(filtrar) datos. Al bucle que se utiliza para validar uno o más datos, también se le conoce como ***filtro***.

Repetitiva hacer...mientras en Diagramas de Flujo

En un Diagrama de Flujo, una instrucción repetitiva **hacer...mientras** se representa de la siguiente manera:

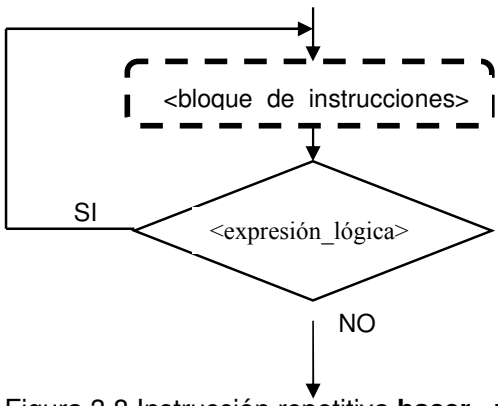


Figura 3.8 Instrucción repetitiva **hacer...mientras**.

Ejemplo 3.16
Se quiere diseñar el algoritmo que muestre por pantalla los primeros diez números naturales:

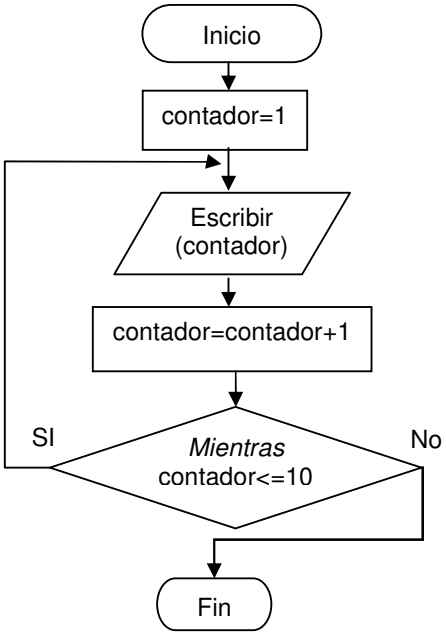


Figura 3.9. Imprime los 10 primeros números naturales

Para saber cuando hacer uso de un bucle u otro, es muy importante conocer bien las diferencias más significativas existentes entre ambos bucles

Mientras	hacer…mientras
1. se evalúa la condición	1. se ejecuta el bloque
2. se ejecuta el bloque	2. se evalúa la condición
Cero o más iteraciones	Una o más iteraciones

Figura 3.9. Diferencia entre mientras y hacer…mientras

Ejemplo 3.17.
Se quiere diseñar el diagrama de flujo que obtiene la sumatoria de números pares, del 0 ... 10

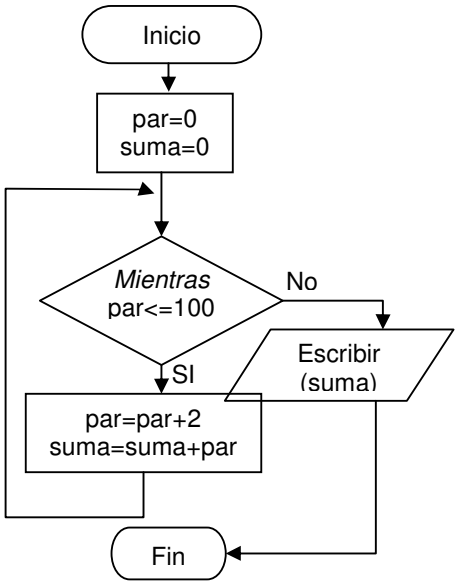


Figura 3.10**.** Suma los números pares de 0 a 100

Repetitiva hacer...mientras en Pseudocodigo

Sintaxis:

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

Como se puede apreciar, la instrucción repetitiva **hacer...mientras**, también hace uso de una condición.
En un bucle **hacer...mientras**, primero se ejecuta el bloque de instrucciones y, después, se evalúa la condición. En el caso de que ésta sea **verdadera**, se vuelve a ejecutar el bloque de instrucciones. Y así sucesivamente, hasta que, la condición sea **falsa**.

Por consiguiente, cuando el flujo de un algoritmo llega a un bucle **hacer...mientras**, existen dos posibilidades:

1. Se ejecuta el bloque de instrucciones y, después, si la condición se evalúa a **falsa**, el bloque de instrucciones no se vuelve a ejecutar, de manera que, el bucle **hacer...mientras** finaliza, habiendo realizado una sola iteración.
2. Se ejecuta el bloque de instrucciones y, a continuación, si la condición se evalúa a **verdadera**, el bloque de instrucciones se vuelve a ejecutar. Y así sucesivamente, hasta que la condición sea **falsa**.

El <bloque_de_instrucciones> de un bucle **hacer...mientras** puede ejecutarse una o más veces (iteraciones). También hay que prevenir que el bucle no sea infinito. En resumen, una **instrucción repetitiva hacer...mientras** permite ejecutar repetidamente (una o más veces) un bloque de instrucciones, mientras que, una determinada condición sea **verdadera**.

Ejemplo 3.18.

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

```
Nombre del algoritmo: Números_del_1_al_10
Variables: contador Tipo entero
 inicio
   contador ← 1   /* Inicialización del contador */
   hacer
     escribir ( contador )        /* Salida */
     contador ← contador + 1     /* Incremento */
   mientras ( contador <= 10 )     /* Condición */
 fin
```

La traza del algoritmo es:

Secuencia:	**Acción (instrucción):**	**Valor de:** contador
1	contador ← 1	**1**
	Inicio de la iteración 1.	
2	**escribir(** contador **)**	**1**
3	contador ← contador **+** 1	**2**
	Fin de la iteración 1.	
4	(Comprobar si contador **<=** 10)	**2**
	La condición es **verdadera**. Inicio de la iteración 2.	
5	**escribir(** contador **)**	**2**
6	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:

- En primer lugar, se le asigna el valor **1** a contador (acción 1).
- A continuación, se ejecuta el bloque de instrucciones del bucle **hacer... mientras**, mostrándose por pantalla el valor de contador (acción 2) y, después, se incrementa en **1** el valor de la variable contador (acción 3).
- Una vez ejecutado el bloque de instrucciones, se evalúa la condición de salida del bucle (contador **<=** 10) (acción 4) 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 se ejecuta diez veces (iteraciones).

Al igual que ocurre con la instrucción repetitiva **mientras**, cualquier pequeño descuido o error al escribir el código del algoritmo, puede dar lugar a que la instrucción repetitiva **hacer...mientras** no funcione correctamente.
Como ya se ha dicho, el bucle **hacer...mientras** puede iterar una o más veces, por tanto, cuando un bloque de instrucciones debe iterar al menos una vez, generalmente, es mejor utilizar un bucle **hacer...mientras** que un bucle **mientras**, como por ejemplo, en el siguiente.

Ejemplo 3.5.19.

Se quiere diseñar el algoritmo de un programa que:
1º) Pida por teclado un número (dato entero).
2º) Pregunte al usuario si desea introducir otro o no.
3º) Repita los pasos 1º y 2º, mientras que, el usuario no responda 'n' de (no).
4º) Muestre por pantalla la suma de los números introducidos por el usuario.

Solución:

```
Nombre del algoritmo: Suma_de_numeros_introducidos_por_el_usuario
Variables: acumulador,numero  Tipo entero
           seguir Tipo caracter
 inicio
  /* En acumulador se va a guardar la suma de los números introducidos por el
usuario.*/

  acumulador ← 0
  hacer
  escribir( "Introduzca un número entero: " )
  leer( número )
```

```
  acumulador ← acumulador + número
  escribir( "¿Desea introducir otro número (s/n)?: " )
  leer( seguir )
 mientras ( seguir <> 'n' )
 /* Mientras que el usuario desee introducir más números, el bucle iterará. */

 escribir ( "La suma de los números introducidos es: ", acumulador )
fin
```

Diferencia entre el bucle Mientras y el Bucle Hacer…Mientras

Variable Acumulador
Estructura (variable) que mantiene un total actualizado sin afectar otros cómputos o procesamientos.
Consiste de dos pasos:

- inicializar la variable
- incrementar el acumulador por un valor variable

En el siguiente problema, vamos a ver un ejemplo de variable contador.

Ejemplo 3.20.
Se quiere diseñar el algoritmo de un programa que:

1º) Pida por teclado un número (dato entero).
2º) Pregunte al usuario si desea introducir otro o no.
3º) Repita los pasos 1º y 2º, mientras que, el usuario no responda 'n' de (no).
4º) Muestre por pantalla la suma de los números introducidos por el usuario.

