Ficha 10

Subproblemas y Funciones

1.] Introducción al concepto de subproblema.

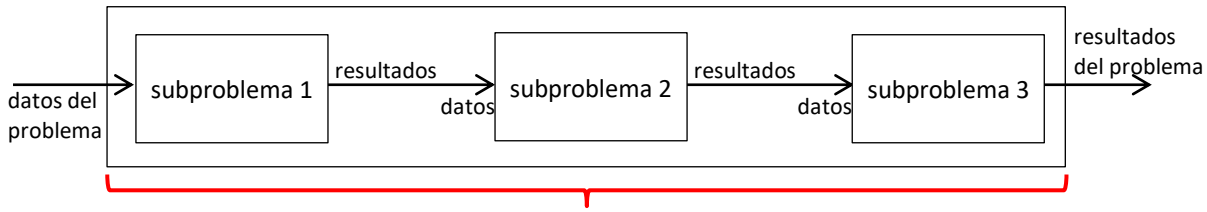
La mayoría de los problemas que normalmente se enfrentan en la práctica son de *estructura compuesta*, es decir, son problemas que pueden ser divididos en subproblemas cada vez menores en complejidad, hasta llegar a subproblemas que no necesiten nuevas subdivisiones (y que por eso se designan como *problemas simples*). Básicamente, un *subproblema* es un problema incluido dentro de otro de estructura más compleja.

Un principio básico en el planteo de algoritmos para resolver problemas, consiste justamente en tratar de identificar los subproblemas simples de un problema compuesto, considerando que cada subproblema simple es también un problema que admite datos, desarrolla procesos y genera resultados.

Esta forma básica de proceder, *centrando el análisis de un problema en los subproblemas que pudiera contener*, ha dado origen a la técnica o paradigma de programación que usaremos y que generalmente se conoce como *programación estructurada*[1], en la que justamente se trabaja de forma que los programadores descomponen un problema en problemas menores, programan por separado los procesos que resuelven esos subproblemas, y luego unen todo para formar el programa completo [1].

Idealmente, y en forma muy general, un problema compuesto se dividirá en tantos subproblemas como se hayan detectado de forma que cada uno de ellos requerirá datos y entregará resultados. Los resultados que cada uno genere, serán datos para los subproblemas siguientes o ya serán los resultados finales esperados, como lo que se ve en la *Figura 1* en la cual se supone un problema compuesto con tres subproblemas incluidos en él.

Figura 1: Esquema general de un *problema compuesto* que incluye tres *subproblemas*.



[1] Un conjunto de reglas y convenciones para trabajar en cierto contexto se designa en general como un *paradigma*. En consecuencia, la técnica de Programación Estructurada se puede designar también en como el *Paradigma de Programación Estructurada*, en contraposición a otras técnicas y formas de trabajo que constituyen otros paradigmas, como el *Paradigma de Programación Orientada a Objetos* o el *Paradigma de Programación Funcional*, entre otros.

La idea de esa figura, (insistimos: muy general y esquemática) es que los datos del problema compuesto original serán tomados a su vez como datos por alguno de sus subproblemas, el cual los procesará y obtendrá ciertos resultados parciales, que serán entregados como datos al siguiente subproblema. En algún momento, alguno de los subproblemas obtendrá y entregará los resultados finales que se esperaban para el problema compuesto original. Los procesos planteados en cada subproblema se pueden considerar independientes entre ellos, pero se terminan relacionando por esta secuencia de resultados – datos compartidos.

Está claro que en una situación real, este esquema puede variar y complicarse mucho más: los datos originales del problema compuesto podrían ser tomados por dos o más subproblemas (y no por uno sólo); los resultados finales podrían llegar a ser obtenidos por más de un subproblema (y no sólo por uno); y los resultados parciales de un subproblema podrían ser tomados como datos o entradas por más de un subproblema posterior (y no sólo por uno…) O incluso más: cualquiera de los subproblemas podría a su vez ser compuesto, dividiéndose en dos o más subproblemas él mismo. Lo que intenta rescatar la *Figura 1* es el fundamento conceptual básico: un problema compuesto contiene subproblemas que aprovechan mutuamente los resultados parciales obtenidos para llegar en algún momento al resultado final esperado.

A modo de ejemplo que permita aclarar el tema, analicemos ahora el siguiente problema de aplicación (la lógica requerida para resolverlo es muy simple, ya que la idea es sólo usarlo como modelo para entrar en el tema de los subproblemas):

Problema 25.) *En el contexto de un estudio estadístico, se requiere un programa que cargue tres números por teclado y luego proceda a determinar el menor de ellos y calcular el cuadrado y el cubo del mismo.*

a.) Identificación de componentes:

- **Resultados:** El menor, su cuadrado y su cubo. (*men*, *cuad*, *cubo*: int)
- **Datos:** Tres números distintos. (*a*, *b*, *c*: int)
- **Procesos:** El problema puede ser dividido en dos *subproblemas*: uno de ellos tiene como objetivo calcular el *menor de los tres valores de entrada*, y el segundo busca *calcular el cuadrado y el cubo de ese valor*. En vez de intentar realizar todo a la vez, lo cual crearía eventuales confusiones, se trata de plantear cada subproblema por separado, y luego unir las piezas para obtener el algoritmo completo que permita desarrollar un programa.

