Por ejemplo, si tu ya antes has escrito recetas de cocina, y en la escuela se te pide hacer un instructivo para armar un papalote, podrías escribir este, relacionándolo con la forma como se escribe una receta, es decir, por analogía!

1.2.3 Buscar cosas que son familias

Esta estrategia es muy parecida a la de resolución por analogía. Sin embargo, la idea es "nunca reinventar la rueda", Si una solución ya existe, se puede usar esta. Es decir, si ya has resuelto el mismo o un problema muy similar antes, sólo repite la solución.
Uno no aprende a ir a comprar el pan, después a ir a comprar leche, después huevo! , simplemente aprende uno a "ir a comprar a la tienda" , es decir, repite uno la misma solución. En programación ciertos problemas se repiten uno y otra vez, por ejemplo obtener la máxima y mínima calificación de un grupo de alumnos, es igual a obtener la temperatura máxima y mínima de un conjunto de datos.

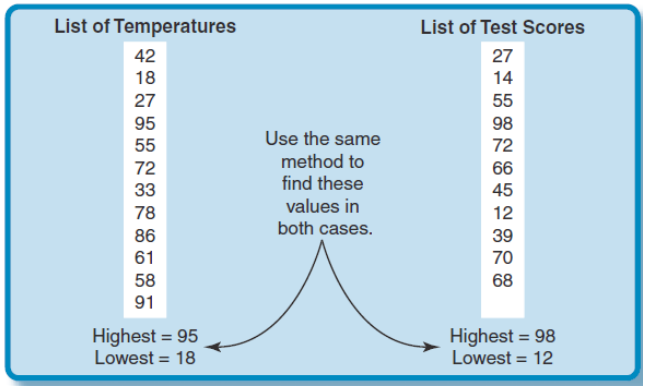


Figura 1.11- obtención de máximo y mínimo

1.2.4 Análisis de medios y fines

La técnica de análisis de medios y fines permite trabajar con un objetivo o fin a la vez. Es decir, esta técnica consiste en descomponer el problema en sub-metas o sub-objetivos, ir escogiendo una sub-meta para trabajar y solucionar ésta, después seleccionar otra sub-meta y así hasta completar la tarea, eliminando los obstáculos que le impiden llegar al estado final.
Un buen ejemplo donde aplicar está técnica, es para el problema de ir de una ciudad a otra, con varias ciudades intermedias. En este caso las sub-metas son las ciudades intermedias y los medios son la forma de llegar de una ciudad a otra.
Por ejemplo si quisiéramos recorrer la ruta Maya, podríamos identificar las ciudades involucradas como sub-metas y decidir como llegar ahí, por ejemplo para llegar de Puebla a San Cristobal, no podría ser por avión, tendríamos que buscar otra sub-meta, Tuxtla Guitierez, y de Tuxtla a San Cristobal por auto, y así a cada localidad.

Figura 1.12 - Aplicar estrategia de medios y fines

1.2.5 Combinar Soluciones

La última técnica para resolver problemas es combinar soluciones *existentes.* Esta técnica podría parecerse a la técnica de *Buscar cosas que sean familiares*, pero en este caso, es **combinar** las soluciones paso a paso, no solamente **unir** soluciones obtenidas separadamente.
Por ejemplo: Para calcular el promedio de una lista de valores debemos hacer dos cosas: sumar los valores y contarlos. Si tenemos dos soluciones separadas, una para sumar los valores y otra para contarlos, podemos combinarlas y así paso a paso ir sumando y contando al mismo tiempo de tal manera que al final tendremos la solución al problema.

Una vez discutidas las diversas formas para abordar un problema, se puede en la fase de resolución la manera en que se va a diseñar el algoritmo particular de solución al mismo, para lo que se introducirá el concepto de algoritmo.

Diseño del algoritmo

Un **algoritmo** puede ser definido como la **secuencia ordenada** de pasos, sin ambigüedades, que conducen a la resolución de un problema dado y expresado en lenguaje natural. Todo algoritmo debe ser:

- **Preciso:** Indicando el orden de realización de cada uno de los pasos.
- **Definido:** Si se sigue el algoritmo varias veces proporcionándole **(consistente)** los mismos datos, se deben obtener siempre los mismos resultados.
- **Finito:** Al seguir el algoritmo, este debe terminar en algún momento, es decir tener un número finito de pasos.

Para diseñar un algoritmo se debe comenzar por identificar las tareas más importantes para resolver el problema y disponerlas en el orden en el que han de ser ejecutadas. Los pasos en esta primera descripción pueden requerir una revisión adicional antes de que podamos obtener un algoritmo claro, preciso y completo.
Este método de diseño de algoritmos en etapas, yendo de los conceptos generales a los de detalle, se conoce como método descendente (top-down).

En un algoritmo se deben de considerar tres partes:

- **Entrada:** Información dada al algoritmo.
- **Proceso:** Operaciones o cálculos necesarios para encontrar la solución del problema.
- **Salida:** Respuestas dadas por el algoritmo o resultados finales de los procesos realizados.

Como ejemplo supongamos que desea desarrollar un algoritmo que calcule la superficie de un rectángulo proporcionándole su base y altura.
Lo primero que debemos hacer es plantearnos las siguientes preguntas:

Especificaciones de entrada
¿Que datos son de entrada?
¿Cuántos datos se introducirán?
¿Cuántos son datos de entrada válidos?

Especificaciones de salida
¿Cuáles son los datos de salida?
¿Cuántos datos de salida se producirán?
¿Qué formato y precisión tendrán los resultados?

El algoritmo que podemos utilizar es el siguiente:
Paso 1. Entrada desde el teclado, de los datos de base y altura.
Paso 2. Cálculo de la superficie, multiplicando la base por la altura.
Paso 3. Salida por pantalla de base, altura y superficie calculada.

El lenguaje algorítmico debe ser independiente de cualquier lenguaje de programación particular, pero fácilmente traducible a cada uno de ellos. Alcanzar estos objetivos conducirá al empleo de **métodos** normalizados para la representación de algoritmos, tales como los diagrama de flujo o **pseudocódigo**.

Verificación de algoritmos

Una vez que se ha terminado de escribir un algoritmo es necesario comprobar que realiza las tareas para las que se ha diseñado y produce el resultado correcto y esperado. El modo más normal de comprobar un algoritmo es mediante su ejecución manual, usando datos significativos que abarquen todo el posible rango de valores. Este proceso se conoce como **prueba del algoritmo**. A continuación se formularán un conjunto de algoritmos:

Ejemplo 1.1
Un procedimiento que realizamos varias veces al día consiste en lavarnos los dientes. Veamos la forma de expresar este procedimiento como un Algoritmo:

1. Tomar la crema dental
2. Destapar la crema dental
3. Tomar el cepillo de dientes
4. Aplicar crema dental al cepillo
5. Tapar la crema dental
6. Abrir la llave del lavamanos
7. Remojar el cepillo con la crema dental
8. Cerrar la llave del lavamanos
9. Frotar los dientes con el cepillo
10. Abrir la llave del lavamanos

11. Enjuagarse la boca
12. Enjuagar el cepillo
13. Cerrar la llave del lavamanos
14. Secarse la cara y las manos con una toalla

Ejemplo 1.2

Veamos que algo tan común como los pasos para cambiar una bombilla (foco) se pueden expresar en forma de Algoritmo:

1. Ubicar una escalera o un banco debajo de la bombilla fundida
2. Tomar una bombilla nueva
3. Subir por la escalera o al banco
4. Girar la bombilla fundida hacia la izquierda hasta soltarla
5. Enroscar la bombilla nueva hacia la derecha en el plafón hasta apretarla
6. Bajar de la escalera o del banco
7. Fin

Actividad 1.1
Discutir en parejas el ejemplo de la bombilla y proponer algunas mejoras. Luego, un voluntario debe pasar al pizarrón y escribir un Algoritmo con participación de toda la clase.
.
Ejemplo 1.3

Consideremos algo más complejo como el algoritmo de Euclides para hallar el Máximo Común Divisor (MCD) de dos números enteros positivos dados. Obsérvese que no se especifica cuáles son los dos números, pero si se establece claramente una restricción: deben ser enteros y positivos.

