CLASE 1

Informática → Es la ciencia que estudia el análisis y resolución de problemas utilizando computadoras.

- **Ciencia:** Se relaciona con una metodología fundamentada y racional para el estudio y resolución de los problemas(se vincula con la matemática y la ingeniería)
- **Resolución:** se puede aplicar para diferentes áreas
- **Computadora:** Máquina digital y sincrónica, con cierta capacidad de cálculo numérico y lógico controlado por un programa almacenado y con probabilidad de comunicación con el mundo exterior. Ayuda al hombre a realizar tareas repetitivas en menor tiempo y con mayor exactitud. **No razona ni crea soluciones**, sino que ejecuta una serie de órdenes que le proporciona el ser humano
- **OBJETIVO:** Es resolver problemas del mundo real utilizando una computadora (utilizando un software)

Paradigmas

1. **Imperativo**
2. Orientado a Objetos
3. Lógico
4. Funcional

Pasos

1. **Poseer un problema**
2. **Modelizar el problema**(qué acciones permitir, como realizarlas y qué efecto genera)

- Pensar que acciones se van a permitir y que implica cada acción permitida
  - Define los mecanismos de interacción necesarios
  - Establece el efecto sobre la máquina y el usuario
  - Indica los informes necesarios
- **Abstracción:** Interpretar los problemas de mundo real(analizando sus aspectos esenciales), para poder expresarlo en términos precisos
- **Modelización:** A partir de la expresión de la abstracción, se busca simplificarlo buscando sus:
  1. Aspectos Principales
  2. Datos a Procesar
  3. Contexto del Problema (este puede imponer restricciones)
- **Especificación:** Determinar el objetivo del problema. Consiste en establecer unívocamente el contexto, las precondiciones y las poscondiciones.

3. **Modularizar la solución**

- Descomponer el problema en partes más pequeñas para obtener una solución
  - Reduce la complejidad
  - Poder reutilizar módulos
  - Distribuir el trabajo

4. **Realizar el programa**

- Se va diseñar su implementación: esto requiere escribir el programa y elegir los datos a representar.

PROGRAMA = ALGORITMO + DATOS

Programa: Es un conjunto de instrucciones(que pueden ejecutarse sobre una computadora), que cumplen con una función específica

Algoritmo: Especificación rigurosa de la secuencia de pasos (instrucciones) a realizar sobre un autómata para alcanzar un resultado deseado en un tiempo finito.

- Suponemos que empieza y termina (**tiempo finito**)
- Se debe expresar en forma clara y unívoca (**especificación rigurosa**)
- Si el **autómata** es una computadora, las instrucciones tiene que ser "entendibles" y ejecutables por la máquina

  Las **instrucciones** (que también se han denominado acciones) representan las operaciones que ejecutará la computadora al interpretar el programa.

Datos: Es una representación de un objeto del mundo real mediante la cual podemos modelizar aspectos del problema que se quiere resolver con un programa sobre una computadora. Puede ser constante o variable.

Para el Desarrollador	Para la Computadora
• **Operatividad**: El programa debe realizar la función para la que fue concebido. • **Legibilidad** : El código fuente de un programa debe ser fácil de leer y entender. Esto obliga a acompañar a las instrucciones con comentarios adecuados. • **Organización**: El código de un programa debe estar descompuesto en módulos que cumplan las subfunciones del sistema. • **Documentados**: Todo el proceso de análisis y diseño del problema y su solución debe estar documentado mediante texto y/o gráficos para favorecer la comprensión, la modificación y la adaptación a nuevas funciones.	• Debe contener instrucciones válidas. • Deben terminar. • No deben utilizar recursos inexistentes.

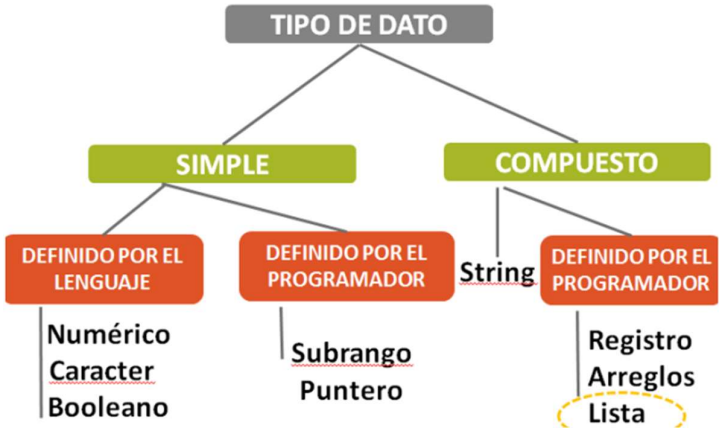
5. **Utilizar la computadora** → Utilizar la computadora

- La computadora es una máquina capaz de aceptar datos de entrada, efectuar con ellos cálculos aritméticos y lógicos y dar información de salida (resultados), bajo control de un programa previamente almacenado en su memoria.
- **Lenguaje de Programación:** Es un conjunto de instrucciones permitidas (las cuales están definidas por reglas semánticas y sintácticas), que se utilizan para expresar soluciones de problemas.

TIPOS DE DATOS

Un tipo de dato es una clase de objetos de datos ligados a un conjunto de operaciones para crearlos y manipularlos.

- Tienen un rango de valores posibles
- Tienen un conjunto de operaciones permitidas
- Tienen una representación interna



- Definido por el programador: Permiten definir nuevos tipos de datos a partir de los tipos simples.
- Definido por el lenguaje: Son provistos por el lenguaje y tanto la representación como sus operaciones y valores son reservadas al mismo

SIMPLE: Aquellos que toman un único valor, en un momento determinado, de todos los permitidos para ese tipo.
COMPUESTO: Pueden tomar varios valores a la vez que guardan alguna relación lógica entre ellos, bajo un único nombre.

Numérico

Representa el conjunto de números que se pueden necesitar. Estos números pueden ser enteros o reales.

