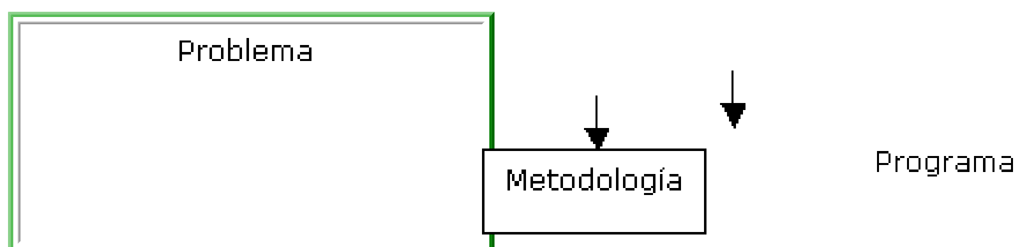
3. METODOLOGÍA DE SOLUCIÓN DE PROBLEMAS CON EL COMPUTADOR

El desarrollo de un programa que resuelva un problema dado es una tarea compleja, ya que es necesario tener en cuenta de manera simultánea muchos elementos. Por lo tanto, es indispensable usar una metodología para solucionar problemas con el computador.

Esta metodología es un conjunto o sistema de métodos, principios y reglas que permiten enfrentar de manera sistemática el desarrollo de un programa que resuelve un problema algorítmico. Estas metodologías generalmente se estructuran como una secuencia de pasos que parten de la definición del problema y culminan con un programa que lo resuelve.



A continuación se presenta de manera general los pasos de una metodología:

Análisis del problema	Con la cual se busca comprender totalmente el problema a resolver.
Especificación del problema	Con la cual se establece de manera precisa las entradas, salidas y las condiciones que deben cumplir.
Diseño del algoritmo	En esta etapa se construye un algoritmo que cumpla con la especificación.
Prueba del algoritmo y refinamiento	Entendimiento Verificación
Codificación	Se traduce el algoritmo a un lenguaje de programación.
Prueba y Verificación	Se realizan pruebas del programa implementado para determinar su validez en la resolución del problema.

Nota: Los ejemplos incluidos en esta sección en pseudicódigo son solo para ilustrar la metodología mas no se espera que el estudiante los comprenda

3.1 DIALOGO

En el primer paso en el proceso de solución a un problema se debe determinar de manera clara y concisa la siguiente información:

1. Los objetos conocidos, es decir, aquellos objetos de los cuales poseemos información total o parcial útil en la búsqueda de los objetos desconocidos.
2. Las condiciones, aquellas relaciones establecidas entre los objetos conocidos y los desconocidos. Para esto se deben encontrar entre otras, la dependencia entre los valores de los objetos desconocidos de los valores de los objetos conocidos y que restricciones le impone el planteamiento del problema a dichos objetos.
3. Los valores posibles que pueden tomar los objetos desconocidos.

Ejemplo. Sean los puntos $P=(a,b)$ y $Q=(c,d)$ que definen una recta, encontrar un segmento de recta perpendicular a la anterior que pase por el punto medio de los puntos dados.

OBJETOS DESCONOCIDOS	Un segmento de recta.
OBJETOS CONOCIDOS	Los puntos *P* y *Q*.
CONDICIONES	El segmento de recta debe pasar por el punto medio entre *P* y *Q*, y debe ser perpendicular a la recta trazada entre *P* y *Q*.

3.2 ESPECIFICACION DE ALGORITMOS

Después de entender totalmente el problema a resolver (lo cual se consigue con la etapa del diálogo), se debe realizar una especificación del algoritmo que permite encontrar su solución. Un algoritmo que no esté claramente especificado puede ser interpretado de diferentes maneras y al diseñarlo se puede terminar con un algoritmo que no sirve para solucionar el problema.

La especificación de un algoritmo se hace mediante una descripción clara y precisa de:

1. Las entradas que el algoritmo recibirá.
2. Las salidas que el algoritmo proporcionará.
3. La dependencia que mantendrán las salidas obtenidas con las entradas recibidas.

