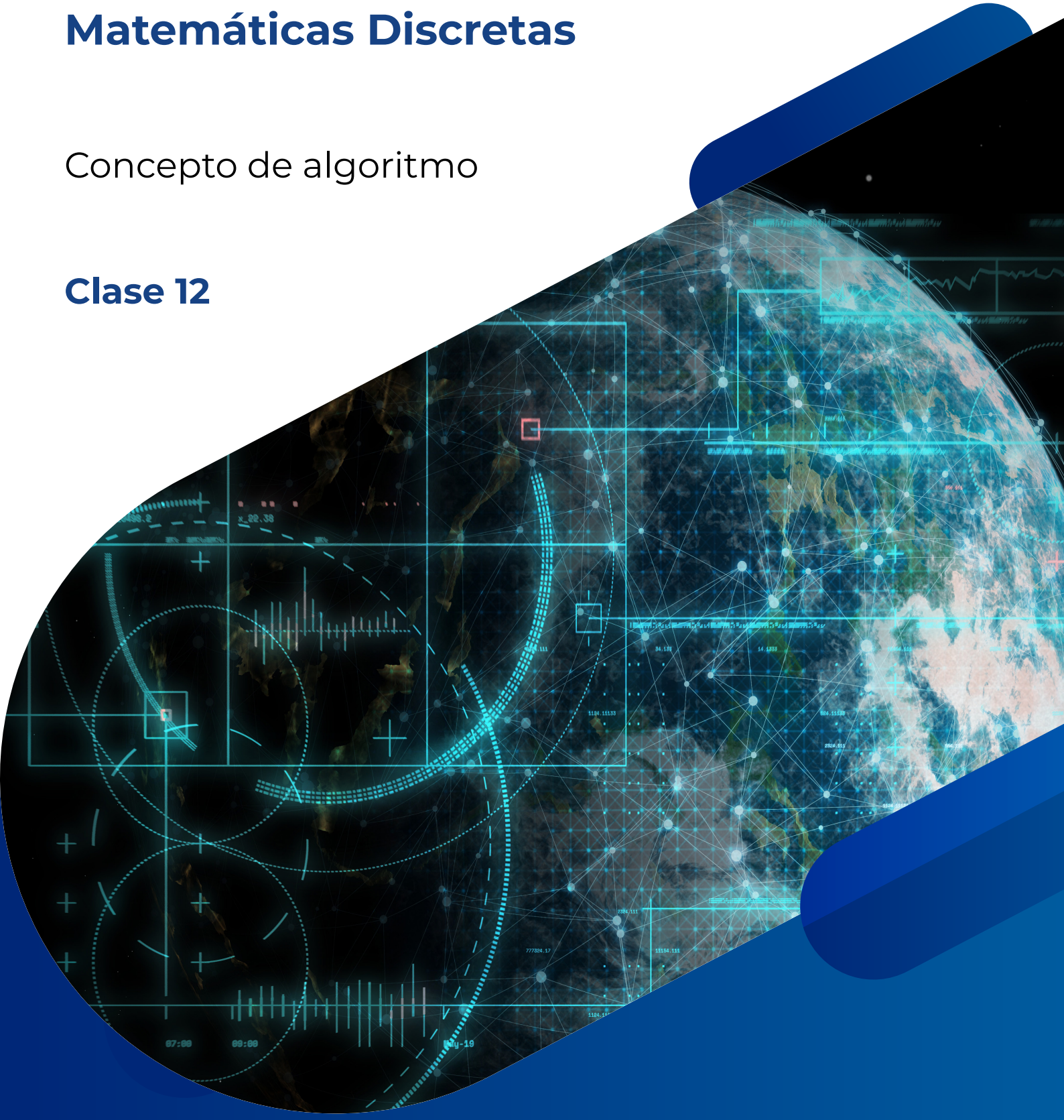
Matemáticas Discretas

Concepto de algoritmo

Clase 12



La excelencia no se improvisa

Introducción de la clase

¿Alguna vez te has preguntado qué hace que un conjunto de instrucciones se considere un algoritmo? En esta clase, nos adentraremos en ese concepto. Veremos que un algoritmo no es solo una lista de pasos, sino una herramienta precisa, finita y efectiva que permite resolver problemas, desde ordenar una lista hasta verificar la validez de una contraseña. A través de ejemplos prácticos y representaciones intuitivas, aprenderás a identificar cuándo un procedimiento es realmente un algoritmo y qué características debe tener para considerarse correcto.

