METODOLOGÍA

¿DE QUE CONSTA ESTE MÓDULO?

En este tema o módulo se tratarán los conceptos básicos de algoritmo, estructura de datos y se desarrollará la metodología para resolución de problemas.

CONCEPTO DE ALGORITMO

En el ámbito de la ciencia de la computación, un **algoritmo** se define como una secuencia **finita**, **ordenada** y **no ambigua** de instrucciones o pasos lógicos que permiten resolver un problema computacional o realizar una tarea específica. Normalmente Transforma una entrada (input) en una salida (output) mediante un proceso bien definido.

Los algoritmos constituyen la base de todo proceso computacional, y su diseño debe garantizar propiedades como **corrección**, **eficiencia**, **finitez**, y en muchos casos, **optimización del rendimiento**. En programación, los algoritmos se implementan mediante lenguajes formales que pueden ser interpretados o compilados por una máquina, permitiendo su ejecución automatizada.

A nivel de sistemas, tanto los sistemas operativos como las aplicaciones de usuario están construidos sobre múltiples algoritmos que gestionan desde operaciones básicas (como la asignación de memoria o el ordenamiento de datos) hasta tareas complejas como la planificación de procesos.

En áreas avanzadas como **Big Data** o **Inteligencia Artificial**, los algoritmos juegan un papel clave en la extracción de conocimiento a partir de grandes volúmenes de datos. En estos contextos, se diseñan algoritmos especializados para el aprendizaje automático, clasificación, regresión, detección de anomalías, entre otros, los cuales permiten modelar y predecir el comportamiento de usuarios o sistemas en tiempo real.

Características fundamentales de un algoritmo

- **Finitud**: debe terminar tras un número finito de pasos.
- **Precisión**: cada paso debe estar claramente definido (sin ambigüedades).
- **Orden lógico**: los pasos deben seguir una secuencia coherente.
- **Eficiencia**: debe optimizar recursos como tiempo y memoria.
- **Generalidad**: debe poder aplicarse en la medida de lo posible a una clase de problemas, no solo a un caso específico.

Componentes de un algoritmo

- **Entrada (Input)**
  Datos necesarios para iniciar el proceso.
- **Proceso**
  Conjunto de pasos o instrucciones que transforman la entrada.
- **Salida (Output)**
  Resultado final después de ejecutar el algoritmo.

CONCEPTO DE ESTRUCCTURA DE DATO

Una **estructura de datos** es una forma particular de organizar, gestionar y almacenar los datos en memoria de manera que permita un **acceso** y **modificación** eficientes. Su objetivo principal es optimizar el uso de recursos computacionales al manipular grandes volúmenes de información o realizar operaciones específicas como búsquedas, inserciones, eliminaciones o recorridos.