  Como cada subproblema es él mismo un problema, entonces cada uno tiene su propio modelo de datos, procesos y resultados. Y en ese sentido, el planteo de cada subproblema puede hacerse como sigue:

  Subproblema 1: *Determinar el menor de los tres valores de entrada.*

  - **Resultados:** El menor de tres números. (*men*: int)
  - **Datos:** Tres números. (*a*, *b*, *c*: int)
  - **Procesos:** El proceso de este subproblema consiste en plantear el esquema de condiciones que permita encontrar el menor entre los valores de las variables *a*, *b*, y *c*, lo cual puede hacer en base al siguiente modelo de pseudocódigo:

```
si  a < b  y  a < c:
     men  =  a
sino:
```

```
si b < c:
    men = b
sino:
    men  = c
```

Subproblema 2: *Calcular el cuadrado y el cubo del menor encontrado.*

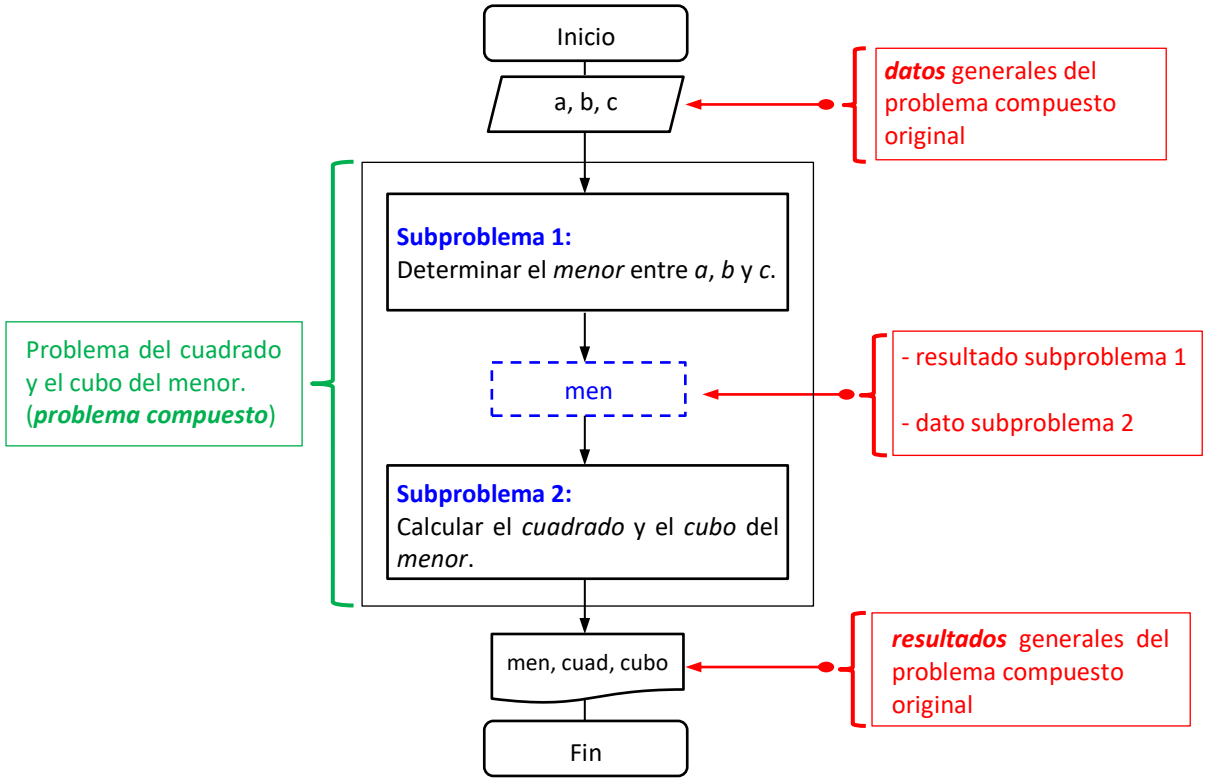
- **Resultados:** El cuadrado y el cubo del menor. (*cuad*, *cubo*: int)
- **Datos:** El menor de los valores de entrada. (*men*: int)
- **Procesos:** En este subproblema, todo el trabajo consiste en calcular el cuadrado y el cubo del menor:

```
cuad  =  men ** 2
cubo  =  men ** 3
```

Aquí debe notarse que este subproblema toma como dato al valor *men* que fue el resultado del subproblema anterior. Pero eso no significa que el valor de *men* debe ser cargado por teclado: su valor surge de los procesos del subproblema anterior, en particular de las condiciones y asignaciones usadas para calcular el valor menor.

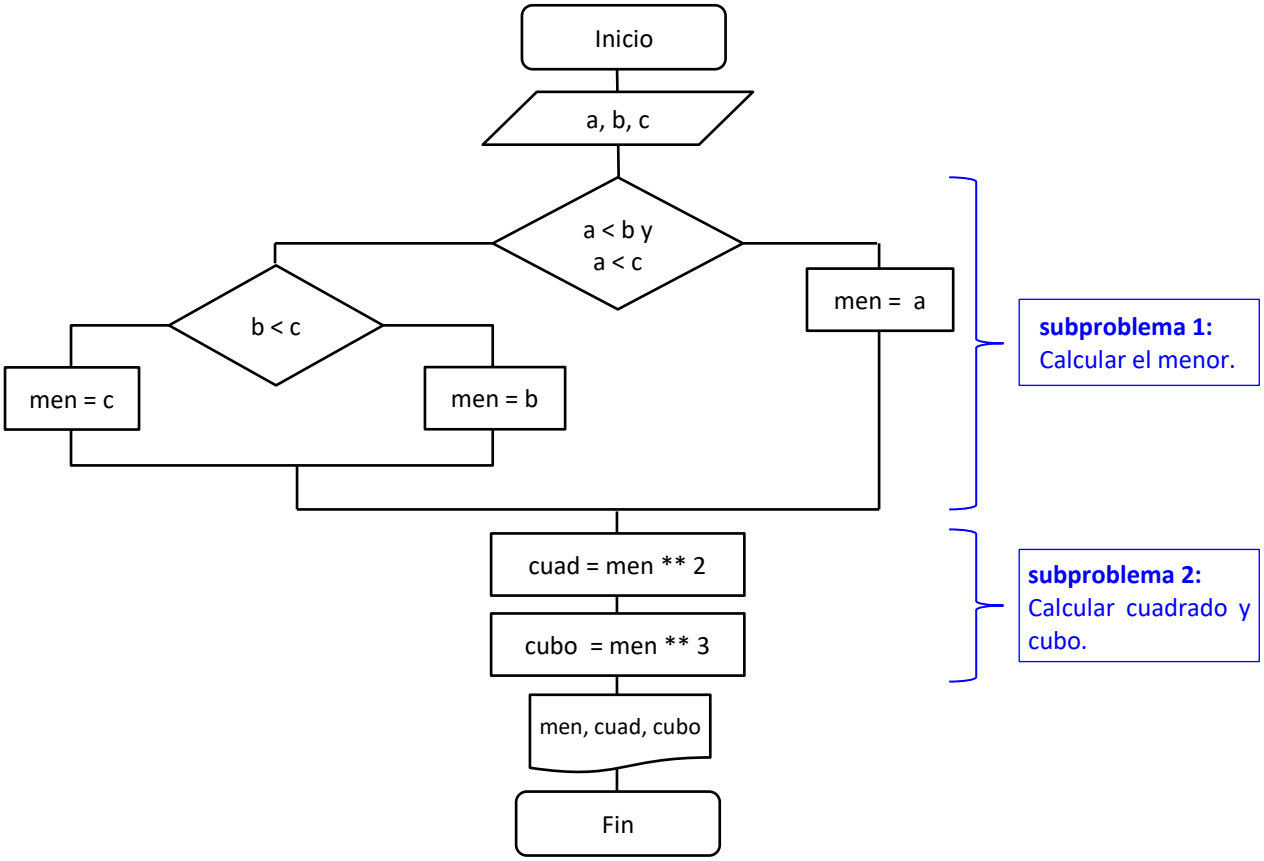
Es fácil ver que con estos dos subproblemas detectados y analizados, la estructura gráfica completa del problema original puede representarse a través del modelo que se ve en la *Figura 2* :