ALGORITMO

Paso 1: Inicio.
Paso 2: Leer los dos números (“a” y “b”). Avanzar al paso 3.
Paso 3: Comparar “a” y “b” para determinar cuál es mayor. Avanzar al paso 4.
Paso 4: Si “a” y “b” son iguales, entonces ambos son el resultado esperado y termina el algoritmo. En caso contrario, avanzar al paso 5.
Paso 5: Si “a” es menor que “b”, se deben intercambiar sus valores. Avanzar al paso 6; si “a” no es menor que “b”, avanzar al paso 6.
Paso 6: Realizar la operación “a” menos “b”, asignar el valor de “b” a “a” y asignar el valor de la resta a “b”. Ir al paso 3.

Actividad 1.2
Reflexiona sobre el lenguaje que utilizas diariamente para comunicarte con tus padres, hermanos, profesores y compañeros. ¿Utilizas un lenguaje preciso? ¿Utilizas vocablos corrientes?

Actividad 1.3
A diferencia de los seres humanos que realizan actividades sin detenerse a pensar en los pasos que deben seguir, los computadores son muy ordenados y necesitan que quien los programa les especifique cada uno de los pasos que debe realizar y el orden lógico de ejecución. Numerar en orden lógico los pasos siguientes (para pescar):

___ El pez se traga el anzuelo.
___ Enrollar el sedal.

___ Tirar el sedal al agua.
___ Llevar el pescado a casa.
___ Quitar el Anzuelo de la boca del pescado.
___ Poner carnada al anzuelo.
___ Sacar el pescado del agua.

Actividad 1.4

Supongamos que desea ir de Chihuahua a Cancún. Para logar esto se tienen varias opciones: usar el autobús, caminar, viajar en motocicleta, por tren, por avión o caminar. Dependiendo de las circunstancias, usted elegirá la opción que más le convenga. Si es que esta apurado, elegirá viajar por avión, si su presupuesto no es muy alto, probablemente elija viajar por autobús. Diseñe un algoritmo siguiendo la técnica de medios y extremos para llegar a la solución del problema.

Unidad 2 Elementos básicos para el desarrollo de Algoritmos

2.1 Arquitectura Funcional de la Computadora.

Como primer paso en el estudio del funcionamiento de la computadora, se inicia dando algunas definiciones elementales y se recomienda la lectura cuidadosa de los conceptos, con el objetivo de entenderlos adecuadamente.

Sistema Conjunto de Unidades que colaboran entre si para realizar una función general y donde cada unidad efectúa un trabajo específico. Por ejemplo: Sistema Circulatorio, Sistema Digestivo, Sistema de Transporte Colectivo, etc.

Computadora Sistema de Procesamiento de Información que recibe datos como entrada, los modifica (procesa) y produce resultados como salida. Las unidades que conforman este sistema son: Unidad de Memoria, Unidad Aritmético Lógica, Unidad de Entrada / Salida y Unidad de Control.

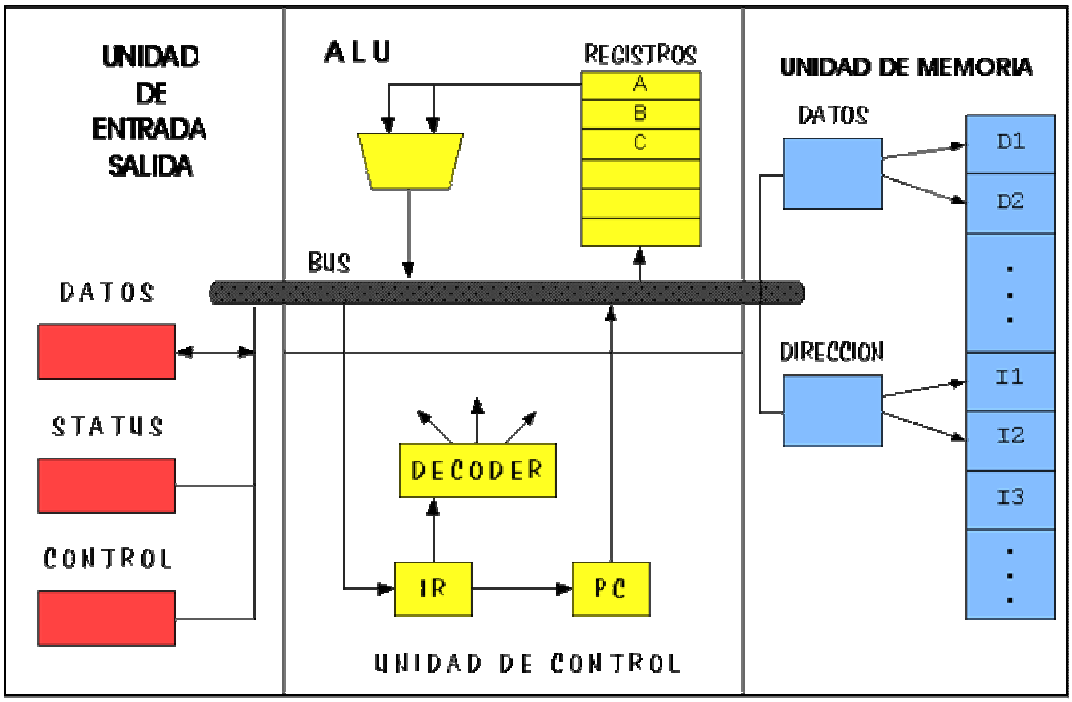


Figura 2.1 Diagrama General de la Computadora

U. de Memoria Conjunto de Celdas de Almacenamiento que guardan Información (pueden ser datos y/o programas). Normalmente la cantidad de celdas se mide en Millones y el tamaño de cada celda se mide en múltiplos de ocho dígitos binarios. Esta unidad se subdivide en Memoria Primaria y Memoria Secundaria en función de sus características (tamaño, velocidad, volatilidad).

U. Aritmético Lógica Conjunto de Circuitos que realizan las operaciones Aritméticas (+, -, *, /) y Lógicas (and, or, neg, xor) de datos almacenados en registros de propósito general. El tamaño de los registros, así como el número de operaciones aritméticas y lógicas diferentes está en función del tamaño de la computadora.

