1. Introducción

Estas notas son una Introducción a la Resolución Sistemática de Problemas mediante Algoritmos. El objetivo de las notas es presentar una metodología que permita obtener soluciones algorítmicas correctas de un problema dado.

La Programación de computadoras comenzó como un arte, y aún hoy en día mucha gente aprende a programar sólo mirando a otros proceder (ej. Un profesor, un amigo, etc.), y mediante el hábito, sin conocimiento de los principios que hay detrás de esta actividad. En los últimos años, sin embargo, la investigación en el área ha arrojado teoría suficiente para que estemos en capacidad de comenzar a enseñar estos principios en los cursos básicos de enseñanza de la programación.

La metodología que proponemos toca la raíz de los problemas actuales en programación y provee principios básicos para superarlos. Uno de estos problemas es que los programadores han tenido poco conocimiento de lo que significa que un programa sea correcto y de cómo probar que un programa es correcto. Cuando decimos “probar” no queremos decir necesariamente que tengamos que usar un formalismo o las matemáticas, lo que queremos expresar es “encontrar un argumento que convenza al lector de la veracidad de algo”. Desafortunadamente, los programadores no son muy adeptos a esto, basta observar la cantidad de tiempo que se gasta en “debugging” (proceso de depuración de programas para eliminar entre otros errores, los errores de lógica), y esto ¿por qué? Porque el método de programación normalmente usado ha sido “desarrollo por casos de prueba”, es decir, el programa es desarrollado sobre la base de algunos ejemplos de lo que el programa debe hacer. A medida que se encuentran más casos de prueba, estos son ejecutados y el programa es modificado para que tome en cuenta los nuevos casos de prueba. El proceso continúa con modificaciones del programa a cada paso, hasta que se piensa que ya han sido considerados suficientes casos de prueba.

Gran parte del problema se ha debido a que no hemos poseído herramientas adecuadas. El razonamiento sobre cómo programar se ha basado en cómo los programas son ejecutados, y los argumentos sobre la corrección (diremos de ahora en adelante “correctitud”, pues el término corrección se entiende como si quisiéramos corregir un programa y lo que queremos es probar que un programa es correcto) de un programa se han basado en sólo determinar casos de prueba que son probados corriendo el programa o simulados a mano. La intuición y el sentido común han sido simplemente insuficientes. Por otro lado, no siempre ha estado claro lo que significa que un programa sea “correcto”, en parte porque la especificación de programas ha sido también un tema muy impreciso.

El principio básico de estas notas será el desarrollo simultáneo de un programa con la demostración de la correctitud. Es muy difícil probar la correctitud de un programa ya hecho, por lo que es más conveniente usar las ideas de prueba de correctitud a medida que se desarrolla el programa. Por otra parte, si sabemos que un programa es correcto, no tendrá sentido hacer muchas pruebas al programa con casos de prueba, sólo se harán las necesarias para corroborar que no cometimos un error, como humanos que somos, por ejemplo, en la transcripción del programa; además , así también reduciremos la cantidad de tiempo

dedicada a la tediosa tarea del "debugging". Para ello trataremos el tema de especificación de programas (o problemas) y el concepto de prueba formal de programas a partir del significado formal (la semántica) de los constructores básicos (es decir, las instrucciones del lenguaje de programación) que utilizaremos para hacer los programas.

Conscientes de que estas notas van dirigidas a un primer curso sobre enseñanza de la programación, el enfoque que llevaremos a cabo trata de balancear el uso de la intuición y el sentido común, con técnicas más formales de desarrollo de programas.

1.1. Problemas, algoritmos y programas

Un *problema* es una situación donde se desea algo sin poder ver inmediatamente la serie de *acciones* a efectuar para obtener ese algo.

