

El presente documento es un apunte que sintetiza los conceptos y sintaxis básicas que utilizaremos en la materia. Para un mayor y mejor detalle consultar los libros que recomienda la cátedra.

Algoritmo	3
Formas de representar un algoritmo	3
Pseudocódigo.....	3
Diagramas de flujo.....	4
Lenguajes de Programación.....	4
Programa.....	5
Programa Fuente o Código Fuente	5
Lenguajes Compilados	5
Programa Objeto o Código Objeto	5
Programa Ejecutable.....	5
Lenguajes Interpretados.....	5
Esquema de Entrada –Proceso – Salida.....	6
Datos	6
Constantes.....	6
Tipo de Datos	7
Variables	7
Nombre de Variables	7
Comentarios.....	8
Entrada y Salida de datos.....	8
Operadores y operaciones.....	10
Operador de asignación	10
Operadores aritméticos	10
Estructura de Control Secuencial	11
Estructura de Control Condicional	11
Condición Lógica	13
Ejemplo Problema Condicional -Formato 1 Sin parte Falsa	13
Ejemplo Problema Condicional – Formato 2: Utilizando la parte falsa.....	14
Operadores Lógicos.....	16
Estructura de Control Iterativa	17
Variables Contadores.....	17
Variables Acumuladores o Sumadores.....	19
Ciclos Exactos vs Ciclos Condicionales.....	20
Componentes del ciclo	20
Ciclo For.....	21
Secuencias de Números: Tipo range	21
range(n).....	21
range(m, n).....	21
range(m, n, p)	22
Números al Azar.....	23
Guía para desarrollar un algoritmo	24
Programación Modular	25
Funciones.....	25
Llamada a una función	26
Correspondencia de parámetros.....	27

Listas.....	27
Operaciones en Listas	27
Crear una lista vacía	27
Crear una lista con n valores numéricos	27
Agregar valores a una lista	27
Subíndice para acceder a una posición de la lista.....	27
Cantidad de elementos de una lista.....	28
Eliminar el último elemento de la lista.....	28
Eliminar un elemento que se encuentra ubicado en una posición i	28
Modificar valores de la lista	29
Intercambiar valores de posición	30
Mostrar los valores de una lista	30
Funciones y Listas	30
Búsqueda Secuencial	33
Métodos de Ordenamiento.....	33
Ordenamiento por Selección	34
Ordenamiento por Burbujeo	34
Ordenamiento por Burbujeo Mejorado.....	34
Ordenamiento por Inserción	35
Búsqueda Binaria.....	35

Algoritmo

Un algoritmo es una secuencia finita repetible y correcta de pasos, que llevan a la solución de un problema dado, los pasos son acciones y condiciones llamadas instrucciones. En general existen varios algoritmos para resolver un problema.

Características importantes de un algoritmo:

Secuencia: Los pasos que la componen tienen que estar ordenados, el orden es importante.

Finita: La secuencia en algún momento tiene que tener un final.

Repetible: La secuencia tiene que ser repetible, si partimos de los mismos datos de entrada, el resultado tiene que ser siempre el mismo. Como una cuenta, $2 + 3$ siempre tiene que dar 5, lo mismo ocurre con los algoritmos.

Correcto: debe resolver el problema para el cuál fue diseñado.

Esta materia se trata de la construcción de algoritmos que van a resolver problemas de la práctica. Se recomienda trabajar con lápiz.

Formas de representar un algoritmo

Un algoritmo se puede definir a través de un pseudocódigo o mediante diagramas de flujo.

Pseudocódigo

Significa escribir las instrucciones en un lenguaje informal (utilizando vocabulario cotidiano). Este procedimiento facilita su escritura en los lenguajes de programación.

Ejemplo: algoritmo para cruzar la calle

Comienzo

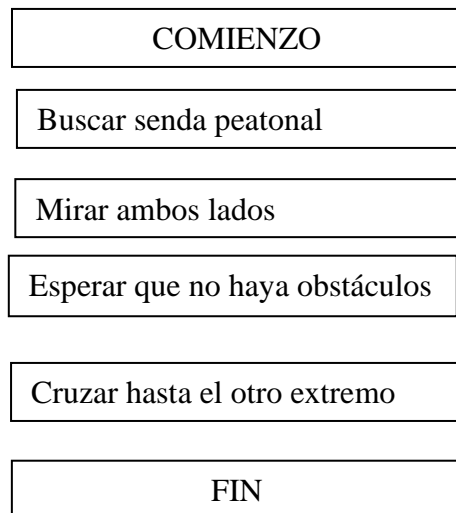
1. Buscar una senda peatonal
2. Mirar a ambos lados
3. Esperar que no haya obstáculos
4. Cruzar hasta el otro extremo

Fin

Diagramas de flujo

Representación gráfica de un algoritmo con símbolos estandarizados, que nos muestra cada uno de los pasos a seguir en la solución de un problema.

Ejemplo: el mismo algoritmo para cruzar la calle sería:



Existen varios modelos diferentes para representar un algoritmo utilizando diagramas de flujos.

Lenguajes de Programación

Es un conjunto de palabras, símbolos y reglas sintácticas mediante los cuales puede indicarse a la computadora los pasos a seguir para resolver un problema. Al igual que los idiomas sirven de vehículo de comunicación entre seres humanos y las computadoras.

Los lenguajes de programación pueden clasificarse por diversos criterios, siendo el más común su nivel de semejanza con el lenguaje natural y su capacidad de manejo de niveles internos de la máquina, una posible clasificación sería:

Lenguaje de bajo nivel: Lenguaje de máquina: Es el lenguaje que entiende el procesador

Lenguaje de medio nivel: Lenguaje C

Lenguaje de alto nivel: Lenguaje Python, Pascal, Java, etc.

Programa

Para generar un programa, o sea, un ejecutable que pueda entender el procesador, se realiza una serie de pasos dependiendo del lenguaje de programación que se utiliza. En cualquiera de los casos, se parte del programa fuente o código fuente para lograr un programa ejecutable utilizando un Compilador o un Intérprete dependiendo del lenguaje de programación.

Programa Fuente o Código Fuente

Es el conjunto de instrucciones escritas en un lenguaje de programación por el programador, los cuales pueden ser Java, C, Python, Pascal. En esta materia utilizaremos el lenguaje Python 3.x

El código Fuente en este primer estado no es directamente ejecutable por la computadora, sino que debe ser traducido a lenguaje máquina que sí pueda ser comprendido por el hardware de la computadora. Para esta traducción se usan los llamados compiladores, ensambladores, intérpretes y otros sistemas de traducción.

Lenguajes Compilados

Para los lenguajes que utilizan compilador, se generan los siguientes archivos:

Ejemplo lenguajes que se utiliza compilador: C, Pascal.

