# Unidad 1- PROGRAMACIÓN ESTRUCTURADA Y MODULARIZACIÓN.

## ¿Cómo diseñar algoritmos y luego implementarlos a partir de un problema específico?

### Conceptos.

1. **Entender el problema**
   Comprender completamente el problema que deseas resolver. Esto implica definir claramente el problema, los requisitos y los datos de entrada/salida.
2. **Planificar y definir objetivos:**
   Establecer metas claras para lo que el algoritmo debe lograr. Esto te ayudará a mantener el enfoque y evaluar el éxito del algoritmo.
3. **Diseñar el algoritmo:**
   Aquí es donde se crea una representación lógica del proceso que resolverá el problema. Utiliza pseudocódigo, diagramas de flujo o cualquier otra técnica para visualizar el algoritmo.
   - Considera las estructuras de control (condicionales, bucles) necesarias para manejar diferentes situaciones.
   - Identifica las variables y los datos que se necesitarán.
4. **Seleccionar las estructuras de datos:**
   Escoge las estructuras de datos adecuadas para almacenar y manipular la información. Pueden incluir arrays, listas, diccionarios, pilas, colas, árboles, etc.
5. **Implementar el algoritmo:**
   Traducir el diseño en un lenguaje de programación específico.

## Metodología para la construcción de algoritmos.

1. Definición del problema
2. Identificar el objetivo
3. Representación del pseudocódigo
4. Diseño de algoritmo en el lenguaje de programación que se requiera

## Programación estructurada.

Módulos.
Importancia de la definición del dominio de los datos de entrada y datos de salida.
Parámetros de entrada y parámetros de salida.
Características del lenguaje de programación.
Trazas. Documentación y estilo.
Buenos hábitos de programación.

# Unidad 2- TIPOS DE DATOS

## ¿Qué tipos de datos serán útiles para el manejo de los problemas específicos?

### Tipos de datos.

- Int
- Double
- Float
- String
- Boolean
- Char

## Importancia de la definición del dominio de los datos de entrada y datos de salida.

Es importante definir los datos de entrada ya que en base a ellos es con lo que se va a trabajar dentro del módulo/método y es importante el dato de salida ya que se tiene que pensar cómo o en que será necesario ese dato para dejar definido su dominio

## Concepto y uso de conversión entre tipos (casting).

### Tipo String

Java utiliza la clase String para manipulación de caracteres.
Se pueden utilizar funciones para pasar de String a Entero

### Entrada/Salida.

## Arreglos unidimensionales y bidimensionales.

### Concepto y aplicaciones sobre arreglos.

Definición Arreglo: colección de valores de datos, todos del mismo tipo.

### Arreglos unidimensionales

Cuando la colección de valores contenidos en un arreglo están numerados en una secuencia de valores, se considera a esta colección de elementos como pertenecientes a un arreglo unidimensional, para distinguirlos de aquellos en dos dimensiones (por ej. una matriz)

Los elementos se diferencian entre sí por el número de índice. Cada celda de este tipo especial de variable es numerada del 0 a n-1 , donde n es el número de celdas del arreglo

Para acceder a un elemento del arreglo se utiliza un índice que determina la posición requerida

**Ejemplo:** un arreglo llamado nota de enteros de 4 elementos

| 97 | 86 | 64 | 78 |
|---|---|---|---|

- nota[0] tiene el valor 97
- nota[1] tiene el valor 86
- nota[2] tiene el valor 64
- nota[3] tiene el valor 78

# Abstracción de Datos - Tipo de dato abstracto (TDA).

## Definición y especificación.

Conjunto de datos que conforman un concepto y una colección de operaciones definidas sobre el mismo.

## Clases y Objetos.

El estado y la funcionalidad de un objeto están definidos en una clase, por lo tanto un objeto es una instancia de la clase en la cual se define su estructura y comportamiento. Cada objeto puede ser accedido o modificado a través de las operaciones definidas por su clase

## Métodos y variables.

Un objeto es una instancia de una clase, donde los componentes pueden ser funciones o datos, denominados respectivamente métodos y atributos, y, se puede restringir la visibilidad de dichos componentes.

La estructura interna del TDA contiene los atributos, dichos atributos son variables como moldes que luego se llenaran con algún valor (según el tipo que está especifique) para que luego sean utilizadas por los métodos

## Arreglos de clases.

claseNombre array[]  = new claseNombre[X];

## Flujos de datos y archivos de entrada y de salida.

No entra en el final (creo) pero es cuando se puede obtener datos para trabajar desde archivos externos (creox2)

## Diseño modularizado de programas con los tipos de datos correctos.

???