Además, conocerás cómo distinguir entre algoritmos correctos e incorrectos, un aspecto central si queremos desarrollar soluciones confiables y seguras. Finalmente, exploraremos un tipo especial de algoritmo que se llama recursivo, y que permite abordar problemas complejos dividiéndolos en versiones más simples de sí mismos. Esta clase te proporcionará los fundamentos necesarios para razonar con claridad sobre procedimientos computacionales, tanto al diseñarlos como al evaluarlos críticamente.

Clase 12: Concepto de algoritmo

RDA 2: Resolver problemas prácticos y teóricos mediante la aplicación de técnicas y herramientas de la Matemática Discreta, como lógica matemática, análisis de conjuntos y teoría de grafos en el manejo de estructuras discretas.

12. Concepto de algoritmo

La palabra «algoritmo» proviene del nombre del matemático persa del siglo IX, Muhammad ibn Musa al-Khwarizmi (Johnsonbaugh 2018), cuyas obras influyeron profundamente en el desarrollo de las matemáticas y la computación. Con el tiempo, su nombre fue latinizado como *Algoritmi*, y de ahí surgió el término que hoy usamos para referirnos a procedimientos precisos para resolver problemas.

En la actualidad, cuando hablamos de algoritmos, generalmente nos referimos a instrucciones claras y finitas que pueden ser ejecutadas por una computadora para transformar entradas en salidas. Sin embargo, los algoritmos no son exclusivos del mundo digital: también están presentes en nuestra vida cotidiana. Por ejemplo, seguir

una receta de cocina para preparar un pastel implica una secuencia finita de pasos, donde se parte de unos ingredientes (entrada) y se obtiene un pastel (salida). Otro ejemplo es el algoritmo que aplicamos al buscar una palabra en un diccionario: comenzamos por la letra inicial, comparamos alfabéticamente y avanzamos hasta encontrar la entrada deseada.

También usamos algoritmos sin darnos cuenta al realizar tareas como calcular el vuelto en una compra, organizar libros por orden alfabético o seguir las instrucciones de montaje de un mueble. En todos estos casos, aplicamos un conjunto de reglas bien definidas para alcanzar un resultado. En el contexto de la computación, la diferencia es que los algoritmos deben estar escritos de forma tan precisa que una máquina pueda ejecutarlos sin ambigüedades ni interpretación humana.

12.1. Definición y propiedades

Resolver un problema mediante un algoritmo requiere, en primer lugar, entender con claridad qué se desea obtener y cómo se presentan las situaciones a resolver. A esto lo llamamos **problema algorítmico** (Martínez 2022), y consiste en tres elementos fundamentales:

- Un **problema a resolver**.
- Una **entrada**, que describe las instancias que deseamos procesar.
- Una **salida**, que representa el resultado esperado y sus propiedades.

Ejemplo 1. *Consideremos el siguiente problema algorítmico:*

- ***Problema:*** *Ordenar una lista finita de números enteros.*
- ***Entrada:*** *Una lista finita de números enteros* $L = [a_1, a_2, \dots, a_n]$.
- ***Salida:*** *Una lista con los mismos elementos, pero ordenados de forma creciente:* $L' = [b_1, b_2, \dots, b_n]$ *tal que* $b_i \leq b_{i+1}$ *para todo* i*, y que* L' *contenga exactamente los mismos elementos que* L.

Un **algoritmo** es una serie de pasos finitos que resuelve un problema algorítmico. Esta serie de pasos debe funcionar correctamente para *todas las posibles instancias válidas*

de entrada, no solo para unos pocos ejemplos. Su diseño implica especificar con claridad cómo se recibe la entrada, cómo se procesa y cómo se obtiene la salida deseada.