Programa Objeto o Código Objeto

Es la traducción del código fuente a lenguaje de máquina mediante el uso de un compilador.

Programa Ejecutable

Esta es la última etapa en el desarrollo del programa, el código ejecutable se obtiene a través del código objeto combinado con otras porciones de código proporcionadas por el fabricante del compilador. El programa ejecutable es el que se ejecutará en la computadora.

Lenguajes Interpretados

Para un lenguaje que utiliza un intérprete, ejemplo Python, no se generan archivos intermedios, el intérprete se encarga de traducir a medida que se va ejecutando cada instrucción.

Durante el curso utilizaremos Python como lenguaje de programación. Es interpretado, por lo que debemos instalar el intérprete en nuestras computadoras para ejecutar los programas que vamos a crear.

Esquema de Entrada –Proceso – Salida

Es una herramienta utilizada por los programadores de sistemas para la solución de un problema. En donde:

Entrada: Son todos los datos que hay que ingresar para la solución del problema.

Proceso: Son las diferentes instrucciones en los cuales se usarán los datos proporcionados por el usuario para resolver el problema.

Salida: La solución del problema.



Datos

Las computadoras son máquinas que procesan datos, esto significa que toman datos de entrada, efectúan operaciones sobre ellos, obteniéndose datos intermedios y, por último, devuelven datos de salida.

Constantes

Las constantes se refieren a valores fijos que no pueden ser alterados por el programa. Por convención, en algunos lenguajes de programación los programadores utilizan letras mayúsculas para nombrar a las constantes, aunque no es obligatorio.

Tipo de Datos

El tipo de dato en cualquier lenguaje indica que valor se guarda en la variable, los tipos de datos con los que trabajaremos son:

int	para números enteros
float	para números reales
str	para caracteres y textos.
bool	para valores <i>True</i> / <i>False</i> exclusivamente.
None	para representar un valor “vacío”

True / *False*: son palabras reservadas, se deben escribir con la primer letra en mayúscula. *False* equivale numéricamente a 0. Cualquier otro número equivale a *True* y su valor por defecto es 1.

En Python los tipos de datos los deduce en el momento en que se crea la variable.

Variables

Los datos se almacenan en variables. No se puede trabajar sobre un dato si no se dispone de una variable que lo represente. Las características de las variables son:

- Posee un nombre que asociamos con el dato.
- Tiene asociada una dirección de memoria donde se guarda el dato.
- Tiene asociada un tipo de dato que describe como es el dato.

Es importante aclarar que las variables simples pueden almacenar o guardar un solo dato a la vez. Agregar otro dato provoca la destrucción del primero.

Nombre de Variables

Los nombres de variables sólo pueden ser combinaciones de números, letras mayúsculas y minúsculas y el guion bajo. El nombre no puede comenzar con un número y no se pueden emplear otros símbolos.

Por convención, podemos utilizar una letra minúscula para comenzar el nombre de una variable y las palabras siguientes, aunque no es obligatorio. Ejemplo:

sumaNumeros
num1
num2
promedio

Para mejor lectura del código fuente, se recomienda utilizar palabras que representen el dato que almacenan.

Nombres a Evitar

Se recomienda NO utilizar los caracteres 'l' (letra ele minúscula), 'O' (letra o mayúscula), o 'I' (letra i mayúscula) como nombres de variables de un solo carácter. En algunas fuentes, estos caracteres son indistinguibles de los numerales uno y cero.

Comentarios

Los comentarios son textos que no son procesados por el compilador. Sirven como información para el programador y documentación para el programa.

Una forma de escribir comentarios en un código Python es utilizando # al inicio de la línea

Ejemplo:

```
# Esto es un comentario
```

Estructura de un programa en Python

El programa fuente se escribe en un lenguaje de programación y debe cumplir las reglas del lenguaje seleccionado.

Las reglas de un programa fuente en general para Python que vamos a utilizar en el curso son las siguientes:

#Programa Nombre: Descripción ◀ Línea de comentario que indica el nombre de nuestro Programa y una descripción de su objetivo principal.

Funciones ◀ Comentario que indica el bloque de declaración de funciones.

def NombreFuncion (argumentos o parámetros de la función):

 <instrucción 1>

 <instrucción 2>

 <instrucción n>

#Inicio Programa Principal ◀ Comentario que indica el bloque de inicio de ejecución del programa.

 <instrucción 1>

 <instrucción 2>

 <instrucción 3>

 <instrucción n>

Entrada y Salida de datos

Para que un programa pueda interactuar con el usuario, se utilizan funciones de entrada y salida de datos, los que nos permiten recibir datos desde el teclado y mostrar información por pantalla. A continuación, se detalla el código que utilizaremos para obtener entrada de datos desde el teclado y mostrar mensajes en la pantalla.

Entrada de datos

Para ingreso de datos desde el teclado utilizaremos la siguiente instrucción:

```
variable= input("Mensaje al usuario")
```

Al ingresar un dato por teclado, el valor se almacenará en la variable asignada en la instrucción. Se debe recordar que input SIEMPRE devuelve un texto, si necesitamos que se guarde en la variable un número para realizar operaciones debemos indicarlo de la siguiente forma:

```
variable= int( input("Mensaje al usuario"))
```

Salida de datos

Para imprimir el mensaje por pantalla, utilizaremos:

```
print('mensaje')
```

Para imprimir una variable:

```
print(variable)
```

Para imprimir mensajes y valores almacenados en variables utilizaremos:

```
print('Mensaje a mostrar ',NombreVariable)
```

El texto que se encuentre entre las comillas simples o comillas dobles será el que se visualizará por pantalla. La variable se reemplaza por el valor almacenado en ella y es lo que se muestra en pantalla, el nombre de la variable NO debe estar entre comillas, ya que sería considerado como parte del mensaje. Se pueden combinar tantos mensajes y variables como se necesite informar.

Ejemplo: Escribir un algoritmo que muestre por pantalla el mensaje Hola Mundo

#Programa PrimerPrograma: Muestra por pantalla un mensaje que dice Hola Mundo

```
print('Hola Mundo')
```

Por pantalla se muestra:

Hola Mundo

Operadores y operaciones

Los operadores permiten manipular los datos almacenados en variables. Los operadores que vamos a utilizar son los siguientes:

Operador de asignación

El operador de asignación que utilizaremos es el símbolo =

Ejemplo:

$a = b$

Esta instrucción guarda o copia el valor de b en a.

Operadores aritméticos

Permiten realizar operaciones aritméticas:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División Real (con decimales)
//	División Entera
%	Módulo (resto de la división entera)
**	Exponente

El operador // no redondea, sino que trunca cociente de la división.