Esta descripción puede ser presentada mediante un diagrama de caja negra como el de la siguiente figura:

3.2.1 Pasos para la especificación de algoritmos

Especificar Entradas	Las entradas corresponden a los objetos conocidos. Se debe indicar claramente la descripción, cantidad y tipo de las mismas.
Especificar Salidas	Las salidas corresponden a los objetos desconocidos del problema. Se debe indicar claramente la cantidad, descripción y tipo de las mismas.
Especificar Condiciones	Se describe claramente como dependen las salidas de las entradas, se puede usar lenguaje matemático o informal.

3.2.2 Ejemplos de especificación

PROBLEMA 1: Construir un algoritmo que calcule el promedio de 4 notas.

ESPECIFICACION A: (Sin diagrama de caja negra)

Entradas	N_1,N_2,N_3,N_4 (notas parciales) de tipo Real.
Salidas	Final (nota final) de tipo Real.
Condiciones	$Final = \frac{N_1+N_2+N_3+N_4}{4}$

ESPECIFICACION B: (Con diagrama de caja negra)

DIAGRAMA DE CAJA NEGRA:

Descripción de Entradas y Salidas	N_i: Nota i-ésima con i=1,2,3,4, Final: Nota Final.
Tipo de Entradas y Salidas	N_1, N_2, N_3, N_4, Final $\in$ Reales.
Condiciones	$Final = \frac{N_1+N_2+N_3+N_4}{4}$

PROBLEMA 2: Construir un algoritmo que determine el mayor de tres números enteros.

ESPECIFICACION A

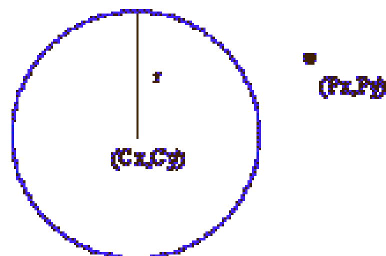
Entradas	A,B,C (números de entrada) de tipo Real.
Salidas	Mayor (valor mayor) de tipo Real.
Condiciones	Mayor debe ser el valor máximo de A, B y C.

ESPECIFICACION B

DIAGRAMA DE CAJA NEGRA:

Descripción de Entradas y Salidas	A, B, C: Números de entrada, Mayor: Valor Mayor.
Tipo de Entradas y Salidas	A, B, C, Mayor $\in$ Enteros
Condiciones	$Mayor = A$ ó $Mayor = B$ ó $Mayor = c$, y $Mayor \geq A$ y $Mayor \geq B$ y $Mayor \geq c$

PROBLEMA 3: Determinar si un punto está dentro de un círculo.



ESPECIFICACION A

Entradas	Cx (coordenada x del círculo) de tipo Real. Cy (coordenada y del círculo) de tipo Real. r (radio del círculo) de tipo Real. Px (coordenada x del punto) de tipo Real. Py (coordenada y del punto) de tipo Real.
Salidas	Pertenece de tipo Booleano, (indica si el punto está dentro o fuera del círculo).
Condiciones	Pertenece = Verdadero, si el punto está dentro del círculo. Pertenece = Falso, si el punto está fuera del círculo.

ESPECIFICACION B

DIAGRAMA DE CAJA NEGRA:

Descripción de Entradas y Salidas	(Cx, Cy): Coordenadas del círculo. r: Radio del círculo. (Px, Py): Coordenadas del punto.
Tipo de Entradas y Salidas	Cx, Cy, r, Px, Py Mayor ∈ Enteros.
Condiciones	$Pertenece = \begin{cases} Verdadero & si\ r \geq \sqrt{(Cx-Px)^2+(Cy-Py)^2} \\ Falso & en\ otro\ caso \end{cases}$

3.3 DISEÑO ESTRUCTURADO DE ALGORITMOS

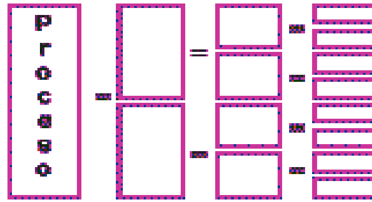
La fase de diseño del algoritmo, es decir, la fase en la que se construye el algoritmo que permitirá encontrar la solución al problema, está dividida en dos pasos importantes:

División: En el que a partir de la especificación del algoritmo se divide el proceso (algoritmo en abstracto) en varios subprocesos hasta llegar al nivel de instrucción.

Abstracción: En el que se revisa que porciones del algoritmo se repiten o son muy utilizadas y con las cuales se construyen funciones y/o procedimientos.

