas a colocar en este bloque se podrían clasificar del siguiente modo:

- De **inicio/fin**. La primera instrucción de este bloque será siempre la de inicio y la última la de fin.
- De **asignación**. Esta instrucción se utiliza para dar valor a una variable en el interior de un programa.
- De **lectura**. Toma uno o varios valores desde un dispositivo de entrada y los almacena en memoria en las variables que aparecen listadas en la propia instrucción.
- De **escritura**. Envía datos a un dispositivo de salida.

La representación de un algoritmo mediante pseudocódigo se muestra a continuación. Esta representación es muy similar a la que se empleará en la escritura al programar.

Algoritmo: nombre_del_algoritmo
//comentario_1
const
  nombre_de_constante_1 **<-** valor_1
  …
var
  tipo_de_dato_1: nombre_de_var_1, nombre_de_var_2, …
  …
inicio
  acción_1
  acción_2
  …
  //Algunas acciones podrían ser de escritura:
  escribir(expresión_cadena)
  //Algunas acciones podrían ser de lectura:
  leer(variable)
  //Algunas acciones podrían ser de asignación:
  nombre_de_variable **<-** expresión
  …
fin

La representación de un algoritmo mediante un diagrama de flujo utiliza unos símbolos estándar. Los principales símbolos son:

Ejemplo 3.1. Diseñar en diagrama de flujo y pseudocódigo un algoritmo que calcule e imprima el área de un triangulo a partir de la base y la altura dadas por el usuario.

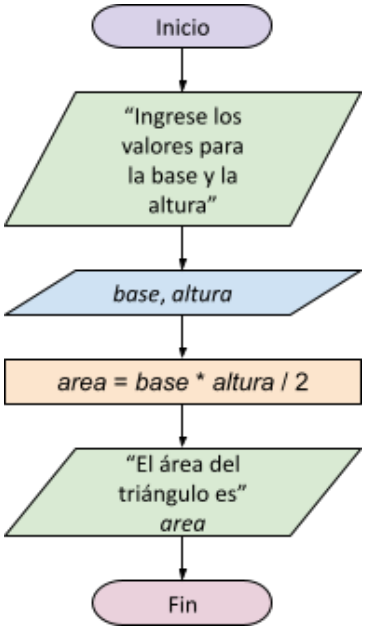
Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Linea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

En pseudocódigo:

Algoritmo: Área de un triángulo
//El usuario ingresa dos números reales asociados a la altura y la
//base de un triángulo y el algoritmo devuelve el valor asociado
//al área del triángulo
var
  real: base, altura, area
inicio
  escribir("Ingrese la altura")
  leer(altura)
  escribir("Ingrese la base")
  leer(base)
  area <- base * altura / 2
  escribir("El área del triángulo es:")
  escribir(area)
fin

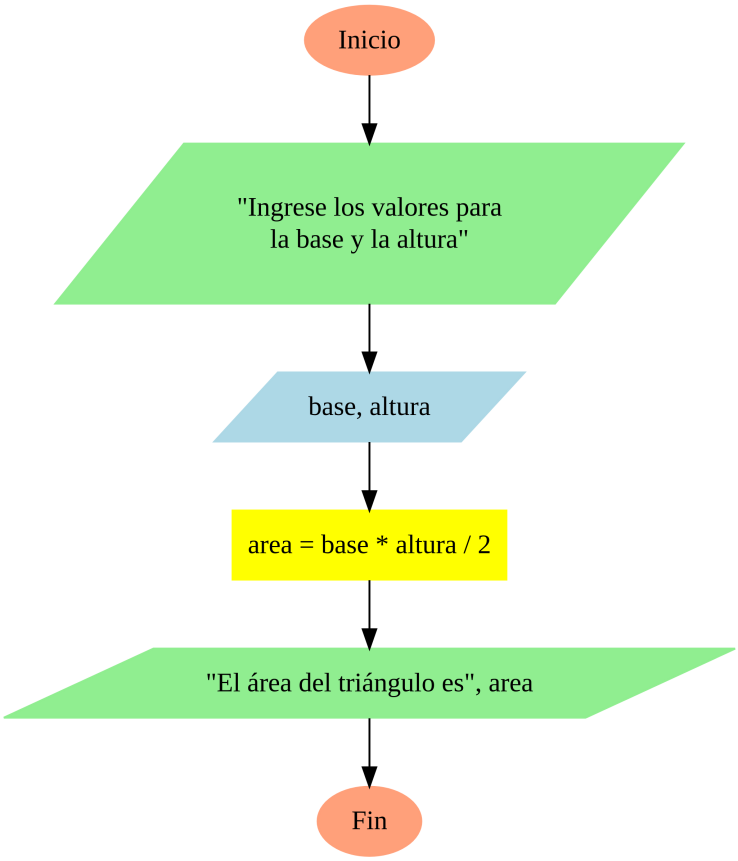
En diagrama de flujo:

- Con ***Google Drawings*** (aplicación en línea https://docs.google.com/drawings), con otro editor de gráficos o con herramientas de diagramación:



- Con ***Graphviz*** dentro de un documento ***Quarto*** (https://quarto.org/docs/authoring/diagrams.html#graphviz):

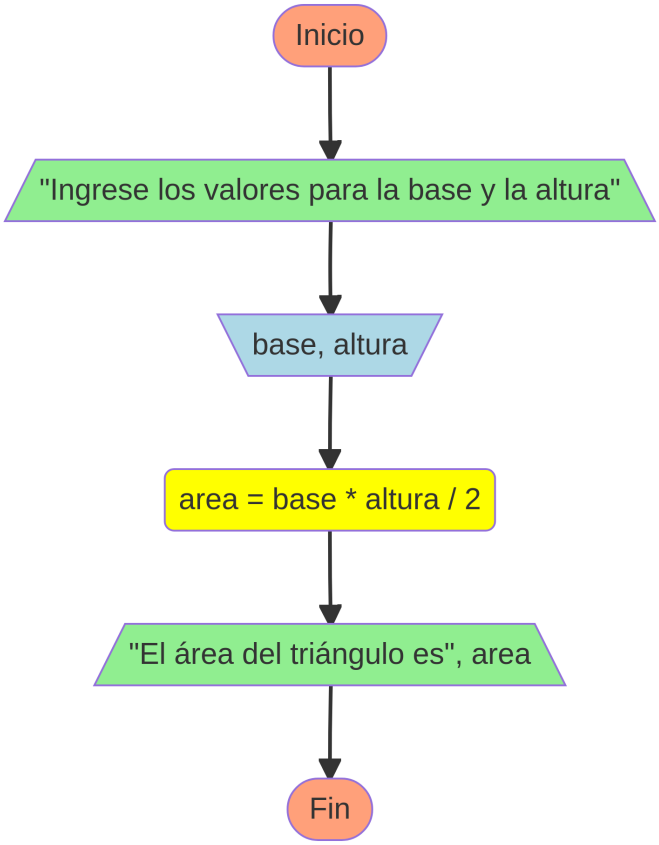
```
digraph area_triang {
  ini [label="Inicio", shape="ellipse", color=lightsalmon, style=filled];
  pedir_val [label="\"Ingrese los valores para \nla base y la altura\"", shape="parallelog
  leer_val [label="base, altura", shape="parallelogram", color=lightblue, style=filled];
  calc [label="area = base * altura / 2", shape="box", color=yellow, style=filled];
  escr_area [label="\"El área del triángulo es\", area", shape="parallelogram", color=ligh
  fin [label="Fin", shape="ellipse", color=lightsalmon, style=filled];
  ini -> pedir_val -> leer_val -> calc -> escr_area -> fin;
}
```



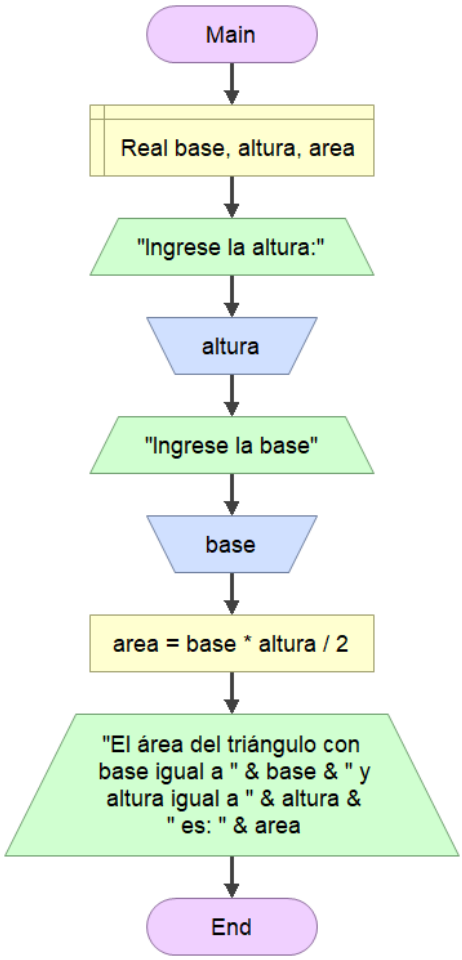
- Con ***Mermaid*** dentro de un documento ***Quarto*** (https://quarto.org/docs/authoring/diagrams.html#mermaid):

```
flowchart TB
  ini([Inicio]) -->
  pedir_val[/"#quot;Ingrese los valores para la base y la altura#quot;"\] -->
```

```
leer_val[\"base, altura"/] -->
calc(area = base * altura / 2) -->
escr_area[/"#quot;El área del triángulo es#quot;", area\] -->
fin([Fin])
style ini fill:lightsalmon
style fin fill:lightsalmon
style pedir_val fill:lightgreen
style escr_area fill:lightgreen
style leer_val fill:lightblue
style calc fill:yellow
```



- Con el programa ***Flowgorithm*** (el instalador se puede encontrar en http://www.flowgorithm.org/index.html):


Main
Real base, altura, area
"Ingrese la altura:"
altura
"Ingrese la base"
base
area = base * altura / 2
"El área del triángulo con base igual a " & base & " y altura igual a " & altura & " es: " & area
End

4 Programación estructurada

En esta sección se hará una revisión de lo que es la programación estructurada.

En esta revisión se hace mención al control del flujo de un algoritmo/programa mediante estructuras de secuencia, selección y repetición.

Así mismo, se espera que al finalizar las actividades de esta sección, el estudiante tenga clara la manera en que se hace el diseño de un algoritmo bajo el paradigma de **programación imperativa estructurada**, ya sea mediante el uso de un diagrama de flujo o de pseudocódigo.

4.1 ¿Qué es la programación estructurada?

La programación estructurada es un conjunto de técnicas para desarrollar algoritmos fáciles de escribir, verificar, leer y modificar. La programación estructurada utiliza:

- **Diseño descendente**. Consiste en diseñar los algoritmos en etapas, yendo de los conceptos generales a los de detalle. El diseño descendente se verá completado y ampliado con el diseño modular.
- **Recursos abstractos**. En cada descomposición de una acción compleja se supone que todas las partes resultantes están ya resueltas, posponiendo su realización para el siguiente refinamiento.
- **Estructuras básicas**. Los algoritmos deberán ser escritos utilizando únicamente tres tipos de estructuras básicas (secuencia, selección y repetición).

4.2 Teorema de Böhm y Jacopini

Para que la programación sea estructurada, los programas han de ser propios. Un programa se define como propio si cumple las siguientes características:

- Tiene un solo punto de entrada y uno de salida.
- Cada acción del algoritmo es accesible, es decir, existe al menos un camino que va desde el inicio, pasa por la acción y llega hasta el fin del algoritmo.
- No tiene bucles infinitos.

El teorema de Böhm y Jacopini [1] dice que: ***"un programa propio puede ser escrito utilizando únicamente tres tipos de estructuras: secuencial, selectiva y repetitiva"***. De este teorema se deduce que se han de diseñar los algoritmos empleando exclusivamente dichas estructuras, la cuales, como tienen un único punto de entrada y un único punto de salida, harán que nuestros programas sean propios.

4.3 Control del flujo de un programa

El flujo (orden en que se ejecutan las sentencias de un programa) es secuencial si no se especifica otra cosa. Este tipo de flujo significa que las sentencias se ejecutan en secuencia, una después de otra, en el orden en que se sitúan dentro del programa. Para cambiar esta situación se utilizan las estructuras que permiten modificar el flujo secuencial del programa. Así, las estructuras de selección se utilizan para seleccionar las sentencias que se han de ejecutar a continuación y las estructuras de repetición (repetitivas o iterativas) se utilizan para repetir un conjunto de sentencias.

4.3.1 Estructura secuencial

Una estructura secuencial es aquella en la cual una acción se ejecuta detrás de otra. El flujo del programa coincide con el orden físico en el que se sitúan las instrucciones:

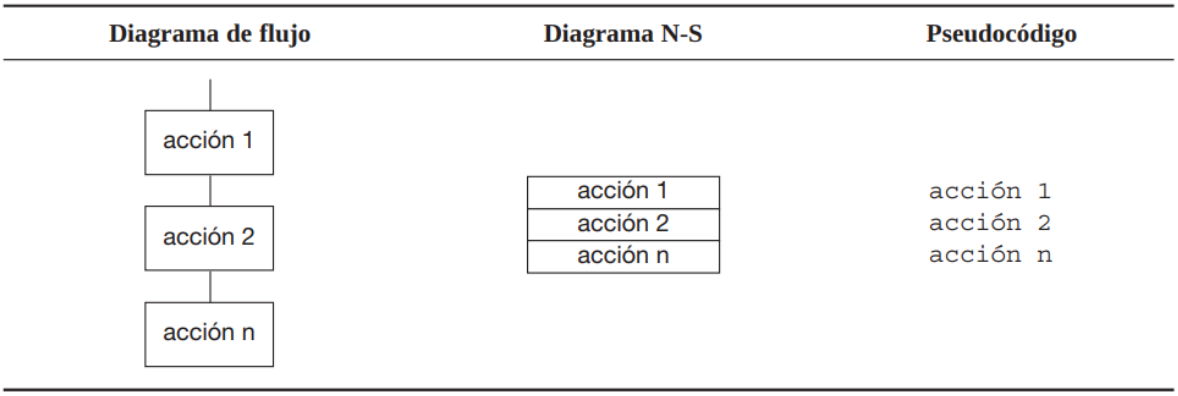


Figura 4.1: Representaciones de la estructura secuencial

4.3.2 Estructura selectiva

Una estructura selectiva es aquella en que se ejecutan unas acciones u otras según se cumpla o no una determinada condición.

Cuando el resultado de evaluar la condición es verdad se ejecutará una determinada acción o grupo de acciones y si el resultado es falso se ejecutará otra acción o grupo de acciones diferentes:

Diagrama de flujo	Diagrama N-S	Pseudocódigo

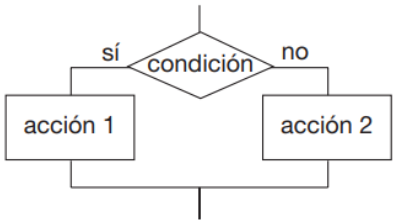
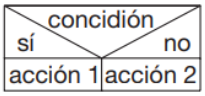
		**si** *<condición>* **entonces**
		  acción 1
		si_no
		  acción 2
		fin_si

Figura 4.2: Representaciones de la estructura selectiva

Si no es necesario ejecutar una determinada acción o acciones cuando el resultado es falso entonces se usa una versión simplificada o reducida de la anterior estructura selectiva:

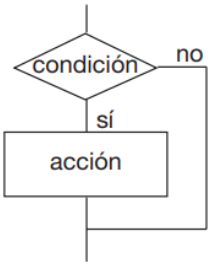
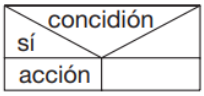
Diagrama de flujo	Diagrama N-S	Pseudocódigo
		**si** *<condición>* **entonces** acción **fin_si**

Figura 4.3: Representaciones de la estructura selectiva simple

En este último caso, también se puede decir que cuando la condición es verdad se ejecutará una determinada acción o grupo de acciones y si el resultado es falso simplemente no se ejecutará.

4.3.3 Estructura iterativa

Las acciones del cuerpo de la estructura iterativa se repiten mientras o hasta que se cumpla una determinada condición. Es frecuente el uso de contadores o banderas para controlar el que las repeticiones continúen o finalicen.

"mientras" (*while*) es la estructura iterativa básica y lo que la caracteriza es que las acciones del cuerpo de la estructura se realizan siempre que la condición sea cierta, finalizando las

repeticiones cuando la condición sea falsa. Además, siempre se pregunta por la condición antes de proceder a ejecutar el cuerpo de la estructura:

Diagrama de flujo

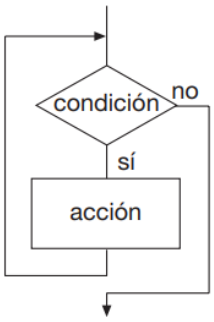
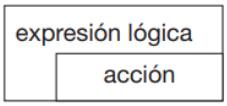


Diagrama N-S



Pseudocódigo

```
mientras <expresión_lógica> hacer
  acción
fin_mientras
```

Figura 4.4: Representaciones de la estructura repetitiva “mientras”

Usualmente se habla de más de una estructura iterativa, pero el Teorema de Böhm y Jacopini garantiza que todo algoritmo se puede escribir usando únicamente un tipo de estructura secuencial, un tipo de estructura selectiva y un tipo de estructura iterativa. Esto quiere decir que no se requiere más de un tipo de estructura iterativa y que siempre se puede reescribir el algoritmo para utilizar un solo tipo de estructura iterativa (lo que sea que se pueda hacer con los otros tipos de estructuras iterativas, también se puede hacer con un tipo de estructura iterativa seleccionado, por ejemplo usando únicamente la estructura iterativa “mientras”).

4.3.4 Anidamiento

Tanto las estructuras selectivas como las repetitivas pueden ir anidadas, unas en el interior de otras.

Una potencial estructura selectiva múltiple no es necesaria ya que sería equivalente a tener varias estructuras selectivas anidadas:

```
si condición_1 entonces
  acciones_1
si_no
  si condición_2 entonces
    acciones_2
```

```
  si_no
    si condición_3 entonces
      acciones_3
    si_no
      acciones_4
    fin_si
  fin_si
fin_si
```

Cuando se inserta una estructura iterativa dentro de otra, la estructura interna ha de estar totalmente incluida dentro de la externa:

```
mientras expresión_lógica_1 hacer
  acciones_1
  mientras expresión_lógica_2 hacer
    acciones_2
  fin_mientras
  acciones_3
fin_mientras
```

> ⚠ ¿Si la `expresión_lógica_1` es verdadera de manera consecutiva un número de veces n_1 y si la `expresión_lógica_2` es verdadera de manera consecutiva un número de veces n_2 en cada ocasión que `expresión_lógica_1` es verdadera, la acción o grupo de acciones del cuerpo de la estructura iterativa interna cuántas veces se ejecutará? ¿Si tengo una operación innecesaria en el cuerpo de dicha estructura, qué tanto se ve afectada la eficiencia del algoritmo? (¿qué tanto aumenta su costo computacional? ¿qué tanto aumenta su tiempo de ejecución?)

4.4 Ejemplos

Ejemplo 4.1. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que obtenga e imprima el valor absoluto ($|x|$) de un número real (x) dado por el usuario mediante el uso del teclado.

> 💡 Análisis
>
> Para cualquier número real x, el **valor absoluto** de x se denota por $|x|$ y se define como:
>
> $$|x| = \begin{cases} x & \text{Si } x \geq 0 \\ -x & \text{Si } x < 0 \end{cases}$$

Diseño en pseudocódigo

Algoritmo: Valor absoluto de un número real
var
  real: x
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo obtiene e imprime por pantalla el valor absoluto de un número real dado por el usuario mediante el uso del teclado")
  escribir("Por favor, ingrese un número real")
  leer(x)
  si x $<$ 0 **entonces**
    x <**-** - x
  fin_si
  escribir("El valor absoluto del número dado es" **&** x)
fin

Ejemplo 4.2. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que obtenga e imprima el resultado de aplicar la función signo $(\text{sign}(x))$ a un número real (x) dado por el usuario.

Análisis

La función signo de un número real x, denotada por $\text{sgn}(x)$, $\text{sign}(x)$ o $\text{signo}(x)$, se puede definir de la siguiente manera:

$$\text{sign}(x) = \begin{cases} 1 & \text{Si } x > 0 \\ 0 & \text{Si } x = 0 \\ -1 & \text{Si } x < 0 \end{cases}$$

Diseño en pseudocódigo

Algoritmo: Valor de la función signo evaluada en un número real
var
  real: x
  entero: res
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo obtiene el valor de la función signo evaluada en un número real dado")
  escribir("Por favor, ingrese un número real")

```
  leer(x)
  si x > 0 entonces
    res <- 1
  si_no
    si x < 0 entonces
      res <- -1
    si_no
      res <- 0
    fin_si
  fin_si
  escribir("La función signo evaluada en" & x & " es igual a " & res)
fin
```

Ejemplo 4.3. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que determine e imprima el mayor de dos números reales dados por el usuario. ¿Qué considera que debería hacer el algoritmo cuando los dos números dados son iguales? ¿hay una única alternativa o debo escoger entre varias posibles formas de atender esa situación?

Análisis

Si los dos números reales dados por el usuario son diferentes, es poco y bastante sencillo lo que hay que hacer, casi totalmente obvio.
Por otro lado, hay varias maneras de manejar la situación cuando los dos números dados son iguales. Una de ellas es seguir pidiendo, tantas veces como sea necesario, un número distinto a los recibidos hasta el momento; otra opción podría ser pedir, tantas veces como sea necesario, dos nuevos números; o incluso otra opción podría ser imprimir que ninguno de los dos es mayor y no hacer nada más.

Diseño en pseudocódigo

```
Algoritmo: Mayor de dos números reales
//En este algoritmo, si los dos números dados son iguales, entonces se pedirá
//tantas veces como sea necesario un número distinto a los recibidos
var
  real: x, y
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo identifica el mayor de dos valores dados")
  escribir("Por favor, ingrese un primer número real")
  leer(x)
  escribir("Por favor, ingrese un segundo número real, distinto al anterior")
```

```
  leer(y)
  mientras x == y hacer
    escribir("Los números dados son iguales. Por favor, ingrese un número real, distinto
a los dados previamente")
    leer(y)
  fin_mientras
  si x > y entonces
    escribir(x & "es el mayor de los dos valores ingresados")
  si_no
    escribir(y & "es el mayor de los dos valores ingresados")
  fin_si
fin
```

Ejemplo 4.4. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que obtenga e imprima el factorial ($k!$) de un número entero no negativo ($k \geq 0$). ¿Qué considera que debería hacer el algoritmo si el número entero dado por el usuario es negativo?

Análisis

El factorial de un número entero positivo k, denotado $k!$ se puede definir como el producto de todos los números enteros positivos menores o iguales que k:

$$k! = (2)(3)\cdots(k-2)(k-1)(k)$$

o lo que es lo mismo,

$$k! = (k)(k-1)(k-2)\cdots(3)(2)$$

Además, es necesario tener en cuenta que,

$$0! = 1$$

y es buena idea tener en cuenta que,

$$1! = 1 \qquad 2! = 2$$

Por otra parte, una forma en que el algoritmo puede atender la situación en la cual el usuario ingresa números enteros no negativos, es seguir pidiendo valores hasta que el usuario ingrese un número entero no negativo.

Diseño en pseudocódigo

```
Algoritmo: Factorial de un número entero no negativo
var
  entero: k, res
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo obtiene el factorial de un número entero no negativo dado")
  escribir("Por favor, ingrese un número entero no negativo")
  leer(k)
  mientras k < 0 hacer
    escribir("Por favor, ingrese un número entero NO NEGATIVO")
    leer(k)
  fin_mientras
  si k < 2 entonces
    res <- 1
  si_no
    res <- k
    mientras k > 2 hacer
      k <- k - 1
      res <- res * k
    fin_mientras
  fin_si
  escribir("El factorial del número dado es" & res)
fin
```

Ejemplo 4.5. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que determine e imprima si un número entero mayor que uno ($k > 1$) dado por el usuario es primo o no lo es.

Análisis

> 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.

Adicionalmente, se sabe que:

- 2 y 3 son números primos
- 2 es el único número par que es primo.
- Un número impar solamente tiene divisores impares.
- Si un número (k) NO es divisible por un número natural (i) mayor o igual que dos

y menor o igual que su raíz cuadrada ($2 \leq i \leq \sqrt{k} \equiv 2^2 \leq i^2 \leq k$) entonces es primo.

Diseño en pseudocódigo

```
Algoritmo: Es o no un número primo
var
  entero: k, i
  booleano: seguirBusc
  cadena: textoRes
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo determina si un número entero mayor que uno es primo o no lo es")
  escribir("Por favor, ingrese un número entero mayor que uno")
  leer(k)
  mientras k < 2 hacer
    escribir("Por favor, ingrese un número entero MAYOR QUE UNO")
    leer(k)
  fin_mientras
  //Se usará " " para los que son primos y " NO " para los que no
  textoRes <- " "
  si k > 3 entonces
    si k % 2 == 0 entonces
      escribir("2 SI es divisor de" & k)
      textoRes <- " NO "
    si_no
      escribir("2 NO es divisor de" & k)
      i <- 3
      seguirBusc <- true
      mientras seguirBusc hacer
        si k % i == 0 entonces
          escribir(i & " SI es divisor de " & k)
          textoRes <- " NO "
          seguirBusc <- false
        si_no
          escribir(i & " NO es divisor de " & k)
          i <- i + 2
          si i * i > k entonces
            seguirBusc <- false
          fin_si
```

```
      fin_si
    fin_mientras
  fin_si
 fin_si
 escribir("El número" & k & textoRes & "ES PRIMO")
fin
```

Ejemplo 4.6. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que obtenga e imprima una aproximación de la raíz cuadrada de un número real no negativo dado (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

Análisis

Los babilonios aproximaban raíces cuadradas haciendo cálculos mediante la media aritmética reiteradamente. En términos modernos, se trata de construir una sucesión $a_0, a_1, a_2, a_3, ...$ dada por:

$$a_n = \frac{1}{2}\left(a_{n-1} + \frac{c}{a_{n-1}}\right)$$

Puede demostrarse que esta sucesión matemática converge de manera que,

$$a_n \xrightarrow[n\to\infty]{} \sqrt{c}$$

Nos damos cuenta que,

$$\begin{aligned} a_n &= \frac{1}{2}\left(a_{n-1} + \frac{c}{a_{n-1}}\right) \\ &= \left(\frac{1}{2}\right) a_{n-1} + \frac{(c/2)}{a_{n-1}} \end{aligned}$$

y como punto de partida o valor inicial podemos utilizar:

$$\begin{aligned} a_1 &= \frac{1+c}{2} \\ &= \left(\frac{1}{2}\right) + (c/2) \end{aligned}$$

Diseño en pseudocódigo

Algoritmo: Raíz cuadrada de un número real no negativo
var

```
  entero: n
  real: c, cMedios, a, aNew, eps
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo obtiene una aproximación de la raíz cuadrada de un número
real no negativo dado (c). El algoritmo se detendrá cuando el cambio de una aproximación
a la siguiente sea inferior a un número real positivo cercano a cero (epsilon)")
  escribir("Por favor, ingrese un número real no negativo (c)")
  leer(c)
  si c == 0 o c == 1 entonces
    escribir("La raíz cuadrada de" & c & " es igual a " & c)
  si_no
    escribir("Por favor, ingrese un numero real positivo cercano a cero (epsilon)")
    leer(eps)
    cMedios <- c / 2
    n <- 2
    a <- 0.5 + cMedios //Este es a_1
    aNew <- 0.5 * a + cMedios / a
    mientras a − aNew > eps hacer
      n <- n + 1
      a <- aNew
      aNew <- 0.5 * a + cMedios / a
    fin_mientras
    escribir("La raíz cuadrada de" & c & " es aproximadamente igual a " & aNew)
    escribir("(con el epsilon dado igual a" & eps & ", la aproximación corresponde al
término" & n & " de la sucesión)")
  fin_si
fin
```

Ejemplo 4.7. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que calcule e imprima una aproximación de $\cos(x)$,

$$\cos(x) = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} x^{2k}$$

para un x dado por el usuario (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

Análisis

Se tiene que,

$$S_n = \left[\sum_{k=0}^{n}\left(\frac{(-1)^k}{(2k)!}x^{2k}\right)\right] \xrightarrow[n\to\infty]{} \cos(x)$$

de donde observamos que,

$$\begin{aligned} S_n &= \left[\sum_{k=0}^{n}\left(\frac{(-1)^k}{(2k)!}x^{2k}\right)\right] \\ &= \frac{1}{1}x^0 - \frac{1}{2}x^2 + \frac{1}{(4)(3)(2)}x^4 - \frac{1}{(6)(5)(4)(3)(2)}x^6 + \cdots \\ &= 1 - \frac{x^2}{2!} + \frac{(x^2)(x^2)}{(4)(3)(2!)} - \frac{(x^2)(x^4)}{(6)(5)(4!)} + \cdots \end{aligned}$$

es decir, la magnitud de cada término cumple que,

$$\begin{aligned} a_i &= \frac{x^i}{i!}, \text{ para } i = 2k = 0, 2, 4, ... \\ &= \frac{x^2}{(i)(i-1)}\frac{x^{i-2}}{(i-2)!} \\ &= \frac{x^2}{(i)(i-1)}a_{i-2} \end{aligned}$$

y estos términos se suman o restan de forma intercalada.
El algoritmo debe detenerse cuando:

$$\begin{aligned} |S_n - S_{n-1}| &< \varepsilon \\ a_i &< \varepsilon \end{aligned}$$

En este caso el valor absoluto no es necesario debido a que $a_i = \frac{x^i}{i!}$ para $i = 2n$ siempre es positivo.

Diseño en pseudocódigo (opción 1)

Algoritmo: Coseno de un número real
var
  entero: i
  real: x, xCuadrNegat, eps, term, res
inicio
  escribir(“Bienvenido”)

```
  escribir("Este algoritmo obtiene una aproximación del coseno de un número real dado (x). El algoritmo se detendrá cuando el cambio de una aproximación a la siguiente sea inferior a un número real positivo cercano a cero (epsilon)")
  escribir("Por favor, ingrese un número real no negativo (x)")
  leer(x)
  res <- 1
  si x != 0 entonces
    escribir("Por favor, ingrese un numero real positivo cercano a cero (epsilon)")
    leer(eps)
    xCuadrNegat <- - x * x
    i <- 2
    term <- xCuadrNegat / i
    res <- res + term
    mientras term > eps o term < -eps hacer
      i <- i + 2
      term <- term * xCuadrNegat / (i * (i - 1))
      res <- res + term
    fin_mientras
  fin_si
  escribir("El coseno de" & x & " es igual a " & res)
fin
```

Diseño en pseudocódigo (opción 2)

```
Algoritmo: Coseno de un número real
var
  entero: i
  real: x, xCuadr, eps, magnTerm, res
  booleano: restar
inicio
  escribir("Bienvenido")
  escribir("Este algoritmo obtiene una aproximación del coseno de un número real dado (x). El algoritmo se detendrá cuando el cambio de una aproximación a la siguiente sea inferior a un número real positivo cercano a cero (epsilon)")
  escribir("Por favor, ingrese un número real no negativo (x)")
  leer(x)
  res <- 1
  si x != 0 entonces
    escribir("Por favor, ingrese un numero real positivo cercano a cero (epsilon)")
    leer(eps)
    xCuadr <- x * x
```

```
    i <- 2
    magnTerm <- xCuadr / i
    restar <- true
    res <- res - magnTerm
    mientras magnTerm > eps hacer
      i <- i + 2
      magnTerm <- magnTerm * xCuadr / (i * (i - 1))
      si restar entonces
        restar <- false
        res <- res + magnTerm
      si_no
        restar <- true
        res <- res - magnTerm
      fin_si
    fin_mientras
  fin_si
  escribir("El coseno de" & x & " es igual a " & res)
fin
```

⚠ ¿Cuál es mejor, la opción 1 o la opción 2? ¿por qué?

Ejemplo 4.8. Realice el **análisis** y **diseño en pseudocódigo** de un algoritmo que obtenga e imprima una aproximación de una raíz de una función de los reales en los reales dada, usando el método de Newton-Raphson (teniendo en cuenta todas las consideraciones del método). Imagine que las funciones matemáticas se pueden declarar y leer como cualquier otro de los tipos de variables que ya hemos visto. Además, no olvide hacer todas las pruebas de escritorio que considere necesarias para garantizar que el diseño es eficaz y eficiente.