Resolver un problema consiste primero que nada en comprender de qué trata y *especificarlo* lo más formalmente posible con el propósito de eliminar la *ambigüedad*, la *inconsistencia* y la *incompletitud*, con miras a obtener un enunciado claro, preciso, conciso y que capture todos los requerimientos. Normalmente la especificación en lenguaje natural lleva consigo problemas de inconsistencia, ambigüedad e incompletitud; es por esto que nos valemos de lenguajes más formales como las matemáticas (lógica, teoría de conjuntos, álgebra, etc.) para evitar estos problemas.

Ejemplo de ambigüedad: la frase "Mira al hombre en el patio con un telescopio" se presta a ambigüedad. ¿Utilizamos un telescopio para mirar al hombre en el patio o el hombre que miramos tiene un telescopio?

Ejemplo de inconsistencia: Por inconsistencia entendemos que en el enunciado podamos incurrir en contradicciones. Ejemplo sobre el procesador de palabras:

- Todas las líneas de texto tienen la misma longitud, indicada por el usuario.
- Un cambio de línea debe ocurrir solo después de una palabra, a menos que el usuario pida explícitamente la división por sílabas.

La inconsistencia resulta al preguntarse ¿Si la palabra es más larga que la línea? Entonces si el usuario no pide explícitamente la división por sílabas, esta línea tendrá mayor longitud.

Ejemplo de incompletitud: Por incompletitud entendemos que no todos los términos estén bien definidos o que no estén capturados todos los requerimientos del problema. En un manual de un procesador de palabras encontramos la frase "Seleccionar es el proceso de designar las áreas de su documento sobre las cuales desea trabajar" ¿qué significa designar, colocar el "ratón" dónde? ¿qué significa área? ¿cómo deben ser las áreas? ¿es una sola área?

En otras palabras, se trata de encontrar una representación del problema donde todo esté dicho, sea mediante gráficos, o utilizando otro lenguaje distinto al natural (ej. las matemáticas). En esta etapa interviene el proceso de abstracción, que permite simplificar el problema, buscando modelos abstractos equivalentes y eliminando informaciones superfluas.

Un problema no estará del todo bien comprendido si no hemos encontrado una representación en la cual todos los elementos que intervienen sean representados sin redundancia, sin ambigüedad y sin inconsistencias. El universo de búsqueda de la solución estará en ese momento bien delimitado y con frecuencia surgirá en forma más clara la dificultad principal del problema. El problema se convierte en más abstracto y más puro. Hablamos entonces de un *enunciado cerrado.*

Por ejemplo, si queremos determinar la ruta más corta para ir de Caracas a Barquisimeto en automóvil, podemos tomar el mapa vial de Venezuela y modelarlo a través de un grafo como en la figura 1, donde los nodos o vértices serían las ciudades y los arcos las vías de comunicación existentes entre las ciudades. A cada arco asociamos el número de kilómetros de la vía que representa y luego intentamos buscar la solución en el grafo.

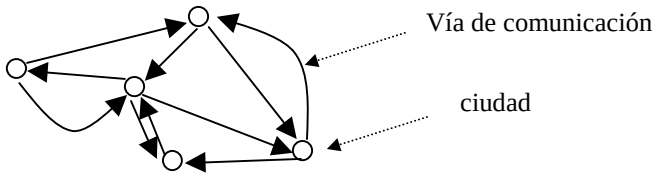

Figura 1

En matemáticas la forma más general de enunciado de un problema se puede escribir como: "Encontrar en un conjunto X dado, los elementos x que satisfacen un conjunto dado de restricciones K(x)". Por ejemplo: encontrar en el conjunto de los números naturales los números x tales que $x^2 + 83 = 37x$.

Observación:

- Cuando damos el espacio X, generalmente damos implícitamente la estructura de X y las operaciones lícitas sobre X. El conocimiento de X es un condensado crucial de información.