Solución:

```
Nombre del algoritmo: Suma_de_numeros_introducidos_por_el_usuario
Variables: acumulador, número  Tipo entero
           seguir Tipo caracter
inicio
  /* En acumulador se va a guardar la suma de los números introducidos por el
usuario.*/

  acumulador ← 0
  hacer
  escribir( "Introduzca un número entero: " )
  leer( número )c
  escribir( "¿Desea introducir otro número (s/n)?: " )
  leer( seguir )
 mientras ( seguir <> 'n' )
 /* Mientras que el usuario desee introducir más números, el bucle iterará. */

 escribir( "La suma de los números introducidos es: ",  acumulador )
fin
```

¿Cuándo usar un bucle u otro?
A la hora de elegir un bucle u otro, debemos hacernos la siguiente pregunta:

- **¿Se conoce, de antemano, el número de veces (iteraciones) que tiene que ejecutarse un determinado bloque de instrucciones?**

Si la respuesta es afirmativa, habitualmente se usa un bucle **Para**. En caso contrario, nos plantearemos la siguiente pregunta:

- **¿El bloque de instrucciones debe ejecutarse al menos una vez?**

En este caso, si la respuesta es afirmativa, generalmente haremos uso de un bucle **hacer...mientras**, y si la respuesta es negativa, usaremos un bucle **mientras**.

3.5.3 Ciclos Anidados

Al igual que las instrucciones alternativas, las instrucciones repetitivas también se pueden anidar, permitiendo las siguientes combinaciones de anidamiento:

- **mientras** en **mientras**
- **mientras** en **hacer...mientras**
- **mientras** en **Para**
- **hacer...mientras** en **hacer...mientras**
- **hacer...mientras** en **Para**
- **hacer...mientras** en **mientras**
- **Para** en **Para**
- **Para** en **mientras**
- **Para** en **hacer...mientras**

De ellas, vamos a estudiar, como ejemplo, la combinación:

- **Para** en **hacer...mientras**

Por otro lado, las instrucciones alternativas y repetitivas también se pueden anidar entre sí, permitiendo realizar 18 combinaciones más de anidamiento:

- **mientras** en doble
- **mientras** en simple
- **mientras** en múltiple

- **hacer...mientras** en doble
- **hacer...mientras** en simple
- **hacer...mientras** en múltiple

- **Para** en doble
- **Para** en simple
- **Para** en múltiple

- Doble en **mientras**
- Doble en **hacer...mientras**
- Doble en **Para**

- Simple en **mientras**
- Simple en **hacer...mientras**
- Simple en **Para**

- Múltiple en **mientras**
- Múltiple en **hacer...mientras**
- Múltiple en **Para**

De ellas, vamos a estudiar, como ejemplo, la combinación:

Simple en para

Bucle Para en Hacer… Mientras

En pseudocodigo, para anidar un bucle **Para** en un bucle **hacer...mientras**, se utiliza la sintaxis:

```
/* Inicio del anidamiento */
   para <variable> ← <valor_inicial> hasta <valor_final>
   [ incremento <valor_incremento> |
     decremento <valor_decremento> ] hacer
      <bloque_de_instrucciones>
   fin_para
   /* Fin del anidamiento */

  mientras ( <expresión_lógica> )
```

Ejemplo 3.21.
Se quiere diseñar el algoritmo que muestre por pantalla la tabla de multiplicar de un número entero introducido por el usuario. El proceso debe repetirse mientras que el usuario lo desee:

```
Nombre del algoritmo: Tabla_de_multiplicar_de_un_número
Variables: número,i  Tipo entero
                seguir Tipo caracter
 inicio
   hacer
      escribir( "Introduzca un número entero: " )
      leer( número )
      escribir( "La tabla de multiplicar del ", número, " es: " )
      /* Inicio del anidamiento */
      para i ← 1 hasta 10 hacer
        escribir( número, " * ", i, " = ", i * número )
      fin_para
      /* Fin del anidamiento */

      escribir( "¿Desea ver otra tabla (s/n)?: " )
      leer( seguir )
   mientras ( seguir <> 'n' )
 fin
```

Anidamiento de alternativa simple en bucle Para

En pseudocódigo, para anidar una alternativa simple en un bucle **Para**, se utiliza la sintaxis:

```
  para <variable> ← <valor_inicial> hasta <valor_final>
  [ incremento <valor_incremento> |
    decremento <valor_decremento> ] hacer

     /* Inicio del anidamiento */
     si ( <expresión_lógica> )
```

```
      <bloque_de_instrucciones>
    fin_si
    /* Fin del anidamiento */

  fin_para
```

Ejemplo 3.22.

Se quiere diseñar el algoritmo de un programa que muestre por pantalla todos los números enteros del 1 al 100 (ambos inclusive) que sean divisibles entre 17 ó 21:
Anidando una alternativa simple en un bucle **Para**, el problema se puede resolver con el siguiente algoritmo:

```
Nombre del algoritmo: algoritmo Números_enteros_divisibles_entre_17_o_21
Variables: número  Tipo entero
 inicio
   para número ← 1 hasta 100 hacer

     /* Inicio del anidamiento */
     si ( número mod 17 = 0 o número mod 21 = 0 )
       escribir( número )
     fin_si
     /* Fin del anidamiento */

   fin_para
 fin
```

E) Instrucciones de Control de Salto

En pseudocodigo, para escribir una instrucción de salto **interrumpir**, se utiliza la sintaxis:

Interrumpir
La instrucción de salto **interrumpir** siempre se usa para interrumpir (romper) la ejecución normal de un bucle, es decir, la instrucción **interrumpir** finaliza (termina) la ejecución de un bucle y, por tanto, el control del programa se transfiere (salta) a la primera instrucción después del bucle.

Ejemplo 3.23
Estúdiese el siguiente algoritmo:

```
Nombre del algoritmo: Números_opuestos_del_menos_10_al_mas_10
Variables: n, a  Tipo entero
 inicio
   a ← 0
   hacer
     escribir( "Introduzca un número entero: " )
     leer( n )
     si ( n = 0 )
       escribir( "ERROR: El cero no tiene opuesto." )
       interrumpir
       /* En el caso de que n sea un cero, el bucle se interrumpe. */
     fin_si
     escribir( "El opuesto es: ", -n )
     a ← a + n
```

```
   mientras ( n >= -10 y n <= 10 )
   escribir( "Suma: ", a )
fin
```

El algoritmo puede ser la solución del problema siguiente:

Diseñe el algoritmo:

1º) Pida por teclado un número (dato entero).
2º) Si el número introducido por el usuario es distinto de cero, muestre por pantalla el mensaje:

- "El opuesto es: *<-número>*".

3º) Repita los pasos 1º y 2º, mientras que, el usuario introduzca un número mayor o igual que -10 y menor o igual que 10.
Cero, si el usuario introduce un cero, el bucle también finaliza, mostrándose por pantalla el mensaje:

- "ERROR: El cero no tiene opuesto.".

4º) Muestre por pantalla la suma de los números introducidos por el usuario.

La traza del algoritmo es:

Secuencia:	**Acción (instrucción):**	**Valor de:**	
		a	n
1	a ← 0	**0**	**?**
	Inicio de la iteración 1.		
2	**escribir("**Introduzca un número entero:**")**	**0**	**?**
3	**leer(** n **)**	**0**	**15**
4	(Comprobar si n = 0)	**0**	**15**
	La condición de la alternativa simple es **falsa**.		
5	**escribir("**El opuesto es: **",** -n **)**	**0**	**-15**
6	a ← a **+** n	**15**	**15**
	Fin de la iteración 1.		
7	(Comprobar si n **>= -**10 **y** n **<=** 10)	**15**	**15**
	La condición del bucle es **falsa**. El bucle finaliza después de 1 iteración.		
8	**escribir("**Suma: **",** a **)**	**15**	**15**

El bucle ha finalizado porque la condición (n **>= -**10 **y** n **<=** 10) es **falsa**, ya que, **15** no es mayor o igual que **-10** y menor o igual que **10**.

Sin embargo, el bucle también puede finalizar, no porque sea **falsa** la condición (n >= -10 **y** n <= 10), sino, porque se ejecute la instrucción **interrumpir**. Esto ocurrirá cuando el usuario introduzca un cero. Por ejemplo:

Veamos otra traza:

Secuencia:	**Acción (instrucción):**	**Valor de:**	
		a	n
1	a ← 0	**0**	**?**
	Inicio de la iteración 1.		
2	**escribir(** "Introduzca un número entero:" **)**	**0**	**?**
3	**leer(** n **)**	**0**	**8**
4	(Comprobar si n = 0)	**0**	**8**
	La condición de la alternativa simple es **falsa**.		
5	**escribir(** "El opuesto es: ", -n **)**	**0**	**-8**
6	a ← a **+** n	**8**	**8**
	Fin de la iteración 1.		
7	(Comprobar si n >= -10 **y** n <= 10)	**8**	**8**
	La condición del bucle es **verdadera**. Inicio de la iteración 2.		
8	**escribir(** "Introduzca un número entero:" **)**	**8**	**8**
9	**leer(** n **)**	**8**	**-7**
10	(Comprobar si n = 0)	**8**	**-7**
	La condición de la alternativa simple es **falsa**.		
11	**escribir(** "El opuesto es: ", -n **)**	**8**	**7**
12	a ← a **+** n	**1**	**-7**
	Fin de la iteración 2.		
13	(Comprobar si n >= -10 **y** n <= 10)	**1**	**-7**
	La condición del bucle es **verdadera**. Inicio de la iteración 3.		
14	**escribir(** "Introduzca un número entero:" **)**	**1**	**-7**
15	**leer(** n **)**	**1**	**0**
16	(Comprobar si n = 0)	**1**	**0**
	La condición de la alternativa simple es **verdadera**.		

