



**Ingeniería en Desarrollo de Software**  
**Primer Semestre**

Programa de la asignatura:  
**Fundamentos de programación**

**Unidad 2. Diseño de algoritmos**

Clave:

<b>TSU</b>	<b>Licenciatura</b>
16141102	15141102

**Universidad Abierta y a Distancia de México**





### Índice

Unidad 2: Diseño de algoritmos .....	3
Presentación.....	3
Propósitos.....	3
Competencia específica.....	4
2.1 Concepto de algoritmo y características .....	4
2.2. Representación de algoritmos.....	6
2.2.1. Pseudocódigo .....	7
2.2.2. Diagrama de flujo .....	8
2.3. Estructuras de control .....	11
2.3.1. Estructuras secuenciales .....	12
2.3.2 Estructuras selectivas .....	14
2.3.3. Estructuras repetitivas.....	18
Fuentes de consulta .....	21



## Unidad 2: Diseño de algoritmos

### Presentación

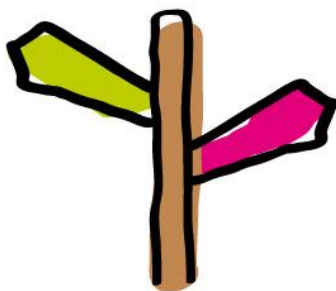
En la unidad anterior se describieron los pasos que se requieren para construir un *software*, los cuales, sin lugar a dudas, constituyen la fase más importante en el diseño de la solución de problemas, ya que es aquí donde se debe crear el modelo que contribuya a la solución del problema en cuestión.

Para llegar a esta solución se requiere no sólo de inteligencia, sino también de creatividad, ya que el programador sólo cuenta con la especificación del problema y su experiencia en resolver problemas de una forma estructurada.

En este apartado te presentaremos formalmente el concepto de **algoritmo**, del que estudiaremos sus características y dos maneras de representarlo: una gráfica, conocida como **diagramas de flujo**; y otra, similar a un lenguaje humano (en este caso español), la cual se denomina **pseudocódigo**. También describiremos los tres tipos de **estructuras de control: secuenciales, selectivas y repetitivas**, que son las instrucciones con que se cuenta en la programación estructurada para diseñar soluciones.

Para lograr nuestro objetivo utilizaremos una situación ficticia a la que llamamos “el mundo de la ardilla”, en donde se deben solucionar problemas mediante un conjunto de instrucciones específicas que puede ejecutar una ardilla sobre un tablero o mundo determinado.

### Propósitos



- Identificar los datos de entrada y la salida de un algoritmo
- Diseñar un algoritmo que solucione un problema.
- Representar el algoritmo en diagrama de flujo y pseudocódigo
- Verificar que el algoritmo calcule el resultado correcto



### Competencia específica



Diseñar algoritmos para resolver problemas mediante su representación en un diagrama de flujo y la elaboración del pseudocódigo.

### 2.1 Concepto de algoritmo y características

Antes de explicar cómo se realizan los algoritmos en la programación, conviene tener claridad en el significado de la palabra **algoritmo**.

La palabra *algoritmo* proviene del nombre de un matemático persa conocido como *Mohammad Al-KhoWârizmi*, nacido alrededor del 780 d.c. en *KhoWârizm*, de ahí el su seudónimo. Se considera como el padre de la algoritmia porque definió las reglas paso a paso para sumar, restar, multiplicar y dividir números decimales. La traducción al latín de *Al-Khwârizmî* es *algoritmi*, que da origen a la palabra *algoritmo* (Joyanes & Zohanero, 2005).

Formalmente, un *algoritmo* se define como una secuencia finita de instrucciones precisas y eficaces para resolver un problema, que trabaja a partir de cero o más datos (*entrada*) y devuelve un resultado (*salida*).<sup>1</sup>

Para ilustrar este concepto se presenta el siguiente escenario ficticio, que hemos llamado:

#### **El mundo de la ardilla**

Supongamos que una ardilla ha sido entrenada para realizar las instrucciones que se muestran en la tabla 2.1, sobre un tablero.

---

<sup>1</sup> Esta definición es una adaptación de la que aparecen en (Viso & Pelaez, 2007, pág. 3)



INSTRUCCIÓN	DESCRIPCIÓN DE LOS MOVIMIENTOS DE LA ARDILLA
avanza()	Se mueve una ubicación en la dirección actual
giralzquierda()	Voltea a la izquierda
dejaBellota()	Coloca una bellota en la ubicación actual
hayBellota()	Responde si hay o no bellotas en la posición actual
hayPared()	Responde si hay o no pared en la ubicación siguiente
recogeBellota()	La ardilla coloca en su boca una bellota que está en la ubicación actual <sup>2</sup>
bellotasRecogidas()	Dice el número de bellotas que tiene en la boca

**Tabla 2.1:** Lista de instrucciones que puede ejecutar la ardilla

Los paréntesis al final de cada instrucción sirven para identificar que se trata de una orden que puede ejecutar la ardilla.