Figura 2: Modelo de subproblemas para el *problema del cuadrado y el cubo del menor*.



b.) Planteo del algoritmo: A partir de la discusión hecha en la identificación de componentes, y tomando como base el modelo de subproblemas de la *Figura 2*, se puede plantear un diagrama de flujo que contemple implícitamente a esos subproblemas, los cuales se muestran *remarcados con llaves de color azul* para facilitar su identificación (ver *Figura 3*):

Figura 3: Diagrama del flujo del problema del cubo y el cuadrado del menor, marcando subproblemas.



c.) Desarrollo del programa: Como siempre, en base al diagrama el script o programa en Python se deduce en forma inmediata:

```
__author__ = 'Cátedra de AED'


# títulos y carga de datos...
print('Determinación del cuadrado y el cubo del menor')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))
c = int(input('Tercer número: '))

# procesos...
# subproblema 1: determinar el menor...
if a < b and a < c:
    men = a
else:
    if b < c:
        men = b
    else:
        men = c

# subproblema 2: calcular el cuadrado y el cubo del menor...
cuad = men ** 2
cubo = men ** 3

# visualización de resultados...
print('El menor es:', men)
print('Su cuadrado es:', cuad)
print('Su cubo es:', cubo)
```

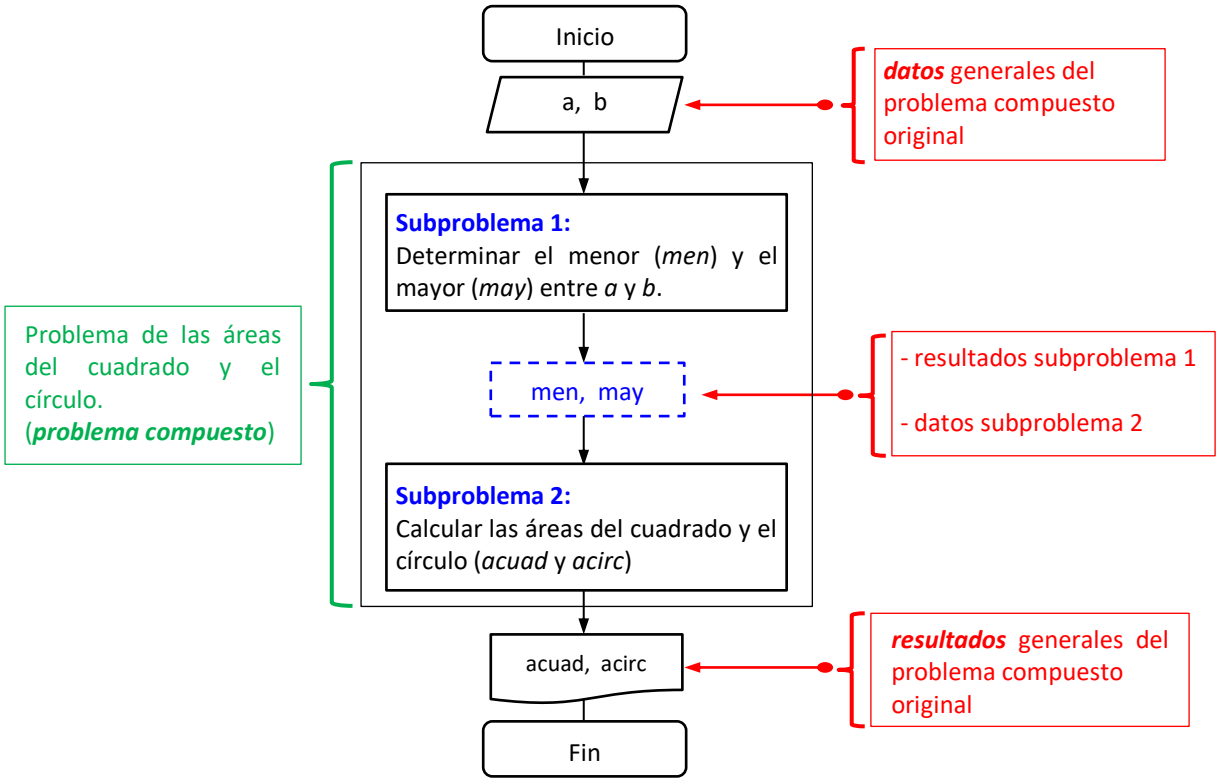
Como ya habrá podido notar, a medida que se estudian problemas de estructura más compleja y se incorporan nuevos elementos (como instrucciones condicionales, instrucciones repetitivas y subproblemas), se hace también cada vez más extenso el análisis del problema a nivel de identificación de datos, resultados y procesos. Hemos indicado en una ficha anterior que en la práctica, el programador hace gran parte de este proceso mentalmente, sin tanto rigor descriptivo previo. Y eso es lo que en general hemos hecho en cada una de las Fichas anteriores: para cada problema, discutir una brevísima identificación de datos y resultados, para pasar luego al planteo del diagrama de flujo o el pseudocódigo., y si fuese necesaria una discusión referida a ciertos aspectos lógicos de la solución, se incorpora brevemente antes de plantear el diagrama [1].