17	**escribir("**ERROR: El cero no tiene opuesto.**")**	**1**	**0**
18	**interrumpir**	**1**	**0**
	El bucle se interrumpe en la 3ª iteración. El control del programa se transfiere (salta) a la primera instrucción después del bucle.		
19	**escribir("**Suma: **",** a **)**	**1**	**0**

Normalmente, cuando en un bucle se utiliza una instrucción **interrumpir**, la ejecución de ésta se condiciona.
En el ejemplo 1, el bucle se interrumpe si la condición (n = 0) es **verdadera**. Nótese que, dicha condición no está contemplada en la condición de salida *estándar* del bucle, por lo que, a la condición (n = 0) se le considera ***condición de salida interna*** del bucle.

Ejemplo 3.23.
No obstante, el problema también se puede resolver sin hacer uso de la instrucción **interrumpir**:

```
Nombre del algoritmo: Números_opuestos_del_menos_10_al_mas_10
Variables: número, acumulador  Tipo entero
 inicio
   acumulador ← 0
   hacer
     escribir( "Introduzca un número entero: " )
     leer( numero )
     si ( numero = 0 )
       escribir( "ERROR: El cero no tiene opuesto." )
     sino
       escribir( "El opuesto es: ", -número )
       acumulador ← acumulador + número
     fin_si
   mientras ( número >= -10 y número <= 10 y número <> 0 )
   escribir( "Suma: ", acumulador )
 fin
```

Obsérvese que, en este algoritmo, sí se contempla en la condición de salida del bucle la posibilidad de que el usuario teclee un cero, en cuyo caso, el bucle deja de iterar de forma *natural*.
Los resultados por pantalla de este algoritmo son idénticos a los del algoritmo anterior.

Instrucción Continuar
En pseudocodigo, para escribir una instrucción de salto **continuar**, se utiliza la sintaxis:

La instrucción de salto **continuar** siempre se usa para *interrumpir* (romper) la ejecución normal de un bucle. Sin embargo, el control del programa no se transfiere a la primera instrucción después del bucle (como sí hace la instrucción **interrumpir**), es decir, el bucle no finaliza, sino que, finaliza la iteración en curso, transfiriéndose el control del programa a la condición de salida del bucle, para decidir si se debe realizar una nueva iteración o no.

Por tanto, la instrucción **continuar** finaliza (termina) la ejecución de una iteración de un bucle, pero, no la ejecución del bucle en sí. De forma que, la instrucción **continuar** salta (no ejecuta) las instrucciones que existan después de ella, en la iteración de un bucle.

Ejemplo 3.24.
En el algoritmo siguiente se muestra como se puede utilizar la instrucción **continuar**:

```
Nombre del algoritmo: Números_opuestos_del_menos_10_al_mas_10
Variables: n, a  Tipo entero
 inicio
   a ← 0
   hacer
     escribir( "Introduzca un número entero: " )
     leer( n )
     si ( n = 0 )
       escribir( "ERROR: El cero no tiene opuesto." )
       continuar
       /* En el caso de que n sea un cero,
          la iteración en curso del bucle
          se interrumpe aquí. */
     fin_si
     escribir( "El opuesto es: ", -n )
     a ← a + n
   mientras ( n >= -10 y n <= 10 )
   escribir( "Suma: ", a )
 fin
```

Este corresponde a:

1º) Pida por teclado un número (dato entero).
2º) Si el número introducido por el usuario es distinto de cero, muestre por pantalla el mensaje:

- "El opuesto es: *<-número>*".

En caso contrario, muestre el mensaje:

- "ERROR: El cero no tiene opuesto.".

3º) Repita los pasos 1º y 2º, mientras que, el usuario introduzca un número mayor o igual que -10 y menor o igual que 10.
4º) Muestre por pantalla la suma de los números introducidos por el usuario.

La traza del algoritmo es:

Secuencia:	**Acción (instrucción):**	**Valor de:**	
		a	n
1	a ← 0	**0**	**?**
	Inicio de la iteración 1.		
2	**escribir("**Introduzca un número entero:**")**	**0**	**?**

3	**leer(** n **)**	**0**	**2**
4	(Comprobar si n = 0)	**0**	**2**
	La condición de la alternativa simple es **falsa**.		
5	**escribir("**El opuesto es: **",** -n **)**	**0**	**-2**
6	a ← a **+** n	**2**	**2**
	Fin de la iteración 1.		
7	(Comprobar si n >= -10 **y** n <= 10)	**2**	**2**
	La condición del bucle es **verdadera**. Inicio de la iteración 2.		
8	**escribir("**Introduzca un número entero:**")**	**2**	**2**
9	**leer(** n **)**	**2**	**0**
10	(Comprobar si n = 0)	**2**	**0**
	La condición de la alternativa simple es **verdadera**.		
11	**escribir("**ERROR: El cero no tiene opuesto.**")**	**2**	**0**
12	**continuar**	**2**	**0**
	La 2ª iteración se interrumpe (finaliza) aquí. El control del programa se transfiere (salta) a la condición de salida del bucle.		
13	(Comprobar si n >= -10 **y** n <= 10)	**2**	**0**
	La condición del bucle es **verdadera**. Inicio de la iteración 3.		
14	**escribir("**Introduzca un número entero:**")**	**2**	**0**
15	**leer(** n **)**	**2**	**-59**
16	(Comprobar si n = 0)	**2**	**-59**
	La condición de la alternativa simple es **falsa**.		
17	**escribir("**El opuesto es: **",** -n **)**	**2**	**59**
18	a ← a **+** n	**-57**	**-59**
	Fin de la iteración 3.		
19	(Comprobar si n >= -10 **y** n <= 10)	**-57**	**-59**
	La condición del bucle es **falsa**. El bucle finaliza después de 3 iteración.		
20	**escribir("**Suma: **",** a **)**	**-57**	**-59**

La instrucción **continuar** se ejecuta cuando el usuario introduce un cero, interrumpiendo la iteración en curso; pero, el bucle solamente finaliza cuando la condición (n >= -10 **y** n <= 10) sea **falsa**.

Normalmente, al igual que ocurre con la instrucción **interrumpir**, cuando en un bucle se utiliza una instrucción **continuar**, la ejecución de ésta también se condiciona. En el ejemplo 3, la iteración en curso del bucle se interrumpe si es **verdadera** la condición (n = 0).

Ejemplo 3.25.
No obstante, el problema también se puede resolver sin hacer uso de la instrucción **continuar**:

```
Nombre del algoritmo: Números_opuestos_del_menos_10_al_mas_10
Variables: número, acumulador  Tipo entero
 inicio
   acumulador ← 0
   hacer
     escribir( "Introduzca un número entero: " )
     leer( número )
     si ( número = 0 )
       escribir( "ERROR: El cero no tiene opuesto." )
     sino
       escribir( "El opuesto es: ", -número )
       acumulador ← acumulador + número
     fin_si
   mientras ( número >= -10 y número <= 10 )
   escribir( "Suma: ", acumulador )
 fin
```

Unidad 4. Arreglos y Cadenas

Esta unidad tiene como objetivo analizar y aplicar las operaciones sobre arreglos y cadenas para la solución de problemas.

Definición de Arreglo:

1. Un arreglo es un conjunto finito e indexado de elementos homogéneos, que se referencian por un identificador común (nombre). La propiedad *indexado* significa que el elemento primero, segundo, hasta el n-ésimo de un arreglo pueden ser identificados por su posición ordinal.

2. Un arreglo es una colección finita, homogénea y ordenada de elementos del mismo tipo.

De manera **formal** se define un arreglo de tamaño n de los elementos de tipo A, es un elemento del espacio n-dimensional del conjunto A, es decir, X es arreglo de tamaño n del tipo A si y solo si $X \in A^n$

Ambas definiciones reconocen los siguientes conceptos:

Finita: Todo arreglo tiene un límite, es decir, debe determinarse cual será el número máximo de elementos que podrán formar parte del arreglo.

Homogénea: Todos los elementos de un arreglo son del mismo tipo o naturaleza (todos enteros, todos booleanos, etc,), pero nunca una combinación

de distintos tipos.

Ordenada: Se debe determinar cuál es el primer elemento, el segundo, el tercero..... y el n-ésimo elemento.

Características de los arreglos.