Análisis

La *serie de Taylor*, alrededor de un punto x_{n-1}, de la función $f(x)$ es,

$$f(x) = f(x_{n-1}) + f'(x_{n-1})(x - x_{n-1}) + \frac{f''(x_{n-1})}{2!}(x - x_{n-1})^2 + \cdots$$

Si utilizamos únicamente los dos primeros términos de la serie y asumimos que a partir de algún n, x_n va a estar muy cerca a una raíz de la función $f(x)$ entonces,

$$0 \approx f(x_n) = f(x_{n-1}) + f'(x_{n-1})(x_n - x_{n-1})$$

al despejar x_n, se tendría que,

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

y

$$x_n \xrightarrow[n\to\infty]{} x$$

para un x tal que $f(x) = 0$.
Es así que al usuario se le debe pedir la función f, la función f', un valor inicial x_0, al menos un ε positivo cercano a cero para determinar cuando detenerse y un máximo al que se espera que nunca llegue n, pero que sirva de tope máximo de iteraciones para evitar que el algoritmo se quede infinitamente buscando una solución.
Además, el algoritmo debe deternerse cuando:

- $\left|f'(x_n)\right| < \varepsilon_1$, para evitar los efectos negativos de una división por cero o por un número muy cercano a cero.
- $\left|f(x_n)\right| < \varepsilon_2$, que es cuando se logra el objetivo deseado, una aproximación de una raíz de la función f.
- $\left|x_n - x_{n-1}\right| < \varepsilon_3$, que es cuando x_n es aproximadamente igual a x_{n-1}, lo que quiere decir que a partir de ese x_n será muy poco lo que se pueda avanzar en lograr el objetivo deseado.

Diseño en pseudocódigo

Algoritmo: Método de Newton-Raphson para encontrar una raíz de un función
//Nos imaginaremos que las funciones matemáticas se pueden declarar y leer
//como cualquier otro tipo de variable
var
  función matemática de $\mathbb{R}$ **en** $\mathbb{R}$: f, df
  real: x0, x, xNew, eps, dfx, fx, delta
  entero: maxIter, n
  booleano: seguir
  cadena: msg
inicio
  escribir(“Bienvenido”)
  escribir(“Método de Newton-Raphson para encontrar una raíz de un función”)
  escribir(“Por favor, ingrese la función objetivo (f)”)
  leer(f)
  escribir(“Por favor, ingrese la derivada de la función objetivo (f’)”)
  leer(df)
  escribir(“Por favor, ingrese un numero real positivo menor que 0.01 (epsilon)”)
  leer(eps)
  escribir(“Por favor, ingrese un numero entero positivo (máximo de iteraciones)”)
  leer(maxIter)
  escribir(“Por favor, ingrese un numero real (x_0)”)

```
  leer(x0)
  xNew <- x0 //Para empezar el siguiente mientras con x0 = x = xNew
  n <- 0
  seguir <- true
  mientras seguir == true y n < maxIter hacer
    x <- xNew
    dfx <- df(x)
    si -eps < dfx y dfx < eps entonces
      msg <- "La derivada evaluada en x_n se acerca demasiado a cero. df(x_n) =" &
dfx
      seguir <- false
    si_no
      fx <- f(x)
      delta <- fx / dfx
      xNew <- x − delta
      n <- n + 1
      si -eps < fx y fx < eps entonces
        msg <- "Se encontró un x para el cual |f(x)| < epsilon. f(x_n) =" & fx
        seguir <- false
      si_no
        si -eps < delta y delta < eps entonces
          msg <- "No hubo un cambio superior a epsilon de x_n a x_{n+1}. x_n -
x_{n+1} =" & delta
          seguir <- false
        fin_si
      fin_si
    fin_si
  fin_mientras
  si seguir == true entonces
    msg <- "Se alcanzó el máximo de iteraciones:" & maxIter & ". x_0 =" & x0 & ". x_n
=" & x & ". x_{n+1} =" & xNew
  fin_si
  escribir(msg)
  escribir("x_0 =" & x0)
  escribir("x_n =" & x)
  escribir("f'(x_{n+1}) =" & df(xNew))
  escribir("f(x_{n+1}) =" & f(xNew))
  escribir("x_{n+1} =" & xNew)
fin
```

4.5 Ejercicios

Parte A

Realice el **análisis** y **diseño** de cada algoritmo requerido, **usando únicamente los elementos incluidos previamente en este material (por ejemplo, no use el operador ^ o ** para una potenciación, ya que ese elemento intencionalmente no fue incluido previamente)**:

1. Se convirtió en el Ejemplo 4.1.
2. Se convirtió en el Ejemplo 4.2.
3. Se convirtió en el Ejemplo 4.3.
4. Un algoritmo que le pida al usuario una temperatura y la escala de dicha temperatura, grados Centigrados o grados Fahrenheit, para calcular e imprimir la temperatura equivalente en la otra escala con respecto a la dada por el usuario (si recibe grados centigrados devolverá grados Fahrenheit y viceversa).
5. Un algoritmo que determine e imprima si la fecha del calendario gregoriano, asociada a un número de día, un número de mes y un número de año dados por el usuario, es una fecha válida o no.
6. Un algoritmo que imprima el número de día, de mes y de año correspondiente a la fecha del día siguiente, a partir de un número de día, número de mes y número de año dados por el usuario para una fecha válida del calendario gregoriano.
7. Se convirtió en el Ejemplo 4.5.
8. Un algoritmo que obtenga e imprima el máximo común divisor de dos números enteros dados por el usuario.
9. Un algoritmo que obtenga e imprima el mínimo común múltiplo de dos números enteros dados por el usuario.
10. Se convirtió en el Ejemplo 4.4.
11. Un algoritmo que obtenga e imprima la potencia k-ésima de un número real x, para k un número entero, con x y k dados por el usuario.
12. Se convirtió en el Ejemplo 4.6.
13. Un algoritmo que calcule e imprima una aproximación de $\exp(x)$,

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

para un x dado por el usuario (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

14. Un algoritmo que calcule e imprima una aproximación de $\ln(x)$,

$$\ln(x) = 2\sum_{k=0}^{\infty}\left[\frac{1}{2k+1}\left(\frac{x-1}{x+1}\right)^{2k+1}\right]$$

para un $x > 0$ dado por el usuario (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

15. Un algoritmo que calcule e imprima una aproximación de $\sin(x)$,

$$\sin(x) = \sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)!}x^{2k+1}$$

para un x dado por el usuario (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

16. Se convirtió en el Ejemplo 4.7.

17. Un algoritmo que calcule e imprima una aproximación de $\arcsin(x)$,

$$\arcsin(x) = \sum_{k=0}^{\infty}\frac{(2k)!}{4^k(k!)^2(2k+1)}x^{2k+1}$$

para un $-1 < x < 1$ dado por el usuario (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

18. Calcule e imprima una aproximación para el número π, teniendo en cuenta que,

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

y que

$$\arctan(x) = \sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}x^{2k+1}$$

(¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

19. Un algoritmo que calcule e imprima una aproximación para el número áureo φ https://youtu.be/aopHcOm7a-w. Tenga en cuenta que,

$$\varphi = \lim_{n \to \infty} \frac{F_{n+1}}{F_n}$$

donde F_n denota el n-ésimo número de la serie de Fibonacci https://youtu.be/B6ztvqvZTsk (¿cuándo se consideraría que ya se llegó a una aproximación suficientemente buena? ¿en relación a ello, qué le pido al usuario como datos de entrada?).

20. Un algoritmo que reciba las notas de una cantidad no predeterminada de estudiantes de cierta asignatura (dejando de recibir notas en el momento en que se introduzca un valor de nota fuera del rango válido entre 0 y 5) y que imprima la cantidad de estudiantes que aprobaron la asignatura y la cantidad de estudiantes que la perdieron.

21. Un algoritmo que reciba las notas de una cantidad no predeterminada de estudiantes de cierta asignatura (dejando de recibir notas en el momento en que se introduzca un valor de nota fuera del rango válido entre 0 y 5) y que imprima el mínimo y el máximo de dichas notas.

22. Un algoritmo que reciba una cantidad no predeterminada de número reales, en donde se le debe preguntar al usuario si desea de dejar de ingresar valores antes de leer cada valor, y que en el momento en que el usuario informe que desea dejar de ingresar valores se impriman la media y la varianza (poblacionales) de los valores ingresados hasta ese momento (¿qué debe hacer mi algoritmo con las potenciales situaciones en donde el usuario ingresa un único valor o cuando no ingresa valor alguno?). Recuerde que:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i$$
$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 = \left(\frac{1}{N} \sum_{i=1}^{N} x_i^2 \right) - \mu^2$$

Parte B

Realice el **análisis** y **diseño en pseudocódigo** de cada algoritmo requerido, **usando únicamente los elementos incluidos previamente en este material**. Imagine que las funciones matemáticas se pueden declarar y leer como cualquier otro de los tipos de variables que ya hemos visto. Además, no olvide hacer todas las pruebas de escritorio que considere necesarias para garantizar que su diseño es eficaz y eficiente:

1. Un algoritmo que obtenga e imprima una aproximación de una raíz, entre dos números reales dados, de una función de los reales en los reales dada, usando el método de bisección (teniendo en cuenta todas las consideraciones del método).

2. Se convirtió en el Ejemplo 4.8.

3. Un algoritmo que obtenga e imprima una aproximación de la derivada (derivada numérica) evaluada en un número real dado, de una función de los reales en los reales dada (bien definida y derivable alrededor del número real dado).

4. Un algoritmo que obtenga e imprima una aproximación de la integral definida entre dos números reales dados, de una función de los reales en los reales dada (bien definida e integrable en el intervalo de integración), usando la regla del trapecio compuesta o regla de los trapecios.

5. Un algoritmo que obtenga e imprima una aproximación de la integral definida entre dos números reales dados, de una función de los reales en los reales dada (bien definida e integrable en el intervalo de integración), usando la regla o método de Simpson (1/3) compuesto.

5 Programación procedimental

En esta sección se hará una revisión de lo que es la programación procedimental. La programación procedimental o programación por procedimientos usualmente es una manera de darle un ingrediente más al paradigma de programación imperativa estructurada. Muchas veces es aplicable tanto en lenguajes de programación de bajo nivel como en lenguajes de alto nivel. Esta técnica consiste en englobar una serie de instrucciones dentro de un procedimiento o función y llamarlo cada vez que se requiera. Por otra parte, la resolución de problemas complejos se facilita considerablemente si estos se dividen en problemas más pequeños (subproblemas). La solución de estos subproblemas se realiza mediante *subalgoritmos*. Estos subalgoritmos o subprogramas están diseñados para realizar alguna tarea específica y pueden ser de dos tipos: funciones o procedimientos.

En esta revisión se hace mención a la programación modular, a las funciones y procedimientos como subalgoritmos, al ámbito de las variables y a los subalgoritmos recursivos.

Se espera que al finalizar las actividades de esta sección, el estudiante tenga clara la manera en que se hace el diseño de un algoritmo bajo el paradigma de programación imperativa estructurada procedimental, ya sea mediante el uso de un diagrama de flujo o de pseudocódigo.

5.1 Programación modular

El diseño descendente consiste en solucionar un problema mediante su descomposición en problemas más pequeños y más sencillos. La programación modular consiste en resolver de forma independiente los subproblemas resultantes de una descomposición. La programación modular amplía y completa el diseño descendente como método de resolución de problemas y permite proteger la estructura de la información asociada a un subproblema.

Cuando se trabaja de este modo, existirá un algoritmo principal o conductor que transferirá el control a los distintos subalgoritmos, los cuales, cuando terminen su tarea, devolverán el control al algoritmo que los llamó. Los subalgoritmos deberán ser de menor tamaño y menor dificultad al algoritmo principal, seguirán todas las reglas de la programación estructurada y podrán ser representados con las herramientas de programación habituales.

El empleo de esta técnica facilita notoriamente el diseño de los programas. Algunas ventajas significativas son:

- Varios programadores podrán trabajar simultáneamente en la confección de un algoritmo, repartiéndose las distintas partes del mismo, ya que los módulos son independientes.
- Se podrá modificar un módulo sin afectar a los demás.
- Las tareas o subalgoritmos sólo se escribirán una vez, aunque sean requeridos en distintas ocasiones en el cuerpo de uno o varios algoritmos.

5.2 Subalgoritmos

5.2.1 Funciones

Una función toma uno o más valores, denominados argumentos o parámetros y, según el valor de éstos, devuelve un resultado a nombre de la función. La definición de una función expresada en pseudocódigo tendría la siguiente forma:

tipo_de_dato_devuelve: **función** nombre_función(tipo_de_dato_1: parámetro_1, tipo_de_dato: demás_parámetros)
inicio_función
  ...
  acciones
  ...
  devolver(expresión)
fin_función

Para invocar a una función se utiliza su nombre seguido por los parámetros entre paréntesis. La llamada a una función se podrá colocar en cualquier instrucción en donde se pueda usar una expresión.

5.2.2 Procedimientos

La declaración de un procedimiento es similar a la de una función. Las pequeñas diferencias son debidas a que el nombre del procedimiento no se encuentra asociado a ningún resultado. La definición de un procedimiento expresada en pseudocódigo tendría la siguiente forma:

procedimiento nombre_procedimiento(tipo_de_dato_1: parámetro_1, tipo_de_dato: demás_parámetros)
inicio_procedimiento
  ...
  acciones
  ...
fin_procedimiento

Un procedimiento se invoca o se llama de la misma manera que una función, pero al no devolver un resultado, un procedimiento solamente se puede usar como instrucción del algoritmo principal y no como parte de una expresión.

5.3 Ambito de las variables

Una variable es global cuando el ámbito en el que dicha variable se conoce es el programa completo. Se consideran como variables globales aquellas que hayan sido declaradas en el algoritmo principal y como locales las declaradas dentro de algún subalgoritmo.

Toda variable que se utilice en un procedimiento o función debe haber sido declarada en el mismo. De esta forma todas las variables del procedimiento serán locales y la comunicación con el programa principal se realizará exclusivamente a través de los parámetros. Al declarar una variable en un subprograma no importa que ya existiera otra con el mismo nombre en el programa principal; ambas serán distintas y, cuando nos encontremos en el subprograma, sólo tendrá vigencia la declaración que hayamos efectuado en él. Trabajando de esta forma obtendremos la independencia modular o de algoritmos deseada (ya sean principales o subalgoritmos).

Ejemplo 5.1. Realice el **análisis** y **diseño** de:

- Un subalgoritmo (función) que reciba un número real y que devuelva el valor absoluto ($|x|$) del número recibido. La única tarea de este subalgoritmo es obtener y devolver el valor absoluto del número que reciba, no le corresponde interactuar de manera alguna con el usuario.
- Un subalgoritmo (procedimiento) que le pida al usuario su nombre, lo salude y le de la bienvenida.
- Un algoritmo principal que le pida al usuario su nombre, lo salude, le dé la bienvenida y, tantas veces como desee el usuario, le pida un número real y le informe su respectivo valor absoluto. Este algoritmo principal debe utilizar los dos subalgoritmos anteriores y se debe encargar de todo aquello que no sea tarea de los subalgoritmos.

Análisis

Para cualquier número real x, el **valor absoluto** de x se denota por $|x|$ y se define como:

$$|x| = \begin{cases} x & \text{Si } x \geq 0 \\ -x & \text{Si } x < 0 \end{cases}$$

Diseño en pseudocódigo

```
//Aquí pondré el subalgoritmo que obtiene el valor absoluto
real: función valAbs(real: x)
inicio_función
  si x < 0 entonces
    x <- -x
  fin_si
  devolver(x)
fin_función

//Aquí pondré el subalgoritmo que le pide al usuario su nombre, lo saluda y le da la bienvenida
procedimiento saludo()
var
  cadena: nombre
inicio_procedimiento
  escribir("Por favor, ingrese su nombre")
  leer(nombre)
  escribir("Buen día" & nombre & ", sea usted bienvenido")
fin_procedimiento

//Aquí pondré el programa principal
Algoritmo: Saluda y obtiene el valor absoluto de cada número que ingrese el usuario
var
  real: x
  cadena: continuar
inicio
  saludo()
  escribir("A continuación se procede a probar el subalgoritmo que obtiene el valor absoluto de un número real dado")
  continuar <- " "
  mientras continuar != "No" hacer
    escribir("Por favor, ingrese un número real")
    leer(x)
    escribir("El valor absoluto de" & x & " es " & valAbs(x))
    escribir("¿Desea continuar probando el subalgoritmo (para finalizar, ingrese la palabra: No)?")
    leer(continuar)
  fin_mientras
fin
```

5.4 Recursión

Un elemento es recursivo si forma parte de sí mismo o interviene en su propia definición. El mecanismo necesario para poder expresar los programas recursivamente es el subalgoritmo. La mayoría de los lenguajes de programación admiten que un procedimiento o función haga referencia a sí mismo dentro de su definición, esto se denomina recursividad directa.

La recursión se puede considerar como una alternativa a la iteración y resulta muy útil cuando se trabaja con problemas o estructuras definidos en modo recursivo (sin embargo las soluciones iterativas son más cercanas a la estructura de funcionamiento de un computador).

Para comprender la recursividad se deben tener en cuenta las siguientes premisas:

- Un método recursivo debe establecer la condición o condiciones de salida.
- Cada llamada recursiva debe aproximar hacia el cumplimiento de la o las condiciones de salida.
- Cuando se llama a un procedimiento o función los parámetros y las variables locales toman nuevos valores, y el procedimiento o función trabaja con estos nuevos valores y no con los de anteriores llamadas.
- Cada vez que se llama a un procedimiento o función los parámetros de entrada y variables locales son almacenados en las siguientes posiciones libres de memoria y cuando termina la ejecución del procedimiento o función son accedidos en orden inverso a como se introdujeron.
- El espacio requerido para almacenar los valores crece conforme a los niveles de anidamiento de las llamadas.
- La recursividad puede ser directa e indirecta. La recursividad indirecta se produce cuando un procedimiento o función hace referencia a otro el cual contiene, a su vez, una referencia directa o indirecta al primero.

Todo algoritmo recursivo puede ser convertido en uno iterativo equivalente, aunque para ello se puede requerir la utilización de ciertas estructuras de datos que permitan el almacenamiento adecuado de ciertos datos.

Ejemplo 5.2. Realice el **análisis** y **diseño** de:

- Un subalgoritmo **RECURSIVO** que reciba un número entero y que devuelva el factorial del número recibido. La única tarea de este subalgoritmo es obtener y devolver el factorial del número que reciba, no le corresponde interactuar de manera alguna con el usuario (¿qué considera que deba hacer el subalgoritmo cuando el número entero que recibe es negativo, teniendo en cuenta que no es trabajo del subalgoritmo interactuar con el usuario?).
- Un algoritmo principal que, tantas veces como desee el usuario, le pida un número entero no negativo y le informe su respectivo factorial. Este algoritmo principal debe utilizar el subalgoritmo anterior y se debe encargar de todo aquello que no sea tarea del mismo.

Análisis

El factorial de un número entero positivo k, denotado $k!$ se puede definir como el producto de todos los números enteros positivos menores o iguales que k:

$$\begin{aligned} k! &= (k)(k-1)(k-2)\cdots(3)(2) \\ &= (k)(k-1)! \end{aligned}$$

Además, es necesario tener en cuenta que,

$$0! = 1$$

y es buena idea tener en cuenta que,

$$1! = 1 \qquad 2! = 2$$

Diseño en pseudocódigo

```
//Aquí pondré el subalgoritmo recursivo que obtiene el factorial
entero: función factRecur(entero: k)
var
  entero: res
inicio_función
  si k < 2 entonces
    res <- 1
  si_no
    res <- k * factRecur(k-1)
  fin_si
  devolver(res)
fin_función

//Aquí pondré el subalgoritmo que le pide al usuario su nombre, lo saluda y le da la bienvenida
procedimiento saludo()
var
  cadena: nombre
inicio_procedimiento
  escribir("Por favor, ingrese su nombre")
  leer(nombre)
  escribir("Buen día" & nombre & ", sea usted bienvenido")
fin_procedimiento
```

```
//Aquí pondré el programa principal
Algoritmo: Saluda y obtiene el factorial (subalgoritmo recursivo) de cada número que ingrese el usuario
var
  entero: k
  cadena: continuar
inicio
  saludo()
  escribir("A continuación se procede a probar el subalgoritmo recursivo que obtiene el factorial de un número entero no negativo dado")
  continuar <- " "
  mientras continuar != "No" hacer
    escribir("Por favor, ingrese un número entero no negativo")
    leer(k)
    escribir("El factorial de" & k & " es " & factRecur(k))
    escribir("¿Desea continuar probando el subalgoritmo (para finalizar, ingrese la palabra: No)?")
    leer(continuar)
  fin_mientras
fin
```

5.5 Ejercicios

Realice el **análisis** y **diseño** de los subalgoritmos y el algoritmo principal requeridos, **usando únicamente los elementos incluidos previamente en este material (por ejemplo, no use el operador ^ o ** para una potenciación, ya que ese elemento intencionalmente no fue incluido previamente)**:

1. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba un número real y que devuelva la función signo $(\text{sign}(x))$ evaluada en el número recibido.
2. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba dos números reales y que devuelva el mayor de los dos. ¿Qué considera que deba hacer el subalgoritmo cuando los dos valores que recibe son iguales, teniendo en cuenta que no es trabajo del subalgoritmo interactuar con el usuario?.
3. Un algoritmo principal para probar, tantas veces como desee el usuario:
   - Un subalgoritmo que recibe un valor asociado a grados Centigrados y devuelve su equivalente en grados Fahrenheit

- Un subalgoritmo que recibe un valor asociado a grados Fahrenheit y devuelve su equivalente en grados Centigrados

El algoritmo principal se debe encargar de pedir al usuario una temperatura y la escala de dicha temperatura, grados Centigrados o grados Fahrenheit, para luego obtener mediante el subalgoritmo respectivo la temperatura equivalente en la otra escala con respecto a la dada por el usuario (si recibe grados Centigrados devolverá grados Fahrenheit y viceversa).

4. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba un número de día, un número de mes y un número de año, y que devuelva el valor booleano que corresponda dependiendo de si la fecha asociada a esos números es una fecha válida (`True`) o no (`False`) del calendario gregoriano.

5. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba un número de día, un número de mes y un número de año de una fecha del calendario gregoriano, y que devuelva la cadena que corresponda a la fecha del día siguiente usando el formato: "año/mes/día".

6. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba un número entero y que devuelva un valor que indique si el número es primo o no lo es. ¿Qué considera que deba hacer el subalgoritmo cuando el número entero que recibe es menor que dos, teniendo en cuenta que no es trabajo del subalgoritmo interactuar con el usuario?

7. Un algoritmo principal para probar, tantas veces como desee el usuario:

   - Un subalgoritmo **ITERATIVO** que reciba dos números enteros, y que devuelva el máximo común divisor de los números recibidos.
   - Un subalgoritmo **RECURSIVO** que reciba dos números enteros, y que devuelva el máximo común divisor de los números recibidos
   - Un subalgoritmo que reciba dos números enteros y que utilice uno de los subalgoritmos anteriores (por ejemplo el iterativo) para devolver el mínimo común múltiplo de los números recibidos.

8. Un algoritmo principal para probar, tantas veces como desee el usuario: un subalgoritmo **ITERATIVO** y uno **RECURSIVO** que reciban un número entero k y que devuelvan el término k-ésimo de la sucesión de los números de Fibonacci https://youtu.be/B6ztvqvZTsk.

9. Un algoritmo principal para probar, tantas veces como desee el usuario: un subalgoritmo **ITERATIVO** y uno **RECURSIVO** que reciban un número real x y un número entero k, y que devuelvan el número real x^k.

10. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba dos números reales, x y ε, y que devuelva la aproximación de $\sqrt{x}$ que corresponda a partir del valor dado para ε ¿Qué considera que deba hacer el subalgoritmo cuando el número real que recibe es negativo, teniendo en cuenta que no es trabajo del subalgoritmo interactuar con el usuario?

11. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba dos números reales, asociados a un x y un ε, y que devuelva la aproximación de $\sqrt[3]{x}$ dada por el valor de ε.

12. Un algoritmo principal para probar, tantas veces como desee el usuario, un subalgoritmo que reciba dos números reales, asociados a un x y un ε, y que devuelva la aproximación de $\sqrt[k]{x}$ dada por el valor de ε.

6 PIEP en Python

En esta sección se hará una revisión de los elementos básicos y de programación imperativa estructurada procedimental (PIEP) que requiere una implementación en Python (obviamente, Python como lenguaje de programación y **NO** como herramienta de cálculo o de análisis de datos).

En esta revisión se hace a una pequeña introducción a Python, a sus elementos básicos, a sus principales estructuras de control, a la creación de subprogramas, y al manejo y creación de scripts y módulos.

Se espera que al finalizar las actividades de esta sección, el estudiante entienda y tenga clara la manera en que un algoritmo, diseñado bajo el paradigma de programación imperativa estructurada procedimental, se puede implementar mediante el uso del lenguaje de programación Python.

> **i** Preparación de clase
>
> - Para las siguientes secciones, lea **todo** y ejecute **todo** el código que allí se incluye, haciendo **todas** las pruebas, cambios y experimentos que se les puedan ocurrir sobre dicho código.
>
> En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó, ejecutó, probó y experimentó; incluya su discusión, reflexiones y conclusiones al respecto; exponga lo que no entendió e intente encontrar por su cuenta respuestas a las preguntas que le surgieron, para poder compartirlas en clase.

6.1 Elementos básicos

Cuaderno computacional en Google Colaboratory:
Elementos básicos en Python

6.2 Estructuras de control

Cuaderno computacional en Google Colaboratory:
Estructuras de control en Python

6.3 Subprogramas

Cuaderno computacional en Google Colaboratory:
Creación de subprogramas en Python

6.4 Scripts y módulos

Cuaderno computacional en Google Colaboratory:
Creación y manejo de scripts y módulos en Python

6.5 Ejemplos

Ejemplo 6.1. Haga un adecuado análisis, diseño e implementación en Python de un subprograma que devuelva el **valor absoluto** de un número real dado. Además, implemente un programa principal que se encargue de interactuar con el usuario y que le permita probar el o los subprogramas requeridos con los valores que desee, tantas veces como lo desee.

Análisis y diseño

Naturalmente, para que la programación sea modular, el programa principal y el subprograma tienen que tener tareas claramente diferenciadas. Por otra parte, el subprograma debe ser totalmente funcional por sí mismo en caso de que se desee utilizar por fuera del programa principal solicitado.
Por esta ocasión, utilizaremos `isinstance()` para que el subprograma tenga un comportamiento en particular cuando no recibe un número real; devolverá `nan` cuando el valor recibido no sea de tipo `int` o `float`.
La siguiente implementación se obtuvo, partiendo del análisis y diseño que se hizo en el Ejemplo 5.1, y usando la sintaxis de Python.

Implementación en Python del subprograma solicitado

Subprograma solicitado (opción 1):

```
# Subprograma solicitado (opción 1)
def mi_val_abs(x):
  '''Aquí va la documentación y ayuda de la función `mi_val_abs`'''
  if isinstance(x, (int, float)):
    if x < 0:
      x = -x
  else:
    print(" ", "-"*80)
    print("  No se recibió un valor `int` o `float`, entonces se devolverá: `nan`")
    print(" ", "-"*80)
    x = float("nan")
  return x
```

Subprograma solicitado (opción 2):

```
# Subprograma solicitado (opción 2)
def mi_val_abs_v2(x):
  '''Aquí va la documentación y ayuda de la función `mi_val_abs_v2`'''
  if isinstance(x, (int, float)):
    x = -x if x < 0 else x
  else:
    print(" ", "-"*80)
    print("  No se recibió un valor `int` o `float`, entonces se devolverá: `nan`")
    print(" ", "-"*80)
    x = float("nan")
  return x
```

Implementación en Python del programa principal

Prueba rápida de los subprogramas (sin un programa principal):

```
  # Una prueba rápida de los subprogramas (sin un programa principal)
  x = -1.234
  print("  mi_val_abs(", x, ") = ", sep="")
  print(mi_val_abs(x))
  print("  mi_val_abs_v2(", x, ") = ", sep="")
  print(mi_val_abs_v2(x))
  print("  abs(", x, ") = ", sep="")
  print(abs(x))
```

```
  mi_val_abs(-1.234) =
1.234
  mi_val_abs_v2(-1.234) =
1.234
  abs(-1.234) =
1.234
```

```
  # Otra prueba rápida de los subprogramas (sin un programa principal)
  x = "Hola"
  print("  mi_val_abs(", x, ") = ", sep="")
  print(mi_val_abs(x))
  print("  mi_val_abs_v2(", x, ") = ", sep="")
  print(mi_val_abs_v2(x))
  print("  abs(", x, ") = ", sep="")
  print(abs(x))
```

```
  mi_val_abs(Hola) =
  --------------------------------------------------------------------------------
  No se recibió un valor `int` o `float`, entonces se devolverá: `nan`
  --------------------------------------------------------------------------------
nan
  mi_val_abs_v2(Hola) =
  --------------------------------------------------------------------------------
  No se recibió un valor `int` o `float`, entonces se devolverá: `nan`
  --------------------------------------------------------------------------------
nan
  abs(Hola) =

TypeError: bad operand type for abs(): 'str'
```

Programa principal que permite probar y comparar el subprograma solicitado:

```
# Programa principal que prueba y compara el subprograma solicitado
print("Este programa prueba las funciones `mi_val_abs` y `mi_val_abs_v2`")
continuar = ""
while continuar != "No":
  x = float(input("\nPor favor, ingrese un número real:"))
  print("mi_val_abs(", x, ") = ", mi_val_abs(x), sep="")
  print("mi_val_abs_v2(", x, ") = ", mi_val_abs_v2(x), sep="")
  print("abs(", x, ") = ", abs(x), sep="")
  continuar = input("\n¿Desea continuar probando (para finalizar, ingrese la palabra: No
print("Fin")
```

Ejemplo 6.2. Haga un adecuado análisis, diseño e implementación en Python de un subprograma que reciba un número entero, y que devuelva el valor booleano que corresponda dependiendo de si el número recibido **es primo** (`True`) **o no** (`False`). Además, implemente un programa principal que se encargue de interactuar con el usuario y que le permita probar el o los subprogramas requeridos con los valores que desee, tantas veces como lo desee.

Análisis y diseño

Naturalmente, para que la programación sea modular, el programa principal y el subprograma tienen que tener tareas claramente diferenciadas. Por otra parte, el subprograma debe ser totalmente funcional por sí mismo en caso de que se desee utilizar por fuera del programa principal solicitado. El subprograma devolverá `nan` si recibe un número entero menor o igual a uno. Sin embargo, el subprograma no tendrá programado un comportamiento en particular para cuando recibe algo diferente a un número entero (algo diferente a un valor de tipo `int`).
La siguiente implementación se obtuvo: partiendo del análisis y diseño que se hizo en el Ejemplo 4.5; separando y diferenciando adecuadamente la tarea del subprograma con respecto a las del programa principal; y usando la sintaxis de Python.

Implementación en Python del subprograma solicitado

```
# Subprograma solicitado
def mi_es_primo(k):
  '''Aquí va la documentación y ayuda de la función `mi_es_primo`'''
  if k > 1:
    res = True
    if k > 3:
      if k % 2 == 0:
        res = False
      else:
        i = 3
        seguir = True
        while seguir:
          if k % i == 0:
            res = False
            seguir = False
          else:
            i = i + 2
            if i * i > k:
              seguir = False
  else:
    print(" ", "-"*80)
    print("  No se recibió un entero mayor que uno, entonces se devolverá: `nan`")
    print(" ", "-"*80)
    res = float("nan")
  return res
```

Implementación en Python del programa principal

Prueba rápida del subprograma (sin un programa principal):

```
# Una prueba rápida de los subprogramas (sin un programa principal)
# Para poder comparar importaré la función `isprime()` del módulo `sympy`
from sympy import isprime
k = 1234567891
print("  mi_es_primo(", k, ") = ", sep="")
print(mi_es_primo(k))
print("  isprime(", k, ") = ", sep="")
print(isprime(k))
```

```
  mi_es_primo(1234567891) =
True
  isprime(1234567891) =
True
```

```
  # Otra prueba rápida de los subprogramas (sin un programa principal)
  # Para poder comparar importaré la función `isprime()` del módulo `sympy`
  from sympy import isprime
  k = -1234
  print("  mi_es_primo(", k, ") = ", sep="")
  print(mi_es_primo(k))
  print("  isprime(", k, ") = ", sep="")
  print(isprime(k))
```

```
  mi_es_primo(-1234) =
  ------------------------------------------------------------------------------
  No se recibió un entero mayor que uno, entonces se devolverá: `nan`
  ------------------------------------------------------------------------------
nan
  isprime(-1234) =
False
```

Programa principal que permite probar el subprograma solicitado:

```
  # Programa principal que prueba el subprograma solicitado
  # Para poder comparar importaré la función `isprime()` del módulo `sympy`
  from sympy import isprime
  print("Este programa prueba la función `mi_es_primo`")
  continuar = ""
  while continuar != "No":
    k = int(input("\nPor favor, ingrese un número entero:"))
    print("  mi_es_primo(", k, ") = ", sep="")
    print(mi_es_primo(k))
    print("  isprime(", k, ") = ", sep="")
    print(isprime(k))
    continuar = input("\n¿Desea continuar probando (para finalizar, ingrese la palabra: No
  print("Fin")
```

Ejemplo 6.3. Haga un adecuado análisis, diseño e implementación en Python de un subprograma con una solución **ITERATIVA** y uno con una solución **RECURSIVA** que reciban un número entero y que devuelvan el **factorial** del número recibido. Además, implemente un

programa principal que se encargue de interactuar con el usuario y que le permita probar el o los subprogramas requeridos con los valores que desee, tantas veces como lo desee.