Una vez que obtenemos una especificación del problema que corresponda a un enunciado cerrado, en la práctica lo que hacemos es precisar aún más la especificación mediante la descomposición del problema en subproblemas más concretos (hacemos un *diseño descendente* del problema), hasta llegar a un punto del refinamiento del enunciado en donde sabemos llevar a cabo las *acciones* en él descritas. En ese momento tenemos la solución del problema, ya sea porque encontramos los resultados buscados o porque obtenemos *un algoritmo* que resuelve el problema, es decir, una descripción de cómo llevar a cabo un conjunto de acciones, que sabemos realizar a priori, sobre los datos del problema, para obtener los resultados buscados; en este último caso decimos que tenemos *una solución algorítmica del problema*.

Por ejemplo: "Determinar x en los números enteros tal que $x + 3 = 2$"

Este enunciado es equivalente a “Determinar x en los números enteros tal que x + 3 – 3 = 2 – 3” porque conocemos las propiedades del conjunto de los números enteros. Pero este enunciado nos lleva, por aplicación de las propiedades de números enteros (son estas propiedades las que consideramos cruciales conocer para resolver el problema), al siguiente enunciado que es la solución del problema: “Determinar x en los números enteros tal que x = -1 ”. En este caso obtenemos directamente el número buscado.

Nuestro interés será buscar la solución de un problema como una descripción de una *acción* que manipula los datos hasta llegar al resultado, a esta descripción la llamamos *algoritmo*. *Una acción* es un evento producido por un actor, que llamamos el ejecutante; por ejemplo, una calculadora puede llevar a cabo la acción de sumar dos números, la calculadora sería el ejecutante de esa acción.

Una acción toma lugar durante un período de tiempo finito y produce un resultado *bien definido y previsto*. La acción requiere de la existencia de datos sobre los cuales se ejecuta para producir nuevos datos que reflejan el efecto esperado. Los datos que nos da el enunciado del problema poseen normalmente una estructura que los caracteriza (ej., son números, conjuntos, secuencias, etc.) y que llamaremos *el tipo del dato o la clase del dato*.

Una clase o tipo de dato viene dado por un conjunto de valores y su comportamiento, es decir, un conjunto de operaciones que podemos realizar con estos valores (por ejemplo, sobre los números naturales podemos sumar dos números, restar, multiplicar, etc.).

Un objeto lo podemos caracterizar por su *identificación*, su *clase ó tipo* y su *valor*. Decimos que un objeto es un *ejemplar (o instancia) de su clase*. Podemos decir que un algoritmo, en vez de manipular datos, manipulará objetos. Por ejemplo: en el problema de la búsqueda del camino más corto entre dos ciudades de Venezuela dado antes, el grafo es la clase de datos involucrada en la especificación final. Un grafo tiene operaciones bien definidas que podemos realizar sobre él, como por ejemplo: agregar un nodo, eliminar un arco, etc. El mapa de Venezuela y las distancias entre ciudades son los datos del problema y si vemos el mapa como un grafo, este mapa particular será un *objeto de la clase grafo*.

El conjunto de valores de los objetos manipulados por una acción, observados a un instante de tiempo dado t del desarrollo de ésta, es llamado el *estado* de los objetos manipulados por la acción en el instante t. Los valores de los objetos al comienzo de la acción los llamamos *los datos de entrada de la acción* y definen el estado inicial. Los valores de los objetos al concluir la acción los llamamos *datos de salida o resultados* y definen el estado final.

Si una acción puede ser descompuesta en un conjunto de acciones a realizar de manera secuencial, entonces diremos que la acción es un *proceso secuencial*. Un proceso secuencial no es más que una secuencia de acciones (o eventos). Retomando la definición del término algoritmo, podemos decir que *un algoritmo* es una descripción de un esquema o patrón de realización de un proceso o conjunto de procesos similares mediante un repertorio finito y no ambiguo de acciones elementales que se supone realizables a priori. Extenderemos el término acción a la descripción misma del evento y diremos que el algoritmo es la acción que hay que llevar a cabo para resolver el problema. Cuando un

algoritmo está expresado en términos de acciones de un lenguaje de programación, diremos que es un *programa*.

