# ESTRUCTURA DE DATOS

Tecnológico Nacional de México
Instituto Tecnológico de Culiacán
Ingeniería Tecnología de la Información y Comunicaciones

*Mtra. María Lourdes Armenta Lindoro*

*Culiacán, Sinaloa. AGOSTO de 2022*

# TEMARIO

## Unidad I. Introducción a la Estructura de Datos

- 1.1 Clasificación de las estructuras de datos.
- 1.2 Tipos de datos abstractos (TDA).
- 1.3 Ejemplos de TDA?s.
- 1.4 Manejo de memoria. 1.4.1 Memoria estática. 1.4.2 Memoria dinámica.
- 1.5 Análisis de algoritmos. 1.5.1 Complejidad en el tiempo. 1.5.2 Complejidad en el espacio. 1.5.3 Eficiencia de los algoritmos

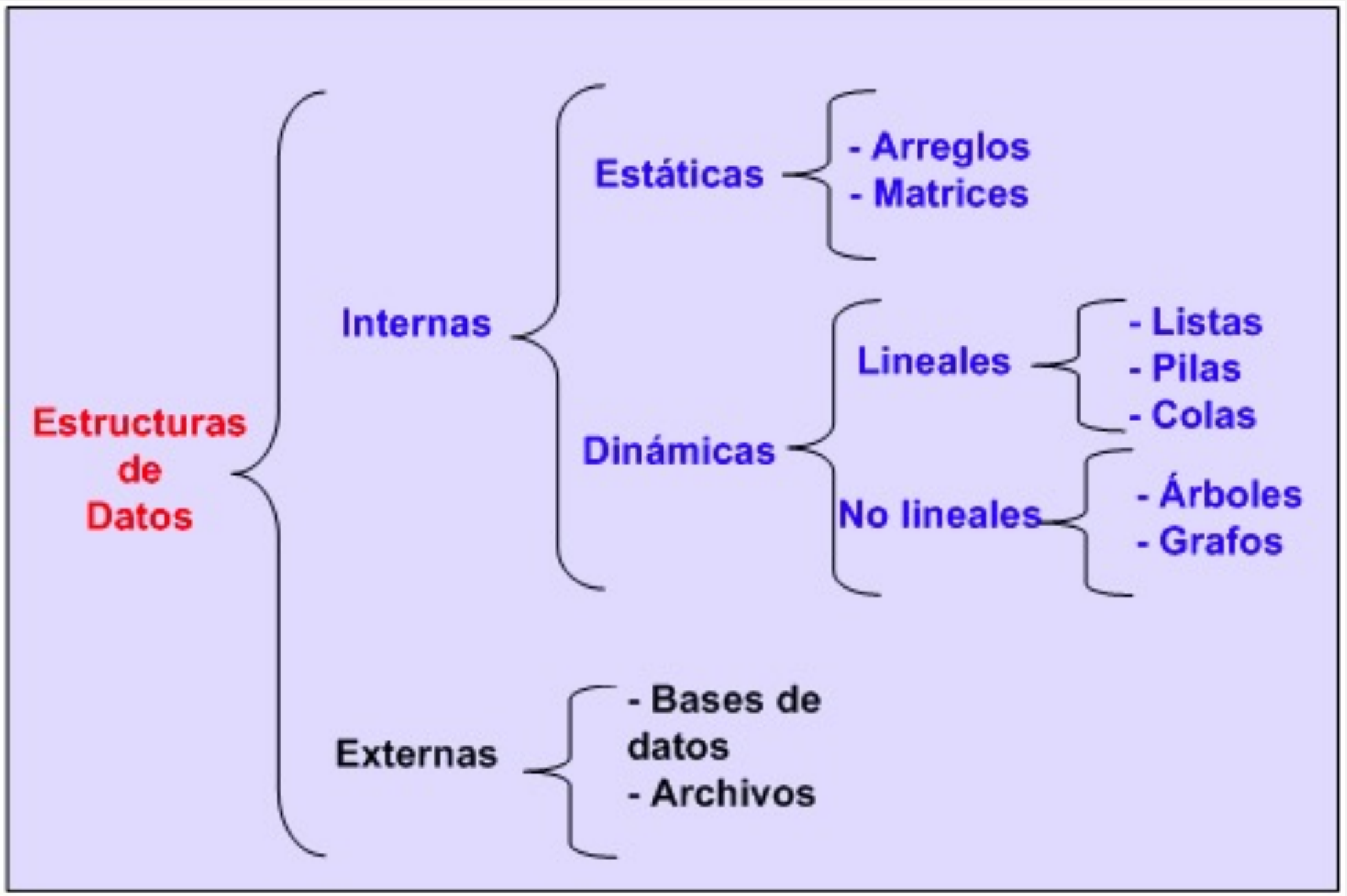
.

# UNIDAD INTRODUCCIÓN A LA ESTRUCTURA DE DATOS

- "Hoy en día el Internet, sistemas operativos, sistemas cliente-servidor, simuladores, videojuegos… todos ellos tienen un común denominador, requieren del uso intensivo de *estructuras de datos* para almacenar, manipular y organizar la información con la que trabajan."
- "El crecimiento vertiginoso del área de desarrollo del software, ha llevado al punto en que las estructuras de datos más comunes ya se encuentran como elementos nativos de algunos lenguajes de programación. Sin embargo, esto no implica la necesidad de conocerlas. Es preciso en su funcionamiento, sus principales características, sus ventajas y desventajas, sus áreas típicas de aplicación…."

# 1.1 CLASIFICACIÓN DE LAS ESTRUCTURAS DE DATOS

- 1.- Tipos primitivos de datos
  - Enteros
  - Punto flotante
  - Carácter
  - Lógico
- 2.- Tipos de datos compuestos y agregadados
  - Arreglos
  - Secuencia o cadena
  - Registro


Estructuras de Datos
Internas
Estáticas
- Arreglos
- Matrices
Dinámicas
Lineales
- Listas
- Pilas
- Colas
No lineales
- Árboles
- Grafos
Externas
- Bases de datos
- Archivos

# LA NECESIDAD DE ESTRUCTURA DE DATOS

Las **estructuras de datos** son útiles **porque** nos permiten tener una batería de herramientas para solucionar ciertos tipos de problemas. Además, nos permiten hacer un software más eficiente optimizando recursos

1. Los datos con los valores que manejamos en la resolución de problemas, los valores de entrada, proceso y salida
2. Tipos de datos, es un conjunto de valores y un conjunto de operaciones definidos por esos valores. Ejemplo: flotantes, enteross, cadenas de caracteres, etc.
3. Tipos de Datos Abstractos, extienden la función de un tipo de datos, ocultando la implementación de las operaciones definidas por el usuario. Esta capacidad de ocultamiento permite desarrollar software reutilizable y extensible.

# ¿QUÉ ES UNA ESTRUCTURA DE DATOS?

Es Cualquier colección o grupo de datos organizados que tengan asociados un conjunto de operaciones para poder manipularlos.

# ABSTRACCIÓN DE DATOS

- ¿Qué es una abstracción?

es un proceso mental mediante el cual se extraen los rangos esenciales de algo para representarlos por medio de un lenguaje gráfico o escrito.

- ¿Por qué es importante la abstración?

es un conjunto de o una acción subjetiva y creativa, Esto es depende del contexto psicológico de la persona que la realiza.

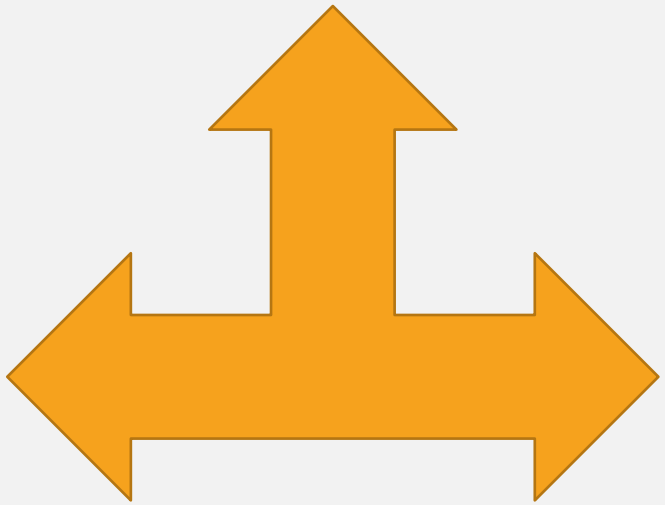
- ¿Qué es una abstracción de datos?

es una técnica O metodología que permite diseñar estructuras de datos

# 1.2 TIPOS DE DATOS ABSTRACTOS (TDA).

2. LOS TIPOS DE DATOS

1.LOS DATOS



3. LOS TIPOS DE ABSTRACTOS DE DATOS

**DIFERENCIA DE ABSTRACCIÓN ENTRE**

# DEFINICIÓN DE UN TAD

- Es un tipo de datos definidos por el programador. Los TAD están formados por datos (estructuras de datos) y las operaciones (procedimientos o funciones) que se realizan sobre esos datos.
- El término "tipo abstracto de dato", se refiere al concepto matemático básico que define el tipo de datos.
- Un TAD es una guía útil para los programadores que quieran usar los tipos de datos en forma correcta.
- Un TAD es independiente del lenguaje de programación aunque facilita los métodos para su desarrollo.