Todo algoritmo debe cumplir con las siguientes propiedades (Johnsonbaugh 2018; Erciyes 2021):

- **Entrada:** Puede recibir cero o más entradas desde el entorno externo. *Ejemplo:* Un algoritmo que genera un número aleatorio no necesita entrada; en cambio, uno que ordena números necesita una lista como entrada.
- **Salida:** Debe producir al menos una salida que sea la solución del problema. *Ejemplo:* El algoritmo que ordena una lista debe devolver una nueva lista ordenada.
- **Definición precisa:** Cada paso debe estar claramente definido y sin ambigüedad. *Ejemplo:* «Suma el número actual con el inmediato anterior» es claro; en cambio, «suma los números anteriores» es ambiguo.
- **Finitud:** El algoritmo debe terminar después de un número finito de pasos. *Ejemplo:* Un algoritmo que suma los primeros n números naturales termina; uno que espera una señal de red puede no terminar nunca.
- **Efectividad:** Cada operación debe ser básica y realizable en un tiempo razonable. *Ejemplo:* Dividir un número entre 2 es efectivo; calcular todos los posibles resultados de una operación sin límite de tiempo no lo es.
- **Corrección:** El algoritmo debe producir resultados correctos para todas las entradas válidas. *Ejemplo:* Un algoritmo que devuelve una lista ordenada de forma incorrecta no es útil, aunque termine rápido.

Es importante diferenciar entre un **algoritmo** y un **programa**. Un algoritmo es la idea general del procedimiento, mientras que un programa es su implementación concreta en un lenguaje que la computadora puede ejecutar. Además, algunos programas pueden estar diseñados para no terminar, como los servidores que esperan peticiones indefinidamente.

Un mismo algoritmo puede expresarse de diferentes maneras(Martínez 2022), dependiendo del nivel de abstracción:

- En **lenguaje natural** (palabras cotidianas).
- En **pseudocódigo** (una mezcla informal de instrucciones estructuradas).
- En **lenguajes de programación** como Python, C++, JavaScript, etc.

A medida que bajamos de nivel, los lenguajes son menos comprensibles para humanos pero más adecuados para que una computadora los ejecute. A esto se le llama nivel de abstracción: lo que está más «arriba» (lenguaje natural) es más expresivo, lo que está «abajo» (lenguaje de programación) es más formal y estructurado.

Incluso dentro de los lenguajes de programación existen distintos niveles. Por ejemplo, los lenguajes de **alto nivel**, como Python o Java, permiten escribir instrucciones cercanas al pensamiento humano, con estructuras de control y manejo automático de memoria. Por otro lado, los lenguajes de **bajo nivel**, como C o ensamblador, ofrecen mayor control sobre los recursos del sistema, pero requieren especificar detalles más técnicos, como la gestión de punteros o registros.

Figura 1

Niveles de abstracción

Nota: creación propia (Merino, A., 2025).

Ejemplo 2. *Consideremos el siguiente problema algorítmico:*

- ***Problema:*** *Contar cuántos números impares hay en una lista.*
- ***Entrada:*** *Un número entero positivo* n *y una lista* $L = [a_1, a_2, \dots, a_n]$.
- ***Salida:*** *La cantidad de enteros en* L *que sean impares.*

El algoritmo que resuelve este problema puede ser expresado en los diferentes niveles:

- ***Descripción en lenguaje natural:***

  Llevaremos una cuenta de cuántos números impares hay. Recorremos la lista de izquierda a derecha. Si el número no es divisible para 2 (es decir, si su residuo al dividirlo entre 2 es diferente a 0), aumentamos el contador. Al final, devolvemos ese número.

- ***Pseudocódigo:***

  Algoritmo 1

  Contar números impares en una lista

  Entrada: Una lista de enteros $L = [a_1, a_2, \ldots, a_n]$
  Salida : Número de elementos pares en L

  contador ← 0
  para cada elemento *en* L **hacer**
  si elemento mód 2 ≠ 0 **entonces**
  contador ← contador + 1
  devolver contador

  Nota: creación propia (Merino, A., 2025).

- ***Código en Python:***

```
def contar_impares(L):
    cont = 0
    for k in L:
        if k % 2 != 0:
            cont += 1
    print(cont)
```

  Nota: creación propia (Merino, A., 2025).

Para explorar una perspectiva más amplia sobre qué es un algoritmo, te recomendamos revisar el siguiente recurso.

- **Título del enlace:** ¿Qué es un algoritmo?
- **Descripción del enlace:** Se presenta una explicación accesible sobre qué es un algoritmo, sus características principales y ejemplos.

Enlace: https://www.datacamp.com/es/blog/what-is-an-algorithm

12.2. Algoritmos correctos e incorrectos

Cuando diseñamos un algoritmo, no basta con que «parezca» que funciona: es esencial asegurarnos de que realmente resuelva el problema planteado en todos los casos posibles. Esta exigencia es similar a lo que ocurre con las demostraciones matemáticas: no se acepta una argumentación basada solo en ejemplos, sino que se requiere una justificación formal, clara y general.