Si observas la lista de instrucciones podrás darte cuenta que, la ardilla no es capaz de voltear a la derecha y mucho menos de responder a órdenes más complejas como *“mueve una bellota que se encuentra en la primera casilla del tablero al final del mismo”*. Sin embargo, podría realizar ambas tareas si se le dan las instrucciones precisas en términos de las acciones que sabe hacer. Por ejemplo, para que la ardilla gire a la derecha tendríamos que ordenarle tres veces que girará a la izquierda, es decir, la secuencia de instrucciones que debe ejecutar es:

```
giralzquierda()  
giralzquierda()  
giralzquierda()
```

Estos pasos constituyen un *algoritmo*, el cual soluciona el problema de hacer que la ardilla gire a la derecha.

Una de las características principales de los algoritmos es que cada paso debe estar definido de forma clara y precisa, sin ambigüedades, de tal manera que pueda ejecutarse de manera inequívoca, por ejemplo, en el mundo de la ardilla, la instrucción *gira()* sería

---

<sup>2</sup>La ardilla posee una bolsa donde almacena cualquier cantidad de bellotas.



una instrucción ambigua, ya que la ardilla no sabría si debe girar a la derecha o a la izquierda.

Otra característica de los algoritmos es que siempre terminan, por lo que no puede ser una lista infinita de pasos. Y tampoco puede contener pasos que sean irrealizables o cuya ejecución sea infinita, pues en este caso no sería posible calcular el resultado deseado, si una instrucción está bien definida y es eficaz se puede asegurar que su ejecución termina con éxito, sin embargo, esto no garantiza, de ninguna manera, que el algoritmo también termine.

Por lo anterior, al diseñar un algoritmo se debe garantizar que dada cualquier entrada siempre termine y calcule la respuesta correcta. De tal manera que todo algoritmo debe tener las siguientes características:

1. **Entrada.**
2. **Salida.**
3. **Definido.**
4. **Eficaz.**
5. **Terminación.**

Una vez que se ha diseñado un algoritmo, se recomienda realizar una *prueba de escritorio* para verificar si funciona correctamente, ésta consiste en ejecutar el algoritmo utilizando papel y lápiz, se propone datos de entrada específicos y se realiza cada una de las instrucciones en el orden establecido, registrando los cambios que se producen después de la ejecución de cada instrucción. De esta manera, se valida que el resultado obtenido en la prueba de escritorio corresponda al resultado deseado (el correcto).

## 2.2. Representación de algoritmos

Existen diversas formas de representar un algoritmo, en la unidad anterior expusimos diversas formas de representar la solución del problema de calcular el área de un rectángulo, por ejemplo, en el programa 1.1 se expresa la solución en pseudocódigo, después en el algoritmo 1.1 se representa en lenguaje natural (español) y en el programa 1.2 se utiliza el lenguaje de programación C, o se puede expresar mediante la fórmula matemática:

$$\text{área} = \text{base} \times \text{altura}$$

Todas estas representaciones, excepto el lenguaje natural, se consideran formales, y cabe mencionar que existen más, sin embargo, las representaciones más comunes son el pseudocódigo y los diagramas de flujo. La primera, generalmente se utiliza por su parecido con el lenguaje natural (español, inglés, francés o cualquier otro) y porque su codificación en un lenguaje de programación estructurado y modular, como C, es directa.





En cambio, los diagramas de flujo son totalmente gráficos, lo que hace más fácil seguir el orden en que se ejecutan las instrucciones. Es importante mencionar que se puede utilizar cualquiera de las dos representaciones para diseñar un algoritmo, pues en cualquiera de los dos se puede expresar cualquier algoritmo estructurado, de tal manera que la más conveniente depende de cada programador. En las siguientes secciones se presenta cada uno de ellos y así podrás decidir cuál prefieres.

### 2.2.1. Pseudocódigo

El *pseudocódigo* es un lenguaje de especificación formal de algoritmos. La solución de un problema se representa de manera narrativa utilizando *palabras claves*, generalmente verbos, escritos en un lenguaje natural, que en nuestro caso será español. Para ilustrarlo construyamos un algoritmo que resuelva el siguiente problema.

**Problema 2.1:** En la figura 2.1.a. se muestra el estado inicial de un tablero, el cual contiene en la primer casilla (de izquierda a derecha) una bellota, representada por un asterisco (\*), y a la ardilla, representada por una flecha que apunta hacia la dirección que está mirando. El problema consiste en diseñar un algoritmo que la ardilla pueda ejecutar para llegar al estado meta representado en la figura 2.1.b., que implica que la ardilla lleve la bellota a la última casilla. Para resolverlo se tiene la siguiente información:

- El mundo es conocido, es decir, se sabe de antemano que el tablero está cercado por paredes y sólo tiene seis casillas colocadas en línea.
- Al inicio la ardilla está en la primera casilla volteando hacia arriba y no tiene ninguna bellota en la boca.
- En la primera casilla hay una bellota.

