Pontificia Universidad Católica del Perú

Estudios Generales Ciencias

1INF01 - Fundamentos de Programación

Guía de laboratorio #7

Ciclos Iterativos Anidados

3 de octubre de 2018

Índice general

Historial de Revisiones

Revisión	Fecha	Autor(es)	Descripción
1.0	11.05.2018	A.Melgar	Versión inicial.

Siglas

ASCII	American Standard Code for Information Interchange
ANSI	American National Standards Institute
EEGGCC	Estudios Generales Ciencias
IDE	Entorno de Desarrollo Integrado
OMS	Organización Mundial de la Salud
PUCP	Pontificia Universidad Católica del Perú
RAE	Real Academia Española

Capítulo 1

Guía de Laboratorio #7

1.1. Introducción

Esta guía ha sido diseñada para que sirva como una herramienta de aprendizaje y práctica para el curso de Fundamentos de Programación de los Estudios Generales Ciencias (EEGGCC) en la Pontificia Universidad Católica del Perú (PUCP). En particular se focaliza en el tema "Ciclos iterativos anidados".

Se busca que el alumno resuelva paso a paso las indicaciones dadas en esta guía contribuyendo de esta manera a los objetivos de aprendizaje del curso, en particular con la comprehensión de las estructuras algoritmicas iterativas anidadas y la instrucción `for`. Al finalizar el desarrollo de esta guía y complementando lo que se realizará en el correspondiente laboratorio, se espera que el alumno:

- Comprenda el funcionamiento de los ciclos iterativos anidados.
- Diseñe algoritmos que contengan ciclos iterativos anidados expresados en pseudocódigos y diagramas de flujo.
- Implemente algoritmos que contengan ciclos iterativos anidados usando un lenguaje de programación imperativo.
- Implemente ciclos iterativos usando instrucciones de ruptura de ciclos.

1.2. Materiales y métodos

Como herramienta para el diseño de pseudocódigos y diagramas de flujo se utilizará PSeInt[1]. El PSeInt deberá estar configurado usando el perfil PUCP definido por los profesores del curso. Como lenguaje de programación imperativo se utilizará el lenguaje ANSI C. Como Entorno de Desarrollo Integrado (IDE) para el lenguaje ANSI C se utilizará Dev C++[2]. No obstante, es posible utilizar otros IDEs como Netbeans y Eclipse.

Para poder comprender los temas y ejemplos que se revisarán en esta guía, es necesario que previamente haya leído y realizado los ejercicios de la guía #6 "Ciclo Iterativo con Entrada Controlada". Muchos de los ejemplos que se tratarán en este documento se contruyen usando el conocimiento obtenido de la guía #6.

1.3. Ciclos Iterativos Anidados

Como ya se vio anteriormente, las estructuras de control de flujo permiten que se modifique el flujo de ejecución de las instrucciones que forman parte de un programa. En particular, las estructuras algorítmicas iterativas

[1] http://pseint.sourceforge.net/
[2] http://www.bloodshed.net/dev/devcpp.html

permiten que los programas ejecuten un conjunto de instrucciones tantas veces como sea necesario. Esto es muy útil para resolver problemas que requieren realizar cálculos de forma repetitiva. Usando estructuras algoritmicas iterativas es posible, por ejemplo, recorrer un rango de valores $[a, b]$, generar los términos de una sucesión, hallar el valor de sumatorias y productorias, ejecutar cálculos reiterativos sobre un conjunto de datos, entre otros.

Recordar que:

- Las estructuras algoritmicas iterativas se clasifican:
  - ciclo iterativo con entrada controlada.
  - ciclo iterativo con salida controlada.
- En el ciclo iterativo con entrada controlada, la evaluación de la condición del ciclo se realiza antes de la ejecución del conjunto de instrucciones. Dependiendo de la condición, puede que el conjunto de instrucciones no se ejecute.
- En el ciclo iterativo con salida controlada, la evaluación de la condición del ciclo se realiza después de la ejecución del conjunto de instrucciones. Se ejecuta por lo menos 1 vez el conjunto de instrucciones.

A menudo existen situaciones en donde es necesario realizar cálculos en cada ciclo[3] dentro de una estructura algorítmica iterativa. A menudo, para realizar estos cálculos, es necesario utilizar otra estructura algorítmica iterativa. Esta situación configura lo que se conoce como ciclo iterativo anidado. Un ciclo iterativo anidado no es más que una estructura algorítmica iterativa, que dentro del bloque de instrucciones que ejecuta en su *bucle*, tiene otra estructura algorítmica iterativa.

En caso de ser necesario anidar dos estructuras algorítmicas iterativas, se tendría las siguientes opciones:

- Un ciclo iterativo con entrada controlada dentro de otro ciclo iterativo con entrada controlada.
- Un ciclo iterativo con salida controlada dentro de otro ciclo iterativo con salida controlada.
- Un ciclo iterativo con entrada controlada dentro de un ciclo iterativo con salida controlada.
- Un ciclo iterativo con salida controlada dentro de un ciclo iterativo con entrada controlada.

Pero es posible anidar más de dos estructuras algorítmicas iterativas por lo que las opciones de combinación que se disponen para escribir algoritmos y programas es vasta. ¿De cuántas maneras puede anidar 3 estructuras algorítmicas iterativas?

Una de las principales desventajas que se produce al utilizar ciclos iterativos anidados es el decremento del rendimiento del programa. Mientras más ciclos iterativos anidados se tengan, más lenta será la ejecución del programa. Es recomendable evitar su uso siempre que esto sea posible. Pero claro, existen situaciones en donde el uso de ciclos anidados se hace inevitable.

En conclusión, un ciclo anidado no es en sí una estructura de control de flujo nueva, sino la utilización de estructuras algorítmicas iterativas dentro de otras estructuras algorítmicas iterativas. Esta guía se enfocará en la anidación de ciclos iterativo con entrada controlada.

1.3.1. Ciclos iterativos anidados en pseudocódigo

Tal como se vio anteriormente, es posible anidar más de un ciclo iterativo con entrada controlada por lo que no es posible presentar todas las posibles combinaciones. A modo de ejemplo se presentará la anidación de 2 ciclos iterativos con entrada controlada. En la figura 1.1 se puede apreciar un ciclo iterativo anidado en pseudocódigo.

[3] también llamado de *bucle*, *loop* o iteración.

Mientras *condicion1* **Hacer**
  conjunto de instrucciones 1a;
  Mientras *condicion2* **Hacer**
    conjunto de instrucciones 2;
  FinMientras
  conjunto de instrucciones 1b;
FinMientras

Figura 1.1: Pseudocódigo: Ciclo iterativo anidado

Recordar que:

En un ciclo iterativo con entrada controlada se deben administrar 3 situaciones:

1. La inicialización de la(s) variable(s) de control que se utilizará(n) en la condición. Esto se realiza antes de iniciar el ciclo.
2. La definición de la condición, usando la(s) variable(s) de control.
3. El aseguramiento que la condición en algún momento llegue a finalizar. Normalmente se realiza en la última instrucción del ciclo.

En general algunas consideraciones se deben tener en cuenta al trabajar con ciclos anidados. Cada anidación agrega complejidad al algoritmo por lo que es importante manterner nombres adecuados de variables. Por ejemplo, cuando se usan ciclos anidados que son controlados por contadores, se suele utilizar i, j, k, l, como nombres de las variables de control. Por lo general i corresponde a la variable de control del ciclo más externo.

1.3.2. Ciclos iterativos anidados en diagrama de flujo

En la figura 1.2 se puede apreciar un ciclo iterativo con entrada controlada anidado representado usando el diagrama de flujo. En el se puede apreciar el ciclo interno formado por el bloque de instrucciones luego de la condición 2 (bloque en rojo). El ciclo externo (bloque en verde) está formado por el bloque de instrucciones luego de la condición 1, incluyendo el ciclo iterativo interno.

1.3.3. Ciclos iterativos anidados en ANSI C

En el programa 1.1 se puede apreciar un ciclo iterativo con una entrada controlada anidado implementado en ANSI C. A pesar que para el compilador de C le es indifente la forma en que se escribe el código, es importante que este se encuentre identado[4] para que sea entendible al ojo humano. Con una correcta identación el ciclo más interno se distingue fácilmente.