Ejemplos:

$10 // 3$ dará como resultado 3.

$10 \% 3$ dará como resultado 1.

$10 / 3$ dará como resultado 3.33333

El orden de las operaciones sigue las reglas matemáticas aprendidas en la escuela primaria, es decir la separación en términos. Para alterar el orden natural utilizamos paréntesis. Tanto como sean necesarios, pero siempre paréntesis.

Por Ejemplo:

$3 + 5 * 2$ primero resuelve la multiplicación y luego la suma, da como resultado 13

$(3+5) * 2$ primero resuelve la suma y luego la multiplicación, da como resultado 16

$((2+4) + 10 * 2) + (3*8)$ --- da como resultado 50

Estructura de Control Secuencial

La estructura secuencial se refiere a una serie de acciones que se ejecutan una seguida de la otra. Luego de la ejecución de una acción o instrucción, el control se transfiere a la siguiente.

Es importante recordar que la ejecución del programa es lineal y de arriba hacia abajo, podemos simbolizar esta estructura de control de la siguiente forma:

Lineal /
Secuencial:

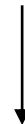


Ejemplo Problema Secuencial:

Ingresa dos números enteros e informar su suma.

#Programa Suma: Suma dos números enteros ingresados por teclado

```
nro1= int( input('Ingrese un número: '))  
nro2= int( input('Ingrese un número: '))  
suma = nro1 + nro2  
print("La suma es: ", suma);
```



Todas las instrucciones se ejecutan una sola vez y finaliza el programa.

Estructura de Control Condicional

Verifica una condición lógica antes de decidir que secuencia de acciones ejecutar. Esta estructura de control puede describirse como “Si la condición se cumple, ejecute la primera secuencia de acciones, sino ejecute la segunda secuencia de acciones. Podemos simbolizar de la siguiente forma:

Condicional /Alternativa:

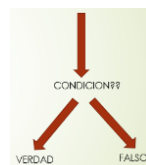
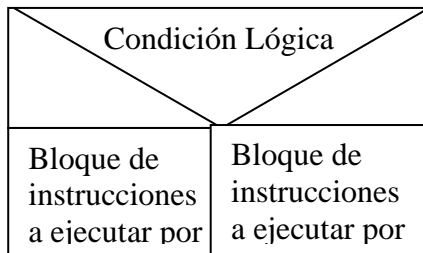
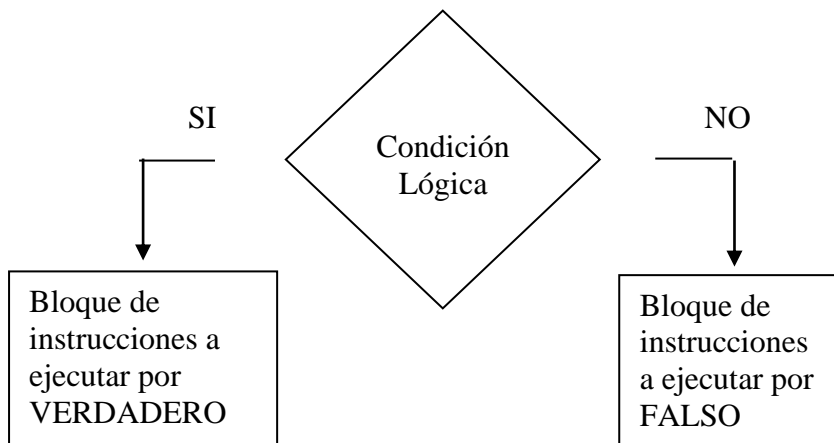


Diagrama Condicional:

Una Forma sería:



Otra forma sería:



También lo podemos representar en el pseudocódigo de la siguiente forma:

```
si Condición Lógica:  
    secuencia de acciones a ejecutar  
    cuando la condición lógica responde  
    VERDADERO  
sino  
    secuencia de acciones a ejecutar  
    cuando la condición lógica responde  
    FALSO
```

Durante el curso utilizaremos pseudocódigo.

Condición Lógica

Definición: Una condición lógica se indica como una expresión que al evaluarla da como resultado verdadero o falso.

Para armar una expresión lógica se utilizan los operadores relacionales que permiten comparar **dos** datos creando una condición lógica, donde se obtendrá una respuesta verdadera o falsa según se cumpla o no la condición.

Operadores relacionales

A continuación, se detallan los símbolos que utilizaremos en el diagrama de flujo para representar las diferentes operaciones relacionales que podemos realizar:

Operador Relacional	Descripción	Condición Lógica
<	Menor	$a < b$
<=	Menor-Igual	$a <= b$
>	Mayor	$a > b$
>=	Mayor-Igual	$a >= b$
==	Igual	$a == b$
!=	Distinto	$a != b$

No se puede cambiar el orden de los símbolos, se deben utilizar como se indican en la tabla, por ejemplo, primero el signo menor < y luego el igual = para comparar por menor o igual.

Ejemplo Problema Condicional -Formato 1 Sin parte Falsa:

Cuando utilizamos la estructura condicional, el bloque de sentencias por falso no es obligatoria, por ejemplo:

Ingresar un valor e informar si es mayor que cuatro.

Datos Entrada: numero

Datos Salida: mensaje o nada

INICIO

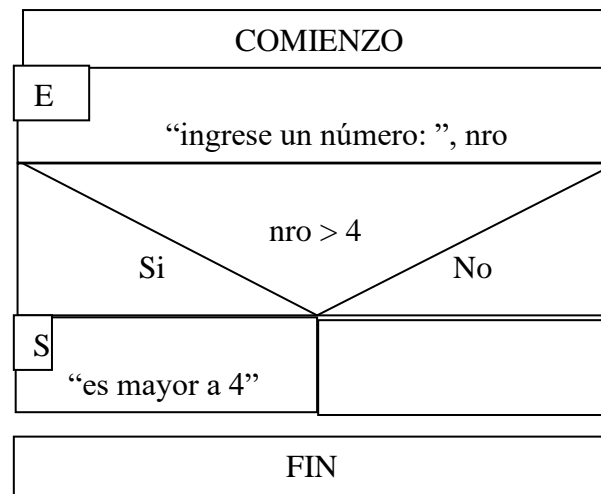
nro = ingresar("Ingrese un número")

si nro > 4 :

 mostrar("El numero ingresado es mayor a cuatro")

FIN

Un modelo en diagrama de flujo sería:



Podemos observar en este ejemplo que por verdadero estamos realizando una instrucción, pero no hay instrucciones a ejecutar por falso. Dependiendo del problema necesitaremos ejecutar o no instrucciones, una o varias, dentro de cada bloque VERDAD o bloque FALSO.