# MÉTODOS PARA ESPECIFICAR UN TAD

El objetivo es describir el comportamiento:

1. Métodos formales
2. Métodos no formales
3. Métodos semiformales

# METODOS FORMALES

- Se basan en teoría de conjuntos, el cálculo de lamba y la lógica de primer orden.
- Un ejemplo de estos métodos es el lenguaje Z.
- En Z se definen construcciones denominadas esquemas para describir el espacio de estado de sistema  y las operaciones que sobre el mismo se efectúan. En los esquemas se declaran variables y predicados que afectan los valores de las variables declaradas.
- En problema de Z es que es muy complejo en su aprendizaje por eso no es muy común usar estos métodos

# METODOS NO FORMALES

- Utilizan lenguaje natural para especifica un TAD

# MÉTODOS SEMIFORMALES

- Son un punto intermedio entre el formal y el no formal
- Utilizan una sintaxis similar a la lenguaje C
- Son los mas utilizados para especificar un TAD
- Existen diferentes forma de representar los métodos semiformales

# EJEMPLO I MÉTODO SEMIFORMAL (CREAR UN TAD RATIONAL)

```
1.  abstract typedef<integer,integer> RATIONAL;
2.  condition RATIONAL[1]!=0;
3.
4.  /* definicion del operador */
5.  abstract RATIONAL makerational(a,b)
6.  int a,b;
7.  precondition b!=0;
8.  postcondition makerational[0]==a;
9.  makerational[1]==b;
10.
11. /* written a+b */
12. abstract RATIONAL add(a,b)
13. RATIONAL a,b;
14. postcondition add[1]==a[1]*b[1];
15. add[0]=a[0]*b[1]+b[0]*a[1];
16.
17. /* mutl a*b*/
18. abstract RATIONAL mult(a,b)
19. RATIONAL a,b;
20. postcondition mult[0]==a[0]*b[0];
21. mult[1]==a[1]*b[1];
22.
23. /* equal */
24. abstract equal(a,b)
25. RATIONAL a,b;
26. postcondition equal==(a[0]*b[1]==b[0]*a[1]);
```

**El operador mult(a.b)** efectúa la multiplicación de dos datos racionales. Su postcondición indica que el racional resultante se calcula multiplicando numeradores con numeradores y denominadores con denominadores

a*b= $\frac{a[0]}{a[1]} * \frac{b[0]}{b[1]} =$

**El operador add(a,b)** efectúa la suma de dos datos racionales. Su postcondición indica que el rational resultante es la operación:

a+b= $\frac{a[0]}{a[1]} + \frac{b[0]}{b[1]} = \frac{a[0]*b[1]+b[0]*a[1]}{a[1]*b[1]}$

**El operador equals(a,b)** indica si dos numeros racionales con iguales. Sus componentes son iguales cuando se reproducen a sus términos mínimos. Una forma de probar la igualdad racional es verificar si los productos cruzados (esto es numerador de uno por el denominador el otro) som iguales. Así los números ½, 2/4, 3/6, 18/36 son iguales

$\frac{1}{2} = \frac{3}{6} = 1 * 6 = 3 * 2$

Esto se expresa en la postcondición:

a=b = $\frac{a[0]}{a[1]} = \frac{b[0]}{b[1]}$ = a[0]*b[1]=b[0]*a[1]

# EJEMPLO 2 DEFINIR TAD MATRIZ DE VALORES DE N FILAS POR M COLUMMAS)

REPRESENTACIÓN GRÁFICA DE LA MATRIZ

| | | | | | | |
|---|---|---|---|---|---|---|
| $X_{0,0}$ | $X_{0,1}$ | $X_{0,2}$ | … | $X_{0,K}$ | … | $X_{0,M-1}$ |
| $X_{1,0}$ | | | | | | |
| $X_{2,0}$ | | | | | | |
| ….. | | | | | | |
| $X_{1,0}$ | | | | $X_{I,K}$ | | |
| … | | | | | | |
| $X_{N-1,0}$ | | | | | | $X_{N-1,M-1}$ |

# ESTA VEZ SE UTILIZARÁ UNA ESPECIFICACIÓN SEMIFORMAL MEDIANTE ESQUEMA

1. TAD <nombre>;
2. <Objeto Abstracto>
3. <Invariante del TAD>
4. <Operaciones>

```
1. %--------------------
2. %TAD MATRIZ
3. %--------------------
4. {inv: N>0,M>0}
5. crearMat: int,int
6. asignarMat: Matriz, int, int, int
7. infoMat. Matriz, int, int
8. filasMat: Matriz
9. columnasMat: Matriz
```

```
11. Matriz crearMat(int fil, int col)
12. /*construye y retorna uma matriz dedimensión [0..fil-
    1,0…col-1] inicio en 0*/
13. { pre:  fil>0, col>0}
14. { post: crearMat es una matriz de dime nsion [0…fil-
    1,0..col-1]}
15. void asignarMat(Matriz mat, inf fil,int col, int val)
16. /*Asigna a la casilla de coordenadas [fil,col] el valor val */
17. { pre:  0<=fil <N, 0=<col<M}
18. { post: mat[fil,col] = val}
19. int infoMat(Matriz mat, int fil, int col)
20. /*Retorna el contenido de la casillas de coordena [fil,col]
21. */
22. { pre:  0<=fil <N, 0 = <col<M}
23. { post: mat[fil,col] = mat[fil,col]}

24. int filasMat (Matriz mat)
25. /*Retorna el númerp de filas de la matriz*/
26. {post : filasMat = N}

27. int columnasMat( Matriz mat)
28. /* Rerotna el número de columnas de la matriz*/
29. { post: columnasMat = M}
```

# EJEMPLO 3 ESPECIFICACIÓN DE UN TAD DE MANERA NO FORMAL

Tipo de Datos: MATRIZ

Operaciones:

- crearMat (int N, int M): Construye y retorna una matriz de dimensión NXM
- asignarMat(Matriz mat, int fil, int col, int val): Asigna a la casilla de coordenadas [fil,col] el valor val
- infoMat(Matriz mat, int fil, int col): Retorna el contenido de la casilla de coordenadas [fil, col]
- filasMat( Matriz mat): Retorna el número de filas de la matriz
- columnasMat (Matriz mat): Retorna el número de columnas de la matriz.

# CLASIFICACIÓN DE LAS OPERACIONES DE UN TAD

1. **Constructoras**. Es la operación encargada de crear elementos del TAD. En el caso típico, es la encargada de crear el objeto abstracto mas simple.
2. **Modificadora**. Es la operación que puede alterar el estado de un elemento del TAD. Su misión es simular una reacción del objeto
3. **Analizadora**. Es una operación que no altera el estado del objeto, sino que tiene como misión consultar su estado y retomar algún tipo de información.

# EN QUE CONSISTE LA ESPECIFICACIÓN LÓGICA DE UN TDA

Es un documento en el que se plasma la abstracción

Este documento puede ser un mapa o plano mediante el cual se implementará la estructura de datos y en el qe se definen las reglas que podrán usarse al aplicarse el TDA.

Esta es una fundamentación matemática, pero también se puede utilizar en lenguaje natural.

Este documento consiste en: 1) Elementos que formaran la estructura de datos 2) Tipos de organización en que se guardarán los elementos 3) Dominio de la estructura y 4) Descripción de la estructura

# 1.- ELEMENTOS CONFORMAN LA ESTRUCTURA DE DATOS

- Aquí se descrinen el tipo de datos individuales que guardará la estructura. Por ejemplo, números enteros, caracteres, fechas, registros de los datos de un empleado, etc

# 2.- TIPO DE ORGANIZACIÓN EN QUE SE GUARDARAN LOS ELEMENTOS

Existen 4 tipo: 1) Lineal 2) Jerárquica 3) Red y 4) Sin relación.

- **Lineal:** Si hay una relación de uno a uno entre los elementos
- **Jerárquica:** Si hay una relación de uno a muchos entre los elementos
- **Red.** Si hay una relación de muchos a muchos entre los elementos
- **Sin relación:** Si no hay relaciones entre los elementos.

# 3. DOMINIO DE LA ESTRUCTURA

- Es un punto opcional, y en el se describirá la capacidad de la estructura en cuanto al rango posible de datos por guardar.

# 4.- DESCRIPCIÓN DE LAS OPERACIONES DE LA ESTRUCTURA

- Cada operación relacionada con la estructura debe describirse con los siguiente puntos:
  - Nombre de la operación
  - Descripción breve de la utilidad
  - Datos de entrada a la operación
  - Datos que genera como salida la operación
  - Precondición: condición que deberá cumplirse antes de utilizar la operación para que se realice sin problemas.
  - Postcondición: Condición en la que queda el TDA después de ejecutar la operación.
- EJEMPLO:

# ESPECIFICACIÓN LÓGICA DEL TIPO DE DATO ABSTRACTO (TDA) **CADENA**

ELEMENTOS: Todos los caracteres alfabéticos (LETRAS MAYÚSCULAS Y MUNÚSCULAS), caracteres numéricos y caracteres especiales.

ESTRUCTURA: Hay una relación lineal entre los caracteres

DOMINIO: existen entre 0 y 80 caracteres en cada valor del TDA CADENA. El dominio será todas aquellas secuencias de caracteres que cumplan con las reglas.

OPERACIONES

**BORRARINICIO**

Utilidad:  Sirve para eliminar le primer carácter de una cadena

Entrada. Cadena S sobre la que se desea eliminar el primer carácter

Salida. El carácter más a la izquierda de la cadena S y la cadena S modificada

Precondición: La cantidad de caracteres es mayor que cero

Postcondición: La cadena S tiene todos los caracteres, menos el primero

**AGREGARFINAL**

# ESPECIFICACIÓN LÓGICA DEL TIPO DE DATO ABSTRACTO (TDA) **CADENA**

**AGREGARFINAL**

Utilidad: Sirve para agregar un carácter al final de una cadena

Entrada: Cadena S y el carácter L, que se añadirá al final de la cadena

Salida: Cadena S modificada

Precondición: La cantidad de caracteres S es menor que 80

Postcondición: La cadena S tiene el carácter L que queda al extremo derecho de la cadena

**VACIA:**

Utilidad: Sirve para verificar si una cadena esta vacía o no

Entrada: Cadena S que se verificará

Salida: VERDADERO si la cadena S no tiene caracteres y FALSO en caso contrario

Precondición: Ninguna

Postcondición: Ninguna

# ESPECIFICACIÓN LÓGICA DEL TIPO DE DATO ABSTRACTO (TDA) **CADENA**

**LLENA:**

Utilidad: Sirve para verificar si una cadena esta llena o no

Entrada: Cadena S que se verificará

Salida: VERDADERO si la cadena S contiene ya 80 caracteres y FALSO en caso contrario

Precondición: Ninguna

Postcondición: Ninguna

**INVIERTA:**

Utilidad: Sirve para invertir el orden de los caracteres en una cadena

Entrada: Cadena S a la que se desea invertir

Salida: Cadena S modificada

Precondición: Ninguna

Postcondición: La secuencia de caracteres en la cadena S se invierte, de forma que el primer carácter toma el lugar del último, y el segundo el lugar del penúltimo y así sucesivamente.

# NIVELES DE ABSTRACCIÓN DE DATOS

1. Nivel Lógico o abstracto
2. Nivel Físico o de Implementación
3. Nivel de Aplicación o Uso

# ¿CÓMO DISTINGUIR LOS NIVELES DE ABSTRACCIÓN? Y ¿QUÉ VENTAJA OFRECE UTILIZAR LA TÉCNICA DE ABSTRACCIÓN DE DATOS?

- Ver Ejemplo: En cierta aplicación científica se requiere calcular el factorial de un número específico. La función requiere aplicar factorial sobre un número es la multiplicación de todos los números.
- Sin embarga la capacidad de los enteros en el lenguaje esta limitada.
- Por lo tanto se requiere de un tipo de dato con una capacidad para almacenar cualquier número entero, sin importar que tan grande sea, si el lenguaje no dispone de un tipo de dato, se puede diseñar un TAD para números enteros grandes, el cual a su vez será una estructura de datos.

# ESPECIFICACIÓN LÓGICA DEL TIPO DE DATO ABSTRACTO (TDA) **NUMEROTE**

ELEMENTOS: UN Número se compone de dígitos

TIPO DE ORGANIZACIÓN: Los dígitos se organizan de manera líneal.