Análisis y diseño

Naturalmente, para que la programación sea modular, el programa principal y el subprograma tienen que tener tareas claramente diferenciadas. Por otra parte, el subprograma debe ser totalmente funcional por sí mismo en caso de que se desee utilizar por fuera del programa principal solicitado. El subprograma devolverá `nan` si recibe un número entero menor a cero. Sin embargo, el subprograma no tendrá programado un comportamiento en particular para cuando recibe algo diferente a un número entero (algo diferente a un valor de tipo `int`).
La siguiente implementación se obtuvo: partiendo del análisis y diseño que se hizo en el Ejemplo 4.4 y el Ejemplo 5.2; separando y diferenciando adecuadamente la tarea del subprograma con respecto a las del programa principal; y usando la sintaxis de Python.

Implementación en Python del subprograma solicitado

Subprograma solicitado (solución iterativa):

```
# Subprograma solicitado (solución iterativa)
def mi_factorial_iter(k):
  '''Aquí va la documentación y ayuda de la función `mi_factorial_iter`'''
  if k >= 0:
    if k < 2:
      res = 1
    else:
      res = k
      while k > 2:
        k = k - 1
        res = res * k
  else:
    print(" ", "-"*80)
    print("  No se recibió un entero no negativo, entonces se devolverá: `nan`")
    print(" ", "-"*80)
    res = float("nan")
  return res
```

Subprograma solicitado (solución recursiva):

```
# Subprograma solicitado (solución recursiva)
def mi_factorial_recur(k):
  '''Aquí va la documentación y ayuda de la función `mi_factorial_recur`'''
  if k > 2:
    res = k * mi_factorial_recur(k - 1)
  elif k > 1:
    res = 2
  elif k >= 0:
    res = 1
  else:
    print(" ", "-"*80)
    print("  No se recibió un entero no negativo, entonces se devolverá: `nan`")
    print(" ", "-"*80)
    res = float("nan")
  return res
```

Implementación en Python del programa principal

Prueba rápida de los subprogramas (sin un programa principal):

```
# Una prueba rápida de los subprogramas (sin un programa principal)
# Para poder comparar importaré la función `factorial()` del módulo `math`:
from math import factorial
k = 6
print("  mi_factorial_iter(", k, ") = ", sep="")
print(mi_factorial_iter(k))
print("  mi_factorial_recur(", k, ") = ", sep="")
print(mi_factorial_recur(k))
print("  factorial(", k, ") = ", sep="")
print(factorial(k))
```

```
  mi_factorial_iter(6) =
720
  mi_factorial_recur(6) =
720
  factorial(6) =
720
```

```python
# Otra prueba rápida de los subprogramas (sin un programa principal)
# Para poder comparar importaré la función `factorial()` del módulo `math`:
from math import factorial
k = -6
print("  mi_factorial_iter(", k, ") = ", sep="")
print(mi_factorial_iter(k))
print("  mi_factorial_recur(", k, ") = ", sep="")
print(mi_factorial_recur(k))
print("  factorial(", k, ") = ", sep="")
print(factorial(k))
```

```
  mi_factorial_iter(-6) =
  ------------------------------------------------------------------------------
  No se recibió un entero no negativo, entonces se devolverá: `nan`
  ------------------------------------------------------------------------------
nan
  mi_factorial_recur(-6) =
  ------------------------------------------------------------------------------
  No se recibió un entero no negativo, entonces se devolverá: `nan`
  ------------------------------------------------------------------------------
nan
  factorial(-6) =

ValueError: factorial() not defined for negative values
```

Programa principal que permite probar y comparar los subprogramas solicitados:

```
# Programa principal que prueba los subprogramas solicitados
# Para poder comparar importaré la función `factorial()` del módulo `math`:
from math import factorial
print("Este programa prueba las funciones `mi_factorial_iter` y `mi_factorial_recur`")
continuar = ""
while continuar != "No":
  k = int(input("\nPor favor, ingrese un número entero:"))
  print("  mi_factorial_iter(", k, ") = ", sep="")
  print(mi_factorial_iter(k))
  print("  mi_factorial_recur(", k, ") = ", sep="")
  print(mi_factorial_recur(k))
  print("  factorial(", k, ") = ", sep="")
  print(factorial(k))
  continuar = input("\n¿Desea continuar probando (para finalizar, ingrese la palabra: No
print("Fin")
```

Ejemplo 6.4. Haga un adecuado análisis, diseño e implementación en Python de un subprograma que reciba cuatro números reales asociados a los coeficientes de un polinomio de grado 3 o inferior, y que **devuelva un subprograma / una "variable" de tipo `function`** asociado a la función matemática de los reales en los reales dada por el polinomio correspondiente a los coeficientes recibidos (es decir, se **devuelve un subprograma / una "variable" de tipo `function`** que tiene la capacidad de recibir un número real y de devolver el polinomio correspondiente evaluado en el número recibido). Además, implemente un programa principal que se encargue de interactuar con el usuario y que le permita probar el o los subprogramas requeridos con los valores que desee, tantas veces como lo desee.

Análisis

Un polinomio de grado tres usualmente se escribe de la forma:

$$p(x) = a_3x^3 + a_2x^2 + a_1x + a_0$$

Sin embargo, la anterior pueden no ser la mejor manera de hacer los cálculos, computacionalmente hablando.
Note que,

$$a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$$

Por otra parte, si a_3 es igual a cero, entonces el polinomio $p(x)$ no sería de grado tres, sería de grado dos; si resulta que a_2 también es igual que cero entonces el polinomio sería de grado uno: y dentro de los polinomios de grado uno tenemos cuatro posibilidades, dependiendo de si a_1 y a_0 son iguales a cero o no.

El subprograma solicitado debe tener en cuenta todo lo anterior para una adecuada implementación modular y computacionalmente eficiente. Además, se debe tener presente que el subprograma debe devolver una "variable" de tipo `function`, capaz de recibir un número real y de devolver el polinomio evaluado en ese número real. En este caso, el subprograma solicitado es una función de orden superior.

Implementación en Python del subprograma solicitado:

```
# Subprograma que apoya el subprograma solicitado
def devuelve_funcion_polinomio_grado1omenos(a_1, a_0):
  '''Aquí va la documentación y ayuda de la función.
  Esta función devuelve una función'''
  if a_1 != 0:
    if a_0 != 0:
      def p(x): return a_1 * x + a_0
    else:
      def p(x): return a_1 * x
  else:
    if a_0 != 0:
      def p(x): return a_0
    else:
      def p(x): return 0
  return p

# Subprograma que apoya el subprograma solicitado
def devuelve_funcion_polinomio_grado2omenos(a_2, a_1, a_0):
  '''Aquí va la documentación y ayuda de la función.
  Esta función devuelve una función'''
  if a_2 != 0:
    def p(x):
      return (a_2 * x + a_1) * x + a_0
  else:
    p = devuelve_funcion_polinomio_grado1omenos(a_1, a_0)
  return p

# Subprograma solicitado
def devuelve_funcion_polinomio_grado3omenos(a_3, a_2, a_1, a_0):
  '''Aquí va la documentación y ayuda de la función.
  Esta función devuelve una función'''
  if a_3 != 0:
    def p(x):
      return ((a_3 * x + a_2) * x + a_1) * x + a_0
  else:
    p = devuelve_funcion_polinomio_grado2omenos(a_2, a_1, a_0)
  return p
```

Implementación en Python del programa principal

Prueba rápida de los subprogramas (sin un programa principal):

```
# Una prueba rápida de los subprogramas (sin un programa principal)
print("$p_1(x) = x^2 - 2$")
p_1 = devuelve_funcion_polinomio_grado3omenos(0, 1, 0, -2)
print("Para x:")
for i in range(-5,6,1):
  print(f'{i:5d}', end="")
print("\nUsando p_1")
for i in range(-5,6,1):
  print(f'{p_1(i):5d}', end="")
print("\nSin usar p_1")
for i in range(-5,6,1):
  print(f'{i*i-2:5d}', end="")
```

```
$p_1(x) = x^2 - 2$
Para x:
   -5   -4   -3   -2   -1    0    1    2    3    4    5
Usando p_1
   23   14    7    2   -1   -2   -1    2    7   14   23
Sin usar p_1
   23   14    7    2   -1   -2   -1    2    7   14   23
```

```
# Otra prueba rápida de los subprogramas (sin un programa principal)
print("$p_2(x) = x^3 - 2x + 2$")
p_2 = devuelve_funcion_polinomio_grado3omenos(1, 0, -2, 2)
print("Para x:")
for i in range(-5,6,1):
  print(f'{i:5d}', end="")
print("\nUsando p_1")
for i in range(-5,6,1):
  print(f'{p_2(i):5d}', end="")
print("\nSin usar p_2")
for i in range(-5,6,1):
  print(f'{i*i*i - 2*i + 2:5d}', end="")
```

```
$p_2(x) = x^3 - 2x + 2$
Para x:
   -5   -4   -3   -2   -1    0    1    2    3    4    5
```

```
Usando p_1
 -113  -54  -19   -2    3    2    1    6   23   58  117
Sin usar p_2
 -113  -54  -19   -2    3    2    1    6   23   58  117
```

Programa principal que permite probar y comparar el subprograma solicitado:

```
# Programa principal que prueba los subprogramas solicitados
print("Este programa prueba la función `devuelve_funcion_polinomio_grado3omenos`")
continuar1 = ""
while continuar1 != "No":
  a_3 = float(input("\nPor favor, ingrese un número real (a_3):"))
  a_2 = float(input("Por favor, ingrese un número real (a_2):"))
  a_1 = float(input("Por favor, ingrese un número real (a_1):"))
  a_0 = float(input("Por favor, ingrese un número real (a_0):"))
  p = devuelve_funcion_polinomio_grado3omenos(a_3, a_2, a_1, a_0)
  print("Se creo la función computacional `p` asociada al polinomio:",
        "\np(x) = (", a_3, ")x^3 + (", a_2, ")x^2 + (", a_1, ")x + (", a_0, ")", sep="")
  continuar2 = ""
  while continuar2 != "No":
    x = float(input("\nPor favor, ingrese un número real (x):"))
    # Se usa la función computacional creada:
    print("  p(", x, ") =", sep="")
    print(p(x))
    # Para comparar:
    print("  ((a_3*x + a_2)*x + a_1)*x + a_0 =")
    print(((a_3*x + a_2)*x + a_1)*x + a_0)
    print("  a_3*(x**3) + a_2*(x**2) + a_1*x + a_0 =")
    print(a_3*(x**3) + a_2*(x**2) + a_1*x + a_0)
    continuar2 = input("\n¿Desea continuar probando el mismo polinomio (para finalizar,
  continuar1 = input("\n¿Desea continuar probando, pero cambiando el polinomio (para fin
print("Fin")
```

Ejemplo 6.5. Haga un adecuado análisis, diseño e implementación en Python de un subprograma que devuelva de manera aproximada una raíz de una función de los reales en los reales dada (encontrar $x \in \mathbb{R}$ tal que $f(x) \approx 0$ para $f : \mathbb{R} \to \mathbb{R}$), usando el método de Newton-Raphson (teniendo en cuenta todas las consideraciones del método). Además,

- Implemente un primer programa principal que se encargue de interactuar con el usuario y que le permita probar y usar el subprograma solicitado para encontrar de manera

aproximada la raíz cuadrada (positiva o negativa) de un número real positivo (c), con los valores que el usuario desee y tantas veces como lo desee.

- Implemente un segundo programa principal que se encargue de interactuar con el usuario y que le permita probar y usar el subprograma solicitado para encontrar de manera aproximada una raíz real de un polinomio de grado 3 o inferior, con los valores que el usuario desee y tantas veces como lo desee.

Análisis y diseño para el subprograma solicitado

La siguiente implementación del subprograma solicitado se obtuvo: partiendo del análisis y diseño en el Ejemplo 4.8, identificando adecuadamente las tareas específicas del subprograma (para que este sea totalmente funcional y útil por sí mismo sin importar si hay o no un programa principal), y usando la sintaxis de Python.
El subprograma devolverá `nan` en todos los casos en donde la aplicación del método no termine "exitosamente". Sin embargo, el subprograma no tendrá programado un comportamiento en particular para cuando recibe un valor de un tipo distinto al tipo de variable esperado para cada parámetro del subprograma. Los parámetros `f` y `df` esperan recibir "variables" de tipo `function` de máximo un parámetro obligatorio. Por otra parte, dos de los cinco parámetros tendrán valores predeterminados, es decir, serán opcionales al momento de llamar la función: `eps = 1e-12` y `max_iter = 50`. Se usará un subprograma propio para el cálculo de valores absolutos (`mi_val_abs()`), y por el momento, el subprograma informará mediante un mensaje por pantalla el motivo por el cual finalizó la aplicación del método.
Para comparar resultados, se importará la función `newton()` del módulo `optimize` del paquete `scipy`.
Aunque solamente se harán pruebas con los dos programas principales solicitados, el subprograma no debe depender de ellos, y por lo tanto, debe poder encontrar de manera aproximada una raíz de una función $f : \mathbb{R} \to \mathbb{R}$ a la que se le pueda aplicar el método. Es decir, debe funcionar para cualquier subprograma de la forma:

```
def nombre_funcion(x):
  ...
  return y
```

que represente computacionalmente el accionar del objeto matemático f, cumpliendo las condiciones que requiere el método en cuestión.

Implementación en Python del subprograma solicitado

```
# Para el subprograma solicitado utilizaré:
# `mi_val_abs()`

# Subprograma solicitado
def mi_newton_raphson(x0, f, df, eps = 1e-12, max_iter = 50):
  xNew = x0 # Para empezar el siguiente `while` con x0 = x = xNew
  n = 0
  seguir = True
  while seguir == True and n < max_iter:
    x = xNew
    dfx = df(x)
    if mi_val_abs(dfx) < eps:
      msg = "La derivada evaluada en x_n se acerca demasiado a cero\n  "
      msg = msg + "df(x_n) = " + str(dfx)
      xNew = float("nan")
      seguir = False
    else:
      fx = f(x)
      delta = fx / dfx
      xNew = x - delta
      n = n + 1
      if mi_val_abs(fx) < eps:
        msg = "Se encontró un x para el cual |f(x)| < epsilon\n  "
        msg = msg + "f(x_n) = " + str(fx)
        seguir = False
      else:
        if mi_val_abs(delta) < eps:
          msg = "No hubo un cambio superior a epsilon de x_n a x_{n+1}\n  "
          msg = msg + "x_n - x_{n+1} = " + str(delta)
          xNew = float("nan")
          seguir = False
  if seguir == True:
    msg = "Se alcanzó el máximo de iteraciones: " + str(max_iter) + "\n  "
    msg = msg + "x_0 = " + str(x0) + ", x_n = " + str(x) + ", x_{n+1} = " + str(xNew)
    xNew = float("nan")
  print("  "+("-"*80), "Mensaje de `mi_newton_raphson()`:", msg, "-"*80, sep="\n  ")
  return xNew
```

Análisis y diseño para el primer programa principal

En el primer programa principal se solicita utilizar el método de Newton-Raphson para encontrar de manera aproximada un x tal que $x = \sqrt{c}$. Es decir, una aproximación para $x \in \mathbb{R}$ tal que,

$$x = \sqrt{c}$$
$$x^2 = c$$
$$x^2 - c = 0$$

En otras palabras, se debe encontrar un $x \in \mathbb{R}$ tal que $f(x) \approx 0$ para,

$$f(x) = x^2 - c$$

Como el método necesita la derivada de $f(x)$, entonces derivando se tiene que,

$$f'(x) = 2x$$

Como $x^2 - c$ es un polinomio de grado 2 y $2x$ es un polinomio de grado 1, se pueden usar los subprogramas `devuelve_funcion_polinomio_grado2omenos()` y `devuelve_funcion_polinomio_grado1omenos()`.

Implementación en Python del primer programa principal

Primera prueba rápida del subprograma solicitado (sin un programa principal):

```
  # Para las pruebas rápidas y de programas principales utilizaré:
  # `devuelve_funcion_polinomio_grado1omenos()`
  # `devuelve_funcion_polinomio_grado2omenos()`
  # `devuelve_funcion_polinomio_grado3omenos()`

  # Prueba rápida del subprograma (sin un programa principal)
  # Para poder comparar importaré la función `newton()` del módulo `scipy.optimize`
  from scipy.optimize import newton
  c = 2 # Se puede cambiar este valor por cualquier otro
  for x0 in (1, -2, 0): # Se puede probar con otros valores
    print(f"\nTomando $f(x) = p(x) = x^2 - {c}$")
    print(f"y $x_0 = {x0}$")
    p = devuelve_funcion_polinomio_grado2omenos(1, 0, -c)
    dp = devuelve_funcion_polinomio_grado1omenos(2, 0)
    x = mi_newton_raphson(x0, p, dp)
    print("  La función `mi_newton_raphson()` devuelve el valor:")
    print(x)
    print("  La función `newton()` del módulo `scipy.optimize` devuelve el valor:")
    print(newton(p, x0, dp))
    print("  Se sabe que $x^2 - c = 0$ para $x = \pm \sqrt{c}$, entonces |x| =")
    print(c**0.5)
```

```
Tomando $f(x) = p(x) = x^2 - 2$
y $x_0 = 1$
  ----------------------------------------------------------------------------
  Mensaje de `mi_newton_raphson()`:
  Se encontró un x para el cual |f(x)| < epsilon
  f(x_n) = 4.440892098500626e-16
  ----------------------------------------------------------------------------
  La función `mi_newton_raphson()` devuelve el valor:
1.414213562373095
  La función `newton()` del módulo `scipy.optimize` devuelve el valor:
1.4142135623730951
  Se sabe que $x^2 - c = 0$ para $x = \pm \sqrt{c}$, entonces |x| =
1.4142135623730951

Tomando $f(x) = p(x) = x^2 - 2$
y $x_0 = -2$
  ----------------------------------------------------------------------------
  Mensaje de `mi_newton_raphson()`:
```

```
  Se encontró un x para el cual |f(x)| < epsilon
  f(x_n) = 4.440892098500626e-16
  ------------------------------------------------------------------------------
  La función `mi_newton_raphson()` devuelve el valor:
-1.414213562373095
  La función `newton()` del módulo `scipy.optimize` devuelve el valor:
-1.4142135623730951
  Se sabe que $x^2 - c = 0$ para $x = \pm \sqrt{c}$, entonces |x| =
1.4142135623730951

Tomando $f(x) = p(x) = x^2 - 2$
y $x_0 = 0$
  ------------------------------------------------------------------------------
  Mensaje de `mi_newton_raphson()`:
  La derivada evaluada en x_n se acerca demasiado a cero
  df(x_n) = 0
  ------------------------------------------------------------------------------
  La función `mi_newton_raphson()` devuelve el valor:
nan
  La función `newton()` del módulo `scipy.optimize` devuelve el valor:

RuntimeError: Derivative was zero. Failed to converge after 1 iterations, value is 0.0.
```

Primer programa principal que permite probar y comparar el subprograma solicitado:

```
# Para las pruebas rápidas y de programas principales utilizaré:
# `devuelve_funcion_polinomio_grado1omenos()`
# `devuelve_funcion_polinomio_grado2omenos()`
# `devuelve_funcion_polinomio_grado3omenos()`

# Programa principal que prueba el subprograma solicitado
# Para poder comparar importaré la función `newton()` del módulo `scipy.optimize`
from scipy.optimize import newton
print("Este programa hace una primera prueba de la función `mi_newton_raphson()`")
print("La idea aquí será encontrar de manera aproximada un x tal que f(x) = x^2 - c = 0,
continuar1 = "Si"
while continuar1 == "Si":
  c = float(input("\nPor favor, ingrese un número real positivo (c):"))
  while c <= 0:
    c = float(input("Por favor, ingrese un número real POSITIVO (c):"))
  p = devuelve_funcion_polinomio_grado2omenos(1, 0, -c)
  dp = devuelve_funcion_polinomio_grado1omenos(2, 0)
  continuar2 = "Si"
  while continuar2 == "Si":
    x0 = float(input("\nPor favor, ingrese un número real (x0):"))
    cambiar = input("¿Desea cambiar el valor predeterminado de epsilon (para cambiarlo,
    if cambiar == "Si":
      eps = float(input("Por favor, ingrese un número real positivo menor que 0.01 (epsi
      while eps <= 0 or eps >= 0.01:
        eps = float(input("Por favor, ingrese un número real POSITIVO MENOR QUE 0.01 (ep
      x = mi_newton_raphson(x0, p, dp, eps)
    else:
      x = mi_newton_raphson(x0, p, dp)
    print("  La función `mi_newton_raphson()` devuelve el valor:")
    print(x)
    print("  La función `newton()` del módulo `scipy.optimize` devuelve el valor:")
    print(newton(p, x0, dp))
    print("  `c**0.5` es igual a:")
    print(c**0.5)
    continuar2 = input("¿Desea probar con otro valor inicial u otro epsilon (para probar
  continuar1 = input("\n¿Desea probar con otro valor c (para probar con otro valor, ingr
print("Fin")
```

Análisis y diseño para el segundo programa principal

En el segundo programa principal, se debe encontrar un $x \in \mathbb{R}$ tal que $f(x) \approx 0$ para,

$$\begin{aligned} f(x) &= a_3x^3 + a_2x^2 + a_1x + a_0 \\ &= ((a_3x + a_2)x + a_1)x + a_0 \end{aligned}$$

Como el método necesita la derivada de $f(x)$, entonces derivando se tiene que,

$$\begin{aligned} f'(x) &= 3a_3x^2 + 2a_2x + a_1 \\ &= (3a_3x + 2a_2)x + a_1 \end{aligned}$$

Para la función objetivo y su derivada se pueden usar los subprogramas `devuelve_funcion_polinomio_grado3omenos()` y `devuelve_funcion_polinomio_grado2omenos()`.

Implementación en Python del segundo programa principal

Segunda prueba rápida del subprograma solicitado (sin un programa principal):

```
  # Para las pruebas rápidas y de programas principales utilizaré:
  # `devuelve_funcion_polinomio_grado1omenos()`
  # `devuelve_funcion_polinomio_grado2omenos()`
  # `devuelve_funcion_polinomio_grado3omenos()`

  # Prueba rápida del subprograma (sin un programa principal)
  # Para poder comparar importaré la función `newton()` del módulo `scipy.optimize`
  from scipy.optimize import newton
  a3 = 1 # Se puede cambiar este valor por cualquier otro
  a2 = 0 # Se puede cambiar este valor por cualquier otro
  a1 = -2 # Se puede cambiar este valor por cualquier otro
  a0 = 2 # Se puede cambiar este valor por cualquier otro
  for x0 in (-2, 1): # Se puede probar con otros valores
    print(f"\nTomando $f(x) = p(x) = ({a3})x^3 + ({a2})x^2 + ({a1})x + ({a0})$")
    print(f"y $x_0 = {x0}$")
    p = devuelve_funcion_polinomio_grado3omenos(a3, a2, a1, a0)
    dp = devuelve_funcion_polinomio_grado2omenos(3*a3, 2*a2, a1)
    x = mi_newton_raphson(x0, p, dp)
    print("  La función `mi_newton_raphson()` devuelve el valor:")
    print(x)
    print("  La función `newton()` del módulo `scipy.optimize` devuelve el valor:")
    print(newton(p, x0, dp))
```

```
Tomando $f(x) = p(x) = (1)x^3 + (0)x^2 + (-2)x + (2)$
y $x_0 = -2$
  ------------------------------------------------------------------------------
  Mensaje de `mi_newton_raphson()`:
  Se encontró un x para el cual |f(x)| < epsilon
  f(x_n) = -5.053735208093713e-13
  ------------------------------------------------------------------------------
  La función `mi_newton_raphson()` devuelve el valor:
-1.7692923542386314
  La función `newton()` del módulo `scipy.optimize` devuelve el valor:
-1.7692923542386314

Tomando $f(x) = p(x) = (1)x^3 + (0)x^2 + (-2)x + (2)$
y $x_0 = 1$
  ------------------------------------------------------------------------------
  Mensaje de `mi_newton_raphson()`:
  Se alcanzó el máximo de iteraciones: 50
```

```
  x_0 = 1, x_n = 0.0, x_{n+1} = 1.0
  ------------------------------------------------------------------------------
  La función `mi_newton_raphson()` devuelve el valor:
nan
  La función `newton()` del módulo `scipy.optimize` devuelve el valor:

RuntimeError: Failed to converge after 50 iterations, value is 1.0.
```

Segundo programa principal que permite probar y comparar el subprograma solicitado:

```
# Para las pruebas rápidas y de programas principales utilizaré:
# `devuelve_funcion_polinomio_grado1omenos()`
# `devuelve_funcion_polinomio_grado2omenos()`
# `devuelve_funcion_polinomio_grado3omenos()`

# Programa principal que prueba el subprograma solicitado
# Para poder comparar importaré la función `newton()` del módulo `scipy.optimize`
from scipy.optimize import newton
print("Este programa hace una primera prueba de la función `mi_newton_raphson()`")
print("La idea aquí será encontrar una raíz real de un polinomio con coeficientes reales
print("$f(x) = p(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$")
continuar1 = "Si"
while continuar1 == "Si":
  a3 = float(input("\nPor favor, ingrese un número real (a3):"))
  a2 = float(input("Por favor, ingrese un número real (a2):"))
  a1 = float(input("Por favor, ingrese un número real (a1):"))
  a0 = float(input("Por favor, ingrese un número real (a0):"))
  p = devuelve_funcion_polinomio_grado3omenos(a3, a2, a1, a0)
  dp = devuelve_funcion_polinomio_grado2omenos(3*a3, 2*a2, a1)
  continuar2 = "Si"
  while continuar2 == "Si":
    x0 = float(input("\nPor favor, ingrese un número real (x0):"))
    cambiar = input("¿Desea cambiar el valor predeterminado de epsilon (para cambiarlo,
    if cambiar == "Si":
      eps = float(input("Por favor, ingrese un número real positivo menor que 0.01 (epsi
      while eps <= 0 or eps >= 0.01:
        eps = float(input("Por favor, ingrese un número real POSITIVO MENOR QUE 0.01 (ep
      x = mi_newton_raphson(x0, p, dp, eps)
    else:
      x = mi_newton_raphson(x0, p, dp)
    print("  La función `mi_newton_raphson()` devuelve el valor:")
    print(x)
    print("  La función `newton()` del módulo `scipy.optimize` devuelve el valor:")
    print(newton(p, x0, dp))
    continuar2 = input("¿Desea probar con otro valor inicial u otro epsilon (para probar
  continuar1 = input("\n¿Desea probar con otro polinomio (para probar con otro polinomio
print("Fin")
```

i ¿Por qué los estadísticos deben conocer acerca del método de Newton-Raphson y otros métodos de optimización?

En la estimación de parámetros por máxima verosimilitud, exceptuando algunos casos, las ecuaciones de verosimilitud o log-verosimilitud,

$$\frac{\partial}{\partial\theta}\mathrm{L}(\theta; x_1, \dots, x_n) = 0 \quad \text{o} \quad \frac{\partial}{\partial\theta}\log \mathrm{L}(\theta; x_1, \dots, x_n) = 0,$$

no pueden solucionarse de manera análitica. Por tanto, es necesario usar métodos numéricos o procedimientos iterativos para poder solucionar el problema de optimización asociado a dichas ecuaciones,

$$\arg\sup_{\theta\in\Theta} \mathrm{L}\left(\theta; x_1, \dots, x_n\right) \quad \text{o} \quad \arg\sup_{\theta\in\Theta} \log \mathrm{L}\left(\theta; x_1, \dots, x_n\right).$$

Existen variados métodos numéricos para resolver problemas de optimización. Buena parte de ellos toman un valor inicial para θ, notado $\hat{\theta}_0$, y a partir de él buscan obtener una secuencia convergente $\left\{\hat{\theta}_k\right\}_{k=0,1,\dots}$. Los algoritmos más comunmente utilizados, los **Hill climbing algorithms**, se basan en una ecuación de actualización de este tipo,

$$\hat{\theta}_{k+1} = \hat{\theta}_k + \lambda_k \, \mathrm{d}_k\left(\hat{\theta}_k\right)$$

donde el vector $\mathrm{d}_k\left(\hat{\theta}_k\right)$ indica la "dirección del k-ésimo paso" y el escalar λ_n representa la "longitud o tamaño de ese k-ésimo paso".
En el caso de los **Gradient ascent algorithms**, la dirección de cada paso es la del gradiente de la función objetivo, $\mathrm{d}_k\left(\hat{\theta}_k\right) = \Delta\left(\hat{\theta}_k\right)$. El gradiente de la función de log-verosimilitud con respecto al vector de parámetros suele denominarse **score**, $\Delta\left(\theta\right) = \frac{\partial}{\partial\theta}\log \mathrm{L}$.
En el método de **Newton–Raphson**, $\lambda_k = 1$ y $\mathrm{d}_k\left(\hat{\theta}\right) = -\mathrm{H}^{-1}\left(\hat{\theta}_k\right)\Delta\left(\hat{\theta}_k\right)$, donde $\Delta(\theta)$ es el score y $\mathrm{H}^{-1}(\theta)$ es el inverso de la matrix Hessiana de la función de log-verosimilitud $\left(\mathrm{H}(\theta) = \frac{\partial^2}{\partial\theta\partial\theta^T}\log \mathrm{L}\right)$, ambos evaluados en $\hat{\theta}_k$.
El método denominado **Fisher's Scoring** es casi identico al método de Newton-Raphson, pero en vez de utilizar el negativo del inverso de la matrix Hessiana de la función de log-verosimilitud $(-\mathrm{H}^{-1}(\theta))$, utiliza el inverso de la matrix de información de Fisher $(\mathcal{I}^{-1}\left(\theta\right))$.
La matrix de **información de Fisher** es la varianza del score,

$$\mathcal{I}\left(\theta\right) = Var\left[\Delta\left(\theta\right)\right] = E\left[\Delta\left(\theta\right)\Delta\left(\theta\right)^T\right]$$

y bajo ciertas condiciones (condiciones de regularidad), también es igual al inverso aditivo del valor esperado de la matrix Hessiana de la función de log-verosimilitud,

$$\mathcal{I}\left(\theta\right) = Var\left[\Delta\left(\theta\right)\right] = E\left[\Delta\left(\theta\right)\Delta\left(\theta\right)^T\right] = -E\left[\mathrm{H}(\theta)\right].$$

Dado que el cálculo de la matrix Hessiana o el de la matrix de información (más el cálculo de la respectiva inversa) son computacionalmente costosos, otras alternativas han sido propuestas. Una de ellas es, por ejemplo, el algoritmo de **Broyden–Fletcher–Goldfarb–Shanno (BFGS)**.
Como si lo anterior no fuera más que suficiente:

> "**Machine learning** also has intimate ties to **optimization**: many learning problems are formulated as **minimization of some loss function** on a training set of examples. Loss functions express the discrepancy between the predictions of the model being trained and the actual problem instances (for example, in classification, one wants to assign a label to instances, and models are trained to correctly predict the pre-assigned labels of a set of examples)."
>
> "**Modern neural networks** model complex relationships between inputs and outputs and find patterns in data. They can learn continuous functions and even digital logical operations. Neural networks can be viewed as a type of mathematical **optimization** — they perform **gradient descent** on a multi-dimensional topology that was created by training the network. The most common training technique is the backpropagation algorithm."
>
> "In a recurrent neural network the signal will propagate through a layer more than once; thus, an RNN is an example of **deep learning**. RNNs can be trained by **gradient descent**, however long-term gradients which are back-propagated can"vanish" (that is, they can tend to zero) or "explode" (that is, they can tend to infinity), known as the vanishing gradient problem."
>
> "Many problems in **AI** can be solved theoretically by intelligently searching through many possible solutions… For many problems, it is possible to begin the search with some form of a guess and then refine the guess incrementally until no more refinements can be made. These algorithms can be visualized as blind **hill climbing**: we begin the search at a random point on the landscape, and then, by jumps or steps, we keep moving our guess uphill, until we reach the top."