- Entero
  - Tipo de dato simple y ordinal
  - Al tener una representación interna tienen un número máximo y otro mínimo
  - Operaciones
    - +, -, * , /, <, >, =, <=, =>, MOD(resto entero), DIV(cociente entero)
- Real
  - Tipo de dato simple (representa números con coma)
  - Al tener una representación interna, tienen un número mínimo y uno máximo
  - Operaciones
    - +, -, *, /, <, >, =, <=, =>

*** El orden de precedencia para la resolución es:*
*1. operadores *, /*
2. operadores +, -
3. operadores div y mod.
*En caso que el orden de precedencia natural deba ser alterado, es posible la utilización de paréntesis dentro de la expresión.***

Lógico

Representa datos que pueden tomar dos valores Verdadero (true) o falso (false)

- Es un tipo de dato simple y ordinal
- Operaciones

    - OR, AND, NOT
- Orden de Precedencia
    1. NOT
    2. AND
    3. OR

Carácter

Representa un conjunto finito y ordenado de caracteres que la computadora reconoce. Un dato de tipo carácter contiene sólo un carácter.

- Tipo de dato simple y ordinal
- ‘a’, ‘b’, ‘@’
- Operaciones
    - =, <, >, <=, =>, <>

Variable

Es una zona de memoria que tiene un contenido.La dirección inicial de esta zona se asocia con el nombre de la variable.
Puede cambiar su valor durante el programa.

Constante

Es una zona de memoria que tiene un contenido.La dirección inicial de esta zona se asocia con el nombre de la variable.
NO puede cambiar su valor durante el programa.

const Identificador = valor;
/----------/

- Los diferentes tipos de datos deben especificarse y a esta especificación dentro de un programa se la conoce como **declaración**.
- Algunos lenguajes exigen que se especifique a qué tipo pertenece cada una de las variables. Verifican que el tipo de los datos asignados a esa variable se correspondan con su definición. Esta clase de lenguajes se denomina **fuertemente tipados (strongly typed)**.
- Otra clase de lenguajes, que verifica el tipo de las variables según su nombre, se denomina **auto tipados (self typed)**.
- Existe una tercera clase de lenguajes que permiten que una variable tome valores de distinto tipo durante la ejecución de un programa. Esta se denomina **dinámicamente tipados (dinamically typed)**.

Pre Condición:

- Lo que nosotros queremos que el programa haga
- Es la información que se conoce como verdadera antes de iniciar el programa (ó módulo).

Post Condición:

- Como quedarian cada una de las variables después de ejecutarse el programa
- Es la información que debería ser verdadera al concluir el programa (o módulo), si se cumplen adecuadamente los pasos especificados.

Read (Lectura)

Se usa para tomar datos desde un dispositivo de entrada (por defecto desde teclado) y asignarlos a las variables correspondientes.

Write (Escritura)

Se usa para mostrar el contenido de una variable, por defecto en pantalla.

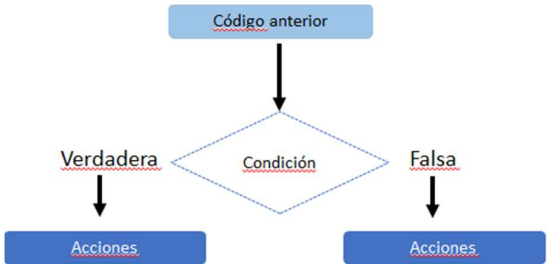
ESTRUCTURAS DE CONTROL

Secuencia

- Representada por una sucesión de operaciones (por ej. asignaciones), en la que el orden de ejecución coincide con el orden físico de aparición de las instrucciones.

Decisión

- En un algoritmo representativo de un problema real es necesario tomar decisiones en función de los datos del problema.

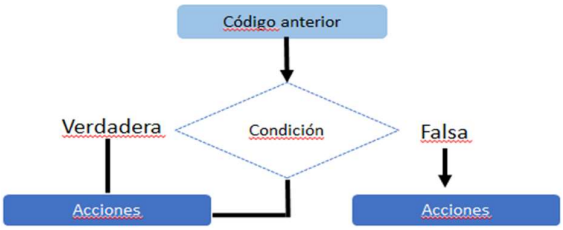


Iteración

- Cuando quiero repetir una cantidad de veces que desconozco (xq depende de una condición), una o un conjunto de acciones
- Pueden ser **Pre-Condicionales** o **Post-Condicionales**

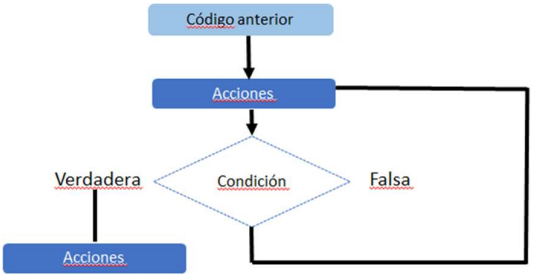
Pre-Condicionales(While)

Evalúan la condición y si es verdadera se ejecuta el bloque de acciones. **Dicho bloque se pueda ejecutar 0, 1 ó más veces.** Se ejecuta mientras sea verdadera. *(El valor inicial de la condición debe ser conocido o evaluable antes de la evaluación de la condición)*

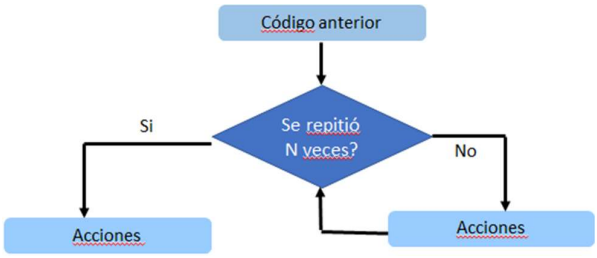


Post-Condicionales(Repeat-Until)

Ejecutan las acciones luego evalúan la condición y ejecutan las acciones mientras la condición es falsa.**Puede ejecutarse 1 o más veces.**



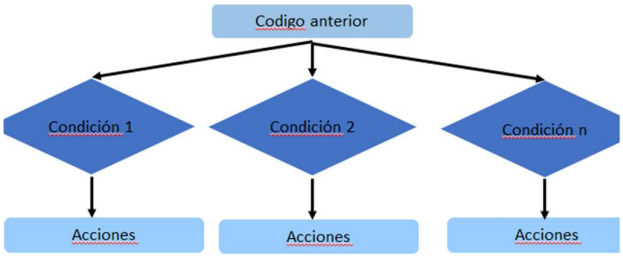
Repetición



- Es una extensión natural de la secuencia. Consiste en repetir N veces un bloque de acciones. **Este número de veces que se deben ejecutar las acciones es fijo y conocido de antemano**.
- Índice:
    - tipo ordinal
    - No puede modificarse dentro del lazo.(se incrementa y decrementa automáticamente)
    - Cuando el for termina la variable índice no tiene valor definido.