Considere entonces un nuevo ejemplo muy simple de aplicación:

Problema 26.) *Se cargan por teclado dos números. Calcular la superficie de un cuadrado, suponiendo como lado del mismo al mayor de los números dados y la superficie de un círculo suponiendo como radio del mismo al menor de los números dados.*

Discusión y solución: En este problema también se presentan dos subproblemas: en el primero se debe buscar el mayor y el menor entre dos valores, y en el segundo se deben calcular las áreas indicadas. Como el cálculo de las áreas no puede hacerse sin conocer cuál de los valores es el mayor y cuál es el menor, es obvio entonces que el subproblema de buscar el mayor y el menor debe resolverse antes que el cálculo de las áreas. Un esquema general de subproblemas podría ser el siguiente:

Figura 4: Modelo de subproblemas para el problema de las áreas del cuadrado y el círculo.



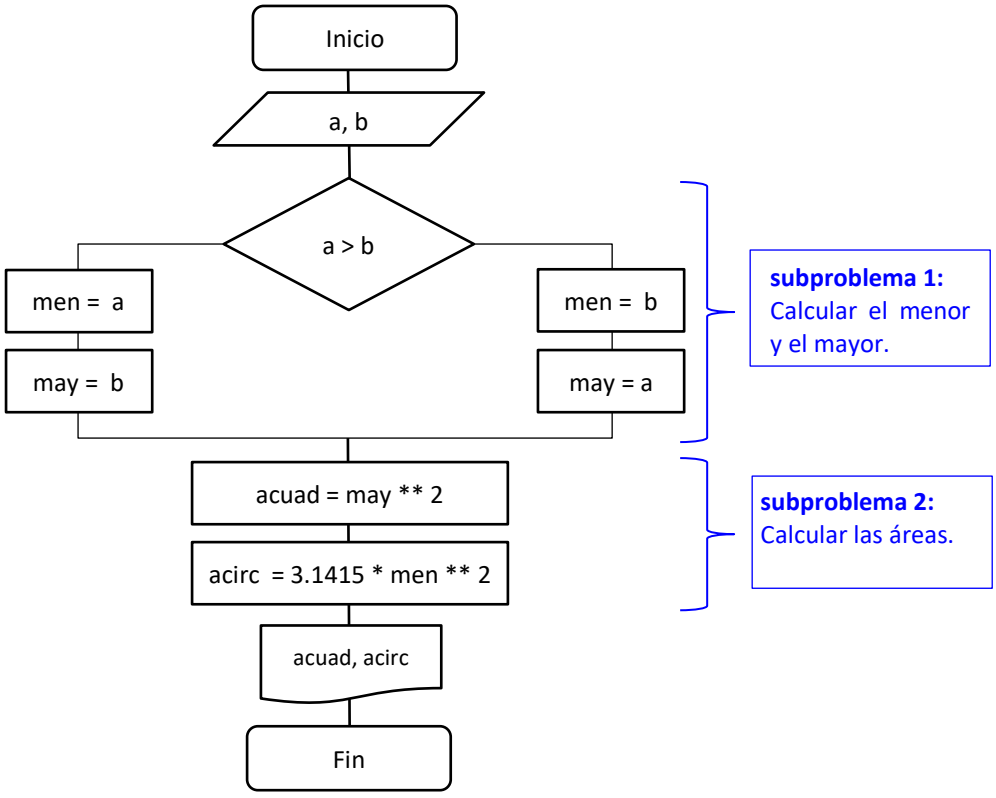
El *subproblema 1* tomará los valores de las variables *a* y *b* cargados por teclado, determinará el menor y el mayor y los copiará a su vez en las *variables temporales men* y *may*. De esta forma, si luego se requiere volver procesar estos valores en diferentes lugares del programa,

no será necesario preguntar nuevamente cuál es el mayor y cuál el menor: a partir del ordenamiento de variables producido por este subproblema, el programa trabajará con las variables *men* y *may* en lugar de *a* y *b*.

El *subproblema 2* simplemente debe calcular las dos áreas. El enunciado del problema indica que debe suponerse un cuadrado cuyo lado sea igual al mayor de los números de entrada (que tenemos copiado en *may*) y debe suponerse también un círculo cuyo radio sea igual al menor de esos datos (que tenemos en *men*). Por lo tanto, aplicando fórmulas muy conocidas de la geometría, el área del cuadrado (*acuad*) será *acuad = may ** 2* (el cuadrado del lado) y el área del círculo (*acirc*) será *acirc = 3.1415 * men ** 2* (*Pi* por radio al cuadrado).

El diagrama de flujo puede verse en la *Figura 5*:

Figura 5: Diagrama de flujo del problema de la superficie del cuadrado y del círculo.



c.) Desarrollo del programa: El script en Python no presenta dificultades. Sólo recuerde la importancia de respetar la indentación, que en el caso de las ramas de la condición en este caso es fundamental [2]:

```
__author__ = 'Cátedra de AED'

# títulos y carga de datos...
print('Cálculo de áreas de un cuadrado y un círculo...')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))

# procesos...
# subproblema 1: determinar el menor y el mayor...
if a > b:
    men = b
    may = a
```

```
else:
    men = a
    may = b

# subproblema 2: calcular la áreas...
acuad = may ** 2
acirc = 3.1415 * men ** 2

# visualización de resultados...
print('Area del cuadrado:', acuad)
print('Area del círculo:', acirc)
```

2.] Subrutinas: introducción y conceptos generales.