3.3.1 División

Consiste en subdividir de manera sistemática el proceso en una colección de pasos más pequeños. Está subdivisión se realiza de manera repetida hasta llegar al nivel de instrucción.



Durante el proceso de división se determina la estructura de control adecuada, ya sea, secuencia, selección, repetición, asignación, lectura o escritura, que se puede asociar con cada subproceso obtenido. Tanto los pasos intermedios de subdivisión como el resultado final pueden ser representados por un diagrama de flujo o por pseudo código.

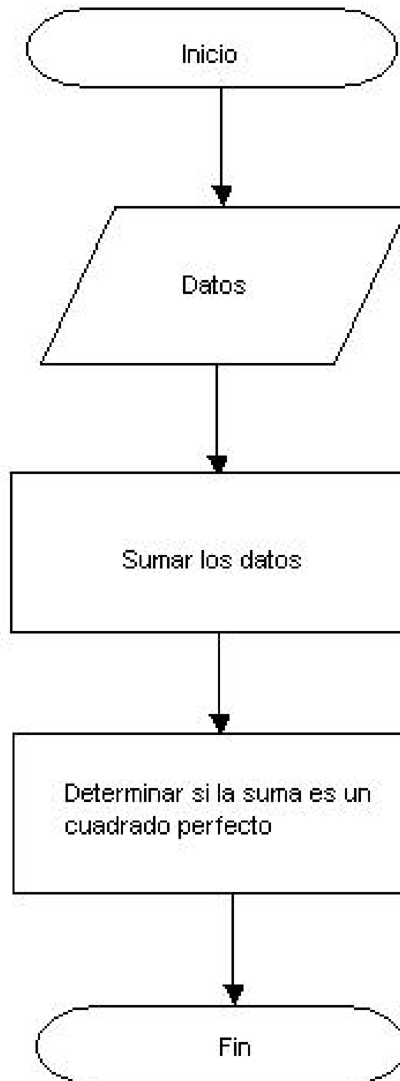
Problema: Realizar un programa que lea una serie de *n* números enteros y determine si la suma de los mismos es un cuadrado perfecto.
Especificación:

donde n ∈N, datos es una colección de n números naturales, Es_cuadrado ∈ Booleano