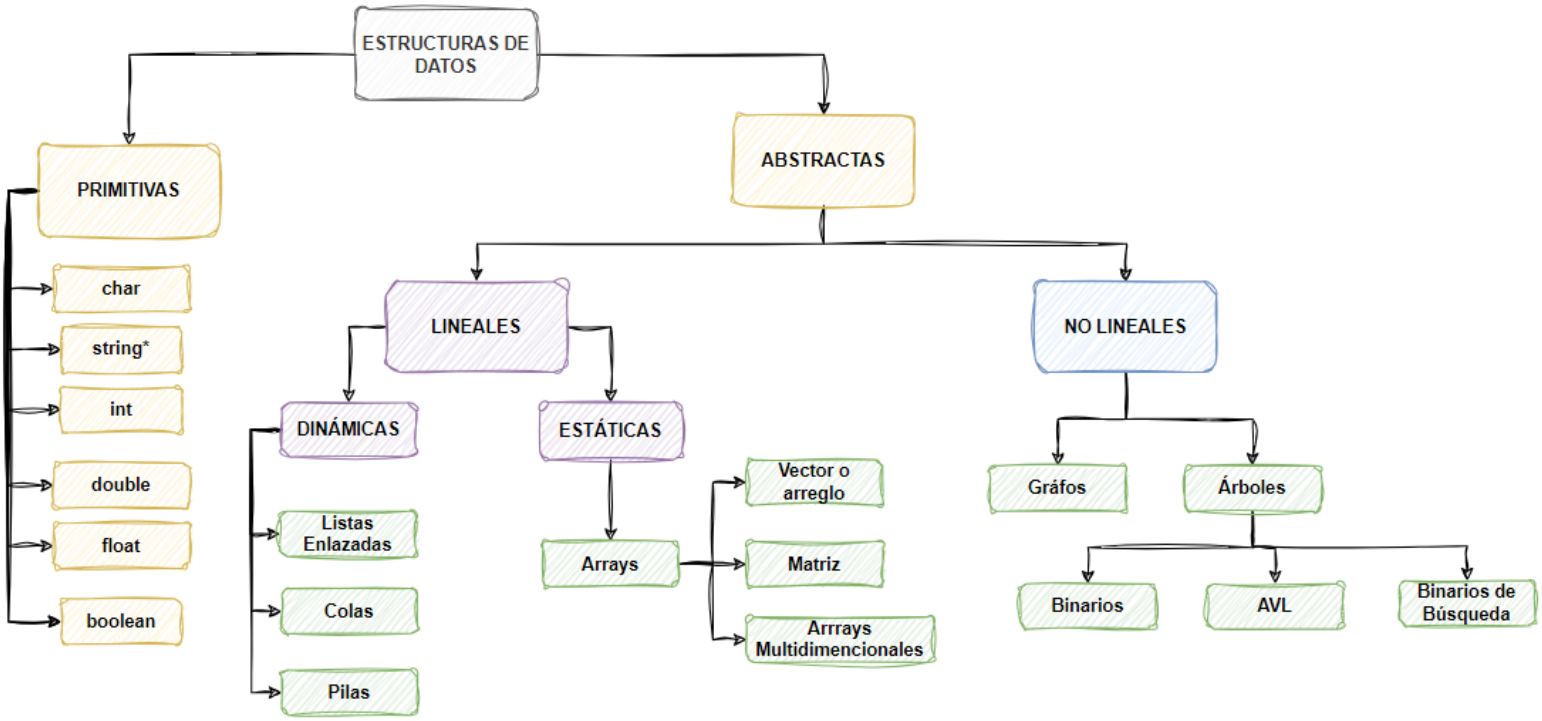
Cada estructura de datos está diseñada con base en una **estrategia algorítmica** que determina cómo se organizan los datos y qué operaciones están permitidas o son eficientes.

Ya sean las más utilizadas comúnmente como las **primitivas**, **arrays**, **clases** o las diseñadas para un propósito específico **listas**, **árboles**, **grafos**, **tablas hash**, etc., una estructura de datos nos permite trabajar en un nivel de **abstracción** la información para luego acceder a ella, modificarla y manipularla.

Las estructuras de datos son ingredientes esenciales para crear **algoritmos** rápidos y potentes.

TIPOS DE ESTRUCTURA DE DATOS

Las estructuras de datos pueden agruparse en dos tipos básicos: estructuras de datos **primitivas** y estructuras de datos **no primitivas o abstractas.**



PRIMITIVAS

Las estructuras de datos **primitivas** son los tipos de datos más básicos que están directamente soportados por un lenguaje de programación. Son los bloques de construcción fundamentales que se utilizan para construir estructuras de datos más complejas. Estas estructuras primitivas generalmente se implementan a nivel del hardware o están muy cerca del hardware, lo que significa que se manejan directamente por el procesador de la computadora.

PRIMITIVOS		
ESTRUCTURA	DESCRIPCIÓN	EJEMPLO
INT	Números enteros (positivos, negativos, cero) sin comas.	1, 2, 5, 1000, 19087
FLOAT	Números con decimales.	3.5, 6.7, 6.987, 20.2
CHAR	Caracter único.	A, B, C, F
STRING*	Cadena de chars, basicamente texto.	Hola mundo!
BOOLEAN	Valores logícos de verdadero o falso.	TRUE, FALSE

* **El STRING** no es un tipo primitivo, es un tipo abstracto. Pero en líneas generales su comportamiento lo podemos asemejar a un primitivo.