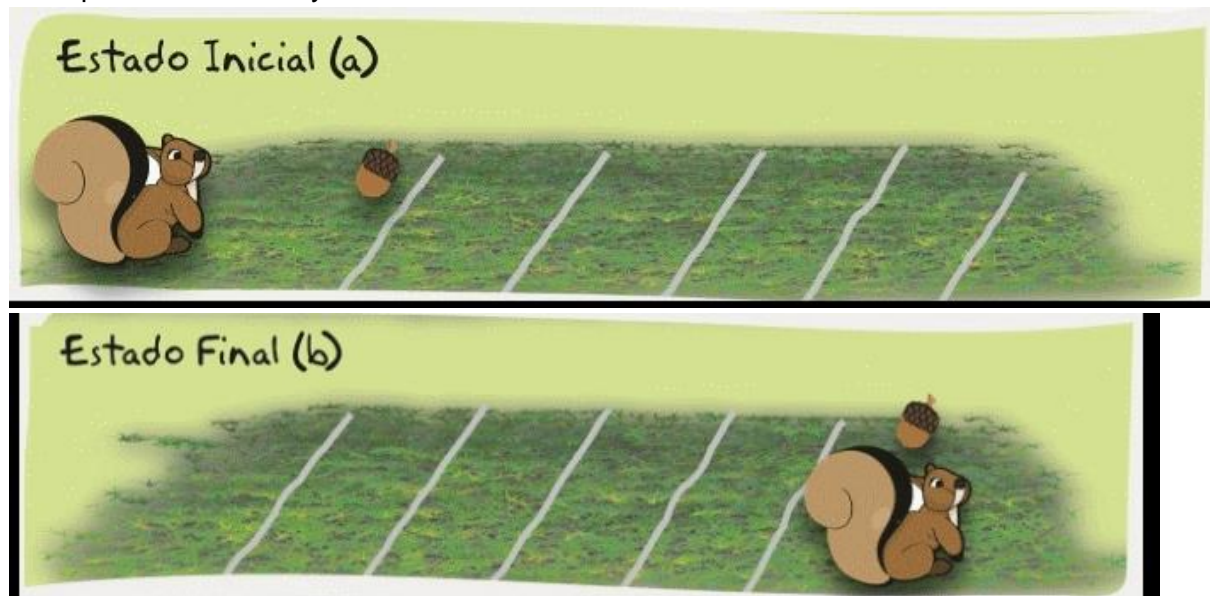


Figura 2.1: Primer mundo lineal



**Análisis:** Haciendo un rápido análisis del problema, nos podemos dar cuenta que la ardilla debe recoger la bellota, avanzar cinco casillas y soltar la bellota, esto traducido en un algoritmo queda de la siguiente forma:

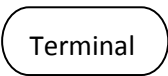


**Algoritmo 2.1.** Primer mundo de la ardilla

En este caso las instrucciones son parecidas al lenguaje natural.

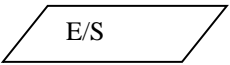



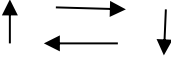

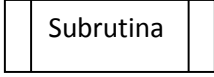
### 2.2.2. Diagrama de flujo

Los **diagramas de flujo** son una representación gráfica de un algoritmo que utiliza **símbolos** para representar las instrucciones y flechas para unirlos e indicar el orden en que deben ejecutarse llamadas **líneas de flujo**. Estos símbolos fueron normalizados por el Instituto Norteamericano de Normalización ANSI (*American National Standards Institute*, por sus siglas en inglés). Los símbolos más utilizados se muestran en el siguiente cuadro:

Símbolo	Descripción
	<i>Terminal</i> . Representa el inicio y el final de un algoritmo.