DOMINIO: se presente que un numerote pueda contener cualquier cantidad de dígitos  bajo cualquier combinación. Sin embargo, se puede limitar a la capacitad de mil dígitos.

OPERACIONES: las mismas operaciones que posee el tipo de dato entero

OPERACIONES:

**SUMA**

UTILIDAD: sirve para sumar dos números

ENTRADA: dos números

SALIDA: un número que guarda la suma de los dos número de entrada

PRECONDICIÓN: Ninguna

POSTCONDICIÓN: El Numerote de salida contiene la suma artimética de dos número de entrada

# ESPECIFICACIÓN LÓGICA DEL TIPO DE DATO ABSTRACTO (TDA) **NUMERO**

**DESPLIEGE**

UTILIDAD: sirve para desplegar en pantalla un Numerote

ENTRADA: Numerote a desplegar

SALIDA: Ninguna (Observar en Pantalla)

PRECONDICIÓN: Ninguna

POSTCONDICIÓN: Ninguna

# ESPECIFICACIÓN DE LOS TAD

El Objetivo es describir el comportamiento del TAD: consta de dos partes, la descripción matemática del conjunto de datos y las operaciones definidas en ciertos elementos de ese conjunto de datos.

La especificación del TAD puede tener un enfoque *informal*, que describe los datos y las operaciones relacionadas en *lenguaje natural*. Otro enfoque mas riguroso, la especificación formal, supone suministrar un conjunto de *axiomas* que describen las operaciones en su aspecto *sintáctico* y *semántico*.

# ESPECIFICACIÓN INFORMAL DE UN TAD

Consta de dos partes:

- Detallar en los datos del tipo los valores que puede tomar
- Describir las operaciones relacionándolas con los datos

# ESPECIFICACIÓN FORMAL DE UN TAD

**Proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir una parte de sintaxis, en cuanto a los tipos de argumentos y al tipo de resultado, y una parte de semática, donde se detalla la expresión formal ha de ser bastante potente para que cumpla con el objetivo de verificar la corrección de la implementación del TAD**

El esquema que sigue consta de una cabecera con el nombre del TAD y los datos:

TAD *nombre del tipo* (valores que toma los datos del tipo)

Sintaxis:

Operación(Tipo de Argumento1, Tipo de Argumento2,….) => Tipo de Resultado

Semántica.

Operación(valores particulares del argumento) => expresión resultado

# ESPECIFICACIÓN FORMAL DE UN TAD

TAD *Conjunto*(colección de elementos sin duplicidad, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones)

**Sintaxis:**

*ConjuntoVacio. => Conjunto

*Añadir(Conjunto,Elemento) = > Conjunto

Retirar(Conjunto,Elemento) = > Conjunto

Pertenece(Conjunto, Elemento) => boolean

EsVacia(Conjunto) = > boolean

Cardinal(Conjunto) => entero

Union(Conjunto, Conjunto) = > Conjunto

*CONSTRUCTORES.-

# ESPECIFICACIÓN FORMAL DE UN TAD

**Semántica**

Para todo elemento e1,e2 que pertenece a un Elemento y Para todo C,D que pertenece a un Conjunto

Añadir(Anadir(C,e1),e1) => Añadir(C,e1)

Añadir(Añadir(C,e1),e2) = >Añadir(Añadir(C,e2),e1)

Retirar(Conjuntovacio,  e1) =>Conjuntovacio

Retirar(Añadir(C,e1),e2) = > **si** e1 = e2 **entonces** Retirar(C,e2)

**si no** Añadir(Retirar(C,e2);e2)

Pertenece(Conjuntovacio,e1) => falso

Pertenece(Añadir(C,e2),e1) => si e1 = e2 entonces cierto

sino pertenece(C,e1)

EsVacio(Conjuntovacio) => cierto

EsVacio(Añadir(C,e1)) => falso

Cardinal(Conjuntovacio) =>Cero

# ESPECIFICACIÓN FORMAL DE UN TAD

**Semántica**

Cardinal(Añadir(C,e1)) => **si** Pertenece(C,e1) **entonces** Cardinal(C)

**sino** 1+Cardinal(C)

Union(Conjuntovacio,ConjuntoVacio) => ConjuntoVacio

Union(Conjuntovacio,Añadir(C,e1)) => Añadir(C,e1)

Union(Añadir(C,e1),D) => Añadir(Union(C,D),e1)

Mostrar(Conjunto) => Cadena

# IMPLEMENTACIÓN DEL TAD CONJUNTO

```
public class Conjunto {
static int M = 20; //Aumento de la capacidad
private Object  cto[];
private int cardinal;
private int capacidad;
//Operaciones
public Conjunto()
{
cto = new Object[M];
cardinal = 0;
capacidad = M;
}

//determina si el conjunto esta vacio
public boolean esVacio()
{
return (cardinal == 0);
}
```

```
//añade un elemento si no está en el conjunto
public void añadir(Object elemento)
{
if (!pertenece(elemento))
{
/*Verifica si hay posiciones libres
* en caso contrario amplia el conjunto
*/
if(cardinal ==capacidad)
{
Object [] nuevoCto;
nuevoCto = new Object[capacidad + M];
for(int k = 0;k<capacidad;k++)
nuevoCto[k] = cto[k];
capacidad += M;
cto = nuevoCto;
System.gc(); // devuelve la memoria no referenciada
}
cto[cardinal++] = elemento;
}
}
```

```
//quita el elemento del conjunto
public void retirar(Object elemento)
{
if(pertenece(elemento))
{
int k = 0;
while (!cto[k].equals(elemento))
k++;
/* desde el elemento k hasta la última posición
* mueve los elementos una posición a la izquierta
*/
for(;k<cardinal;k++)
cto[k] = cto[k+1];
cardinal--;
}
}
```

# IMPLEMENTACIÓN DEL TAD CONJUNTO

```
//busca si un elemento pertenece al conjunto
public boolean pertenece(Object elemento)
{
int k = 0;
boolean encontrado = false;
while (k<cardinal && !encontrado)
{
encontrado = cto[k].equals(elemento);
k++;
}
return encontrado;
}
//devuelve el número de elementos
public int cardinal()
{
return this.cardinal;
}
```

```
//Operacion union de dos conjuntos
public Conjunto union(Conjunto c2)
{
Conjunto u = new Conjunto();
// primero copia el primer operando de la union
for(int k = 0;k<cardinal;k++)
u.cto[k] = cto[k];
u.cardinal = cardinal;
//añade los elementos de c2 no incluidos
for (int k=0;k<c2.cardinal;k++)
u.añadir(c2.cto[k]);
return u;
}
```

```
public Object elemento(int n)
throws Exception
{
if (n<=cardinal)
return cto[--n];
else
throw new Exception("Fuera de Rango");
}
}
```

# EJERCICIO IMPLEMENTACIÓN DE UNA CLASE EN JAVA

En un sistema bancario, las cuentas de cheques de los clientes pueden considerarse objetos. Todos ellos pertenecerían a una clase llamada Cuente_de_cheques que define, para cada instancia, los siguientes datos:

**Nombre de la cuenta**

**Nombre del cliente**

**Sucursal donde se abrió la cuenta**

**Saldo de la cuenta**

Entre los métodos que puedes asociarse a los objetos de la clase Cuenta_de_Cheques:

**Alta_de_cuenta.** Se ejecutaría al crear un objeto de la clase Cuenta_de_cheques, e implica solicitar del teclado los datos correspondientes para guardarlos en losnatributos del objeto y colocar el valor de cero al saldo actual

**Deposito.** Agrega la cantidad específica al saldo de la cuenta

**Retiro.-** Quita el saldo de la cuenta la cantidad especificada, validando que sea posible retirar esa cantidad.

**Muestra_saldo.** Este método muestra en pantalla el saldo de la cuenta correspondiente.

# INDEPENDENCIA DE DATOS Y EL OCULTAMIENTO DE INFORMACIÓN

# TIPOS DE DATOS ABSTRACTOS (TDA)

La *abstracción de datos* es la técnica para inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, faciliten la escritura del programa.

La técnica de abstracción de datos es una técnica potente de propósito general que, cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles.

# REPRESENTACIÓN DE ESTRUCTURAS DE DATOS

- Objetivos
  - Entender qué es la representación física de una estructura de datos
  - Describir las características que distinguen a la representación por posiciones y a la representación por ligas o encadenamiento.
  - Distinguir la diferencia entre una dirección de memoria física o real y una dirección lógica o relativa (arreglos)
  - Entender cuáles son las diferencias entre el uso de la memoria estática Y la memoria dinámica.
  - Comprender la ventaja de utilizar la memoria dinámica es aplicaciones de software.
  - Aplicar correctamente la regla para utilizar a la memoria dinámica.
  - Ejemplificar los conceptos con la programación del nivel físico de un TDA utilizando la POO y los atributos dinámicos.

# TIPOS DE DATOS ABSTRACTOS (TDA)

En términos más precisos, se indica que un tipo de dato definible por el usuario se denomina tipo abstracto de dato (TAD) si:

- Existe una construcción del lenguaje que le permite asociar la representación de los datos con las operaciones que lo manipulan;

- La representación del nuevo tipo de dato está oculta de las unidades de programa que lo utilizan.

# TIPOS DE DATOS ABSTRACTOS (TDA)

Las clases de Java o de C++ cumplen las dos condiciones: agrupan los datos junto a las operaciones, y su representación queda oculta de otras clases.