U. de Entrada / Salida Conjunto de Registros (Datos, Estado y Control) que permiten realizar las operaciones de comunicación (E/S) con el exterior. Estas dos funciones, la Entrada y la Salida, deben verse como actividades separadas. Los registros están conectados físicamente a los dispositivos correspondientes, por ejemplo, los registros de salida del Puerto Paralelo están conectados a la Impresora.

U. de Control Conjunto de registros que conforman la parte activa de la computadora. La función principal es la de realizar el ciclo de máquina: Alimenta una instrucción, Decodificas la Instrucción y Ejecuta la Instrucción.

Comentarios:

- No puede realizar cualquier operación (hacer pasteles).
- La computadora puede realizar las operaciones Aritmético Lógicas definidas en su Unidad solamente.
- Es posible construir nuevas operaciones tomando como base las ya existentes (coseno, tangente, suma de vectores, etc.).
- La computadora es una herramienta para el humano.

- La solución a los problemas computacionales las da el especialista y no la computadora.

2.2 Variables Computacionales

Las variables pueden verse desde diferentes puntos de vista como el matemático, computacional, filosófico, etc. En nuestro caso, debemos ver las variables desde un enfoque computacional (funcional).

Variable: Área de almacenamiento, de tamaño fijo y que sirve para guardar un tipo de dato específico. Las variables tiene dos elementos: El Nombre de la variable y el Contenido de la variable. El nombre de la variable sirve para distinguirla de otras variables y el contenido se refiere al dato que es guardado en ese lugar. La variable puede almacenar un sólo tipo de dato como puede ser datos enteros, reales, booleanos, caracteres, etc.

2.3 Operaciones Aritméticas y Lógicas

Las operaciones Aritméticas y Lógicas que una computadora puede hacer están en función del tamaño y del propósito de dicha computadora. Es decir, si una computadora es chica entonces el número de operaciones aritmético y Lógicas también es "chico". Por ejemplo en una computadora personal (pequeña), el número de operaciones será alrededor de 10 (aproximadamente), mientras que en una computadora "grande" puede llegar a 20 o 25. Debemos recordar que todas estas operaciones se realizan por medio de circuitos electrónicos y que entre mas circuitos tenga, su costo será mucho más alto.

Las operaciones Aritméticas básicas, en una computadora pequeña, son: Incremento, Decremento, Suma, Resta, Multiplicación y División. Las Operaciones Lógicas básicas son: AND, OR y Negación.
Como notación, normalmente se utilizan los símbolos siguientes para los operadores Aritméticos y Lógicos:

	Símbolo
Incremento	++
Decremento	--
Suma	+
Resta	-
Multiplicación	*
AND	&&
OR	||
Negación	!

Tabla 2.1 Operadores Aritméticos y Lógicos.

En computadoras "grandes" las operaciones pueden ser: Incremento, Decremento, Suma, resta, Multiplicación, División, Residuo, Raíz Cuadrada, Raíz Cúbica, Valor Absoluto, Potencia, Logaritmo, Seno, Coseno, Tangente, AND, OR, Negación, OR exclusivo, NAND, NOR, etc.

Además de lo operadores Aritméticos y los Lógicos existen también los operadores relacionales. Dichos operadores se basan en el concepto de ORDEN que

existe dentro de los conjuntos, es decir, los números enteros, los números reales, incluso los caracteres (en computación) son conjuntos ordenados.
Como notación, normalmente se utilizan los símbolos siguientes para los operadores relacionales:

	Símbolos
Igual que	==
Menor que	<
Mayor que	>
Menor o igual que	<=
Mayor o igual que	>=
Distinto que	!=

Tabla 2.2 Operadores Relacionales

Los operadores relacionales son binarios, es decir, que requieren dos operandos y el resultado de la operación puede ser "Falso" o "Verdad". Algunos autores utilizan Apagado/Prendido, 0/1, No/Si.

2.4 Expresiones

Una expresión es una combinación de operandos (variables), constantes y operadores. La expresión debe estar bien escrita, es decir, siguiendo las reglas de los operadores.
Las expresiones se pueden dividir en expresiones aritméticas o expresiones lógicas.

Ejemplos de expresiones aritméticas:
25
(50 * 2) + 10
Salario * 0.15

Las expresiones normalmente se usan junto con otros conceptos como las proposiciones (también llamadas sentencias) que son estructuras formadas por una variable, seguida de un símbolo de asignación (también llamado operador de asignación) y finalmente una expresión. Las expresiones incluyen los paréntesis como símbolos de apoyo.

La manera más fácil de ver esto, es con ejemplos.

Ejemplos de proposiciones con expresiones aritméticas:

Base = (500 * 2) + 100
Salario = DiasTrabajados * SalarioDiario
Impuesto = Salario * 0.15

Ejemplos de proposiciones con expresiones lógicas:

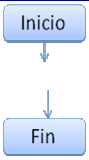
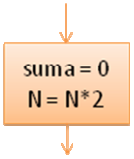
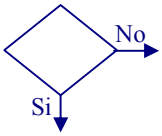
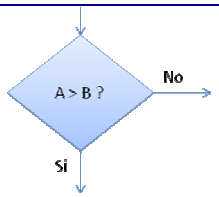
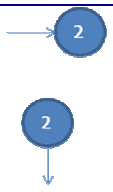
Mayor = a > b
Menor = (a < b) && (b < c)

Unidad 3.

Diseño estructurado usando herramientas de representación de algoritmos

3.1 Diagramas de flujo

Un **diagrama de flujo** es una representación gráfica de un algoritmo que se caracteriza por usar símbolos gráficos, para expresar de forma sencilla y clara el orden lógico en el que se realizan las acciones de un algoritmo. Los símbolos utilizados han sido normalizados por el Instituto Norteamericano de Normas (ANSI) [1], y los más frecuentemente utilizados se muestran en la Tabla 3.1

Símbolo	Significado o Función	Descripción	Ejemplo
	Inicio / Fin	Indica el inicio o el fin del diagrama de flujo.	
	Entrada / Salida	Permite la lectura de datos de entrada (comúnmente el teclado) así como la salida de datos (comúnmente la pantalla).	
	Proceso	Se utiliza para realizar asignaciones, operaciones, pasos, procesos y en general instrucciones de programas de cómputo.	
	Decisión	Permite evaluar una condición y dependiendo del resultado se sigue por una de las ramas o caminos alternativos.	
	Conector	Conector para unir el flujo a otra parte del diagrama.	

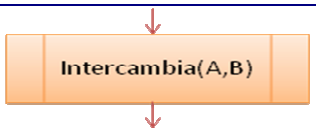
	Subprograma	Permite el llamado a otros diagramas de flujo	

Tabla 3.1. Símbolos más utilizados en los Diagramas de Flujos

Los símbolos mostrados en la tabla 3.1.1 son unidos por medio de flechas que indican el flujo de control del diagrama ver tabla 3.1.2.