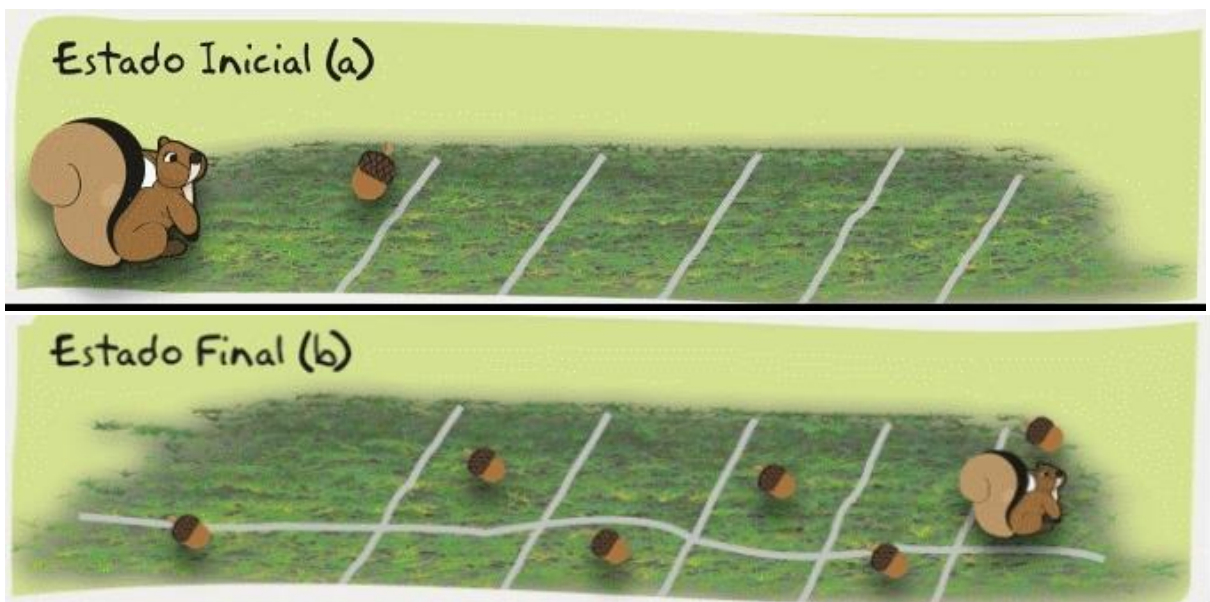
	<i>Entrada y Salida (E/S).</i> Representa la lectura de datos desde el dispositivo de entrada estándar, así como la impresión de datos en el dispositivo de salida estándar.
	<i>Proceso.</i> Representa cualquier tipo de operación que pueda originar un cambio de la información almacenada en memoria, asignaciones u operaciones aritméticas.
	<i>Decisión.</i> Nos permite analizar una situación, con base en los valores verdadero y falso. Toma una decisión de las instrucciones que a continuación ejecuta el algoritmo.
	<i>Conector.</i> Sirve para enlazar dos partes cualesquiera del diagrama que están en la misma página.
	<i>Línea de flujo.</i> Indica el orden de la ejecución de las operaciones. La flecha indica cuál es la siguiente instrucción que se debe realizar.
	<i>Conector.</i> Conecta a dos puntos del diagrama cuando éstos se encuentran en páginas diferentes. Representa el inicio y el final de un programa.
	<i>Llamada a subrutina.</i> Llama a un proceso determinado o subrutina. Una subrutina es un módulo independiente del módulo principal, que realiza una tarea determinada y al finalizar regresa el control de flujo al módulo principal.

**Tabla 2.2** Símbolos de los diagramas de flujo

**Problema 2.2:** Ahora la tarea de la ardilla es que cambie las bellotas que están en la primera fila (ver figura 2.2.a) a la segunda y viceversa, dejándolas en la misma columna (ver figura 2.2.b).

Las condiciones de inicio son:

- El mundo es conocido y sabemos exactamente dónde hay bellotas.
- La ardilla no tiene ninguna bellota en la boca al inicio.
- El mundo está encerrado por paredes y si la ardilla choca contra una se considerará un error garrafal.
- En este punto los científicos ya entrenaron a la ardilla para ejecutar la orden *giraDerecha()*, por lo tanto, ya puede ser usada en el algoritmo.



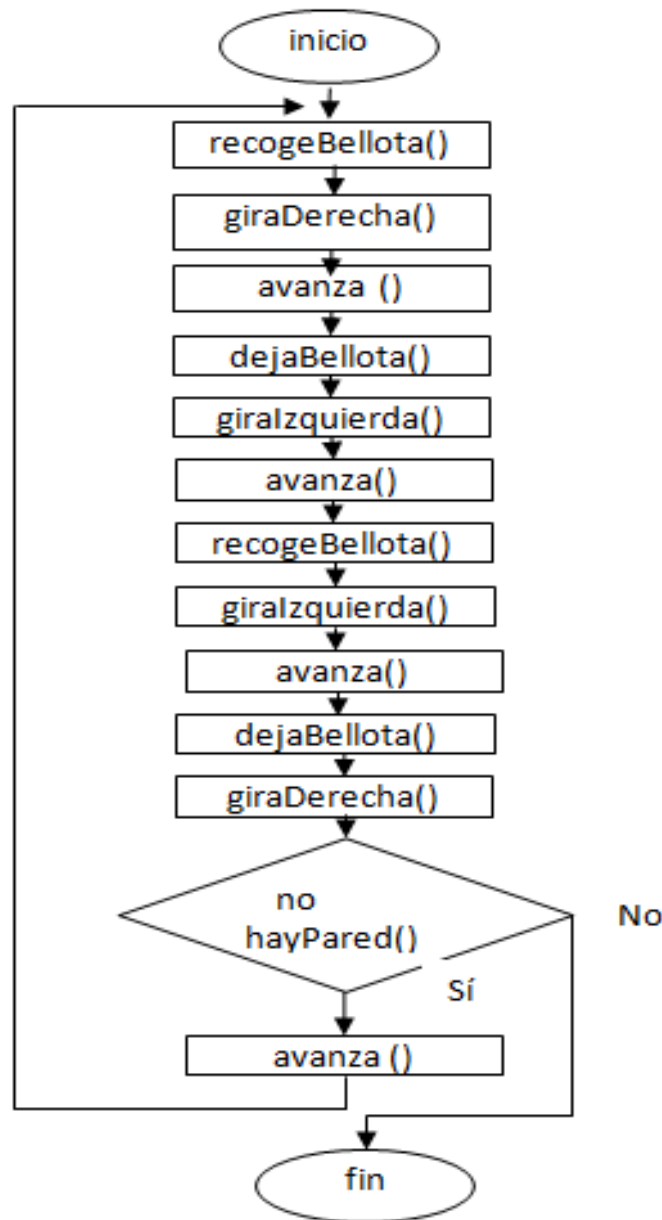
**Figura 2.2:** Segundo mundo de la ardilla