CLASE 2

Selección



- Permite realizar una o varias acciones dependiendo del valor de una variable de tipo ordinal.(Evalúa una sola vez, si es verdadera una condición sale de la estructura)
- *Pseudo:*

```
case (variable) of
  condicion1:   accion1;
  condición 2:  begin
                  acción2;
                  acción3;
                end;
  …..
  condición n:   acción;
end;
```

- La variable debe ser ordinal
- Las opciones deben ser disjuntas

Maximos y Minimos

- Utilizar una variable que representará al máximo o al mínimo.
- Inicializar la variable antes de comenzar la lectura de los datos. **El máximo en un valor bajo y el mínimo en un valor alto.**
- Actualizar la variable máximo o mínimo cuando corresponda.

Tipos de datos definidos por el Usuario

- Aumento de la riqueza **expresiva** del lenguaje, con mejores posibilidades de abstracción de datos.

- Mayor **seguridad** respecto de las operaciones que se realizan sobre cada clase de datos.
- **Límites** preestablecidos sobre los valores posibles que pueden tomar las variables que corresponden al tipo de dato.

TDDU: Un tipo de dato definido por el usuario es aquel que no existe en la definición del lenguaje, y el programador es el encargado de su especificación.

Type
identificador = tipo;

Ventajas

- **Flexibilidad:** en el caso de ser necesario modificar la forma en que se representa el dato.*(sólo se debe modificar una declaración en lugar de un conjunto de declaraciones de variables)*
- **Documentación:** se pueden usar como identificador de los tipos, nombres autoexplicativos, facilitando de esta manera el entendimiento y lectura del programa.
- **Seguridad:** se reducen los errores por uso de operaciones inadecuadas del dato a manejar.

Subrango

Es un **tipo ordinal,** que consiste de una sucesión de valores de un tipo ordinal (predefinido o definido por el usuario) tomado como base.

- Es Simple
- Es Ordinal
- Existe en la mayoría de los lenguajes
- *Pseudo*

  Type
  nombre = valor1..valor2;

- Operaciones Permitidas
  - **Asignación**
  - **Comparación**
  - Todas las operaciones permitidas para el tipo base
- Operaciones No Permitidas
  - Depende del tipo base

CLASE 3

Modularización

Significa dividir un problema en partes funcionalmente independientes, que encapsulan operaciones y datos.(cada módulo debe tener una función)

- Cada subproblema está en un mismo nivel de detalle.
- Cada subproblema puede resolverse independientemente.

- Las soluciones de los subproblemas pueden combinarse para resolver el problema original.

Módulo

- Tarea específica bien definida que se comunican entre sí adecuadamente y cooperan para conseguir un objetivo común.
- Encapsula acciones tareas o funciones
- En ellos se pueden representar los objetivos relevantes del problema a resolver.
- Metodología TOP-DOWN

Ventajas

- Mayor productividad: Al dividir un sistema de software en módulos funcionalmente independientes, un equipo de desarrollo puede trabajar simultáneamente en varios módulos, incrementando la productividad (es decir **reduciendo el tiempo de desarrollo**)
- Reusabilidad: la posibilidad de utilizar repetidamente el producto de software desarrollado.(**reusar código**)
- Facilidad de Crecimiento: permite disminuir los riesgos y costos de incorporar nuevas prestaciones a un sistema en funcionamiento.
- Facilidad de Mantenimiento
- Legibilidad: mayor claridad para leer y comprender el código fuente.*(el ser humano maneja y comprende con mayor facilidad un número limitado de instrucciones directamente relacionadas)*

Procedures

Conjunto de instrucciones que realizan una tarea específica y **retorna 0, 1 ó más valores.**

```
procedure nombre;
var
 ....      → Variables Locales
begin
 ....      → Código del Procedimiento
end;
```

Function

Conjunto de instrucciones que realizan una tarea específica y **retorna un único valor de tipo simple.**

```
Function nombre(parámetros) :tipo;
var
 ....
begin
 ....
  nombre:= valor a retornar;
end;
```

La función retorna a la misma línea del llamado y los procedimiento retornan a la línea siguiente

Invocación

- Invocación con variable: El resultado se asigna a una variable del mismo tipo que devuelve la función

  aux:= auxiliar;
  write (aux);

- Invocación en un write: El resultado se puede mostrar en una sentencia write.

  write ('El resultado es:',auxiliar);

- Invocación en condicional: Se puede invocar a la función dentro de un if, o while.

  if (auxiliar = 2,5) then
  ....
  while (auxiliar = 2,5) do
  ....

Alcance de las Variables

```
Program alcance;
Var
  a,b: integer;   → Variables Globales del Programa (Pueden ser usadas en todo el
                    programa (incluyendo módulos))
procedure prueba;
Var
  c: integer;   → Variable Local(Pueden ser usadas sólo en el proceso que están
                  declaradas)
Begin
End.
Var
  d:integer;   → Variable Local del Programa(Pueden ser usadas sólo en el cuerpo del
                 programa)
Begin
End.
```

Si es una variable utilizada en un proceso:

- Se busca si es variable local
- Se busca si es un parámetro
- Se busca si es variable global al programa

Si es una variable usada en un programa:

- Se busca si es variable local al programa
- Se busca si es variable global al programa

Parámetros

¿Cómo se pueden comunicar los módulos(entre sí o con en el programa)? se pueden utilizar variables globales y parámetros

- Desventajas de usar **variables globales**
    - Ocupa demasiada memoria
    - Demasiados identificadores
    - No se especifica la comunicación entre los módulos
    - Conflictos de nombres de identificadores utilizados por diferentes programadores.
    - Posibilidad de perder integridad de los datos, al modificar involuntariamente en un módulo datos de alguna variable que luego deberá utilizar otro módulo

Los **Parámetros** son variables que tienen como característica transferir información entre módulos.