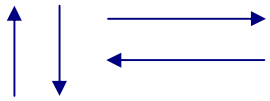
Símbolo	Significado o Función
	**Dirección del flujo del diagrama**

Tabla 3.2. Flechas de control de flujo del diagrama

3.1.1 Reglas para la construcción de diagramas de flujo

A continuación se indican algunas reglas que permiten la construcción de diagramas de flujo [2]:

1. Todo diagrama de flujo debe tener un *inicio* y un *fin*.
2. Las líneas utilizadas para indicar la dirección del flujo del diagrama deben ser rectas, verticales y horizontales
3. Todas las líneas utilizadas para indicar la dirección del flujo del diagrama deben estar conectadas a alguno de los símbolos mostrados en la tabla 3.1.1.
4. El diagrama de flujo debe ser construido de arriba hacia abajo y de izquierda a derecha.
5. Si el diagrama de flujo requiriera más de una hoja para su construcción, debemos utilizar los conectores.

3.2 Pseudocódigo

El pseudocódigo es una herramienta de programación en la que las instrucciones se escriben en palabras similares al inglés o español, que facilitan tanto la escritura como la lectura de programas. En esencia, el pseudocódigo se puede definir como un lenguaje de especificación de algoritmos [1].

Las palabras reservadas comúnmente utilizadas en pseudocódigo son: ***Inicio, Fin, Leer, Escribir, Si, Entonces, Sino, FinSi, Según, FinSegun, Mientras, Hacer, FinMientras, Repetir, HastaQue, Para, FinPara, Desde***, entre otras.

3.2.1. Características del pseudocódigo

A continuación se describen las características principales del pseudocódigo:

- Mantiene una identación o sangría adecuada para facilitar la identificación de los elementos que lo componen.
- Permite la declaración de los datos (constantes y/o variables) manipulados por el algoritmo.
- Dispone de un conjunto pequeño de palabras reservadas (normalmente escritas con letra negrita) para expresar las acciones del algoritmo.
- Supera las dos principales desventajas del diagrama de flujo: lento de crear y difícil de modificar.
- Permite el seguimiento de la lógica de un algoritmo.

3.3 Estructuras de secuencia

Las estructuras de secuencia también son conocidas como *sentencias* o proposiciones.

Una *sentencia o proposición* es una unidad completa, ejecutable en sí misma.

La ejecución de estas sentencias se realiza de manera *secuencial*, es decir, cada una a continuación de la anterior, empezando por la primera y terminando con la última.

Proposición Simple

Consiste en una sóla sentencia:

Por ejemplo:

a) Edad ← 18
b) SUMA ← DATO1 + DATO2 + DATO3
c) ESCRIBE(‘ Dame tu opción’)

En algunos lenguajes de programación las sentencias terminan con el carácter punto y coma (;).

Proposición Compuesta o bloque

Es un conjunto de declaraciones y sentencias (proposiciones) agrupadas dentro de los delimitadores INICIO y FIN.

Ejemplos:

a) Una proposición compuesta de 2 enunciados:

```
INICIO
   X ← 5
   Y ← 89
FIN
```

b) Una proposición compuesta de 4 enunciados:

```
INICIO
    PRECIO ← 89.50
    INTERES ← 0.15
    COSTO_FINAL ← PRECIO + INTERES
   ESCRIBE(‘El costo final del producto es: ’, COSTO_FINAL)
FIN
```

3.4 Estructuras de Control

Las estructuras de control, también conocidas como *estructuras de decisión*, controlan el flujo de ejecución de un programa. Permiten seleccionar la próxima proposición a ejecutarse dependiendo del valor de una condición (cierta o falsa). Para construir esta condición se usarán las expresiones lógicas o relacionales.

Entre las estructuras de control se tienen las siguientes: condicional simple, condicional doble y condicionales múltiples.

Expresiones lógicas y relacionales

a) Operadores relacionales

Los operadores relacionales se usan para comparar expresiones. Una expresión que contiene un operador relacional evalúa a cierto (1) o falso (0).

Sintaxis:

expresion1 *operador relacional* expresion2

Operadores relacionales

Operador	**símbolo (en pseudocódigo)**
Igual	=
Mayor que	>
Menor que	<
Mayor qué o igual a	>=
Menor qué o igual a	<=
Diferente	<>

Al comparar una expresión es como si se realizaran preguntas:

Ejemplos: Supongamos que se tienen las variables x=5, y=100, z= -2

Expresión relacional	**Pregunta**	**Evalúa a**
x > y	¿ x es mayor que y ?	Falso
x = y	¿ x es igual a y ?	Falso
y = 100	¿ y es igual a 100 ¿	Cierto
x<= y	¿ x es menor o igual a y ?	Cierto
x <> y	¿ x es distinto de y ?	Cierto
(5 + 80) < 200	¿ 5 + 80 es menor que 200 ?	Cierto
x*2 > (x + y + z)	¿ x*2 es mayor que x + y +z ?	Falso

b) Operadores lógicos

Algunas veces se necesitará hacer más de una pregunta relacional al mismo tiempo. Los operadores lógicos permitirán combinar dos o más *expresiones relacionales* en una sola expresión que evalúa a cierto (1), o falso (0).

Sintaxis:

Para operadores **o** e **y**

expresion1 *operador lógico* expresión2

Para operador **no**

no expresión

Operadores lógicos

Operador	**símbolo (en pseudocódigo)**
y	y o bien and
o	o o bien or
no	no o bien not

Estos operadores evaluarán a cierto o falso según las siguientes combinaciones:

Operador y

expresión1	expresión2	expresión1 y expresión2
cierto	cierto	cierto
cierto	falso	falso
falso	cierto	falso
falso	falso	falso

Tabla 3.3

Operador o

expresión1	expresión2	expresión1 o expresión2
cierto	cierto	cierto
cierto	falso	cierto
falso	cierto	cierto
falso	falso	falso

Tabla 3.4

Operador **no**

expresión	no expresión
Cierto	Falso
Falso	Cierto

Tabla 3.5

Ejemplos: Supongamos que se tienen las variables x=5, y=100, z= -2

expresión operador expresión **Pregunta**
evalúa a

(x > y) **y** (x = y) ¿x es mayor que y **y** x es igual a y?
Falso
F F

(x > y) **y** (y = 100) ¿x es mayor que y **y** y es igual a 100?
Falso
F C

(x <> y) y (y = 100) ¿x es distinto de y **y** y es igual a 100?
Cierto
C C

(x > y) **o** (y = 100) ¿x es mayor que y o y es igual a 100?
Cierto
F C

(x <= y) **o** (x = 50) ¿x es menor o igual a y **o** x es igual a 50?
Cierto
C F

(x = y) **y** ((x > y) **o** (y = 100)) ¿x es distinto de y? **y** ¿ x es mayor que y o y es igual a 100? Falso
F F o C
F **y** C

((x = y) **y** (x > y)) **o** (y = 100) ¿ x es distinto de y **y** x es mayor que y? **o** ¿y es igual a 100? Cierto
F **y** F
F o C

3.4.1 Condicional Simple

Se evalúa la condición, si la condición es cierta se realiza proposición, y si es falsa entonces no se realiza la proposición.