Programa 1.1: ANSI C: Ciclo iterativo anidado

```
1
2       while (condicion1){
3               conjunto de instrucciones1a;
4               while (condicion2){
5                       conjunto de instrucciones2;
6               }
7               conjunto de instrucciones1b;
8       }
```

[4]Movimiento de un bloque de instrucciones a la derecha, muy similar al sangrado o sangría de los documentos de texto

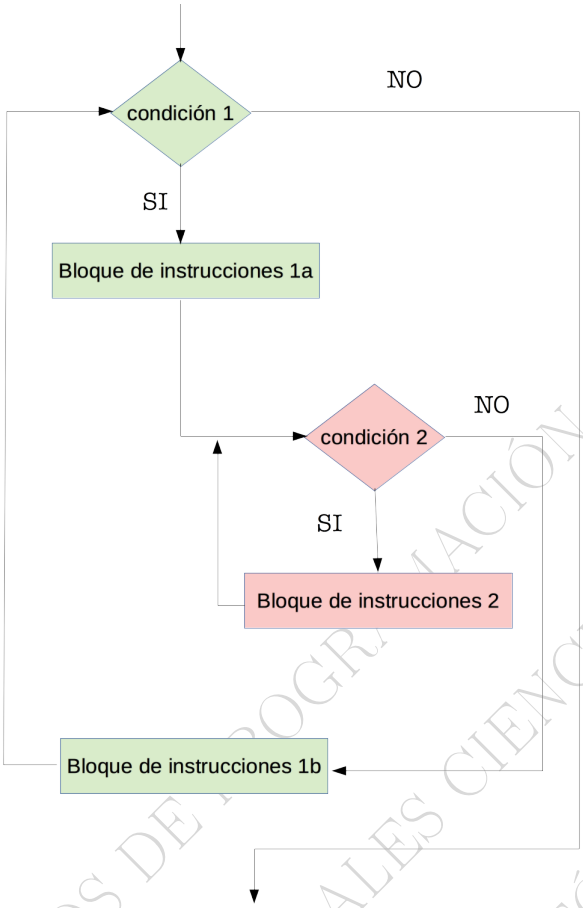

Figura 1.2: Diagrama de Flujo: Ciclo iterativo anidado

1.4. El triángulo de Pascal

Blaise Pascal fue un matemático francés que popularizó este triángulo de números. Está formado por una serie de números distribuidos en infinitas filas y alineados de una forma particular. En la figura 1.3 se puede apreciar las primeras 5 filas del triángulo de Pascal.

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Figura 1.3: Triángulo de Pascal

El triángulo de Pascal se construyen de la siguiente forma:

- La primera fila posee solamente un 1.
- La segunda posee dos 1, a izquierda y derecha del uno anterior: 1 1.

- La tercera fila se construye de la siguiente manera: se suman los dos números de la fila anterior y se coloca el resultado (en este caso 2 que resulta de sumar $1 + 1$), debajo del espacio que dejan dichos números, y a izquierda y derecha se colocan dos 1. El resultado final es: 1 2 1 .
- La filas posteriores, se construyen de forma muy similar a la tercera: debajo de cada espacio entre dos números de la fila anterior se escribe la suma de dichos números, y a izquierda y derecha se coloca dos 1. Por ejemplo, la fila 3 quedaría así: 1 3 3 1, la fila 4 así: 1 4 6 4 1.

Con el triángulo de Pascal es posible calcular las potencias de binomios $(a + b)^n$, donde a y b son variables cualquiera y n corresponde con el exponente que define la potencia. Los números de cada fila del triángulo de Pascal está relacionada con los coeficientes binomiales, de esta forma la primera fila (1) contiene el coeficiente para el binomio $(a + b)^0$, la segunda fila (1 1) contiene los coeficientes para el binomio $(a + b)^1$, la tercera fila (1 2 1) contiene los coeficientes para el binomio $(a + b)^2$, la cuarta fila (1 3 3 1) contiene los coeficientes para el binomio $(a + b)^3$ y así sucesivamente.

Recordar que:

$$
\begin{aligned}
&(a+b)^0 = 1\\
&(a+b)^1 = 1a + 1b\\
&(a+b)^2 = 1a^2 + 2ab + 1b^2\\
&(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3\\
&\ldots\\
&(a+b)^n = k_1a^n + k_2a^{n-1}b + k_3a^{n-2}b^2 + \cdots + k_{n-1}ab^{n-1} + k_nb^n
\end{aligned}
$$

¿De qué manera se puede automatizar la generación de las filas del triángulo de Pascal? Para poder responder a esta pregunta es necesario recordar el teorema del binomio. Este teorema estable que:

$$(x + y)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} x^{n-k}y^k$$

La misma fórmula se puede expresar usando el combinatorio de la siguiente manera:

$$(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k}y^k$$

Recordar que:

Se define número combinatorio como el valor numérico de las combinaciones ordinarias (sin repetición) de un conjunto de p elementos tomados en grupos de r, siendo p y r dos números enteros y positivos tales que $p \geq r$. Matemáticamente, un número combinatorio se expresa como:

$$C_r^p = \binom{p}{r} = \frac{p!}{r! \times (p-r)!}$$

En realidad la parte de la sumatoria que interesa para la generación del triángulo de Pascal es la relacionada a los coeficientes, es decir $\sum_{k=0}^{n} \binom{n}{k}$. La unica consideración que debemos tener es que en realidad no se realizará una suma en lugar de sumar se imprimirá cada término de la sumatoria. En la figura 1.4 se puede apreciar el triángulo de Pascal presentado anteriormente pero con los número expresados en función de los números combinatorios.

Sabiendo esto, ¿Cómo imprimir un triángulo de Pascal dada una determinada cantidad n de filas? Se propondrá una alternativa de solución a este problema adoptando una estrategia *bottom-up*, es decir de "de abajo hacia arriba". Primero se calculará un número combinatorio, posteriormente se generará una determinada fila del

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

Figura 1.4: Coeficientes del triángulo de Pascal

triángulo de Pascal para finalmente presentar un algoritmo que imprima las n primeras filas del triángulo de Pascal. Se utilizarán estructuras algorítmicas iterativas anidadas. La solución se presentará tanto en diagrama de flujo como en pseudocódigo.

1.4.1. Cálculo del número combinatorio

Vamos a "olvidarnos" por un instante del problema de la impresión de las filas del triángulo de Pascal para enfocarnos en el cálculo del número combinatorio. Como podrá recordar $C_r^p = \binom{p}{r} = \frac{p!}{r! \times (p-r)!}$. Entonces el problema se reduce al cálculo de 3 números factoriales dado 2 variables como entradas (en este caso p y r con $p \geq r$). Los números factoriales a calcular serían: $p!$, $r!$ y $(p-r)!$.

Anteriormente se ha visto cómo realizar el cálculo de el factorial de un número. Básicamente es la productoria $\prod_{i=1}^{n} i$. La implementación de esta productoria se realiza con un ciclo iterativo con entrada controlada usando un contador para recorrer desde i hasta n y una variable que almacene la productoria. Al finalizar el ciclo, la variable que almacena la productoria tendría el factorial requerido.

Pero, ¿Cómo calcular los 3 números factoriales requeridos? Una primera idea podría ser la utilización de 3 ciclos iterativos secuenciales, uno detrás de otro. En el primero se recorre con un contador hasta p, luego en el segundo se recorre hasta r y finalmente en el tercer ciclo se recorre hasta $(p-r)$. Dentro de cada ciclo iterativo se realizan los cálculos respectivos de la productoria. A pesar que esta idea permitiría obtener los factoriales requeridos, no es una alternativa de solución eficiente pues se tendría 3 ciclos iterativos, lo cual es complicado de entender y sobre todo complica el mantenimiento del programa.

Entonces, ¿Cómo realizar el cálculo en una sola iteración? Se sabe que $p \geq r$, por lo tanto se puede deducir que $p \geq (p-r)$. De la definición del factorial se puede afirmar que $p! \geq r!$ y $p! \geq (p-r)!$. En términos prácticos significa que recorriendo el número mayor, en este caso p, se pueden calcular los 3 factoriales. Esto es posible pues los rangos de los otros números son menores y son un subconjunto de $[1..p]$. En la propuesta se utilizará una variable denominada *contador* para recorrer el rango de $[1..p]$. Se utilizarán 3 variables para almacenar los factoriales $fact\_p$ que almacenará el factorial de p, $fact\_r$ que almacenará el factorial de r y $fact\_p\_menos\_r$ que almacenará el factorial de $(p-r)$. Todos estas variables se inicializaran en 1.

El cálculo de $fact\_p$ es similar a los casos vistos anteriormente. La diferencia está en el calculo de $fact\_r$ y $fact\_p\_menos\_r$. En el caso de $fact\_r$ solo se realizará la productoria si es que $contador \leq r$. En el caso de $fact\_p\_menos\_r$ solo se realizará la productoria si es que $contador \leq (p-r)$. Todo esto se realiza en el mismos ciclo iterativo.