# TIPOS DE DATOS ABSTRACTOS (TDA)

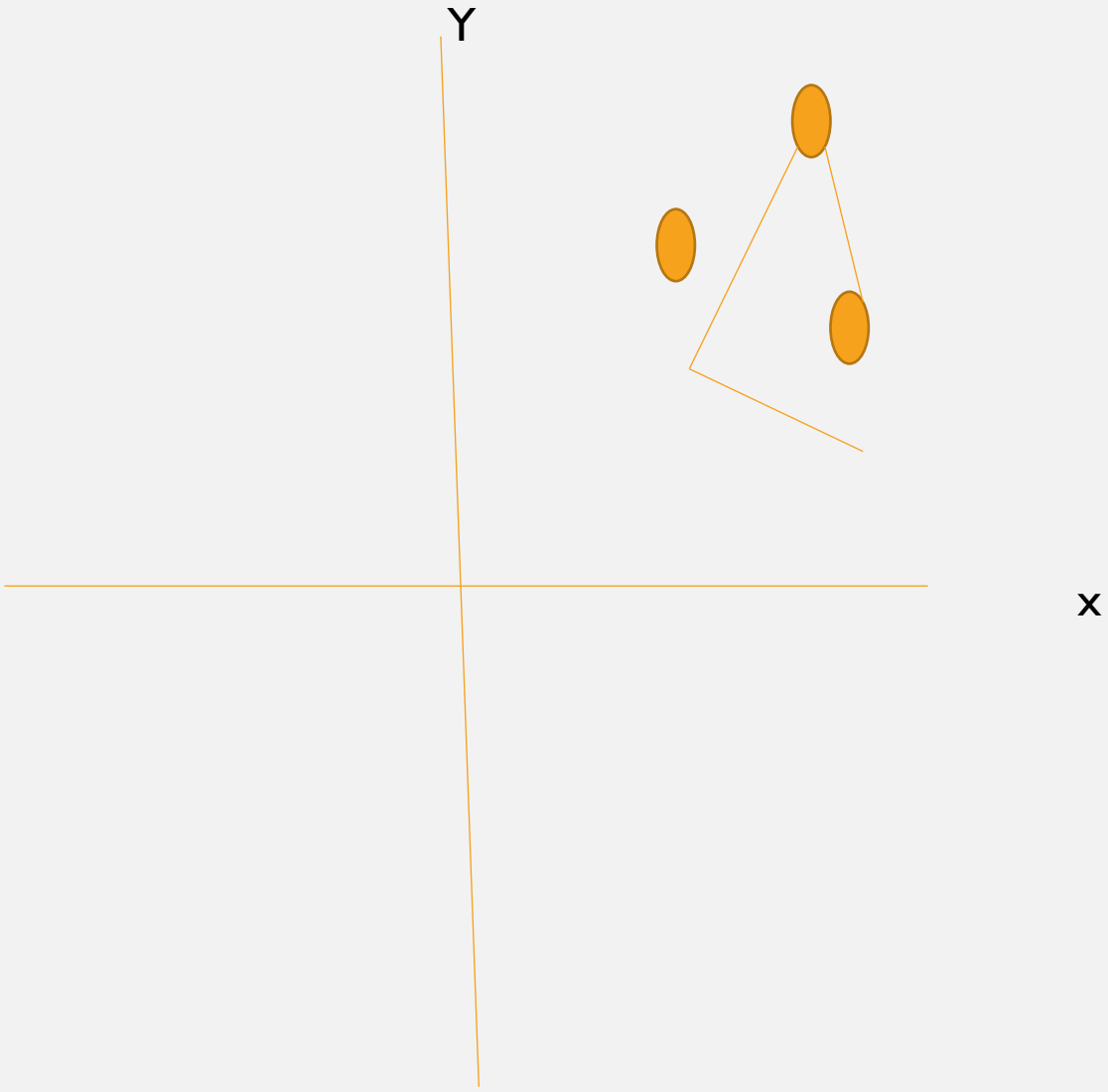
Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina tipo abstracto de datos (TAD) para diferenciarlo del tipo fundamental (predefinido) de datos.

Por ejemplo, en Java, el tipo Punto, que representa las coordenadas x e y de un sistema de coordenadas rectangulares, no existe. Sin embargo, es posible implementar el tipo abstracto de datos, considerando los valores que se almacenan en las variables y qué operaciones están disponibles para manipular estas variables.

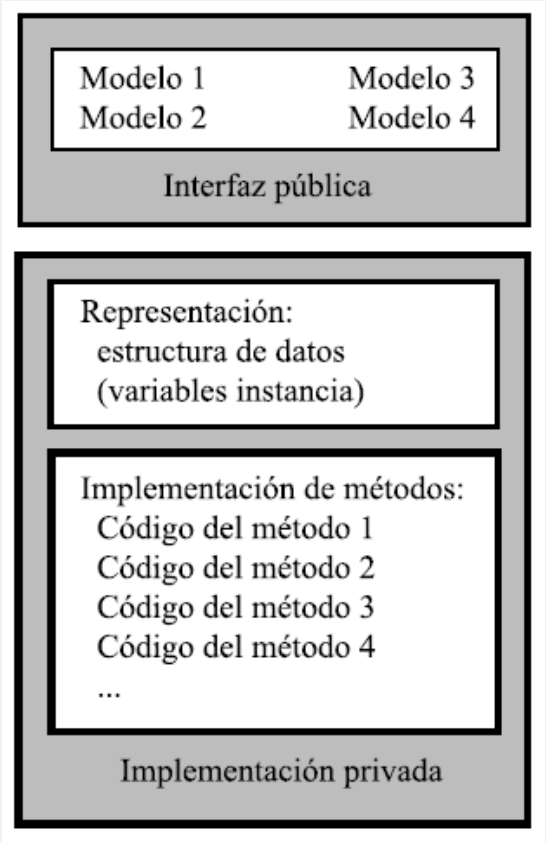
En esencia, un tipo abstracto es un tipo de dato que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre ellos. Un TAD se compone de *estructuras de datos* y los *procedimientos* o *funciones* que manipulan esas estructuras de datos.

# ESPEFICICACIÓN DE UN PUNTO EN EL PLANO CARTESIANO



# TIPOS DE DATOS ABSTRACTOS (TDA)

La estructura de un tipo abstracto de dato (clase), desde un punto de vista global, se compone de la interfaz y de la implementación.



# TIPOS DE DATOS ABSTRACTOS (TDA)

Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes.

Los algoritmos utilizados para implementar cada una de las operaciones de los TAD están encapsuladas dentro de los propios TAD.

La característica de ocultamiento de la información significa que los objetos tienen *interfaces públicas*. Sin embargo, las representaciones e implementaciones de esas interfaces son *privadas*.

# VENTAJAS DE LOS TIPOS DE DATOS ABSTRACTOS (TDA)

1. Permiten una mejor conceptualización y modelización del mundo real. Mejoran la representación y la comprensibilidad. Clarifican los objetos basados en estructuras y comportamientos comunes.

2. Mejoran la robustez del sistema. Si hay características subyacentes en los lenguajes, permiten la especificación del tipo de cada variable. Los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.

3. Mejoran el rendimiento (prestaciones). Para sistemas *tipeados* (tipificados), el conocimiento de los objetos permite la optimización de tiempo de compilación.

# VENTAJAS DE LOS TIPOS DE DATOS ABSTRACTOS (TDA)

4. Separan la implementación de la especificación. Permiten la modificación y la mejora de la implementación sin afectar la interfaz pública del tipo abstracto de dato.
5. Permiten la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
6. Recogen mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un TAD lo hace teniendo en cuenta las operaciones o la funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los usuarios de un TAD se comunican con éste a partir de la interfaz que ofrece el TAD mediante funciones de acceso. Podría cambiarse la implementación del tipo de dato sin afectar al programa que usa el TAD ya que para el programa está oculta.

# IMPLEMENTACIÓN DE LOS TIPOS DE DATOS ABSTRACTOS (TDA)

Las unidades de programación de lenguajes que pueden implementar un TAD reciben distintos nombres:

| Lenguaje | Unidad de programación |
|---|---|
| **Modula-2** | módulo |
| **Ada** | paquete |
| **C++** | clase |
| **Java** | clase |

En estos lenguajes se definen la especificación del TAD, que declara las operaciones y los datos, y la implementación, que muestra el código fuente de las operaciones, que permanece oculto al exterior del módulo.

# ESPECIFICACIÓN DE LOS TAD

- El objetivo de la especificación es describir el comportamiento del TAD; consta de dos partes, la descripción matemática del conjunto de datos y la de las operaciones definidas en ciertos elementos de ese conjunto de datos.

- La especificación del TAD puede tener un enfoque *informal*, que describe los datos y las operaciones relacionadas en *lenguaje natural*.

- Otro enfoque más riguroso, la especificación *formal*, supone suministrar un conjunto de *axiomas* que describen las operaciones en su aspecto *sintáctico* y *semántico*.

# ESPECIFICACIÓN INFORMAL DE UN TDA

Consta de dos partes:

- Detallar en los datos del tipo de valores que puedan tomar.
- Describir las operaciones relacionándolas con los datos

TAD *nombre del tipo* (valores y su descripción)

<Terminarla en clase>

# TIPOS DE DATOS ABSTRACTOS (TDA)

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como int, char y float en Java, C y C++.

Lenguajes de programación como Java tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato*, **TAD**, (*abstract data type*, **ADT**).

El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

# 1.3 EJEMPLOS DE TDA

## TIPOS ABSTRACTOS DE DATOS EN JAVA

Las implementaciones de un TDA en Java se realizan de forma natural mediante una clase.

Dentro de la clase va a residir la representación de los datos junto con las operaciones (métodos de la clase). La **interfaz** del tipo abstracto queda determinada mediante la etiqueta *public*, que se aplicará a los métodos de la clase que represente operaciones.

Ejemplo del TAD *Punto* para representar la abstracción de un punto en el espacio tridimensional, ejemplo:

```
package tda;


public class Punto {
private int x,y,z;
String dimension;
public Punto(int coorx,int coory)
{
this.x=coorx;
this.y=coory;
}
public Punto(int coorx,int coory,int coorz)
{
this.x = coorx;
this.y = coory;
this.z = coorz;
}
public Punto() {
this.x = 0;
this.y = 0;
}
public int getX() {
return x;
}
public int getY() {
return y;
}
public int getZ() {
return z;
}
public String getDimension() {
return dimension;
}
public void setX(int valor)
{
this.x = valor;
}
public void setY(int valor)
{
this.y = valor;
}
public void setZ(int valor)
{
this.z = valor;
}
public void setDimension(String valor)
{
this.dimension = valor;
}
```

# EJERCICIO CLASE PUNTO

- Ver documento WORD en donde viene la definición del Ejercicio.

# 1.5 ANÁLISIS DE ALGORITMOS.

- 1.5.1 Complejidad en el tiempo.
- 1.5.2 Complejidad en el espacio.
- 1.5.3 Eficiencia de los algoritmos

# EFICIENCIA Y EXACTITUD

- Diseñar un algoritmo que sea fácil de entender, codificar y depurar
- Diseñar un algoritmo que haga un uso eficiente de los recursos de la computadoa

# PROPIEDADES DE LOS ALGORITMOS

1. Especificación precisa de la entrada
2. Especificación precisa de cada instrucción
3. Exactitud, corrección
4. Etapas bien definidas y concretas
5. Número finito de pasos.
6. Debe terminar
7. Descripción del resultado o efecto.

# ¿ES UN ALGORITMO LA INSTRUCCIÓN SIGUIENTE?

*Escribir una lista de todos los enteros positivos.*

# EFICIENCIA DE UN ALGORITMO

Raramente existe un único métodoo para resolver un algoritmo. Cuando se comparan dos algoritmos diferentes que resuelven el mismor problema, normalmente se encontrará que un algoritmo es un orden de magnitud más eficiente que el otro. Por lo que es importante, en ese sentido que el programador sea capaz de reconocer y elegir el algoritmo mas eficiente.