# Unidad 3- ANÁLISIS DE ALGORITMOS

## ¿Cuáles son los algoritmos óptimos para cada tipo de problema?

La elección del algoritmo adecuado depende del problema específico que estés tratando de resolver

- El bucle `for` es útil cuando sabes de antemano cuántas veces debes repetir una tarea. Es especialmente útil para recorrer arreglos o listas con un índice o contador.
- El bucle `while` es adecuado cuando no sabes cuántas veces debes repetir una tarea de antemano y la repetición depende de una condición. Ingresa 0 o más veces.
- El bucle `do-while` es similar al bucle `while`, pero garantiza que el bloque de código se ejecute al menos una vez antes de verificar la condición.
- La estructura `if-else` se utiliza para tomar decisiones basadas en una condición.
- La sentencia `switch` se utiliza para tomar decisiones basadas en el valor de una expresión.

## Concepto de recursión.

Técnica de programación que nos permite que un bloque de instrucciones se ejecute “n” veces sin utilizar estructuras repetitivas, por lo que en ocasiones reemplaza a dichas estructuras repetitivas.

### Características fundamentales de la recursividad.

- Al menos un caso trivial o base, es decir, que no vuelva a invocarse y que se activa cuando se cumple cierta condición.
- Un caso general que es el que vuelve a invocar al algoritmo con un subcaso del mismo garantizando llegar al caso base o trivial.

## Algoritmos recursivos.

### Iteración y recursividad.

La ejecución de un algoritmo recursivo implica que una nueva versión del algoritmo comienza a ejecutarse, por lo tanto debe organizar la manera de guardar determinada información de cada llamada.

### Manejo interno de la recursividad.

El compilador gestiona esta tarea a través de una estructura tipo pila: lo último que entra es lo primero que sale.

Se crea una pila para cada argumento, una pila para cada variable local, y una pila para la dirección de retorno en la que se coloca la próxima invocación al algoritmo.

La estructura de pila que utiliza la memoria es la misma que utilizamos al construir la traza.

# Algoritmos vuelta atrás.

## Tipos de Algoritmos.

- **Simple:** Aquella en cuya definición sólo aparece una llamada recursiva.

- **Múltiple:** Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de hacer de forma iterativa

- **Anidada:** En algunos de los argumentos de la llamada recursiva hay una nueva llamada a sí misma

## Técnicas de diseño de algoritmos.

La recursividad es una herramienta que puede ayudar a reducir la complejidad de un programa ocultando algunos detalles de la implementación

Se recomienda en general, utilizar la solución no recursiva cuando en el planteo no se reconocen ventajas en lo que se refiere al uso de los resultados parciales. Aunque cuando la complejidad de la solución no recursiva se torna inmanejable, es preferible utilizar un algoritmo recursivo como en los algoritmos de recorrido de árboles y grafos.

# Análisis de tiempos y espacios de ejecución.

## Orden, análisis de orden.

O(1) Complejidad constante
O(log n) Complejidad logarítmica
O(n) Complejidad lineal
O(n log n)
O(n2 ) Complejidad cuadrática
O(n3 ) Complejidad cúbica
O(nk ) Complejidad polinómica
(k >3) O (kn ) Complejidad exponencial

## Trazas.

Resolucion de cómo calcular los órdenes en base a las estructuras que se evalúen

### Secuencias de instrucciones.

Asignaciones, accesos, y operaciones matemáticas valen tiene de valor 1 tiempo

### Alternativa simple.

```
Si (cond) ENTONCES
     …...S1
FIN SI
```

Requiere: Tiempo Si = $t_{cond} + t_1$

### Alternativa compuesta.

```
Si (cond) ENTONCES
    S1
SINO
    S2
FIN SI
```

Requiere: Tiempo Si/Sino = $t_{cond} + max(t_1, t_2)$

### Iterativas tipo PARA.

```
PARA i ← 1 HASTA n HACER
    S1
FIN PARA
```

Requiere:Tiempo para = $t_{ini} + cantIt *(t_{cond} + t_{int} + t_{inc})+ t_{cond}$

***cantIt*** = *(valor Final – valor inicial) /paso*

### Iterativas tipo MIENTRAS.

```
MIENTRAS cond HACER
    S1
FIN MIENTRAS
```

Requiere:Tiempo mientras = $cantIt *(t_{cond} + t_{int} )+ t_{cond}$

### Iterativas tipo REPETIR HASTA.

```
REPETIR
     S2
HASTA cond
```

Requiere:Tiempo REPETIR HASTA = $+ cantIt *(t_{int} + t_{cond})$

## Algoritmos fundamentales: ordenamiento y búsqueda.

Algoritmos numéricos y algoritmos combinatorios.
Generación e implementación modularizada de los algoritmos más eficientes con tipos de datos.
Algoritmos fundamentales de ordenamiento y búsqueda.
Algoritmos numéricos y algoritmos combinatorios.