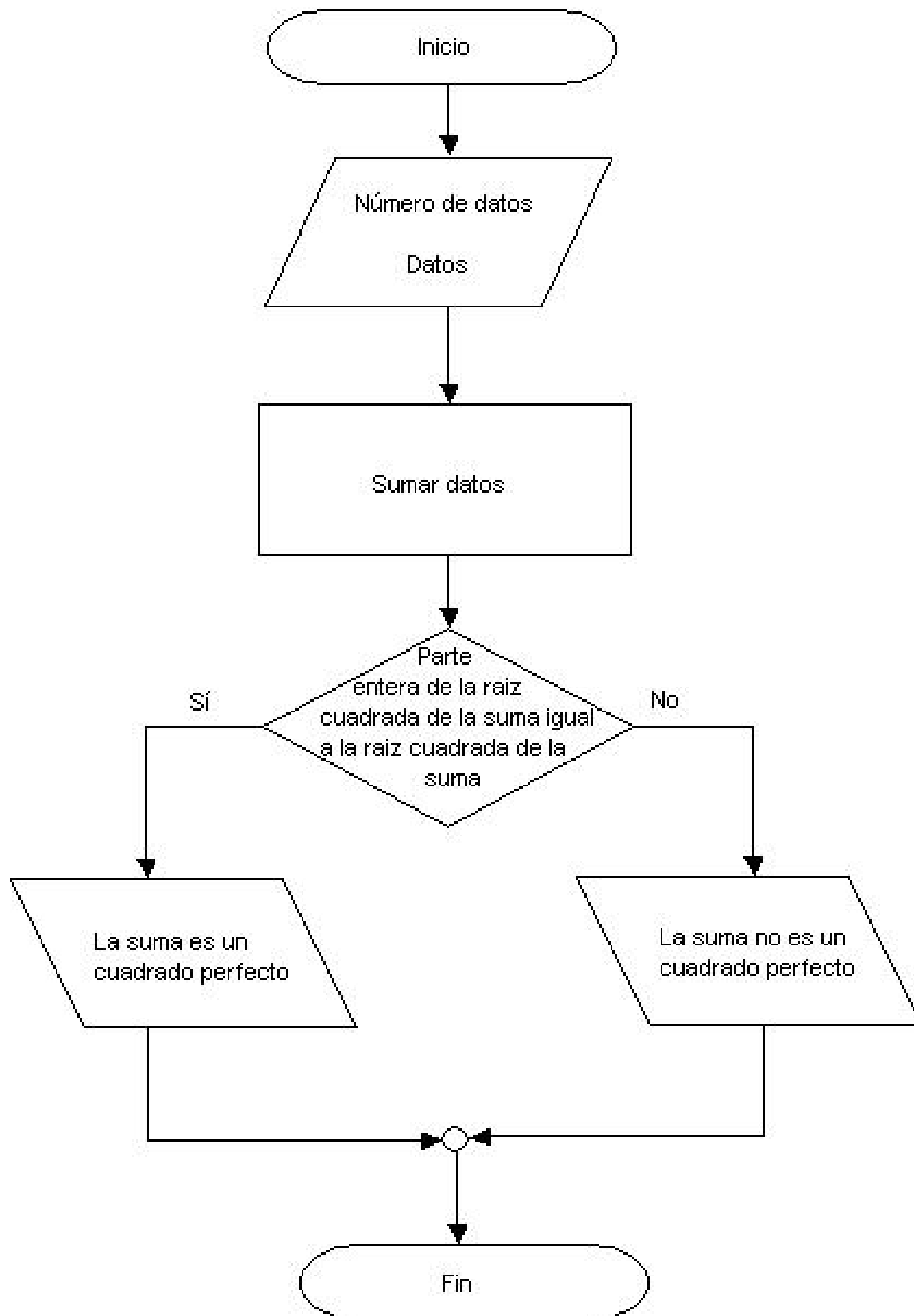
$$Es_cuadrado = \begin{cases} V & \text{si } \sum_{i=1}^{n} datos_i \text{ es cuadrado perfecto} \\ F & \text{en otro caso} \end{cases}$$

División:

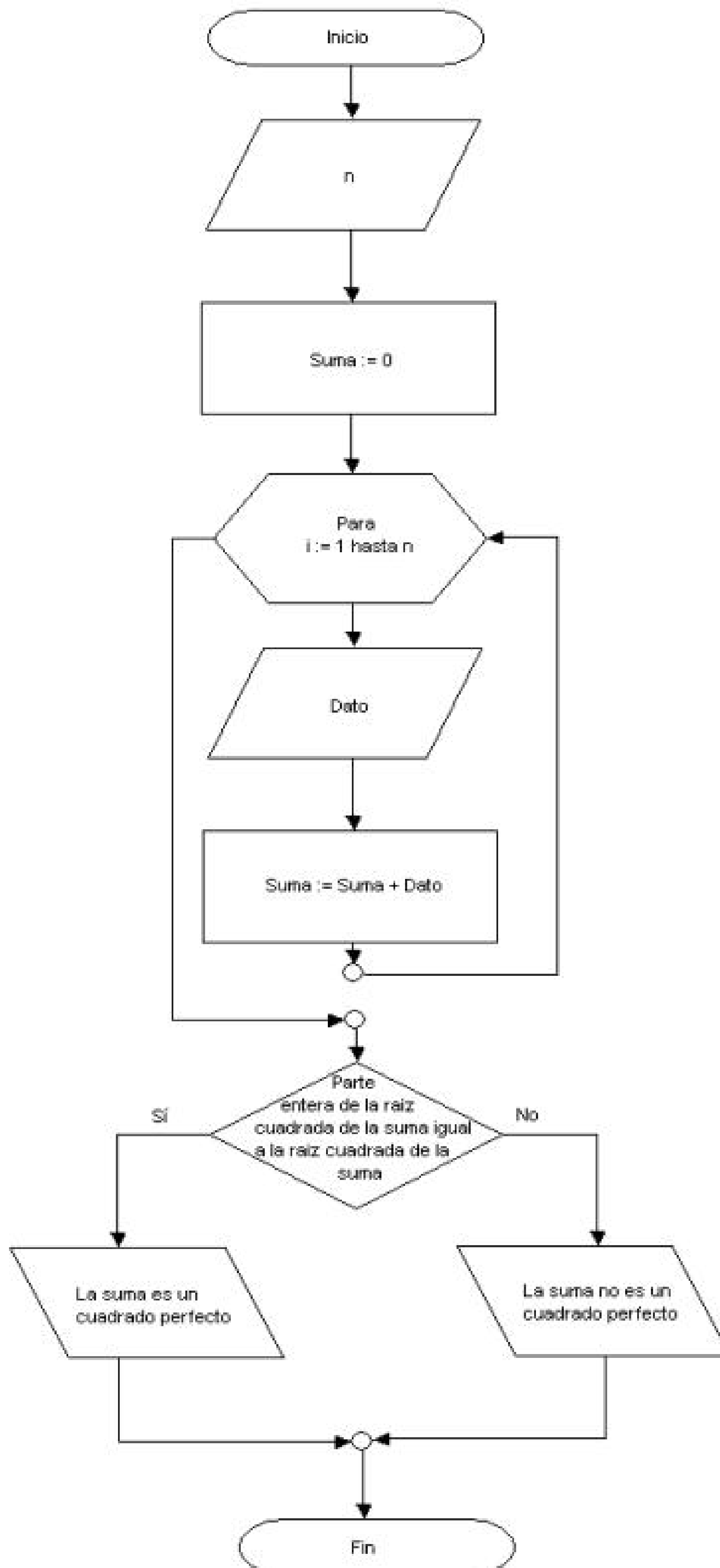
Primera Iteración

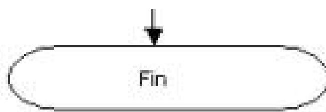


Segunda Iteración



Tercer Iteración (final en este ejemplo)


Inicio
n
Suma := 0
Para
i := 1 hasta n
Dato
Suma := Suma + Dato
Parte
entera de la raiz
cuadrada de la suma igual
a la raiz cuadrada de la
suma
Sí
No
La suma es un
cuadrado perfecto
La suma no es un
cuadrado perfecto
Fin



Pseudo-código

```
 procedimiento principal()
variables
n : entero
suma : entero
dato : entero
i : entero
inicio
escribir ( "Número de datos:")
leer( n)
suma := 0
para( i:=1 hasta n) hacer
  escribir("ingrese un dato:" )
  leer( dato)
  suma := suma + dato
fin_para
si(piso( raiz2( suma ) ) = raiz2(suma ))
   entonces
               escribir ("La suma es un cuadrado perfecto")
sino
     escribir ("La suma no es un cuadrado perfecto")
fin_si
fin_procedimiento
```

3.3.2 Definición de abstracciones

Identificar que secuencias de pasos se utilizan más de una vez en diferentes partes del proceso.

- Recolectar estas secuencias de pasos en funciones y procedimientos según sea el caso.
- Documentar cada función y procedimiento especificando claramente:
- El propósito de la función (o procedimiento).
- El nombre, tipo y propósito de cada argumento.
- El resultado (o efectos laterales).

Resultado final: Diagrama de flujo (o pseudo código) final incluyendo las funciones y procedimientos.

Ejemplo:

Problema: Desarrollar un programa que dados dos conjuntos finitos de enteros y un entero, determine si el entero dado esta o no en alguno de los dos conjuntos.
Solución: Se seguirá la metodología propuesta en este libro.

Dialogo: El primer paso es entender completamente el problema y que esquemas de representación se usarán. En general, se pueden utilizar arreglos de elementos de tipo **T** para representar conjuntos finitos de elementos de tipo **T**[1]. El uso de arreglos para representar conjuntos, requiere que se realicen ciertas validaciones

que garanticen la validez como conjunto, por ejemplo, que en el arreglo no este un mismo elemento dos veces (en un conjunto un elemento esta solo una vez). Teniendo en cuenta estos razonamientos, se puede decir que:

- Los objetos conocidos son dos conjuntos (arreglos) y el entero.
- El objeto desconocido es un valor de verdad.

La condición es que si el entero dado está en alguno de los conjuntos dados el valor de verdad será verdadero y será falso si no está en alguno de los dos.