Al final del ciclo iterativo se tendrán los valores requeridos de los factoriales en $fact\_p$, $fact\_r$ y $fact\_p\_menos\_r$ respectivamente. Usando estas variables se puede calcular fácilmente el $\binom{p}{r}$. En la figura 1.5 se puede apreciar un algoritmo representado en diagrama de flujo que lee dos números e imprime el combinatorio de ambos. No se ha realizado la validación de $p \geq r$, $p \geq 0$ y $r \geq 0$ pues a pesar que en este algoritmos son leídos, en el siguiente p y r serán tomados de datos de otro ciclo iterativo.

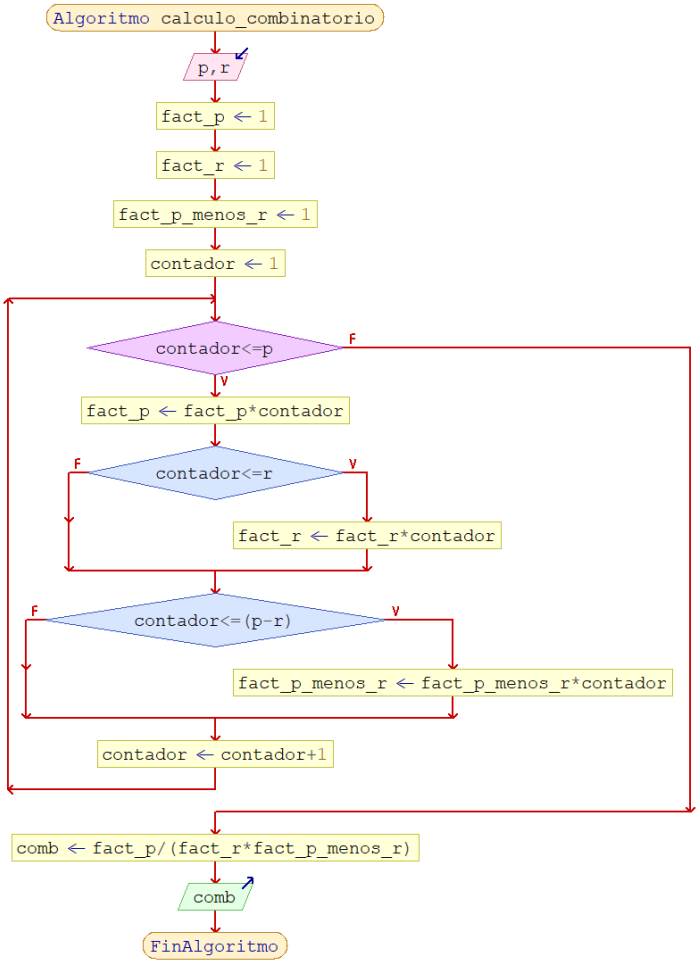

Figura 1.5: Diagrama de flujo: Cálculo del número combinatorio

Para poner en práctica (*A*)

- ¿Qué cambios debería realizar en el diagrama de flujo si en lugar de calcular los factoriales en un solo ciclo iterativo utiliza tres ciclos iterativos secuenciales?
- ¿Qué cambios debería realizar en el diagrama de flujo si se requiere verificar que $p \geq r$, $p \geq 0$ y $r \geq 0$?
- Exprese este mismo algoritmo en pseudocódigo.
- Implemente este algoritmo usando el lenguaje ANSI C.

1.4.2. Cálculo de una fila del triángulo de Pascal

Ahora que se sabe cómo calcular el combinatorio de dos números, se procederá a imprimir una determinada fila del triángulo de Pascal. Recuerde que se intenta resolver la sumatoria $\sum_{k=0}^{n} \binom{n}{k}$, en donde n representa el exponente del binomio así como el número de la fila del triángulo. Recordemos además que en lugar de acumular la suma, se estarán imprimiendo los valores de la sumatoria.

Dado un número de fila n del triángulo de Pascal esta fila tendrá n números que se calculan como sigue:

si $n = 1 \Rightarrow p = 0 \Rightarrow \binom{0}{0}$
si $n = 2 \Rightarrow p = 1 \Rightarrow \binom{1}{0}\ \binom{1}{1}$
si $n = 3 \Rightarrow p = 2 \Rightarrow \binom{2}{0}\ \binom{2}{1}\ \binom{2}{2}$
si $n = 4 \Rightarrow p = 3 \Rightarrow \binom{3}{0}\ \binom{3}{1}\ \binom{3}{2}\ \binom{3}{3}$
si $n = k \Rightarrow p = k - 1 \Rightarrow \binom{p}{0}\ \binom{p}{1}\ \binom{p}{2}\ \binom{p}{3} \cdots \binom{p}{p-1}\ \binom{p}{p}$

Como se puede apreciar dado un número de fila n se puede calcular fácilmente el número p del combinatorio $\binom{p}{r}$, $p = n - 1$. Además se puede apreciar que r varía en el rango $[0..p]$. Como ya se vio anteriormente, un rango se puede recorrer con una estructura iterativa con entrada controlada. Con p y r se puede calcular el combinatorio usando la propuesta de la sección anterior. Como la propuesta para el cálculo se basa en una iterativa con entrada controlada y como para recorrer el rango $[0..p]$ también se usará una iterativa con entrada controlada, estamos frente a un caso de estructuras iterativas anidadas.

En el ciclo más externo de la estructura anidada tiene por objetivo recorrer el rango $[0..p]$. Para esto se lee la variable n la cual representará el número de la fila. Para iterar se usará la variable $cont\_n$ que será inicializado con 0 e irá hasta $n - 1$ que en nuestro análisis representa el valor p del combinatorio. La condición de esta iteración es $cont\_n < n$, no toma n sino llega hasta $n - 1$. Se hubiera podido colocar como condición también $cont\_n \leq (n - 1)$.

En cada iteración se calcula el valor de p y r. A p se le asigna $n-1$ y a r el valor en curso del $cont\_n$. Recuerde que p permanece fijo[5] en la impresión de la fila y r varía. Con estos valores se calcula el número combinatorio $\binom{p}{r}$.

Para el cálculo del número combinatorio, antes de iniciar el ciclo interno, se inicializan las variables que contendrán el factorial ($fact\_p$, $fact\_r$, $fact\_p\_menos\_r$) con 1 así como el contador del ciclo interno ($cont_fact$[6]). El ciclo interno es en esencia el mismo de la sección anterior.

En la figura 1.6 se puede apreciar un algoritmo representado en diagrama de flujo que lee un número de fila del triángulo de Pascal e imprime los números que conforman la fila. No se ha realizado la validación de $n \geq 0$ pues a pesar que en este algoritmos n es leído, en el siguiente algoritmo n será tomado como dato de otro ciclo iterativo.

Para poner en práctica (B)

- Si en el ciclo iterativo más externo cambia la condición de $cont\_n < n$ por $cont\_n \leq (n - 1)$, ¿El algoritmo retorna los mismos resultados?
- Si la instrucción $p \leftarrow n - 1$ es movida para fuera del ciclo iterativo más externo, ¿El algoritmo retorna los mismos resultados?
- Exprese este mismo algoritmo en pseudocódigo.
- Implemente este algoritmo usando el lenguaje ANSI C.

1.4.3. Impresión del triángulo de Pascal

Hasta el momento se ha podido imprimir los números de determinada fila n del triángulo de Pascal. ¿Cómo hacer para imprimir k filas? Si se desean imprimir k filas se tendrá que utilizar el algoritmo anterior k veces, usando un contador de filas para recorrer el rango $[1..k]$. Estamos nuevamente frente a una situación en donde se tiene un ciclo iterativo anidado.

Se utilizará la variable $total\_filas$ para solicitar al usuario el número de filas que se requiere imprimir. Se utilizará además la variable $cont\_filas$ para recorrer el rango $[1..total\_filas]$. Para esto se utilizará un ciclo iterativo con entrada controlada usando la condición $cont\_filas \leq total\_filas$. Lo que se hace en cada ciclo

[5] Dado que p tiene siempre el mismo valor en la iteración, este podría ser inicializado fuera del ciclo iterativo. En esta alternativa de solución se optó por mantenerlo dentro de la iteración para no confundir en la solución final.

[6] En el algoritmo anterior está variable se llamaba *contador*, es este algoritmo se ha optado por cambiar de nombre pues se utilizan dos contadores en dos ciclos diferentes. De esta forma queda evidente que se refiere al contador para el cálculo del factorial.