NO PRIMITIVAS O ABSTRACTAS

También se denominan estructuras de datos definidas por el usuario. Las estructuras de datos no primitivas se derivan de las estructuras de datos primitivas combinando dos o más de estas, a su vez las estructuras abstractas pueden dividirse en estructuras de datos **lineales** y estructuras de datos **no lineales**.

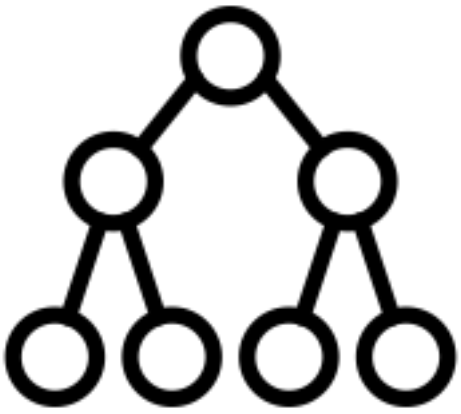
¿Qué es una estructura de datos lineal?

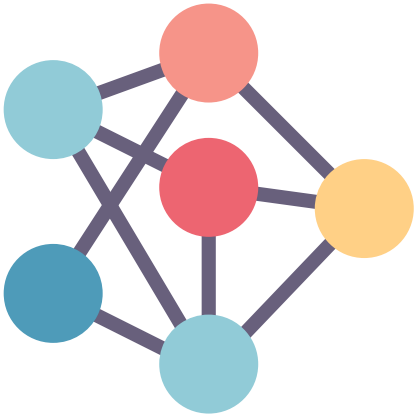
En una estructura de datos lineal, los elementos se disponen de forma secuencial y pueden ser **estáticos** o **dinámicos**.

En las estructuras de datos lineales estáticas, los tamaños y estructuras asociados a las posiciones de memoria son fijos en el momento de compilación, pero en las estructuras dinámicas, las posiciones de memoria asociadas cambian. Una estructura de datos estática es un **array** (de una o más dimensiones) mientras que las estructuras de datos dinámicas pueden ser listas enlazadas, pilas y colas.

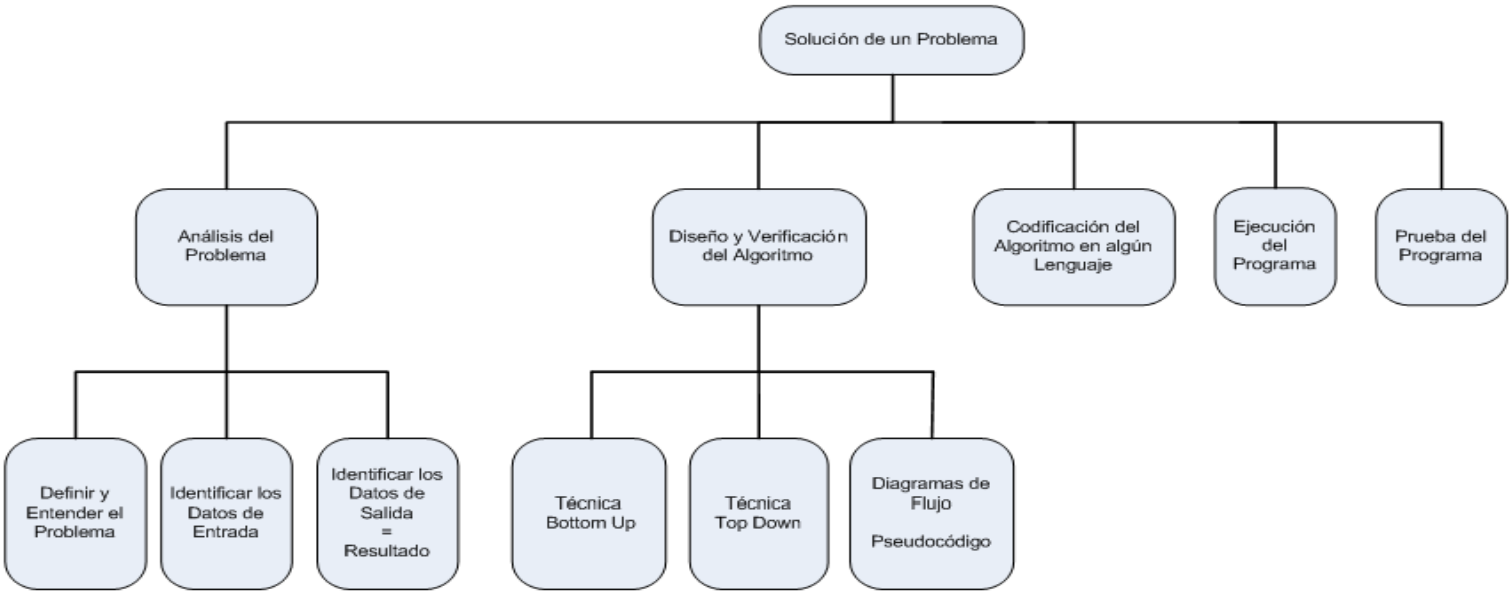
¿Qué es una estructura de datos NO lineal?