**Ordenamiento de Burbuja:**
**Descripción:** Arrastra los elementos más grandes al final haciendo intercambios.

- Comienza con dos bucles anidados que recorren el arreglo.
- Compara elementos adyacentes y los intercambia si están en el orden incorrecto.
- Repite este proceso hasta que no haya más intercambios que realizar.

**Ordenamiento de Burbuja Mejorado:**
**Descripción:** Similar al Burbuja pero con una optimización. Se usa una bandera **(ordenado)** y que no vuelve a comparar los últimos elementos que ya se ordenaron anteriormente.

- Utiliza un bucle **while** y un marcador **booleano "ordenado"** para verificar si el arreglo ya está ordenado.
- Dentro del bucle while, utiliza un bucle for para comparar elementos adyacentes y realizar intercambios si es necesario.

**Ordenamiento por Inserción:**
**Descripción:** Este algoritmo divide el arreglo en dos partes: una parte ordenada y una parte desordenada. Luego, toma elementos de la parte desordenada e los inserta en la parte ordenada en su posición adecuada, desplazando los elementos mayores según sea necesario.

- Utiliza un bucle for para recorrer el arreglo desde el segundo elemento hasta el final.
- En cada iteración, toma el elemento actual y lo compara con los elementos anteriores para encontrar la posición correcta.
- Desplaza los elementos mayores que el elemento actual una posición a la derecha para hacer espacio e inserta el elemento en la posición correcta.

**Ordenamiento por Selección:**
**Descripción:** Busca el elemento más pequeño en el arreglo y lo coloca en la posición adecuada. Luego, busca el siguiente elemento más pequeño y lo coloca en la siguiente posición, y así sucesivamente, hasta que todo el arreglo está ordenado.

- Utiliza un bucle for para recorrer el arreglo.
- En cada iteración, busca el elemento más pequeño en el segmento no ordenado del arreglo.
- Intercambia el elemento más pequeño con el elemento en la posición actual del bucle for.
- Repite este proceso hasta que todo el arreglo esté ordenado.

Tiempo de ejecución (Mejor y Peor caso) **Burbuja, Burbuja Mejorado, Inserción**

- Mejor caso: O(n) - Cuando el arreglo ya está ordenado.
- Peor caso: O(n^2) - Cuando el arreglo está en orden inverso.

Tiempo ejecución (Mejor y Peor caso) **Selección**

- Mejor caso: O(n^2)
- Peor caso: O(n^2)

**QuickSort:**
**Descripción:** Selecciona un elemento como pivote y participa el arreglo en dos subarreglos: uno con elementos menores que el pivote y otro con elementos mayores. Luego, aplica recursivamente QuickSort a los subarreglos.

**MergeSort:**
**Descripción:** Divide recursivamente el arreglo en mitades hasta que se obtienen subarreglos de un solo elemento. Luego, combina los subarreglos de manera ordenada para obtener un nuevo arreglo ordenado.

**HeapSort:**
**Descripción:** Convierte el arreglo en un montículo binario, que es una estructura de datos especial donde el elemento máximo (o mínimo) se encuentra en la raíz. Luego, extrae el elemento máximo (que es el primer elemento del montículo) y lo coloca en la posición final del arreglo ordenado.

Tiempo de ejecución (Mejor y Peor caso) **QuickSort**

- Mejor caso: O(n log n) - Esto ocurre cuando el pivote divide el arreglo en dos mitades aproximadamente iguales en cada iteración.
- Peor caso: O(n^2) - Cuando el pivote elegido siempre es el elemento más pequeño o más grande, lo que resulta en divisiones desequilibradas.

Tiempo de ejecución (Mejor y Peor caso) **MergeSort, HeapSort**
- Divide el arreglo en mitades, lo que garantiza el mismo rendimiento **(MergeSort)**
- La estructura del montículo binario y su organización garantiza un mismo rendimiento **(HeapSort)**

- Mejor caso: O(n log n)
- Peor caso: O(n log n)