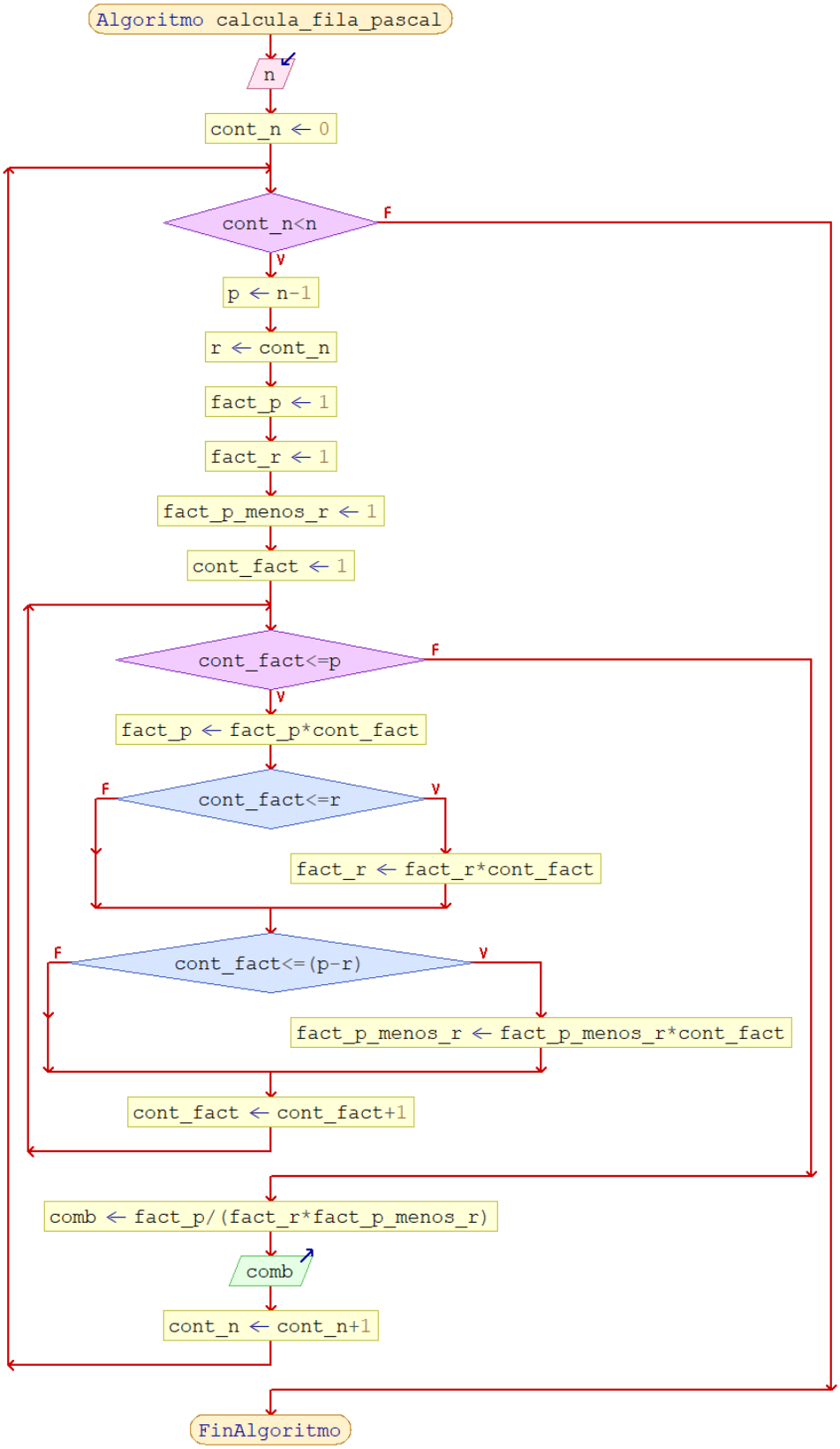

Figura 1.6: Diagrama de flujo: Cálculo de una fila del triángulo de Pascal

es invocar al algoritmo anterior haciendo que en cada iteración n sea igual a $cont\_filas$. Recuerde que en el algoritmo anterior n representaba al número de fila que es justamente lo que se está recorriendo en este algoritmo.

En la figura 1.7 se puede apreciar el pseudocódigo que permite imprimir las primeras k filas del triángulo de Pascal. En la 1.7 se puede apreciar la salida obtenida por este pseudocódigo cuando el usuario ingresa el valor de 4 a la variable $total\_filas$.

```
1  Algoritmo triangulo_de_pascal
2      Escribir 'Ingrese cantidad de filas: '
3      Leer total_filas
4      cont_filas<-1
5      Mientras  cont_filas<=total_filas Hacer
6          Escribir 'Fila: ', cont_filas
7          n<-cont_filas
8          cont_n <- 0
9          Mientras cont_n<n Hacer
10             p <- n-1
11             r <- cont_n
12             fact_p <- 1
13             fact_r <- 1
14             fact_p_menos_r <- 1
15             cont_fact <- 1
16             Mientras cont_fact<=p Hacer
17                 fact_p <- fact_p*cont_fact
18                 Si cont_fact<=r Entonces
19                     fact_r <- fact_r*cont_fact
20                 FinSi
21                 Si cont_fact<=(p-r) Entonces
22                     fact_p_menos_r <- fact_p_menos_r*cont_fact
23                 FinSi
24                 cont_fact <- cont_fact+1
25             FinMientras
26             comb <- fact_p/(fact_r*fact_p_menos_r)
27             Escribir comb
28             cont_n <- cont_n+1
29         FinMientras
30         cont_filas <- cont_filas+1
31     FinMientras
32 FinAlgoritmo
```

Figura 1.7: Pseudocódigo: Impresión del triángulo de Pascal

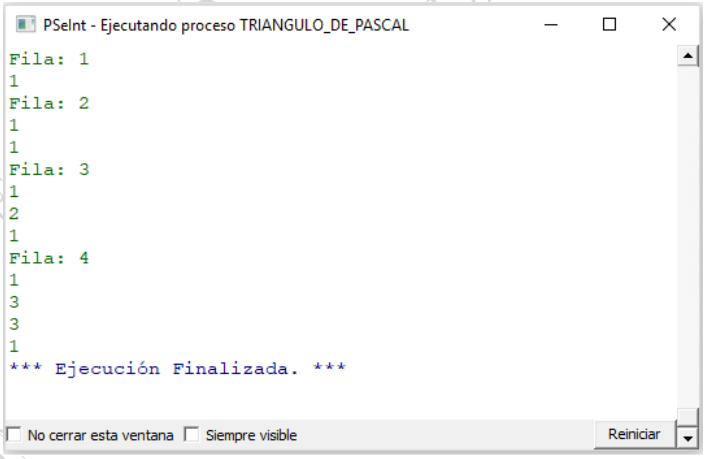

Figura 1.8: Salida: Impresión del triángulo de Pascal

Para poner en práctica (*C*)

- Implemente este algoritmo usando el lenguaje ANSI C.
- ¿Qué alteraciones debería realizar en el algoritmo si se desea imprimir solamente las filas pares del triángulo de Pascal?
- ¿Qué alteraciones debería realizar en el algoritmo si se desea sumar los números del triángulo de Pascal?

1.5. La instrucción for

El ciclo iterativo con entrada controlada se puede implementar en el ANSI C mediante la instrucción `while`. Tal como se puede apreciar en los ejemplos realizados, siempre que se realiza un ciclo iterativo con entrada controlada debe de i) inicializarse la(s) variable(s) de control de flujo, luego ii) elaborar una condición para el control del flujo y finalmente iii) actualizar la(s) variable(s) de control para asegurarse que en algún momento la condición de control de flujo falle y se termine el *bucle*. La inicialización de la(s) variable(s) de control de flujo se suele realizar antes de iniciar el ciclo y la actualización se suele realizar en la última instrucción del ciclo, tal como se puede apreciar en el programa 1.2.

Programa 1.2: ANSI C: instrucción while

```
1  inicialización de variable(s) de control;
2  while (condición de control de flujo){
3      conjunto de instrucciones;
4      actualización de variable(s) de control
5  }
```

Debido a que estas 3 situaciones (inicialización, condición, actualización) están siempre presentes en un ciclo iterativo con entrada controlada, el lenguaje C permite que se pueda escribir, de forma agrupada en una sola instrucción. Esta instrucción es la instrucción `for`. En el programa 1.3 se puede apreciar la sintaxis de la instrucción `for`. En la práctica funciona de la misma manera que la instrucción `while`.