Como vimos, a partir de la noción de *subproblema* un problema principal puede descomponerse en partes o subproblemas más sencillos para facilitar su planteo y luego unir las piezas para lograr el planteo final. Sin embargo, en la forma en que aquí se trabajó, la división en subproblemas resultó ser un simple planteo de tipo mental: ni el diagrama de flujo del problema ni el programa o script hecho en Python muestran en forma clara cuales son los subproblemas en los que *se supone* está dividido el problema (a lo sumo, hemos incluido llaves y etiquetas de colores para remarcar la presencia de cada subproblema, pero sólo a modo de recurso informal). Y esa es la cuestión que aquí analizamos: se asume que quien estudie el diagrama de flujo y el programa en Python, comprende (o al menos intuye) la división en subproblemas que el programador definió.

Pero ¿por qué debe ser así? ¿No pueden plantearse el diagrama de flujo y el programa de forma tal que los subproblemas en que fueron divididos queden evidenciados *físicamente*, y no ya *intuitivamente*? En otras palabras: ¿cómo puede representarse un subproblema en un diagrama de flujo? ¿y en un programa hecho en Python? La respuesta a ambas cuestiones es usar lo que genéricamente se denominan *subrutinas*, y que se implementan mediante recursos que se designan de distintas maneras en cada lenguaje. En Python, las subrutinas se implementan mediante *funciones* [3].

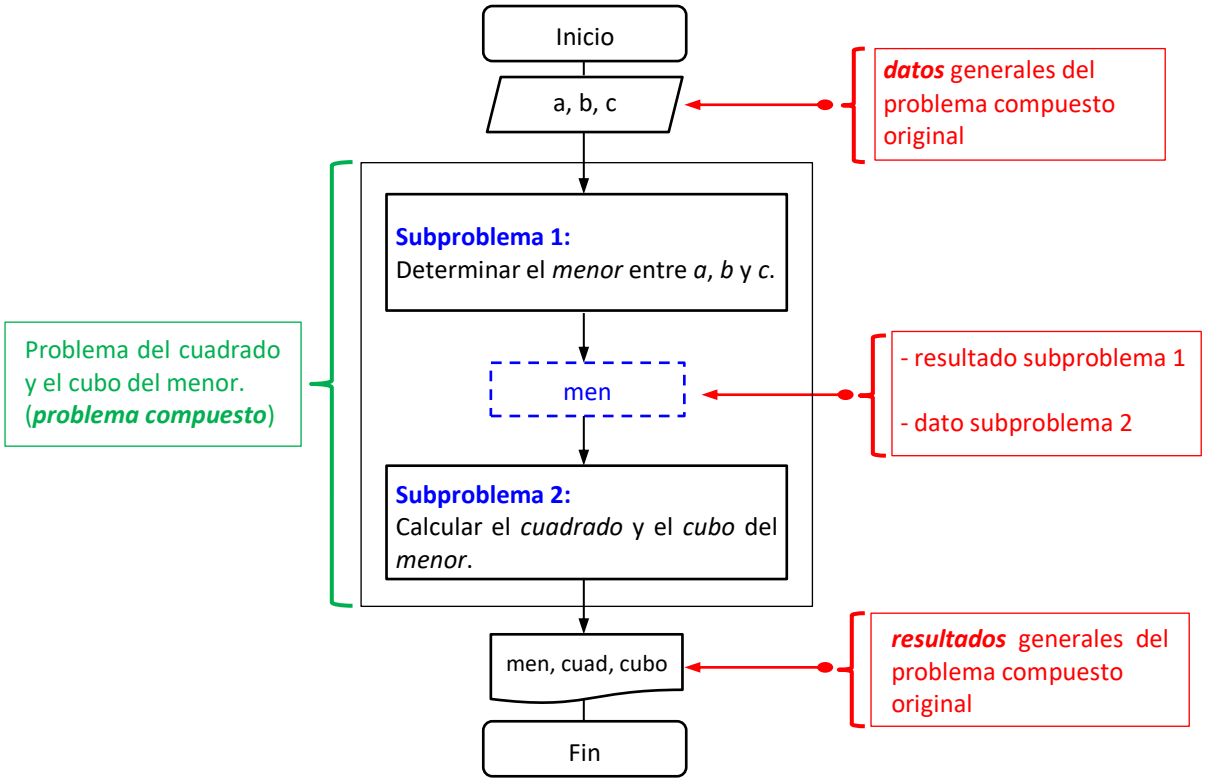
Esencialmente, una *subrutina* (o *subproceso*) es un segmento de un programa que se escribe en forma separada del programa principal. A cada subrutina el programador asocia un nombre o identificador y mediante ese identificador la subrutina puede ser activada todas las veces que sea necesario. Esto puede parecer complicado pero piense el estudiante que, aún sin saberlo, ya ha estado usando subrutinas: la función *print()* que utilizaba para mostrar salidas por consola, es una subrutina que ya viene lista para usar con el lenguaje. En ese sentido, como ya se sugirió, una *función* es una *subrutina* escrita en Python. También son ejemplos de subrutinas implementadas como funciones en Python [3] algunas otras conocidas, como *int()*, *float()*, *input()*, *pow()*, etc.

El lenguaje Python ya provee una serie de subrutinas listas para ser usadas, en forma de funciones incluidas en varias *librerías de funciones*, pero la idea ahora es que el programador desarrolle sus propias subrutinas, en forma de funciones propias definidas y escritas por el propio programador. Aunque lo que sigue no es una regla taxativa, lo que básicamente suele hacerse en el *Paradigma de la Programación Estructurada*, es definir una subrutina (en Python, una función) por cada subproblema que el programador detecte.

A modo de ejemplo podemos volver sobre el *problema 25* de esta misma Ficha, cuyo enunciado era básicamente: *cargar tres números por teclado, determinar el menor de ellos y*

calcular el cuadrado y el cubo del mismo. Como se vio oportunamente, este problema incluía dos subproblemas: el primero era determinar el menor, y el segundo era calcular el cuadrado y el cubo de ese menor. El gráfico que sigue en la *Figura 6*, es el mismo que se mostró en la *Figura 2*, y expone la estructura general de subproblemas que se sugirió para el planteo de la solución:

Figura 6: Modelo de subproblemas - *Problema del cuadrado y el cubo del menor*.