**Eficiencia.** Es la propiedad mediante la cual un algoritmo debe alcanzar la solución al problema en le tiempo mas corto posible utilizando la cantidad mas pequeña posible de recursos físicos y que sea compatible con su exactitud o corrección.

**Análisis de Algoritmos.** Método que permite medir la dificultad inherente de un problema

Existen métodos que se basan en:

1.- Número de operaciones que debe efectuar el algoritmo para realizar una tarea

2.- Medir el tiempo que se emplea en llevar a cabo una determinada tarea

Ambos métodos presentan difucultados ya que se quiere generalizar algo que depende del ámbiente de procesamiento, el tamaño de la muestra, etc.

**Algoritmia.** Termino que define el estudio sistemático de las técnicas fundamentales utilizadas para diseñar y analizar algoritmos eficientes.

Se basa fundamentalmente en el análisis de la ejecución del bucle cuando contiene  o en el caso de funciones lineales que no contienen bucles, el número de iteraciones que contiene.

**Le eficiencia como factor espacio-tiempo debe estar estrechamente relacionada con la buena calidad, el funcionamiento y la factibilidad de mantenimiento del programa.**

# FUNCIÓN DE COMPLEJIDAD

- Sea A un algoritmo, la función de complejidad del algoritmo A, T(n) se define como el número máximo de operaciones elementales que utiliza el algoritmo para resolver un problema de tamaño n.

Matemáticamente:

$T(n) = Max\{n_x : n_x$ es el número de operaciones que utiliza A para resolver una instancia x de tamaño n$\}$

Una operación elemental es cualquier operación cuyo tiempo de ejecución es ocotado por una constante (que tenga tiempo constante). Por ejemplo: una operación lógica, una operación aritmética, una asignación o la invocación a un método.

Lógica: a>b, a>=b

Artimética: a+b, a/b

Asignación: a = b

Invocación a un método

# FORMATO GENERAL DE LA EFICIENCIA

En general se puede expresar mediante la función:

*f(n) = eficiciencia.*

Es decir, el eficiencia del algoritmo se examina como una función del número de elementos que tienen que se procesados:

1. Bucles lineales
2. Bucles algoritmicos
3. Bucles anidados

# BUCLES LINEALES

¿Cuántas veces se repite el cuerpo del bucle en el siguiente código?

```
1 i = 1
2 iterar (i<=n)
    1código de la aplicación
    2 i = i + 1
3 fin_iterar
```

f(n) = n

```
1 i = 1
2 iterar (i <= n)
    1 código de la aplicación
    2 i = i + 2
3 fin_iterar
```

f(n) = n/2

# BUCLES ALGORÍTMICOS

Considerando un bucle en el que su variable de control de multiplique o divida dentro de dicho bucle ¿Cuántas veces se repetirá el cuerpo del bucle ?

```
1 i = 1
2 mientras (i<1000)
    código de la aplicación
    i  = i * 2
3 fin_mientras
```

```
1 i = 1000
2 mientras (i > = 1)
    código de la aplicación
    i  = i / 2
3 fin_mientras
```

# FORMATO GENERAL DE LA EFICIANCIA

**Análisis de los bucles de multiplicación y división**

| Bucle Multiplicar | | Bucle dividir | |
|---|---|---|---|
| Iteración | Valor de i | Iteración | Valor de i |
| 1 | 1 | 1 | 1000 |
| 2 | 2 | 2 | 500 |
| 3 | 4 | 3 | 250 |
| 4 | 8 | 4 | 125 |
| 5 | 16 | 5 | 62 |
| 6 | 32 | 6 | 31 |
| 7 | 64 | 7 | 15 |
| 8 | 128 | 8 | 7 |
| 9 | 256 | 9 | 3 |
| 10 | 512 | 10 | 1 |
| Salida | 1024 | Salida | 0 |

Bucle multiplicar $2^{\text{iteraciones}} < 1000$

Bucle de división $1000 / 2^{\text{iteraciones}} >= 1$

Generalizando

$f(n) = [\log_2 n]$

## Bucles anidados

Bucles que contienen otros bucles, se debe determinar cuántas iteraciones contiene cada bucle. El total es el producto del número de iteraciones del bucle interno y el número de iteraciones del bucle externo.

Iteraciones = iteracuones del bucle externo * iteraciones del bucle interno

Existen tres tipos: 1) logaritmico 2) cuadrático dependiente y 3) cuadrático

| Lineal logarítmica | Dependiente cuadrática | Cuadrática |
|---|---|---|
| | | |
| $f(n) = [nLog_2 n]$ | $f(n) = \frac{n(n+1)}{2}$ | $f(n) = n^2$ |
| | | |

# CALCULAR LA FUNCIÓN DE TIEMPO DEL SIGUIENTE CÓDIGO

```
public class DemoSuma {
public static void main(String[] args) {
    1) int suma=0;
    2) for (int i=1;i<=n;i++)
    3)     suma = suma+i;
}

}
```

- Para Calcular T(n), debemos analizar las siguientes intrucciones:
- Instrucción 1, es una asignación y ocupa una unidad de tiempo
- Instrucción 2, tiene involucrada una asignación que utiliza una unidad de tiempo, n+1 comparaciones y n incrementos, por lo que ocupa 1+n+1+n =2n+2 unidad de tiempo.
- Instrucción 3, ocupa dos unidades de tiempo, una para la suma y otra para la asignación, por lo que ocupa 2n unidades de tiempo

El tiempo total del algorirmo

$T(n) = 1 + 2n+2 + 2n = 4n+3$

$T(30) = 4(30 ) + 3 = 120 +3 = 123$

T(30) = 123 operaciones elementales

# PROPIEDADES SUMATORIO

Antes de aprender a calcular la función de la complejidad de un algoritmo, se van a requerir ciertas bases matemáticas, las cuales se resumen a continuación.

- $\sum_{i=1}^{n} 1 = 1 + 1 + 1 \ldots = n$
- $\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \left( \frac{n(n+1)}{2} \right)$
- $\sum_{i=1}^{n} i^2 = i^1 + i^2 + i^3 + \cdots + n^2 = \left( \frac{n(n+1)(2n+1}{6} \right)$

# FUNCIONES PISO(*FLOOR*) Y TECHO(*CEILING*)

| | | |
|---|---|---|
| Valor x | Piso | Techo |
| 12/5 = 2.4 | 2 | 3 |
| 2.7 | 2 | 3 |
| -2.7 | -3 | -2 |
| -2 | -2 | -2 |
| | | |

# CALCULO DE LA COMPLEJIDAD

- T(n) de un algoritmo se puede evaluar desde tres puntos de vista:
  - Peor caso
  - Caso medio
  - Mejor caso

- El estudio de los algoritmos se evalúa en el ***Peor caso.***
- Para hacerlo, de deben calcular el número de **operaciones elementales** de un algoritmo, para lo cual se deben analizar las sentencias: *consecutivas, condicionales y ciclos*.

https://www.youtube.com/watch?v=HU4wJzqjCw0

# CALCULAR T(N) PARA SENTENCIAS CONSECUTIVAS

- Son aquellas que tienen una secuencia, que van una detrás de otra.
- Sumatorio de las operaciones elementales que haya en cada sentencia
- Matemáticamente:
  - Sentencia 1: ejecuta $n_1$ operaciones elementales
  - Sentencia 2: ejecuta $n_2$ operaciones elementales
  - Sentencia 3: ejecuta $n_3$ operaciones elementales
  - ……
  - Sentencia k: ejecuta $n_k$ operaciones elementales

Total de operaciones elementales =

$T(n) = \sum_{i=1}^{k} n_1 = n_1 + n_2 + \ldots + n_k$

Ejemplo:

```
a=10
b=a+12
imprimir(b)
```

= 1+ 2 + 1 = 3

En el caso que las sentencias sean procedimientos o funciones:

Procedimiento 1: ejecuta $f_1(n)$ operaciones elementales

Procedimiento 2: ejecuta $f_2$ (n)operaciones elementales

Procedimiento 3: ejecuta $f_3$ (b)operaciones elementales

……

Procedimiento k: ejecuta $f_k$ (n)operaciones elementales

Total de operaciones elementales =

$T(n) = \sum_{i=1}^{k} f(i)$

# CALCULAR T(N) PARA CONDICIONES

1 Si <condicion> entonces

2 Proceso A

3 Sino

4 Proceso B

La condición de la linea 1 siempre se ejecuta y puede tener una cantidad de operaciones elementales que llamaremos $n_c$ (operaciones elementales del condicional). El proceso A de la linea 2 tiene una complejidad $F_A(n)$ y se ejecuta si la condición es verdadera. EL proceso B de la linea 4 tiene una complejidad $F_B(n)$ y se ejecuta si la condición es falsa. Por lo tanto, como la complejidad se calcula en el **peor de los casos** se debe escoger la cantida máxima de operaciones entre $F_A(n)$ y $F_B(n)$, quedaría

$n_c$ + Max{$F_A(n)$, $F_B(n)$,} Fórmula complejidad para las condicionaes donde,

$n_c$ es la cantidad de operaciones elementales del condicional

$F_A(n)$ *es la función de complejidad del proceso A*

$F_b(n)$ *es la función de complejidad del proceso B*

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE ALGORITMO

- Calcular la complejidad del siguiente algoritmo:

1. Si (a>b y b>c) entonces
2. y = 10
3. y =x+y
4. Sino
5. y=1

- Lo primero es calcular las operaciones elementales de cada línea

| | | |
|---|---|---|
| Linea | Instrucción | |
| 1 | Si (a>b y b>c) entonces | nc = 3 (dos operaciores relacionales y una lógica) |
| 2 | y = 10 | 1 |
| 3 | y =x+y | 2 (la asignación y la suma) |
| 4 | Sino | |
| 5 | y=1 | 1 |

# CONTINUACIÓN

Enseguida, se analiza la complejidad por línea, de lo más interno a lo mas externo:

Lineas 2-3: la complejidad es 1+2 = 3 Proceso A

Lineas 5: la complejidad es 1 Proceso B

Lineas 1-5: se analiza la complejidad de todo el condicional y se aplica la fómula:

$n_c$ + Max

$n_c$ + Max{$F_A(n), F_B(n),$} =

1) 3 +Max{3,1}
2) 3 + 3
3) 6

Por lo que: T(n) = 6, lo que significa que el algoritmo ejecuta 6 operaciones elementales

# CALCULAR T(N) PARA CICLOS

Existen TRES Casos:

**Ciclos caso A.** En cuando el proceso dentro del ciclo es constante, es decir, siempre se ejecuta en un tiempo constante, ejemplo:

1. i = 1
2. mientras i<=n haga
3. <Proceso cualquiera)

**Ciclos caso B.** Es cuando el proceso dentro del ciclo No es constante, es decir es una función que depende de una variable, ejemplo:

1. i =1
2. mientras i<=n haga
3. proceso(i)