Especificacion:

donde,

A :es un conjunto de enteros.
B : es un conjunto de enteros. elemento : es el entero a comprobar si esta en alguno de los conjuntos. bandera : es un booleano que indica si el el elemento esta o no esta.

$$bandera = \begin{cases} V \text{ si } elemento \in A \vee elemento \in B \\ F \text{ en otro caso} \end{cases}$$

División:

Primer Iteración.

Inicio
1. Leer primer conjunto.
2. Leer segundo conjunto.
3. Leer entero.
4. Determinar si el entero esta en el primer conjunto.
5. Determinar si el entero esta en el segundo conjunto.
6. Imprimir el resultado.
Fin

Segunda Iteración.

1. Leer primer conjunto se divide en:
1.1. Leer un dato.
1.2. Determinar si el elemento no esta en el primer conjunto.
1.3. Si no esta agregar el dato al primer conjunto, si ya esta mostrar un mensaje de error.
1.4. Preguntar si el usuario desea ingresar un nuevo elemento al primer conjunto.
1.5. Si el usuario desea ingresar un nuevo elemento volver a 1.1.
2. Leer segundo conjunto se divide en:
2.1. Leer un dato.
2.2. Determinar si el elemento no esta en el segundo conjunto.
2.3. Si no esta agregar el dato al segundo conjunto, si ya esta mostrar un mensaje de error.
2.4. Preguntar si el usuario desea ingresar un nuevo elemento al segundo conjunto.
2.5. Si el usuario desea ingresar un nuevo elemento volver a 2.1
3. Leer entero.
4. Determinar si el entero esta en el primer conjunto.
5. Determinar si el entero esta al segundo conjunto.
6. Imprimir el resultado.

Tercera Iteración, asociar las instrucciones apropiadas.

```
procedimiento principal()
variables
i : entero
j : entero
n : entero
m : entero
elemento : entero
continuar : carácter
bandera1 : booleano
bandera2 : booleano
A : arreglo [100] de entero
B : arreglo [100] de entero
/* el codigo siguiente lee el conjunto A*/
inicio
n := 0
escribir(“Desea ingresar elementos al conjunto A (S/N):”)
leer( continuar)
mientras( continuar = ‘S’ | continuar = ‘s’) hacer
   escribir( “Ingrese el elemento al conjunto A:”)
   leer( elemento)
/* el codigo siguiente prueba si el elemento esta en el conjunto A */
  i := 0
  mientras( i<n & A[i]<> elemento) hacer
            i := i+1
 fin_mientras
 si( i = n) entonces
     A[n] := elemento
     n := n+1
 sino
     escribir( “Error: el elemento ya esta en el conjunto A” )
```

```
  fin_si
  escribir( “Desea ingresar mas elementos al conjunto A (S/N)” )
  leer( continuar)
fin_mientras
/* el codigo siguiente lee el conjunto B */
m := 0
escribir(“Desea ingresar mas elementos al conjunto B (S/N)” )
leer( continuar)
mientras( continuar = ‘S’ | continuar = ‘s’) hacer
     escribi( “Ingrese el elemento al conjunto B:”)
     leer( elemento)
/* el codigo siguiente prueba si el elemento esta en el conjunto B */
     i := 0
     mientras i<m & B[i] <> elemento hacer
        i := i+1
     fin_mientras
     si (i = m) entonces
        B[m] := elemento
        m := m+1
     sino
        escribir( “Error: el elemento ya esta en el conjunto B” )
     fin_si
     escribir( “Desea ingresar mas elementos al conjunto B (S/N)” )
     leer( continuar)
fin_mientras
/* el codigo siguiente lee un elemento a probar */
escribir( “Ingrese el dato a probar en los conjuntos” )
leer( elemento)
/* el codigo siguiente prueba si el elemento esta en el conjunto A */
   i := 0
   mientras( i<m & A[i] <> elemento) hacer
          i := i+1
    fin_mientras
si( i = n) entonces
     bandera1 := falso
sino
     bandera1 := verdadero
fin_si
/* el codigo siguiente prueba si el elemento esta en el conjunto B */
i := 0
mientras( i<m & B[i] <> elemento) hacer
          i := i+1
fin_mientras
si(i = m) entonces
     bandera2 := falso
sino
     bandera2 := verdadero
fin_si
/* el codigo siguiente determina si el elemento esta en alguno de los dos conjuntos */
si(bandera1 | bandera2) entonces
     escribir( “El dato dado esta en alguno de los dos conjuntos” )
sino
     escribir( “El dato dado esta en ninguno de los conjuntos” )
fin_si
fin_procedimiento
```