**Análisis:** De acuerdo con la figura 2.2, inciso a, para que la ardilla cumpla con su tarea debe realizar los siguientes pasos: recoger la bellota, girar a la derecha, avanzar, dejar la bellota, girar a la izquierda, avanzar, recoger la bellota, girar a la izquierda, avanzar, dejar la bellota, voltear a la derecha y avanzar. Hasta este punto las coordenadas de la ardilla son: primera fila y tercera casilla (volteando a la derecha, como al inicio).



Si la ardilla repite otra vez este bloque de instrucciones, logrará cambiar las siguientes dos bellotas; al repetirlo nuevamente cambiaría las últimas dos, salvo que cuando la ardilla avance después de haber dejado la bellota chocará contra la pared, por lo tanto, antes de que avance –última instrucción del bloque – tenemos que verificar que no haya pared. La condición para que la ardilla repita el bloque de instrucciones es que no haya pared.

De lo anterior tenemos el siguiente algoritmo representado en diagrama de flujo.



Algoritmo 2.2. Solución problema 2.2

### 2.3. Estructuras de control

Los primeros lenguajes de programación de alto nivel permitían realizar “saltos” a diferentes líneas del código mediante la instrucción GOTO, esto tiene el gran inconveniente que cuando se hacía una modificación en el programa, era necesario modificar todas las instrucciones GOTO para asegurar que los saltos se hicieran a las



líneas de código correctas. Además de lo tedioso que podía ser estar corrigiendo el programa, las instrucciones GOTO lo hacían difícil de leer.

En 1966 Corrado Böhm y Giuseppe Jacopini demostraron que “cualquier algoritmo puede diseñarse e implementar utilizando únicamente tres tipos de estructuras de control: secuenciales, condicionales y repetitivas; esto es, sin utilizar GOTO”(Böhm & Jacopini, 1966), basándose en este resultado, a principios de los años 70's Edsger Dijkstra se dio cuenta que la forma en la que los lenguajes de programación de alto nivel podían modificarse sin problemas era eliminando las instrucciones GOTO (o similares), así que propuso un nuevo estilo de programación al que llamó *programación estructurada*, ésta incluye estructuras secuenciales, selectivas y repetitivas, conocidas como *estructuras de control*.

### 2.3.1. Estructuras secuenciales

Las *estructuras secuenciales* son un bloque de instrucciones que se ejecutan una tras otra, en el mismo orden en el que están escritas.

Un ejemplo de este tipo de instrucciones son todas las que se utilizaron en el algoritmo 2.1. Veamos otro ejemplo.

**Problema 2.3:** Ahora la ardilla se enfrenta a un nuevo mundo (ver figura 2.3) en el que su tarea consiste en recoger las dos bellotas colocadas en la posiciones indicadas por la figura 2.3.a y llevarlas a la última casilla de la primera fila, como se muestra en la figura 2.3.b. Considerando que tenemos un mapa del nuevo mundo y sabemos en qué casillas están colocadas las bellotas diseñemos un algoritmo para que la ardilla realice su cometido.

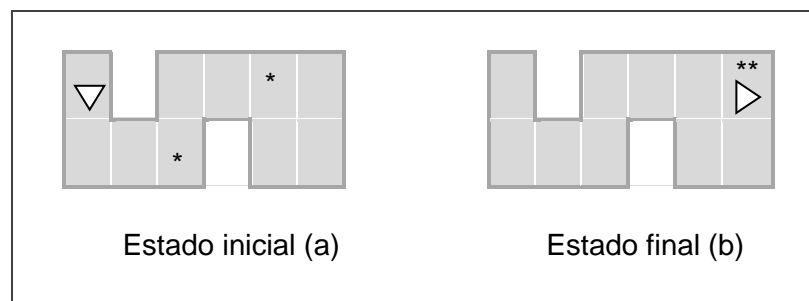


Figura 2.3. Tercer mundo de la ardilla

**Análisis:** Nuevamente el problema planteado es muy sencillo de analizar, la ardilla debe hacer los movimientos que le permitan recoger la primera bellota, después ir por la segunda y llegar a la última casilla de la primera fila. Otra posible opción es que recoja la primera bellota, la lleve a la primera casilla, regrese por la segunda bellota y también la lleve a la primera casilla. Esta última opción requiere más esfuerzo por parte de la ardilla,





dado que la ardilla no tiene limitado el número de bellotas que puede llevar en la boca, entonces la primera opción es más eficiente. El algoritmo quedaría como:



**Algoritmo 2.3.** Solución problema 2.3.

Las instrucciones selectivas, más usuales, que una computadora es capaz de realizar son: *Imprimir*, *Leer* y *Asignar*.