- Tienen un único nombre de variable, que representa todos los elementos.
- Contienen un índice o subíndice, los cuales diferencian a cada elemento del arreglo.
- Se pueden realizar ciertas operaciones como son: recorridos, ordenaciones y búsquedas de elementos.
- El número total de elementos del arreglo (NTE) es igual al límite superior (LS), menos límite inferior NTE = LS - LI + 1
- El tipo de índice puede ser cualquier tipo ordinal (carácter, entero, enumerado)
- El tipo de los componentes puede ser cualquiera (entero, real, cadena de caracteres, registro, etc.)
- Se utilizan [ ] para indicar el índice de un arreglo. Entre los [ ] se debe escribir un valor ordinal (puede ser una variable, una constante o una expresión que dé como resultado un valor ordinal).
- Por tanto, si un arreglo tiene *n* componentes, la última localidad está dada por *n*, como se muestra en la siguiente figura 4.1

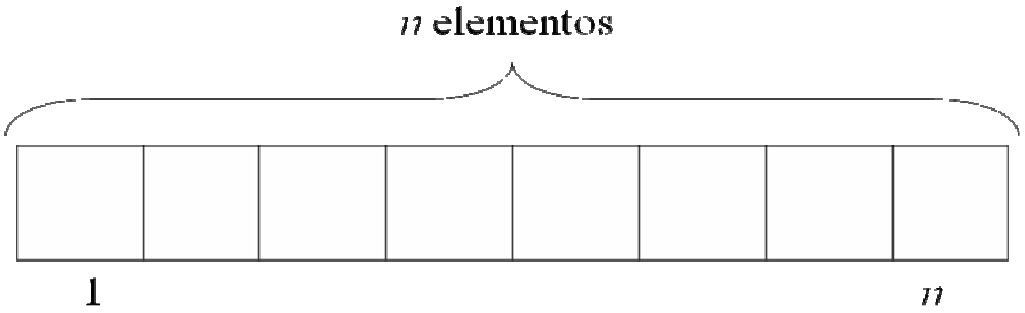


Figura 4.1 . Representación de un Arreglo

En la Figura 4.2 es un arreglo llamado EDADES donde se almacenan las edades de un grupo de alumnos de la clase de Natación de la BUAP.

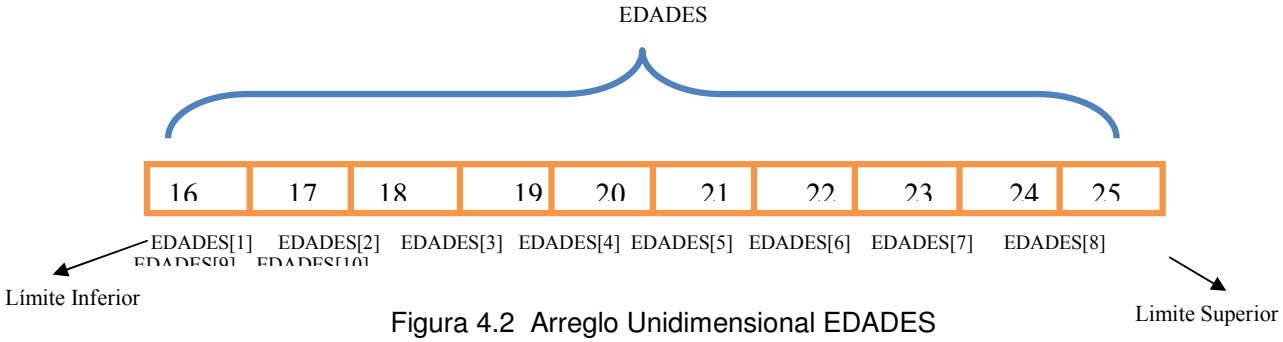


Figura 4.2 Arreglo Unidimensional EDADES

Identifiquemos lo siguiente:

- EDADES es el nombre del arreglo
- EDADES[1] es el primer elemento del arreglo EDADES que hace referencia al elemento de la posición 1, cuya edad almacenada es 16.
- EDADES[10] es el último elemento del arreglo EDADES que hace referencia al elemento de la posición 10, cuya edad almacenada es 25.
- 1, 2, 3, ... n, son los índices o subíndices del arreglo (pueden ser enteros, no negativos, variables o expresiones enteras).
- NTE= LS - LI + 1= 10-1+1=10 elementos.

Clasificación de los arreglos.

Los arreglos se clasifican en:

- Unidimensionales (Vectores): un sólo índice
- Bidimensionales (Tablas o Matrices): dos índices
- Multidimensionales: más de dos índices

4.1 Arreglos unidimensionales

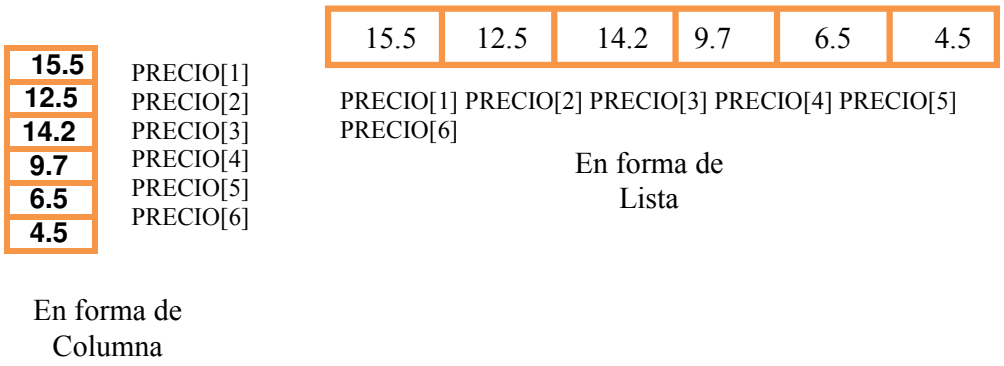
Un arreglo de una dimensión o vector es un conjunto finito y ordenado de elementos homogéneos.

Se pueden representar como una lista o columna de datos del mismo tipo, a los que colectivamente nos referimos mediante un nombre.

Los arreglos unidimensionales deben cumplir lo siguiente:

- Compuesto por un número de elementos finito.
- Tamaño fijo: el tamaño del arreglo debe ser conocido en tiempo de compilación.
- Homogéneo: todos los elementos son del mismo tipo.
- Son almacenados en posiciones contiguas de memoria, cada uno de los cuales se les puede acceder directamente.
- Cada elemento se puede procesar como si fuese una variable simple ocupando una posición de memoria.

Ejemplo 4.1 : Dado un arreglo unidimensional denominado PRECIO cada uno de sus elementos se designará por ese mismo nombre diferenciándose únicamente por su correspondiente subíndice.



Los elementos del arreglo unidimensional PRECIO, se representan con la siguiente notación: PRECIO[1], PRECIO[2], PRECIO[3], ..., PRECIO[N].

Es decir, todos los elementos pertenecen al mismo arreglo unidimensional llamado PRECIO y se referencian a través del subíndice que indica la posición que ocupan en el arreglo. Dichos elementos se manipulan como variables individuales. De esta manera se le puede asignar un valor a un elemento, por ejemplo PRECIO [1]=15.5

En un arreglo unidimensional o vector se distinguen tres elementos:

a) Nombre del arreglo
b) Longitud o rango del arreglo
c) Subíndices del arreglo

A continuación se muestra el arreglo unidimensional Figura 4.3 identificando los tres elementos.

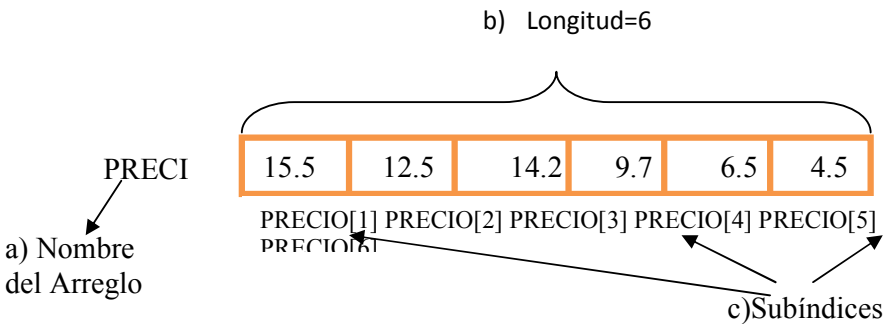


Figura 4.1 Elementos del arreglo unidimensional

Definición de Arreglos

Un arreglo debe ser definido con un nombre, la longitud (tamaño del arreglo) y su tipo de datos. Para la definición de un arreglo utilizaremos la siguiente notación:

Identificador_Arreglo= ARREGLO [Lim_inf..Lim_sup] DE tipo_dato

Donde

Identificador_Arreglo: Es el nombre del arreglo para reconocer a los elementos.
Lim_inf: es el primer elemento del arreglo
Lim_sup:es el último elemento del arreglo, para definir el número de elementos que tendrá el arreglo
Tipo_dato: se declara el tipo de datos para todos los elementos del arreglo.