6.6 Ejercicios

Para cada uno de los siguientes puntos:

- Implemente un programa principal que se encargue de interactuar con el usuario y que le permita probar el o los subprogramas requeridos con los valores que desee, tantas veces como lo desee.

- Las soluciones deben ser iterativas, a menos que explicitamente se pida un solución recursiva. En aquellos casos en los que la solución debe ser iterativa, su subprograma puede hacer llamados a sí mismo, si usted así lo desea o lo requiere por alguna razón, siempre y cuando esos llamados no estén haciendo el trabajo que deberían estar haciendo estructuras iterativas.
- El parámetro asociado a un número real positivo cercano a cero (ε) deberá tener como valor predeterminado: $1 \times 10^{-12} =$ `1e-12`.
- La idea es que siempre hagan sus propios subprogramas/funciones para luego comparar resultados con los que se obtienen al usar operadores o funciones ya existentes y disponibles en Python. Las funciones básicas y los operadores básicos que se referencian en Elementos Básicos son los únicos elementos ya existentes y disponibles en Python para los que no es necesario hacer una implementación propia equivalente. Como en varios ejercicios se espera una implementación propia para algún tipo de exponenciación, el operador `**` o la función `pow()` se pueden usar **pero obviamente sólo para comparar resultados**. No olviden lo que se encuentra en la Introducción de la Presentación del curso y que les mencioné el primer día de clase:

  > Esta NO es una asignatura para aprender a usar los paquetes, módulos o "comandos" de una herramienta informática en particular. Es un curso para entender **el cómo y el porqué funcionan los elementos informáticos básicos** que son usados por estadísticos / "científicos de datos" en su quehacer. En el curso, **los estudiantes deben hacer sus propias implementaciones para la solución de diferentes problemas, que luego podrán comparar con las implementaciones existentes en el software especializado que deseen**.

Haga un adecuado **ANÁLISIS**, **DISEÑO** e **IMPLEMENTACIÓN EN PYTHON** de:

1. Un subprograma que reciba un número real (c) y que devuelva la **función signo** ($\text{sign}(c)$) evaluada en el número recibido.
2. Tres subprogramas: uno que reciba un valor asociado a grados Centigrados y que devuelva su equivalente en **grados Fahrenheit**, otro que reciba un valor asociado a grados Fahrenheit y que devuelva su equivalente en **grados Centigrados**, por último, un subprograma que reciba un número real asociado a los grados y una cadena asociada a la escala de una temperatura válida (por ejemplo, "C" para Centigrados y "F" para Fahrenheit), y que utilice los subprogramas anteriores para devolver la temperatura equivalente en la otra escala con respecto a la recibida (si recibe grados Centigrados devolverá grados Fahrenheit y viceversa).
3. Un subprograma que reciba un número de día, un número de mes y un número de año, y que devuelva el valor booleano que corresponda, dependiendo de si la fecha asociada a esos tres datos es una **fecha válida** (`True`) **o no** (`False`) del calendario gregoriano.

4. Un subprograma que reciba un número de día, un número de mes y un número de año de una fecha del calendario gregoriano, y que devuelva la cadena que corresponda a la **fecha del día siguiente** usando el formato: "año/mes/día".

5. Un subprograma que reciba un número entero (r), con valor predeterminado igual a cero, y que pida al usuario por pantalla, tantas veces como sea necesario, un número entero (k) mayor o igual al número entero recibido por el subprograma ($k \geq r$). Es decir, un subprograma que se encargue de la tarea de seguir pidiendo al usuario que **ingrese un número mayor o igual a un número entero dado** hasta que el usuario efectivamente lo haga.

6. Un subprograma que reciba dos números reales ($a < b$), y que pida al usuario por pantalla, tantas veces como sea necesario, un número real (c) que esté entre los dos números reales recibidos por el subprograma ($a < c < b$). Es decir, un subprograma que se encargue de la tarea de seguir pidiendo al usuario que **ingrese un número real entre dos números reales dados** hasta que el usuario efectivamente lo haga.

7. Un subprograma con una solución **ITERATIVA** y uno con una solución **RECURSIVA** que reciban dos números enteros y que devuelvan el **máximo común divisor** de los números recibidos. Adicionalmente, un subprograma que reciba dos números enteros y que utilice uno de los subalgoritmos anteriores (por ejemplo el iterativo) para devolver el **mínimo común múltiplo** de los números recibidos.

8. Un subprograma con una solución **ITERATIVA** y uno con una solución **RECURSIVA** que reciban un número entero (k) y que devuelvan el término k-ésimo de la sucesión de los **números de Fibonacci** https://youtu.be/B6ztvqvZTsk.

9. Un subprograma, con una solución **ITERATIVA**, que reciba un número real positivo cercano a cero (ε), y que devuelva una aproximación para el **número áureo** https://youtu.be/aopHcOm7a-w (φ) a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

   - $$\frac{F_{n+1}}{F_n} \xrightarrow[n\to\infty]{} \varphi$$
   - La solución debe ser computacionalmente mejor, es decir, debe tener un menor número total de operaciones, que simplemente hacer `Fibonacci(n+1) / Fibonacci(n)` para una función `Fibonacci(k)` que devuelva el término k-ésimo de la sucesión de los números de Fibonacci.

10. Un subprograma que reciba un número real positivo cercano a cero (ε) y que devuelva una aproximación para el **número** π a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

    - $$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

- $$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} c^{2k+1} \xrightarrow[n\to\infty]{} \arctan(c)$$
- La solución debe ser computacionalmente mejor, es decir, debe tener un número total de operaciones mucho menor, que simplemente hacer `4 * (4 * arctan(0.2) - arctan(1/239))` para una función `arctan()` que devuelva el arcotangente de un número real.

11. Un subprograma que reciba un número real (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación del **seno** de c a partir del valor dado para ε. Cuando se requiera puede poner al computador a calcular, una única vez por ejecución del subprograma, una aproximación del número π o tomar $\pi \approx 3.1415926535897932$. El subprograma solicitado debe tener en cuenta que:

    - Si $c < 0$, entonces
      $$\sin(c) = -\sin(-c),$$
      lo que reduce el problema a calcular el seno de un valor mayor que cero (0).
    - Si $c \geq 2\pi$, entonces
      $$\sin(c) = \sin(c - k\,2\pi),$$
      lo que reduce el problema a calcular el seno de un valor entre 0 y 2π.
    - Si $\pi \leq c < 2\pi$, entonces
      $$\sin(c) = -\sin(c - \pi),$$
      lo que reduce el problema a calcular el seno de un valor entre 0 y π.
    - Si $\frac{\pi}{2} \leq c < \pi$, entonces
      $$\sin(c) = \sin(\pi - c),$$
      lo que reduce el problema a calcular el seno de un valor entre 0 y $\frac{\pi}{2}$.
    - Si $\frac{\pi}{4} \leq c < \frac{\pi}{2}$, entonces
      $$\sin(c) = \cos\left(\frac{\pi}{2} - c\right),$$
      lo que reduce el problema a calcular el coseno de un valor entre 0 y $\frac{\pi}{4}$.
    - $$\sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} c^{2k+1} \xrightarrow[n\to\infty]{} \sin(c),$$
      que se debe utilizar unicamente para calcular el seno de un valor entre 0 y $\frac{\pi}{4}$.

12. Un subprograma que reciba un número real (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación del **coseno** de c a partir del valor dado para ε. Cuando se requiera puede poner al computador a calcular, una única vez por ejecución del subprograma, una aproximación del número π o tomar $\pi \approx 3.1415926535897932$. El subprograma solicitado debe tener en cuenta que:

- Si $c < 0$, entonces
$$\cos(c) = \cos(-c),$$
lo que reduce el problema a calcular el coseno de un valor mayor que cero (0).
- Si $c \geq 2\pi$, entonces
$$\cos(c) = \sin(c - k\,2\pi),$$
lo que reduce el problema a calcular el coseno de un valor entre 0 y 2π.
- Si $\pi \leq c < 2\pi$, entonces
$$\cos(c) = -\cos(c - \pi),$$
lo que reduce el problema a calcular el coseno de un valor entre 0 y π.
- Si $\frac{\pi}{2} \leq c < \pi$, entonces
$$\cos(c) = -\cos(\pi - c),$$
lo que reduce el problema a calcular el coseno de un valor entre 0 y $\frac{\pi}{2}$.
- Si $\frac{\pi}{4} \leq c < \frac{\pi}{2}$, entonces
$$\cos(c) = \sin\left(\frac{\pi}{2} - c\right),$$
lo que reduce el problema a calcular el seno de un valor entre 0 y $\frac{\pi}{4}$.
-
$$\sum_{k=0}^{n} \frac{(-1)^k}{(2k)!} c^{2k} \xrightarrow[n\to\infty]{} \cos(c),$$
que se debe utilizar únicamente para calcular el coseno de un valor entre 0 y $\frac{\pi}{4}$.

13. Un subprograma que reciba dos números enteros no negativos (k y $r \leq k$), y que devuelva la cantidad de reordenamientos posibles de r elementos que no se repiten tomados de un conjunto de k elementos (número de **permutaciones** o variaciones sin repetición, por ejemplo ver: Combinatoria - Wikipedia). El subprograma solicitado debe tener en cuenta que:

    - El número de permutaciones sin elementos repetidos es
    $$P(k, r) = kPr = \frac{k!}{(k - r)!}$$
    - La solución debe computacionalmente mejor, es decir, debe tener un número total de operaciones menor, que simplemente hacer `Factorial(k) / Factorial(k-r)` para una función `Factorial()` que devuelva el factorial de un número entero no negativo.

14. Un subprograma que reciba dos números enteros no negativos (k y $r \leq k$), y que devuelva la cantidad de subconjuntos posibles de r elementos tomados de un conjunto de k elementos (número de **combinaciones** sin repetición, por ejemplo ver: Combinatoria - Wikipedia). El subprograma solicitado debe tener en cuenta que:

- El número de combinaciones (coeficiente binomial) es

$$C(k,r) = kCr = \binom{k}{r} = \frac{k!}{r!(k-r)!}$$

- La solución debe ser computacionalmente mejor, es decir, debe tener un número total de operaciones menor, que simplemente hacer `Factorial(k) / ( Factorial(r) * Factorial(k-r) )` o `Permutacion(k,r) / Factorial(r)` para una función `Factorial()` que devuelva el factorial de un número entero no negativo y una función `Permutacion(,)` que devuelva el número de permutaciones.

15. Un subprograma con una solución **ITERATIVA** y uno con una solución **RECURSIVA** que reciban un número real (c) y un número entero (k), y que devuelvan la **potencia con exponente entero** c^k.

16. Un subprograma que reciba un número real (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación de la **raíz cuadrada en los complejos** de c a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

    - Si $c < 0$, entonces
    $$\sqrt{c} = 0 + \sqrt{-c}\,i \in \mathbb{C},$$
    lo que reduce el problema a calcular la raíz cuadrada de un valor mayor que cero.
    - Si
    $$\begin{aligned} a_n &= \frac{1}{2}\left(a_{n-1} + \frac{c}{a_{n-1}}\right) \\ &= \left(\frac{1}{2}\right) a_{n-1} + \frac{(c/2)}{a_{n-1}} \end{aligned}$$
    entonces,
    $$a_n \xrightarrow[n\to\infty]{} \sqrt{c}$$
    - Como punto de partida o valor inicial se puede utilizar:
    $$\begin{aligned} a_1 &= \frac{1+c}{2} \\ &= \left(\frac{1}{2}\right) + (c/2) \end{aligned}$$

    En qué aspectos mejora o empeora el subprograma cuando adicionalmente se tiene en cuenta que:

    - Si $0 < c < 1$, entonces
    $$\sqrt{c} = \frac{1}{\sqrt{1/c}},$$
    lo que reduce el problema a calcular la raíz cuadrada de un valor mayor que uno.

- Si $c > 1$, entonces
$$\sqrt{c} = \sqrt{(4^k)\,(d)} = 2^k\sqrt{d},$$
en donde k es un entero no negativo y d es un número real entre uno y cuatro, lo que reduce el problema a calcular la raíz cuadrada de un valor entre uno y cuatro.

17. Tres subprogramas que reciban un número real (c) y un número real positivo cercano a cero (ε) y que devuelvan, respectivamente, una aproximación del **arco tangente**, del **arco seno** y del **arco coseno** de c a partir del valor dado para ε. Cuando se requiera puede poner al computador a calcular, una única vez por ejecución del subprograma, una aproximación del número $\sqrt{3}$ o tomar $\sqrt{3} \approx 1.7320508075688773$ y una aproximación del número π o tomar $\pi \approx 3.1415926535897932$. El subprograma solicitado debe tener en cuenta que:

    - Si $c < 0$, entonces
    $$\arctan(c) = -\arctan(-c),$$
    lo que reduce el problema a calcular el arco tangente de un valor mayor que cero.
    - Si $c > 1$, entonces
    $$\arctan(c) = \frac{\pi}{2} - \arctan\left(\frac{1}{c}\right),$$
    lo que reduce el problema a calcular el arco tangente de un valor menor o igual que uno.
    - Si $c > 2 - \sqrt{3}$, entonces
    $$\arctan(c) = \frac{\pi}{6} + \arctan\left(\frac{\sqrt{3}\,c - 1}{\sqrt{3} + c}\right),$$
    lo que reduce el problema a calcular el arco tangente de un valor menor o igual que $2 - \sqrt{3}$.
    -
    $$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} c^{2k+1} \xrightarrow[n\to\infty]{} \arctan(c),$$
    que se debe utilizar unicamente para calcular el arco tangente de un valor entre 0 y $2 - \sqrt{3}$.
    -
    $$\arcsin(c) = \arctan\left(\frac{c}{\sqrt{1-c^2}}\right)$$
    (utilice su subprograma que calcula la raíz cuadrada de un número)
    -
    $$\arccos(c) = \frac{\pi}{2} - \arcsin(c)$$

18. Un subprograma que reciba un número real (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación de la **raíz cúbica en los reales** de c a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

- Si $c < 0$, entonces
$$\sqrt[3]{c} = -\sqrt[3]{-c} \in \mathbb{R},$$
lo que reduce el problema a calcular la raíz cúbica de un valor mayor que cero.
- Si
$$\begin{aligned} a_n &= \frac{1}{3}\left(2a_{n-1} + \frac{c}{a_{n-1}^2}\right) \\ &= \left(\frac{2}{3}\right) a_{n-1} + \frac{(c/3)}{a_{n-1}^2} \end{aligned}$$
entonces,
$$a_n \xrightarrow[n\to\infty]{} \sqrt[3]{c}$$
- Como punto de partida o valor inicial se puede utilizar:
$$\begin{aligned} a_1 &= \frac{2+c}{3} \\ &= \left(\frac{2}{3}\right) + (c/3) \end{aligned}$$

19. Un subprograma que reciba un número entero diferente de cero ($k \neq 0$), un número real (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación de una **raíz k-ésima en los reales** de c a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

- Si $k < 0$, entonces
$$\sqrt[k]{c} = \sqrt[(-k)]{c^{-1}} = \sqrt[(-k)]{\frac{1}{c}}$$
lo que reduce el problema a calcular una raíz entera positiva de un valor real.
- Para $k = 1$ la solución es trivial, para $k = 2$ utilice su subprograma que calcula la raíz cuadrada de un número.
- Si k es impar mayor que uno y $c < 0$, entonces
$$\sqrt[k]{c} = -\sqrt[k]{-c} \in \mathbb{R},$$
lo que reduce el problema a calcular una raíz entera positiva de un valor mayor que cero (0).
- Si k es par mayor que dos y $c < 0$, entonces
$$\sqrt[k]{c} = \sqrt[k]{-c} \in \mathbb{R},$$
lo que reduce el problema a calcular una raíz entera positiva de un valor mayor que cero (0).

- Si
$$a_n = \frac{1}{k}\left((k-1)a_{n-1} + \frac{c}{a_{n-1}^{(k-1)}}\right)$$
$$= \left(\frac{k-1}{k}\right)a_{n-1} + \frac{(c/k)}{a_{n-1}^{k-1}}$$
entonces,
$$a_n \xrightarrow[n\to\infty]{} \sqrt[k]{c}$$
- Como punto de partida o valor inicial se puede utilizar:
$$a_1 = \frac{(k-1)+c}{k}$$
$$= \left(\frac{k-1}{k}\right) + (c/k)$$

20. Un subprograma que reciba un número real (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación de la **función exponencial** evaluada en c, a partir del valor dado para ε. Cuando se requiera puede poner al computador a calcular, una única vez por ejecución del subprograma, una aproximación del número e o tomar e $\approx 2.7182818284590452$. El subprograma solicitado debe tener en cuenta que:
    - Si $c < 0$, entonces
    $$\exp(c) = \frac{1}{\exp(-c)}$$
    lo que reduce el problema a calcular la función exponencial de un valor mayor que cero.
    - Si $c > 1$, entonces,
    $$\exp(c) = \exp(k+d) = \exp(k)\exp(d)$$
    en donde k es un entero positivo y d es un número real entre cero y uno, lo que reduce el problema a calcular la función exponencial de un valor entero positivo y un valor entre cero y uno.
    - Si c es un entero positivo, entonces,
    $$\exp(c) = \mathrm{e}^c = \underbrace{\mathrm{e}\ldots\mathrm{e}}_{c \text{ veces}}$$
    (utilice su subprograma que calcula la potencia con exponente entero de un número real)
    -
    $$\sum_{k=0}^{n} \frac{c^k}{k!} \xrightarrow[n\to\infty]{} \exp(c)$$
    que se debe utilizar unicamente para calcular la función exponencial de un valor entre cero y uno.

21. Un subprograma que reciba un número real positivo (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación del **logaritmo natural** de c a partir del valor dado para ε. El subprograma solicitado debe usar el subprograma del punto anterior (función exponencial) y el subprograma que implementa el método de Newton-Raphson (Ejemplo 6.5). El subprograma solicitado adicionalmente debe recibir un valor inicial y un máximo número de iteraciones para que estos le sean entregados al subprograma que implementa el método de Newton-Raphson (un usuario del subprograma solicitado debe tener la posibilidad de dar el valor inicial y el máximo número de iteraciones que desee). ¿Cuáles deberían ser el valor inicial y el máximo número de iteraciones sugeridos o propuestos por quien programa, pensando en los usuarios del subprograma solicitado que no quieran o no sepan qué valores dar para esos dos parámetros?

22. Un subprograma que reciba un número real positivo (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación del **logaritmo natural** de c a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

$$2\sum_{k=0}^{n}\frac{1}{2k+1}\left(\frac{c-1}{c+1}\right)^{2k+1}\xrightarrow[n\to\infty]{}\ln(c)$$

    En qué aspectos mejora o empeora el subprograma cuando adicionalmente se tiene en cuenta que:

    - Si $0 < c < 1$, entonces
    $$\ln(c) = -\ln\left(\frac{1}{c}\right),$$
    lo que reduce el problema a calcular el logaritmo natural de un valor mayor que uno.
    - Si $c > 1$, entonces
    $$\ln(c) = \ln\left((d)\left(2^k\right)\right) = \ln(d) + k\ln(2),$$
    en donde k un entero no negativo y d es un número real entre uno y dos, lo que reduce el problema a calcular el logaritmo natural de un valor entre uno y dos.

    Cuando se requiera puede poner al computador a calcular, una única vez por ejecución del subprograma, una aproximación del número $\ln(2)$ o tomar $\ln(2) \approx 0.6931471805599453$.

23. Un subprograma que reciba un número real positivo (c) y un número real positivo cercano a cero (ε), y que devuelva una aproximación del **logaritmo natural** de c a partir del valor dado para ε. El subprograma solicitado debe usar un subprograma adicional que:

    - Reciba dos números reales positivos, un número real positivo cercano a cero (ε) y un entero m con valor predeterminado igual a 64.
    - Devuelva una aproximación de la media aritmético-geométrica (por ejemplo, consultar Logarithm - Wikipedia. Arithmetic–geometric mean approximation)

Cuando se requiera puede poner al computador a calcular, una única vez por ejecución del subprograma, una aproximación del número $\ln(2)$ o tomar $\ln(2) \approx 0.6931471805599453$ y una aproximación del número π o tomar $\pi \approx 3.1415926535897932$.

24. Un subprograma que reciba un número real no negativo (c), un número real (d) y un número real positivo cercano a cero (ε), y que devuelva una aproximación de la **potencia con exponente real** c^d, a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

$$c^d = \exp\left((d)\ln(c)\right)$$

(utilice alguno de sus subprogramas que calculan el logaritmo natural y la función exponencial de un número)

25. Un subprograma que reciba un número real positivo (c), un número real positivo diferente de uno ($d \neq 1$) y un número real positivo cercano a cero (ε), y que devuelva una aproximación del **logaritmo con base positiva diferente de uno** de c, a partir del valor dado para ε. El subprograma solicitado debe tener en cuenta que:

$$\log_d c = \frac{\ln(c)}{\ln(d)}$$

(utilice alguno de sus subprogramas que calculan el logaritmo natural de un número)

26. Un subprograma que:

    - Reciba dos números reales ($a < b$), un número real positivo cercano a cero (ε) y la implementación / el subprograma / la "variable" de tipo `function` correspondiente a una función matemática definida sobre el intervalo $[a, b]$.
    - Devuelva una aproximación de una raíz entre a y b de la función recibida (encontrar $x \in [a, b]$ tal que $f(x) \approx 0$ para $f : [a, b] \to \mathbb{R}$), a partir del valor dado para ε, usando el **método de bisección**, teniendo en cuenta todas las consideraciones del método.

27. Un subprograma que:

    - Reciba dos números reales ($a < b$), un número entero positivo ($k > 0$) (número de subintervalos) y la implementación / el subprograma / la "variable" de tipo `function` correspondiente a una función matemática definida sobre el intervalo $[a, b]$ ($f : [a, b] \to \mathbb{R}$).
    - Devuelva una aproximación de la integral definida entre a y b de la función recibida (encontrar $A \approx \int_a^b f(x)\,\mathrm{d}x$), a partir del valor dado para ε, usando la **regla del trapecio compuesta**, teniendo en cuenta todas las consideraciones del método (por ejemplo consultar Regla del trapecio - Wikipedia. Regla del trapecio compuesta).

28. Un subprograma que:

- Reciba dos números reales ($a < b$), un número entero par positivo ($k > 0$) (número de subintervalos) y la implementación / el subprograma / la "variable" de tipo `function` correspondiente a una función matemática definida sobre el intervalo $[a, b]$ ($f : [a, b] \to \mathbb{R}$).
- Devuelva una aproximación de la integral definida entre a y b de la función recibida (encontrar $A \approx \int_a^b f(x)\,dx$), a partir del valor dado para ε, usando la **regla o método de Simpson 1/3 compuesto**, teniendo en cuenta todas las consideraciones del método (por ejemplo consultar Regla de Simpson - Wikipedia. Regla de Simpson 1/3 compuesta).

29. Un subprograma que:

    - Reciba un número real positivo cercano a cero (ε_0) y la implementación / el subprograma / la "variable" de tipo `function` correspondiente a una función matemática de los reales en los reales ($f : \mathbb{R} \to \mathbb{R}$), derivable en todo su dominio.
    - **Devuelva un subprograma / una "variable" de tipo `function`** que:
        - Reciba un número real (x) y un número real positivo cercano a cero (ε), cuyo valor predeterminado debe ser ε_0.
        - Devuelva una **aproximación de la derivada (derivada numérica)** de la función f en x, a partir del valor dado para ε, usando ***Newton's difference quotient*** (Numerical differentiation - Wikipedia. Finite differences) **integrada** con la recomendación de una buena selección del *step size, h* (Numerical differentiation - Wikipedia. Step size), lo cual también se menciona en la sección "5.7 Numerical Derivatives" del libro Numerical Recipes. The Art of Scientific Computing.

30. Un subprograma que:

    - Reciba un número real positivo cercano a cero (ε_0) y la implementación / el subprograma / la "variable" de tipo `function` correspondiente a una función matemática de los reales en los reales ($f : \mathbb{R} \to \mathbb{R}$), derivable en todo su dominio.
    - **Devuelva un subprograma / una "variable" de tipo `function`** que:
        - Reciba un número real (x) y un número real positivo cercano a cero (ε), cuyo valor predeterminado debe ser ε_0.
        - Devuelva una **aproximación de la derivada (derivada numérica)** de la función f en x, a partir del valor dado para ε, usando ***the symmetric difference quotient*** (Numerical differentiation - Wikipedia. Finite differences) **integrada** con la recomendación de una buena selección del *step size, h* (Numerical differentiation - Wikipedia. Step size), lo cual también se menciona en la sección "5.7 Numerical Derivatives" del libro Numerical Recipes. The Art of Scientific Computing.

7 Arreglos

En las secciones anteriores se ha trabajado con tipos de datos primitivos o elementales (*atomic types*) que representan un único dato como un número entero o real, una letra o un valor booleano. Sin embargo, en muchas situaciones se requiere trabajar con una colección de valores que agrupados hacen parte de un todo. Los tipos de datos compuestos son estructuras de datos que permiten en una misma variable e identificador puede representar más de un dato o valor primitivo. Hay dos tipos de datos compuestos con los que ya hemos trabajado, uno de ellos son las cadenas, que son una secuencia de carácteres, y el otro son los números complejos, que están compuestos por dos números reales.

En esta sección se hará una revisión de los arreglos o *arrays*, tipos de datos compuestos y estructuras de datos básicas que incorporan una gran mayoría de lenguajes de programación. Un arreglo es una colección finita de datos del mismo tipo, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común. Para referirse a un determinado elemento de un arreglo se deberá utilizar un índice, el cual especifica su posición relativa dentro del arreglo. Los arreglos podrán ser unidimensionales o multidimensionales, sin embargo, como la memoria de la computadora es lineal, un arreglo multidimensional tendrá que tener un equivalente linealizado (un arreglo unidimensional equivalente) para su almacenamiento en memoria.

Se espera que al finalizar las actividades de esta sección, el estudiante tenga clara la manera en que se hace uso de los arreglos en el diseño de un algoritmo bajo el paradigma de programación imperativa estructurada.

Preparación de clase

- Leer las secciones 7.1 a 7.6 del libro: L. Joyanes Aguilar, *Fundamentos de programación: algoritmos, estructura de datos y objetos*. McGraw Hill, **2020** [Online]. Disponible en http://www.ebooks7-24.com.ezproxy.unal.edu.co/?il=10409 o en http://www.ebooks7-24.com.ezproxy.biblored.gov.co/?il=10409

En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó en el texto, incluya su discusión, reflexiones y conclusiones al respecto; exponga lo que no entendió e intente encontrar por su cuenta respuestas a las preguntas que le surgieron, para poder compartirlas en clase.

7.1 Arreglos unidimensionales

Ya se mencionó que un arreglo es una colección finita de datos del **mismo** tipo, que se almacenan en posiciones consecutivas de memoria y reciben un nombre común. En el caso de un arreglo unidimensional estamos hablando de una estructura de datos que tendría cierta equivalencia o similaridad con el objeto matemático: vector, ya que ambos corresponden a una secuencia de valores ordenados, que son indexados por un único índice asociado a una única dimensión. El tamaño de un arreglo unidimensional es la cantidad de elementos del arreglo en su única dimensión.

Las variables de tipo arreglo se deben declarar como cualquier otra variable. Sin embargo, cuando un arreglo se declara, adicionalmente se debe informar el tipo de dato de los elementos del arreglo. Es decir, la declaración se debe hacer en función de otro tipo de dato conocido o previamente definido. En pseudocódigo, la declaración de un arreglo unidimensional podría hacerse de la siguiente manera:

```
var
  arreglo tipo_dato: nombre_arreglo[tamaño_arreglo], …
  …
```

Dependiendo del lenguaje de programación, la indexación de los elementos del arreglo: `nombre_arreglo` va de `0` a `tamaño_arreglo - 1` o de `1` a `tamaño_arreglo`. Usando los valores del índice, los elementos de un arreglo son igualmente accesibles y cualquiera de ellos puede ser seleccionado para lo que se desee hacer. Para seleccionar el elemento en la posición i del arreglo, se escribe: `nombre_arreglo[i]`. Para todo efecto práctico, cada elemento del arreglo: `nombre_arreglo` se comporta como si fuese una variable de tipo: `tipo_dato`.

Los elementos de un arreglo se pueden recorrer por completo, mediante el uso de una estructura iterativa y una variable que pase por todos los valores de la indexación. Por ejemplo:

```
i <- 0
mientras i < tamaño_arreglo hacer
  escribir(nombre_arreglo[i])
  i <- i + 1
fin_mientras
```

De la misma manera en que ocurre con los demás tipos de datos, un subprograma puede recibir variables de tipo arreglo o puede devolver una variable de tipo arreglo. La capacidad de devolver una variable de tipo arreglo permite a un subprograma devolver uno o más valores o datos de un mismo tipo de dato.

Especialmente en el caso de los arreglos, debemos tener claras las dos formas distintas en que un programa principal puede pasar / un subprograma puede recibir los parámetros, **parámetros por valor** y **parámetros por referencia**.

7.1.1 Paso de parámetros por valor

Paso por valor o **paso por copia** significa que cuando se llama un subprograma, este recibe una copia de los valores de los parámetros en un espacio de memoria aparte. Si el valor de un parámetro se modifica mediante una o más asignaciones o instrucciones del subprograma, el cambio solamente tiene efecto dentro del subprograma y no por fuera de él.

En varios lenguajes de programación, los parámetros que sean de un tipo de dato primitivo o elemental se suelen pasan por valor / copia.

7.1.2 Paso de parámetros por referencia

Paso por referencia o **paso por dirección** significa que cuando se llama un subprograma, este recibe la dirección de lo que se le halla dado a los parámetros. Esto quiere decir que las etiquetas, identificadores o nombres de los parámetros al interior del subprograma quedan apuntando a elementos con sus propios espacios de memoria que existen y se encuentran fuera del subprograma. Esto también quiere decir que cuando una o más asignaciones o instrucciones del subprograma modifican los valores de los parámetros, estas modificaciones tienen efecto en los espacios de memoria que existen y seguirán existiendo fuera del subprograma (las modificaciones dentro del subprograma tendrán efecto dentro y fuera del subprograma).

En varios lenguajes de programación, los parámetros que sean de tipo de dato arreglo (**array**) se suelen pasar por referencia / dirección.

7.2 Arreglos multidimensionales

Los arreglos multidimensionales son aquellos en donde cada elemento del arreglo está indexado por más de un índice (por más de un subíndice cuando los elementos del arreglo se denotan de la siguiente manera: $a_{i,j,k,l}$). La dimensión de un arreglo es la cantidad de índices distintos que tenga o use el arreglo para sus elementos (en $a_{i,j,k,l}$ serían 4 dimensiones). Cada índice tendrá una cantidad máxima de valores posibles y esa cantidad se denomina el tamaño de cada dimensión (n, m, o y p si $0 \leq i < n$, $0 \leq j < m$, $0 \leq k < o$, $0 \leq l < p$). La cantidad total de elementos del arreglo es igual al producto de los tamaños de todas las dimensiones ($(n)(m)(o)(p)$). Cuando se habla del tamaño de un arreglo, se está refiriendo a los tamaños de cada dimensión (el arreglo con elementos $a_{i,j,k,l}$ es de tamaño $n \times m \times o \times p$).

En pseudocódigo, la declaración de un arreglo multidimensional podría hacerse de la siguiente manera:

var
  arreglo tipo_dato: nombre_arreglo[tam_dim1, tam_dim2, tam_dim3, …], …
  …

Obviamente, un arreglo bidimensional necesita dos índices para poder identificar a cada elemento del arreglo (de la misma manera en que cada elemento de una matriz tiene asociado dos subíndices). La referencia a un determinado elemento del arreglo bidimensional requiere emplear un primer valor para el primer índice y un segundo valor para el segundo índice (si es de alguna utilidad, el primer valor se puede interpretar como un número de fila y el segundo como un número de columna, pero no se está obligado a seguir esta interpretación). Por ejemplo, `nombre_arreglo[0, 0]` hace referencia al elemento de la posición: 0, 0 (en la interpretación mencionada, el elemento de la fila 0 y la columna 0). Cada elemento del arreglo tendrá todos los beneficios y limitaciones de una variable de tipo: `tipo_dato`. Para seleccionar un elemento de un arreglo multidimensional bastaría hacer: `nombre_arreglo[i, j, k, ...]`, para los valores i, j, k, etc. que correspondan a la posición del elemento deseado.