**Ciclos caso C**. Es cuando No se conoce la cantidad de veces que itera un ciclo, ejemplo:

1. mientras (condicion) haga
2. proceso

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO A)

```
1 suma=0
2 i=1
3 mientras (i<n)
    4 suma = suma + i
    5 i=i+1
6 imprimir (suma)
```

**Formula complejidad ciclo mientas:**
$1+(n+1)*n_c+n*n_P$

| LINEA | INSTRUCCION | OPERACIONES ELEMENTALES |
|---|---|---|
| 1 | suma=0 | 1 |
| 2 | i=1 | 1 |
| 3 | mientras (i<n) | nc=1 |
| 4 | suma = suma + i | 2 |
| 5 | i=i+1 | 2 |
| 6 | imprimir (suma | 1 |

*n* cantidad de iteraciones del ciclo
$n_c$ cantidad de operaciones elementales condicional
$n_p$ cantidad de operaciones elementales del proceso

$1+(n+1)*n\ n_c+n*\ n_p$
$1+(n+1)*1+n*4$
$5n+2$
$5n+2+2$
$5n+4$
$T(n)=5n+4$
$O(n)=$ n complejidad lineal

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO A)

**Formula complejidad ciclo para:**
$1+(n+1)*n_c+n*(n_p+1)$

```
1 f = 0
2 para(i=1;i=n;++)
3  f = f*i
4 imprimir f
```

| LINEA | INSTRUCCION | OPERACIONES ELEMENTALES |
|---|---|---|
| 1 | f = 0 | 1 |
| 2 | para(i=1;i=n;++) | nc=1 |
| 3 | f = f*i | np=2 |
| 4 | imprimir (f) | 1 |

Del mas interno a lo externo:
líneas 2-3:Caso A,aplicamos fórmula:
$1+(n+1)*n_c+n*(n_p+1)$
Remplazamos:
$1+(n+1)*1+n*(2+1)$
$4n + 2$
líneas 1-4: sumanos a la complejidad
$4n + 2 + 2$
$4n + 4$
$T(n) = 4n+4$
$O(n) = n$ compledidad lineal

$n$ cantidad de iteraciones del ciclo
$n_c$ cantidad de operaciones elementales condicional
$n_p$ cantidad de operaciones elementales del proceso

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO B)

```
1   s=0
2   1 = 1
3    mientras (i<=n) haga
4       j=1
5       mientras (j<=i) haga
6           s = s +1
7          j= j +1
8        i = i+1
9     imprimir s
```

**Formula complejidad ciclo mientras:**
$1+(n+1)*n_c+\sum_{i=1}^{n} f(i)$

*n* cantidad de iteraciones del ciclo
$n_c$ cantidad de operaciones elementales condicional
*f(i)* es la función de la complejidad del proceso que depede de i

| LINEA | INSTRUCCION | OPERACIONES ELEMENTALES |
|---|---|---|
| 1 | s=0 | 1 |
| 2 | 1 = 1 | 1 |
| 3 | mientras (i<=n) | $n_c$=1 |
| 4 | j=1 | 1 |
| 5 | mientras (j<=i) | $n_c$=1 |
| 6 | s = s +1 | 2 |
| 7 | j= j +1 | 2 |
| 8 | i= i+1 | 2 |
| 9 | imprimir s | 1 |

lineas 6-7 = 2+2 =4 np=4 (proceso constante)

lineas 4-7 =1 + (n+1)*$n_c$+n*$n_p$

= 1+(i+1)*1+i*4 = 5i+2

lineas 2-8 = ciclo externo

1+(n+1)*$n_c$+$\sum_{i=1}^{n} f(i)$

1+(n+1)*$n_c$+$\sum_{i=1}^{n}(5i+2)$

Aplicando sumatorio a cada término:

1+n+1+ $\sum_{i=1}^{n}(5i)$ + $\sum_{i=1}^{n}(2)$

Aplicando propiedad sumatoria, la constante puede salir de la sumatoria:

1+n+1+ 5 $\sum_{i=1}^{n}(i)$ + 2$\sum_{i=1}^{n}(1)$

Resolviendo la sumatorio:

2+n+5*n$\frac{(n+1)}{2}$ + 2n

5/2 $n^2$ + 11/2 n+2

Ahora sumalos lineas 1 y 9

5/2 $n^2$ + 11/2 n +2 + 2 = 5/2 $n^2$ + 11/2 n+4

T(n) = 5/2$n^2$+11/2n+4

O(n) = $n^2$ Complejidad cuadrática

| LINEA | INSTRUCCION | OPERACIONES ELEMENTALES |
|---|---|---|
| 1 | s=0 | 1 |
| 2 | 1 = 1 | 1 |
| 3 | mientras (i<=n) | nc=1 |
| 4 | j=1 | 1 |
| 5 | mientras (j<=i) | nc=1 |
| 6 | s = s +1 | 2 |
| 7 | j= j +1 | 2 |
| 8 | i= i+1 | 2 |
| 9 | imprimir s | 1 |

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO B)

```
1   s=0
2   para (i=2;i<n;i++)
3     para (j=1;j<=i;i++
4         s = s +1
9   imprimir s
```

| LINEA | INSTRUCCION | OPERACIONES ELEMENTALES |
|---|---|---|
| 1 | s=0 | 1 |
| 2 | para (i=2;i<n;i++) | $n_c$=1 |
| 3 | para (j=1;j<=i;i++ | $n_c$=1 |
| 4 | s = s +1 | $n_p$=2 |
| 5 | imprimir s | 1 |

Formula complejidad ciclo para:
$1+(n+1)*n_c+ n+ \sum_{i=1}^{n} f(i)$

*n* cantidad de iteraciones del ciclo
$n_c$ cantidad de operaciones elementales condicional
*f(i)* es la función de la complejidad del proceso que depede de i

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO B)

Formula complejidad ciclo para:

Iniciamospor líneas mas internas linea 3 -4 :corresponde caso A,la variable controladora es j que empieza en 1 y va hasta i, por lo que itera i veces.Aplicamos formula:
$1+(n+1)*n_c+n*(n_p+1)$
Remplazamos:
$1+(i+1)*1 + n+i*(2+1) = 4i + 2$
linea 2-4 : ciclo externo, la variable controladora i comienza en 2, itera hasta n-1 **(es decir se repite n-2 veces)**[1]. La complejidad del proceso depende i, por lo que aplicanos formula:
$1+(n+1)*n_c+ n+ \sum_{i=1}^{n} f(i)$
Remplazamos
$1+(n-2+1)*1+(n-2)+ \sum_{i=1}^{n-1}(4i+2)$

| LINEA | INSTRUCCION | OPERACIONES ELEMENTALES |
|---|---|---|
| 1 | s=0 | 1 |
| 2 | para (i=2;i<n;i++) | $n_c$=1 |
| 3 | para (j=1;j<=i;i++ | $n_c$=1 |
| 4 | s = s +1 | $n_p$=2 |
| 5 | imprimir s | 1 |

1)Para encontrar la cantidad de veces que se repite un ciclo se puede utilizar la fórmula : LimiteSuperior – LimiteInferior+1 en este caso (n-1)-2+1 = n -2

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO B)

Simplificando:

$2n-2+\sum_{i=2}^{n-1}(4i+2)$

Aplicando la sumatoria de cada término:

$2n-2+\sum_{i=2}^{n-1}(4i)+\sum_{i=2}^{n-1}(2)$

Aplicando las propiedades de sumatorio:

$2n-2+4\sum_{i=2}^{n-1}(i)+2\sum_{i=2}^{n-1}(1)$

**Operando las sumatorias (1):**

$2n-2+4*(\sum_{i=1}^{n-1}(i)-\sum_{i=1}^{n-1}(i))+2*(\sum_{i=1}^{n-1}(1)-\sum_{i=1}^{n-1}(1))$

Resolviendo

$2n-2+4(\frac{(n-1)(n)}{2})-1+2(n-2)$

Haciendo operaciones algebraicas

$2n^2+2n-10$

línea 1-5: a la complejidad anterior, sumanos las la linea 1 y 5, es decir 2

$2n^2+2n-10+2$

$2n^2+2n-8$

Finalemente $T(n) = 2n^2+2n-8$

$O(n^2)$ = Complejidad cuadrática

(1) Para resolver $\sum_{i=2}^{n-1}(i)$ Se establece esta igualdad:

$\sum_{i=1}^{n-1}(i) = \sum_{i=1}^{n-1}(i) + \sum_{i=2}^{n-1}(i)$

Despejandondo tenemos:

$\sum_{i=2}^{n-1}(i) = \sum_{i=1}^{n-1}(i) - \sum_{i=1}^{1}(i)$

De igual manera podemos deducir que:

$\sum_{i=2}^{n-1}(1) = \sum_{i=1}^{n-1}(1) - \sum_{i=1}^{1}(i)$

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO C)

```
1 t = 0 (1)
2 i = 1 (1)
3 mientras (i<=n) haga
4    t = t +1 (2)
5.   i = i +2  (2)
6  imprimir t (1)
```

***fórmula:***

$(n_r+1)*n_c+(n_r*n_p)$

$n_r$ cantidad de veces que itera el ciclo, o la cantidad de veces que se ejecuta el proceso

$n_c$ cantidad de operaciones elementales condicional

$n_p$ cantidad de operaciones elementales del proceso

| n | número de veces que se repite el ciclo |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 3 |
| 7 | 4 |
| 8 | 4 |

# CALCULAR LA COMPLEJIDAD DEL SIGUIENTE PROGRAMA (CASO C)

Lineas 4-5 = 2 + 2 =4

$n_p = 4$

Lineas 3-5: corresponde ciclo caso C. Es un bucle que se repite a $\left\lceil \frac{n}{2} \right\rceil$, aplicamos fórmula :

$(n_r+1)*nc+(n_r*np)$

$(\left\lceil \frac{n}{2} \right\rceil + 1)*1+ (\left\lceil \frac{n}{2} \right\rceil *4)$

$5 *\left\lceil \frac{n}{2} \right\rceil + 1$

Linea 1-6:

$5 * \left\lceil \frac{n}{2} \right\rceil + 1 + 3$

$5 * \left\lceil \frac{n}{2} \right\rceil + 4$

Finalmente:

$T(n) = 5* \left\lceil \frac{n}{2} \right\rceil +4$

| n | número de veces que se repite el ciclo |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 2 |
| 5 | 3 |
| 6 | 3 |
| 7 | 4 |
| 8 | 4 |

# FORMATO GENERAL DE LA EFICIANCIA

***f (n)* = eficiencia**

La eficiencia se examina como una función del número de elementos que tienen que se procesado:

**Bucles lineales**

En estos bucles las sentencias del *cuerpo del bucle* un número determinado de veces, que determinan le eficiancia del mismo. Son los bucles normalmente del término dominante.