- **Data Hiding o Ocultamiento de Datos** significa que todo dato que es relevante para un módulo debe ocultarse (NO deben ser visibles) del resto de módulos. Nosotros decidimos qué valores le entregamos al módulo y que debe devolver.
- Los datos compartidos entre módulos deben especificarse como parámetros.
- **Los parámetros se relacionan por posición y no por nombre**

Parámetro por Valor

- Un dato de entrada por valor es llamado parámetro IN y significa que el módulo recibe (sobre una variable local) un valor proveniente de otro módulo (o del programa principal).*Puede realizar operaciones y/o cálculos, pero no producirá ningún cambio xq recibe una copia.*

Parámetro por Referencia

- La comunicación por referencia (OUT, INOUT) significa que el módulo recibe el nombre de una variable (**referencia a una dirección**) conocida en otros módulos del sistema. *Puede operar con ella y su valor original dentro del módulo, y las modificaciones que se produzcan se reflejan en los demás módulos que conocen la variable.*
- Siempre debe pasarse por una variable, nunca por un valor

Características de los Parámetros

- El número y tipo de los argumentos utilizados en la invocación a un Procedimiento deben coincidir con el número y tipo de parámetros del encabezamiento del módulo.
- Un parámetro por valor debiera ser tratado como una variable de la cual el Procedimiento hace una copia y la utiliza localmente.
- Los parámetros por referencia operan directamente sobre la dirección de la variable original(**por lo tanto no requiere memoria local**).

CLASE 4

ESTRUCTURAS DE DATOS

Son conjuntos de distintas variables(pueden ser de distinto tipo), relacionadas entre sí

Clasificación de ESTRUCTURAS DE DATOS

- **Elementos:** de que tipo son sus elementos
    - Homogénea: elementos del mismo tipo
    - Heterogénea: elementos de distinto tipo
- **Tamaño:** si puede variar su tamaño a lo largo del programa

    - Estática: el tamaño no varía
    - Dinámica: el tamaño varia
- **Acceso:** como se puede acceder a sus elementos
    - Secuencial: Para acceder a un elemento particular se debe respetar un orden predeterminado
    - Directo: Se puede acceder a un elemento particular, directamente, sin necesidad de pasar por los anteriores a él
- **Linealidad:** como están almacenados los elementos que la componen
    - Lineal: guardan una relación de adyacencia ordenada donde a cada elemento le sigue uno y le precede uno, solamente.
    - No Lineal: Para un elemento dado pueden existir 0, 1 ó más elementos que le suceden y 0, 1 ó más elementos que le preceden.

Registros

Tipo de datos que permite agrupar diferentes clases de datos en una estructura única bajo un solo nombre.
Características:

1. Los valores pueden ser de distinto tipo (**HETEROGÉNEA**)
2. los valores almacenados se denominan campos, donde cada uno tiene un identificador
3. El almacenamiento ocupado es fijo (**ESTÁTICA**)

Declaración

```
Type
nombre = record
  campo1: tipo;
  campo2: tipo;
  ...
end;
Var
 d1, d2: nombre;
```

- Se nombra cada campo.
- Se asigna un tipo a cada campo.
- Los tipos de los campos deben ser estáticos.

Asignación

```
Begin
  d1:= d2;  → asignó 2 registros iguales
End;
```

Para acceder a un campo: variableregistro.nombrecampo

- **NO se puede leer, imprimir o comprar directamente una variable tipo registro** se debe leer, imprimir o comparar por campo.

CLASE 6

ARREGLOS

Es una estructura de datos compuesta(almacena más de un valor al mismo momento) que permite acceder a cada componente por una variable índice, que da la posición de la componente(elemento dentro del arreglo) dentro de la estructura de datos.

Características:

1. HOMOGÉNEA: los elementos son del mismo tipo
2. ESTÁTICA: el tamaño no cambia durante la ejecución
3. INDEXADA: para acceder a un elemento del arreglo se va a tener que usar un índice(ordinal)

- Tipo de dato compuesto definido por el programador

Clasificación:

1. Vectores: tipo de dato arreglo con un solo índice (sólo una dimensión)
2. Matrices: tipo de dato arreglo con 2 indices (2 dimensiones)
3. Tensores

Declaración

```
Type                                  → Rango (índice) ordinal
    nombre= array [rango] of tipo;    → Tipo debe ser estático
```

Asignación de valor

```
Program uno;
 type
   numeros= array [1..7]  of integer;
var
   VN: numeros;
begin
   VN[3]:= 7;
end.
```

→ NO se puede leer directamente una variable tipo arreglo
read(VN);

→ NUNCA voy a poder saltear valores al cargar

→ NO se puede usar una función por ser una variable compuesta

Carga

```
Program uno;
Const
  tam = 7;
type
  numeros= array [1..tam]  of integer;
```

```
procedure cargar (var a:numeros);
var
  i:integer;
begin
   for i:= 1 to tam do
```

```
var
  VN: numeros;
begin
  cargar (VN);
end.
```

```
    read (a[i]);
    sumar DimL +1
end;
```

Impresión

- No se puede imprimir directamente una variable de tipo arreglo
- Se imprime elemento a elemento

```
Program uno;
Const
  tam = 7;
type
  numeros= array [1..tam]  of integer;
var
 VN: numeros;
begin
   cargar (VN);
   imprimir (VN);
end.
```

```
procedure imprimir(a:numeros);
var
  i:integer;
begin
   for i:= 1 to tam do
      write (a[i]);
end;
```

Recorridos

- <u>total:</u> se recorre todo el vector (for)
- <u>Parcial:</u> se recorre hasta que pase algo (while)

Dimensión

- <u>Física:</u> → para cargar datos
    - Se especifica en el momento de la declaración y determina su ocupación máxima de memoria
    - Es la cantidad máxima de elementos que el arreglo **puede tener cargado**
- <u>Lógica:</u> → para recorrer el arreglo
    - Es cuantos elementos efectivamente tiene cargado
    - Se determina cuando se cargan contenidos a los elementos del arreglo
    - Nunca va a superar a la física

Agregar → Siempre al final del arreglo

- *Pseudo:*
  verificar si hay espacio
  agregar al final del elemento
  incrementar la cantidad de elementos

```
Procedure agregar (var a :números; var dim:integer; var pude:boolean;valor:integer);
Begin
  pude:= false;
  if ((dim + 1) <= física) then begin   → verificar si hay lugar
   pude:= true;   → solo para enviar algún tipo de información al principal(puede estar o no)
   dim:= dim + 1;              → incrementar la DimL
   a[dim]:= valor;             → agregar el valor
  end;
end.
```

Insertar→ agregar un valor en el arreglo en una posición(debe ser válida) determinada, para esto hay que hacer un espacio(correr un lugar lugar para delante los elementos, reemplazar y sumar el elemento a la dimL)