En base a lo anterior, el algoritmo general contendría entonces dos *subrutinas*: una para calcular el mayor y el menor entre los dos números de entrada, y otra para calcular el cuadrado y el cubo.

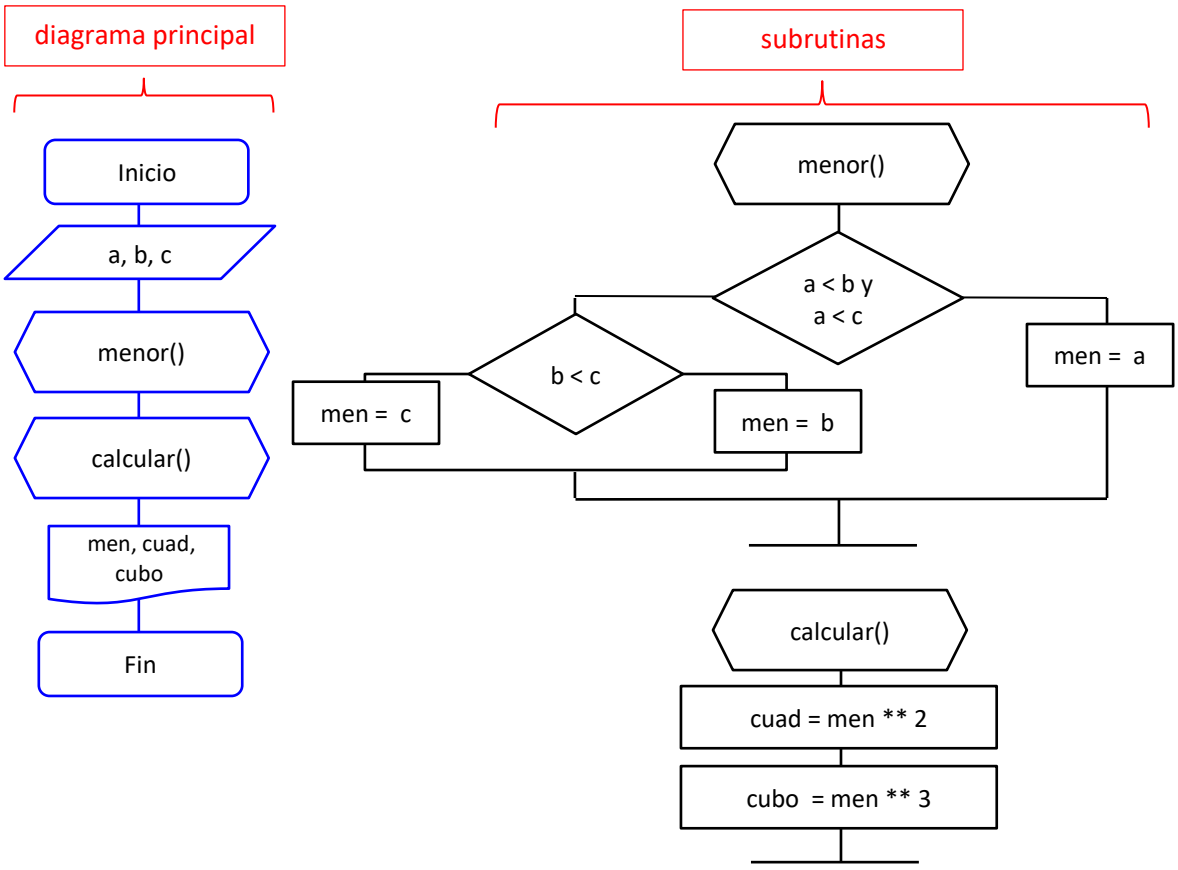
En un diagrama de flujo es especialmente fácil introducir nuevas reglas para denotar la presencia de una subrutina: simplemente se diagrama la misma por separado. El diagrama que se ve en la *Figura 7* (*página 193*) es el mismo que el que anteriormente mostramos en la *Figura 6* para el problema del cuadrado y el cubo, pero replanteado para incluir en forma explícita las subrutinas previstas por el programador [1].

Como puede verse, ahora hay un *diagrama principal* (que comienza y termina con los clásicos símbolos ovales de *Inicio* y *Fin*) y varios diagramas separados: uno por cada *subrutina* (es decir, uno por cada subproblema), pero de forma tal que el programador coloca un *nombre* o *identificador* a cada subrutina que grafica por separado. Por convención, y para anticiparnos a lo que luego se debe hacer en Python para declarar e invocar a una función, colocaremos al final de los nombres asociados a cada función un *par de paréntesis vacíos*.

En el *diagrama principal*, en lugar de graficar todos los procesos, sólo *se dejan indicados los nombres de las subrutinas* que los deben realizar, usando para ello el *símbolo de subrutina* (el hexágono de bases alargadas). Los nombres de cada subrutina son elegidos por el

programador, debiendo seguir para ello las mismas reglas y convenciones que valen para formar el nombre de una variable. Aquí, la primera subrutina se llama *menor()*, y la otra se llama *calcular()*:

Figura 7: Diagrama del *problema del cuadrado y el cubo*, replanteado para incluir *subrutinas*.



El diagrama de los procesos que realiza cada subrutina se desarrolla por separado, pero se comienza cada gráfico de subrutina con *el mismo símbolo hexagonal que se usó para dejarla indicada en el diagrama principal*. Los procesos internos de cada subrutina se grafican de la misma forma y con los mismos símbolos usados en cualquier diagrama de flujo. Y para indicar que el gráfico de la subrutina ha terminado, simplemente se coloca un trazo horizontal al final de su gráfica.

Observe que la lógica del diagrama no ha cambiado en absoluto, sino que sencillamente se ha estructurado el gráfico de manera diferente para que cada subproblema pueda ser entendido y estudiado por separado. De esta forma, si un problema es muy complicado y está dividido en muchos subproblemas, no será necesario plantear un "diagrama gigante" que abarque toda la lógica del mismo, sino que cada parte puede ser planteada y analizada por separado, incluso en momentos diferentes y por distintas personas.

3.] Funciones en Python - Parámetros y retorno de valores.