Programa 1.3: ANSI C: instrucción for

```
1  for(inicialización de variable(s) de control;condición de control de flujo;actualización de variable(s) de control)
2      conjunto de instrucciones;
```

1.5.1. Implementación del triángulo de Pascal usando for

En el programa 1.4 se presenta la implementación del algoritmo trabajado previamente para el problema del triángulo de Pascal usando ANSI C pero implementando los ciclos iterativos usando la instrucción `for`. Si usted ha realizado la implementación de este algoritmo usando la instrucción `while` podrá apreciar que al usar la instrucción `for` la cantidad de instrucciones se reducen, esto ocurre pues la inicialización y la actualización de la variable de control ocurren en la misma línea. Además, para algunos codificadores, se aprecia mejor el control de flujo del ciclo iterativo.

Hay algunos aspectos de la implementación del programa 1.4 que vale la pena comentar. En primer lugar en la línea 7 se puede observar la siguiente instrucción `for`.

```
    for (cont_filas = 1; cont_filas <= total_filas; cont_filas++){
    ...
}
```

Usando la instrucción `for` fácilmente se puede apreciar como se realiza el control del ciclo. El ciclo se controla con la variable $cont\_filas$ la cual se inicializa en 1 y se incremente en 1 al final de cada ciclo. Antes de ejecutar el *bucle*, se evalua la condición $cont\_fila \leq total\_filas$. El ciclo solo se ejecutará si esta condición se cumple, es decir si el valor de la expresión es 1 (se hace *verdadera*).

Esta instrucción `for` equivale a la siguiente instrucción `while`. La principal desventaja del uso de la instrucción `while` es que la inicialización y la actualización de la variable de control se encuentran en diferentes líneas, haciendo la comprehensión del flujo una tarea un poco más compleja que cuando se utiliza la instrucción `for`. Esto se torna más evidente cuando se utilizan ciclos iterativos anidados.

```
cont_filas = 1;
while (cont_filas <= total_filas){
...
cont_filas++;
}
```

Programa 1.4: Triángulo de Pascal implementado en ANSI C usando for

```
1  #include <stdio.h>
2
3  int main() {
4      int total_filas, cont_filas;
5      printf("Ingrese cantidad de filas: ");
6      scanf("%d", &total_filas);
7      for (cont_filas = 1; cont_filas <= total_filas; cont_filas++) {
8          int n, cont_n;
9          printf("Fila %d\n", cont_filas);
10         for (n = cont_filas, cont_n = 0; cont_n < n; cont_n++) {
11             int p, r, cont_fact, fact_p, fact_r, fact_p_menos_r, comb;
12             p = n - 1;
13             r = cont_n;
14             fact_p = fact_r = fact_p_menos_r = 1;
15             for (cont_fact = 1; cont_fact <= p; cont_fact++) {
16                 fact_p *= cont_fact;
17                 if (cont_fact <= r)
18                     fact_r *= cont_fact;
19                 if (cont_fact <= (p - r))
20                     fact_p_menos_r *= cont_fact;
21             }
22             comb = fact_p / (fact_r * fact_p_menos_r);
23             printf("%d ", comb);
24         }
25         printf("\n");
26     }
27     return 0;
28 }
```

En la línea 10 se puede apreciar el siguiente ciclo `for`. Observe la expresión de inicialización de varibles de control. ¿Qué de diferente tiene con las demás instrucciones `for`?

```
...
for (n = cont_filas, cont_n = 0; cont_n < n; cont_n++) {
...
}
```

Como podrá observar, la diferencia radica que en este ciclo `for` se está realizando la inicialización de 2 variables de control (n y $cont\_n$). Esto es posible realizar tanto en la inicialización como en la actualización, basta separar las expresiones por una coma (`,`). Ambas inicializaciones se ejecutan en la misma instrucción.

Otro aspecto interesante para comentar en este programa aparece en la línea 14. Lo que sucede en esta expresión es lo que se conoce como asignación múltiple. El lenguaje C permite que se pueda asignar un valor a varias variables en una única instrucción. En la asignación múltiple, el operando que está más a la derecha puede ser un valor constante (como 1), una variable (como x) o una expresión (como $x + 5$), los demás operandos necesariamente deben ser variables pues se les asignará un valor.

```
fact_p = fact_r = fact_p_menos_r = 1;
```

En la línea 16 se tiene la siguiente instrucción. ¿Qué hace esta instrucción?

```
fact_p *= cont_fact;
```

Estamos frente a lo que en C se conoce como operadores de asignación. El operador `*=` realiza una multiplicación y asignación al mismo tiempo, de esta manera `fact_p *= cont_fact;` es equivalente a `fact_p =`

`fact_p*cont_fact;`. Existen varios operadores de asignación que soporte el lenguaje C, entre ellos: `+=`, `-=`, `*=`, `/=` y `%=`.

1.6. El cuadrado mágico

Un cuadrado mágico es una estructura de números organizados en filas y columnas de igual tamaño de tal forma que todas las filas, todas las columnas y las diagonales sumen el mismo número. Este número se denomina la constante mágica.

Dado que un cuadrado mágico tiene el mismo número n de filas y de columnas, se suelen colocar números entre 1 y n^2 en el cuadrado. De esta forma la constante mágica se calcula como $C_n = \frac{n(n^2+1)}{2}$.

Tabla 1.1: Cuadrado mágico

a	b	c
d	e	f
g	h	i

Si el cuadrado que se encuentra en la tabla 1.1 fuese mágico, se deberían cumplir ciertas condiciones:

- Todas las filas suman lo mismo y la suma es igual a la constante mágica. Es decir $a+b+c = d+e+f = g+h+i = C_n$.
- Todas las columnas suman lo mismo y la suma es igual a la constante mágica. Es decir $a+d+g = b+e+h = c+f+i = C_n$.
- Las diagonales suman lo mismo y la suma es igual a la constante mágica. Es decir $a+e+i = c+e+g = C_n$.

Si se tuviera un cuadrado mágico de orden 3, entonces la constante mágica $C_3 = \frac{3(3^2+1)}{2} = \frac{3(9+1)}{2} = 15$. Una posible solución a un cuadrado mágico se puede apreciar en la tabla 1.2.

Tabla 1.2: Cuadrado mágico de orden 3

8	1	6
3	5	7
4	9	2

¿De qué manera se pueden generar cuadrados mágicos de orden 3 usando el lenguaje C? Se pueden generar de diversas maneras, en esta guía nos basaremos en la técnica de la fuerza bruta. La técnica de la fuerza bruta consiste en un método directo para resolver un problema, por lo general basada directamente en el enunciado del problema y en las definiciones de los conceptos involucrados **[?]**. Fuerza bruta implica la capacidad de un computador para realizar cálculos, no suelen ser algoritmos que utilicen inteligencia en su diseño.

Usando la técnica de la fuerza bruta lo que se realizará es la generación de todas las posibles combinaciones de cuadrados de orden 3 y para cada uno verificar las condiciones de problema. Se imprimirán los cuadrados que cumplan la condición.

Pero, ¿cómo se generan todas las posibles combinaciones? Para esto usaremos el ciclo iterativo y la anidación de estas iteraciones tantas veces como sea necesario. Procederemos a analizar el problema e ir proponiendo alternativas de solución. Iniciaremos generando filas, luego filas con números diferentes, posteriormente generaremos cuadrados y aplicaremos a estos la condición del cuadrado mágico.

1.6.1. Imprimiendo filas

Analicemos la tabla 1.1. La primera fila está formada por los números a, b y c. Se sabe que tanto a, b como c varía en el rango [1..9]. Entonces lo que se requiere hacer es iterar para cada número del 1 al 9. Pero para

generar todas las posibles combinaciones, estas iteraciones deben estar anidadas. En el programa 1.5 se puede apreciar una propuesta para generar todas las posibles filas.

Programa 1.5: Imprimiendo filas

```
#include <stdio.h>

int main() {
    int a,b,c;

    for(a=1;a<=9;a++)
        for(b=1;b<=9;b++)
            for(c=1;c<=9;c++)
                printf("%d %d %d\n", a,b,c);
    return 0;
}
```

La salida de este programa generaría lo siguiente:

```
1 1 1
1 1 2
1 1 3
1 1 4
.....
.....
.....
9 9 7
9 9 8
9 9 9
```

Para poner en práctica (*D*)

- Implemente el algoritmo usando la instrucción `while`.
- Como podrá observar la impresión inicia con la fila 1, 1, 1 y finaliza con la fila 9, 9, 9. ¿Qué cambios deberá realizar en el programa para que la impresión inicie con la fila 9, 9, 9 y finalice con la fila 1, 1, 1?