La representación en diagrama de flujo de estas instrucciones se ilustra en la siguiente tabla, en cuanto que la representación en diagrama de flujo se utilizan los mismos verbos y símbolos pero encerrados entre un símbolo de proceso.

Tipo	Pseudocódigo	Diagrama de flujo	Descripción
Asignación	<code>&lt;var&gt; ← &lt;expresión&gt;</code>	<code>&lt;var&gt; ← &lt;expresión&gt;</code>	Asigna el valor de la expresión variable <var>
Entrada y Salida	Imprimir <mensaje/ variable>	Imprimir var/mens	Envía a pantalla un mensaje o el valor de la variable indicada. En caso de que se imprima un mensaje debe estar escrito entre comillas, si es el valor de una variable sólo se pondrá el nombre de la variable (sin comillas).
	Leer <variable>	imprimir var/mens	Lee un dato del teclado y lo almacena en la variable indicada.

**Tabla 2.3** Estructuras secuenciales





### 2.3.2 Estructuras selectivas

En esencia, las *estructuras selectivas* se utilizan cuando la solución de un problema conlleva tomar una decisión, ya que se ejecuta un conjunto determinado de instrucciones dependiendo de si se cumple o no una condición en un momento determinado. Por ejemplo, la ardilla solamente puede avanzar si se no hay pared, en este caso la condición es *no hayPared()* y la acción que se realiza es *avanza()*. Revisemos el siguiente ejemplo:

**Problema 2.4:** Nuevamente la ardilla está en el mundo lineal que se ilustra en la figura 2.4.a, tiene que recoger una bellota y llevarla a la última casilla como se muestra en la figura 2.4.b, sólo que ahora no sabe con precisión en que casilla está la bellota y la única información con la que cuenta es la siguiente:

- a) En el tablero hay una sola bellota. Las casillas donde puede estar son la tercera o la quinta, lo cual se representa con un círculo en la figura 2.4.a.
- b) Al inicio la ardilla no tiene ninguna bellota en la boca.
- c) Es un error ordenar a la ardilla que recoja una bellota en una casilla cuando esta no contiene nada pues la ardilla no sabrá que hacer.
- d) La ardilla ya ha sido entrenada para decir si hay bellota.