Sintaxis:

Si condición entonces
proposición

Ejemplos.

a) Supongamos que se tienen las variables x=5, y=100

Si (x = y) **entonces**
Escribe('Las variables x e y tienen el mismo valor ')

La condición evalúa a falsa, por lo que no se imprimirá ningún mensaje en pantalla.

b) Supongamos que se tienen las variables x=5, y=5

Si (x = y) **entonces**
Escribe('Las variables x e y tienen el mismo valor ')

La condición evalúa a Cierta, por lo que en pantalla se visualizará el mensaje:

Las variables x e y tienen el mismo valor

c) Supongamos las variables COSTO_FINAL = 0.0, COSTO = 1989.50, INTERES = 50.00

Si (COSTO < = 5000) **y** (INTERES = 50.00) **entonces**
COSTO_FINAL ← COSTO + INTERES

La condición evalúa a cierta por lo que el valor actual de COSTO_FINAL es: 2039.50

d) Supongamos k=0, x =10, y =7, w =0

si (x < 50) entonces
INICIO
w ← x + y
k ← w-5 + y
FIN

Después de realizar la decisión, el valor de la variable k será: 19

3.4.2 Condicional doble

Se evalúa la condición, si la condición es cierta se realizará proposición1 y si condición es falsa se realizará la proposición 2.

Sintaxis:

Si condición entonces
Proposición 1
Si no
Proposición 2

Ejemplos:

a) Supongamos que se tienen las variables x=5, y=100

Si (x = y) **entonces**
Escribe('Las variables x e y tienen el mismo valor ')
Si no
Escribe('La variable x tiene un valor menor al valor de la variable y')

Imprimirá en pantalla:

La variable x tiene un valor menor al valor de la variable y

b) Supongamos las variables NUM1 = 50 y NUM2=10, NUM3 = 0

```
Si  (NUM1 <> 0 ) entonces
    NUM3  ← NUM1 + NUM2
Si no
    NUM3  ←  NUM1 – NUM2
```

Después de llevarse a cabo “la decisión”, la variable NUM3 tendrá el valor: 60

c) Supongamos k = 10, R = 2

```
    Si ( R  < 100 )
    INICIO
        R = R + k
        ESCRIBE(‘ R =  ’, R)
    FIN
Si no
        ESCRIBE(‘R =’, r)
```

Imprimirá en pantalla: R = 12

d) Supongamos k = 10, R = 300

```
    Si ( R  < 100 )
    INICIO
        R = R + k
        ESCRIBE(‘ R =  ’, R)
    FIN
Si no
        ESCRIBE(‘R =’, R)
```

Imprimirá en pantalla: R = 300

3.4.3 Condicional múltiple

Permite realizar una bifurcación múltiple, ejecutando una entre varias partes del programa, según se cumpla una entre *n* condiciones.

Sintaxis:

Si *selector* igual

Valor1: Hacer proposición 1
Valor2: Hacer proposición 2
Valor3: Hacer proposición 3
.
.
Valor n: Hacer proposición *n*
De otra forma: *Hacer proposición x* *(esta proposición es opcional)*
FIN

Selector es una variable que puede tomar solamente alguno de los valores: valor 1, valor 2, valor 3,...valor n.

Dependiendo del valor que tome selector, se ejecutará alguna de las proposiciones.

Si selector toma valor 1, entonces ejecutará la proposición 1, o bien si selector toma valor 2, entonces ejecutará la proposición 2, y así sucesivamente.

Pero si selector NO toma alguno de los valores, valor 1 hasta valor n, entonces ejecutará la proposición por omisión que es la proposición x.

Ejemplo 3.1: Dados la categoría y el sueldo del trabajador, calcular el aumento correspondiente, teniendo en cuenta la siguiente tabla:

Categoría	Aumento
'A'	15%
'B'	10%
'C'	7%

Tabla 3.6

```
ESCRIBE('Dame la categoría')
LEE(Categ)
ESCRIBE('Dame el sueldo')
LEE(sueldo)

    Si categ igual
      'A': sueldo_nuevo ← sueldo*0.15
      'B': sueldo_nuevo ← sueldo*0.10
      'A': sueldo_nuevo ← sueldo*0.7
    FIN

ESCRIBE('Categoria, sueldo', Categ, sueldo_nuevo)
```

3.5 Estructuras de repetición

Las estructuras de repetición, permiten la ejecución de una lista o secuencia de instrucciones (<bloque de instrucciones>) en varias ocasiones. El número de veces que el bloque de instrucciones se ejecutará se puede especificar de manera explícita, o a través de una condición lógica que indica cuándo se ejecuta de nuevo y cuándo no. A cada ejecución del bloque de instrucciones se le conoce como una iteración.

3.5.1 Ciclos con contador

Ciclo Para

El ciclo **Para** ejecuta un bloque de instrucciones un número determinado de veces. Este número de veces está controlado por una variable contadora (de tipo entero) que toma valores desde un límite inferior hasta un límite superior. En cada ciclo después de ejecutar el bloque de instrucciones, la variable contadora es incrementada en 1 automáticamente y en el momento en que la variable sobrepasa el

límite superior, el ciclo termina. El valor final de la variable contadora depende del lenguaje de programación utilizado, por lo tanto, no es recomendable diseñar algoritmos que utilicen el valor de la variable contadora de un ciclo **Para**, después de ejecutar el mismo. De la definición de ciclo **Para**, se puede inferir que el bloque de instrucciones no se ejecuta si el límite inferior es mayor al límite superior.

Repetitiva Para en Diagramas de Flujo

En un Diagrama de flujo, una instrucción repetitiva **Para** se puede representar del siguiente modo:

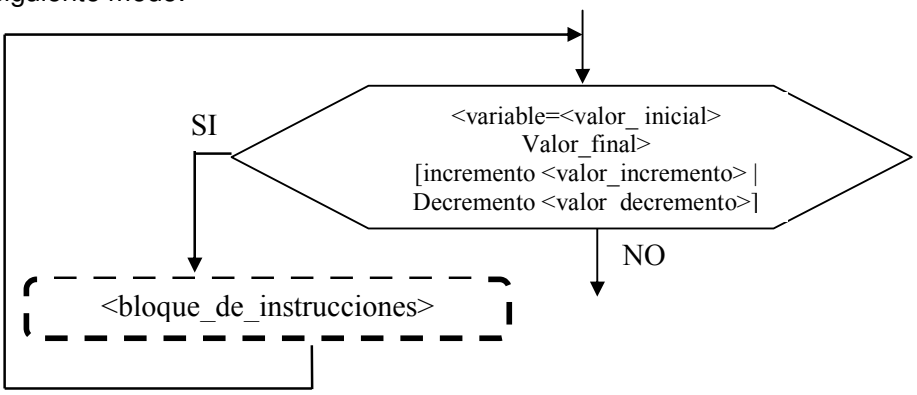


Figura 3.1 Instrucción repetitiva **Para**.