En Python sería:

```
#Programa MayorCuatro: informar si un numero es mayor a cuatro
nro=int(input("Ingrese un número"))
if nro > 4 :
    → print("El numero ingresado es mayor a 4")
```

La flecha no se utiliza, en el ejemplo se indica para representar el espacio obligatorio que se debe respetar para indicar en Python que esa instrucción sólo se ejecute si la condición del if es verdadera.

En Python no hay símbolos que indiquen donde comienzan y terminan los bloques de ejecución por verdad y por falso del condicional. La indentación del código es importante en Python ya que delimita la ejecución de las instrucciones.

Ejemplo Problema Condicional – Formato 2: Utilizando la parte falsa sería, por ejemplo:

Ingresar un número e informar si es mayor, menor o igual que cuatro.

En pseudocódigo sería:

Datos Entrada: numero

Datos Salida: mensaje o nada

INICIO

nro = ingresar(“Ingrese un número”)

si nro > 4 :

 mostrar(“El numero ingresado es mayor a cuatro”)

sino:

si nro < 4 :

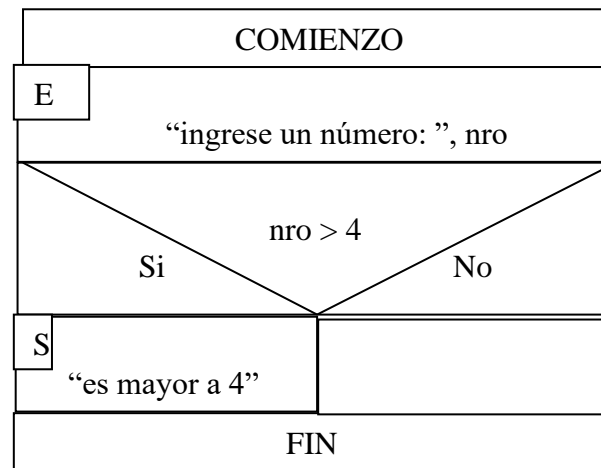
 mostrar(“El numero ingresado es mayor a cuatro”)

sino:

 mostrar(“El numero ingresado es mayor a cuatro”)

FIN

Un modelo en diagrama de flujo sería:



En este caso vemos cómo podemos combinar el uso de condicionales dependiendo de la decisión que quiero tomar en el algoritmo. La ejecución del programa es lineal, de arriba hacia abajo, pero en un punto toma una decisión. Se pueden combinar la cantidad de selección que se crean necesarias para resolver el problema.

En Python:

```
#Programa MayorCuatro
nro=int(input("Ingrese un número"))
if nro > 4 :
    print("El numero ingresado es mayor a 4")
else :
    if ( Nro < 4 ) :
        print ("El numero ingresado es menor a 4")
    else :
        print("El numero ingresado es igual a 4")
```

Operadores Lógicos

A veces es necesario evaluar más de una condición para tomar una decisión. Para ello se utilizan los operadores lógicos que permiten unir más de una condición lógica de diferentes formas creando una condición compuesta.

A continuación, se detallan los símbolos que utilizaremos en el diagrama de flujo para representar las diferentes operaciones relacionales que podemos realizar

Operador Lógico	Descripción	Ejemplo
and	Operador Y (and) , para que sea verdadera, las dos condiciones lógicas deben ser verdaderas.	cond1 and cond2
or	Operador O (or) , para que sea verdadera, al menos una condición lógica debe ser verdadera.	cond1 or cond2

Tabla de verdad del Operador Y (and)

Condición A	Condición B	Condición A and Condición B
VERDADERO	VERDADERO	VERDADERO
VERDADERO	FALSO	FALSO
FALSO	VERDADERO	FALSO
FALSO	FALSO	FALSO

Es importante destacar que al unir condiciones con **and**, solamente se obtendrá como resultado **verdadero** cuando se cumplan **todas** las condiciones unidas por este conector.

Tabla de verdad del Operador O (or)

Condición A	Condición B	Condición A or Condición B
VERDADERO	VERDADERO	VERDADERO
VERDADERO	FALSO	VERDADERO
FALSO	VERDADERO	VERDADERO
FALSO	FALSO	FALSO

Es importante destacar que, al unir condiciones con **or**, solamente se obtendrá como resultado **falso** cuando se **No** cumplan **Todas** las condiciones unidas por este conector.

Estructura de Control Iterativa

Repite una secuencia de acciones mientras se cumpla una condición lógica. Esta estructura de control puede describirse como “Mientras la condición se cumple, ejecuta la secuencia de acciones seguidas con una sangría (espacios) hacia la derecha, cuando la condición se hace falsa, continúa la ejecución del programa”.

Cada repetición se llama “iteración”. Podemos simbolizar de la siguiente forma:

Ciclo / Iterativa:



Existen diferentes estructuras de repetición, por el momento utilizaremos la siguiente Sintaxis iteración

```
while <expresión lógica>:  
    <Instrucción 1 >  
    <Instrucción 2 >  
    < Instrucción n>
```

Variables Contadores

El contador se utiliza para llevar la cuenta de determinadas acciones que se pueden solicitar durante la resolución de un problema. Son variables cuyo valor se incrementa o se decrementa en una unidad, utilizando una constante.

Se debe realizar la inicialización del contador. La inicialización consiste en poner el valor inicial de la variable que representa al contador. Generalmente se inicializa con el valor 0, aunque no siempre es así, depende del problema a resolver.

Ejemplo inicializar un contador llamado cont:

cont = 0

Ejemplo incrementar el contador llamado cont:

cont = cont + 1

Ejemplo ciclo con Contador: Desarrollar un programa que cuente hasta 5.

En pseudocódigo:

INICIO

contador=1

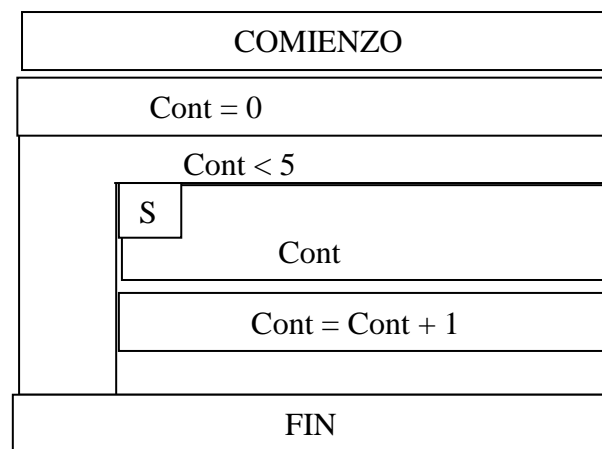
mientras contador <= 5 :

 mostrar(contador)

 contador = contador + 1

FIN

En diagrama, una forma sería:



En Python:

```
#Programa CuentaNumeros
cont = 0
while cont < 5 :
    print (cont)
    cont = cont +1
```

Variables Acumuladores o Sumadores

Un acumulador o sumador es una variable en la memoria cuya misión es almacenar cantidades variables. Son variables cuyo valor se incrementa o se decrementa en un valor que no tiene por qué ser fijo.