- *Pseudo:*
  Verificar si hay lugar
  Verificar que la posición sea válida.
  Hacer lugar para poder insertar el elemento.
  Incrementar la cantidad de elementos

```
Procedure insertar(var a :números; var dim:integer;valor:integer;pos:integer;var
                        pude: boolean);
Var
  i:integer;
Begin
  pude:= false;
  if (((dim + 1)<=física) and (pos<=dim) and (pos>=1))then → verifico lugar y posición
    begin                                                    válida
     for i:= dim down to pos do        → hacer lugar
       a[i+1]:= a[i];
     pude:= true;
     a[pos]:= valor;      → inserto el valor
     dim:= dim + 1;       → incremento la cantidad de elementos
    end;
end;
```

Eliminar

- *Pseudo:*
  Verificar que la posición sea válida.
  Hacer el corrimiento.
  Decrementar la cantidad de elementos.

```
Procedure eliminar (var a :números; var dim:integer;pos:integer;var pude:boolean);
Begin
  pude:= false;
  if ((pos>=1)and (pos<=dim))then begin   → verificar la posición
    for i:= pos to (dim-1) do  → hacer el corrimiento
      a[i]:= a[i+1];
    pude:= true;
    dim:= dim - 1;                → Decrementar la cantidad de elementos
  end;
end;
```

Buscar

***Busqueda lineal:** comenzar desde el inicio de la estructura comparando cada uno de los elementos hasta encontrarlo o llegar hasta el final**

- V Desordenado: recorrer todo el vector(en caso de no encontrarlo) o detenerme cuando encuentre el elemento
- V Ordenado: búsqueda mejorada y dicotómica

V Desordenado

```
function buscar (a:números; dim:integer, num:integer): boolean;
Var
  pos:integer;
  esta:boolean;
Begin
  esta:= false;
  pos:=1;
  while ( (pos <= dim) and (not esta) ) do
     begin
       if (a[pos]=num) then esta:=true
       else
          pos:= pos + 1;
     end;
   buscar:= esta;
end.
```

V Ordenado

- Búsqueda Mejorada: El número de comparaciones es en promedio ((dimL+1)/2)

```
function buscar (a:números; dim:integer, num:integer): boolean;
Var
  pos:integer;
Begin
```

```
  pos:=1;
  while ( (pos <= dim) and (a[pos]<num)) do
    begin
      pos:= pos + 1;
    end;
  if ( (pos <= dim) and (a[pos]= num)) then buscar:=true
  else buscar:= false;
end.
```

- Búsqueda Dicotómica→ El número de comparaciones es en promedio

(1+log2(dimL+1)/2)

- *Pseudo:*

Se calcula el elemento que está en la posición del medio
Si es el elemento que busco, entonces la búsqueda termino
Si NO es el elemento que busco, entonces
Comparo contra el valor del medio
Elijo del vector la mitad que me convenga(derecha o izq) y repito

```
Procedure  BusquedaBinaria ( Var vec: numeros, dimL: integer,bus: integer; var ok :
                                    boolean);
 Var
    pri, ult, medio : integer;
 Begin
     ok:= false;
     pri:= 1 ;  ult:= dimL;  medio := (pri + ult ) div 2 ;
     While  ( pri < = ult ) and ( bus <> vec[medio]) do
       begin
         if ( bus < vec[medio] ) then
           ult:= medio -1 ;
         else pri:= medio+1 ;
         medio := ( pri + ult ) div 2 ;
       end;
     if (pri <=ult) and (bus = vec[medio]) then ok:=true;
   end;
```

Ordenar

- formas:
  1-Selección
  2-Intercambio
  3-Inserción

1-Selección

- *Pesudo:*

Es un algoritmo de dimL pasadas
Para cada pasada i

Se elige el mínimo en el vector a partir de la posición (i+1) hasta el final
Si el mínimo de vector es más chico que lo que está almacenado en la posición i del vector se intercambia.

```
Procedure Ordenar (var v: numeros; dimLog: integer);
var i, j, p, item: integer;
begin
    for i:=1 to dimLog-1 do begin → busca el mínimo v[p] entre v[i], ..., v[N]
       p := i;
       for j := i+1 to dimLog do
          if v[ j ] < v[ p ] then p:=j;
      item := v[ p ];                    → intercambia v[i] y v[p]
      v[ p ] := v[ i ];
      v[ i ] := item;
    end;
end;
```

CLASE 9: PUNTEROS

Como la capacidad del CPU es limitada. La memoria va a ser limitada, tanto la dinámica como la estática, que integran la CPU,serán limitadas.((((rever))))

Las variables ocupan espacio dentro de la memoria(varían según el idioma):

↓ Integer: 4 bytes

↓ Char: 1 byte

↓ Real: 8 bytes

↓ Boolean: 1 byte

↓ Registro: la suma de los campos

↓ Arreglos: dimensión Física * tipo elemento

↓ Puntero: 4 bytes

↓

↦ Estáticas: no pueden cambiar el tamaño que ocupan

↓ reservan memoria cuando se declaran

↓

↦ Dinámicas: pueden cambiar su tamaño durante la ejecución del programa

PUNTERO variable usada para ALMACENAR una **DIRECCIÓN DE MEMORIA**, donde

se encuentra el dato realmente.

- La dirección se encuentra en la memoria dinámica, esta dirección se relaciona con un espacio en memoria estática
- En la memoria estática SIEMPRE va a ocupar 4 bytes
- Puede reservar y liberar espacio en el programa
- Un dato referenciado NO tiene memoria

Características:

1. Es un tipo de DATO SIMPLE que contiene una dirección(que se encuentra en la memoria dinámica).
2. Sólo pueden apuntar a direcciones almacenadas en memoria dinámica(heap)
3. Sólo puede apuntar a un ÚNICO tipo de dato.
4. Se indica con ^ (para acceder a su contenido)

Declaración

TYPE

identificador= ^TipoVariableApuntada; → Cualquier tipo de dato

Creación new (variable puntero); → a la variable p se le asigna una dirección, el lenguaje le asigna una dirección(el usuario no puede asignarla)

```
Program uno;
Type
  puntero = ^integer;
Var
 num:integer;
 p:puntero;
Begin
 new (p);
 …
End.
```

Eliminación dispose(variable puntero)

- Asigna a que p ya no tiene más direcciónes(deja al puntero sin la dir asignada), cortando el enlace que había con su dirección, y **libera la zona de memoria** para poder volver a ser utilizada por ese u otro puntero

```
Program uno;
Type
 puntero = ^integer;
Var
 p:puntero;
Begin
 new (p);
  …
```

```
  dispose(p);
End.
```

Liberación p:=nil;