Ejemplos definiciones del arreglo:

Sea un arreglo A de 5 elementos de tipo entero
A=ARREGLO [1..5] DE enteros

Sea un arreglo NÚMEROS de 4 elementos de tipo real

NÚMEROS=ARREGLO[1..4] DE reales

Sea un arreglo LETRAS de 3 elementos de tipo caracteres

LETRAS=ARREGLO[1..3] DE caracteres

Para referirse a cada elemento de un arreglo se utiliza su índice. El índice es asignado generalmente por la posición que ocupa el elemento en el arreglo.

Para el arreglo A [3, -6, 1, 24, 8] y referirse al elemento uno del arreglo A se especifica la posición A[1]) siendo 3.

El índice puede ser obtenido mediante cualquier expresión numérica (enteros, variables o expresiones enteras) que de cómo resultado un número entero. Cuando se usa una expresión para obtener un índice se debe garantizar que el resultado de la expresión esté entre los límites permitidos de los índices para el arreglo.

Ejemplos.

Si se tiene el arreglo NÚMEROS[4.2, 0.0, -33.76, 19.7e-2], la variable i tiene el valor de uno (1) primera posición y la variable j tiene el valor de cuatro (4) ultima posición, entonces:

- El elemento [i+j/2] del arreglo es -33.76, ya que al evaluar la expresión da como resultado la posición 3.
- El elemento (j-3*i) del arreglo es 4.2, ya que al evaluar la expresión da como resultado la posición 1.
- El elemento (2*i-j) del arreglo no existe, ya que al evaluar la expresión da como resultado –2 y esa posición no existe.

Operaciones con Arreglos Unidimensionales.

Las operaciones más comunes que se pueden realizar con los arreglos son:

- Asignación
- Lectura
- Escritura
- Recorrido
- Actualización (insertar, borrar, modificar)
- Ordenación
- Búsqueda

Como los arreglos son datos estructurados, muchas de estas operaciones se deben aplicar sobre cada elemento y no de manera global.

- **Asignación**

En general no es posible asignar directa un valor a todo el arreglo, sino que se debe asignar el valor deseado a cada elemento. La manera de asignar (insertar) un valor en cada elemento del arreglo unidimensional es mediante el subíndice que indica la posición, se puede utilizar la siguiente forma:

<NombreVector>[subíndice]← <Valor>

Ejemplos:

- A es un arreglo de números enteros y se le asigna en la posición 1 el elemento 10, A[1]← 10

- PAIS es un arreglo de caracteres y se le asigna en la posición 2, el elemento 'Francia'. PAIS[2]← 'Francia'

- NÚMEROS es un arreglo de números reales y se le asigna en la posición 5 el elemento 2.5. NÚMEROS[5]← 2.5

- PRECIO es un arreglo de números reales y se le asigna en la posición 3, el elemento de la posición 2 +10.5, PRECIO[3]← PRECIO[2]+10.5

No es posible asignar directamente un valor a todo el arreglo, por lo que se realiza utilizando alguna estructura de repetición para recorrer cada posición del arreglo como son: Para-Hasta-Hacer, Mientras-Hacer, Repite-Hasta, como se muestra a continuación:

```
Para i desde 1 Hasta N hacer

        ARREGLO[i]←valor
Fin_Para
```

- **Lectura**

Este proceso de lectura de un arreglo consiste en leer un valor de cada elemento del arreglo y asignarlo a una variable. La lectura se realiza de la siguiente manera:

```
Para i desde 1 Hasta N hacer
        x←ARREGLO[i]
 Fin_Para
```

Ejemplo:

Leer los datos del arreglo A de longitud 20 y mostrarlos por pantalla, uno a uno.

```
Escribir('Los valores del arreglo son : ')
Para i desde 1 Hasta N hacer
                x←ARREGLO[i]
                Escribir(X)
Fin_Para
```

También se puede leer de manera directa y escribir el elemento del arreglo

```
Escribir('Los valores del arreglo son : ')
Para i desde 1 Hasta N hacer
                Escribir(ARREGLO[i])
Fin_Para
```

- **Escritura**

Consiste en asignarle un valor a cada elemento del arreglo.

```
Para i desde 1 Hasta N hacer

        ARREGLO[i]←x
Fin_Para
```

- **Recorrido o Acceso Secuencial**

Se accede a los elementos de un arreglo para introducir datos en él (asignar valores ó leer) y para ver su contenido (escribir). A la operación de efectuar una acción general sobre todos los elementos de un vector se le denomina recorrido del vector.

Estas operaciones se realizan utilizado estructuras repetitivas, cuyas variables de control (por ejemplo, i) se utilizan como subíndices del vector (por ejemplo, A(i)). El incremento de la variable de control produce el tratamiento sucesivo e individual de los elementos del vector.

Ejemplo: Cálculo de la suma y promedio de los primeros 10 elementos enteros de un vector Z

```
Inicio
Suma ← 10
        Para i ← 1 Hasta 10 Hacer
                Leer(Z[i])
        Fin_para
        Para i ← 1 Hasta 10 Hacer
                Suma ← Suma + Z[i]
        Fin_para
        Promedio ← Suma/10
        Escribir(Suma, Promedio)
Fin
```

- **Actualización**

Dentro de esta operación se encuentran las operaciones de **eliminar, insertar y modificar datos**. Para realizar este tipo de operaciones se debe tomar en cuenta si el arreglo está o no ordenado.

Para arreglos ordenados los algoritmos de inserción, borrado y modificación son los siguientes:

a) Insertar.

- Para insertar un elemento X en un arreglo A que encuentra ordenado, debe verificarse que exista espacio.
- Luego tendrá que encontrase la posición en la que debería estar el nuevo valor para no alterar el orden del arreglo.
- Una vez detectada la posición, se procederá a recorrer todos los elementos desde la misma hasta N-ésima posición, un lugar a la derecha
- Finalmente se asignará el valor de X, en la posición encontrada.

```
Algoritmo
        Inicio
                Si  n <  Máximo_elementos
                        i← 1
                        Mientras (i ≤N) y (A[i] < X)
                                i← i + 1
                        Fin_mientras
                        Si (i >N) o (A[i] > X)  entonces
                                Pos← −1
                        Sino
                                Pos← 1
                        Fin_Si
```

```
                Si Pos > 0 entonces
                        Escribir ('El elemento ya existe')
                Sino
                    N← N + 1
                    Pos← Pos * (−1)
                    i← N
                    Repetir desde N hasta  Pos+1
                            A[i] ← A[i − 1]
                            I← i − 1
                    Fin_Repetir
                    A[Pos] ←X
                Fin_si
            Sino
                Escribir ('No hay espacio en el arreglo')
            Fin_si
    Fin
```

b) Eliminar.

- Para eliminar un elemento X de un arreglo ordenado A, debe verificarse que el arreglo no este vacío, sino no se puede realizar esta operación.
- Si se cumple esta condición, entonces tendrá que buscarse la posición del elemento a eliminar
- Buscando el elemento, se recorre el arreglo hasta encontrar el elemento para eliminar

Algoritmo

```
    Inicio
    Si N>0 entonces
        Leer X
        i←1
            Mientras (i<=N) y  A[i] < X )
                    i←i+1
            Fin_mientras
            Si  (arreglo[i] > X) o ( i>N) entonces
             Pos ← −1
            Sino
                 Pos ←1
            Fin_si
            Si Pos <= 0 entonces
                    Escribir ('El elemento no existe')
            Sino
                N← N − 1
                i← Pos
                Repetir  desde Pos hasta  N
                        A[i] ← A[i + 1]
                        I← i + 1
                Fin_Repetir
              Fin_si
```

```
            Sino
                  Escribir ('El arreglo esta vacio')
            Fin_si
      Fin
```

c) Modificar.

- El procedimiento es muy similar a la eliminación con la variante de que el valor X se modificara por el valor Y, teniendo en cuenta que el arreglo ordenado no se altere.
- Si esto sucede se debe reordenar el arreglo o bien primero se elimina el elemento y después se inserta

```
Algoritmo
      Inicio
            Si N>=1 entonces
                  i←1
                  encontrado←falso
                  Mientras i<N y encontrado = falso hacer
                        Si arreglo[i]=valor entonces
                              arreglo[i]←valor _ nuevo
                              encontrado←verdadero
                        sino
                              i←i+1
                        Fin_si
                  Fin_Mientras
            Fin_si
      Fin
```

4.2 Aplicaciones sobre arreglos unidimensionales

4.2.1 Búsquedas en Arreglos

Una búsqueda es el proceso mediante el cual podemos localizar un elemento con un valor específico dentro de un conjunto de datos. Terminamos con éxito la búsqueda cuando el elemento es encontrado.

A continuación veremos el algoritmo de la búsqueda más simple.

- **Búsqueda Secuencial**

A este método también se le conoce como búsqueda lineal y consiste en empezar al inicio del conjunto de elementos , e ir a través de ellos hasta encontrar el elemento indicado ó hasta llegar al final de arreglo.

Este es el método de búsqueda más lento, pero si nuestro arreglo se encuentra completamente desordenado es el único que nos podrá ayudar a encontrar el dato que buscamos.