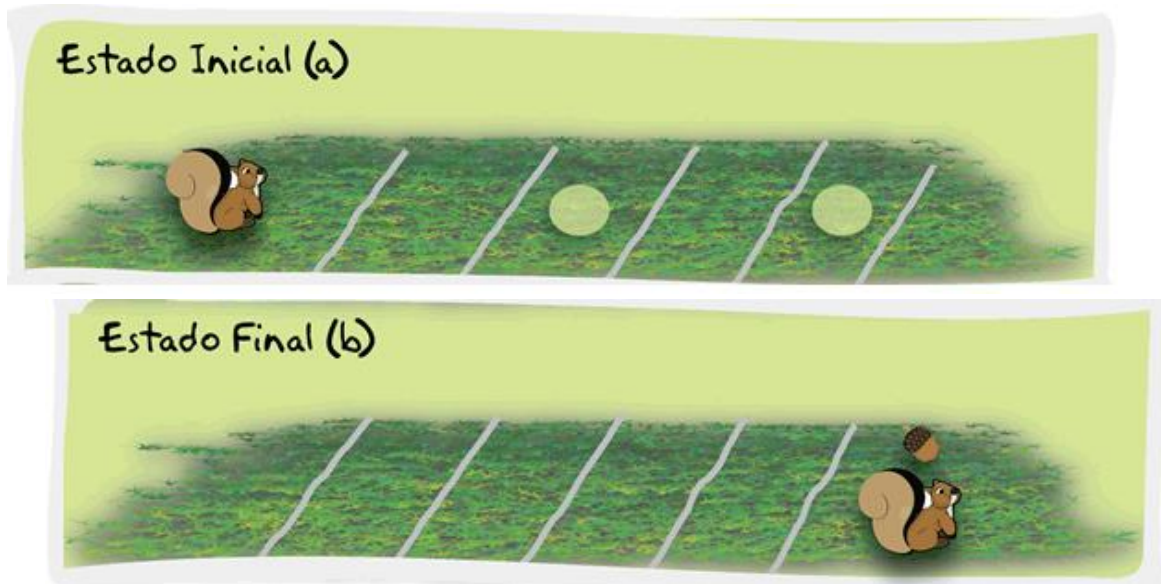


Figura 2.4. Cuarto mundo de la ardilla

**Análisis:** En este caso es necesario asegurarnos que en la casilla hay una bellota, antes de ordenarle a la ardilla que la recoja, para ello vamos a utilizar una estructura selectiva, como la ardilla ya fue entrenada para decir si hay una bellota, entonces esto lo utilizaremos como condición. Ya que tenemos dos posibles lugares dónde la ardilla puede encontrar la bellota, ordenaremos a la ardilla que avance hasta la tercera casilla, si hay una bellota entonces lo recoge y después la lleva a la última casilla, sino la ardilla avanza hasta la quinta casilla y ahí recoge la bellota, esto sin preguntar si ahí se encuentra pues



una de las aseveraciones en el planteamiento del problema es que en el tablero hay una bellota, así que si éste no estaba en la tercera casilla es seguro que está en la quinta.

```
Inicio
avanza()
avanza()
si haybellota()
entonces
    recogeBellota()
    avanza()
    avanza()
    avanza()
    dejaBellota()

Sino
    avanza()
    avanza()
    recogeBellota()
    avanza()
    dejaBellota()

Fin Si-SiNo
Fin
```

**Algoritmo 2.4.** La ardilla toma decisiones en un mundo lineal, versión 1

Observa que tanto en el primer caso (Si) como en el segundo (Sino) cuando la ardilla está en la quinta casilla y ya recogió la bellota, las siguientes órdenes es que avance y deje la bellota (ambas están remarcadas), de tal manera que podemos modificar el algoritmo de la siguiente forma:



```
Inicio
avanza()
avanza()
si hayBellota() entonces
    recogeBellota()
    avanza()
    avanza()
Sino
    avanza()
    avanza()
    recogeBellota()
Fin Si-SiNo
    avanza()
    dejaBellota()
Fin
```

**Algoritmo 2.5.** La ardilla toma decisiones en un mundo lineal, versión 2

También podemos utilizar la estructura Si dos veces, una para preguntar si la bellota está en la tercera casilla y otra para preguntar en la quinta, como se muestra en el siguiente algoritmo.

```
Inicio
avanza()
avanza()
Si hayBellota() entonces
    recogeBellota()
Fin Si
avanza()
avanza()
Si hayBellota() entonces
    recogeBellota()
Fin Si
avanza()
dejaBellota()
Fin
```

**Algoritmo 2.6.** La ardilla toma decisiones en un mundo lineal, versión 3



A diferencia de los dos algoritmos anteriores, en éste la ardilla va a verificar en las dos casillas si hay bellota, aunque la haya encontrado en la primera opción, esto implica un poco más esfuerzo para la ardilla.

Por otro lado, observa que en los algoritmos 2.4 y 2.5 se definieron instrucciones para el caso que se cumple la condición (Si) y para el caso que no (Sino); en cambio, en este último algoritmo sólo se ha definido un conjunto de instrucciones que se ejecuta si la condición se cumple, de no ser así no hay instrucciones específicas y la ardilla continuará realizando las siguientes instrucciones del algoritmo. Es importante destacar que ambas estructuras son equivalentes, es decir, que los problemas que se solucionan con una también es posible hacerlo con la otra.

Existen tres tipos de estructuras selectivas que se clasifican de acuerdo al número de alternativas:

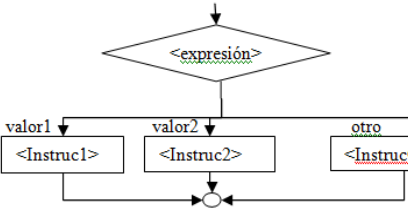
- Simples (Si)*: involucra un único bloque de instrucciones que se ejecuta sólo si una condición se cumple.
- Dobles (Si-Sino)*: abarca dos bloques de instrucciones, uno se ejecuta cuando la condición es verdadera y el otro cuando es falsa.
- Múltiples (Seleccionar)*: tiene más de dos bloques de instrucciones que se pueden ejecutar conforme al valor que tenga una variable, cada bloque equivale a un valor diferente.

En la siguiente tabla se muestra la representación en pseudocódigo y diagrama de flujo de estas estructuras.

Tipo	Pseudocódigo	Diagrama de flujo	Descripción
Estructuras selectivas simples	Si <condición> entonces <instrucciones> Fin_Si		Si la condición es verdadera, se ejecuta el conjunto de instrucciones.
Estructuras selectivas dobles	Si <condición> entonces <instrucciones V> Sino <instrucciones F> Fin_Si-Sino		Si la condición se cumple se ejecuta el conjunto de instrucciones V, en caso contrario se realizan las instrucciones F.