Para recorrer por completo los elementos de un arreglo bidimensional se requiere usar dos estructuras iterativas anidadas y dos variables que pasen por todos los valores de ambas indexaciones (una estructura y una variable por cada una de las dos dimensiones). Por ejemplo, fijando el valor para el primer índice mientras se recorren los valores del segundo índice (recorrido por filas, es decir se fija una fila mientras se recorren las columnas, para luego pasar a la siguiente fila, etc.):

```
i <- 0
mientras i < tam_dim1 hacer
  j <- 0
  mientras j < tam_dim2 hacer
    escribir(nombre_arreglo[i, j])
    j <- j + 1
  fin_mientras
  i <- i + 1
fin_mientras
```

o, fijando el valor para el segundo índice mientras se recorren los valores del primer índice (recorrido por columnas, es decir se fija una columna mientras se recorren las filas, para luego pasar a la siguiente columna, etc.):

```
j <- 0
mientras j < tam_dim2 hacer
  i <- 0
  mientras i < tam_dim1 hacer
    escribir(nombre_arreglo[i, j])
    i <- i + 1
  fin_mientras
  j <- j + 1
fin_mientras
```

Para recorrer los elementos de un arreglo de d dimensiones, se requieren d estructuras iterativas anidadas con sus respectivas d variables para los d índices.

7.3 Ejercicios

7.3.1 Parte A

Para los siguientes puntos:

- Implemente un único programa principal que se encargue de interactuar con el usuario y que le permita probar los subalgoritmos requeridos con los valores que desee.
- Los subalgoritmos deben recibir todo arreglo como un parámetro por referencia / dirección.

Realice un adecuado **análisis** y **diseño** de un subalgoritmo cuya única tarea sea:

1. Almacenar en todos los elementos un arreglo real `arregloRecibeDatos` de tamaño `k`, un mismo número real `real1` (El subalgoritmo recibe `arregloRecibeDatos`, `real1` y `k`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que hace para una variable de tipo arreglo, algo similar a lo que hace `variable = 1.2` para `variable` de tipo de dato real.

2. Imprimir por pantalla los valores almacenados en cada uno de los elementos de un arreglo real `arregloTieneDatos` de tamaño `k` (El subalgoritmo recibe `arregloTieneDatos` y `k`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que hace para una variable de tipo arreglo, algo similar a lo que hace `escribir(variable)` para una `variable` de un tipo de dato elemental / primitivo.

3. Almacenar en un arreglo real `arregloRecibeDatos` de tamaño `k`, los números reales que el usuario ingrese por teclado (El subalgoritmo recibe `arregloRecibeDatos` y `k`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que hace para una variable de tipo arreglo, algo similar a lo que hace `leer(variable)` para un `variable` de un tipo de dato elemental / primitivo.

4. Almacenar en un arreglo real `arregloRecibeDatos` de tamaño `k`, los valores de cada uno de los elementos de otro arreglo real `arregloTieneDatos` del mismo tamaño (El subalgoritmo recibe `arregloRecibeDatos`, `arregloTieneDatos` y `k`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que hace para variables de tipo arreglo, algo similar a lo que hace `variable1 = variable2` para una `variable1` y `variable2` de un tipo de dato elemental / primitivo.

5. Almacenar en un arreglo real `arregloRecibeDatos` de tamaño `k`, una secuencia de valores generados a partir de: un número real `valIni` para el valor inicial de la secuencia y un número real `valIncrem` para el incremento (o decremento si es un número negativo) que se debe aplicar (El subalgoritmo recibe `arregloRecibeDatos`, `k`, `valIni` y `valIncrem`. El subalgoritmo no devuelve.). No es necesario un número real para el valor final de la secuencia, ya que el valor final queda totalmente determinado por el valor inicial, el incremento y el tamaño del arreglo. Si el valor inicial es 1.23, el incremento es -0.98 y el

tamaño es 5, entonces al valor final no le queda de otra que ser -2.69 (Los valores que recibiría `arregloRecibeDatos` serían 1.23, 0.25, -0.73, -1.71 y -2.69).

6. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el inverso aditivo de cada uno de los elementos de otro arreglo real `arregloTieneDatos` del mismo tamaño (El subalgoritmo recibe `arregloResultado`, `arregloTieneDatos` y `k`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que hace para variables de tipo arreglo, algo similar a lo que hace `resultado = -variable` para `resultado` y `variable` de tipo de dato real.

7. Devolver la suma de todos los elementos de un arreglo real `arregloTieneDatos` de tamaño `k` (El subalgoritmo recibe `arregloTieneDatos` y `k`. El subalgoritmo devuelve un número real `realResultado`.).

8. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, la suma término a término de los elementos de dos arreglos reales `arregloPrimerSumando` y `arregloSegundoSumando`, del mismo tamaño que `arregloResultado` (El subalgoritmo recibe `arregloResultado`, `arregloPrimerSumando`, `arregloSegundoSumando` y `k`. El subalgoritmo no devuelve.).

9. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, la suma de un número real `realSumando1` con cada uno de los elementos de un arreglo real `arregloSumando2` del mismo tamaño que `arregloResultado` (El subalgoritmo recibe `arregloResultado`, `realSumando1`, `arregloSumando2` y `k`. El subalgoritmo no devuelve.). ¿Qué debo hacer si adicionalmente quiero un subalgoritmo que haga la suma de cada uno de los elementos de un arreglo real `arregloSumando1` con un número real `realSumando1` (Este subalgoritmo recibe `arregloResultado`, `arregloSumando1`, `realSumando2` y `k`. Este subalgoritmo no devuelve)?.

10. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, la suma acumulada de los elementos de otro arreglo real `arregloTieneDatos` del mismo tamaño (El subalgoritmo recibe `arregloResultado`, `arregloTieneDatos` y `k`. El subalgoritmo no devuelve.). Es decir, para un arreglo con los valores `2.1`, `4.3`, `6.5` y `8.7` (en ese orden), el arreglo resultante con la suma acumulada deberá tener los valores `2.1`, `6.4`, `12.9` y `21.6` (en ese orden).

11. Devolver el producto de todos los elementos de un arreglo real `arregloTieneDatos` de tamaño `k` (El subalgoritmo recibe `arregloTieneDatos` y `k`. El subalgoritmo devuelve un número real `realResultado`.).

12. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el producto término a término de los elementos de dos arreglos reales `arregloPrimerFactor` y `arregloSegundoFactor`, del mismo tamaño que `arregloResultado` (El subalgoritmo recibe `arregloResultado`, `arregloPrimerFactor`, `arregloSegundoFactor` y `k`. El subalgoritmo no devuelve.).

13. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el producto de un número real `realFactor1` con cada uno de los elementos de un arreglo real `arregloFactor2` del mismo tamaño que `arregloResultado` (El subalgoritmo recibe dos arreglos y un número real. El subalgoritmo no devuelve.). ¿Qué debo hacer si adicionalmente quiero un subalgoritmo que haga la suma de cada uno de los elementos de un arreglo real `arregloFactor1` con un número real `realFactor1` (Este subalgoritmo recibe `arregloResultado`, `arregloFactor1`, `realFactor2` y `k`. Este subalgoritmo no devuelve)?

14. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el producto acumulado de los elementos de otro arreglo real `arregloTieneDatos` del mismo tamaño (El subalgoritmo recibe `arregloResultado`, `arregloTieneDatos` y `k`. El subalgoritmo no devuelve.). Es decir, para un arreglo con los valores `2.0`, `3.0`, `5.0` y `7.0` (en ese orden), el arreglo resultante con el producto acumulado deberá tener los valores `2.0`, `6.0`, `30.0` y `210.0` (en ese orden).

15. Devolver el producto punto / escalar de los dos objetos matemáticos vector asociados a dos arreglos reales `arregloPrimerFactor` y `arregloSegundoFactor` del mismo tamaño `k` (El subalgoritmo recibe `arregloPrimerFactor`, `arregloSegundoFactor` y `k`. El subalgoritmo devuelve un número real `realResultado`.).

16. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, la diferencia término a término de los elementos de dos arreglos `arregloMinuendo` y `arregloSustraendo`, del mismo tamaño que `arregloResultado`. (El subalgoritmo recibe `arregloResultado`, `arregloMinuendo`, `arregloSustraendo` y `k`. El subalgoritmo no devuelve.).

17. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, la diferencia de cada uno de los elementos de un arreglo real `arregloMinuendo`, del mismo tamaño que `arregloResultado`, con un número real `realSustraendo`. (El subalgoritmo recibe `arregloResultado`, `arregloMinuendo`, `realSustraendo` y `k`. El subalgoritmo no devuelve.).

18. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, la diferencia de un número real `realMinuendo` y cada uno de los elementos de un arreglo real `arregloSustraendo` del mismo tamaño que `arregloResultado` (El subalgoritmo recibe `arregloResultado`, `realMinuendo`, `arregloSustraendo` y `k`. El subalgoritmo no devuelve.).

19. Almacenar en un arreglo real `arregloResultado` de tamaño `k-1`, la diferencia "respecto a un (1) elemento hacia atrás" de otro arreglo real `arregloTieneDatos` de tamaño `k` (El subalgoritmo recibe `arregloResultado`, `arregloTieneDatos` y `k`. El subalgoritmo no devuelve.). Es decir, para un arreglo con los valores `1.2`, `3.4`, `7.8` y `-9.0` (en ese orden), el arreglo resultante con la diferencia "respecto a un (1) elemento hacia atrás" deberá tener los valores `2.2`, `4.4` y `-16.8` (en ese orden).

20. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el cociente término a término de los elementos de dos arreglos reales `arregloDividendo` y `arregloDivisor`,

del mismo tamaño que `arregloResultado`. (El subalgoritmo recibe `arregloResultado`, `arregloDividendo`, `arregloDivisor` y `k`. El subalgoritmo no devuelve.).

21. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el cociente de cada uno de los elementos de un arreglo real `arregloDividendo`, del mismo tamaño que `arregloResultado`, con un número real `realDivisor`. (El subalgoritmo recibe `arregloResultado`, `arregloDividendo`, `realDivisor` y `k`. El subalgoritmo no devuelve.).
22. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, el cociente de un número real `realDividendo` y cada uno de los elementos de un arreglo real `arregloDivisor` del mismo tamaño que `arregloResultado` (El subalgoritmo recibe `arregloResultado`, `realDividendo`, `arregloDivisor` y `k`. El subalgoritmo no devuelve.).
23. Devolver el valor booleano que corresponda, dependiendo de si son iguales, los dos objetos matemáticos vector asociados a dos arreglos reales `arregloDatosVector1` y `arregloDatosVector2`, ambos de tamaño `k` (El subalgoritmo recibe `arregloDatosVector1`, `arregloVector2` y `k`. El subalgoritmo devuelve un valor booleano `boolResultado`.). Es decir, un subalgoritmo que hace para variables de tipo arreglo, algo similar a lo que hace `variable1 == variable2` para `variable1` y `variable2` de un tipo de dato elemental / primitivo.
24. Devolver el valor booleano que corresponda, dependiendo de si son diferentes, los dos objetos matemáticos vector asociados a dos arreglos reales `arregloDatosVector1` y `arregloDatosVector2`, ambos de tamaño `k` (El subalgoritmo recibe `arregloDatosVector1`, `arregloVector2` y `k`. El subalgoritmo devuelve un valor booleano `boolResultado`.). Es decir, un subalgoritmo que hace para variables de tipo arreglo, algo similar a lo que hace `variable1 <> variable2` para `variable1` y `variable2` de un tipo de dato elemental / primitivo.
25. Devolver la norma euclidiana al cuadrado del objeto matemático vector asociado a un arreglo real `arregloDatosVector` de tamaño `k` (El subalgoritmo recibe `arregloDatosVector` y `k`. El subalgoritmo devuelve un número real `realResultado`.).
26. Devolver la distancia euclidiana al cuadrado entre los dos objetos matemáticos vector asociados a dos arreglos reales `arregloDatosVector1` y `arregloDatosVector2`, ambos de tamaño `k` (El subalgoritmo recibe `arregloDatosVector1`, `arregloVector2` y `k`. El subalgoritmo devuelve un número real `realResultado`.).

7.3.2 Parte B

Para los siguientes puntos:

- Los subalgoritmos deben recibir todo arreglo como un parámetro por referencia / dirección.

- Se debe poder trabajar y manipular los datos de lo que podría ser un arreglo multidimensional, sin ningún problema, mediante el uso de un arreglo unidimensional. Esto para mostrar que todo arreglo multidimensional internamente es una colección lineal de posiciones consecutivas de memoria (es decir, internamente es equivalente a un arreglo unidmensional).

Realice un adecuado **análisis** y **diseño** de un subalgoritmo cuya única tarea sea:

1. Almacenar en un arreglo (unidimensional) real `arregloRecibeDatosMatriz` de tamaño `k = n*p`, los elementos de un objeto matemático matriz de tamaño $n \times m$ que el usuario ingrese por teclado (El subalgoritmo recibe `arregloRecibeDatosMatriz`, `n` y `p`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que almacena adecuadamente en un arreglo unidimensional, los datos de una matriz dada elemento a elemento por el usuario mediante el teclado (Ayuda: pida y almacene los datos de la matriz de manera consecutiva por columnas).

2. Imprimir por pantalla los valores almacenados en cada uno de los elementos de un arreglo (unidimensional) real `arregloTieneDatosMatriz` de tamaño `k = n*p` que tiene almacenados *adecuadamente por columnas* los datos de un objeto matemático matriz de tamaño $n \times p$ (El subalgoritmo recibe `arregloTieneDatosMatriz`, `n` y `p`. El subalgoritmo no devuelve.). Es decir, un subalgoritmo que imprima adecuadamente el contenido de un arreglo unidimensional, que tiene almacenados los datos de una matriz, que por ejemplo se recibió usando el subalgoritmo del punto anterior.

3. Devolver el elemento de la posición `i`, `j` de los datos de un objeto matemático matriz de tamaño $n \times p$, que fue almacenado *adecuadamente por columnas*, en un arreglo (unidimensional) real `arregloTieneDatosMatriz` de tamaño `k = n*p` (El subalgoritmo recibe `arregloTieneDatosMatriz`, `i`, `j`, `n` y `p`. El subalgoritmo devuelve un número real `realElemento`.). Es decir, un subalgoritmo que devuelve un elemento de una matriz que está almacenada en un arreglo unidimensional.

4. Almacenar en un arreglo (unidimensional) real `arregloMatrizResultado` de tamaño `k = n*p`, los datos de la matriz resultado del producto término a término entre dos matrices almacenadas en los arreglos unidimensionales: `arregloMatrizPrimerFactor` y `arregloMatrizSegundoFactor` del mismo tamaño que `arregloMatrizResultado` (El subalgoritmo recibe `arregloMatrizResultado`, `arregloMatrizFactor1`, `arregloMatrizFactor2`, `n` y `p`. El subalgoritmo no devuelve.). ¿hay varias potenciales soluciones para el subalgoritmo solicitado? ¿entre ellas cuál sería la más eficiente? ¿es posible aprovechar el subalgoritmo que hace el producto término a término de los elementos de dos arreglos reales, solicitado en un ejercicio de la Parte A?

5. Almacenar en un arreglo (unidimensional) real `arregloMatrizResultado` de tamaño `k = n*p`, los datos de la matriz resultado del producto matricial entre dos matrices almacenadas en los arreglos unidimensionales: `arregloMatriz1` de tamaño `k1 = n*m` y `arregloMatriz2` de tamaño `k2 = m*p` (El subalgoritmo recibe

`arregloMatrizResultado`, `arregloMatrizFactor1`, `arregloMatrizFactor2`, `n`, `m` y `p`. El subalgoritmo no devuelve.).

7.3.3 Parte C

¿Por qué los estadísticos deben saber acerca de la generación de valores pseudoaleatorios y los métodos Monte Carlo?

> "Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle. They are often used in physical and mathematical problems and are most useful when it is difficult or impossible to use other approaches. Monte Carlo methods are mainly used in three problem classes: optimization, numerical integration, and generating draws from a probability distribution."
> Monte Carlo method - Wikipedia

Para los siguientes puntos:

- Los subalgoritmos deben recibir todo arreglo como un parámetro por referencia / dirección.
- Consultar:

  Blanco Castañeda, Liliana. (2004). ***Probabilidad***. Editorial UN.
  Capítulo 8: Simulaciones básicas 8.1: Generación de números aleatorios
  https://repositorio.unal.edu.co/bitstream/handle/unal/53471/9587014499.PDF?sequence=2

  y

  Press, W. H., William, H., Teukolsky, S. A., Saul, A., Vetterling, W. T., & Flannery, B. P. (2007). ***Numerical recipes 3rd edition: The art of scientific computing***. Cambridge university press.
  Chapter 7: Random Numbers
  7.0: Introduction, 7.1: Uniform Deviates
  http://numerical.recipes/book/book.html

Realice un adecuado **análisis** y **diseño** de un subalgoritmo cuya única tarea sea:

1. Almacenar en un arreglo entero `arregloResultado` de tamaño `k`, una secuencia de k valores pseudoaleatorios enteros, generados mediante el método denominado *linear congruential generator* (método que tiene una fórmula de la forma `(multiplier * seed +`

`summand) mod modulus`). El subalgoritmo recibe `arregloResultado`, `seed`, `multiplier`, `summand`, `modulus`, `k`. El subalgoritmo no devuelve.

2. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, una secuencia de k valores pseudoaleatorios enteros entre dos números enteros dados `entero1` y `entero2`, añadiendo un paso más a la generación de valores enteros con el método: *linear congruential generator* (método que tiene una fórmula de la forma `(multiplier * seed + summand) mod modulus`). El subalgoritmo recibe `arregloResultado`, `entero1`, `entero2`, `seed`, `multiplier`, `summand`, `modulus`, `k`. El subalgoritmo no devuelve. Si ya tengo un arreglo con valores pseudoaleatorios enteros generados, ¿puedo transformar esos valores que ya tengo, para obtener un arreglo con valores pseudoaleatorios enteros entre dos números enteros dados `entero1` y `entero2`?

3. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, una secuencia de k valores pseudoaleatorios reales entre 0 y 1, añadiendo un paso más a la generación de valores enteros con el método: *linear congruential generator* (método que tiene una fórmula de la forma `(multiplier * seed + summand) mod modulus`). El subalgoritmo recibe `arregloResultado`, `seed`, `multiplier`, `summand`, `modulus`, `k`. El subalgoritmo no devuelve. Si ya tengo un arreglo con valores pseudoaleatorios enteros generados, ¿puedo transformar esos valores que ya tengo, para obtener un arreglo con valores pseudoaleatorios reales entre 0 y 1?

4. Almacenar en un arreglo real `arregloResultado` de tamaño `k`, una secuencia de k valores pseudoaleatorios reales entre dos números reales dados `real1` y `real2`, añadiendo un par de pasos más a la generación de valores enteros con el método: *linear congruential generator* (método que tiene una fórmula de la forma `(multiplier * seed + summand) mod modulus`). El subalgoritmo recibe `arregloResultado`, `real1`, `real2`, `seed`, `multiplier`, `summand`, `modulus`, `k`. El subalgoritmo no devuelve. ¿Si ya tengo un arreglo con valores pseudoaleatorios enteros generados, ¿puedo transformar esos valores que ya tengo, para obtener un arreglo con valores pseudoaleatorios enteros entre reales dados `real1` y `real2`?

Parte III

POA

8 PIEP y POA en R

En esta sección se hará una revisión de los elementos básicos de programación imperativa estructurada procedimental (PIEP) y de programación orientada a arreglos (POA) de una implementación en R (obviamente, R como lenguaje de programación y **NO** como herramienta de cálculo o de análisis de datos).

En esta revisión se hace una pequeña introducción a R, a sus elementos básicos, a sus principales estructuras de control, a la creación de subprogramas, y al manejo de sus tipos de datos compuestos.

Se espera que al finalizar las actividades de esta sección, el estudiante entienda y tenga clara la manera en que un algoritmo, diseñado bajo los paradigmas de programación imperativa estructurada procedimental y orientado a arreglos, se puede implementar mediante el uso del lenguaje de programación R.

> **Preparación de clase**
>
> - Para la siguiente sección, lea **todo** y ejecute **todo** el código que allí se incluye, haciendo **todas** las pruebas, cambios y experimentos que se les puedan ocurrir sobre dicho código.
>
> En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó, ejecutó, probó y experimentó; incluya su discusión, reflexiones y conclusiones al respecto; exponga lo que no entendió e intente encontrar por su cuenta respuestas a las preguntas que le surgieron, para poder compartirlas en clase.

8.1 Elementos básicos de PIEP y POA en R

Cuaderno computacional en Google Colaboratory:
Elementos básicos, estructuras de control, creación de subprogramas y tipos de datos compuestos en R

8.2 Ejemplos

Ejemplo 8.1. Haga un adecuado análisis, diseño e implementación en R de un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número entero y que devuelvan el **factorial** del número recibido. Implemente una prueba rápida (una serie de instrucciones) que le permita probar los subprogramas requeridos para varios valores distintos. Por último, implemente una serie de instrucciones que permita realiar una comparación de tiempos entre las dos soluciones solicitadas y la función `factorial()` de R.

Análisis

El factorial de un número entero positivo k, denotado $k!$ se puede definir como el producto de todos los números enteros positivos menores o iguales que k:

$$k! = (2)(3)\cdots(k-2)(k-1)(k)$$

o lo que es lo mismo,

$$k! = (k)(k-1)(k-2)\cdots(3)(2)$$

Además, es necesario tener en cuenta que,

$$0! = 1$$

y es buena idea tener en cuenta que,

$$1! = 1 \qquad 2! = 2$$

Implementación en R (solución de PIEP)

```r
# Solamente sirve para un parámetro vector con un solo elemento:
mi_fact_klength1_PIEP <- function(klength1){
  if(length(klength1) == 1){
    if(klength1 > 2){
      res <- klength1
      while(klength1 > 2){
        klength1 <- klength1 - 1
        res <- res * klength1
      }
      return(res)
    }else{
      if(klength1 >= 0){
        return(ifelse(klength1 == 0, 1, klength1))
      }
    }
  }
  return(NaN)
}

# Factorial elemento a elemento de un vector dado:
mi_fact_PIEP <- function(k){
  n <- length(k)
  res <- NULL
  for(i in 1:n){
    res[i] = mi_fact_klength1_PIEP(k[i])
  }
  return(res)
}
```

Implementación en R (solución de POA)

```r
# Solamente sirve para un parámetro vector con un solo elemento:
mi_fact_klength1_POA <- function(klength1){
  if(length(klength1)==1){
    if(klength1 > 2){
      return(prod(2:klength1))
    }else{
      if(klength1 >= 0){
        return(ifelse(klength1 == 0, 1, klength1))
      }
    }
  }
  return(NaN)
}

# `Vectorize()` toma una función y devuelve una función capaz de devolver
# el mismo resultado pero elemento a elemento de un vector dado:
mi_fact_POA_Vectorize <- Vectorize(mi_fact_klength1_POA)

# Implementación propia (con la ayuda de `sapply()`) capaz de devolver el
# factorial elemento a elemento de un vector dado:
mi_fact_POA_sapply <- function(k){
  sapply(k, mi_fact_klength1_POA)
}
```

Prueba rápida

```r
v <- c(-14, -8, 0, 1, 2, 8, 14)
for(i in v){
  cat("(", sprintf(i, fmt='%3.0f'), ")!\t",
      sprintf(mi_fact_klength1_PIEP(i), fmt='%12.0f'), "\t",
      sprintf(mi_fact_klength1_POA(i), fmt='%12.0f'), "\t",
      sprintf(factorial(i), fmt='%12.0f'), "\n", sep = "")
}
```

```
(-14)!          NaN             NaN             NaN
( -8)!          NaN             NaN             NaN
(  0)!            1               1               1
(  1)!            1               1               1
```

```
(  2)!             2               2               2
(  8)!         40320           40320           40320
( 14)!   87178291200     87178291200     87178291200
```

```r
v <- c(-14, -8, 0, 1, 2, 8, 14)
resul <- c(mi_fact_PIEP(v), mi_fact_POA_Vectorize(v),
           mi_fact_POA_sapply(v), factorial(v))
resul <- matrix(resul, ncol = 4)
colnames(resul) <- c("mi_fact_PIEP(v)", "mi_fact_POA_Vectorize(v)",
                     "mi_fact_POA_sapply(v)", "factorial(v)")
rownames(resul) <- paste0("(", v, ")!")
resul
```

```
      mi_fact_PIEP(v) mi_fact_POA_Vectorize(v) mi_fact_POA_sapply(v)
(-14)!             NaN                      NaN                   NaN
(-8)!              NaN                      NaN                   NaN
(0)!                 1                        1                     1
(1)!                 1                        1                     1
(2)!                 2                        2                     2
(8)!             40320                    40320                 40320
(14)!     87178291200              87178291200           87178291200
      factorial(v)
(-14)!          NaN
(-8)!           NaN
(0)!              1
(1)!              1
(2)!              2
(8)!          40320
(14)!  87178291200
```

Comparación de tiempos

```r
# `require()` intenta cargar la librería, si no la puede cargar devuelve FALSE
if (!require(microbenchmark)){
  install.packages("microbenchmark") # Instala librería
  library(microbenchmark) # Carga librería
}
```

```
Loading required package: microbenchmark
```

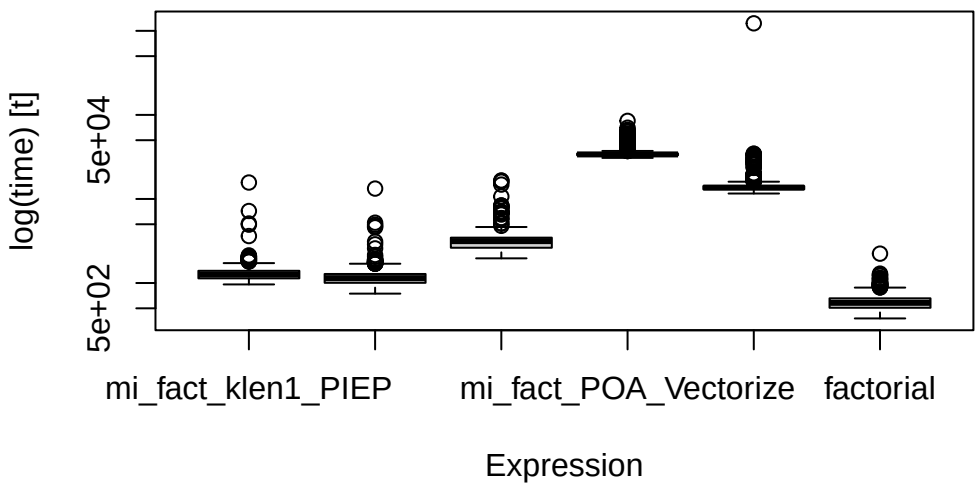
```r
set.seed(10310506)
k <- sample(8:14, 1)
cat("k =", k)
```

```
k = 13
```

```r
mbm <- microbenchmark("mi_fact_klen1_PIEP"={mi_fact_klength1_PIEP(k)},
                      "mi_fact_klen1_POA"={mi_fact_klength1_POA(k)},
                      "mi_fact_PIEP"={mi_fact_PIEP(k)},
                      "mi_fact_POA_Vectorize"={mi_fact_POA_Vectorize(k)},
                      "mi_fact_POA_sapply"={mi_fact_POA_sapply(k)},
                      "factorial"={factorial(k)},
                      times=1e3)
mbm # Tabla
```

```
Unit: nanoseconds
                 expr   min      lq      mean  median      uq     max neval
    mi_fact_klen1_PIEP   962  1135.5  1309.838  1263.5  1397.5   15682  1000
     mi_fact_klen1_POA   748  1005.0  1188.228  1142.0  1281.5   13291  1000
          mi_fact_PIEP  1965  2627.5  3189.759  3151.5  3465.0   16600  1000
 mi_fact_POA_Vectorize 30810 33085.0 34872.041 33880.5 34746.5   84871  1000
    mi_fact_POA_sapply 11609 13029.0 15365.418 13664.0 14256.5 1228847  1000
             factorial   379   507.5   593.854   581.0   657.5    2235  1000
```

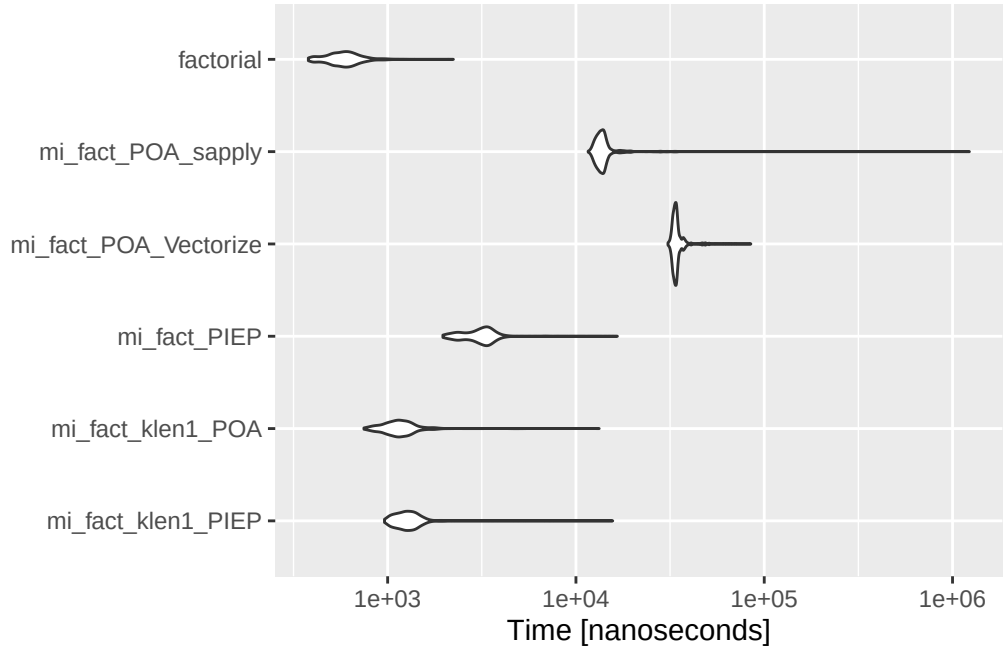
```r
boxplot(mbm, bty="n") # Gráfico boxplot
```



```r
if (!require(ggplot2)){
  install.packages("ggplot2")
  library(ggplot2)
}
```

```
Loading required package: ggplot2
```

```r
autoplot(mbm) # gráfico parecido al boxplot
```



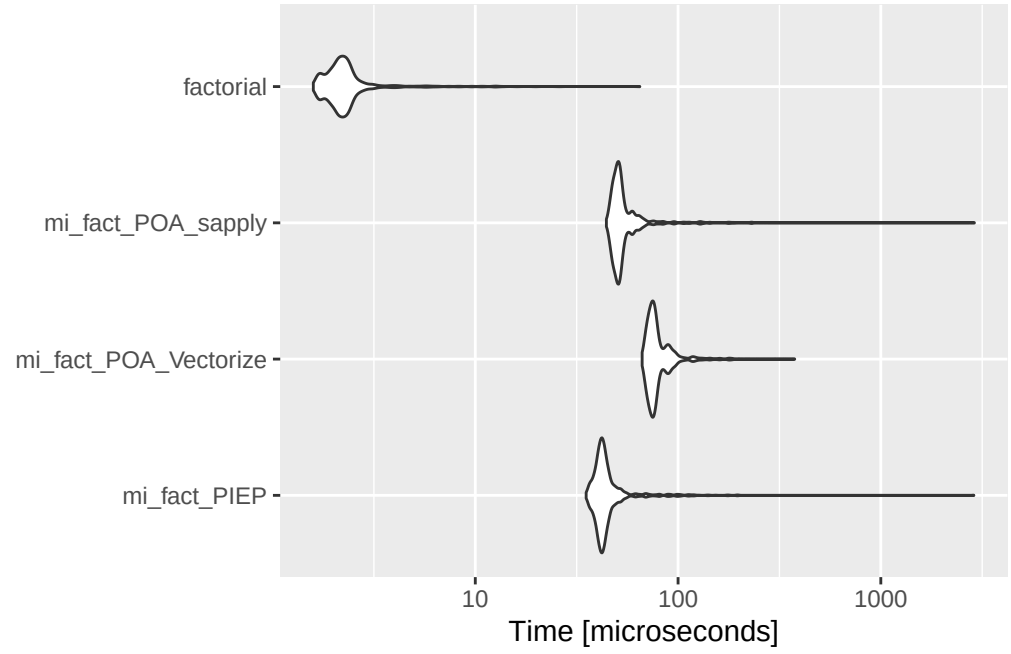

```r
# `require()` intenta cargar la librería, si no la puede cargar devuelve FALSE
if (!require(microbenchmark)){
  install.packages("microbenchmark") # Instala librería
  library(microbenchmark) # Carga librería
}
set.seed(10310506)
k <- sample(0:15, 25, replace=TRUE)
print(k)
```

```
 [1]  5  0  6 14 15  7 10 13  6  4 15 13  4  2 15 13 13  5 12  8  7  2  2  9  1
```

```r
mbm <- microbenchmark("mi_fact_PIEP"={mi_fact_PIEP(k)},
                      "mi_fact_POA_Vectorize"={mi_fact_POA_Vectorize(k)},
                      "mi_fact_POA_sapply"={mi_fact_POA_sapply(k)},
                      "factorial"={factorial(k)},
                      times=1e3)
mbm # Tabla
```

```
Unit: microseconds
                  expr    min      lq      mean  median      uq      max neval
          mi_fact_PIEP 35.209 40.4685 48.832832 42.4425 45.2075 2876.685  1000
 mi_fact_POA_Vectorize 66.508 72.8145 82.438810 76.1160 82.8050  376.105  1000
    mi_fact_POA_sapply 44.312 48.9350 59.391458 51.1950 54.5640 2893.457  1000
             factorial  1.589  1.9740  2.508096  2.1880  2.3955   64.809  1000
```

```r
if (!require(ggplot2)){
  install.packages("ggplot2")
  library(ggplot2)
}
autoplot(mbm) # gráfico parecido al boxplot
```



Ejemplo 8.2. Haga un adecuado análisis, diseño e implementación en R de un subprograma

con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que devuelvan una aproximación de la función **coseno** evaluada en un número real dado entre 0 y $\frac{\pi}{4}$. Los subprogramas solicitados deben tener en cuenta que:

- $$\sum_{k=0}^{n} \frac{(-1)^k}{(2k)!} c^{2k} \xrightarrow[n\to\infty]{} \cos(c)$$
- En esta ocasión, fije la cantidad de términos a usar de la serie. Es decir, primero **analice** y determine, a lo sumo, cuántos términos son mayores que 1×10^{-15} para c entre 0 y $\frac{\pi}{4}$, y luego en la **implementación** use siempre esa misma cantidad de términos.