De manera análoga, la **correctitud de un algoritmo** implica que, para toda entrada válida, el algoritmo produce la salida correcta según la especificación del problema. En otras palabras, un algoritmo es correcto si funciona correctamente en *todas* las instancias que debería resolver, no solo en algunas.

Para lograrlo, es importante acompañar cada algoritmo de una descripción precisa de lo que hace en cada paso, y de una demostración matemática que respalde su funcionamiento. Aunque en este curso no abordaremos a fondo las técnicas formales de verificación —como la inducción matemática, las invariantes de bucle o los esquemas de Hoare— es fundamental que desde ahora desarrolles el hábito de preguntarte no solo si un algoritmo *funciona*, sino si puedes *justificar por qué* funciona.

Veamos un ejemplo de un algoritmo que parece funcionar, pero en realidad no es correcto (Martínez 2022).

Ejemplo 3. *Consideremos el siguiente problema algorítmico:*

- ***Problema:*** *Determinar si cierta expresión es un número primo.*
- ***Entrada:*** *Un número entero positivo* n.
- ***Salida:*** *Decidir si el número* $n^2 - n + 41$ *es un número primo.*

Si probamos manualmente los valores de $n = 1,2,3,\dots,10$ *, obtenemos las expresiones* $41,43,47,53,61,71,83,97,113,131$*, que son todas ellas números primos. A partir de esto, alguien podría proponer un algoritmo que simplemente diga «sí» para cualquier valor de* n*, confiando en que la expresión siempre produce un número primo.*

Sin embargo, esto sería un error. Aunque la expresión $n^2 - n + 41$ *produce números primos durante un buen tramo inicial, no lo hace para siempre. Por ejemplo, si evaluamos en* $n = 41$*, obtenemos:*

$$41^2 - 41 + 41 = 1681.$$

Este número no es primo, ya que $1681 = 41 \times 41$*. Por tanto, el algoritmo que responde siempre «sí» falla para esta instancia. Y basta con que falle en un solo caso para que el algoritmo sea incorrecto.*

Este ejemplo muestra la importancia de no confiar únicamente en pruebas empíricas o ejemplos limitados. Aunque el algoritmo funcione para muchos valores de entrada, no podemos asegurar su validez general sin una demostración matemática. Demostrar la correctitud de un algoritmo es una tarea rigurosa, que en general requiere herramientas formales de razonamiento lógico y matemático, y cuya enseñanza excede el alcance de este curso. No obstante, debemos estar atentos a los posibles fallos y desconfiar de algoritmos que no tienen una justificación clara de por qué funcionan.

12.3. Algoritmos recursivos

Una de las ideas más elegantes y poderosas en el diseño de algoritmos es la **recursión**. Una función se dice *recursiva* cuando se define a sí misma sobre entradas más pequeñas o simples del mismo problema. De manera general, podemos pensar en un proceso recursivo como aquel que resuelve un problema dividiéndolo en subproblemas de la misma naturaleza, hasta llegar a casos lo suficientemente simples como para resolverse directamente.

Formalmente, la recursión se basa en el siguiente principio:

Teorema 1: Principio de recursión finita.

Dados A un conjunto, $c \in A$, y una función $g: A \to A$, existe una única función $g: \mathbb{N} \to A$ tal que

1. $f(0) = c$, y
2. $f(n+1) = g\big(f(n)\big)$, para todo $n \in \mathbb{N}$.

Este principio nos permite definir funciones paso a paso: partimos de un valor inicial (caso base) y vamos generando los siguientes valores aplicando una misma regla (paso recursivo). Por ejemplo, si deseamos definir la potencia a^n para $a \in \mathbb{R}\backslash\{0\}$ y $n \in \mathbb{N}$, podemos usar una función recursiva.

Tomemos $A = \mathbb{R}$, $c = 1$ y la función

$$\begin{array}{rccc} g\colon & \mathbb{R} & \to & \mathbb{R} \\ & x & \mapsto & x \cdot a. \end{array}$$

Gracias al principio de recursión finita, existe una única función $f\colon \mathbb{N} \to \mathbb{R}$ tal que:

- $f(0) = 1$
- $f(n+1) = g\big(f(n)\big) = f(n) \cdot a$, para todo $n \geq 0$

Si denotamos por a^n a $f(n)$, obtenemos la definición recursiva de las potencias:

- $a^0 = 1$
- $a^{n+1} = a^n \cdot a$, para todo $n \geq 0$

Por motivos de programación, es más común escribir esta definición de forma descendente:

- $a^0 = 1$
- $a^n = a^{n-1} \cdot a$, para todo $n > 0$

En términos generales, el principio de recursión finita nos dice que, para definir una función $f\colon \mathbb{N} \to A$, basta con establecer un **paso base** (el valor inicial) y un **paso recursivo** (una regla que permita construir el siguiente valor a partir de los anteriores).