Se utiliza para efectuar sumas sucesivas. La principal diferencia con el contador es que el incremento o decremento de cada suma es variable en lugar de constante como en el caso del contador.

En las instrucciones de preparación se realiza la inicialización del acumulador o sumador. La inicialización consiste en poner el valor inicial de la variable que representa al sumador. Generalmente se inicializa con el valor 1, aunque no siempre es así, depende del problema a resolver.

Ejemplo inicializar un sumador llamado sum:

```
sum = 1
```

Ejemplo incrementar el contador según el valor de la variable numero:

```
sum = sum + numero
```

Es importante recordar inicializar el contador o sumador con el valor adecuado ya que necesita del valor anterior para sumarlo, algunos entornos de programación no inicializan las variables automáticamente, por lo que nos dará error o valores no deseados al ejecutar nuestro programa.

Ejemplo con Acumulador: Desarrollar un programa que sume los primeros 5 números.

En pseudocódigo:

INICIO

contador=1

suma=0

mientras contador <= 5 :

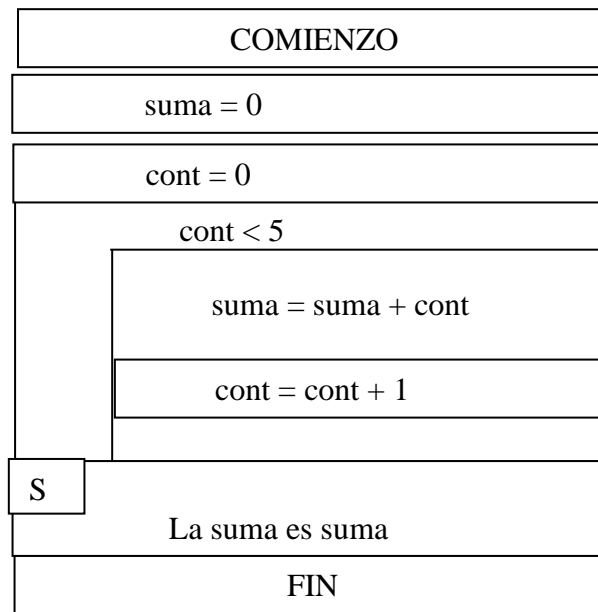
 suma=suma + contador

 contador = contador + 1

mostrar("la suma de los primeros 5 números es", suma)

FIN

En diagrama, una forma sería:



En Python:

```
#Programa SumaNumeros
suma = 0
cont = 0
while cont < 5 :
    suma= suma + cont
    cont = cont + 1

print("La Suma es:", suma)
```

Ciclos Exactos vs Ciclos Condicionales

Podemos clasificar a los ciclos en

Ciclos Exactos:

Cuando conocemos al crear el algoritmo la cantidad de veces que se va a repetir el ciclo, en general, sin los ciclos controlados por un contador.

Ciclos Condicionales:

Es el ciclo que se repite mientras se cumpla una condición y no conocemos cuántas veces va a repetir el ciclo, depende de cálculos o ingreso de datos desde el teclado.

Componentes del ciclo

Los componentes de un ciclo son:

- Expresiones de inicialización
- Condiciones de terminación
- Acciones a realizar dentro del ciclo

➤ Expresiones de finalización

Al diseñar un algoritmo donde se decide utilizar ciclo, es importante identificar las instrucciones correspondientes en cada parte del mismo, prestar atención a las expresiones lógicas y asegurar que alguna de las acciones dentro del ciclo logre volver falsa la expresión lógica para que no sea un **ciclo infinito**.

Ciclo For

Un ciclo **for** es una estructura de control que repite un bloque de instrucciones un número determinado de iteraciones. Es otra forma de escribir los ciclos exactos.

Sintaxis:

```
for <variable> in <elemento a recorrer>:  
    <instrucción 1>  
    <instrucción 2>  
    <instrucción n>
```

En este curso:
Secuencia de
números

Es una alternativa al ciclo while cuando están controlados por un contador.

Secuencias de Números: Tipo range

Para crear la secuencia de números a recorrer, utilizamos la función range. Podemos utilizarlo con uno, dos o tres parámetros:

<code>range(n)</code>	m: valor inicial
<code>range(m, n)</code>	n: valor final (que no se alcanza nunca)
<code>range(m, n, p)</code>	p: el paso (el aumento entre valores)

range(n)

Crea una secuencia de **n** números consecutivos que empieza en **0** y termina en **n-1**

En este curso: n > 0

Ejemplo:

`range(4)`
0, 1, 2, 3

`range(10)`
0, 1, 2, 3, 4, 5, 6, 7, 8, 9

range(m, n)

Crea una secuencia de números consecutivos que empieza en **m** y termina en **n-1**

En este curso: m < n

`range(2,6)`
2, 3, 4, 5

`range(5, 10)`
5, 6, 7, 8, 9

`range(0, 4)`
0, 1, 2, 3

`range(m, n, p)`

Crea una secuencia de números consecutivos que empieza en **m** y termina en **n-1**, aumentando los valores de **p** en **p**

En este curso **p > 0** y además **p NO PUEDE SER CERO**, produce error

```
range(1, 10, 2)  
1, 3, 5, 7, 9
```

```
range(5, 21, 3)  
5, 8, 11, 13, 15, 17, 20
```

Ciclo For con una secuencia numérica

```
0, 1, 2, 3
```

```
for i in range(4):  
    print(i)
```

Números al Azar

Son números que los crea la computadora. En todos los lenguajes vienen funciones listas para usar, las funciones están agrupadas en bibliotecas y en el caso de Python se agrupan en módulos.

Python tiene funciones para crear números al azar en el módulo random. Los módulos se agregan a nuestro programa utilizando la instrucción import. Como primera línea de código, se agregan todos los módulos necesarios:

import random


Para generar números al azar entre dos valores, ambos incluidos, utilizaremos la función randint, que debe ir precedida por el nombre del módulo, separado por un punto:

random.randint(mínimo, máximo)

Los números al azar, los utilizaremos para cargar las listas, a fin de evitar el ingreso de tantos valores a través del teclado.