Algoritmo de Búsqueda Secuencial

```
Para i desde 1  hasta N hacer
        Si(a[i]= b) entonces
                Escribir 'Valor encontrado'
        Sino
        Escribir 'Valor no encontrado'
        Fin_si
Fin_Para
```

Ejemplo 4.2**:** Desarrollar un algoritmo para buscar el elemento "Manzana" dentro del arreglo FRUTAS. Imprimir si se encuentra o no y dado el caso su posición.

```
Inicio
Encontrado = Falso
// Lectura del valor a buscar
        Leer(fruta)
        //Búsqueda del valor en el arreglo
        Para i ← 1 Hasta 10 Hacer
        Si FRUTAS[i] = fruta entonces
                Encontrado ← Verdadero
                Posición ←  i
        Fin_si
        Fin_para
        // Notificación de los resultados de la búsqueda
        Si Encontrado = Verdadero Entonces
        Escribir('Fruta encontrada en la posición: ', posicion)
        Sino
                Escribir('Fruta no encontrada')
        Fin_si
Fin
```

Ejemplo 4.3. Desarrollar un algoritmo que busque el elemento mayor de un arreglo A de tamaño n de números reales.

```
Inicio
// Asignamos el primer elemento del arreglo A para iniciar la búsqueda y
 encontrar el elemento mayor
      x ←A[1]
        i ←1
        Mientras (i<n ) hacer
                si (x < A[i]) entonces
                      x ←A[i]
                 Fin_si
                i ←i + 1
        Fin_mientras
        Escribir ('El elemento mayor buscado es:')
        Escribir (A[i])
Fin
```

4.2.2 Ordenamiento en Arreglos.

El ordenamiento de un arreglo es la organización de los valores de los elementos del arreglo de acuerdo a un criterio específico.

Esta operación es importante en los arreglos ya que permite que sus elementos estén ordenados de forma ascendente o descendente para poder llevar otra operación como la búsqueda de manera más fácil.

Vamos a revisar dos métodos de ordenación lo más sencillos.

1. Ordenación por Burbuja
2. Ordenación de Selección

Ordenación por Burbuja

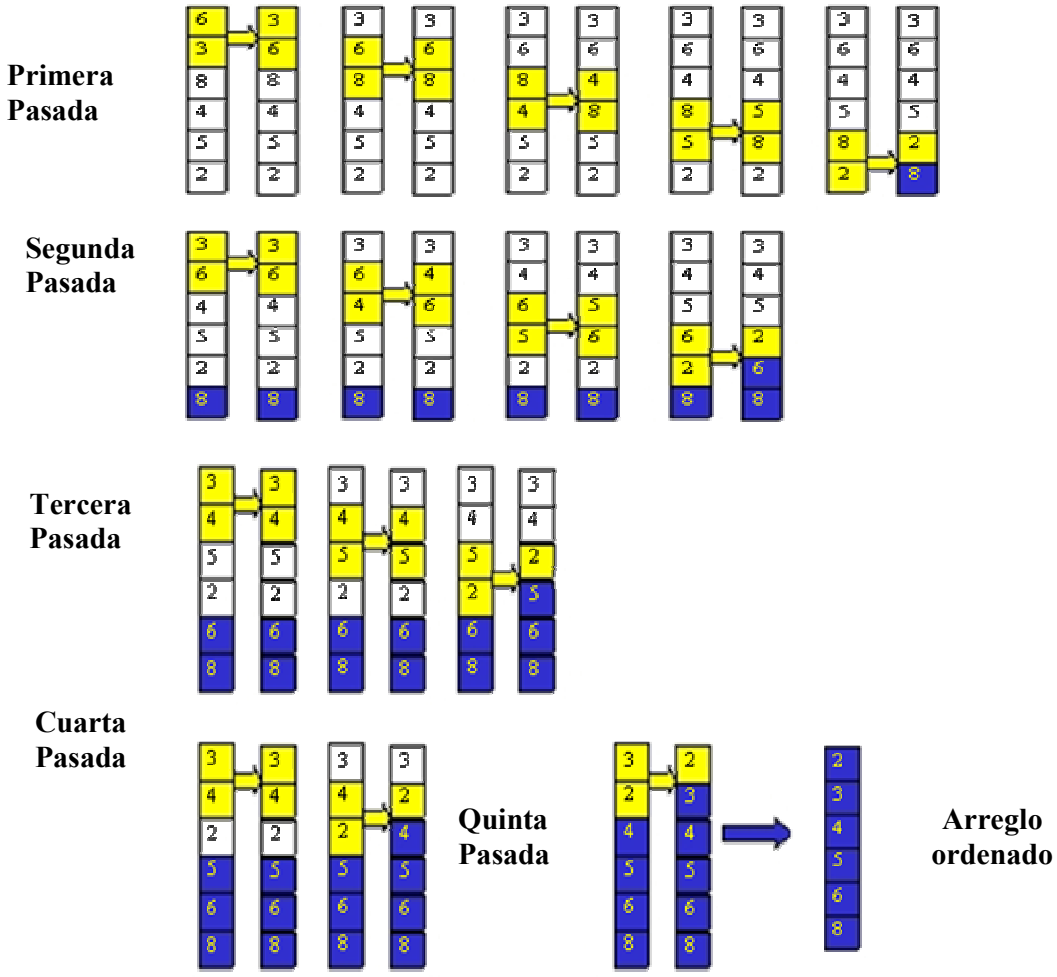
Es el método de ordenación más utilizado por su fácil comprensión y programación, pero es importante señalar que es el más ineficiente de todos los métodos. Este método consiste en llevar los elementos menores a la izquierda del arreglo ó los mayores a la derecha del mismo según el tipo de ordenación. La idea básica del algoritmo es comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que todos se encuentren ordenados.

Algoritmo de Burbuja

```
Inicio
        i ←1
        Mientras i < N hacer
                Para   j ← i+1 hasta j< N  hacer
                        Si arreglo[i] > arreglo[j] entonces
                           aux← arreglo[i]
                           arreglo[i]) ← arreglo[j])
                           arreglo[j]) ←aux
                        Fin_si
                Fin_Para
           i ← i +1
         Fin_mientras
Fin
```

Ejemplo 4.3**:**

Realizaremos la ordenación de los siguientes elementos: 6,3,8,4,5,2 utilizando el Algoritmo de Burbuja



Ordenamiento por Selección

Este método consiste en encontrar el elemento más pequeño de la lista y colocarlo en la primera posición, luego se encuentra el siguiente elemento más pequeño y se lleva a la segunda posición. Se continúa el proceso hasta llegar a la posición penúltima del vector.

Para obtener el elemento más pequeño se compara el primer valor con cada uno de los restantes y cada vez que se encuentra un valor más pequeño se intercambian los valores.

Ejemplo 4.4

```
 Inicio
// Encontrar la posición del menor valor, se utilizan dos índices: J para el elemento fijo y  K para recorrer los elementos que se comparan.

Inicio
Para J ← 1 Hasta 4 Hacer
        Para K ← J + 1 Hasta 5 Hacer
```

```
            Si Num[K] < Num[PosMenor] Entonces
            // Intercambiar valores en el vector
                  aux  Num[J]
                  Num[J]  Num[k]
                  Num[k]  aux
            Fin_si
      Fin_ Para
Fin_Para
Fin
```

4.3 Manejo de cadenas

Una cadena es un conjunto de caracteres incluido el espacio en blanco, por ejemplo tenemos: “hola”, “123vb”, “v bg%.”. Generalmente una cadena va encerrada entre comillas (“ ”).

La longitud de una cadena es el número de caracteres que contiene. Una cadena vacía es la que no tiene ningún carácter y se representa como “”. Una constante de tipo cadena es un conjunto de caracteres válidos encerrados entre comillas, por ejemplo “numero1”, “&/ #$%”. Una variable de cadena es aquella cuyo contenido es una cadena de caracteres, por ejemplo cadena=”una cadena.”, str=”-23.56”. El último carácter de la cadena marca el fin de la cadena, en este caso utilizaremos el carácter ‘\0’ para denotar fin de cadena.

Operaciones básicas con cadenas:

- Asignación.
  Apellido←”Juarez”
- Entrada/ Salida
  Leer(nombre, estado_civil)
  Escribir(nombre, apellido)
- Para el cálculo de la longitud de una cadena se da el número de caracteres que hay en una cadena que está entre comillas, incluyendo los espacios en blanco.
- Para la comparación de cadenas se comparan caracteres o cadenas de caracteres para ver si son iguales o no. Según el código ASCII hay un orden de caracteres, así "A" es menor que "C", debido a que el valor de “A” es 65 y el de “C” es 67, además de que las letras mayúsculas son distintas de las minúsculas y de la “A” a la “Z” está antes que la “a”, ya que el valor de “a” es 97 y de la “A” es 65 en como ya se había mencionado, y se termina hasta la “Z” con el valor de 90.
- La concatenación se define como la unión de varias cadenas de caracteres en una sola, conservando el orden de los caracteres de cada una de ellas.
  Cuando se combinan dos cadenas con el operador de concatenación, la segunda cadena se agregará directamente al final de la primera. En códigos postales y números telefónicos se suele usar caracteres ya que no se necesita operar los números y así podemos usar el guión.
- La extracción de subcadenas es una subcadena que es una porción de la cadena original.
- La búsqueda de información, consiste en buscar una subcadena o cadena dentro de otra mayor. Nos devuelve el número de la posición donde inicia la cadena buscada, -1 si no la encuentra.
- Encontrar el punto medio, este nos devuelve la mitad de la posición de la cadena.
- Truncar cadenas, se pretende quedarse con los primeros *n* caracteres de la cadena, eliminando el resto de los caracteres.