Estructuras selectivas múltiples	Seleccionar <expresión> <b>caso</b> <valor1>: <instrucciones1> <b>caso</b> <valor2>: <Instrucciones2> . . . <b>otro:</b> <instruccionesOtro> Fin_Seleccionar		Compara el valor de expresión con cada uno de los valores correspondientes a cada caso, es decir cada <i>valor K</i> y sólo si son iguales realiza las instrucciones correspondientes.
----------------------------------	---	--	--

**Tabla 2.4** Estructuras selectivas

En la unidad cuatro estudiaremos a mayor detalle cada una de estas estructuras.

### 2.3.3. Estructuras repetitivas

Las *estructuras repetitivas*, también llamadas *ciclos*, permiten ejecutar varias veces un bloque de instrucciones en función de una condición. Para ilustrar esto, volvamos al problema 2.1 del subtema 2.2.1; en este mundo la ardilla debe llevar una bellota desde la primera casilla hasta la última en un mundo lineal (ver figura 2.1). Observa que una vez que la ardilla recoge la bellota y está viendo de frente, debe avanzar una y otra vez mientras no se tope con la pared, esto se puede modelar con un ciclo de la siguiente manera.



**Algoritmo 2.7.** Solución problema 2.1 utilizando ciclos





Generalmente, un ciclo se utiliza cuando descubrimos un patrón, tal como se hizo en el análisis del problema 2.2. Si observas el algoritmo 2.2, verás que al final hay una flecha que regresa a la primera instrucción, representado con ello un ciclo. La presentación en pseudocódigo de este algoritmo sería la siguiente:

```
Inicio
Mientras no(hayPared()) hacer
  recogeBellota()
  giraIzquierda()
  giraIzquierda()
  giraIzquierda()
  avanza()
  dejaBellota()
  giraIzquierda()
  avanza()
  recogeBellota()
  giraIzquierda()
  avanza()
  dejaBellota()
  giraIzquierda()
  giraIzquierda()
  giraIzquierda()
Si no(hayPared()) entonces
  avanza()
Fin Si
Fin Mientras
Fin
```

**Algoritmo 2.8:** Solución del problema 2.2 utilizando ciclos

La clave para utilizar un ciclo es identificar el conjunto de instrucciones que se deben repetir y la condición para que se ejecuten. Al igual que en las estructuras selectivas, existen diferentes estructuras repetitivas que se diferencian, principalmente, por el orden en el que se evalúa la condición. Éstas son:

- Mientras-hacer:* en este ciclo primero se verifica que la condición sea verdadera y en tal caso se ejecuta el bloque de instrucciones y se repite nuevamente el ciclo.
- Hacer-Mientras:* en esta estructura primero se realizan las instrucciones y después se verifica la condición, si se cumple se repite el ciclo.
- Desde-mientras:* funciona igual que Mientras pero tiene asociada una variable que sirve como contador para controlar el número de veces que se repite un ciclo, de tal manera que la condición involucra al contador.

La representación en pseudocódigo y diagrama de flujo de estas estructuras se muestran en la siguiente tabla:



Tipo	Pseudocódigo	Diagrama de flujo	Descripción
Ciclo Mientras (while)	<b>Mientras</b> <condición> <b>hacer</b> <instrucciones> <b>Fin_Mientras</b>		Verifica si la <i>condición</i> se cumple, en tal caso ejecuta el conjunto de <i>instrucciones</i> y se vuelve a repetir el ciclo.
Ciclo Hacer-Mientras (do while)	<b>Hacer</b> <instrucciones> <b>Mientras</b> <condición>		A diferencia del <i>Mientras</i> , esta estructura primero ejecuta el conjunto de <i>instrucciones</i> y después verifica que la <i>condición</i> se cumpla, en caso de ser verdadera se repite el ciclo.
Ciclo Desde-mientras (for)	<b>Desde</b> <inicialización> <b>mient</b> <b>ras</b> <condición>, <incremento/decreme nto> <instrucciones> <b>Fin_Desde</b>		Inicializa el valor del <i>contador</i> , verifica si la <i>condición</i> se cumple y en tal caso ejecuta las <i>instrucciones</i> , posteriormente <i>incrementa</i> o <i>decrementa</i> la variable <i>contador</i> .

Tabla 2.5 Estructuras repetitivas

En la unidad 4 estudiaremos con mayor detalle cada una de estas estructuras.



### Fuentes de consulta

- Böhm, C., & Jacopini, G. (1966). Flow diagrams, Turing machines, and languages only with two formation rules". *Communications of the ACM*, 9 (5), 366-371.
- Cairó, O. (2005). *Metodología de la programación: Algoritmos, diagramas de flujo y programas*. México, D.F.: Alfaomega.
- Joyanes, L., & Zohanero, I. (2005). *Programación en C. Metodología, algoritmos y estructuras de datos*. España: Mc Graw Hill.
- Reyes, A., & Cruz, D. (2009). *Notas de clase: Introducción a la programación*. México, D.F.: UACM.
- Viso, E., & Pelaez, C. (2007). *Introducción a las ciencias de la computación con Java*. México, D.F.: La prensas de ciencias, Facultad de Ciencias, UNAM.