- Deja al puntero sin la dirección asignada, cortando el enlace que había con su dirección. PERO el espacio de **memoria queda ocupada**,aunque no se pueda acceder(se libera cuando el programa termina)

```
Program uno;
Type
  puntero = ^integer;
Var
  p:puntero;
Begin
  new (p);
  …
  p:= nil;
End.
```

Asignación de dos variables puntero := → NO se debe asignar una dirección manualmente

```
Program uno;
Type
   puntero = ^integer;
Var
  q:puntero;
  p:puntero;
Begin
   new (p);
   q:=p;      → Asigna la misma dirección a q de p
End.                    ↪ Cuando se modifique el contenido de la dirección, cambiará en los 2
```

Acceder al contenido de una variable puntero ^

- el puntero debe tener una dirección asignada para poder asignarle contenido

```
Program uno;
Type
  puntero = ^integer;  → Están permitidas todas las operaciones para ese tipo
Var
 p:puntero;
Begin
  new(p);
  p^:= 123;
End.
```

<u>Acciones permitidas:</u>

- p = nil → compara si el puntero p no tiene dirección asignada
- p = q → compara si los punteros p y q apuntan a la misma dirección de memoria
- p^ = q^ → compara si los punteros p y q tienen el mismo contenido

Acciones NO permitidas:

- read (p)
- write (p)
- p:= ABCD → asignar dirección de forma manual
- p>q → preguntar si el puntero tiene una dirección mayor a otro

Ocupación de memoria

- En su declaración siempre va a ser de 4 bytes(en la memoria estática)
- En su inicialización ocupa sus 4 bytes(del puntero), más el contenido que tiene (en la memoria dinámica)

CLASE 10: LISTAS

Conjunto de elementos HOMOGÉNEOS(del mismo tipo), con una relación LINEAL (todos los elementos van a tener sucesor y un predecesor, en el curso: listas simple solo el sucesor de cada elemento) y DINÁMICA (puede agregar y eliminar elementos durante la ejecución del programa)

Características:

1. compuesta por NODOS (que se conectan por medio de enlaces o punteros)
2. Para AGREGAR un elemento a la lista usare la operación (NEW) y para ELIMINAR un elemento usare(DISPOSE)
3. Pueden no estar almacenados en una lista contigua en la memoria. Los elementos pueden estar DISPERSOS en toda la memoria, pero mantienen el ORDEN lógico INTERNO
4. Tipo de dato COMPUESTO definido por el programador

Declaración → la lista es un puntero que es un registro, que va a tener el tipo de dato y la dirección del puntero siguiente

```
Type
   nombreTipo= ^nombreElemento;
   nombreElemento= Record
                     elementos : tipoElemento;
                     punteroSig: nombreTipo;
                   end;
Var
  Pri: nombreTipo;    → {Memoria estática reservada}
```

Crear una lista vacía → Significa indicar que la lista no tiene dirección de comienzo asignado (el puntero inicial de la lista no tiene una dirección asignada → nil)

```
Program uno;
Type listaE= ^datosEnteros;
       datosEnteros= Record
       elem:integer;
       sig:listaE;
   end;
Var
 PriE: listaE;
Begin
 PriE:= nil;    →    se puede modularizar
End.
```

```
Procedure crearLista (var p: listaE);
 begin
   p:= nil;
 end;
```

Agregar un elemento al inicio → por cada elemento a agregar, hay que modificar el puntero inicial

- *pseudo:*

  Reservo espacio en memoria.
  si (es el primer elemento a agregar)
  asigno el puntero inicial.
  sino
  actualizo el puntero al siguiente del nuevo elemento.
  actualizo el puntero inicial de la lista

```
Procedure agregarAdelante (var p:listaE; valor:integer);
Var
 aux:listaE;
Begin
 new (aux); aux^.elem:= valor; aux^.sig:=nil;   → creación de espacio
 if (p = nil) then
     p:= aux                → evaluación del caso y reasignación de punteros
 else
     begin
       aux^.sig:= p;
       p:=aux;
     end;
End;
```

Recorrer una lista → Avanzar por cada uno de los elementos

- *pseudo:*

  Inicializo una variable auxiliar
  mientras (no sea el final de la lista)
  proceso el elemento

avanzo el puntero auxiliar

```
Program uno;
Type listaE= ^datosEnteros;
    datosEnteros= Record
       elem:integer;
       sig:listaE;
    end;
Var
 PriE: listaE;
 valor:integer;
Begin
  PriE:= nil;
  cargarLista (PriE);
  imprimirLista(priE);
End.
```

```
Procedure cargarLista (var p:listaE);
Var
  aux:listaE;
  valor:integer
Begin
   read (valor);
   while (valor <> 0) do
        begin
Hecho antes →  agregarAdelante(p,valor);
         read (valor);
        end;
End;
```

```
Procedure imprimirLista (p:listaE);
Begin
 while  (p <> nil) do
   begin
    write (p^.elem); → proceso el elemento
    p:= p^.sig;
   end;
End;
```

Agregar un elemento al final → Recorrer la lista, llegar al final y reacomodar los punteros

- ***pseudo 1:*** → ineficiente(solo lleva el puntero inicial)

  Genero espacio de memoria y cargo los datos aux
  Inicializo un puntero auxiliar actual (act)
  mientras (no se llegue al último elemento)
  avanzo el puntero actual (act)
  Reacomodo los punteros

↪ Genero espacio para el nuevo elemento

↪ Recorro la lista hasta llegar al último elemento.

↪ Reasigno los punteros

→ <u>Desventaja:</u> tiene que pasar por todos los elementos para llegar al final

```
Procedure agregarAlFinal (var p:listaE;valor:integer);
Var
 aux,act:listaE;
Begin
 new (aux); aux^.elem:=valor;   aux^.sig:=nil;
```

```
  if (p = nil) then
    p:= aux
  else begin
    act:= p;
    while (act^.sig <> nil) do
      act:= act^.sig;
    end;
    act^.sig:= aux;
  end;
End;
```

- ***pseudo 2:*** → eficiente(llevo dos punteros uno inicial y otro el final)

  Genero espacio de memoria y cargo los datos aux
  si (es el primer elemento)
  asigno el primer y último puntero de la lista
  sino
  actualizo el puntero siguiente del último elemento
  actualizo el último elemento

↪ Genero espacio para el nuevo elemento

↪ Utilizo un puntero que mantiene la dirección del último elemento.

↪ Reasigno los punteros.