¿cuántas veces se repite el cuerpo del siguiente bucle?

i = 0

Iterar (i<=n)

  código de la apliación

  i = i + 1

Fin de iteración

Si n=100 entonces f(n) = 100

f(n) = n

¿cuántas veces se repite el cuerpo del siguiente bucle?

i = 0
Iterar (i<=n)
  código de la apliación
  i = i + 2
Fin de iteración
Si n=100 entonces f(n) = 100

f(n) = n / 2

# FORMATO GENERAL DE LA EFICIANCIA

**Bucles algoritmicos**

En estos bucles las sentencias del *cuerpo del bucle* un número determinado de veces, que determinan le eficiancia del mismo. Son los bucles normalmente del término dominante.

¿cuántas veces se repite el cuerpo del siguiente bucle?

```
i = 1
mientras (i<1000)
   código de la apliación
   i = i * 2
Fin de mientras
```

```
i = 1000
mientras (i>=1)
   código de la apliación
   i = i / 2
Fin de mientras
```

# FORMATO GENERAL DE LA EFICIANCIA

**Análisis de los bucles de multiplicación y división**

| Bucle Multiplicar | | Bucle dividir | |
|---|---|---|---|
| Iteración | Valor de i | Iteración | Valor de i |
| 1 | 1 | 1 | 1000 |
| 2 | 2 | 2 | 500 |
| 3 | 4 | 3 | 250 |
| 4 | 8 | 4 | 125 |
| 5 | 16 | 5 | 62 |
| 6 | 32 | 6 | 31 |
| 7 | 64 | 7 | 15 |
| 8 | 128 | 8 | 7 |
| 9 | 256 | 9 | 3 |
| 10 | 512 | 10 | 1 |
| Salida | 1024 | Salida | 0 |

Bucle multiplicar $2^{\text{iteraciones}} < 1000$

Bucle de división $1000 / 2^{\text{iteraciones}} >= 1$

Generalizando

$f(n) = [\log_2 n]$

## Bucles anidados

Bucles que contienen otros bucles, se debe determinar cuántas iteraciones contiene cada bucle. El total es el producto del número de iteraciones del bucle interno y el número de iteraciones del bucle externo.

Iteraciones = iteracuones del bucle externo * iteraciones del bucle interno

Existen tres tipos: 1) logaritmico 2) cuadrático dependiente y 3) cuadrático

| Lineal logarítmica | Dependiente cuadrática | Cuadrática |
|---|---|---|
| | | |
| $f(n) = [nLog_2 n]$ | $f(n) = \frac{n(n+1)}{2}$ | $f(n) = n^2$ |
| | | |

# ANÁLISIS DE RENDIMIENTO

La medida de rendimiento de un programa se consigue mediante la complejidad del espacio y tiempo.

**Complejidad de espacio:** es la cantidad de memoria que se necesita para ejecutar un programa hasta la terminación.

**Complejidad de tiempo:** es la cantidad de tiempo de computadora que se necesita para utilizarlo. Se utiliza la función *T(n)*. Representa el número de unidades de tiempo tomadas por un programa para cualquier entrada n.

Si la función *T(n)* de un programa es *T(n)= c * n*, entonces el tiempo de ejecución el linealmente proporcional al tamaño de la entrada, se dice que es *tiempo lineal* o simplemente *lineal*

*Ejemplo*

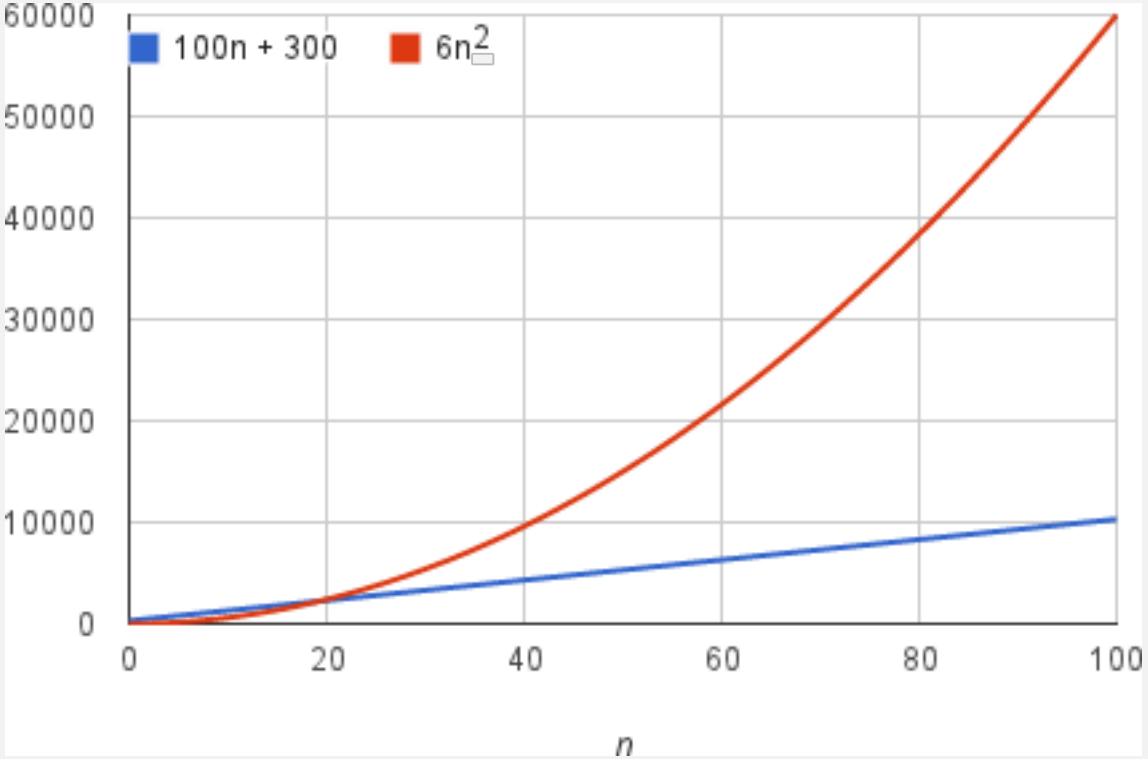
```
double serie(double x, int b) {
    double s;
    int i;
    s  = 0.0;                 //t1
    for(i =1;i<=n;i++) {  //t2
       s += i*x;             //t3
    }
    return s;                 //t4;
}
T(n) = t1 + n*t2 + n*t3 +t4
```

El tiempo crece a medida que lon hace n, por lo que se expresa el tiempo de ejecución con respecto al valor de n, entonces:
**Peor Caso.** El peor tiempo que se puede tener, adecuado para algoritmos con tiempo de respuesta críticos
**Mejor Caso.** Indica el mejor tiempo que podemos tener
**Caso Medio.** El tiempo medio de ejecición sobre todas las posibles entradas de tamaño n, es la más práctica

# NOTACIÓN O-GRANDE (COTA SUPERIOR DE AJUSTE)

- Métrica para medir la eficiciencia de un Algoritmo
- El tiempo de ejecución de un algoritmo depende de cuánto tiempo le tome a una computadora ejecutar las líneas de código del algoritmo, y eso depende de la velocidad de la computadora, el lenguaje de programación y el compilador que traduce el programa del lenguaje de programación al código que se ejecuta directamente en la computadora, entre otros factores.
- Primero, necesitamos determinar cuánto tiempo se tarda el algoritmo, en términos del tamaño de su entrada.
- Segundo, qué tan rápido crece una función con el tamaño de la entrada. A esto lo llamamos la **tasa de crecimiento** del tiempo de ejecución. Para mantener las cosas manejables, tenemos que simplificar la función para extraer la parte más importante y dejar de lado las partes menos importantes.

- Por ejemplo, supón que un algoritmo, que corre con un entrada de tamaño n, se tarda $6n^2 + 100n + 300$ instrucciones de máquina , el término $6n^2$ se vuelve mas grande que el resto de los términos, 100n + 300, confirme la n se hace grande, 20 en este caso, como se muestra en la gráfica, con valores de n de 0 a 100.



# ORDEN DE MAGNITUD (NOTACIÓN O GRANDE)

- Si T(n) = 1, se dice que es de orden constante y se denota **O(1)**
- Si T(n) = $\log_2(n)$, se dice que es de orden logarítmico y se denota **O(logn)**
- Si T(n) = an+b, siendo a y b constantes numéricas, se dice que es de orden lineal, y se denota **O(n)**
- Si T(n) = nlog2(n, se dice que es de orden n$\text{Log}_2(n)$, y se denota como **O(n$\text{Log}_2$n)**
- Si T(n) =$an^2+bn+c$, siendo a,b y c constanes numéricas, se dice que es de orden cuadrático y se denota como **O($n^2$)**
- Si T(n)= $an^3+bn^2+cn+d$, siendo a,b,c y d constanes numéricas, se dice que es de orden cúbico y se denota como **O($n^3$)**
- Si T(n) = $m^n$, donde, donde m=1,2,3…, se dice que es de orden polinomial.
- Si T(n) = $c^n$, donde c>1, se dice que es de orden exponencial. Por ejemplo T(n) = $3^n$ es **O($3^n$)**
- Si T(n) = n!, se dice que es de orden factorial y se denota como **O(n!)**

# NOTACIÓN O-GRANDE

Seguir ejemplo en:

https://es.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Con la notación O se expresa una aproximación de la relación entre el tamaño de un problema y la cantidad de proceso necesario para hacerlo. Ejemplo

$f(n) = x^2$ - 2n + 3 entonces

$f(x) = es\ O(n^2)$

Visualizar el siguiente video:

1)https://www.youtube.com/watch?v=O5LiA5ireA4

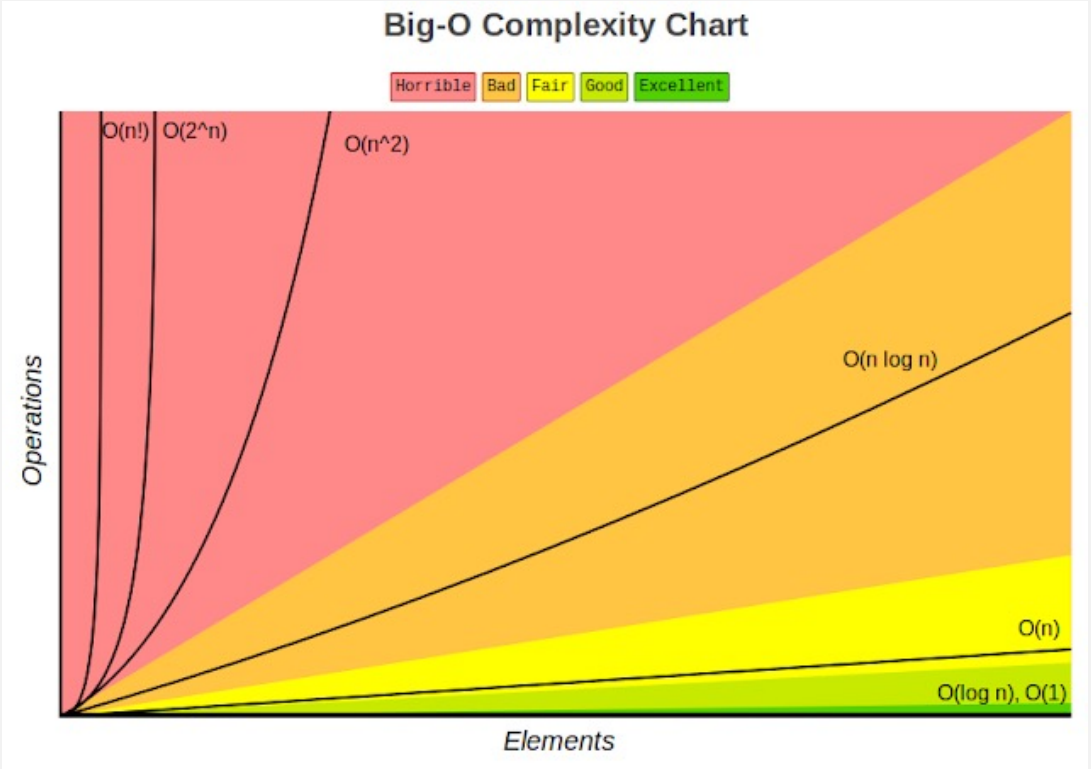
2) https://www.youtube.com/watch?v=IZgOEC0NIbw

3) https://www.youtube.com/watch?v=HEISXs0wYIM

Determinación de la notación O:

1. En cada término, establecer el coeficiente del término 1.
2. Mantener el término mayor de la función y descartar los restantes. Los términos se ordenan de menor a mayor:

1) $\log_2 n$ 2) n 3) $n\log_2 n$ 4) $n^2$ 5) $n^3$ … 6) $n^k$ 7) $2^n$ 8)n!



# EJEMPLO: CALCULAR LA FUNCIÓN O GRANDE DE EFICIENCIA DE LAS SIGUIENTES FUNCIOES

**Caso a**

**$f(n) = 1/2n(n+1) = 1/2n^2+1/2n$**

1.- Se eliminan todos los coeficientes y se obtiene:

$n^2+n$

2.- Se eliminan los factores más pequeños

$n^2$

3.- La notación O correspondiente:

$O(f(n)) = O(n^2)$

**Caso b**

**$f(n) = a_j n^k + a_{j-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$**

1.- Se eliminan todos los coeficientes y se obtiene:

$f(n) = n^k + n^{k-1} + \ldots + n^2 + n + 1$

2.- Se eliminan los factores más pequeños

$n^k$

3.- La notación O correspondiente:

$O(f(n)) = O(n^k)$

# PROPIEDADES DE LA NOTACIÓN O-grande

1. Siendo c una constante, *c*O(f(n)) = O(f(n))*.
   1. ejemplo, si $f(n) = 3n^4$, entonces $f(n) = 3*O(n^4) = O(n^4)$
2. $O(f(n)) + O(g(n)) = O(f(n))+g(n))$
   1. ejemplo, si $f(n) = 2e^n$ y $g(n) = 2n^3$:
   2. $O(f(n)) + O(g(n)) = O(f(n)+g(n)) = O(2e^n + 2n^3) = O(e^n)$
3. Maximo(O(f(n)),O(g(n))) = O(Maximo(f(n),g(n))
   1. ejemplo:
   2. Maximo(O(log(n)),O(n)) = O(Maximo(Log(n),n)) = O(n)
4. O(f(n)) * O(g(n)) = O(f(n)*g(n))
   1. ejemplo, si $f(n) = 2n^3$ y g(n) = n:
   2. $O(f(n))*O(g(x)) = O(f(x)*g(x)) = O(2n^3*n) = O(n^4)$.
5. $O(Log_a(n))= O(Log_b(n))$ para a,b>1

   Las funciones logarítmicas son de orden logarítmicos, independientemente de la base del logaritmo
6. (loga(n!)) = O(n*log(n)).
7. Para k>1 $O(\sum_{i=1}^{n} i^k) = O(n^{k+1}$
8. $O(\sum_{i=2}^{n} \log(i)) = O(n\log(n))$

# COMPLEJIDAD DE LAS SENTENCIAS BÁSICAS DE JAVA

- Las sentencias de asignación son de orden constante O(1)
- La complejidad de una sentencia de selección es el máximo de las complejidades del bloque *then* y del bloque *else*.
- La complejidad de una sentencia de selección múltiple (*switch*) es el máximo de las complejidades de cada uno ed los bloques *case*.
- Para calcular la complejidade de un bucle, condicional o automático, se ha de estimar el número máximo de iteraciones para el *peor caso*; entonces la complejidad del bucle es el producto del número de iteraciones por la complejidad de las sentencias que forman el cuerpo del bucle.
- La complejidad de un bloque se calcula como la suma de las complejidades de cada secuencia del bloque.
- La complejidad de la llamada a un método es de orden 1, complejidad constante. Es necesario considerar la complejidad del método invocado

# Determinar la complejidad del método

```
double mayor(double x,double y) {
   if(x>y)
      return x;
   else
      return y;
}
```

El método consta de una sentencia de selección, cada alternativa tiene una complejidad constante, O(1); entonces,la complejidad del método mayor() es O(1)

# Determinar la complejidad del método

```
void escribeVector(double[]x,int n) {
   int j;
   for(j=0;j<n;j++)
   {
      System.out.println(x[j]);
   }
}
```

El método consta de bucle que se ejecuta n veces, O(n). El cuerpo del bucle es la llamada a println(),complejidad constante O(1), como conclusion, la complejidad del método es:

O(n)* O(1) = O(n)

# Determinar la complejidad del método

```
double suma(double[]d,int n){
   int k;double s;
   k=s=0;
   while (k<n)
   {
      s+=d[k];
      if(k==0)
         k=2;
      else
         k=2*k;
   }
   return s;
}
```

suma() esta formado por una sentencia de asignación múltiple, O(1), de un bucle condicional y la sentencia que devuelve control de complejidad constante, O(1).Por consiguiente la complejidad del método está determinado por el bucle. El número de iteraciones es igual al número de veces que el algoritmo multiplica por dos la variable k. Si t es el número de veces que k se puede multiplicar hasta alcanzar el valor de n, que hace que termine el bucle, entonces $k = 2^t$.

$0,2,2^2,2^3,\ldots\ldots 2^t >= n$

Tomando logaritmos $t >= \log^2 n$;por consiguiente el máximo de iteraciones es $t=\log_2 n$ El cuerpo del bucle consta de una sentencia simople y una sentencia de selección de complejidad O(1), por lo que:

$O(Log_2 n) * O(1) = O(Log_2 n) = O(Log_n)$ = complejidad logarítmica

# Determinar la complejidad del método

```
void traspuesta(float[][]d,int n){
   int i,j;
   for(i=n-2;i>0;i--)
   {
      for(j=i+1;j<n;j++)
      {
         float t;
         t=d[i][j];
         d[i][j]=d[j][i];
         d[j][i]= t;
      }
   }
}
```

El método consta de dos bucles for anidados. El bucle interno esta formado por tres sentencias de complejidad constante, O(1). El bucle externo siempre realiza n-1 veces el bucle interno. A su ve, el bucle interno hace k veces su bloque de sentencias, k varia de 1 a n-1, de modo que el número total de iteraciones es:

$$C=\sum_{k=1}^{n-1} k$$

El desarrollo del sumatorio produce la expresión

$$(n-1)+(n-2)+\ldots.1 = \frac{n(n-1)}{2}$$

Aplicando las propiedades de la complejidad del método $O(n^2)$, ya que es el término que domina en el tiempo de ejecución es $n^2$, se dice que la complejidad es cuadrática.

# ESTRUCTURAS DINÁMICAS Y ESTÁTICAS

En programación, el concepto de “estructura” equivale a “colección” o “conjunto”. Una estructura de datos representa una colección de datos

Las estructuras contienen datos que pueden ser coherentes u homogéneos.

Por ejemplo, los atributos de una clase de Java componen componen un conjunto de datos que son coherentes entre sí.

# ESTRUCTURAS DINÁMICAS Y ESTÁTICAS

```
public class Persona{
    private String dni;
    private String nombre;
    private Date FechaNacimiento;
    // Métodos get y set para los atributos.
}
```

```
public class Fecha{
    private int dia;
    private int mes;
    private int anio;
    //Métodos get y set para los atributos.
}
```

# ESTRUCTURAS DINÁMICAS Y ESTÁTICAS

La memoria estática es memoria porque se reserva desde el momento que un programa inicia hasta que el programa termina.

El almacenamiento en memoria estática no puede cambiar su tamaño. Su uso y su posición relativa a los otros componentes de la aplicación es típicamente determinada cuando el código fuente de la aplicación es compilado.

Dentro de este tipo de estructura en Java se encuentran los ***arrays*** y las ***clases***.

En el caso de las clases, queda claro que la cantidad elementos que una clase podrás contener dependerá de la cantidad de atributos que le declaremos mientras la estamos programando.

En el caso de los arreglos aunque cada permite declarar su capacidad física en tiempo de ejecución una vez declarada esa capacidad el arreglo permanecerá estático Y no podrá ser redimensionado

# ESTRUCTURAS DINÁMICAS Y ESTÁTICAS

La memoria dinámica es memoria que es requerida y administrada cuando el programa se esta ejecutando.

Los parámetros de la memoria dinámica no pueden ser especificados cuando un programa se compila debido a que los factores como el tamaño y el tiempo de vida no son conocidos hasta en tiempo de ejecución.

Aunque la memoria dinámica puede permitir mayor flexibilidad, el uso de memoria estática permite a una aplicación ejecutarse más rápido debido a que no tiene que realizar  ningún mantenimiento extra en tiempo de ejecución.

# ESTRUCTURAS DINÁMICAS

- Una estructura dinámica consiste en una colección de datos cuya capacidad física puede ser modificada Durante la ejecución de un programa.
- Si se necesitan más capacidad porque siquiera contener una mayor cantidad de elementos entonces estructura dinámica podrá crecer. Y si ya no se necesita tanta capacidad porque decidimos desprender de alguno de los elementos que contenía la colección, la estructura dinámica podrá decrecer y, de este modo, insumo una menor cantidad de amor.
- Cuando los elementos que contiene una estructura dinámica están almacenados uno a continuación del otro, decimos que se trata de una estructura línea.
- Por el contrario si para un determinado elemento tengo una colección no podemos determinar unívoca mente qué que sucede estamos en presencia de una estructura de datos no tiene