1.6.2. Imprimiendo filas con números diferentes

Si observa detalladamente la salida del programa, verá que existen filas para las cuales los números que las componen se repiten. Por ejemplo la segunda fila es 1, 1, 2, se tienen dos veces 1. Para el caso del cuadrado mágico, estas filas no formarían parte de la solución. El programa debe de filtrarlas, es decir no las debe imprimir.

¿Cómo se puede realizar el filtro? Para esto se utilizará una instrucción selectiva. Solo se imprimirá si se cumple la condición de que todas los números sean diferentes.

¿Cómo establecemos que todos los números sean diferentes? En el programa la variable a representa al número a, la variable b al número b y la variable c al número c de la fila. Debe cumplirse que $a \neq b$, $b \neq c$ y $a \neq c$. Esta condición la podemos expresar en C de la siguiente manera $a! = b$ && $b! = c$ && $a! = c$. Recuerde que en C `!=` es el operador de comparación y retorna 1 si los operandos son diferentes. `&&` es el operador de conjunción y retorna 1 si los operandos son diferentes de 0.

En el programa 1.6 se puede apreciar un programa que imprime las filas que cumplen la condición previamente establecida.

Programa 1.6: Imprimiendo filas con números diferentes

```
#include <stdio.h>
```

```
2
3  int main() {
4      int a,b,c;
5
6      for(a=1;a<=9;a++)
7          for(b=1;b<=9;b++)
8              for(c=1;c<=9;c++)
9                  if (a!=b && b!=c && a!=c)
10                   printf("%d %d %d\n", a,b,c);
11     return 0;
12 }
```

La salida de este programa generaría lo siguiente:

```
1 2 3
1 2 4
1 2 5
1 2 6
1 2 7
1 2 8
1 2 9
1 3 2
.....
.....
.....
```

1.6.3. Imprimiendo "cuadrados"

Ya se ha conseguido imprimir una fila de números, ¿cómo se podrían generar los números para el cuadrado completo? Hasta el momento se han utilizado 3 ciclos iterativos para generar una fila, si se desean generar 3 filas se deberá utilizar 9 ciclos iterativos anidados. En el programa 1.7 se puede apreciar un programa que imprime las 3 filas de un posible cuadrado mágico. Las variables a, b, c representan a la primera fila, las variables d, e, f representan a la segunda fila y las variables g, h, i a la tercera fila. Se ha impreso unos asteriscos antes de cada cuadrado para poder diferenciarlos en la salida.

Programa 1.7: Imprimiendo "cuadrados"

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b, c, d, e, f, g, h, i;
5
6      for (a = 1; a <= 9; a++)
7          for (b = 1; b <= 9; b++)
8              for (c = 1; c <= 9; c++)
9                  for (d = 1; d <= 9; d++)
10                     for (e = 1; e <= 9; e++)
11                         for (f = 1; f <= 9; f++)
12                             for (g = 1; g <= 9; g++)
13                                 for (h = 1; h <= 9; h++)
14                                     for (i = 1; i <= 9; i++) {
15                                         printf("*****\n");
16                                         printf("%d %d %d\n", a, b, c);
17                                         printf("%d %d %d\n", d, e, f);
18                                         printf("%d %d %d\n", g, h, i);
19                                     }
20     return 0;
21 }
```

La salida de este programa generaría lo siguiente:

```
.....
```

```
.....
*****
1 1 1
1 1 4
7 9 2
*****
1 1 1
1 1 4
7 9 3
.....
.....
```

Para poner en práctica (E)

- Implemente el algoritmo usando la instrucción `while`. Para este problema en particular, ¿conviene utilizar la instrucción `while` o `for`?

1.6.4. Imprimiendo "cuadrados" con números diferentes

Si observa la salida del programa anterior, verá que dentro de un mismo cuadrado, existen números que se repiten. Se debe aplicar la misma lógica que se aplicó cuando se filtraron las filas que tenían números repetidos. La principal diferencia ahora es que no se tienen 3 variables sino 9 y se debe garantizar que todos sean diferentes entre todos. En el programa 1.8 se apreciar entre la línea 15 y la 24 la condición para filtrar los cuadrados con números diferentes. Lo extensa de la condición se debe a que se deben colocar la expresión lógica que garantice que todos las variables sean diferentes entre sí.

Programa 1.8: Imprimiendo "cuadrados" con números diferentes

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b, c, d, e, f, g, h, i;
5
6      for (a = 1; a <= 9; a++)
7        for (b = 1; b <= 9; b++)
8          for (c = 1; c <= 9; c++)
9            for (d = 1; d <= 9; d++)
10             for (e = 1; e <= 9; e++)
11               for (f = 1; f <= 9; f++)
12                 for (g = 1; g <= 9; g++)
13                   for (h = 1; h <= 9; h++)
14                     for (i = 1; i <= 9; i++)
15                       if (a != b && a != c && a != d && a != e && a != f && a != g &&
16                           a != h && a != i &&
17                           b != c && b != d && b != e && b != f && b != g && b != h &&
18                           b != i &&
19                           c != d && c != e && c != f && c != g && c != h && c != i &&
20                           d != e && d != f && d != g && d != h && d != i &&
21                           e != f && e != g && e != h && e != i &&
22                           f != g && f != h && f != i &&
23                           g != h && g != i &&
24                           h != i) {
25                           printf("*****\n");
26                           printf("%d %d %d\n", a, b, c);
27                           printf("%d %d %d\n", d, e, f);
28                           printf("%d %d %d\n", g, h, i);
29                       }
30     return 0;
31 }
```

La salida de este programa generaría lo siguiente:

```
. . . . .
. . . . .
*****
1 3 7
8 9 6
5 4 2
*****
1 3 7
9 2 4
5 6 8
. . . . .
. . . . .
```

1.6.5. Imprimiendo "cuadrados" mágicos

Ahora que se ha generada todas las posibles combinaciones de cuadrados que contienen números diferentes hay que agregarle la condicion del problema. Un cuadrado mágico es aquel en el que la suma de las todas las filas, columnas y diagonales es igual a la constante mágica. En este caso 15. En el programa 1.9 se aprecia la condición del cuadrado mágico entre las líneas 25 y 32.

Programa 1.9: Imprimiendo "cuadrados" mágicos

```
1  #include <stdio.h>
2
3  int main() {
4      int a, b, c, d, e, f, g, h, i;
5
6      for (a = 1; a <= 9; a++)
7        for (b = 1; b <= 9; b++)
8          for (c = 1; c <= 9; c++)
9            for (d = 1; d <= 9; d++)
10             for (e = 1; e <= 9; e++)
11               for (f = 1; f <= 9; f++)
12                 for (g = 1; g <= 9; g++)
13                   for (h = 1; h <= 9; h++)
14                     for (i = 1; i <= 9; i++) {
15                       if (a != b && a != c && a != d && a != e && a != f && a != g &&
16                           a != h && a != i &&
17                           b != c && b != d && b != e && b != f && b != g && b != h &&
18                           b != i &&
19                           c != d && c != e && c != f && c != g && c != h && c != i &&
20                           d != e && d != f && d != g && d != h && d != i &&
21                           e != f && e != g && e != h && e != i &&
22                           f != g && f != h && f != i &&
23                           g != h && g != i &&
24                           h != i) {
25                           if ((a + b + c) == 15 &&
26                                   (d + e + f) == 15 &&
27                                   (g + h + i) == 15 &&
28                                   (a + d + g) == 15 &&
29                                   (b + e + h) == 15 &&
30                                   (c + f + i) == 15 &&
31                                   (a + e + i) == 15 &&
32                                   (g + e + c) == 15) {
33                                     printf("*****\n");
34                                     printf("%d %d %d\n", a, b, c);
35                                     printf("%d %d %d\n", d, e, f);
36                                     printf("%d %d %d\n", g, h, i);
37                                 }
38                           }
39                     }
40     return 0;
41 }
```

La salida de este programa generaría lo siguiente:

```
*****
2 7 6
9 5 1
4 3 8
*****
2 9 4
7 5 3
6 1 8
.....
.....
```

Para poner en práctica (*F*)

- ¿Qué cambios debería realizar en el programa anterior si en lugar de imprimir los cuadrados mágicos le solicitan que cuente los cuadrados mágicos de orden 3?
- Implemente un programa que permita imprimir y contar los cuadrados mágicos de orden 4.

Para pensar (*A*)

- En el programa anterior existe una estructura selectiva anidada (vea la línea 15 y la línea 25), ¿Es necesario que esta estructura se encuentre anidada? ¿Se podría juntar las dos condiciones en una sola expresión? ¿Es esto conveniente?
- Para la implementación de la generación de cuadrados mágicos de orden 4, conviene utilizar la instrucción `while` o la instrucción `for`.