- Convertir cadenas a números o viceversa, esto funciona, cuando los caracteres son dígitos, y lo que se pretende es que esa cadena de dígitos se convierta en su respectivo valor numérico.
- Insertar una cadena dentro de otra.
- Borrar cadenas.
- Sustituir una cadena por otra.
- Invertir el orden de una cadena.

Ejemplo 4.5.
El siguiente algoritmo sustituye las e por *.

```
Inicio
        Escribir ("escriba una palabra")
        Leer (str)
        Para i=1 hasta len(str) hacer
                Si str[i] = `e´ entonces
                   str[i] = `*´
                Fin_si
        Fin_desde
        Escribir (str)
Fin
```

Ejemplo 4.6.
El siguiente algoritmo imprime una cadena de manera invertida

```
Inicio
        Escribir ("escriba una palabra")
        Leer (str)
        Para i=len(str) hasta 1, con decrementos hacer
                Escribir (str[i])
        Fin_para
Fin
```

Ejemplo 4.7.
El siguiente algoritmo realiza lo siguiente, dada una cadena en minúsculas, la convierte en mayúsculas

```
Inicio
        Escribir ("escriba una palabra")
        Leer (str)
        Para i=1 hasta len(str) hacer
                Si 'a'<=str[i]<='z' entonces
                    Valor(str[i])← Valor(str[i])+32
                Fin_si
        Fin_para
Fin
```

Ejemplo 4.8.
El siguiente algoritmo verifica si la cadena que se introdujo es una cadena de '1' y '0', esto es un número binario.

```
Inicio
        Escribir ("escriba la cadena")
        Leer (str)
        es_bin←true
```

```
        Para i=1 hasta len(str) hacer
                Si str[i]≠'1' o str[i] ≠'0' entonces
                    es_bin← false
                Fin_si
        Fin_para
        Si es_bin=true entonces
            Escribir ("es un número binario")
        Sino
            Escribir ("no es un número binario")
        Fin_si
Fin
```

Actividades:

4.1 Obtenga la longitud de una cadena

4.2 Concatene dos cadenas

4.3 Compare dos cadenas y diga cual cadena esta antes que la otra.

4.4 Dadas dos cadenas, diga si la primera cadena es subcadena de la otra.

4.5 Dada una cadena y un número *n*, el cual debe ser menor a la longitud de la cadena, trunque la cadena, devolviendo los primeros *n* caracteres.

4.6 Dada una cadena y un carácter, diga cuantas veces se repite dicho carácter se repite dentro de la cadena.

4.7 Dada una cadena y dos caracteres, sustituya cada instancia en la cadena del primer carácter por el segundo.

4.8 Dada una cadena, separarla por palabras, esto es tomando en consideración el espacio, imprimir en pantalla cada palabra en renglones distintos, por ejemplo sea cad="hola a todo el mundo", en pantalla se presentará:

"hola"
"a"
"todo"
"el"
"mundo"

4.9 Dada una cadena comprobar si es un palíndromo.

4.3 Arreglos bidimensionales

Unidad 5. Diseño Modular

5.1 Introducción

Para comprender la programación modular, es importante describir la siguiente analogía:

Suponga que la señora Juanita prepara memelas en un lugar cercano al zócalo de la Ciudad. Ella sola realiza el conjunto de tareas que implican preparar memelas, es decir, pica cebollas, muele la salsa en la licuadora, fríe las tortillas, vierte salsa sobre las tortillas, revisa que estén en su punto, además cobra y devuelve cambio. Todas estas actividades las realiza una sola persona.

Si hablamos de programación, los programas que sólo tienen la función principal, son como la señora Juanita, que realizan un conjunto de tareas secuencialmente, lo cual hace más complicada su legibilidad y mantenimiento.

Regresando a la analogía, la señora ahora contrata un cierto número de personas para ayudarle en el proceso de elaboración de memelas. A cada señora le es encomendada una tarea o función. Cuando alguna de las señoras contratadas termina su tarea, ésta informa a la señora Juanita que ha terminado y le devuelve el resultado de su tarea. Cuando todas las señoras han terminado su tarea, el problema complejo de elaborar memelas ha concluido. El problema fue dividido entre varias señoras, para su realización, éstas fueron coordinadas por la señora Juanita que indica el orden de realización de las tareas.

En la programación modular, el problema complejo es dividido en subproblemas más pequeños o más fáciles de resolver. Cada problema se resuelve por una función o modulo. Una vez que cada modulo resuelve su tarea, el problema complejo ha sido resuelto en su totalidad [2].

Si algún subproblema, aún es demasiado complejo, éste es dividido a su vez, en subproblemas, y así sucesivamente. Observemos el siguiente diagrama para ilustrar lo anterior.

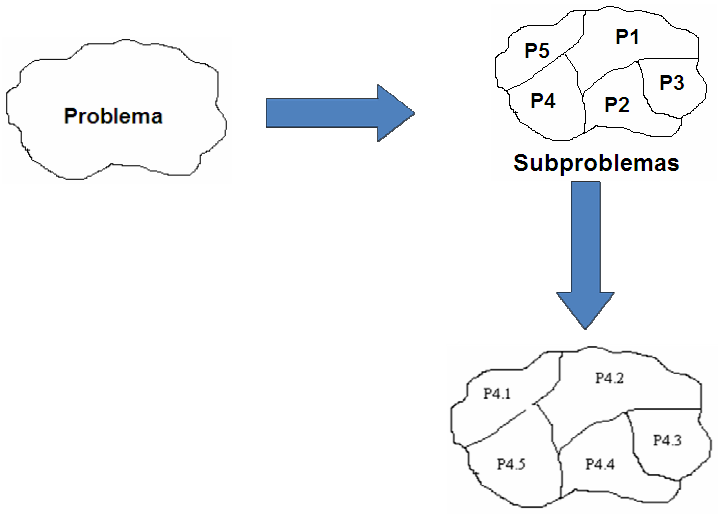


Figura 5.1 División de problemas en subproblemas

5.2 Conceptos de módulo

a) Es un conjunto de instrucciones o sentencias que resuelven una tarea especifica, al cual se hace referencia por medio de un nombre. Éste puede invocarse varias veces en un programa.
b) Es un proceso con cierto grado de independencias del modulo principal (programa principal).
c) Es un subprograma que resuelve una parte del problema.
d) El concepto de modulo tiene varias acepciones, dependiendo del lenguaje de programación: subrutinas (Basic y Fortran), procedimientos y funciones (Pascal y Delphi, Ada, etc), funciones o métodos (C++, Java y C#)

Declaración de un módulo

Los módulos tienen un formato bien definido para su utilización:

- Tiene un tipo de retorno

- Tiene un nombre o identificador.
- Puede tener o no lista de parámetros (elementos de comunicación con el modulo principal)
- Tiene un cuerpo o conjunto de instrucciones que resuelven una tarea especifica.
- Pueden o no devolver un valor del algún tipo de dato básico, estructurado o definido por el usuario.

Esquemáticamente podemos observar el formato de un módulo como el siguiente:

[Tipo] Nombre del modulo (tipo $param_1$, tipo $param_2$,… tipo $param_n$)
Inicio
Sentencia 1;
Sentencia 2;
Sentencia n;
[Devolver (expresión)]
FinModulo

Esquema 5.2 Sintaxis de la declaración de un módulo en pseudocódigo [2]

Los elementos del formato de un módulo que se encuentran entre corchetes indican que son opcionales, es decir, podemos prescindir de ellos. La lista de parámetros aparece después de nombrar al módulo y siempre se encuentra delimitada por los paréntesis, el número de parámetros van desde 0 hasta n.

Estos parámetros se les denominan **parámetros formales**, mientras que cuando se realiza el llamado del módulo desde cualquier otro, a estos parámetros se les llama **actuales** y se enlazan con cada uno de los formales para llevar a cabo la ejecución.

En particular un módulo puede ser de dos tipos: **funciones** o **procedimientos**, las **funciones** se caracterizan porque siempre devuelven un valor mientras que los **procedimientos** pueden devolver 0,1 o n valores en forma de lista de parámetros.