Abstracción: En el código obtenido mediante la fase de división se puede apreciar la existencia de porciones de código que aunque no son iguales pero son muy parecidas. Este es el caso de las porciones de código que permiten leer los conjuntos A y B (lineas de código 11-32 y 33-54), y las porciones de código que permiten determinar si un elemento esta en un conjunto (lineas de código 18-29, 40-51, 58-69 y 70-71). De esta manera se pueden crear un procedimiento que permita leer el conjunto y una función de retorne si un elemento esta en un conjunto o no que permite abstraer estas porciones de código. La función **pertenece** se define así:

```
pertenece: Arreglo[100] de Entero x Entero x Entero -> Booleano
( A     ,             ,    n    ,    e ) ÞV si e = A[i] para algún i
Þ F en otro caso
```

Se puede observar que esta función además de recibir el arreglo de datos y el elemento, recibe un entero adicional *n*. Este entero se utiliza para indicar el tamaño del conjunto dado. Esta función se codifica como sigue:

```
funcion pertenece( A :arreglo[100] de entero, n :entero, e :entero ):booleano
 variables
  bandera :booleano
  i : entero
 inicio
/* el codigo siguiente prueba si el elemento esta en el conjunto */
  i := 0
  mientras( i<n & A[i] <> e) hacer
          i := i+1
  fin_mientras
  si( i = n) entonces
       bandera := falso
  si_no
       bandera := verdadero
  fin_si
retornar bandera
fin_funcion
```

El procedimiento de leer el conjunto se puede codificar como sigue:

```
procedimiento leer_conjunto( var A :arreglo [100] de entero, var n :entero, c :carácter )
    variables

     i :entero
     elemento :entero
     continuar :carácter
inicio
/* el codigo siguiente lee el conjunto */
  n := 0
  escribir( "Desea ingresar elementos al conjunto ", c, " (S/N)")
  leer( continuar)
  mientras( continuar = 'S' | continuar = 's') hacer
    escribir( "Ingrese el elemento al conjunto ", A , " :" )
    leer( elemento)

/* el código siguiente prueba si el elemento esta en el conjunto y de no ser así lo
adiciona */
    si ¡pertenece( A, n, elemento ) entonces
      A[n] := elemento
      n := n+1
    sino
      escribir( "Error: el elemento ya esta en el conjunto " )
    fin_si
    escribir( "Desea ingresar mas elementos al conjunto (S/N)?")
    leer( continuar)
  fin_mientras
fin_procedimiento
```

Se puede observar que tanto el arreglo de datos como la variable de tamaño del conjunto pasan por referencia. Esto es debido a que este procedimiento modificará el arreglo de datos enviado como argumento, pues en éste es donde se almacenarán los datos leídos, y el argumento n donde se almacenará el tamaño del conjunto leido. Adicionalmente se pasa un caracter por valor, que indica el nombre del conjunto a leer, para imprimirlo en pantalla mientras se leen los elementos del conjunto de tal manera que el usuario conozca el nombre del conjunto que esta ingresando.

Otro aspecto importante que se puede destacar en este procedimiento es que usa la función ***pertenece*** para determinar si se debe o no adicionar el elemento leído.

De esta manera el algoritmo principal se puede presentar como sigue:

```
procedimiento principal()
variables
  n :entero
  m :entero
  elemento :entero
  A :arreglo [100] de entero
  B :arreglo [100] de entero
inicio
/* el codigo siguiente lee el conjunto A */
leer_conjunto( A, n, 'A' )
/* el codigo siguiente lee el conjunto B */
leer_conjunto( B, m, 'B' )
/* el codigo siguiente determina si el elemento esta en alguno de los dos
conjuntos */
si pertenece( A, n, elemento ) | pertenece( B, m, elemento ) entonces
   escribir( "El dato dado esta en alguno de los dos conjuntos")
sino
     escribir( "El dato dado esta en ninguno de los conjuntos" )
fin_si
fin_procedimiento
```

Se puede apreciar la reducción de líneas de código del programa y la facilidad de lectura de cada uno de estos algoritmos (función, procedimiento y algoritmo principal) respecto al algoritmo inicial realizado sin abstracción. Se deja al lector la escritura del programa completo, utilizando las reglas descritas en esta sección.