1.7. Los números Armstrong

Un número Armstrong, también llamado número narcisista, es todo aquel número que es igual a la suma de cada uno de sus dígitos elevado al número total de dígitos.

A continuación siguen algunos ejemplos de números Armstrong.

- $371 = 3^3 + 7^3 + 1^3$. Total de dígitos 3.
- $8208 = 8^4 + 2^4 + 0^4 + 8^4$. Total de dígitos 4.
- $4210818 = 4^7 + 2^7 + 1^7 + 0^7 + 8^7 + 1^7 + 8^7$. Total de dígitos 7.

Se desea elaborar un programa en ANSI C que permita simular un juego con el usuario para adivinar números Armstrong. El programa solicitará un número entero al usuario. Si el número ingresado por el usuario es un número Armstrong, el programa termina, caso contrario seguirá solicitando un número al usuario.

Para resolver este problema se debe primero contar los dígitos de un número, luego sumar los dígitos del número elevando cada dígito al total de dígitos para finalmente hacer el control de flujo. Se introducirá además dos nuevas instrucciones que permiten interrumpir el control de flujo.

1.7.1. Contando los dígitos de un número

Lo primero que se debe hacer es realizar el conteo de dígitos del número ingresado por el usuario. En el programa 1.10 se solicita el ingreso de un número y se almacena en la variable *numero*. El control del flujo se realiza con la instrucción `while` usando la variable *encontrado* para el control. El ciclo iterativo se ejecutará mientras *encontrado* sea diferente de 0. Por este motivo la variable *encontrado* se inicializa con 0 para que la primera vez se ejecute. La idea es que cuando el usuario ingrese un número que sea Armstrong, la variable

encontrado se actualice con el valor de 1 y en ese momento la condición del `while` falla. Si *encontrado* tiene el valor de 1, !*encontrado* tendría el valor de 0 haciendo que la condición del `while` sea $falsa$.

Para encontrar la cantidad de dígitos de *numero* lo que se realiza son divisiones sucesivas entre 10 hasta que se llegue a 0. La cantidad de dígitos se almacena en la variable *total_digitos*. Se utiliza una copia de *numero* para hacer las divisiones sucesivas. En este programa dicha variable se denomina *numero_original*. Se usa esta variable pues como se realizarán divisiones sucesivas, el número original se perdería y no se podría hacer la comparación con la suma de dígitos. Recuerde que `numero_original /= 10` equivale a `numero_original = numero_original / 10`.

Programa 1.10: Contando los dígitos de un número

```
#include <stdio.h>

int main() {
    int encontrado = 0, numero;
    while (!encontrado){
        int numero_original, total_digitos;
        printf("Ingrese un numero Armstrong: ");
        scanf("%d", &numero);

        numero_original = numero;
        total_digitos = 0;
        while (numero_original != 0){
          ++total_digitos;
          numero_original /= 10;
        }
    }
    return 0;
}
```

Si el usuario ingresa los números 1, 12, 123, 1234 y 12345, la salida de este programa generaría lo siguiente:

```
Ingrese un numero Armstrong: 1
Total de digitos = 1
Ingrese un numero Armstrong: 12
Total de digitos = 2
Ingrese un numero Armstrong: 123
Total de digitos = 3
Ingrese un numero Armstrong: 1234
Total de digitos = 4
Ingrese un numero Armstrong: 12345
Total de digitos = 5
```

Para poner en práctica (G)

- Implemente el programa utilizando la intrucción `for` en lugar del `while`.
- Existe otra manera de obtener la cantidad de dígitos de un número usando el logaritmo en base 10 de un número. Recuerde que: $\log(10) = 1$, $\log(100) = 2$, $\log(1000) = 3$. La cantidad de cifras de un número cualquiera se obtiene sumando 1 a la parte entera de su logaritmo decimal. En C puede obtener el logaritmo en base 10 de un número usando la función `log10` cuyo prototipo se encuentra en el archivo de cabecera `math.h`. Cambie el programa para usar el logaritmo en el cálculo de los dígitos de un número.

1.7.2. Sumando los dígitos del número

Una vez hallada la cantidad de dígitos del número, se procede a calcular la sumatoria de los dígitos del número elevando cada uno de estos dígitos a la cantidad de digitos calculado anteriormente. Para calcular los dígitos

se utiliza la misma técnica de divisiones sucesivas entre 10. El dígito se obtiene en cada iteración usando el módulo del número dividido entre 10. Al final de la iteración se realiza la actualización de la variable de control de flujo, si la *suma* == *numero* entonces a la variable *encontrado* se le asigna el valor de 1. Haciendo que el ciclo finalice. Este tipo de control se conoce como ciclo iterativos controlados por centinela. Se utilizan cuando no se sabe cuántas veces se realizará el ciclo. En el programa 1.11 se puede apreciar el centinela *encontrado*.

Programa 1.11: Sumando los dígitos del número

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main() {
5      int encontrado = 0, numero;
6      while (!encontrado){
7          int numero_original, total_digitos, suma, digito, i;
8          printf("Ingrese un numero Armstrong: ");
9          scanf("%d", &numero);
10
11         numero_original = numero;
12         total_digitos = 0;
13         while (numero_original != 0){
14           ++total_digitos;
15           numero_original /= 10;
16         }
17
18         i=1;
19         suma = 0;
20         numero_original = numero;
21         while (numero_original != 0){
22             digito = numero_original % 10;
23             suma += (int)pow(digito, total_digitos);
24             numero_original /= 10;
25         }
26         if (suma==numero){
27             encontrado=1;
28             printf("El numero %d es Armstrong\n", numero);
29         }
30     }
31     return 0;
32 }
```

1.7.3. Filtrando los números negativos

El programa anterior no realiza ningún control en caso el número fuese negativo o 0. Si deseamos filtrar las situaciones en donde no se ingresa un número que cumple la condición deseada, se puede utilizar una instrucción selectiva. En el programa 1.12 se puede apreciar el programa con el filtro mencionado en la línea 11.

Programa 1.12: Filtrando los números negativos

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main() {
5      int encontrado = 0, numero;
6      while (!encontrado) {
7          int numero_original, total_digitos, suma, digito, i;
8          printf("Ingrese un numero Armstrong: ");
9          scanf("%d", &numero);
10
11         if (numero > 0) {
12             numero_original = numero;
13             total_digitos = 0;
14             while (numero_original != 0) {
15                 ++total_digitos;
16                 numero_original /= 10;
17             }
18
```

```
19              i = 1;
20              suma = 0;
21              numero_original = numero;
22              while (numero_original != 0) {
23                  digito = numero_original % 10;
24                  suma += (int) pow(digito, total_digitos);
25                  numero_original /= 10;
26              }
27              if (suma == numero) {
28                  encontrado = 1;
29                  printf("El numero %d es Armstrong\n", numero);
30              }
31          }
32      }
33      return 0;
34  }
```

Si el usuario ingresa los números −4, −6, −7, 0, 0 y 153, la salida de este programa generaría lo siguiente:

```
Ingrese un numero Armstrong: -4
Ingrese un numero Armstrong: -6
Ingrese un numero Armstrong: -7
Ingrese un numero Armstrong: 0
Ingrese un numero Armstrong: 0
Ingrese un numero Armstrong: 153
El numero 153 es Armstrong
```

1.7.4. Uso de la instrucción continue

El usar una instrucción selectiva efectivamente hace que se consiga el objetivo de filtrado de números pero incremente la complejidad del programa. Si deseamos que cuando el usuario ingreso un número que sea menor o igual a 0 no se haga nada y se vuelva a solicitar otro número se puede utilizar la instrucción `continue`.

La instrucción `continue` es una de las intrucciones de ruptura de flujo. Hace que el control de flujo se dirija hacia la siguiente iteración. En términos prácticos, si se ejecuta un `continue`, el control del programa se dirige hacia el inicio del ciclo iterativo. Lo que resta del *loop* no se ejecuta.

En el programa 1.13 puede verse una instrucción `continue` en la línea 12. Esto significa que cuando un número es menor o igual a 0 esta instrucción se ejecuta. Esto hace que las líneas 14 hasta la 32 no se ejecuten enviando el control hacia la línea 6 iniciando nuevamente una iteración.