Adicionalmente, implemente una prueba rápida (una serie de instrucciones) que le permita probar los subprogramas requeridos para varios valores distintos. Por último, implemente una serie de instrucciones que permita realizar una comparación de tiempos entre las dos soluciones solicitadas y la función `cos()` de R.

Análisis

Para c entre 0 y $\frac{\pi}{4}$,

$$\frac{1}{(2k)!} c^{2k} < \frac{1}{(2k)!} \left(\frac{\pi}{4}\right)^{2k}$$

o sea que si se encuentra k_0 tal que para todo $k > k_0$ se tiene que,

$$\frac{1}{(2k)!} \left(\frac{\pi}{4}\right)^{2k} < 1 \times 10^{-15}$$

entonces

$$\frac{1}{(2k)!} c^{2k} < 1 \times 10^{-15}$$

para todo $k > k_0$ y para todo c entre 0 y $\frac{\pi}{4}$.
Reescribiendo,

$$\begin{aligned} \frac{1}{(2k)!} \left(\frac{\pi}{4}\right)^{2k} &< 1 \times 10^{-15} \\ \left(\frac{\pi}{4}\right)^{2k} &< \frac{(2k)!}{1 \times 10^{15}} \\ \left(\frac{\pi}{4}\right)^{2k} - \frac{(2k)!}{1 \times 10^{15}} &< 0 \end{aligned}$$

Encontrando numéricamente una raíz para la función $f(k) = \left(\frac{\pi}{4}\right)^{2k} - \frac{(2k)!}{1\times 10^{15}}$, concluimos que $\frac{1}{(2k)!} \left(\frac{\pi}{4}\right)^{2k} < 1 \times 10^{-15}$ para todo $k > 8.000312$.
Es decir, para un c entre 0 y $\frac{\pi}{4}$, los términos $k = 0, 1, \ldots, 8$ de la serie serán los únicos que podrían llegar a ser mayores que 1×10^{-15}. En otras palabras, los términos $k = 0, 1, \ldots, 8$

serán más que suficientes para cualquier c entre 0 y $\frac{\pi}{4}$, ya que, para todo c entre 0 y $\frac{\pi}{4}$, los términos $k = 9$ en adelante serán menores que 1×10^{-15}.

Implementación en R (solución de PIEP)

```
# Unicamente para $0 \leq c \leq \frac{\pi}{4}$
mi_cos_serie_clen1_PIEP <- function(clen1){
  res <- 1.0
  if(clen1 != 0){
    cCuadr <- clen1 * clen1
    magnTerm <- cCuadr / 2
    restar <- TRUE
    res <- res - magnTerm
    iMax <- 16
    for(i in seq(4, iMax, 2)){
      magnTerm <- magnTerm * cCuadr / (i * (i - 1))
      if(restar){
        restar <- FALSE
        res <- res + magnTerm
      }
      else{
        restar <- TRUE
        res <- res - magnTerm
      }
    }
  }
  res
}

mi_cos_serie_PIEP <- function(c){
  len <- length(c)
  res <- NULL
  for(i in 1:len){
    res[i] = mi_cos_serie_clen1_PIEP(c[i])
  }
  res
}
```

Implementación en R (solución de POA)

```r
# Solamente sirve para un parámetro vector con un solo elemento:
mi_cos_serie_clen1_POA <- function(clen1){
  kMax <- 8
  iMax <- 16
  res <- 1
  if(clen1 != 0){
    num <- cumprod(rep(-clen1*clen1, kMax)) # -x^2, x^4, -x^6, ..., -x^{14}, x^{16}
    den <- cumprod(2:iMax)[c(TRUE, FALSE)] # 2!, 4!, 6!, ..., 16!
    res = res + sum(num/den)
  }
  res
}

# `Vectorize()` toma una función y devuelve una función capaz de devolver
# el mismo resultado pero elemento a elemento de un vector dado:
mi_cos_serie_POA_Vectorize <- Vectorize(mi_cos_serie_clen1_POA)

# Implementación propia (con la ayuda de `sapply()`) capaz de devolver el
# coseno elemento a elemento de un vector dado:
mi_cos_serie_POA_sapply <- function(c){
  sapply(c, mi_cos_serie_clen1_POA)
}

# Implementación propia capaz de devolver el
# coseno elemento a elemento de un vector dado:
mi_cos_serie_POA <- function(c){
  kMax <- 8
  iMax <- 16
  indx <- c != 0
  c <- c[indx]
  len <- length(c)
  num <- matrix(rep(-c*c, kMax), len)
  num <- apply(num, 1, cumprod)
  den <- cumprod(2:iMax)[c(TRUE, FALSE)]
  res <- NULL
  res[indx] <- 1 + (1/den) %*% num
  res[!indx] <- 1
  res
}
```

Prueba rápida

```
v <- seq(0, pi/4, length=7)
for(i in v){
  cat("cos(", sprintf(i, fmt='%18.16f'), ")\t",
      sprintf(mi_cos_serie_clen1_PIEP(i), fmt='%20.16f'), "\t",
      sprintf(mi_cos_serie_clen1_POA(i), fmt='%20.16f'), "\t",
      sprintf(cos(i), fmt='%20.16f'), "\n", sep = "")
}
```

```
cos(0.0000000000000000)   1.0000000000000000      1.0000000000000000      1.000000000000000
cos(0.1308996938995747)   0.9914448613738104      0.9914448613738104      0.991444861373810
cos(0.2617993877991494)   0.9659258262890684      0.9659258262890683      0.965925826289068
cos(0.3926990816987241)   0.9238795325112867      0.9238795325112867      0.923879532511286
cos(0.5235987755982988)   0.8660254037844386      0.8660254037844387      0.866025403784438
cos(0.6544984694978735)   0.7933533402912352      0.7933533402912352      0.793353340291235
cos(0.7853981633974483)   0.7071067811865475      0.7071067811865475      0.707106781186547
```

```
v <- seq(0, pi/4, length=7)
resul <- c(mi_cos_serie_PIEP(v), mi_cos_serie_POA_Vectorize(v),
           mi_cos_serie_POA_sapply(v), mi_cos_serie_POA(v),  cos(v))
resul <- matrix(resul, ncol = 5)
colnames(resul) <- c("mi_cos_serie_PIEP(v)", "mi_cos_serie_POA_Vectorize(v)",
                     "mi_cos_serie_POA_sapply(v)", "mi_cos_serie_POA(v)", "cos(v)")
rownames(resul) <- paste0("cos(", v, ")")
resul
```

```
                     mi_cos_serie_PIEP(v) mi_cos_serie_POA_Vectorize(v)
cos(0)                          1.0000000                      1.0000000
cos(0.130899693899575)          0.9914449                      0.9914449
cos(0.261799387799149)          0.9659258                      0.9659258
cos(0.392699081698724)          0.9238795                      0.9238795
cos(0.523598775598299)          0.8660254                      0.8660254
cos(0.654498469497873)          0.7933533                      0.7933533
cos(0.785398163397448)          0.7071068                      0.7071068
                     mi_cos_serie_POA_sapply(v) mi_cos_serie_POA(v)    cos(v)
cos(0)                                 1.0000000          1.0000000 1.0000000
cos(0.130899693899575)                 0.9914449          0.9914449 0.9914449
cos(0.261799387799149)                 0.9659258          0.9659258 0.9659258
cos(0.392699081698724)                 0.9238795          0.9238795 0.9238795
cos(0.523598775598299)                 0.8660254          0.8660254 0.8660254
```

```
cos(0.654498469497873)                0.7933533          0.7933533 0.7933533
cos(0.785398163397448)                0.7071068          0.7071068 0.7071068
```

Comparación de tiempos

```
# `require()` intenta cargar la librería, si no la puede cargar devuelve FALSE
if (!require(microbenchmark)){
  install.packages("microbenchmark") # Instala librería
  library(microbenchmark) # Carga librería
}
set.seed(11081348)
c <- runif(1, 0, pi/4)
cat("c =", c)
```

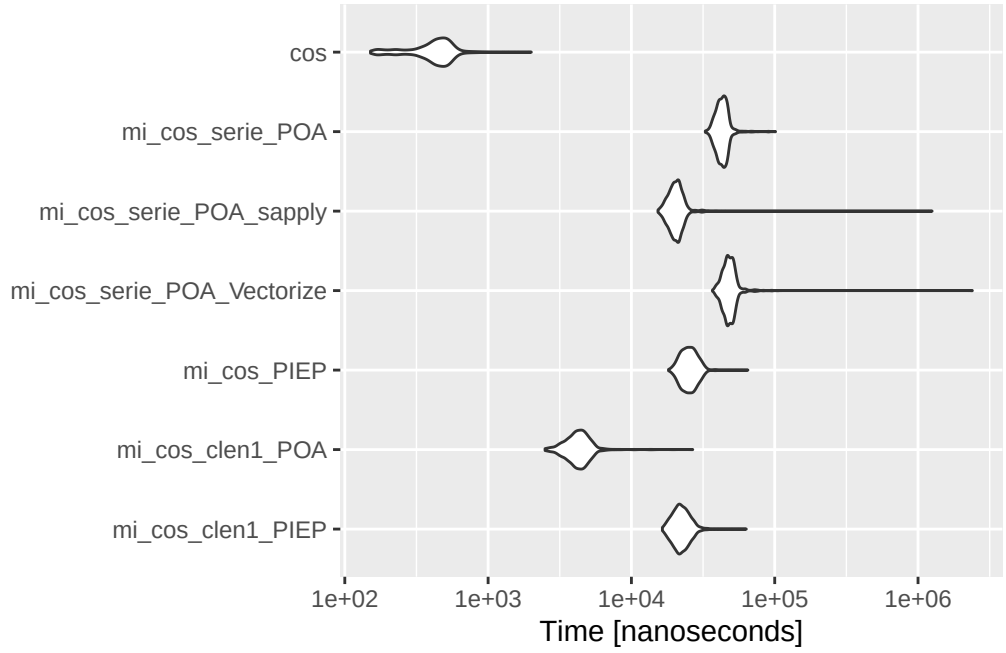
```
c = 0.06901471
```

```
mbm <- microbenchmark("mi_cos_clen1_PIEP"={mi_cos_serie_clen1_PIEP(c)},
                      "mi_cos_clen1_POA"={mi_cos_serie_clen1_POA(c)},
                      "mi_cos_PIEP"={mi_cos_serie_PIEP(c)},
                      "mi_cos_serie_POA_Vectorize"={mi_cos_serie_POA_Vectorize(c)},
                      "mi_cos_serie_POA_sapply"={mi_cos_serie_POA_sapply(c)},
                      "mi_cos_serie_POA"={mi_cos_serie_POA(c)},
                      "cos"={cos(c)},
                      times=1e3)
mbm # Tabla
```

```
Unit: nanoseconds
                      expr   min      lq      mean  median      uq     max
         mi_cos_clen1_PIEP 16441 20200.5 22747.599 22152.5 24427.5   63317
          mi_cos_clen1_POA  2496  3745.0  4342.456  4252.5  4755.5   26799
               mi_cos_PIEP 18144 22809.0 25611.851 25171.0 27699.5   65007
 mi_cos_serie_POA_Vectorize 36720 44915.5 50784.479 47857.0 51070.0 2392551
    mi_cos_serie_POA_sapply 15315 18856.5 21868.661 20381.5 21818.0 1253521
           mi_cos_serie_POA 32775 40093.5 43174.787 42935.5 45577.5  101338
                        cos   151   347.0   433.883   439.0   510.5    2002
 neval
  1000
  1000
  1000
  1000
  1000
```

```
1000
1000
```

```
if (!require(ggplot2)){
  install.packages("ggplot2")
  library(ggplot2)
}
autoplot(mbm) # gráfico parecido al boxplot
```



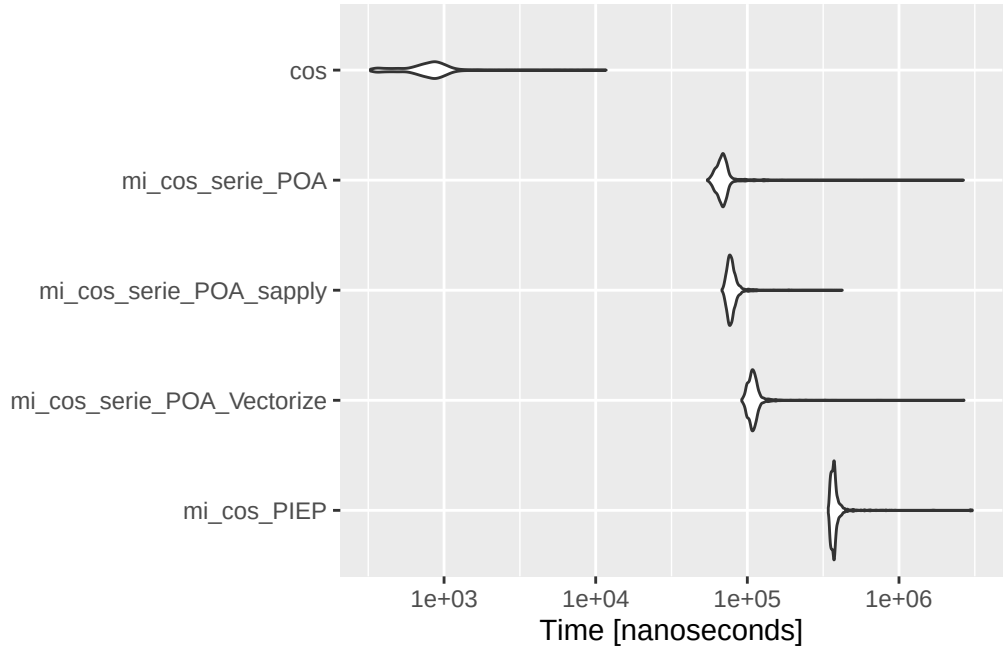

```
# `require()` intenta cargar la librería, si no la puede cargar devuelve FALSE
if (!require(microbenchmark)){
  install.packages("microbenchmark") # Instala librería
  library(microbenchmark) # Carga librería
}
set.seed(11081348)
c <- runif(20, 0, pi/4)
print(c)
```

```
 [1] 0.06901471 0.10078974 0.11829167 0.60410898 0.17810267 0.08819629
 [7] 0.46757247 0.42381621 0.45641103 0.74174346 0.75628327 0.74122467
[13] 0.55322552 0.47472847 0.37166666 0.32539549 0.01587700 0.48829730
[19] 0.15890795 0.58516338
```

```r
mbm <- microbenchmark("mi_cos_PIEP"={mi_cos_serie_PIEP(c)},
                      "mi_cos_serie_POA_Vectorize"={mi_cos_serie_POA_Vectorize(c)},
                      "mi_cos_serie_POA_sapply"={mi_cos_serie_POA_sapply(c)},
                      "mi_cos_serie_POA"={mi_cos_serie_POA(c)},
                      "cos"={cos(c)},
                      times=1e3)
mbm # Tabla
```

```
Unit: nanoseconds
                      expr    min       lq       mean   median       uq
               mi_cos_PIEP 341629 360265.5 397530.829 371763.0 382729.0
 mi_cos_serie_POA_Vectorize  91882 103729.5 117519.808 108654.5 113936.5
    mi_cos_serie_POA_sapply  68030  75066.0  81131.943  77835.0  81563.0
           mi_cos_serie_POA  54502  64746.5  73860.043  68483.0  72157.0
                      cos    327    650.5    846.888    811.0    941.0
     max neval
 3045700  1000
 2684092  1000
  422662  1000
 2663587  1000
   11729  1000
```

```r
if (!require(ggplot2)){
  install.packages("ggplot2")
  library(ggplot2)
}
autoplot(mbm) # gráfico parecido al boxplot
```



8.3 Ejercicios

- En las soluciones de PIEP, **NO** debe haber llamados recursivos de una o más funciones que puedan ser reemplazados por el uso de estructuras iterativas.
- En las soluciones de POA (*array-oriented*), **NO** debe haber uso de estructuras iterativas que puedan ser reemplazadas por el uso de operaciones o funciones básicas de tipos de datos compuestos.
- Para los subprogramas solicitados pueden usar todos los operadores y las funciones que se referencian en Elementos básicos, estructuras de control, creación de subprogramas y tipos de datos compuestos en R (intencionalmente, el operador `^` no fue referenciado), todos los demás operadores y funciones de R los pueden usar para la parte de prueba y comparación de resultados.

Haga un adecuado **ANÁLISIS**, **DISEÑO** e **IMPLEMENTACIÓN EN R** de:

1. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número real (c) y un número entero (k), y que devuelvan la **potencia con exponente entero** c^k.

2. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número real (c) y un vector de números reales asociados a los coeficientes de un polinomio en orden ascendiente ($a_0 =$ `v[1]`, $a_1 =$`v[2]`, ..., $a_{n-1} =$`v[n]`

y $p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$), y que devuelvan el **polinomio evaluado** en el número real recibido ($p(c)$).

3. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban dos números enteros no negativos (k y $r \leq k$), y que devuelvan la cantidad de reordenamientos posibles de r elementos que no se repiten tomados de un conjunto de k elementos (número de **permutaciones** o variaciones sin repetición, por ejemplo ver: Combinatoria - Wikipedia). Los subprogramas solicitados deben tener en cuenta que:

   - El número de permutaciones sin elementos repetidos es

   $$P(k, r) = kPr = \frac{k!}{(k-r)!}$$

   - La solución debe computacionalmente mejor, es decir, debe tener un número total de operaciones menor, que simplemente hacer `Factorial(k) / Factorial(k-r)` para una función `Factorial()` que devuelva el factorial de un número entero no negativo.

4. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciba dos números enteros no negativos (k y $r \leq k$), y que devuelvan la cantidad de subconjuntos posibles de r elementos tomados de un conjunto de k elementos (número de **combinaciones** sin repetición, por ejemplo ver: Combinatoria - Wikipedia). Los subprogramas solicitados deben tener en cuenta que:

   - El número de combinaciones (coeficiente binomial) es

   $$C(k, r) = kCr = \binom{k}{r} = \frac{k!}{r!(k-r)!}$$

   - La solución debe ser computacionalmente mejor, es decir, debe tener un número total de operaciones menor, que simplemente hacer `Factorial(k) / ( Factorial(r) * Factorial(k-r) )` o `Permutacion(k,r) / Factorial(r)` para una función `Factorial()` que devuelva el factorial de un número entero no negativo y una función `Permutacion(,)` que devuelva el número de permutaciones.

5. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número real (c) y que devuelvan una aproximación del **seno** de c ($\sin(c)$). Los subprogramas solicitados deben tener en cuenta que:

   - Si $c < 0$, entonces

   $$\sin(c) = -\sin(-c),$$

   lo que reduce el problema a calcular el seno de un valor mayor que cero (0).

- Si $c \geq 2\pi$, entonces
$$\sin(c) = \sin(c - k\,2\pi),$$
lo que reduce el problema a calcular el seno de un valor entre 0 y 2π.
- Si $\pi \leq c < 2\pi$, entonces
$$\sin(c) = -\sin(c - \pi),$$
lo que reduce el problema a calcular el seno de un valor entre 0 y π.
- Si $\frac{\pi}{2} \leq c < \pi$, entonces
$$\sin(c) = \sin(\pi - c),$$
lo que reduce el problema a calcular el seno de un valor entre 0 y $\frac{\pi}{2}$.
- Si $\frac{\pi}{4} \leq c < \frac{\pi}{2}$, entonces
$$\sin(c) = \cos\left(\frac{\pi}{2} - c\right),$$
lo que reduce el problema a calcular el coseno de un valor entre 0 y $\frac{\pi}{4}$.
-
$$\sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} c^{2k+1} \xrightarrow[n\to\infty]{} \sin(c),$$
que se debe utilizar unicamente para calcular el seno de un valor entre 0 y $\frac{\pi}{4}$.
- En esta ocasión, fije la cantidad de términos a usar de la serie. Es decir, primero analice y determine, a lo sumo, cuántos términos son mayores que 1×10^{-15} para c entre 0 y $\frac{\pi}{4}$, y luego en la implementación use siempre esa misma cantidad de términos.

6. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número real (c) y que devuelvan una aproximación del **arco tangente** de c ($\arctan(c)$). Puede tomar $\sqrt{3} \approx 1.7320508075688773$. El subprograma solicitado debe tener en cuenta que:

   - Si $c < 0$, entonces
   $$\arctan(c) = -\arctan(-c),$$
   lo que reduce el problema a calcular el arco tangente de un valor mayor que cero.
   - Si $c > 1$, entonces
   $$\arctan(c) = \frac{\pi}{2} - \arctan\left(\frac{1}{c}\right),$$
   lo que reduce el problema a calcular el arco tangente de un valor menor o igual que uno.
   - Si $c > 2 - \sqrt{3}$, entonces
   $$\arctan(c) = \frac{\pi}{6} + \arctan\left(\frac{\sqrt{3}\,c - 1}{\sqrt{3} + c}\right),$$
   lo que reduce el problema a calcular el arco tangente de un valor menor o igual que $2 - \sqrt{3}$.

- $$\sum_{k=0}^{n} \frac{(-1)^k}{2k+1} c^{2k+1} \xrightarrow[n\to\infty]{} \arctan(c),$$

  que se debe utilizar unicamente para calcular el arco tangente de un valor entre 0 y $2-\sqrt{3}$.
- En esta ocasión, fije la cantidad de términos a usar de la serie. Es decir, primero analice y determine, a lo sumo, cuántos términos son mayores que 1×10^{-15} para c entre 0 y $2-\sqrt{3}$, y luego en la implementación use siempre esa misma cantidad de términos.

7. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número real (c) y que devuelvan una aproximación de la **función exponencial** evaluada en c ($\exp(c)$). Puede tomar e $\approx 2.7182818284590452$. El subprograma solicitado debe tener en cuenta que:

   - Si $c < 0$, entonces
     $$\exp(c) = \frac{1}{\exp(-c)}$$
     lo que reduce el problema a calcular la función exponencial de un valor mayor que cero.
   - Si $c > 1$, entonces,
     $$\exp(c) = \exp(k+d) = \exp(k)\exp(d)$$
     en donde k es un entero positivo y d es un número real entre cero y uno, lo que reduce el problema a calcular la función exponencial de un valor entero positivo y un valor entre cero y uno.
   - Si c es un entero positivo, entonces,
     $$\exp(c) = \mathrm{e}^c = \underbrace{\mathrm{e} \ldots \mathrm{e}}_{c \text{ veces}}$$
     (utilice su subprograma que calcula la potencia con exponente entero de un número real)
   - $$\sum_{k=0}^{n} \frac{c^k}{k!} \xrightarrow[n\to\infty]{} \exp(c)$$
     que se debe utilizar unicamente para calcular la función exponencial de un valor entre cero y uno.
   - En esta ocasión, fije la cantidad de términos a usar de la serie. Es decir, primero analice y determine, a lo sumo, cuántos términos son mayores que 1×10^{-15} para c entre 0 y 1, y luego en la implementación use siempre esa misma cantidad de términos.

8. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un número real (c) y que devuelvan una aproximación de la **función logaritmo natural** evaluada en c ($\ln(c)$). Puede tomar $\ln(2) \approx 0.6931471805599453$. El subprograma solicitado debe tener en cuenta que:

   - Si $0 < c < 1$, entonces
   $$\ln(c) = -\ln\left(\frac{1}{c}\right),$$
   lo que reduce el problema a calcular el logaritmo natural de un valor mayor que uno.
   - Si $c > 1$, entonces
   $$\ln(c) = \ln\left((d)\left(2^k\right)\right) = \ln(d) + k\ln(2),$$
   en donde k un entero no negativo y d es un número real entre uno y dos, lo que reduce el problema a calcular el logaritmo natural de un valor entre uno y dos.
   -
   $$2\sum_{k=0}^{n} \frac{1}{2k+1}\left(\frac{c-1}{c+1}\right)^{2k+1} \xrightarrow[n\to\infty]{} \ln(c)$$
   que se debe utilizar unicamente para calcular el logaritmo natural de un valor entre uno y dos.
   - En esta ocasión, fije la cantidad de términos a usar de la serie. Es decir, primero analice y determine, a lo sumo, cuántos términos son mayores que 1×10^{-15} para c entre 1 y 2, y luego en la implementación use siempre esa misma cantidad de términos.

9. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que devuelvan un **valor cercano al número** π, usando la generación de valores seudoaleatorios / por **Monte Carlo**. Para el subprograma solicitado pueden usar la función `runif()` de R.

10. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que devuelvan un **valor cercano a la integral definida** entre dos números reales dados, de una función de los reales en los reales dada (integrable y bien definida en el intervalo de integración), usando la generación de valores seudoaleatorios / por **Monte Carlo** (Por ejemplo, consultar Integración de Monte Carlo - Wikipedia). Para el subprograma solicitado pueden usar la función `runif()` de R.

11. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que devuelvan una **aproximación a la integral definida** entre dos números reales dados, de una función de los reales en los reales dada (integrable y bien definida en el intervalo de integración), usando el método o regla del **trapecio compuesta** con todas sus consideraciones (Por ejemplo, consultar Regla del trapecio compuesta. Regla del trapecio - Wikipedia).

12. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que devuelvan una **aproximación a la integral definida** entre dos números reales dados, de una función de los reales en los reales dada (integrable y bien definida en el intervalo de integración), usando el método o regla de **Simpson 1/3 compuesta** con todas sus consideraciones (Por ejemplo, consultar Regla de Simpson 1/3 compuesta. Regla de Simpson - Wikipedia).

13. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que devuelvan una **aproximación a la integral definida** entre dos números reales dados, de una función de los reales en los reales dada (integrable y bien definida en el intervalo de integración), usando el método o regla de **Simpson 3/8 compuesta** con todas sus consideraciones (Por ejemplo, consultar Regla de Simpson 3/8 compuesta. Regla de Simpson - Wikipedia).

14. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban un vector de medias y una matriz de covarianzas, y devuelvan una matriz con n filas asociadas a n valores seudoaleatorios provenientes de una vector aleatorio normal multivariado con el vector de medias y la matriz de covarianzas recibidos. Para el subprograma solicitado pueden usar las funciones `rnorm()` y `chol()` de R.

15. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban una matriz de datos cuantitativos y que devuelvan una lista con: (1) una matriz que tenga los primeros cuatro (r = 1, 2, 3 y 4) momentos no centrales, (2) una matriz que tenga los primeros cuatro momentos centrales, y (3) una matriz que tenga los primeros cuatro momentos estándarizados, de todos y cada uno de los vectores columna de la matriz recibida. Tenga en cuenta que:

    Momentos ordinarios o no centrales (población finita de tamaño N):

    $$\gamma'_r = E\left[X^r\right] = \frac{1}{N}\sum_{i=1}^{N} X_i^r$$

    Note que γ'_1 es la media o valor esperado μ.

    Momentos centrales:

    $$\gamma_r = E\left[\left(X - \gamma'_1\right)^r\right] = E\left[\left(X - \mu\right)^r\right]$$

    Por lo tanto,

$$\gamma_1 = \gamma_1' - \mu = 0$$
$$\gamma_2 = \gamma_2' - \mu^2 = \sigma^2$$
$$\gamma_3 = \gamma_3' - 3\mu\gamma_2' + 2\mu^3$$
$$\gamma_4 = \gamma_4' - 4\mu\gamma_3' + 6\mu^2\gamma_2' - 3\mu^4$$
$$...$$

Note que γ_2 es la varianza σ^2.

Momentos estándar o estandarizados:

$$\gamma_r^* = \frac{\gamma_r}{\sigma^r}$$

Note que $\gamma_1^* = 0$, $\gamma_2^* = 1$, γ_3^* es el coeficiente de asimetría (Pearson) y γ_4^* es el coeficiente de apuntamiento o curtosis (curtosis de Pearson). El exceso de curtosis (curtosis de Fisher) sería igual a $\gamma_4^* - 3$.

16. Un subprograma con una solución de **PIEP** y uno con una solución de **POA (*array-oriented*)** que reciban una matriz de datos cuantitativos y que devuelvan una matriz con todos y cada uno de los vectores columna estandarizados de la matriz recibida.

    Si las columnas de una matriz $N \times p$ corresponden a las variables cuantitativas $V_1, V_2, ..., V_p$, entonces una matriz con los vectores columna estandarizados sería una matriz $N \times p$ en donde la columna i-ésima sería la variable cuantitativa $Z_i = \frac{V_i - \mu_i}{\sigma_i}$, donde μ_i es la media de V_i y σ_i es la desviación estándar de V_i (asumamos que los datos que tenemos son poblacionales).

Parte IV

Clases y Objetos (CyO)

9 Introducción CyO

En esta sección se hará una revisión de lo que son clases y objetos (CyO), a manera de una pequeña introducción a la programación bajo el paradigma orientado a objetos.

En esta revisión se hace mención a temas de clases y objetos relacionados con encapsulamiento, ocultamiento de la información, herencia, polimorfismo y sobrecarga de operadores.

Se espera que al finalizar las actividades de esta sección, el estudiante tenga claras las principales características de clases y objetos, ya sea para su uso dentro del paradigma de programación orientada a objetos (POO), o como un elemento adicional que se puede utilizar dentro del paradigma de programación imperativa estructurada procedimental (PIEP).

9.1 Mecanismos de abstracción

Los primeros programadores de software han utilizado diferentes métodos de abstracción antes de llegar al paradigma de la programación orientada a objetos. Desde una perspectiva histórica, el uso de la abstracción por la programación orientada objetos no es más que la progresión natural de la abstracción luego de ir desde funciones hasta tipos abstractos de datos.

9.1.1 Funciones y procedimientos

Las funciones y procedimientos son los mecanismos de abstracción más antiguos y más ampliamente utilizados por los programas. Las **funciones** permiten que determinadas tareas se puedan utilizar en muchos sitios, incluso en diferentes aplicaciones, se recogen en un lugar y se reutilizan. Los **procedimientos** permiten a los programadores organizar tareas repetitivas de una sola vez. Estas dos abstracciones evitan duplicaciones de código.

Las funciones y procedimientos permiten a los programadores la capacidad de implementar el **ocultamiento de la información**. Un programador escribe una función o conjunto de funciones que se utilizarán por muchos otros programadores. Otros programadores que no necesitan conocer los detalles exactos de la implementación, sólo necesitan conocer la **interfaz** (es decir, lo que debe entrar y lo que debe salir del subprograma respectivo). Desgraciadamente, las funciones no llegan a ser el mecanismo más eficiente para un completo ocultamiento de información.