Ejemplo 3.2.
Se quiere diseñar el algoritmo que muestre por pantalla los primeros diez números naturales:

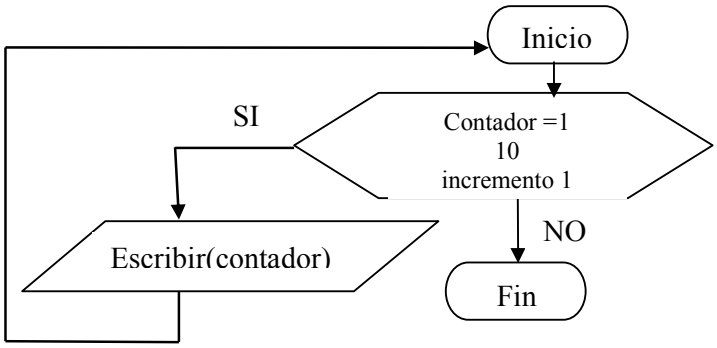


Figura 3.2. Imprime los primeros 10 números naturales

Ejemplo 3.2.
Se quiere diseñar el diagrama de flujo que obtiene la sumatoria de números pares, del 0...100

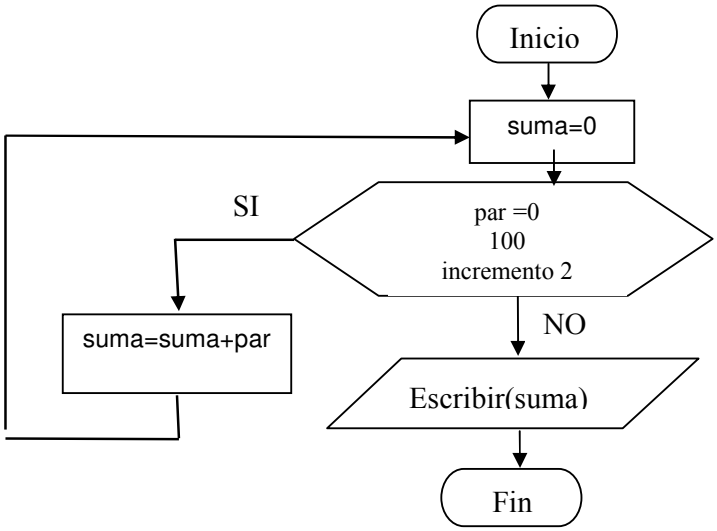


Figura 3.3. Suma los números pares de 0 a 100

Ciclo Para en Pseudocodigo

para <variable> ← <valor_inicial> **hasta** <valor_final>
[**incremento** <valor_incremento>] **hacer**
  <bloque_de_instrucciones>
fin_para

En una instrucción repetitiva **Para**, siempre se utiliza una <variable> a la que se le debe asignar un <valor_inicial>. En cada iteración del bucle, al valor de la <variable> se le suma el <valor_incremento> y, cuando la <variable> supera el <valor_final>, el bucle finaliza.
En consecuencia, cuando el flujo de un algoritmo llega a un bucle **Para**, en primer lugar, se asigna el <valor_inicial> a la <variable> y, a partir de ese instante, existen dos posibilidades:

1. Si el valor de la <variable> es mayor que el <valor_final>, entonces no se ejecuta el bloque de instrucciones, y el bucle **Para** finaliza sin realizar ninguna iteración.
2. Si el valor de la <variable> es menor o igual que el <valor_final>, entonces, se ejecuta el bloque de instrucciones y, después, se le suma el <valor_incremento> a la <variable>, volviéndose, de nuevo, a comparar el valor de la <variable> con el <valor_final>, y así sucesivamente, hasta que el valor de la <variable> sea mayor que el <valor_final>.

En resumen, una **instrucción repetitiva Para** permite ejecutar, repetidamente, un bloque de instrucciones, en base a un valor inicial y a un valor final.
El bucle **Para** es ideal usarlo cuando, de antemano, ya se sabe el número de veces (iteraciones) que tiene que ejecutarse un determinado bloque de instrucciones. El bucle P**ara** es una variante del bucle **mientras** (se discute a continuación) y, al igual que éste, puede iterar cero o más veces. Sin embargo, el bucle **Para** sólo se suele usar cuando se conoce el número exacto de veces que tiene que iterar el bucle.

Tipo de variables útiles para la iteración

Cuando se diseñan algoritmos que incluyen estructuras de control repetitivas, existen ciertas Variables que cumplen una función específica en cada iteración del ciclo, las más comunes son:

• Las variables contadoras
• Las variables acumuladoras
• Las variables bandera.

Variables contadoras
Como su nombre lo indican estas variables se usan fundamentalmente para contar, por lo tanto deben ser de tipo entero. Un ejemplo de este tipo de variables es la variable de control en un Ciclo **Para**.
Una variable contadora se incrementa (o decrementa) en un valor constante en cada iteración del ciclo.

Variables acumuladoras
La función de una variable acumuladora es almacenar valores numéricos que generalmente se Suman (o multiplican) en cada iteración, por lo tanto la variable debe ser de tipo entero o real.

Variables bandera
Una variable bandera es utilizada dentro de la condición de un ciclo, para determinar cuándo un Ciclo se sigue iterando o cuando no. De esta manera una variable bandera debe ser de tipo booleano o entero.

Ejemplo 3.3
Se quiere diseñar el algoritmo que muestre por pantalla los primeros diez números naturales:

```
Nombre del algoritmo: Numeros_del_1_al_10
Variables: contador  Tipo entero
 inicio
   para contador ← 1 hasta 10 incremento 1 hacer
     escribir( contador )
   fin_para
 fin
```

Ejemplo 3.4
Cuando el incremento es 1, se puede omitir la palabra reservada **incremento**, y su valor.

```
Nombre del algoritmo: Numeros_del_1_al_10
Variables: contador  Tipo entero
 inicio
   /* Al no aparecer el valor del incremento,
      se entiende que es 1. */

   para contador ← 1 hasta 10 hacer
     escribir( contador )
   fin_para
 fin
```

La traza de ambos algoritmos es la misma:

Secuencia:	**Acción (instrucción):**	**Valor de:** contador
1	contador ← 1	**1**
2	(Comprobar si contador es menor o igual que **10**)	**1**
	contador sí es menor o igual que **10**. Inicio de la iteración 1.	
3	**escribir(** contador **)**	**1**
	Fin de la iteración 1.	
4	(Sumar a contador el valor **1**)	**2**
5	(Comprobar si contador es menor o igual que **10**)	**2**
	contador sí es menor o igual que **10**. Inicio de la iteración 2.	
6	**escribir(** contador **)**	**2**
	Fin de la iteración 2.	
...		
n-3	(Comprobar si contador es menor o igual que **10**)	**10**
	contador sí es menor o igual que **10**. Inicio de la iteración 10.	
n-2	**escribir(** contador **)**	**10**
	Fin de la iteración 10.	
n-1	(Sumar a contador el valor **1**)	**11**
n	(Comprobar si contador es menor o igual que **10**)	**11**
	contador no es menor o igual que **10**. El bucle finaliza después de 10 iteraciones.	

Figura 3.4. Traza del ejemplo 3.5.3 y ejemplo 3.5.4.

3.5.2 Ciclos Condicionales (Ciclo mientras y Ciclo hacer….mientras)

El **ciclo mientras** permite ejecutar un bloque de instrucciones, mientras una expresión lógica dada se cumpla, es decir, mientras su evaluación dé como resultado verdadero.