↪ Reasigno la dirección del último elemento

→ Ventaja:como tengo el puntero final, no tengo que recorrer toda la lista

```
Begin
  PriE:= nil;
  read (valor);
  agregarAlFinal2 (priE,ult,valor);
End.
Procedure agregarAlFinal2 (var p,ult:listaE; valor:integer);
Var
  aux:listaE;
Begin
  new (aux); aux^.elem:= valor; aux^.sig:=nil;
  if (p = nil) then
     p:= aux; ult:=aux;
  else
      begin
        ult^.sig:= aux;
        ult:= aux;
      end;
End;
```

Insertar → agregar un elemento manteniendo el orden en la lista

Casos:

1. La lista está vacía.
   - *Pseudo:*
     Género espacio para el nuevo elemento
     Asigno como dirección inicial de la lista este nuevo espacio

2. El elemento a insertar va al comienzo de la lista.
   - *Pseudo:*
     Genero espacio para el nuevo elemento
     Preparo los punteros para el recorrido.
     Reorganizo punteros.
     Actualizo la dirección del primer puntero.

3. El elemento a insertar va al medio de la lista.
   - *Pseudo:*
     Genero espacio para el nuevo elemento
     Preparo los punteros para el recorrido.
     Reorganizo punteros.

4.El elemento a insertar va al final de la lista.
   - *Pseudo:*
     Genero espacio para el nuevo elemento
     Preparo los punteros para el recorrido.
     Reorganizo punteros.

```
Procedure insertar (var p:listaE valor:integer);
Var
  aux,act,ant:listaE;
Begin
  new (aux); aux^.elem:= valor; aux^.sig:=nil;
  if (p = nil) then
    p:= aux
  else begin
    act:= p; ant:=p;
    while (act <> nil) and (act^.elem<aux^.elem) do begin
      ant:=act;   act:= act^.sig;
    end;
  end;
if (act= p) then
 begin
   aux^.sig:= p;     p:= aux;
 end
else
 begin
  ant^.sig:=aux;    aux^.sig:= act;
 end
End;
```

Buscar un elemento se debe recorrer la lista y cuando se llegue al final o se encuentre al elemento detenerse

Lista Desordenada:

```
Function buscar(pri:listaE; valor:integer):boolean;
Var
 act:listaE;
 encontré:boolean;
Begin
  act:=p;
  encontré:= false;
  while (act <> nil) and (encontré = false) do
  begin
    if (act^.elem = valor) then encontré:= true
    else
     act:= act^.sig;
  end;
  buscar:= encontre;
  end;
```

Lista Ordenada:

```
Function buscar(pri:listaE; valor:integer):boolean;
Var
 act:listaE;
 encontré:boolean;
Begin
  act:=p;
  encontré:= false;
  while (act <> nil) and (act^.elem < valor) do
  begin
    act:= act^.sig;
  end;
  if (act <> nil) and (act^.elem = valor) then
      encontre:= true;
  buscar:= encontre;
  end;
```

Eliminar un elemento recorrer la lista y cuando el elemento coincida eliminar utilizando un dispose

Casos:

1. La lista está vacia.
2. El elemento a eliminar está al comienzo.
    - *Pseudo:*
      Preparo los punteros para el recorrido.
      Reorganizo punteros, incluyendo al puntero inicial.

Realizo el dispose.

3. El elemento a eliminar no está al comienzo.
   - *Pseudo:*
     Preparo los punteros para el recorrido.
     Reorganizo punteros
     Realizo el dispose.

4.El elemento a eliminar no está.
   - *Pseudo:*
     Preparo los punteros para el recorrido.
     Como el elemento no está no hago nada.

```
Procedure eliminar(var p:listaE valor:integer);
Var
 act,ant:listaE;
Begin
  act:=p;
  while (act <> nil) and (act^.elem <> valor) do begin
    ant:=act;
    act:= act^.sig;
  end;
  if (act <> nil) then begin
     if (act = p) then
        p:= p^.sig;
    else
      ant^.sig:= act^.sig;
   dispose (act);
 end;
End;
```

Elimina solo un elemento de la lista

CLASE 12: ARRAYS / LISTAS

	Array	Lista
características	Estática Homogénea Indexada	Dinámica Homogénea Lineal
Almacenamiento	Al ser **ESTÁTICA**, se encuentra alojado en posiciones de memoria consecutivas.	Al ser **DINÁMICA**, cada vez que se necesita agregar un elemento se solicita memoria, por lo tanto no existe relación entre las posiciones de memoria de los elementos(las posiciones son determinadas por el lenguaje).

Agregar	**Adelante:** Se debe hacer un corrimiento para generar el espacio e incrementar la dimensión lógica. **Puede no poder hacerse la operación.** **Al Final:** Se agrega el elemento en la dimensión lógica + 1 y se incrementa la dimensión lógica. **Puede no poder hacerse la operación.**	**Adelante:** Se debe solicitar un espacio y reacomodar los punteros. **Siempre puede hacerse la operacion** **Al Final:** Existen dos alternativas: - Recorrer toda la lista, solicitar espacio y reacomodar los punteros. -Llevar un puntero que contenga la dirección del último elemento de la lista, solicitar espacio y reacomodar los punteros. **Siempre puede hacerse la operación.**
Insertar	Debe buscarse la posición, una vez encontrada se realiza un corrimiento para hacer lugar, se carga el elemento y se incrementa la dimensión lógica. **Puede ser que no se pueda hacer la operación.**	Se solicita espacio para el nuevo dato, para el cual debe buscarse la posición y una vez encontrada se reorganizan los punteros. **Siempre puede hacerse la operación**.
Eliminar (Ordenado y Desordenado)	Debe buscarse el elemento, una vez que el elemento es encontrado, realizar un corrimiento desde la posición donde está el elemento hasta el final y decrementar la dimensión lógica. Si es ordenado se puede utilizar la búsqueda dicotómica de orden log, para mejorar los tiempos(mejor que listas)	Debe buscarse el elemento, una vez que el elemento es encontrado, se reorganizan los punteros.
Acceso a estructura	Al ser **INDEXADA**, para acceder a la posición I (si es válida), se utiliza el acceso directo.	Al ser **LINEAL**, para acceder a la posición I, se debe pasar por los I-1 elementos anteriores.
Memoria Ocupada	Al ser **ESTÁTICA**, siempre ocupa la misma cantidad de memoria estática. *la que ocupe mas va a variar dependiendo de si hay elementos o no*	Al ser **DINÁMICA**, su tamaño cambia al agregar o eliminar elementos
Parámetros en las operaciones	Cuando un arreglo que es pasado por valor **por valor** se copia todo el arreglo en el parámetro correspondiente. Cuando es pasado **por referencia** se pasa sólo la dirección.(en este caso se pasan 4 bytes que es lo mismo que un puntero de una lista).	Cuando una lista es pasada **por valor** se copia la dirección inicial de la lista. Cuando es pasada **por referencia** se pasa la dirección inicial de la lista.