9.1.2 Tipos de datos abstractos

Un tipo abstracto de datos es un **tipo de dato definido por el programador** que se puede manipular de un modo similar a un **tipo de dato predefinido o preexistente**. Los programadores pueden crear instancias de tipos abstractos de datos asignando valores válidos a las variables de dichos tipos. Además se pueden utilizar las funciones para manipular los valores asignados a las variables.

En síntesis, los tipos datos abstractos permiten las siguientes tareas:

1. Extender un lenguaje de programación añadiendo tipos de datos definidos por el usuario.
2. Poner a disposición de otro código, un conjunto de funciones definidas por el programador, que se utilizan para manipular los valores de las variables de los tipos definidos por el programador.
3. Proteger (ocultar) los datos asociados a las instancias con el tipo y limitar el acceso a los datos, haciendo que sólo por las funciones definidas por el programador puedan acceder a los datos.
4. Hacer tantas distancias como se desee del tipo de dato definido por el programador.

9.2 Modelado del mundo

Uno de los problemas (y tal vez el más importante) del **paradigma procedimental** es que la disposición independiente de datos y funciones realiza un pobre modelado de las cosas tangibles e intangibles del mundo real. En el mundo físico, se interactua con **objetos** tales como personas y autos. Dichos objetos no son ni como los datos ni como las funciones. Los objetos complejos del mundo real tienen **atributos**, **comportamiento** e **identidad**.

Un objeto no sólo encapsula su estado (variables) sino también su comportamiento. El comportamiento de un objeto se describe en términos de servicios (operaciones) proporcionados por ese objeto que modifican o inspeccionan el estado del objeto. Estos servicios se invocan enviando mensajes, del objeto que solicita el servicio, al objeto sobre el que se envía el mensaje. Normalmente el único medio para acceder a un objeto es a través de las operaciones. Estas operaciones, por consiguiente, actúan como una interfaz al objeto.

9.2.1 Atributos

Los **atributos** son las propiedades de los objetos (a veces denominados características). Por ejemplo, en el caso de personas, el color de los ojos, el título de un empleado, y en el caso de autos, la potencia, el número de puertas o el modelo. A su vez los atributos del mundo real son equivalentes a los datos de un programa, tienen un cierto valor específico, tal como azul (color de lo ojos), 150 CV (potencia en caballos de vapor en automóviles) o cinco (para el número de puertas).

Los atributos representan la identificación y propiedades descriptivas. A nivel de lenguaje de programación los **objetos** que tienen el mismo conjunto de **atributos** se dice pertenecen a la misma **clase**.

El **estado** de un objeto agrupa los valores instantáneos de todos los atributos del mismo, donde un atributo es una información que cualifica al objeto. Cada atributo puede tomar un valor en un ámbito dado. El estado de un objeto, en un instante dado, corresponde a una selección de valores, entre todos los valores posibles para cada atributo. El estado del objeto puede cambiar con el tiempo.

Por ejemplo, un auto perteneciente a la clase `Auto` tiene los atributos: `Marca`, `color`, `peso` y `potencia`.

Un objeto de la clase `Auto` tendría ciertos valores para dichos atributos:

- `Audi`
- `Azul cielo`
- `1.100 Kg`
- `150 CV`

El estado del auto es variable, es decir, los valores de los atributos podrían ser unos en un momento y otros en otro momento. Algunos componentes son constantes (marca, país de su fabricación, etc.). La cantidad de litros en el tanque de gasolina varía a medida que se usa el auto; el color puede cambiar si su dueño decide pintarlo con un color distinto al de fábrica.

9.2.2 Comportamiento

El comportamiento es algo que un objeto del mundo real hace en respuesta a cualquier estímulo. Por ejemplo, si usted solicita a su empresa un aumento de sueldo, normalmente le contestarán si o no. Si se aplica una acción sobre los frenos de un coche, normalmente, se detendrá. "Aumentar el sueldo" y "detener" son ejemplos de comportamiento. El **comportamiento** es como una **función**, una función es llamada para que haga al y ese algo se hace.

El comportamiento agrupa todas las competencias de un objeto y describe las **acciones** y **reacciones** de ese objeto. Cada elemento de comportamiento se denomina **operación**. Las operaciones de un objeto se ejecutan a consecuencia de un estímulo externo, representado en forma de un **mensaje** enviado por sí mismo o por otro objeto.

9.2.3 Identidad

La **identidad** es una propiedad fijada por la cual se identifica a un objeto de otro. Si tiene dos tazas de café del mismo juego, se dice que son iguales pero no idénticas. Ambas son de igual tamaño, color, forma, material, están vacías (o llenas), etc, es decir, son iguales, pero no son idénticas, ya que usted puede elegir para tomar su café una u otra.

Un objeto posee una identidad que caracteriza su propia existencia. La identidad permite distinguir los objetos de una manera no ambigua, independientemente de su estado. Esto permite distinguir dos objetos en los que todos los valores de sus atributos son idénticos.

9.2.4 Paso de mensajes

Una acción se inicia por una petición de un servicio (mensaje) que se envía a un objeto específico. El paradigma de **programación imperativo estructurado procedimental** presta especial importancia a las **funciones**, mientras que el paradigma **orientado a objetos** proporciona especial importancia al objeto. Por ejemplo, se puede llamar a una función para que ponga un dato en un tipo de dato "cola" o bien transmitir un mensaje al objeto "cola" para que realice la acción de recibir o poner un valor en sí misma. El resultado puede llegar a ser el mismo pero el proceso para llegar a dicho resultado no es el mismo.

El paso de mensajes proporciona la capacidad de sobrecargar nombres y reutilizar software; esto no es posible utilizando el paradigmo imperativo. Implícito a la idea de mensaje es la idea de que la interpretación de un mensaje puede variar para objetos de diferentes clases. Es decir, el comportamiento dependerá de la clase de objeto que recibe el mensaje. El mensaje/servicio "poner" puede significar una cosa en objetos de la clase "cola" y otra muy distinta en cualquier otro objeto.

Con frecuencia, hay servicios/mensajes para solicitar cierta información de un objeto. Esta información puede ser relativa al estado de la instancia del objeto, pero también puede implicar un cálculo de algún tipo.

9.3 El enfoque orientado a objetos

La idea fundamental que subyace en los lenguajes orientados a objetos es combinar en una única unidad tanto datos como funciones que operan sobre esos datos, tal unidad se denomina *objeto*. Un objeto es una abstracción de algo en un dominio del problema a resolver.

El mecanismo/concepto fundamental de la programación orientada a objetos es el **objeto**. Un objeto consta de los **atributos** (**datos**) y los **métodos** (**subalgoritmos**) que actúan sobre los datos. Los datos no deben ser accesibles directamente a los usuarios del objeto. El acceso a los datos se debe garantizar únicamente por los métodos proporcionados por el objeto. La orientación a objetos se denomina así porque este método ve las cosas que son parte del mundo real como objetos. Un teléfono es un objeto; la silla en que está sentado es un objeto; el libro que está leyendo es un objeto, etc. De igual forma son objetos, una bicicleta, una póliza de seguros, etc. Los objetos individuales de una clase se llaman instancias de esa clase. Por ejemplo, Clase: `Mesa`. Instancias: `Mi mesa`, `Tu mesa`, etc. Estas instancias tienen los mismos atributos y posiblemente diferentes valores (es decir, con un estado diferente).

Los objetos que tienen el mismo conjunto de atributos se dice que pertenecen a la misma **clase**. Los objetos individuales de una clase se llaman **instancias** de esa clase. Las instancias tienen los mismos atributos pero con valores posiblemente diferentes (es decir, con estados posiblemente distintos).

En el paradigma de programación orientado a objetos se **encapsula** datos (atributos) y métodos (comportamiento) en objetos. Los datos y métodos de un objeto se conectan juntos. Los objetos tienen la propiedad del **ocultamiento de la información**. Esto significa que aunque los objetos pueden conocer como comunicarse con otros objetos a través de interfaces bien definidas, los objetos normalmente no están autorizados a conocer como se implementan otros objetos (los detalles de implementación se ocultan dentro de los propios objetos). Se puede conducir un auto eficientemente sin conocer los detalles de cómo funciona internamente el sistema de transmisión, los frenos o el motor.

Los programadores bajo el paradigma imperativo estructurado procedimental se centran en escribir funciones. Los grupos de acciones que realizan algunas tareas se forman en funciones y las funciones se agrupan para formar programas. Los datos son muy importantes como soporte a las acciones que se ejecutan.

Los programadores orientados a objetos se centran en crear sus "propios tipos de variables" denominados **clases**. Cada clase declara o especifica los datos, así como el conjunto de métodos que manipulan los datos. Igual que se llama variable a una instancia de un tipo predefinido (por ejemplo, `int`), una instancia de una clase se denomina objeto. Las clases se utilizan entonces para instanciar objetos que al trabajar juntos como un sistema resolverán el problema de interés.

Los métodos (funciones) de un objeto proporcionan la única manera para acceder a sus datos. Si se desean leer los datos de un objeto, se llama a una función miembro del objeto. **No se debería poder acceder a los datos directamente**. Los datos están ocultos, así están protegidos de modificaciones accidentales. Los datos y las funciones se dice que están encapsulados en una única entidad. Si se desea modificar los datos de un objeto, se conoce exactamente cuales métodos interactúan con dichos datos. Ninguna otra función puede acceder a los datos. Esto simplifica la escritura, depuración y mantenimiento del programa.

Listados con objetos del mundo real que se pueden corresponden con objetos en programación orientada a objetos:

Objetos físicos

- Automóviles en un sistema de simulación de tráficos.
- Camiones en una empresa transportista.
- Componentes eléctricos en un programa de diseño de circuitos.
- Países en un modelo de comercio.
- Aviones en un sistema de tráfico aéreo.

Elementos de un sistema informático

- Unidad central, teclado, impresora, unidad de discos, unidad de DVDCDRW, etc.
- Menús.
- Ventanas.
- Objetos gráficos (líneas, rectángulos, circulos, etc.).

Estructuras de datos

- Pilas.
- Colas.
- Listas enlazadas.
- Árboles binarios.
- Árboles binarios de búsqueda.
- Dobles colas.

Tipos de datos definidos por el usuario

- El tiempo en formato horario.
- Números complejos.
- Puntos de un plano.
- Ángulos de un sistema.
- Fecha de un día.

Recursos Humanos

- Empleados.
- Clientes.
- Proveedores.
- Socios.
- Vendedores.

9.4 Clases

Casi todos los lenguajes de programación tienen tipos de datos incorporados. Por ejemplo, un tipo de dato entero (`int`) está predefinido en varios lenguajes, y si así se desea, se pueden declarar tantas variables de tipo `int` como sea necesario en su programa:

- `int día`
- `int mes`
- `int divisor`
- `int salario`

De modo similar se pueden definir muchos objetos de la misma clase. En una clase se especifica qué datos (atributos) y funciones (métodos) tendrán los objetos de esa clase. La definición de la clase no crea objetos, al igual que la simple existencia de tipos de datos no crea ninguna variable. Una clase es por consiguiente una descripción de un número determinado de objetos similares. `Ricky Martin`, `Chayanne` y `Shakira` son miembros de la clase "`cantante latino`". No hay ninguna persona llamada "`cantante latino`" sin embargo hay personas específicas con nombres específicos que son miembros de esta clase si poseen unas características determinadas.

9.4.1 Identificación y responsabilidad de una clase

Cinco grandes categorías facilitan la identificación de qué tipos de cosas son clases:

Cosas tangibles

Avión	Fuente de alimentación	Libro	Auto
Reactor nuclear	Circuito de frenos	Perro	Moto
Caballo de carreras	Parque nacional	Gato	Banco
Curso	Grupo de clase	Fotocopia	Barco

Roles jugados por personas o instituciones

Doctor	Empleado	Supervisor
Paciente	Gerente	Jefe Departamento
Enfermero	Cliente	Ingeniero de Sistemas
Director	Socio	Analista

Incidentes/Eventos

Vuelo	Accidente	Rendimiento
Suceso	Rotura de un sistema	Llamada telefónica
Salida de un tren	Despegue de un avión	Llegada de un avión

Interacciones

Compra	Arco (entre dos nodos)
Cargo en tarjeta de crédito	Reunión
Intersección	Contrato

Especificaciones

Producto de seguros	Artículo de un libro
Tipo de tarjeta de crédito	Tipo de crédito

Cada clase de un sistema debe ser responsable de un aspecto del mismo. Las propiedades localizadas en una misma área de responsabilidad se deben agrupar en una única clase y no dividirse en diversas clases. Cada responsabilidad se asigna a una única clase. El principio de coherencia exige que todas las propiedades de una clase deben formar una conexión lógica. Si por ejemplo se desea crear una clase *Cliente* se debe determinar primero su responsabilidad: ¿sobre qué elementos es responsable esta clase?. Por ejemplo:

9.4.2 Representación gráfica de una clase

Una clase se representa en UML (*Unified Modeling Language*) con un rectángulo que se divide horizontalmente en tres bandas. La banda superior contiene el nombre de la clase; la banda central los atributos de la clase; y la banda inferior contiene las operaciones/métodos de la clase. Los atributos con un signo más (+) delante de ellos son públicos, que significa que otras clases pueden acceder a ellos. Los atributos precedidos con un signo menos (-) son privados, lo que indica que sólo la clase y sus objetos pueden acceder a ellos. Por último, los atributos protegidos rotulados con el número de signo numeral (#) pueden ser utilizados únicamente por la clase y por cualquier descendiente de la clase (en herencia hablaremos de clases descendientes).

9.4.2.1 Atributos

Un atributo es una propiedad de una clase. Describe el rango de valores que la propiedad puede tener en los objetos de esa clase. Una clase puede tener cero o más atributos. Los atributos pueden mostrar su tipo así como su valor por defecto.

9.4.2.2 Operaciones/Métodos

Una operación es algo que una clase puede hacer. Se nombra de igual forma que los atributos.

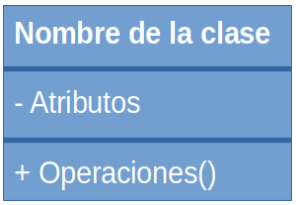
9.4.2.3 Ejemplos

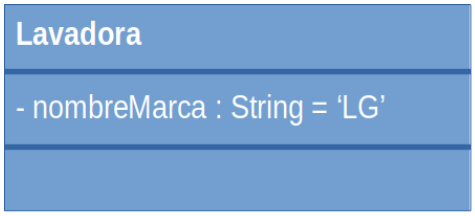
- Clase `Motocicleta` (atributos y operaciones/métodos).
- Clase `Factura` (atributos).
- Clase `Cuenta Corriente` (atributos y operaciones/métodos).

Cliente
• Administrar todos los datos personales de un cliente • Administrar direcciones, detalles de telecomunicaciones y cuentas de bancos

Direcciones
• Administrar y representar una dirección postal • Comprar la dirección frente a las tablas existentes y clases y códigos postales, insofar como esto es responsable o útil

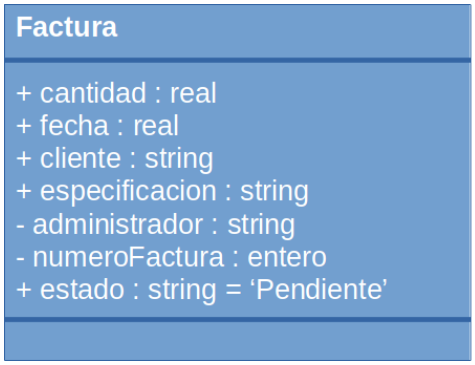
CuentaBanco
• Administra y representa una cuenta en una institución financiera • En el caso de las cuentas domésticas del banco, comprueba el código de ordenación frente a una tabla existente de códigos de ordenación

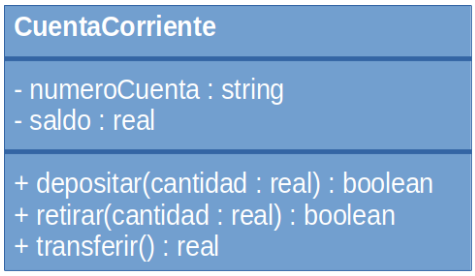



Lavadora
- nombreMarca : String = ‘LG’


Lavadora
- nombreMarca : String = ‘LG’
+ encender()
+ apagar()
+ lavar()
+ aclarar()
+ secar()


Motocicleta
- marca : String
- color : String
- cilindraje : real
- velocidadMaxima : real
+ arrancar()
+ acelerar()
+ frenar()


Factura
+ cantidad : real
+ fecha : real
+ cliente : string
+ especificacion : string
- administrador : string
- numeroFactura : entero
+ estado : string = ‘Pendiente’



9.4.3 Declaración de clases

En pseudocódigo, se podría hacer la declaración de clases de la siguiente manera:

clase nombre_de_clase

//Declaración de atributos

//Declaración de operaciones (métodos y constructores)

fin_clase

El nombre/etiqueta `nombre_de_clase` tiene que ser un identificador válido.

9.4.3.1 Los miembros de una clase y de un objeto

Los miembros de una clase son los **atributos** (o variables de instancia) y los **métodos**. Los métodos son acciones que se realizan por un objeto de una clase. Una invocación a un método es una petición que le llega al método para que ejecute su acción dentro del objeto al que pertenece. La invocación de un método se denominaría también *llamar a un método* o *pasar un mensaje a un objeto*.

Existen dos tipos de métodos, aquellos que devuelven un valor único y aquellos que ejecutan alguna acción distinta de devolver un único valor. La diferencia entre unos y otros en pseudocódigo se destacará mediante el empleo de las palabras reservadas `procedimiento` y `funcion`. El paso de parámetros se regirá por las normas habituales descritas al tratar procedimientos y funciones. Por ejemplo, en una clase `CuentaCorriente`, el método `depositar`, que no devuelve ningún valor, se declararía:

público procedimiento depositar(**real** cantidad)
...
inicio
...
fin_procedimiento

El método `obtenerSaldo`, también de la clase `CuentaCorriente`, se supone devuelve el saldo y su declaración sería:

público real función obtenerSaldo()
  …
inicio
  …
  devolver(…)
fin_función

9.4.3.2 Método constructor

Un **constructor** es un método que tiene el mismo nombre que la clase, cuyo propósito es inicializar los miembros datos (atributos) de un nuevo objeto y que se ejecuta automáticamente cuando se crea un objeto de una clase. Sintácticamente es similar a un método. Dependiendo del número y tipos de los argumentos proporcionados, una función o método constructor se llama automáticamente cada vez que se crea un objeto. Si no se ha definido ningún constructor en la clase, el compilador proporciona un constructor por defecto. Cuando se define un constructor no se puede especificar un valor de retorno, un constructor nunca devuelve un valor. Un constructor puede, sin embargo, tomar cualquier número de parámetros (cero o más). A su rol como "inicializador", un constructor puede tener otras tareas, que se ejecutarán cada vez que se cree un objeto de la clase. En pseudocódigo, la definición de un método constructor se podría hacer de la siguiente manera:

constructor nombre_de_clase[(lista_parámetros_formales)]
//declaración de variables locales
inicio
  //código del constructor
fin_constructor

Por ahora asumiremos que cuando un objeto ya no se necesita y se queda sin referencias, la memoria ocupada por ese objeto se libera automáticamente, sin necesidad de realizar una destrucción explícita del objeto (no se requerirá un método destructor).

9.4.3.3 Ejemplo

Declaración de una clase `auto` asociada al objeto/entidad/concepto: automóvil.

clase auto

  var
    privado entero: FechaDeFabricación
    privado real: Filometraje

```
  privado cadena: LicenciaDeRodamiento

 público constructor auto()
 inicio
  ...
 fin_constructor

 público entero función obtenerFechaFabricación()
  ...
 inicio
  ...
  devolver(...)
 fin_función

 público procedimiento obtenerKilometraje()
  ...
 inicio
  ...
 fin_procedimiento

  ...

fin_clase
```

9.5 Objetos

9.5.1 Representación gráfica de un objeto

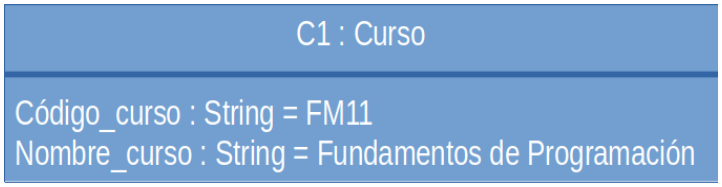
La notación UML de un objeto es un rectángulo con dos compartimentos. El compartimento superior contiene el nombre del objeto y el nombre de la clase a la cual pertenece ese objeto. La sintaxis sería:

nombreobjeto: **nombreclase**

El compartimiento inferior contiene la lista de nombres de atributos y sus valores. Los tipos de atributos se pueden mostrar utilizando la sintaxis:

nombreatributo: **tipo** = valor

Por ejemplo, el objeto `C1` de la clase `Curso` con dos atributos podría representarse así:



9.5.2 Instanciación de objetos

La siguiente sentencia define dos objetos, `obj1` y `obj2`, de la clase `NombreClase`.

NombreClase : obj1, obj2

La definición de un objeto es similar o equivalente a la definición de una variable. Se reserva espacio en memoria para el objeto con todos sus atributos. La creación física del objeto se denomina **instanciación**. La instanciación se efectúa mediante la palabra `nuevo`, empleada por ejemplo en una sentencia de asignación de la siguiente forma,

//llamada al constructor por defecto (que en este caso no tiene parámetros)
obj1 <- **nuevo** NombreClase()

Para acceder a los elementos de un objeto (atributos y métodos) se emplea el operador punto: ..

nombre_objeto.nombre_atributo
nombre_objeto.nombre_método(valores_para_los_parámetros_del_metodo)

La llamada o invocación a un método se puede realizar de dos formas, dependiendo de que el método devuelva o no un valor.

1. Si el método devuelve un valor, la llamada al método se trata normalmente como un valor.
2. Si el método realiza una acción distinta a devolver un valor, una llamada al método debe ser una sentencia/instrucción.

Así dada la declaración: `CuentaCorriente: miCuenta`, las llamada a los métodos `depositar` y `obtenerSaldo` se podrían efectuar de la siguiente forma:

miCuenta.depositar(200000)
saldo <- miCuenta.obtenerSaldo()

9.6 Encapsulamiento y visibilidad (ocultamiento)

Una característica de los objetos es que mantienen unidos o encapsulados sus miembros, es decir sus características (atributos) y su comportamiento (métodos). Además, un principio básico en programación orientada a objetos es el **ocultamiento de la información**. El ocultamiento

de la información significa que a uno o más miembros de un objeto, no se pueden acceder por los métodos de otros objetos (y en general, no se pueden acceder por funciones externas a la clase). Para controlar el acceso a los miembros de una clase se utilizan tres diferentes **especificadores de acceso**: `público`, `privado` y `protegido`.

El mecanismo principal para ocultar datos es ponerlos en una clase y hacerlos privados. A los atributos o métodos **privados** sólo se puede acceder desde dentro de la clase. Por el contrario los atributos o métodos **públicos** son accesibles desde el exterior de la clase. Los miembros **protegidos** son accesibles por métodos de la misma clase y de las clases derivadas, así como por clases amigas. Normalmente una clase debe tener todos o casi todos sus atributos privados y sus métodos públicos. Las reglas de **visibilidad** complementan el concepto de **encapsulamiento** de forma que las estructuras de datos internas utilizadas en la implementación de una clase no pueden ser accesadas directamente. Cuando una clase tiene su estructura interna oculta (es decir, privada) se facilita el mantenimiento del software, ya que se podrá mejorar la estructura de la clase sin tener que modificar los programas que la utilizan. Internamente se cambia lo que se desea pero manteniendo la misma interfaz con el exterior, de esa manera todo se mantiene igual para todo lo externo que interactue con los objetos de esa clase.

9.7 Jerarquía de clases

Generalización (o **especialización**) es una relación taxonómica entre un elemento general y uno especial (o viceversa), donde el elemento especial añade propiedades al general y se comporta de un modo compatible con él.

En la generalización o especialización, las propiedades se estructuran jerárquicamente. Propiedades de significado general se asignan a las clases más generales (**superclases**) y las propiedades más especiales se asignan a clases que están subordinadas a las clases generales (**subclases**). Por consiguiente, las propiedades de las superclases son conferidas/otorgadas (*bestowed*) a las subclases. Una subclase contiene tanto sus propias propiedades como las de sus superclases. Al heredar todas las propiedades de sus superclases, las subclases pueden modificarlas y ampliarlas, pero no pueden eliminarlas ni suprimirlas.

La diferencia entre las superclases y subclases se realiza, normalmente, mediante al menos una característica específica, denominada **discriminador**.

En UML se define a la generalización como herencia. De hecho, **generalización** es el concepto y **herencia** se considera la implementación del concepto en un lenguaje de programación.

9.7.1 Herencia

Es la capacidad para crear nuevas clases (descendientes) que se construyen sobre otras que ya existen. ***Una clase derivada hereda el código (métodos) y los datos (atributos) de las clases base, añadiendo su propio código especial y sus datos adicionales,***

incluso, puede modificar aquellos elementos de las clases base que necesita sean diferentes. La herencia puede ser simple o múltiple. En **herencia simple**, una clase derivada hereda exactamente de una sola clase base (tiene sólo un padre). La **herencia múltiple** implica múltiples clases bases, es decir, una sola clase derivada tiene varios padres (lo cual tiene como desventaja que puede generar ambigüedades). Para definir clases descendientes se debe especificar en la cabecera de la clase, tras las palabras `hereda_de`, la o las clases ascendientes o antepasado, también denominadas clases base o superclases.

clase nombre_clase **hereda_de** [**especificador_acceso**] clase_base
  // lista_de_miembros
fin_clase

- El `especificador_acceso` que declara el tipo de herencia es opcional (`publico`, `privado` o `protegido`). La accesibilidad de los miembros heredados en la clase derivada vendrá dada por la combinación de los modificadores de los miembros de la clase base con el tipo de herencia. Así, un *especificador de acceso* `publico`, significa que los miembros públicos de la clase base son miembros públicos de la clase derivada. Con herencia `privada` los miembros públicos y protegidos de la clase base se vuelven miembros privados de la clase derivada.
- La clase base (`clase_base`) es el nombre de la clase de la que se deriva la nueva clase (`nombre_clase`).
- La `lista_de_miembros` consta de atributos y métodos que se adicionan a (o en donde se modifican) los miembros heredados.

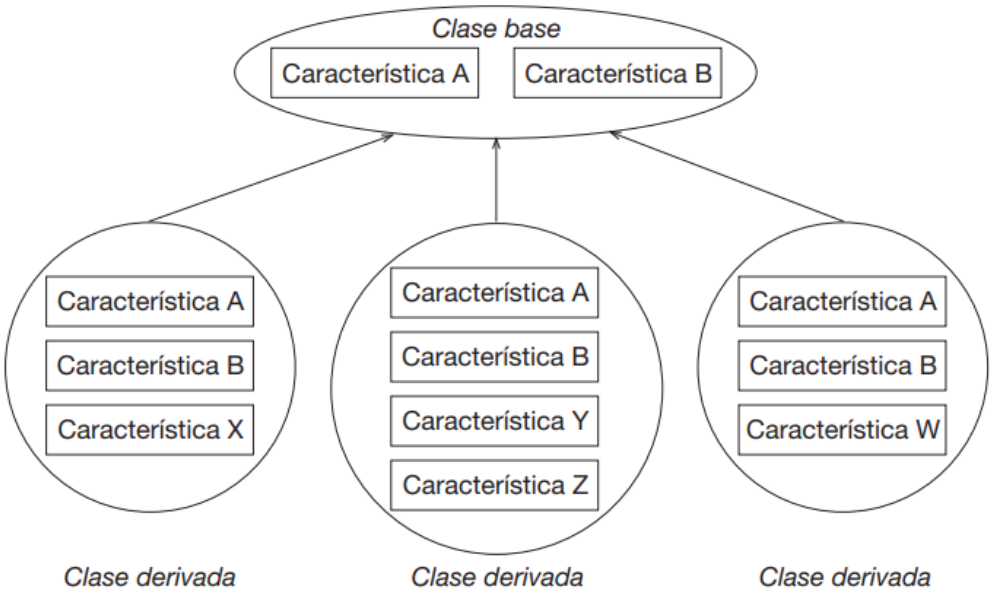
La propiedad de la herencia hace las tareas de programación mucho más fáciles y flexibles, no siendo necesario describir cada una de las características explícitamente para cada clase, ya que las clases pueden heredar características de otras clases.

Como ya se comentó en una sección anterior, **encapsular** es una característica muy potente, y junto con el **ocultamiento de la información**, representan propiedades principales de los objetos. La **orientación a objetos** se caracteriza, además de por las propiedades anteriores, por incorporar la característica de **herencia**, propiedad que permite a los objetos ser construidos a partir de otros objetos. Dicho de otro modo, la capacidad de un objeto para utilizar las estructuras de datos y los métodos previstos en antepasados o ascendientes. Esto permite reutilizar código anteriormente ya desarrollado.

La herencia se apoya en el significado de ese concepto en la vida diaria. Así, las clases básicas o fundamentales se dividen en subclases. Los animales se dividen en mamíferos, anfibios, insectos, pájaros, peces, etc. La clase vehículo se divide en subclase automóvil, motocicleta, camión, autobús, etc. Es así como todos los vehículos citados tienen un motor y ruedas, que son características comunes; si bien los camiones tienen una caja para transportar mercancías, mientras que las motocicletas tienen un manillar en lugar de un volante.

La herencia supone una clase base y una jerarquía de clases. Las clases derivadas heredan atributos y métodos de su clase base, añadiendo sus propios atributos y métodos, e incluso

cambiando aquellos elementos de la clase base que necesita sean diferentes. Las clases derivadas se crean en un proceso de definición de nuevos tipos y reutilización del código anteriormente desarrollado en la definición de sus clases base. Las clases que heredan propiedades de una clase base pueden a su vez servir como definiciones base de otras clases. Las jerarquías de clases se organizan en forma de árbol. La herencia es un mecanismo potente para tratar con la evolución natural de un sistema con modificación incremental.



La relación de herencia se representa mediante una flecha larga que apunta de la subclase a la superclase. Las flechas pueden ser dibujadas de dos formas: directamente de las subclases a las superclases o bien combinadas en una línea común. Como ya se ha comentado, la herencia es la manifestación más clara de la relación de generalización/especialización y a la vez una de las propiedades más importantes de la orientación a objetos. Todos los lenguajes de programación orientados a objetos soportan en su propio lenguaje construcciones que implementan de modo directo la relación jerárquica entre clases. La herencia es la relación que existe entre dos clases, en la que una clase denominada derivada se crea a partir de otra ya existente, denominada clase base. Este concepto nace de la necesidad de construir una nueva clase a partir de una existente. Así, por ejemplo, si existe una clase `figura` y se desea crear una clase `triángulo`, esta clase `triángulo` puede derivarse de `figura` ya que tendrá en común con ella un estado y un comportamiento, aunque `triángulo` luego tendrá sus características propias. `triángulo` "**es-un**" tipo de `Figura`. Otro ejemplo, puede ser `vendedor` que "**es-un**" tipo de `empleado`.

A veces es difícil decidir cuál es la relación de herencia más óptima entre clases dentro del diseño de un programa. Una vista de los empleados basada en el modo de pago puede dividir a los empleados con salario mensual fijo; empleados con pago por horas de trabajo y empleados a comisión por las ventas realizadas. Una vista de los empleados basada en el estado de dedicación a la empresa: dedicación plena o dedicación parcial. Una vista de empleados basada en el estado

laboral del empleado con la empresa: fija o temporal. Una dificultad a la que suele enfrentar el diseñador es que en los casos anteriores un mismo empleado puede pertenecer a diferentes grupos de trabajadores. Una pregunta usual es ¿cuál es la relación de herencia que describe la mayor cantidad de variación en los atributos de las clases y operaciones? ¿esta relación ha de ser el fundamento del diseño de clases? Evidentemente la respuesta adecuada sólo se podrá dar cuando se tenga presente la aplicación real a desarrollar.

Por otra parte es importante tener presente que, **la clase base y la clase derivada tienen código (métodos) y datos (atributos) comunes, de modo que si se decidiera crear la clase derivada totalmente separada, por aparte o independiente de la clase base, se duplicaría mucho de lo que ya se ha escrito para la clase base**.

9.7.1.1 Tipos de herencia

Existen dos mecanismos de herencia utilizados comúnmente en programación orientada a objetos: **herencia simple** y **herencia múltiple**. Herencia simple es aquel tipo de herencia en la cual un objeto (clase) puede tener sólo un ascendiente, o dicho de otro modo, una subclase puede heredar datos y métodos de una única clase. Herencia múltiple es aquel tipo de herencia en la cual una clase puede tener más de un ascendiente inmediato, o lo que es igual, adquirir datos y métodos de más de una clase. No todos los lenguajes de programación soportan herencia múltiple.