Características que deben tener los módulos

a) Deben esta jerarquizados
b) Deben ser pequeños y sencillos
c) Deben ocultar los detalles poco importantes a módulos superiores en jerarquía.
d) Debe usar tantos módulos de más baja jerarquía como sea necesario para cumplir con el modo b)
e) Deben usar estructuras de datos y control adecuados para cumplir con el punto b)
f) Deben ser legibles, es decir que no solo el autor pueda entenderlos, sino cualquiera que tenga acceso a ellos con un conocimiento fundamental de programación

5.3 Ventajas de la modularidad

El uso de modularidad para diseñar programas tiene varias ventajas:

a) Disminuye la complejidad del problema a resolver.
b) Aumenta la legibilidad y confiabilidad
c) Disminuye costos computacionales.
d) Se obtiene un mejor control del proyecto.
e) Se pueden agregar nuevos módulos.
f) Facilita el mantenimiento

La programación modular se representa por medio de bloques donde cada uno hace referencia a un modulo. En la representación esquemática, se manifiesta la comunicación entre el módulo principal y los módulos secundarios.

El modulo principal coordina a todos los demás módulos. Debe ser claro y conciso, y reflejar los elementos fundamentales del problema.

5.4 Proceso de modularización ("divide y vencerás")

El proceso de subdivisión del problema complejo en subproblemas más fáciles de resolver consiste en la comprensión total del problema y posteriormente se procede a la subdivisión de éste en partes más pequeñas.

Se llevan a cabo las siguientes fases para la modularización:

1) Se realiza un análisis del problema considerando las especificaciones del mismo (comprensión del problema)
2) Se forma un primer modulo enunciando el problema en términos de la solución a éste.
3) Se toma este módulo y se busca la forma de dividirlo en otro módulos más pequeños que ejecuten tareas o funciones especificas.
4) Se repite el paso 3) para cada módulo nuevo definido, hasta llegar a un nivel de detalle adecuado, es decir, hasta que cada módulo ejecute una tarea especifica, que este claramente definida y que el diseño de la lógica del mismo resulte fácil (utilizando pseudocódigo).
5) Se realizan pruebas de escritorio parciales a cada módulo, para asegurarse que éstos realizan correctamente su tarea especifica.
6) Se realiza una prueba de integración en la que se consideran todos los módulos, así como el módulo principal.
7) Si es el caso, se codifican los módulos en un lenguaje de programación, iniciando desde los módulos del nivel más inferior en la estructura jerárquica de la solución.

5.5 Llamada a los módulos

Variables globales, locales y parámetros

En el diseño modular participan tres tipos de variables que reciben su nombre dependiendo de su ámbito de operación.

- Las ***variables globales*** son aquellas que se definen o declaran fuera del módulo principal y tienen efecto en todos los módulos. Esto es, una variable global puede ser modificada en cualquier módulo incluido el principal.

- Las ***variables locales*** son las que se definen dentro de un módulo, y son distintas a las que se definen en otros módulos, a pesar de que pueden tener el

mismo nombre. El contexto de estas variables es el módulo donde se declaran, fuera de éstos no son reconocidas o válidas.

- Los ***parámetros*** son un mecanismo de comunicación o intercambio de información entre los módulos, inclusive el módulo principal. Se relacionan de acuerdo a su posición y el tipo de dato asociado.

Existen dos tipos de parámetros:

Por Valor. Son constantes, el valor en sí mismo de los datos que se envían como argumentos en la invocación o llamada al módulo, la cual se realiza, escribiendo únicamente el dato del módulo y sus parámetros en el caso de que tenga.

Una característica de los paramentos por valor es que éstos copian sus valores en los argumentos de los módulos. Al finalizar la llamada del módulo, los parámetros no se ven modificados; no alteran su valor.

Por referencia: son variables; es copia de la referencia o localidad de memoria donde se encuentran los datos que se envían como argumentos en la invocación. La característica de estos parámetros es que son modificados dentro del módulo en el cual fueron invocados. Con el uso de parámetros por referencia, un módulo puede devolver más de un valor al módulo principal o a los módulos que lo invoquen.

Módulos predefinidos por el sistema (funciones)

Existen algunos módulos definidos por defecto en el sistema, los cuales se pueden clasificar dependiendo de la función de éstos. Para el uso de estos módulos, solamente debemos invocarlos mediante su nombre y parámetros, agregando previamente la librería donde se encuentran definidos. La librería contiene a un conjunto de módulos que se pueden utilizar. Estos módulos pueden ser: matemáticos, carácter, cadena, etc.

Por ejemplo la librería matemática contiene los módulos entre otros [2]:

Función	Descripción
exp(x)	**exponencial de x, x es entero o real**
ln(x)	**logaritmo neperiano de x, x es entero o real**
raiz2(x)	**raíz cuadrada de x, x es entero o real**
seno(x)	**Seno de x, x es entero o real**
redondeo(x)	**redondea x, x es real**

Tabla 5.1 División de problemas en subproblemas

Ejemplo 5.1

Si deseas calcular la raíz cuadrada de un número, podemos realizar las siguientes operaciones:

```
Leer x;
Real  r = raiz2( x )  // calcula la raíz cuadrada de x
Escribir  r
```

Esquema 5.1 Sintaxis de la declaración de un módulo en pseudocódigo

5.5 Ejemplos de programación modular

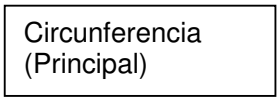
Ejemplo 5.1 Circunferencia

Elaborar un algoritmo modular para calcular el área, perímetro y diámetro de una circunferencia de radio R.

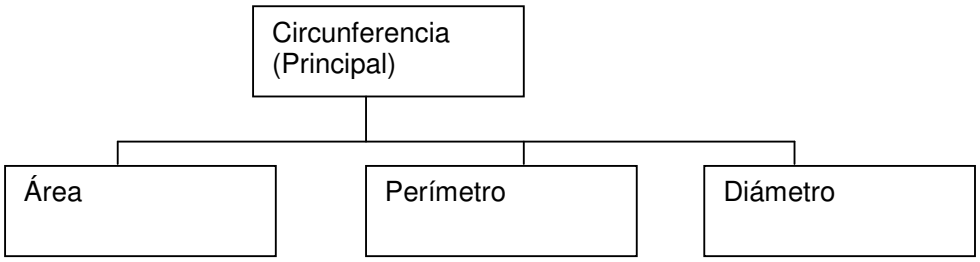
Paso 1. Se consideran los datos de entrada, salida y procesos para realizar los cálculos que resuelven el problema.
Datos de entrada: Radio **R**
Datos de salida:

```
Area = Pi*R*R
Perímetro= 2*Pi*R
Diámetro = 2*R
```

Paso 2.



Paso 3. El problema puede ser dividido en los siguientes módulos



En este caso no es necesario seguir dividiendo los módulos en submódulos.

Paso 4: Diseñar la lógica de programación de cada modulo, iniciando desde los módulos de nivel mas inferior.

```
Real Area (real R)// declaración del módulo
Inicio
        Real a;
         a=Pi*R*R
```

```
        Regresar( a)
FinModulo

Real Perimetro(real R)
Inicio
        Real p;
         p=2*Pi*R
        Regresar( p)
FinModulo


Real Diametro(real R)
Inicio
        Real d;
         d=2*R
        Regresar( d)
FinModulo

Circunferencia (Principal)
Inicio
        Real radio, A,P,D
        Escribir "Ingrese radio:"
        Leer radio
        A= Area( radio) //llamada del modulo
        P= Perimetro( radio)
        D= Diametro( radio)
        Escribir  " El area es = ", A
        Escribir  " El perimetro es = ", P
        Escribir  " El Diametro es = ", A
FinModulo
```

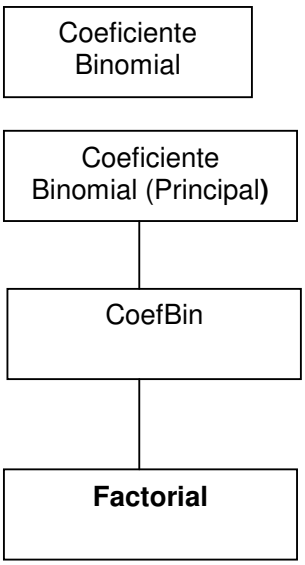
Paso 5. Realizar pruebas de escritorio a cada modulo individualmente
Paso 6. Realizar pruebas de escritorio integrando todos los modulo.

Ejemplo 5.2

Coeficiente Binomial. Algoritmo modular para calcular el coeficiente binomial definido como:

$$C(n, k) = n! / k! * (n - k)!$$

Análisis:
Datos: n, k
Procesos: Factorial de un número y *C(n,k).*




```
Real Factorial (real n)
Inicio
        Real f=1;
        Entero i
        Para i= 1 hasta n hacer
                f = f *i
        finpara
        regresar(f)
FinModulo

Real CoefBin( real n, real k)
Inicio
Regresar( factorial(n) / (factorial(k)* factorial (n-k))
FinModulo

Nada CoeficienteBinomial( Principal )
Inicio
        Real n, k
        Leer n, k
        Escribir  " Coeficiente Binomial =",  CoefBin(n, k )
FinModulo (Principal)
```