Ejemplo: Crear números entre 1 y 10 inclusive:

```
import random  
  
numero = random.randint(1,10)  
print("El numero creado es", numero)
```



PRIMER LINEA DE CODIGO

Guía para desarrollar un algoritmo

Una forma posible para pensar un algoritmo sería:

1. Determinar cuál es el objetivo del algoritmo (cuáles son las entradas y que información tiene que salir). Si esto no está claro, es imposible desarrollar el algoritmo correcto, o sea, que resuelva el problema planteado.
2. Determinar cuáles son las variables que ya identificas
 - a. Variables necesarias para la entrada de datos.
 - b. Variables necesarias para la salida de datos.
 - c. Posiblemente aparecen más variables cuando se resuelve el algoritmo, no se pueden determinar ahora.
3. **Pensar** cuales serían las instrucciones necesarias para
 - a. Realizar la lectura y carga de la/s variable/s de entrada.
 - b. Luego, analizar que instrucciones son necesarias para cumplir con el objetivo. Pueden aparecer nuevas variables o modificar las del paso 2.
 - c. Realizar la salida de datos.

Para este paso es importante tener en claro cómo funcionan las estructuras de control e ir adquiriendo técnicas de resolución de problemas que veremos durante la cursada. Este es el paso que más tiempo va a llevar especialmente en los primeros algoritmos que uno comienza a resolver.

4. Cuando el algoritmo parece estar finalizado, tomar un conjunto de datos de entrada y seguir el algoritmo paso a paso para asegurar si cumple el objetivo o tiene un error de diseño, esto se conoce como “prueba de escritorio”. En caso de error, revisar el punto 3, modificar el algoritmo y volver a realizar el punto 4 hasta que funcione.
5. Una vez finalizado el algoritmo, se traduce a un lenguaje de programación, en nuestro caso Python y se realiza el proceso de depuración y ejecución del programa.

Programación Modular

Uno de los métodos para resolver problemas complejos es dividirlo en problemas más pequeños, llamados subproblemas. Estos problemas pueden a su vez ser divididos en problemas más pequeños hasta que son solucionados.

Esta técnica de dividir el problema principal en subproblemas se denomina “divide y vencerás”.

El problema principal se resuelve con el programa principal y cada subproblema es deseable que sea independiente de los restantes y se resuelven mediante módulos llamados procedimientos y funciones.

Ventajas de la programación modular:

- ✓ Se puede probar a los sub-algoritmos en forma independiente.
- ✓ Permite que diferentes diseñadores trabajen simultáneamente.
- ✓ Pueden utilizarse más de una vez.

Funciones

Son uno de los elementos básicos de la programación. Los lenguajes de programación incorporan funciones predefinidas, por ejemplo, print o input y también tienen la posibilidad de crear funciones definidas por el programador.

Una función es un subprograma que devuelve un único resultado al programa que lo invoca. La sintaxis en Python para crear funciones es la siguiente:

```
def NombreFuncion( lista de parámetros formales) :  
    Cuerpo de la función.  
  
    return valor
```

La palabra reservada **def** inicia la declaración de la función, seguida del nombre que se le quiere dar a la misma, luego entre paréntesis se indican los parámetros por donde la función recibe los datos de entrada para procesar. En el cuerpo de la función mediante la palabra reservada **return** se informa el dato de salida. En el caso de no especificar el valor de retorno de la función, Python retorna el valor None, que es la forma de expresar el vacío (null) en Python.

Lista de parámetros formales: sirven para pasar información a la función y/o devolver información a la unidad del programa que lo invoca o lo llama. Son las variables que se indican entre paréntesis. Por el momento, utilizaremos los parámetros únicamente como entrada de datos.

Variables locales o auxiliares: Dentro de un subprograma se puede crear variables que estarán disponibles durante el funcionamiento del mismo. Su valor se pierde una vez que se finaliza la ejecución del subprograma.

Ejemplo: Desarrollar una función para calcular el factorial de un número n que se recibe por parámetro.

```
def factorial (n):  
    resultado = 1  
    while i < n :  
        resultado = resultado * i  
        i = i + 1  
    return resultado
```

Llamada a una función

La llamada a una función se realiza en una sentencia de asignación o en un parámetro a otra función. Sólo en el caso que la función retorne siempre None podemos llamarla en una instrucción sin asignación, caso contrario perdemos el dato de salida de la función para continuar nuestro programa.

listas de parámetros actuales: son variables o constantes que tienen que tener los valores que se pasan a la función.

Ejemplo: Desarrollar un programa que calcule el factorial de un número utilizando la función factorial. El código fuente completo quedaría:

```
# Programa CalculaFactorial  
  
# Funciones  
def factorial (n):  
    resultado = 1  
    i=1  
    while i <= n :  
        resultado = resultado * i  
        i = i + 1  
    return resultado  
  
# Programa Principal  
nro = int (input ('Ingrese un numero'))  
result = factorial(nro)  
print('El factorial es', result)
```

también puede ser:

```
nro = int (input ('Ingrese un numero'))  
print('El factorial es', factorial(nro))
```

Se dice que el programa principal “llama” a la función factorial y le pasa por parámetro el dato para calcular el factorial.

Correspondencia de parámetros

IMPORTANTE: Los parámetros actuales en la invocación de la función debe coincidir en número, orden con los parámetros formales de la función.

Listas

Una lista es una secuencia de valores que pueden o no ser del mismo tipo de dato, en este curso utilizaremos únicamente secuencias de valores del mismo tipo de dato.

Operaciones en Listas

Crear una lista vacía

Se utilizan los corchetes vacíos para indicar que se crea una lista sin elementos

```
miLista = []
```

Crear una lista con n valores numéricos

Se utilizan los corchetes para delimitar una lista de valores separados por coma

```
miLista = [valo1, valor2, ..., valorN]
```

En este curso utilizaremos las listas con valores del mismo tipo de dato, ejemplo:

```
miLista = [32, 23, 54, 21]
```

Agregar valores a una lista

Agrega un elemento al final de la lista con el valor pasado por parámetro.

```
miLista.append( valor )
```

Subíndice para acceder a una posición de la lista

Cada valor de la lista se referencia utilizando un subíndice, deben ser números enteros, en Python siempre comienzan en cero y terminan uno antes del tamaño de la lista. Se escriben luego del nombre de la lista, encerrados entre corchetes. Ejemplo:

Si tenemos la lista: `miLista = [1, 7, 5, 6, 9]`

`miLista[0]` En la posición 0 se encuentra el valor 1
`miLista[1]` En la posición 1 se encuentra el valor 7
`miLista[2]` En la posición 2 se encuentra el valor 5
`miLista[3]` En la posición 3 se encuentra el valor 6
`miLista[4]` En la posición 4 se encuentra el valor 9

1	7	5	6	9
0	1	2	3	4

Pueden usarse constantes, variables y expresiones aritméticas, ejemplo:

4	7	9	2	1	8	6	0	5	3
0	1	2	3	4	5	6	7	8	9

vec

```
print(vec[2])    # Constante  
a = 4  
print(vec[a])    # Variable  
print(vec[a+3])  # Expresión
```

Cantidad de elementos de una lista

La función `len` nos retorna la cantidad de elementos que tiene una lista.

```
len(miLista)
```

Eliminar el último elemento de la lista

El método `pop` sin parámetros, quita el elemento que se encuentra en la última posición de la lista.

```
miLista.pop()
```

1	7	5	6	9
0	1	2	3	4

1	7	5	6
0	1	2	3

Eliminar un elemento que se encuentra ubicado en una posición *i*

El método `pop` con un parámetro *i*, quita el elemento que se encuentra en la posición *i* de la lista.

miLista.pop(2)

1	7	5	6	9
0	1	2	3	4

1	7	6	9
0	1	2	3

Tener en cuenta cuando se eliminan elementos:

- ✓ La cantidad de elementos disminuye, por lo tanto, cambia la longitud de la lista.
- ✓ Los elementos que se encuentran a la derecha del elemento eliminado, cambian de posición ya que se produce un corrimiento de esos elementos y se elimina la última posición de la lista.
- ✓ Al recorrer y eliminar los elementos de una lista, es recomendable utilizar una estrategia que recorra del último al primer elemento.

Modificar valores de la lista

Podemos modificar los valores que contiene una lista, referenciando con el subíndice a la posición y utilizando el operador de asignación, ejemplo:

miLista

4	7	9	2	1	8	6	0	5	3
0	1	2	3	4	5	6	7	8	9

miLista[4] = 5

El valor 1 que se encontraba en la posición 4 de la lista, se cambia por un 5:

4	7	9	2	5	8	6	0	5	3
0	1	2	3	4	5	6	7	8	9

a= 3

miLista[a] = miLista[a] + 1

El valor 2 que se encontraba en la posición 1 de la lista, se incrementa en uno:

4	7	9	3	5	8	6	0	5	3
0	1	2	3	4	5	6	7	8	9

b= 7

miLista[b] = miLista[b + 1]

El valor 0 que se encontraba en la posición 7, se cambia por el valor 5 que se encuentra en la posición 8

4	7	9	3	5	8	6	5	5	3
0	1	2	3	4	5	6	7	8	9

Intercambiar valores de posición

Para intercambiar dos valores que se encuentran en pos1 y en pos2, es necesario utilizar una variable auxiliar, que permita temporalmente guardar un valor para luego copiarlo dentro de la lista:

Paso 1: copiar en <u>aux</u> el valor de <u>vec[Pos1]</u>	<u>aux</u> = lista[Pos1]
Paso 2: copiar en <u>vec[Pos1]</u> el valor de <u>vec[Pos2]</u>	lista[Pos1] = lista[Pos2]
Paso 3: copiar en <u>vec[Pos2]</u> el valor de <u>aux</u>	lista[Pos2] = <u>aux</u>

EL VALOR DE Pos1 y Pos2 DEPENDE DE LOS ELEMENTOS QUE SE QUIEREN INTERCAMBIAR

```
aux = miLista[0]
miLista[0] = miLista[5]
miLista[5] = aux
```

4	7	9	3	5	8	6	5	5	3
0	1	2	3	4	5	6	7	8	9

8	7	9	3	5	4	6	5	5	3
0	1	2	3	4	5	6	7	8	9

Mostrar los valores de una lista

En general, en los lenguajes de programación si se desea mostrar por pantalla los valores de una lista, necesitamos un ciclo para recorrer cada posición y mostrar el valor, de la siguiente forma:

```
#Mostrar la lista, recorriendo la lista
def MostrarLista(lista):
    i = 0
    while i < len(lista):
        print(lista[i], end=" ")
        i = i + 1
    print()
```

Python nos permite mostrar la lista utilizando la función print, sin necesidad de realizar un ciclo:

```
print(miLista)
```

Funciones y Listas

A continuación, veremos algunos ejemplos de funciones utilizando listas:

Ejemplo 1: Función Crear una lista con elementos ingresados desde teclado hasta ingresar -1

```
#Crear una lista hasta ingresar un -1
def GenerarLista():
    lista=[]
    nro = int(input("Ingrese un numero, -1 para finalizar"))
    while nro != -1:
        lista.append(nro)
        nro = int(input("Ingrese un numero, -1 para finalizar"))
    return lista
```

Ejemplo 2: Crear una lista con N elementos al azar de dos dígitos. En este caso la función recibe cuántos elementos se desean crear

Se puede implementar con un ciclo while:

```
import random

#Crear una lista de n numeros al azar
#de dos digitos
def GenerarListaAzar(n):
    lista=[]
    cont = 0
    while cont < n:
        nro=random.randint(10,99)
        lista.append(nro)
        cont = cont + 1

    return lista
```

Se puede implementar con un ciclo for:

```
import random

#Crear una lista de n numeros al azar
#de dos digitos
def GenerarListaAzar(n):
    lista=[]
    for cont in range(0, n):
        nro=random.randint(10,99)
        lista.append(nro)

    return lista
```

Estas funciones se encargan de crear la lista, por lo que es necesario que retorne la misma para poder ser utilizada en el programa principal, para utilizar esta función sería:

#Programa Principal

N = int(input("ingrese la cantidad de elementos que desea crear"))

miLista = generarListaAzar(N)

Ejemplo 3: Mostrar una lista por pantalla separando cada elemento con un espacio.

```
#Mostrar la lista, separando los elementos con un espacio
def MostrarLista(lista):
    i = 0
    while i < len(lista):
        print(lista[i], end=" ")
        i = i + 1
    print()
```

`end=" "` dentro del print indica que la siguiente impresión debe aparecer en el mismo renglón, separada por un espacio.

Esta función no retorna ningún valor, su objetivo es mostrar algo por pantalla, al momento de llamar a la función NO se debe asignar a una variable, ya que NO retorna nada.