Cuando se trata de una estructura de datos no lineal, los datos están conectados a varios otros elementos y no están organizados secuencialmente. Aquí, es posible que un elemento de datos pueda conectarse con más de un elemento de datos, las estructuras de datos de **grafos y árboles** son estructuras de datos no lineales.





METODOLOGIA DE RESOLUCIÓN DE PROBLEMAS



ANALISIS DEL PROBLEMA

Definición y entendimiento del problema

Los problemas requieren de una **definición** clara y precisa. Es importante conocer y **entender** lo que se pretende obtener. Mientras esto no esté claro no tiene sentido pasar a la siguiente etapa.

Identificar los datos de entrada:

¿Qué datos necesita el algoritmo para funcionar correctamente?

- Ejemplo: si queremos calcular el promedio de notas, los datos de entrada son las calificaciones.

Identificar los datos de salida:

¿Qué resultados esperamos?

- Ejemplo: el promedio numérico (un número decimal o entero redondeado, según el contexto).

Luego de identificar las entradas y salidas se debe especificar :

Las Pre condiciones:

Condiciones que el algoritmo **asume** que deben cumplir los datos de entrada incluyendo las relaciones entre ellos.

Las Post condiciones:

Condiciones que cumplen los datos de salida y las relaciones entre los datos de entrada y los datos de salida.

Visualicemos lo expuesto con un ejemplo:

Se nos pide: Desarrollar una función que, dada dos posiciones, una inicial y una final, retorne el valor máximo encontrado en un vector de enteros solo considerando las posiciones antes mencionadas.

1) Definición y entendimiento del problema:
   En este caso, la letra o enunciado es muy simple, prácticamente define el problema en si mismo, no siempre tendremos esta facilidad.

   Pero un par definiciones más depuradas serian estas:

   - Crear una función o procedimiento que retorne **el máximo valor** dentro de un **subconjunto de elementos** de un vector, delimitado por dos índices dados como parámetros.
   - Crear una función que devuelva **el valor más grande** en un segmento de un vector de enteros, definido entre dos posiciones.

   Ahora hay que entender que se quiere lograr, para eso podemos visualizar el problema y realizarnos cuestionamientos que servirán para definir entradas, salidas, pre y post condiciones.

20	3	9	4	23	54	-4
0	1	2	3	4	5	6

   - ¿Qué retorna con desde 1 y hasta 4?
   - ¿Qué retorna con desde 3 y hasta 8?
   - ¿Qué retorna con desde -2 y hasta 2?
   - ¿Qué retorna con desde 5 y hasta 1?
2) Identificar los valores de entrada:
   - Un **vector de enteros** que contiene los elementos.

- Un **índice desde** que representa la posición inicial.
- Un **índice hasta** que representa la posición final.

3) Identificar valores de salida:
   - Un entero que representa el **valor máximo** del vector entre el índice de **desde** hasta el índice **hasta** inclusive.
4) Especificar pre y post condiciones:

Pre condiciones.