La división en subproblemas puede hacerse en un diagrama usando la técnica vista en la sección anterior. Si ahora se desea escribir el programa o script en Python, pero también dividiendo al mismo en subrutinas, puede hacerse (según ya dijimos) recurriendo a *funciones*. Como vimos, la idea es simple y directa: En Python, una *función* es una subrutina:

un segmento de programa que se codifica en forma separada, con un nombre asociado para poder activarla. En Python, una función tiene dos partes [3] [2]:

- La ***cabecera***: también llamada *encabezado* de la función. Es la primera línea de la función, en ella se indica el nombre de la misma, entre otros elementos que oportunamente veremos (y que se designan como *parámetros*).
- El ***bloque de acciones***: es la sección donde se indican los procesos que lleva a cabo la función. Este bloque debe comenzar a renglón seguido de la cabecera, y debe ir indentado hacia la derecha del comienzo de la cabecera. Es típico (como también veremos) que este bloque finalice con una instrucción de retorno de valor, llamada *return*.

A modo de ejemplo, veamos la forma que podría tener en Python la función que corresponde a la subrutina *menor()* de nuestro programa anterior:

```
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn
```

Aquí, la línea *def menor(n1, n2, n3):* es la ***cabecera*** de la función. Las variables *n1*, *n2*, *n3* encerradas entre los paréntesis se designan como *parámetros*, y esencialmente contienen *los datos* que la función debe procesar. Como se ve, es también en la cabecera donde el programador indica el nombre que llevará la función: en este caso, el nombre elegido fue *menor*.

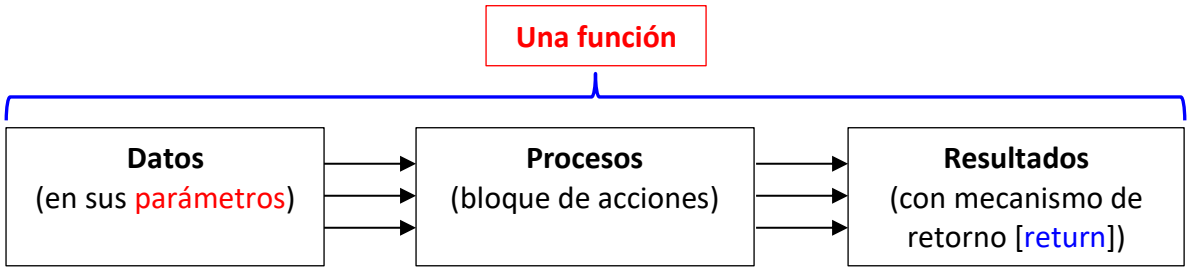
El resto de la función es el ***bloque de acciones*** de la misma. Como ya se dijo, el bloque se escribe a renglón seguido de la cabecera, e indentado hacia la derecha del comienzo de la misma. En ese bloque se escriben las instrucciones que la función debe ejecutar cuando sea invocada o activada desde otro lugar del programa. En este caso el bloque de la función contiene una *instrucción condicional anidada* para determinar cuál de los valores *n1*, *n2*, o *n3* es el menor, asignándolo en la variable *mn* de acuerdo al resultado de la verificación. En el ejemplo, la última instrucción del bloque de acciones es la instrucción *return mn*. Esa instrucción hace que la función, antes de finalizar, devuelva el valor contenido en la variable *mn* al punto desde el cual la función haya sido invocada. La función se dará por terminada cuando el intérprete Python encuentre la primera línea cuya columna de indentación vuelva a ser la que corresponde al inicio de la cabecera.

Intuitivamente, una función puede entenderse como un proceso separado (también designado como *caja negra* debido a que esos procesos permanecen ocultos para quien usa la función desde el exterior)[2] que acepta *entradas o datos* (los *parámetros*), procesa esos

[2] La idea de *caja negra* referida a un proceso oculto nos remite a la inquietante película canadiense *Cube* (conocida como *Cubo* en nuestro país) de 1997, dirigida por *Vincenzo Natali* y protagonizada (entre otros) por *Nicole de Boer*. Varias personas que no se conocen entre sí aparecen (sin explicación alguna) encerradas en una habitación con puertas en cada pared, en el piso y en el techo. Cada una de esas puertas lleva a su vez a otra habitación igual a la primera, y los enloquecidos prisioneros deben buscar la forma de salir de ese "cubo mágico" sin morir en el intento… ya que muchas de esas habitaciones tienen trampas mortales. Se convirtió en una película de culto del cine de ciencia ficción – terror, y derivó en una secuela: *Cube 2: Hypercube* [año 2002] y una precuela: *Cube Zero* [año 2004].

datos para obtener uno o más *resultados o salidas*, y devuelve esos *resultados* mediante la instrucción *return*. La figura que sigue es un esquema de esta idea:

Figura 8: Esquema general y conceptual de una función.



El programa completo que resuelve el problema pero ahora usando funciones en Python, podría tener el siguiente aspecto (analizaremos con detalle en las secciones que siguen de esta ficha la forma en que trabajan los *parámetros* y la instrucción *return*):

```
__author__ = 'Cátedra de AED'


# subproblema 1: determinar el menor...
def menor(n1, n2, n3):
    if n1 < n2 and n1 < n3:
        mn = n1
    else:
        if n2 < n3:
            mn = n2
        else:
            mn = n3
    return mn


# subproblema 2: calcular cuadrado y cubo...
def calcular(mn):
    c2 = mn ** 2
    c3 = mn ** 3
    return c2, c3


# script principal...
# títulos y carga de datos...
print('Cálculo del cuadrado y el cubo del menor...')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))
c = int(input('Tercer número: '))

# procesos...
# invocar las funciones en el orden correcto de aplicación...
men = menor(a, b, c)
cuad, cubo = calcular(men)

# visualización de resultados...
print('Menor:', men)
print('Cuadrado:', cuad)
print('Cubo:', cubo)
```

La idea general es que por cada subrutina planteada en el diagrama de flujo o ideada por el programador se plantea una función, y la definición (el desarrollo de la cabecera y el bloque de acciones) de esas funciones debe hacerse en cualquier lugar que esté antes que el script desde donde se hacen las invocaciones a ellas: el intérprete lanzará un error si se intenta llamar a una función cuya definición aparezca más adelante (más abajo) de ese script en el código fuente. Por este motivo, el *script principal* (el bloque de código encargado de leer los valores de *a*, *b* y *c*, invocar a las funciones y luego mostrar los resultados), está *debajo* de las definiciones de ambas funciones.