A primera vista, se puede suponer que la herencia múltiple es mejor que la herencia simple; sin embargo, como ahora comentaremos, no siempre será así. En general, prácticamente todo lo que se puede hacer con herencia múltiple se puede hacer con herencia simple, aunque a veces resulta más difícil (toca esforzarse un poquito más, lo que no debería tener nada de malo). Una dificultad surge con la herencia múltiple cuando se combinan diferentes tipos de

objetos, cada uno de los cuales define métodos o campos con el mismo nombre. Supongamos dos tipos de objetos pertenecientes a las clases *Gráficos* y *Sonidos*, y se crea un nuevo objeto denominado *Multimedia* a partir de ellos. *Gráficos* tiene tres campos datos: *tamaño*, *color* y *mapasDeBits*, y los métodos *dibujar*, *cargar*, *almacenar* y *escala*; *Sonidos* tiene tres campos dato, *duración*, *voz* y *tono*, y los métodos *reproducir*, *cargar*, *escala* y *almacenar*. Así, para un objeto *Multimedia*, el método *escala* significa poner el sonido en diferentes tonalidades, o bien aumentar/reducir el tamaño de la escala del gráfico. Naturalmente, el problema que se produce es la ambigüedad, y se tendrá que resolver con una operación de prioridad que el correspondiente lenguaje deberá soportar y entender en cada caso (tal vez, una de las razones por las cuales no todos los lenguajes soportan la herencia múltiple. Además, la herencia múltiple es un tema controvertido que tiene simpatizantes y detractores).

9.7.1.2 Clases abstractas

Con frecuencia, cuando se diseña un modelo orientado a objetos es útil introducir clases a cierto nivel que pueden no existir en la realidad (tangible e intangible) pero que pueden ser construcciones útiles. Estas clases se conocen como **clases abstractas**. Las clases abstractas son clases en las que algunos o todos los miembros no tienen implementación. Los métodos sin implementación se pueden declarar especificando exclusivamente la cabecera y no pueden ser privados.

Una clase abstracta normalmente ocupa una posición adecuada en la jerarquía de clases que le permite actuar como un depósito de métodos y atributos compartidos para las subclases de nivel inmediatamente inferior. Se espera que las clases abstractas no tengan instancias directamente. Se utilizan para agrupar otras clases y capturar información que es común al grupo. Las subclases de clases abstractas, es decir las clases derivadas de una clase abstracta, se encargarán de implementar los métodos heredados y naturalmente sí tendrán instancias.

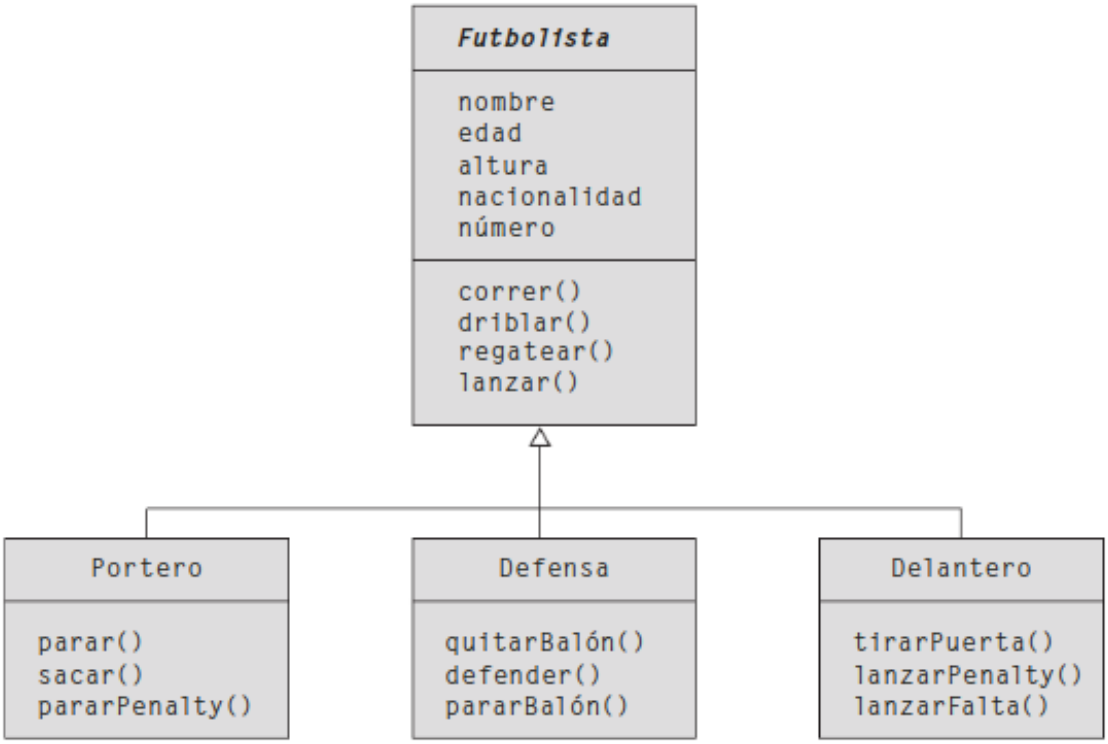
Por ejemplo, una clase abstraca *Vehículo* debe tener atributos (*color*, *año de fabricación*, etc.) y métodos abstractos que especifiquen datos y comportamientos comunes de todos los vehículos (*arrancar*, *frenar*, etc.). *Auto*, *Moto* y *Barco* representan clases que requieren implementar, por ejemplo, el método heredado *frenar*, entre otros.

9.7.1.3 Ejemplo

Clases asociadas a los objetos/entidades/conceptos: jugadores de futbol.

9.8 Polimorfismo

Otra propiedad importante de la programación orientada a objetos es el **polimorfismo**. Polimorfismo puro se produce cuando un único método se puede aplicar a una variedad de tipos o



clases de objetos. En polimorfismo puro hay un método (el cuerpo del código) y un número de interpretaciones (significados diferentes). ***El polimorfismo supone que un mismo mensaje puede producir acciones (resultados) totalmente diferentes cuando se recibe por objetos diferentes***.

El polimorfismo, en su expresión más simple, es el uso de un nombre o un símbolo (por ejemplo, un operador) para representar o significar más de una acción dependiendo de los objetos involucrados. Esta propiedad, en su concepción básica, se encuentra en casi todos los lenguajes de programación, de tal forma que es posible que el símbolo `+` sirva tanto para realizar sumas aritméticas como para concatenar (unir) cadenas. Cuando a un operador existente en el lenguaje, tal como `+`, `=` o `*`, se le asigna la posibilidad de operar sobre un nuevo tipo de dato, se dice que está **sobrecargado**. ***La sobrecarga, que puede ser de operadores y de métodos, es una clase de polimorfismo***.

9.9 Ejercicios

9.9.1 Parte A

Definir y diseñar las clases, junto con sus principales atributos/características y acciones/funcionalidades, que permitan representar los siguientes objetos/entidades/conceptos:

1. Número complejo.
2. Vector en $\mathbb{R}^3$.
3. Círculo, y con las clases derivadas para los conceptos: esfera y cilindro.
4. Polinomio de grado 3 o inferior.
5. Generador de valores seudoaleatorios "igualmente probables" usando el *linear congruential generator*.

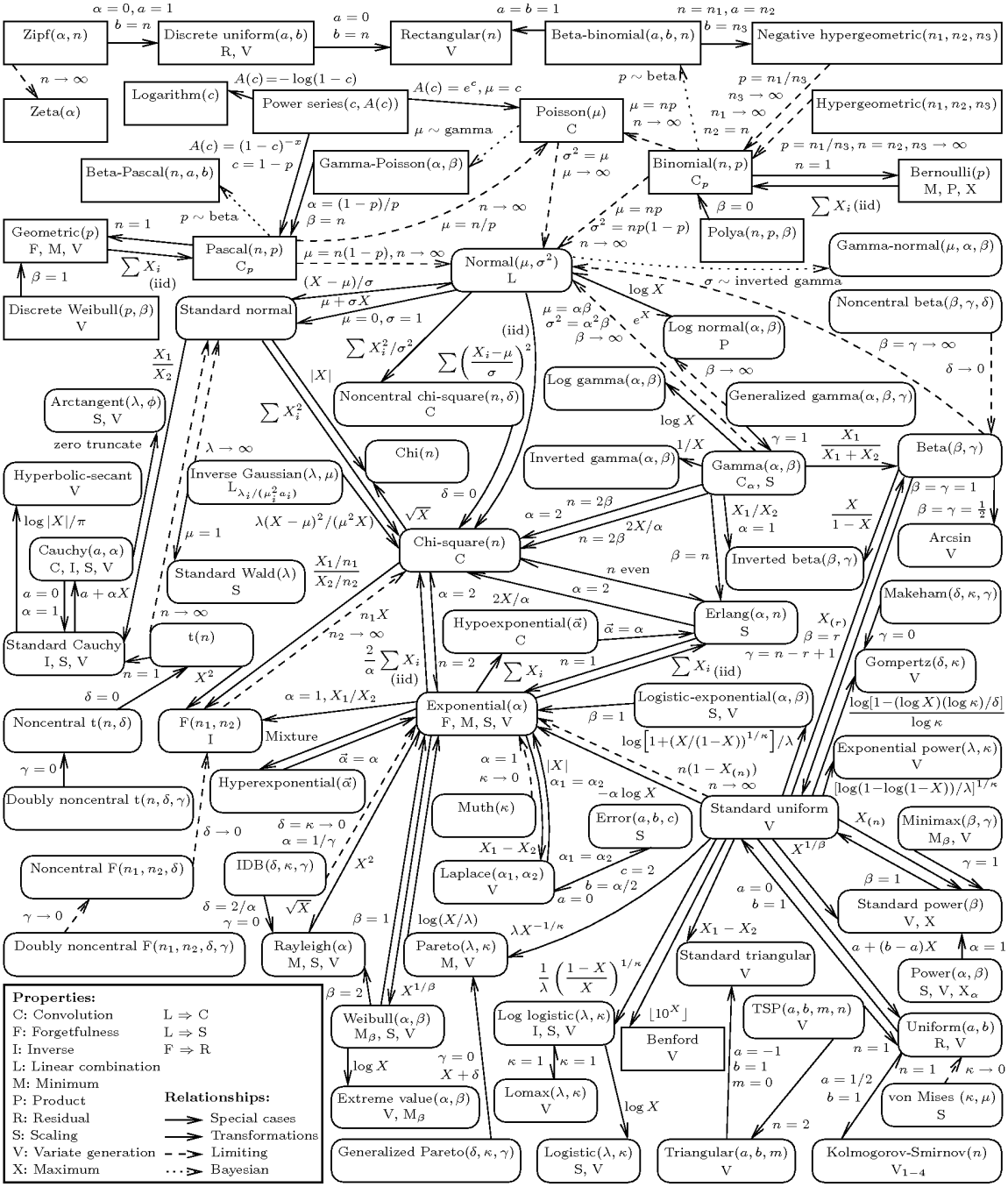
9.9.2 Parte B

Definir y diseñar las clases, junto con sus principales atributos/características y acciones/funcionalidades, que permitan representar las variables aleatorias (que son objetos/entidades/conceptos intangibles):

1. Con distribución Bernoulli.
2. Con distribución binomial.
3. Con distribución Poisson.
4. Con distribución uniforme continua.
5. Con distribución logística.
6. Con distribución exponencial (parametrizada con parámetro de escala, en vez de parámetro de tasa).
7. Con distribución Erlang (parametrizada con parámetro de escala, en vez de parámetro de tasa).

A continuación encontrarán algunas de las distribuciones más conocidas y utilizadas, junto con las relaciones que hay entre ellas http://www.math.wm.edu/~leemis/chart/UDR/UDR.html:

También para una consulta (rápida pero precavida) pueden ir a: https://en.wikipedia.org/wiki/List_of_probability_distributions.


Zipf(α, n)
Discrete uniform(a, b)
R, V
Rectangular(n)
V
Beta-binomial(a, b, n)
Negative hypergeometric(n1, n2, n3)
Zeta(α)
Logarithm(c)
Power series(c, A(c))
Poisson(μ)
C
Hypergeometric(n1, n2, n3)
Beta-Pascal(n, a, b)
Gamma-Poisson(α, β)
Binomial(n, p)
Cp
Bernoulli(p)
M, P, X
Geometric(p)
F, M, V
Pascal(n, p)
Cp
Polya(n, p, β)
Gamma-normal(μ, α, β)
Discrete Weibull(p, β)
V
Standard normal
Normal(μ, σ2)
L
Noncentral beta(β, γ, δ)
Log normal(α, β)
P
Arctangent(λ, φ)
S, V
Noncentral chi-square(n, δ)
C
Log gamma(α, β)
Generalized gamma(α, β, γ)
Chi(n)
Inverted gamma(α, β)
Gamma(α, β)
Cα, S
Beta(β, γ)
Hyperbolic-secant
V
Inverse Gaussian(λ, μ)
Cauchy(a, α)
C, I, S, V
Chi-square(n)
C
Inverted beta(β, γ)
Arcsin
V
Standard Wald(λ)
S
Makeham(δ, κ, γ)
Standard Cauchy
I, S, V
t(n)
Hypoexponential(α⃗)
C
Erlang(α, n)
S
Gompertz(δ, κ)
V
Noncentral t(n, δ)
F(n1, n2)
I
Exponential(α)
F, M, S, V
Logistic-exponential(α, β)
S, V
Exponential power(λ, κ)
V
Doubly noncentral t(n, δ, γ)
Hyperexponential(α⃗)
Muth(κ)
Error(a, b, c)
S
Standard uniform
V
Minimax(β, γ)
Mβ, V
Noncentral F(n1, n2, δ)
IDB(δ, κ, γ)
Laplace(α1, α2)
V
Standard power(β)
V, X
Doubly noncentral F(n1, n2, δ, γ)
Rayleigh(α)
M, S, V
Pareto(λ, κ)
M, V
Standard triangular
V
Power(α, β)
S, V, Xα
Weibull(α, β)
Mβ, S, V
Log logistic(λ, κ)
I, S, V
Benford
V
TSP(a, b, m, n)
V
Uniform(a, b)
R, V
Extreme value(α, β)
V, Mβ
Lomax(λ, κ)
V
von Mises (κ, μ)
S
Generalized Pareto(δ, κ, γ)
Logistic(λ, κ)
S, V
Triangular(a, b, m)
V
Kolmogorov-Smirnov(n)
V1−4
Properties:
C: Convolution
F: Forgetfulness
I: Inverse
L: Linear combination
M: Minimum
P: Product
R: Residual
S: Scaling
V: Variate generation
X: Maximum
L ⇒ C
L ⇒ S
F ⇒ R
Relationships:
Special cases
Transformations
Limiting
Bayesian

10 CyO en Python

En esta sección se hará una revisión de la implementación de clases y objetos en Python (obviamente, Python como lenguaje de programación y **NO** como herramienta de cálculo o de análisis de datos).

En esta revisión se hace mención a la creación de clases y objetos, y a la implementación en Python del ocultamiento de la información, de la herencia y del polimorfismo.

Se espera que al finalizar las actividades de esta sección, el estudiante entienda y tenga clara la manera en que se deben implementar clases y objetos mediante el uso del lenguaje de programación Python.

> **Preparación de clase**
>
> Para la siguiente sección, lea **todo** el texto y ejecute **todo** el código que allí se incluye, haciendo **todas** las pruebas, cambios y experimentos que se le puedan ocurrir sobre dicho código.
> En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó, ejecutó, probó y experimentó; incluya su discusión, reflexiones y conclusiones al respecto; exponga las preguntas que le surgieron y las respuestas que intentó encontrar para dichas preguntas. Todo lo anterior para ser compartido y discutido en clase.

10.1 Elementos básicos de POO en Python

Cuaderno computacional en Google Colaboratory:
Clases, objetos, herencia y polimorfismo en Python

10.2 Ejercicios

11 Algunas CyO *Built-in* en Python

En esta sección se hará una revisión de algunas clases de particular importancia o relevancia, que están incluidas dentro de Python. Estás clases tienen objetivos particulares muy puntuales y permiten a los programadores hacer ciertas tareas generales y específicas.

> **i** Preparación de clase
>
> Para las siguientes secciones, lea **todo** el texto y ejecute **todo** el código que allí se incluye, haciendo **todas** las pruebas, cambios y experimentos que se le puedan ocurrir sobre dicho código.
> En sus propias palabras, explique lo que le transmitió y lo que le enseño cada parte de lo que leyó, ejecutó, probó y experimentó; incluya su discusión, reflexiones y conclusiones al respecto; exponga las preguntas que le surgieron y las respuestas que intentó encontrar para dichas preguntas. Todo lo anterior para ser compartido y discutido en clase.

11.1 Cadenas y *collection types*

En el siguiente cuaderno se hará una revisión de las clases/objetos *colección* de Python, en representación de los tipos de datos compuestos y de las estructuras de datos básicas. En esta revisión se hace mención a las clases: cadena (`str`), tupla (`tuple`), lista (`list`), conjunto (`set`) y diccionario (`dict`).

Cadenas, tuplas, listas, conjuntos y diccionarios

11.2 Excepciones o errores

En el siguiente cuaderno se hará una revisión de las clases/objetos derivados de la clase `Exception` de Python, para la gestión, control, manejo o tratamiento de excepciones/errores.

Manejo de excepciones

11.3 Archivos de texto plano

En el siguiente cuaderno se hará una revisión de las clases/objetos básicos en Python para el manejo de archivos de texto plano (`io.TextIOWrapper` y `io.TextIOBase`).

Manejo de archivos de texto plano

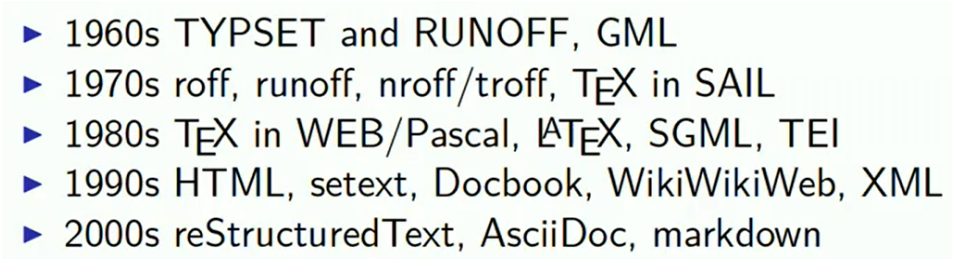
11.4 Ejercicios

A Lenguajes de marcado

En este apéndice se hará una revisión de algunos lenguajes de marcado.

Lenguaje de marcado - Wikipedia, la enciclopedia libre: https://es.wikipedia.org/wiki/Lenguaje_de_marcado

A brief history of text markup languages - Tony Ibbs: https://youtu.be/P-7hwjocEpM



A.1 TeX

A.2 LaTeX

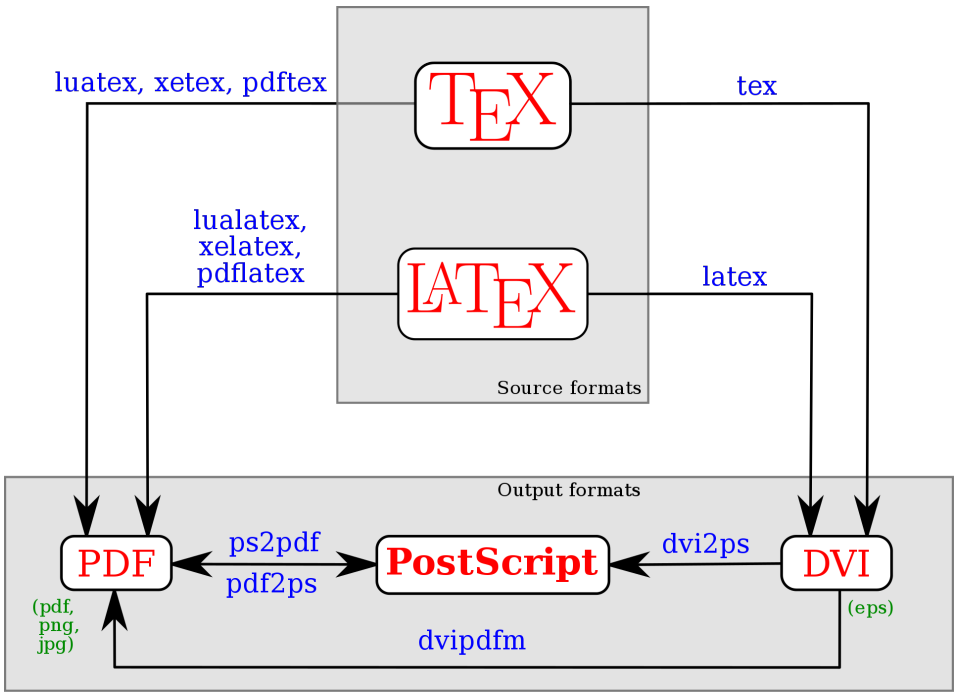
A.2.1 Edición y procesamiento

Editores (que he utilizado):

- TeXstudio, TeXmaker, TeXworks (Windows)
- Kile (Linux)
- Overleaf (Web)
- Cualquier editor de texto y el uso de línea de comandos.

Distribuciones que se encargan del procesamiento del archivo fuente (que he utilizado):

- MikTeX (Windows)
- TeX Live (Linux)



Referencias (que he consultado):

- ***Universo LaTeX***: http://ciencias.bogota.unal.edu.co/menu-principal/publicaciones/biblioteca-digital/matematicas/ o en este enlace
- ***LaTeX - Wikibooks, open books for an open world***: https://en.wikibooks.org/wiki/LaTeX
- ***LaTeX Math for Undergrads***: http://tug.ctan.org/info/undergradmath/undergradmath.pdf
- ***LaTeX Cheat Sheet***: https://wch.github.io/latexsheet/latexsheet.pdf

Ejemplo A.1. Documento con formato de artículo obtenido usando LaTeX.

Archivo fuente: Ejemplo1LaTeX.tex

En línea de comandos:

- Con

  ```
  latex Ejemplo1LaTeX.tex
  ```

  la salida (output) es: Ejemplo1LaTeX.dvi

- Con

  ```
  pdflatex Ejemplo1LaTeX.tex
  ```

  la salida (output) es: Ejemplo1LaTeX.pdf

Ejemplo A.2. Presentación o diapositivas formato *beamer* obtenidas usando LaTeX y BibTeX para el manejo de la bibliografía.

Archivo fuente: Semana02-DescripcionUnaVariable.tex

Archivo fuente bibtex: bibliography.bib

- BibTeX en Google Scholar: Ir a https://scholar.google.com y para una fuente que se quiera citar dar clic en "Cite".
- BibTeX en Google Books: Ir a https://books.google.com y para una fuente que se quiera citar dar clic en "Create citation".

A.2.2 Algunas plantillas de interés

- Revista Colombiana de Estadística - LaTeX template (revcoles)

  https://revistas.unal.edu.co/index.php/estad

  con esta misma plantilla: symposium (Simposio Colombiano de Estadística), project (trabajo de grado, Carrera de Estadística), report (trabajo para un curso).

- LaTeX Plantilla para la presentación de Tesis UN

  https://repositorio.unal.edu.co/handle/unal/55948

A.3 HTML

A veces Markdown se queda corto para lo que uno desea, y si la salida va a ser un archivo `.html`, entonces se pueden utilizar marcajes del lenguaje HTML (aunque con ello ya no sea tan fácil que con el mismo archivo fuente se puedan generar diferentes salidas, por ejemplo archivos `.html` y `.pdf` que tengan el mismo contenido en su propio formato).

Para los que por su cuenta quieran empezar a aprender el lenguaje HTML, podría servirles el tutorial que se encuentra en el siguiente enlace: https://www.w3schools.com/html/default.asp

Ejemplo A.3. Documento HTML.

Abra el siguiente archivo con un editor de texto plano como notepad (abrir como…) y luego con un navegador web: EjemploHTML.html

Adicionalmente, para los que por su cuenta quieran *empezar a hacerse una idea rápida* de:

- CSS: https://www.w3.org/Style/Examples/011/firstcss.es.html
- XML: http://recursostic.educacion.es/observatorio/web/fr/software/programacion/675-xml

A.4 Markdown

- Markdown fue creado por John Gruber en 2004, quien trabajó con Aaron Schwartz en la sintaxis.
- John Gruber define Markdown de la siguiente manera:

  > "Markdown is a converting text to HTML for web editors. Markdown allows you to write using an easy to read and write plain text format, then convert it to structurally valid XHTML (or HTML)."

- La definición de Gruber muestra que Markdown está dirigido a cualquiera que esté interesado en producir contenido para la Web. Markdown es libre y de código abierto.

A.4.1 Los beneficios de Markdown

- Markdown provee significado semántico al contenido de una manera relativamente simple.
- Se puede escribir extremadamente rápido contenido enriquecido con formato (si se compara con escribir directamente las correspondientes etiquetas html)
- Es muy fácil leer Markdown en texto plano antes de producir una salida HTML.
- En la mayoría de los casos se puede trabajar de corrido en un documento sin las pausas o interrupciones que implica usar el ratón o dar clic a botones/menús.
- Es independiente de plataforma/editor, así que el contenido no está atado al formato propio del software que se haya usado para escribirlo.
- Markdown es lígero, lo que significa que no es necesario aprender tantas cosas para poder empezar a usarlo.
- En especial, algunos periodistas y blogueros lo prefieren y lo alaban, gracias a que les ayuda a hacer su trabajo lo más fácil, rápido y simple posible.

A.4.2 Algunas críticas que se le hacen a Markdown

- Markdown no fue diseñado específicamente para escribir documentación, así que es lógico que tenga ciertas limitantes al usarlo para eso.
- A diferencia de otros lenguajes de marcado, Markdown no está estandarizado, por lo que en algunos casos no se sabe con absoluta certeza cómo será la visualización en un navegador en particular.
- Hay demasiadas variantes de Markdown, debido a que las personas lo adaptan y extienden para que tenga las funcionalidades que deseen, y como consecuencia estas variantes no son compatibles entre sí.
- Markdown combina el significado semántico del texto y la manera en que debería ser mostrado, para algunas personas eso no es apropiado.
- Es limitado en cuanto al estilo en que se puede mostrar el contenido (sin embargo, Markdown fue diseñado intencionalmente así).
- ***Markdown Reference***: https://commonmark.org/help/
- ***Markdown Tutorial***: https://commonmark.org/help/tutorial/index.html

Ejemplo A.4. Documento usando Markdown.

Archivo fuente: EjemploMarkdown.md

Usando pandoc:

1. Instale pandoc (https://pandoc.org/installing.html):
2. En línea de comandos (por ejemplo con `cmd` en Windows) ejecute:
   - Para obtener como salida un archivo `.html` que usa mathjax para las fórmulas matemáticas:

     ```
     pandoc EjemploMarkdown.md -f markdown -t html -s --mathjax -o EjemploMarkdown.htm
     ```

   - Para obtener como salida un archivo `.html` que usa mathml para las fórmulas matemáticas:

     ```
     pandoc EjemploMarkdown.md -f markdown -t html -s --mathml -o EjemploMarkdown.html
     ```

   - Para obtener como salida un archivo `.tex`:

     ```
     pandoc EjemploMarkdown.md -s -o EjemploMarkdown.tex
     ```

- Para obtener como salida un archivo `.pdf`:

```
pandoc EjemploMarkdown.md -s -o EjemploMarkdown.pdf
```

A.5 R Markdown

- ***R Markdown***: https://rmarkdown.rstudio.com/
- ***Gallery***: https://rmarkdown.rstudio.com/gallery.html
- ***Get Started***: https://rmarkdown.rstudio.com/lesson-1.html
- ***R Markdown Reference Guide***: https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf
- ***Dynamic documents with rmarkdown cheatsheet***: https://github.com/rstudio/cheatsheets/raw/master/rmarkdown.pdf
- Algunos paquetes para obtener documentos con algunos formatos adicionales (artículos, libros, presentaciones, etc.):

```
install.packages("tufte", "flexdashboard", "xaringan", "revealjs", "prettydoc", "rmdf
```

- ***R Markdown: The Definitive Guide***: https://bookdown.org/yihui/rmarkdown/

Ejemplo A.5. Documento usando R Markdown.

Archivo fuente: EjemploRMarkdown.Rmd

Dentro de `R`, con la librería `rmarkdown` instalada y con `pandoc` instalado y correctamente referenciado dentro de `R` (`Sys.setenv(RSTUDIO_PANDOC="C:/Program Files/RStudio/bin/pandoc/")`) ejecute:

- Para producir como salida un archivo `.html` (EjemploRMarkdown.html):

```
rmarkdown::render('EjemploRMarkdown.Rmd', 'html_document')
```

- Para producir como salida un archivo `.pdf` (EjemploRMarkdown.pdf):

```
rmarkdown::render('EjemploRMarkdown.Rmd', 'pdf_document')
```

A.6 Quarto

- ***Quarto***: https://quarto.org/
- ***Quarto - Gallery***: https://quarto.org/docs/gallery/
- ***Quarto - Get Started***: https://quarto.org/docs/get-started/
- ***Quarto - Guide***: https://quarto.org/docs/guide/

A.7 Ejercicios

1. Cree un archivo `.tex` e incluya en él todos los ejemplos que se encuentran en los capítulos 3 y 4 del libro:

   De Castro Korgi, Rodrigo. (2007). ***El universo LaTeX***. Universidad Nacional de Colombia. http://ciencias.bogota.unal.edu.co/menu-principal/publicaciones/biblioteca-digital/matematicas/

   A partir de dicho archivo genere o produzca el `.pdf` correspondiente y observe/compare la manera en que allí se visualiza cada ejemplo incluido en el archivo `.tex`.

2. Usando los lenguajes de marcado: **Markdown para el texto con formato** y **LaTeX para las fórmulas matemáticas**, cree un archivo `.Rmd` para producir un documento `.pdf` en donde reproduzca al menos 5 páginas del libro:

   Blanco Castañeda, Liliana (2004). ***Probabilidad***. Universidad Nacional de Colombia. https://repositorio.unal.edu.co/handle/unal/53471

   o del libro:

   Mayorga A., J. Humberto (2004). ***Inferencia Estadística***. Universidad Nacional de Colombia. https://repositorio.unal.edu.co/handle/unal/53475

   Para que este punto sea lo más formativo posible, escoja páginas consecutivas que tengan la mayor cantidad y complejidad de texto con formato (títulos, negrilla, cursiva, enumeraciones, etc.), fórmulas, gráficas, tablas, enlaces, entre otros.

3. Haga el mínimo de modificaciones al archivo `.Rmd` creado en el punto anterior para producir un documento `.html`.

4. Haga el mínimo de modificaciones al archivo `.Rmd` creado en el punto anterior para producir un documento `.html` que use uno de los temas incluidos en el paquete `prettydoc` (*engine* `html_pretty`). Además, haga la modificación requerida para que el documento `.html` no requiera de conexión a Internet para mostrar las expresiones/fórmulas matemáticas.

5. Usando los lenguajes de marcado: **Markdown para el texto con formato** y **LaTeX para las fórmulas matemáticas**, cree un archivo `.Rmd` para producir una presentación `xaringan` (https://bookdown.org/yihui/rmarkdown/xaringan.html) de mínimo 15 diapositivas. El tema de la presentación será la sección que usted desee del libro:

   Blanco Castañeda, L. (2004). ***Probabilidad***. Universidad Nacional de Colombia. https://repositorio.unal.edu.co/handle/unal/53471

   o del libro:

   Mayorga A., J. Humberto (2004). ***Inferencia Estadística***. Universidad Nacional de Colombia. https://repositorio.unal.edu.co/handle/unal/53475

   Para que este punto sea lo más formativo posible, complemente el tema seleccionado con otras fuentes bibliográficas y trate de incluir todo tipo de elementos en la presentación: texto con diversos formatos (títulos, negrilla, cursiva, enumeraciones, etc.), fórmulas, gráficas, tablas, enlaces, entre otros.

B R como herramienta

En este apéndice se hará una revisión de algunos elementos disponibles (preconstruidos) en R, que le podrían ser de utilidad a una persona que desee usar dicho software como una herramienta de cálculo o de análisis de datos.

Introducción a R como herramienta de cálculo o análisis de datos

C Python como herramienta

En este apéndice se hará una revisión de algunos elementos disponibles (preconstruidos) en Python, que le podrían ser de utilidad a una persona que desee usar dicho software como una herramienta de cálculo o de análisis de datos.

Introducción a Python como herramienta de cálculo o análisis de datos

Bibliografía

[1] Corrado Böhm y Giuseppe Jacopini. “Flow diagrams, Turing machines and languages with only two formation rules”. En: *Communications of the ACM* 9.5 (1966), págs. 366-371.