No hay que cometer el error de ser muy vagos en la definición de las pre condiciones, un ejemplo de una definición deficiente podría ser:

Pre: desde y hasta deben ser válidos. #ESTO ES ERRONEO

Gracias al entendimiento obtenido en la fase de definición y entendimiento del problema podemos llegar a una definición más precisa y correcta:

Pre:

- desde >= 0
- hasta >= 0
- desde < largo del vector
- hasta < largo del vector
- desde <= hasta
- el vector no sea nulo ni vacío

Post condiciones

Lo mismo explicado para las **pre**, aplica para las **post** condiciones:

Post: Retorna el máximo. #ESTO ES ERRONEO

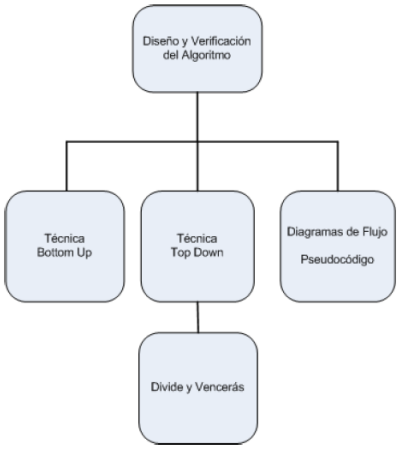
Post: Retorna el máximo valor del vector entre los índices desde y hasta inclusive.

En resumen:

Antes:
//Pre: desde y hasta deben ser válidos.
//Post: Retorna el máximo.

Después:
//Pre: desde >= 0, hasta >=0, hasta < largo del vector, desde < largo del vector, desde <= hasta, vector no sea nulo ni vacío
//Post: Retorna el máximo valor del vector entre los índices desde y hasta inclusive.

DISEÑO DEL PROBLEMA



Esta etapa es fundamental en la resolución de problemas. Consiste en **idear una solución estructurada y precisa** antes de implementarla en un lenguaje de programación. Aquí se establece el *cómo* resolver el problema, sin aún preocuparse del *con qué* (lenguaje, sintaxis, etc.).

En primer lugar, tratamos de plantear la lógica de solución del problema.

Luego representar el algoritmo en una forma clara, abstracta y verificable.

Validar que el algoritmo **cumple correctamente con los requerimientos** del problema.

Técnica Bottom-Up (de abajo hacia arriba):

Comienza con funciones simples que resuelven partes pequeñas.

Se integran en estructuras más grandes.

Es útil cuando ya se conocen soluciones parciales que luego se combinan.

- Ejemplo: resolver subproblemas en programación dinámica.

Técnica Top-Down (de arriba hacia abajo):

Comienza con la idea general y se divide en tareas más pequeñas (modularización).

Permite planificar primero la estructura general.

- Ejemplo: "mostrar menú", "leer datos", "calcular promedio", "mostrar resultado" (cada paso se detalla después).

Ejemplo abstracto de la diferencia entre estas técnicas:

Top-Down (de arriba hacia abajo)

Se empieza con la idea **GRANDE,** por ejemplo “quiero hacer una fiesta”.
Y luego se **divide** en partes más pequeñas.

Paso a paso (Top-Down):

- **Organizar la fiesta**
    - Elegir lugar
    - Hacer lista de invitados
    - Comprar comida
    - Preparar música
    - Decorar
- **Comprar comida**
    - Hacer lista del súper
    - Ir al supermercado
    - Cocinar
- **Decorar**
    - Comprar globos
    - Poner las luces
    - Armar la mesa

El problema grande se va descomponiendo en tareas más pequeñas hasta llegar a actividades simples y manejables.

***Pensar en lo grande primero y luego lo dividirlo*.**

Bottom-Up (de abajo hacia arriba)

Se empieza con las cosas pequeñas que son conocidas y realizables, y luego **se unen** para lograr el objetivo general.

Paso a paso (Bottom-Up):

- Comprar globos
- Poner luces
- Armar la mesa
  → Ya está lista la decoración
- Hacer lista del súper
- Comprar comida
- Cocinar
  → Ya está lista la comida
- Invitar a las personas
  → Ya están los invitados
- Elegir música
  → Ya hay música

- Juntar todo eso
  → ¡Fiesta lista!