CLASE 13: CORRECCIÓN Y EFICIENCIA

CORRECCIÓN

Un **programa es correcto** si se realiza de acuerdo a sus especificaciones (cumple con los requerimientos propuestos).

TÉCNICAS:

- **TESTING**: Dar evidencias de que el programa hace lo esperado, se deben diseñar un conjunto de casos de pruebas para ver si el programa funciona correctamente
    - Decidir cuáles aspectos del programa deben ser testeados y encontrar datos son necesarios para esto
    - Determinar qué resultado se espera para cada situación
    - Poner atención en los casos límite.
    - Diseñar casos de prueba sobre la base de lo que hace el programa y no de lo que se escribió del programa. Lo mejor es hacerlo antes de escribir el programa.

Una vez que ya tengo el programa y el plan de pruebas se debe analizar los casos de prueba y corregir si hay errores. Esto se debe hacer hasta que ya no hayan errores.

- **DEBUGGING**: Es el proceso de descubrir y reparar la causa del error.Agregar sentencias extras al programa que permite ver dónde está el problema(ej. write)
    - Los errores pueden provenir de 2 fuentes
        - El diseño no es el adecuado
        - El programa no está escrito correctamente
    - Tipos de Errores
        - Sintácticos: se detectan en la compilación
        - Lógicos: se detectan en la ejecución
        - De sistema: son raros, ej. corte de luz

- **WALKTHROUGHS**: Es recorrer un programa(ir leyéndolo) frente a una audiencia.
    - La persona no comparte preconceptos(lo analiza objetivamente) y está predispuesta a descubrir errores u omisiones.

- **VERIFICACIÓN**: Es controlar que se cumplan las pre y post condiciones del programa.

Para verificar si un programa es correcto se pueden aplicar todas las tecnicas mas de una vez

EFICIENCIA

Se la define como una métrica de calidad de algoritmos, asociada con una **utilización óptima de los recursos del sistema** de cómputo donde se ejecutará el algoritmo.

Estudia el tiempo que tarda un algoritmo en ejecutarse y la memoria que requiere para su ejecución.

Está relacionada con:

- Datos de entrada (tamaño y cantidad)
- El tiempo del algoritmo base
- Calidad del código generado por el compilador (no lo podemos controlar)
- Naturaleza y rapidez en la ejecución de las instrucciones de máquina

Tiempo del Algoritmo:

- Existen algoritmos que el tiempo de ejecución depende de la cantidad de datos de entrada o su tamaño
- Existen otros algoritmos donde el tiempo de ejecución es una función de la entrada "específica"(se elige el peor caso)

Análisis Empírico: Se escribe el código, se ejecuta en una máquina y se mide el tiempo

- Ventaja:
  - El tiempo es preciso
  - Fácil de realizar
- Desventaja
  - Tengo que escribir el programa primero
  - Obtiene valores exactos para una máquina y unos datos determinados (dependiente de la máquina donde se ejecute)
  - Requiere ejecutarlo varias veces

Análisis Teórico: Se hace un análisis sobre algunas de las instrucciones que hace el programa y se estima un tiempo que el algoritmo necesita para su ejecución, **no se necesita ejecutarlo**.

- Ventajas
  - Obtiene valores aproximados
  - Es aplicable en la etapa de diseño de los algoritmos (se puede aplicar sin necesidad de implementar el algoritmo)
  - El análisis es independiente de la máquina donde se ejecute

Se define **T(n) al tiempo de ejecución** de un programa con una entrada de tamaño n. La idea es expresar la función del tiempo de alguna manera, buscando entender su comportamiento con grandes volúmenes de datos:

T(n) = ...

- **El Tiempo es la suma de cada una de las instrucciones**
- A partir del tiempo se calcula el **orden del algoritmo** (es elegir el máximo entre las instrucciones del programa)**es elegir como esta n(si es constante, logarítmico, orden 1,etc)**

1. Considerar el número de operaciones elementales que emplea el algoritmo.
2. Considerar que una operación elemental utiliza un tiempo constante para su ejecución, independientemente del tipo de dato con el que trabaje.
3. Suponer que cada operación elemental se ejecutará en una unidad de tiempo (dejando de lado la magnitud).
4. Suponer que una operación elemental es una asignación, una comparación o una operación aritmética simple.

asignaciones = tiempo constante 1
suma = suma de sus asignaciones
**** Los comentarios, declaraciones y operaciones de entrada/salida (Read / Write), no se consideran al realizar el cálculo****

REGLAS GENERALES

1. **Sentencias Consecutivas:** maximo de todas las instrucciones
   max(inst1, inst2)
2. **For / For Anidados:** Se debe calcular la cantidad de operaciones elementales que se ejecutan dentro del FOR y multiplicarla por la cantidad de veces que se ejecuta la instrucción FOR.

```
Program temperaturas;

Var valor, total: real;
Begin
 total:= 0;
 for i:= 1 to 30 do begin
   read (valor);
   total:= total + valor;
 end;
 prom:= total div 30;
 write('Temperatura Promedio:', prom);
end;
```

la instrucción FOR realiza:
asignación inicial i:=1 (1)
testeo de i <=30 (31)
incrementos de i:= i+1 (30*2) entonces 1+31 + 60 = 92 op
En general : **3*n+2, siendo n la cantidad de repeticiones o 2n+2...**

Total -> (2*30) + (3*30+2) + 3 = 155 op. elem.

3. **While / Repeat...Until:** Se debe calcular la cantidad de operaciones elementales que se ejecutan dentro del WHILE y multiplicarla por la cantidad de veces que se ejecuta el WHILE. Como no se conoce esa cantidad se considera el PEOR CASO. Por ejemplo, se supone una cantidad
4. **If / Else:** En el caso de una sentencia IF en su forma completa (then/else), debe calcularse la cantidad de operaciones que se realizan en cada parte y se debe elegir aquella que consuma más tiempo
   max(If, Else)