Programa 1.13: Uso de la instrucción continue

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5       int encontrado = 0, numero;
6       while (!encontrado) {
7           int numero_original, total_digitos, suma, digito, i;
8           printf("Ingrese un numero Armstrong: ");
9           scanf("%d", &numero);
10
11          if (numero <= 0)
12              continue;
13
14          numero_original = numero;
15          total_digitos = 0;
16          while (numero_original != 0) {
17              ++total_digitos;
18              numero_original /= 10;
19          }
20
21          i = 1;
22          suma = 0;
```

```
23          numero_original = numero;
24          while (numero_original != 0) {
25              digito = numero_original % 10;
26              suma += (int) pow(digito, total_digitos);
27              numero_original /= 10;
28          }
29          if (suma == numero) {
30              encontrado = 1;
31              printf("El numero % d es Armstrong\n", numero);
32          }
33      }
34      return 0;
35  }
```

1.7.5. Uso de la instrucción break

La otra instrucción de ruptura de flujo es la intrucción `break`. La instrucción `break` hace que un ciclo iterativo termine su ejecución completa. Luego de un `break` no se ejecutará ningún *blucle* independientemente si la condición de control de flujo se cumple o no. En el programa 1.14 se puede apreciar el programa anterior modificado para que el control de salida de la iteración se realice mediante un `break`. Ya no se controla el ciclo por el centinela *encontrado*, en lugar de eso la condición del `while` es 1, eso significa un ciclo que nunca termina. Pero en la línea 31 se puede observar un uso del `break` que hará que ciclo termine cuando se encuentre un número Armstrong.

Programa 1.14: Uso de la instrucción break

```
1   #include <stdio.h>
2   #include <math.h>
3
4   int main() {
5       int numero;
6       while (1) {
7           int numero_original, total_digitos, suma, digito, i;
8           printf("Ingrese un numero Armstrong: ");
9           scanf(" %d", &numero);
10
11          if (numero <= 0)
12              continue;
13
14          numero_original = numero;
15          total_digitos = 0;
16          while (numero_original != 0) {
17              ++total_digitos;
18              numero_original /= 10;
19          }
20
21          i = 1;
22          suma = 0;
23          numero_original = numero;
24          while (numero_original != 0) {
25              digito = numero_original % 10;
26              suma += (int) pow(digito, total_digitos);
27              numero_original /= 10;
28          }
29          if (suma == numero) {
30              printf("El numero %d es Armstrong\n", numero);
31              break;
32          }
33      }
34      return 0;
35  }
```

1.8. Ejercicios propuestos

1.8.1. SEND+MORE=MONEY

El problema $SEND + MORE = MONEY$ consisten en encontrar valores distintos para los dígitos D, E, M, N, O, R, S, Y de forma tal que que S y M sean diferentes de 0 y se satisfaga la ecuación $SEND + MORE = MONEY$. Elabore un programa en ANSI C que permite encontrar los valores para los dígitos del problema $SEND + MORE = MONEY$.

Sugerencia

- Utilice la técnica de la fuerza bruta para generar todas las posibles combinaciones para las letras D, E, M, N, O, R, S, Y.
- Forme el número $SEND$, el número $MORE$ y el número $MONEY$. Si D, E, M, N, O, R, S, Y son variables que contienen dígitos, el número $SEND$ se puede obtener de la siguiente manera: $SEND = S * 1000 + E * 100 + N * 10 + D$.
- Utilice una estructura selectiva para verificar si es que se satisface la ecuación.

1.8.2. Cambio de base

Elabore un programa en ANSI C que permita realizar el cambio de base de una lista de números ingresados por el usuario. La base destino será ingresada al inicio del programa y deberá ser un número en el rango [2..9]. El programa terminará cuando se ingrese el número 0 o un número negativo. Se asumirá que los números ingresados se encuentran en base 10.

Ejemplo de ejecución del programa:

```
Ingrese base: 3
Ingrese numero: 15
El numero 15 en base 3 es = 120

Ingrese numero: 26
El numero 27 en base 3 es = 222

Ingrese numero: 0
```

Sugerencia

- Para realizar el cambio de base, divida el número por la base de forma sucesiva hasta llegar a 0. En el resto de la división encontrará el dígito. Con en digito encontrado puede ir armando el número en la otra base.
- Utilice un ciclo iterativo controlado por un centinela. El centinela podría ser una variable como *encontrado* que cuando se ingresa un número 0 se le asigna el valor de 1. Esta variable la podría inicializar en 0.

1.8.3. El algoritmo de Euclides

El algoritmo de Euclides es un método que permite calcular el máximo común divisor de dos números. Dados dos enteros a y b cuyo máximo común divisor se desea hallar, y asumiendo que $a > 0$, $b > 0$ y $a \geq b$, se realiza lo siguiente:

- Se calcula r como el resto de la división entre a y b.

- Si r es igual a 0 el algoritmo termina y se dice que b es el máximo común divisor de a y b.
- Caso contrario máximo común divisor de a y b será igual al máximo común divisor de b y r, por lo que se repite el procedimiento nuevamente ahora con b y r.

Se pide que elabore un programa en ANSI C que permita leer una lista de n números todos ellos mayores que 0 e imprima el máximo común divisor. El programa terminará cuando se ingrese el número 0 o un número negativo.

Ejemplo de ejecución del programa:

```
Ingrese numero: 224
Ingrese numero: 693
Ingrese numero: 504
Ingrese numero: 0

El maximo comun divisor es: 7
```

Sugerencia

- Para hallar el máximo común dividor de varios números puede aplicar el algoritmo de Euclides con los dos primeros números. Luego use este número (máximo común divisor) para calcular el máximo común divisor con el tercer número, el resultado lo puede usar para calcular el máximo común divisor con el cuarto número y así sucesivamente.
- Tenga presente que para que el algoritmo funcion a debe ser mayor o igual que b. Los números ingresados por el usuario no necesariamente siguen un orden establecido por lo que será necesario calcular el mayor de 2 números.
- Al igual que en el caso anterior, utilice un ciclo iterativo controlado por un centinela. El centinela podría ser una variable como *encontrado* que cuando se ingresa un número 0 se le asigna el valor de 1. Esta variable la podría inicializar en 0.

1.8.4. Suma de fracciones

Se pide que elabore un programa en ANSI C que permita leer una lista de fracciones e imprima la suma de fracciones. El programa terminará cuando se ingrese la fracción que tenga como numerador 0 o un número negativo.

Ejemplo de ejecución del programa:

```
Ingrese fraccion (numerador denominador): 1 2
Ingrese fraccion (numerador denominador): 3 4
Ingrese fraccion (numerador denominador): 1 5
Ingrese fraccion (numerador denominador): 0 0

La suma es (numerador denominador): 29 20
```

Sugerencia

- Para hallar la suma de fracciones se sugiere que sume las dos primeras fracciones, el resultado de esto lo sume con la tercera fracción, el resultado de esta con la cuarta fracción y así sucesivamente.
- Recuerde que para sumar dos fracciones $\frac{a}{b} + \frac{b}{c} = \frac{(m.c.m.(b,c)/b) \times a + (m.c.m.(b,c)/c) \times b}{m.c.m.(b,c)}$
- Recuerde que $m.c.m.(a,b) = \frac{a \times b}{m.c.d.(a,b)}$.
- Al igual que en el caso anterior, utilice un ciclo iterativo controlado por un centinela. El centinela podría ser una variable como *encontrado* que cuando se ingresa un número 0 se le asigna el valor de 1. Esta variable la podría inicializar en 0.

1.8.5. Números primos gemelos

Dos números p y q son números primos gemelos si es que tanto p como q son números primos y además la diferencia entre ambos es exactamente 2. De esta manera los siguientes pares de números son números gemelos $(3,5)$, $(11,13)$, $(17,19)$, $(29,31)$, $(41,43)$.

Se pide que elabore un programa en ANSI C que imprima los n primeros pares de números gemelos.

Sugerencia

- Se sugiere que genere los números primos controlando la iteración por un centinela. Guarde el último número primo generado para que pueda ser comparado en la siguiente iteración.

1.8.6. Números perfectos

Cuando la suma de los divisores de un número es igual al propio número, se dice que es un número perfecto. De esta forma por ejemplo el número 28 es perfecto dado que $28 = 1 + 2 + 4 + 7 + 14$.

Se pide que elabore un programa que solicite al usuario un número e imprima si este número es un número perfecto. El programa terminará cuando se ingrese el número 0 o un número negativo.

1.8.7. El último teorema de Fermat

El último teorema de Fermat, conjeturado por Pierre de Fermat en 1637, pero no demostrado, establece que: si n es un número entero mayor que 2, entonces no existen números enteros no nulos x, y y z, tales que se cumpla la igualdad: $x^n + y^n = z^n$.

Se pide que elabore un programa en ANSI C que verifique si el último teorema de Fermat se cumple para el conjunto de valores x, y y z tales que $x < 100$, $y < 100$, $z < 100$, $n \in [3..5]$