Por otro lado, un algoritmo es **recursivo** si resuelve un problema reduciéndolo a uno o más casos del mismo problema con entradas más pequeñas. Esto requiere siempre identificar dos cosas:

- Un **caso base**, que pueda resolverse directamente.
- Un **caso recursivo**, que reduzca el problema original y lo acerque al caso base.

Ejemplo 4. *Consideremos el siguiente problema algorítmico:*

- ***Problema:*** *Calcular el factorial de un número natural* n.
- ***Entrada:*** *Un número natural* $n \in \mathbb{N}$.
- ***Salida:*** *El valor de* $n! = 1 \cdot 2 \cdot 3 \cdots n$.

La función factorial se puede definir de forma recursiva de la siguiente manera:

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n \cdot (n-1)!, & \text{si } n > 0 \end{cases}$$

Esta definición captura los dos aspectos la función recursiva:

- *El* ***caso base****: cuando* $n = 0$*, el resultado es* 1.
- *El* ***caso recursivo****: cuando* $n > 0$*, el resultado se obtiene multiplicando* n *por el factorial de* $n - 1$.

Ahora que entendemos la lógica detrás del factorial recursivo, podemos traducir la definición matemática a un lenguaje de programación. A continuación, se presenta el algoritmo primero en pseudocódigo para ilustrar la lógica de manera abstracta, y luego en Python, un lenguaje conocido por su sintaxis limpia que facilita la comprensión de la recursión.

Algoritmo 2

Cálculo recursivo del factorial

```
Entrada: Un número natural n
Salida  : El valor de n!

Función factorial(n):
    si n = 0 entonces
        devolver 1
    en otro caso
        devolver n · factorial(n - 1)
```

Nota: creación propia (Merino, A., 2025).

Ahora veamos cómo se traduce este mismo procedimiento a un lenguaje de programación.

Código 2

Función recursive para calcular el factorial

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Nota: creación propia (Merino, A., 2025).

Como puedes ver, la sintaxis del código es un reflejo directo de la definición matemática. La condición return 1 corresponde al caso base, mientras que la línea return n * factorial(n - 1) representa el caso recursivo, llamando a la misma función para resolver una parte más pequeña del problema.

Para fortalecer tu comprensión sobre la recursividad, te recomendamos ver el siguiente video. Ofrece una explicación clara, acompañada de visualizaciones que ayudan a entender cómo funciona un algoritmo recursivo paso a paso.

- **Título del enlace:** Recursividad | FÁCIL de entender y visualizar
- **Descripción del enlace:** Video didáctico que explica el concepto de recursividad con ejemplos visuales.

Enlace: https://youtu.be/YwRjEOFxvO0?si=VS9CA0jXdmozaMx7

Referencias

Erciyes, K. (2021). Discrete Mathematics and Graph Theory. Springer.

Johnsonbaugh, R. (2018). Discrete Mathematics (8.a ed.). Pearson Educación.

Martínez, L. (2022). Problemas y algoritmos — Matemáticas Discretas para Ciencia de Datos. https://madi.nekomath.com/P3/ProblemasAlgoritmos.html

Términos

- **Algoritmo:** Conjunto finito de pasos bien definidos que, dados ciertos datos de entrada, permite obtener un resultado esperado o resolver un problema específico.

- **Algoritmo recursivo:** Algoritmo que resuelve un problema reduciéndolo a uno o más subproblemas del mismo tipo, utilizando llamadas a sí mismo.

Profundización

Recurso 1

- **Título del recurso:** Ejemplos de algoritmos
- **Descripción del enlace:** Este recurso presenta ejemplos de varios algoritmos clásicos y su implementación en python.
- **Enlace:** https://colab.research.google.com/github/andres-merino/MatematicasDiscretas-Virtual-05-N0279/blob/main/pages/clases/clase12/EjemploAlgoritmos.ipynb

PUCE Virtual

La excelencia no se improvisa

síguenos