La expresión lógica se denomina condición y siempre se evalúa antes de ejecutar el bloque de instrucciones. Si la condición no se cumple, el bloque no se ejecuta. Si la condición se cumple, el bloque se ejecuta, después de lo cual la instrucción vuelve a empezar, es decir, la condición se vuelve a evaluar.

En el caso en que la condición evalúe la primera vez como falsa, el bloque de instrucciones no será ejecutado, lo cual quiere decir que el número de repeticiones o iteraciones de este bloque será cero. Si la condición siempre evalúa a verdadero, la instrucción se ejecutará indefinidamente, es decir, un número infinito de veces.

Repetitiva mientras en Diagramas de Flujo

Sintaxis:

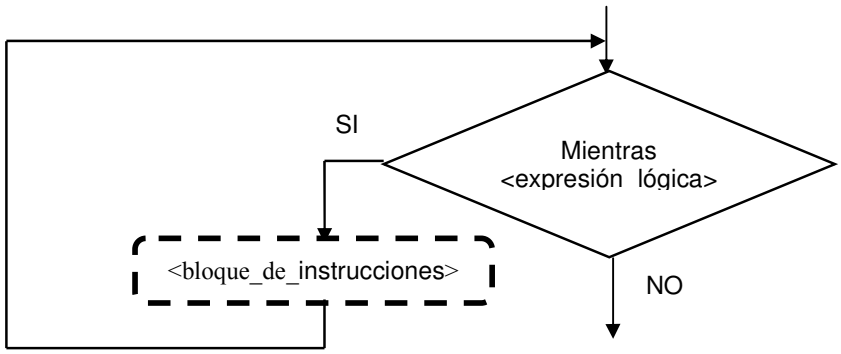


Figura 3.5**.** Instrucción repetitiva **mientras**.

Ejemplo 3.5.
Se quiere diseñar el diagrama de flujo que muestre por pantalla los primeros diez números naturales:

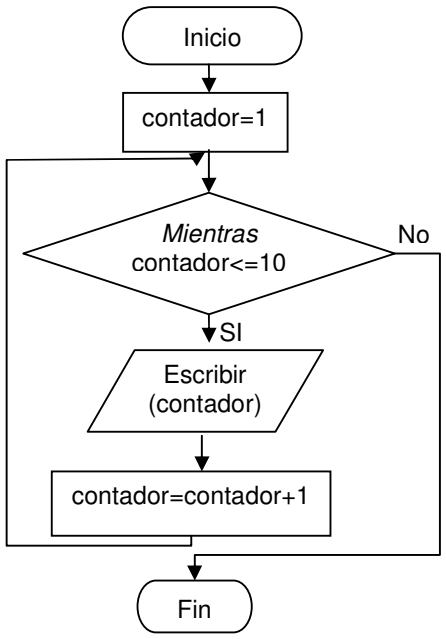


Figura 3.6. Imprime los primeros 10 números naturales

Ejemplo 3.6.
Se quiere diseñar el diagrama de flujo que obtiene la sumatoria de números pares, del 0...100

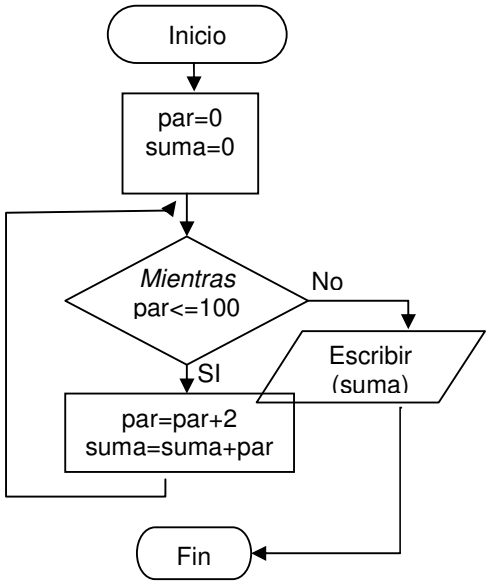


Figura 3.7**.** Suma los números pares de 0 a 100

Repetitiva mientras en Pseudocodigo

Sintaxis

mientras (<expresión_lógica> **)**
<bloque_de_instrucciones>
fin_mientras

Cuando la condición de un bucle **mientras** se evalúa siempre a **verdadera**, se dice que se ha producido un ***bucle infinito***, ya que, el algoritmo nunca termina. Un bucle infinito es un error lógico.
Es importante hacer hincapié en el hecho de que, en un bucle **mientras**, primero se evalúa la condición y, en el caso de que ésta sea **verdadera**, entonces se ejecuta el bloque de instrucciones.
Para que un bucle **mientras** no sea infinito, en el bloque de instrucciones debe ocurrir algo para que la condición deje de ser **verdadera**. En la mayoría de los casos, la condición se hace **falsa** al cambiar el valor de una variable.
En resumen, una **instrucción repetitiva mientras** permite ejecutar, repetidamente, (cero o más veces) un bloque de instrucciones, mientras que, una determinada condición sea **verdadera**.

Ejemplo 3.7.
Obtener un Algoritmo que obtenga la numeración de 1 al 100, usando un ciclo mientras, mostrar cada resultado.

Nombre del algoritmo: Serie_del_1_al_100
Variables: x Tipo **entero**

```
Inicio
  x ← 1
 mientras (x < 101)
    escribir (x)
    x ← x + 1
 fin_mientras
fin
```

Ejemplo 3.8.

Obtener un Algoritmo que sume todos los elementos de la serie del 1 al 100:

Nombre del algoritmo: Suma_del_1_al_100
Variables: X, S Tipo **entero**
Inicio
  S ⇐ 0
  X ⇐ 1
mientras (x **<=** 100**)**
  S⇐ S **+** X
  X⇐ X **+** 1
fin_mientras
escribir (S**)**
fin

Variable Contador
Variable que almacena la cantidad de veces que un proceso dentro de un ciclo se ha completado.
Consiste de dos pasos:

- inicializar la variable
- incrementar la variable por un valor constante al ser completado el ciclo.

Para comprender qué es una variable contador, estúdiese el siguiente ejemplo.

Ejemplo 3.9.
Se quiere diseñar el algoritmo que muestre por pantalla los primeros diez números naturales:

Nombre del algoritmo**:** Numeros_del_1_al_10
Variables: contador Tipo **entero**
inicio
  contador ← 1 /* Inicialización del contador */
  mientras (contador **<=** 10 **)** /* Condición */
    escribir(contador **)** /* Salida */
    contador ← contador **+** 1 /* Incremento */
  fin_mientras
fin

Para comprender el funcionamiento de este algoritmo, se va a estudiar su traza o prueba de escritorio.