#Programa Principal

N = int(input("ingrese la cantidad de elementos que desea crear"))

miLista = generarListaAzar(N) *#función que retorna una lista, la recibe la variable miLista*

MostrarLista(miLista) *#función que no retorna valor.*

Ejemplo 4: Modificar los elementos de una lista. Duplicar todos sus valores.

```
#Duplicar cada valor de la lista
def Duplicar(lista):
    i = 0
    while i < len(lista):
        lista[i] = lista[i] * 2
        i = i + 1
```

Ejemplo 5: Eliminar un valor de la lista.

```
def EliminarValor(lista, valor):
    i = len(lista) - 1
    while i >= 0 :
        if valor == lista[i]:
            lista.pop(i)
            i = i - 1
```

Las funciones que solamente modifican los elementos de la lista, no es necesario que retornen la misma, ya que la modificación se realiza sobre la lista original recibida por parámetro. Es importante prestar atención si la función No retorna la lista, No se debe asignar:

```
#Programa Principal
n = int(input("ingrese la cantidad de elementos que desea crear"))
miLista = generarListaAzar(n) #función que retorna una lista, la recibe la variable miLista
MostrarLista(miLista) #funciones que no retorna valor.
Duplicar(miLista)
MostrarLista(miLista)
```

Ejemplo 6: Función para retornar la suma de los elementos de la lista.

```
def SumarLista(lista):
    suma = 0

    for i in range(len(lista)):
        suma = suma + lista[i]

    return suma
```

```
#Programa Principal
N = int(input("ingrese la cantidad de elementos que desea crear"))
miLista = generarListaAzar(N)
MostrarLista(miLista)
print("La suma de los valores de la Lista es", SumarLista(miLista))
```


Búsqueda Secuencial

Para determinar si un elemento se encuentra o no en una lista y hasta se puede saber cuántas veces aparece un valor en una lista, podemos utilizar la búsqueda secuencial, consiste en ir recorriendo la lista elemento por elemento hasta encontrar el valor buscado o hasta llegar al final de la lista, lo que significa que el valor no se encuentra presente.

Si queremos saber cuantas veces aparece un valor en una lista, se debe recorrer completa, si solamente queremos saber si existe el valor en la lista, podemos dejar de buscar una vez encontrado el elemento buscado.

Ejemplo de una función que realiza la búsqueda secuencial de un elemento y retorna verdadero (True) o falso (False) si no lo encuentra:

Ejemplo 7: Función para realizar una búsqueda secuencial, retorna True o False consiste en ir recorriendo la lista elemento por elemento hasta encontrar el valor buscado o hasta llegar al final de la lista, lo que significa que el valor no se encuentra presente.

```
#Buscar un valor, retornar True o False
def BuscarValorSecuencial(lista, nro):
    i = 0
    while i < len(lista) and lista[i] != nro:
        i = i + 1

    if i == len(lista):
        encontrado = False
    else:
        encontrado = True
    return encontrado
```

Las funciones que retornan verdadero o falso, sirven para tomar una decisión al momento de llamar a la función:

```
#Programa Principal

N = int(input("ingrese la cantidad de elementos que desea crear"))

milista = generarListaAzar(N) #función que retorna una lista, la recibe la variable milista
MostrarLista(milista)        #función que no retorna valor.

valor = int(input("ingrese el valor a buscar:"))
if BuscarValorSecuencial(milista, valor) == True: #función que no retorna True/False.
    print("El valor se encuentra en la lista")
else:
    print("El valor No se encuentra en la lista")
```

Métodos de Ordenamiento

El objetivo de los métodos de ordenamiento, es lograr tener los valores respetando un orden ascendente o descendente dentro de la lista. Existen muchos algoritmos para lograr ordenar los valores. En este curso analizaremos tres algoritmos:

Ordenamiento por Selección

ORDENAR DE MENOR A MAYOR: Consiste en buscar el menor elemento de la lista y lo intercambia con la primer posición, luego busca el Segundo menor elemento y lo intercambia con el de la segunda posición, y así sucesivamente.

```
def metodoSeleccion(lista):  
    for i in range(0, len(lista)-1):  
        for j in range ( i + 1, len(lista)):  
            if lista[i] > lista[j]:  
                aux = lista[i]  
                lista[i] = lista[j]  
                lista[j] = aux
```

Ordenamiento por Burbujeo

ORDENAR DE MENOR A MAYOR: Consiste en recorrer un vector, tomando el primer elemento y compararlo contra el segundo. Si el primer elemento es mayor, intercambiarlo de lugar. Tomar el segundo y compararlo contra el tercero, y intercambiarlo con el mismo criterio. De esta manera, al finalizar el proceso, el mayor quedará colocado en el lugar que le corresponde. Este proceso se repite $N - 1$ veces, siendo N la cantidad de elementos.

```
def metodoBurbuja(lista):  
    for recorrido in range(1, len(lista)):  
        for i in range( len(lista)-recorrido):  
            if lista [ i ] > lista [ i+1 ] :  
                aux = lista [ i ]  
                lista [ i ] = lista [ i+1 ]  
                lista [ i+1 ] = aux
```

Ordenamiento por Burbujeo Mejorado

El método de Burbujeo aunque el vector esté ordenado, completo y compara todos los elementos. Este algoritmo se puede mejorar, detectando cuando ya se encuentra ordenado para finalizar el proceso en ese momento. Para eso se utiliza una variable que marca si se realizan intercambios, en el ejemplo a continuación, esa marca es una variable entera que se llama Ordenado

```
def metodoBurbuja(lista):  
    ordenado=1  
    while (ordenado == 1):  
        ordenado =0  
        for i in range(len(lista)-1):  
            if lista[ i ] > lista[ i+1 ]:  
                aux = lista[ i ]  
                lista[ i ] = lista[ i+1 ]  
                lista[ i+1 ] = aux  
                ordenado = 1
```

Ordenamiento por Inserción

ORDENAR DE MENOR A MAYOR: Consiste en tomar el primer elemento como si fuera una lista ordenada e ir insertando cada elemento en su posición correcta. Para eso va desplazando los elementos hasta encontrar su ubicación.

Este método no realiza intercambios, sino desplazamientos.

Este proceso se repite $N - 1$ veces, siendo N la cantidad de elementos.

```
def metodoInsercion(lista):  
    for i in range(1, len(lista)):  
        valorInsertar = lista[i]  
        j = i  
        while j > 0 and lista[j-1] > valorInsertar :  
            lista[j] = lista[j-1]  
            j = j-1  
        lista[j] = valorInsertar
```

Búsqueda Binaria

La búsqueda binaria, nos permite determinar si un elemento se encuentra en la lista, aprovecha el hecho de contar con una lista ordenada. Esto le permite completar el proceso de búsqueda en mucho menos tiempo que la búsqueda secuencial.

Procedimiento: Se verifica si el elemento a buscar se encuentra en la mitad de la lista. Si no está, resulta fácil deducir para qué lado podría llegar a encontrarse debido a que la lista se encuentra ordenada. Se descarta una mitad y se repite el proceso sobre la otra mitad.

```
def BusquedaBinaria(lista, dato):  
    izq = 0  
    der = len(lista) - 1  
    pos = -1  
    while izq <= der and pos == -1:  
        centro = (izq + der) // 2  
        if lista[centro] == dato:  
            pos = centro  
        elif lista[centro] < dato:  
            izq = centro + 1  
        else:  
            der = centro - 1  
    return pos
```