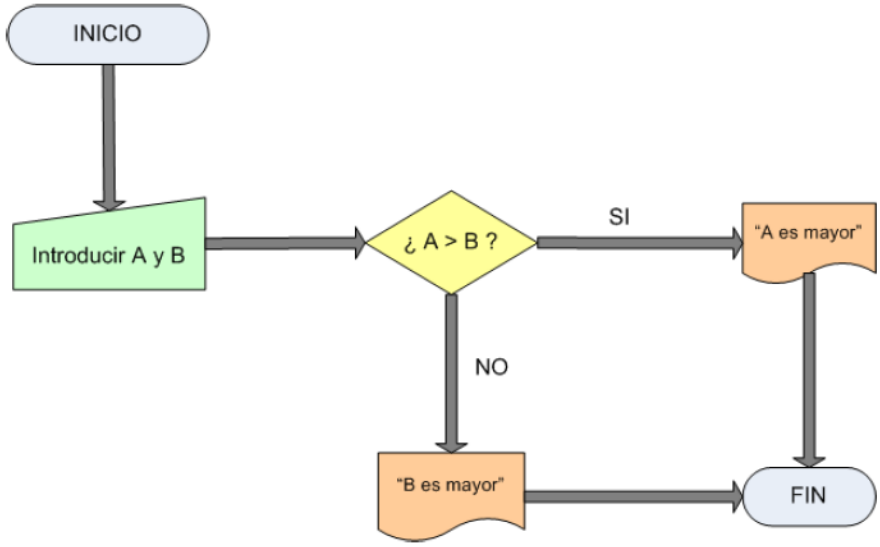
Primero se identifican las partes pequeñas que se pueden realizar, y luego se integran para formar el conjunto.

Diagramas de flujo y pseudocódigo:

Diagramas de flujo: Representación gráfica que muestra los pasos del algoritmo con símbolos (inicio, procesos, decisiones).

Pseudocódigo: Escritura del algoritmo usando una mezcla de lenguaje natural y notación estructurada, sin preocuparse por la sintaxis de un lenguaje formal.

Ejemplo, mayor entre dos números:



CODIFICACIÓN DEL ALGORITMO EN ALGÚN LENGUAJE

En esta etapa transformamos el algoritmo diseñado en código real, utilizando un lenguaje de programación (Java, Python, C++, etc.).

Aspectos importantes:

- Elegir el lenguaje adecuado para el problema y el contexto.
- Traducir cada paso del pseudocódigo o diagrama de flujo a instrucciones válidas.
- Usar buenas prácticas: nombres descriptivos, comentarios, indentación, control de errores, modularidad (métodos o funciones).

- Asegurar que el código sea **mantenible y entendible** para otros desarrolladores.

EJECUCIÓN DEL PROGRAMA

Una vez codificado, el programa debe ser ejecutado para observar su comportamiento.

¿Qué se hace aquí?

- Probar con algunos casos de entrada para ver si el programa funciona.
- Detectar errores comunes como:
    - Errores de sintaxis (problemas con la escritura del lenguaje).
    - Errores de ejecución (division por cero, acceso a índices inválidos, etc).
- Confirmar que el programa **compila/interpreta correctamente** y entrega resultados visibles.

PRUEBA DEL PROGRAMA (TESTING)

Después de que el programa se ejecuta, debemos verificar si resuelve el problema **correctamente en todos los casos posibles**.

Tipos de prueba:

- **Casos típicos**: Entradas normales y comunes.
- **Casos extremos o borde**: Valores muy grandes o pequeños, límites de entrada, valores específicos que se intuye pueden dar errores.
- **Casos inválidos o inesperados**: Para verificar si el programa los maneja correctamente (validación de datos).

¿Qué buscamos?

- Validar que **el programa cumple con los requisitos** planteados al inicio.
- Detectar **errores lógicos**: errores en el diseño del algoritmo que hacen que la salida sea incorrecta.
- Evaluar la eficiencia del programa: ¿tarda mucho con entradas grandes?, ¿consume mucha memoria?