La traza del algoritmo es:

Secuencia:	**Acción (instrucción):**	**Valor de:** contador
1	Contador ← 1	**1**
2	(Comprobar si contador **<=** 10)	**1**
	La condición es **verdadera**. Inicio de la iteración	

	1.	
3	**escribir(** contador **)**	**1**
4	contador ← contador **+** 1	**2**
	Fin de la iteración 1.	
5	(Comprobar si contador **<=** 10)	**2**
	La condición es **verdadera**. Inicio de la iteración 2.	
6	**escribir(** contador **)**	**2**
7	contador ← contador **+** 1	**3**
	Fin de la iteración 2.	
...		
n-3	(Comprobar si contador **<=** 10)	**10**
	La condición es **verdadera**. Inicio de la iteración 10.	
n-2	**escribir(** contador **)**	**10**
n-1	contador ← contador **+** 1	**11**
	Fin de la iteración 10.	
n	(Comprobar si contador **<=** 10)	**11**
	La condición es **falsa**. El bucle finaliza después de 10 iteraciones.	

Explicación de la traza:

- Primeramente, se le asigna el valor 1 a contador (acción 1).
- En segundo lugar, se evalúa la condición (contador <= 10) (acción 2) y, puesto que es verdadera, se ejecuta el bloque de instrucciones del bucle mientras.
- Así que, por pantalla se muestra el valor de contador (acción 3) y, después, se incrementa en 1, el valor de la variable contador (acción 4).
- Terminada la ejecución del bloque de instrucciones, se vuelve a evaluar la condición (contador <= 10) (acción 5) y, puesto que es verdadera, se ejecuta de nuevo el bloque de instrucciones.
- Y así sucesivamente, mientras que, la condición sea verdadera, o dicho de otro modo, hasta que, la condición sea falsa.

- En este algoritmo, el bloque de instrucciones del bucle mientras, se ejecuta diez veces (iteraciones).

En el algoritmo del ejemplo se ha utilizado un contador, además en este caso, el valor de la variable contador se ha visualizado en cada iteración.

Cambios en un bucle Mientras

Ejemplo 3.10.
Se quiere diseñar el algoritmo de un programa que muestre por pantalla los primeros diez números naturales, pero a la inversa, es decir, del 10 al 1

```
Nombre del algoritmo: Numeros_del_10_al_1
Variables: contador Tipo entero
 inicio
   contador ← 10              /* Cambio 1 */
   mientras ( contador >= 1 )   /* Cambio 2 */
     escribir( contador )
     contador ← contador - 1   /* Cambio 3 */
   fin_mientras
fin
```

Para que el algoritmo realice la nueva tarea encomendada, ha sido necesario realizar tres cambios en los aspectos más críticos del bucle **mientras**:

1. ***La inicialización de la variable*** contador (cambio 1): necesaria para que la condición pueda evaluarse correctamente cuando el flujo del algoritmo llega al bucle **mientras**.
2. ***La condición del bucle*** **mientras** (cambio 2): afecta al número de iteraciones que va a efectuar el bucle. También se le conoce como condición de salida del bucle.
3. ***La instrucción de asignación*** (cambio 3): hace variar el valor de la variable contador dentro del bloque de instrucciones. De no hacerse correctamente, el bucle podría ser infinito.

Errores en un bucle mientras
En este apartado vamos a ver varios ejemplos de posibles errores que se pueden cometer al escribir un bucle **mientras**, tomando como referencia el ejemplo del apartado anterior.

Ejemplo 3.11.
Un pequeño descuido, como por ejemplo, no escribir de forma correcta la condición del bucle, puede producir un bucle infinito:

```
Nombre del algoritmo: Ciclo_infinito
Variables: contador Tipo entero
 inicio
   contador ← 10              /* Cambio 1 */
   mientras ( contador <= 10 )  /* Descuido */
     escribir( contador )
     contador ← contador - 1   /* Cambio 3 */
   fin_mientras
 fin
```

Ejemplo 3.12.

Otro error muy frecuente es inicializar mal la variable que participa en la condición del bucle:

```
Nombre del algoritmo: Imprime_sólo_el_1
Variables: contador Tipo entero
 inicio
    contador ← 1              /* Descuido */
    mientras ( contador >= 1 )   /* Cambio 2 */
       escribir( contador )
       contador ← contador - 1   /* Cambio 3 */
    fin_mientras
 fin
```

Por pantalla sólo se mostrará el número **1**:

Ejemplo 3.13.

También es un error muy típico olvidarse de escribir alguna instrucción, como por ejemplo, la instrucción de asignación.

contador ← contador **-** 1
del bloque de instrucciones del bucle:

```
Nombre del algoritmo: Sólo_ciclo_infinito
Variables: contador Tipo entero
inicio
   contador ← 10              /* Cambio 1 */
   mientras ( contador >= 1 )   /* Cambio 2 */
      escribir( contador )
                          /* Descuido */
   fin_mientras
fin
```

De nuevo, por pantalla, se obtiene la salida de un bucle infinito:

Ejemplo 3.14.

Como ya se ha dicho, un bucle **mientras** puede iterar cero o más veces. Así, por ejemplo, en el algoritmo siguiente existe un error lógico que provoca que el bucle no itere ninguna vez.

```
Nombre del algoritmo: Cero_iteraciones
Variables: contador Tipo entero
inicio
   contador ← 0              /* Descuido */
   mientras ( contador >= 1 )   /* Cambio 2 */
      escribir( contador )
      contador ← contador - 1   /* Cambio 3 */
   fin_mientras
fin
```

Por pantalla no se mostrará nada:

En este caso, se ha producido un error lógico, ya que, para que el bucle iterase diez veces, se debería haber asignado a la variable contador el valor **10**, en vez del 0.
No obstante, bajo determinadas circunstancias, sí puede tener sentido hacer uso de un bucle **mientras**, el cual pueda no iterar ninguna vez.

Uso de un bucle Mientras que puede no iterar
Para comprender el porqué puede tener sentido utilizar un bucle **mientras**, el cual pueda no iterar ninguna vez, estúdiese el siguiente problema.

Ejemplo 3.15.
Se quiere diseñar el algoritmo de un programa que:
1º) Pida por teclado la nota (dato real) de una asignatura.
2º) En el caso de que la nota sea incorrecta, muestre por pantalla el mensaje:

- "ERROR: Nota incorrecta, debe ser >= 0 y <= 10".

3º) Repita los pasos 1º y 2º, mientras que, la nota introducida sea incorrecta.
4º) Muestre por pantalla:

- "APROBADO", en el caso de que la nota sea mayor o igual que 5.
- "SUSPENDIDO", en el caso de que la nota sea menor que 5.

El algoritmo propuesto es:

```
Nombre del algoritmo: Aprobado_o_suspendido
Variables: nota Tipo real
inicio
   escribir( "Introduzca nota (real): " )
   leer( nota )

   /* Si la primera nota introducida por el usuario
      es correcta, el bucle no itera ninguna vez. */

   mientras ( nota < 0 o nota > 10 )
      escribir( "ERROR: Nota incorrecta, debe ser >= 0 y <= 10" )
      escribir( "Introduzca nota (real): " )
      leer( nota )
   fin_mientras

   /* Mientras que el usuario introduzca una nota  incorrecta, el bucle iterará. Y cuando
introduzca una nota correcta, el bucle finalizará. */

   si ( nota >= 5 )
      escribir( "APROBADO" )
   sino
      escribir( "SUSPENDIDO" )
   fin_si
fin
```

En el algoritmo, el bucle **mientras** se ha usado para validar la nota introducida por el usuario. En programación, es muy frecuente usar el bucle **mientras** para validar