En el *script principal* la línea men = menor(a, b, c) procede a *invocar* (o *llamar*) a la función *menor()*. Cuando decimos que una función es invocada o llamada, queremos decir que esa función es activada para que en ese momento se ejecuten las instrucciones que contiene en su bloque de acciones. Muy en general, cuando se invoca una función se escribe su nombre (*menor* en este caso) y luego entre paréntesis se escriben las variables o valores que se quieren enviar a la función a modo de datos. Esas variables se designan en general como *parámetros de la función*. En el programa anterior, al invocar a la función *menor()* se le están enviando tres parámetros: las variables *a*, *b* y *c* que se acaban de cargar por teclado en el script principal. La función toma los valores de esas variables, y automáticamente los asigna en las variables que ella misma tiene declaradas en su propia cabecera (en este caso, las variables *n1*, *n2* y *n3*). Como estas variables contienen ahora una copia de los valores que fueron enviados desde el script principal, la función puede operar con ellas y obtener el resultado esperado.

Cuando una función termina de ejecutarse, es común que disponga de uno o más valores obtenidos como resultado del proceso llevado a cabo por ella (en el caso del ejemplo, la variable *mn* con el valor del menor). Para que esos resultados sean *entregados* (o *retornados*) al punto desde el cual se llamó a la función, se usa la instrucción *return*. En la última línea de la función *menor()* puede verse la instrucción *return mn*, la cual esencialmente toma el valor de la variable *mn* y lo deja disponible para la instrucción desde la cual se haya invocado a la función. En este caso, recordemos que la función *menor()* fue invocada desde el script principal haciendo men = menor(a, b, c). Esto quiere decir que la función tomará una copia de las variables *a*, *b* y *c*, buscará el menor entre esos valores, y retornará ese menor (almacenado internamente en la variable *mn*) para que sea asignado a su vez en la variable *men* del script principal. Así, si se hace una invocación de la forma:

```
a, b, c, = 3, 6, 2
men = menor(a, b, c)
```

entonces la función *automáticamente* asignará los valores 3, 6 y 2 en las variables *n1*, *n2* y *n3* declaradas en su cabecera. Luego comprobará cuál de esas tres variables contiene el valor menor, el cual será asignado en la variable interna *mn*. Y finalmente, la instrucción *return mn* retornará el valor de *mn* que será asignado a su vez en la variable *men*. Por lo tanto, *men* quedará valiendo 2. La *Figura 9* (*página 197*) muestra en forma esquemática lo que ocurre al invocar a la función.

Es interesante notar que este proceso de toma de parámetros y retorno de valor ocurre en forma automática: es el propio Python el que copia los valores de las variables enviadas (*a*, *b*, y *c* en este caso) en las variables declaradas en la cabecera de la función (*n1*, *n2* y *n3*) y luego es también Python el que controla el retorno del resultado con la instrucción *return*. Y por cierto, esos mecanismos se activarán de la misma forma si la función es invocada para

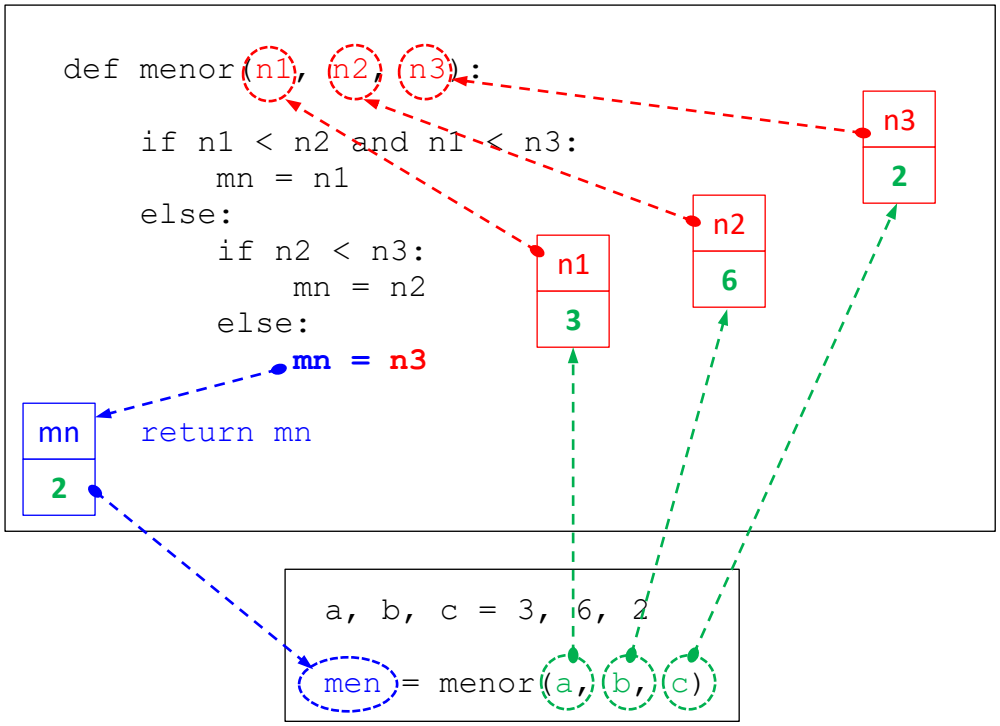
buscar el menor entre otras variables o valores constantes, como se ve en las invocaciones que siguen:

```
x, y, z = 30, 21, 75
m = menor(x, y, z)
# valor final de m: 21

p = menor(12, 100, 52)
# valor final de p: 12

t = 4
v = menor(5, t, 3)
# valor final de v: 4
```

Figura 9: Paso de parámetros a una función - Mecanismo de retorno en una función.



Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

[3] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: https://docs.python.org/3/.