3.4 CODIFICACION

Cuando ya se ha diseñado completamente el algoritmo y se tiene escrito en algún esquema de representación (pseudo-código o diagrama de flujo), el siguiente paso es codificarlo en el lenguaje de programación definido para tal fin.

En este momento es cuando el programador interactua con el computador mediante la herramienta de software que disponga para codificar en el lenguaje seleccionado.

EJEMPLO. Tómese como base el pseudo-código desarrollado en la sección anterior, el programa en C++ para este pseudo-código sería:

```
#include <iostream.h>
bool  pertenece( int A[100, int n, int e)
{
   bool bandera;
    int i;
/* el codigo siguiente prueba si el elemento esta en el conjunto */
  i = 0;
   while(> i<n && A[i] != e )
{
   i = i+1;
 };
  if( i == n )
{
   bandera = false;
   }
  else
  {
   bandera = true;
};
 return bandera;
};
void leer_conjunto( intA[100], int & n, char c )
{
int i;
int elemento;
char continuar;
/* el codigo siguiente lee el conjunto */
n = 0;
  cout << "Desea ingresar elementos al conjunto";
  cout << c << " (S/N)";
  cin >> continuar;
  while( continuar == 'S' || continuar == 's' )
{
  cout << "Ingrese el elemento al conjunto" << A << " :";
  cin >> elemento;

/* el codigo siguiente prueba si el elemento esta en el conjunto y de no ser así lo adiciona */
  if( !pertenece( A, n, elemento ) )
{
  A[n] = elemento;
  n = n+1;
  }
 else
 {
  cout << "Error: el elemento ya esta en el conjunto";
 };
 cout << "Desea ingresar mas elementos al conjunto (S/N)" ;
 cin>> continuar;
 };
};
void main()
{
 int n;
```

```
 int m;
 int elemento;
 int A[100];
 int B[100];
/* el codigo siguiente lee el conjunto A */
 leer_conjunto( A, n, 'A' );

/* el codigo siguiente lee el conjunto B */
 leer_conjunto( B, m, 'B' )

/* el codigo siguiente determina si el elemento esta en alguno de los dos conjuntos */
 if( pertenece(A, n, elemento) || pertenece(B, m, elemento) )
{
 cout << "El dato dado esta en alguno de los dos conjuntos";
 }
 else
 {
 cout<< "El dato dado esta en ninguno de los conjuntos";
 };
};
```

3.4.1 Prueba de escritorio

La prueba de escritorio es una herramienta útil para entender que hace un determinado algoritmo, o para verificar que un algoritmo cumple con la especificación sin necesidad de ejecutarlo.
Básicamente, una prueba de escritorio es una ejecución 'a mano' del algoritmo, por lo tanto se debe llevar registro de los valores que va tomando cada una de las variables involucradas en el mismo.

A continuación se muestra un ejemplo de prueba de escritorio del siguiente algoritmo:

```
procedimiento principal()
variables
   suma :entero
   entrada :entero
   menor :entero
inicio
leer( entrada)
menor := entrada
suma:= 0
mientras (entrada <> 0) hacer
  si (entrada < menor) entonces
      menor =entrada
  fin_si
  suma:= suma + entrada
  leer( entrada)
fin_mientras
```

```
escribir( “valor menor:” )
escribir( menor)
escribir( “Suma:”)
escribir( suma)
fin_procedimiento
```

INSTRUCCIÓN	entrada	menor	suma	Pantalla
leer (entrada)	10			
menor := entrada		10		
suma :=0			0	
suma :=suma + entrada			10	
leer (entrada)	7			
menor := entrada		7		
suma:=suma + entrada			17	
leer(entrada)	9			
suma :=suma + entrada			26	
leer (entrada)	0			
escribir (“valor menor:**”)**				Valor Menor
escribir (menor)				7
escribir (“Suma:**”)**				Suma:
escribir (suma)				26

---

[1] El programador avanzado, que conozca estructuras de datos sabrá, que es posible y mejor representar los elementos de un conjunto mediante un Arbol.