Por ejemplo: hacer una torta es una acción, esa acción se descompone en varias acciones más simples (agregar los ingredientes, mezclarlos, etc.) que serían el proceso que se lleva a cabo. La receta de la torta es el algoritmo o la descripción del proceso a seguir para hacer la torta.

Cuando decimos "conjunto de procesos similares" queremos decir que un algoritmo normalmente describe un proceso no sólo para un conjunto de datos de entrada específicos, sino para todos los posibles valores que puedan tener los objetos involucrados. Por ejemplo, cuando describimos el algoritmo de la multiplicación de dos números naturales no lo hacemos para dos números específicos (por ejemplo, 4 y 29) sino de una manera más general, de forma que describa la multiplicación de cualesquiera dos números naturales.

Ejemplo:

Enunciado del problema: Un vehículo que se desplaza a una velocidad constante v, consume l litros de gasolina cuando recorre k kilómetros. Determinar la cantidad r de gasolina que consumiría el vehículo a la velocidad v si recorre x kilómetros.

Note que el problema tiene un enunciado más general que el problema que consiste en buscar la ruta mínima entre dos ciudades específicas de Venezuela, en el sentido que independientemente de los valores de v, k, l y x (implícitamente cada una de las cantidades deben ser no negativas), queremos determinar un algoritmo que resuelva el problema (que calcule la cantidad r). Por lo tanto un mismo algoritmo deberá resolver el problema para valores particulares de v, k, l y x. Podemos ver a v, k, l, x, r como variables (al igual que en matemáticas) que pueden contener valores de un determinado tipo o clase de datos. En este caso el tipo de dato de todas estas variables es "número real".

De igual forma, podemos formular un problema más general que el dado sobre la ruta mínima entre dos ciudades de Venezuela, de la siguiente forma: Queremos buscar la ruta mínima para ir de una ciudad X a una ciudad Y en un país Z. En este caso tendremos tres variables que representan los objetos involucrados en el problema, a saber, dos ciudades y un país. Normalmente los problemas se presentan de esta forma. Es más útil encontrar un algoritmo para multiplicar dos números cualesquiera, que un algoritmo que sirva para multiplicar 2 por 3 solamente!

Ejemplos sobre correctitud versus análisis de casos de prueba

Primer ejemplo:

Ejemplo que muestra que la intuición no basta para desarrollar un algoritmo que resuelve un problema:

Se quiere pelar un número “suficiente” de papas que están en un cesto. El cesto de papas puede vaciarse en cualquier momento y podemos reponer el cesto con papas una vez éste se vacíe.

Solución:

Versión 1:

Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
    Si el cesto no está vacío entonces pelar una papa

Este algoritmo funciona cuando el número de papas a pelar es menor o igual al número de papas en el cesto. Si el cesto posee menos papas que las requeridas entonces caemos en un ciclo infinito (proceso sin fin) una vez que se vacíe el cesto

Versión 2:

Mientras el cesto no esté vació haga lo siguiente:
    Pelar una papa
    Pelar una papa

Se pelan tantas papas como hay en el cesto si el número de papas originalmente es par. Si el número inicial de papas es impar entonces hay una acción en el proceso que no puede ser ejecutada. Sólo cuando el cesto tiene el número de papas requerido el algoritmo resuelve el problema.

Versión 3:

Mientras el cesto no esté vacío y el número de papas peladas sea insuficiente haga lo siguiente: Pelar una papa

Este algoritmo no cae en ciclo infinito ni tratará de ejecutar una acción que es imposible de realizar. Sin embargo, solo cuando el cesto contiene el número de papas requerido el algoritmo resuelve el problema.

Versión 4:

Mientras el cesto no esté vacío hacer lo siguiente:
    Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
        Pelar una papa

Hagamos un diagrama de estados a medida que se ejecuta el proceso partiendo de todos los posibles estados iniciales. Este diagrama también se conoce como “corrida en frío” o “simulación del algoritmo”.

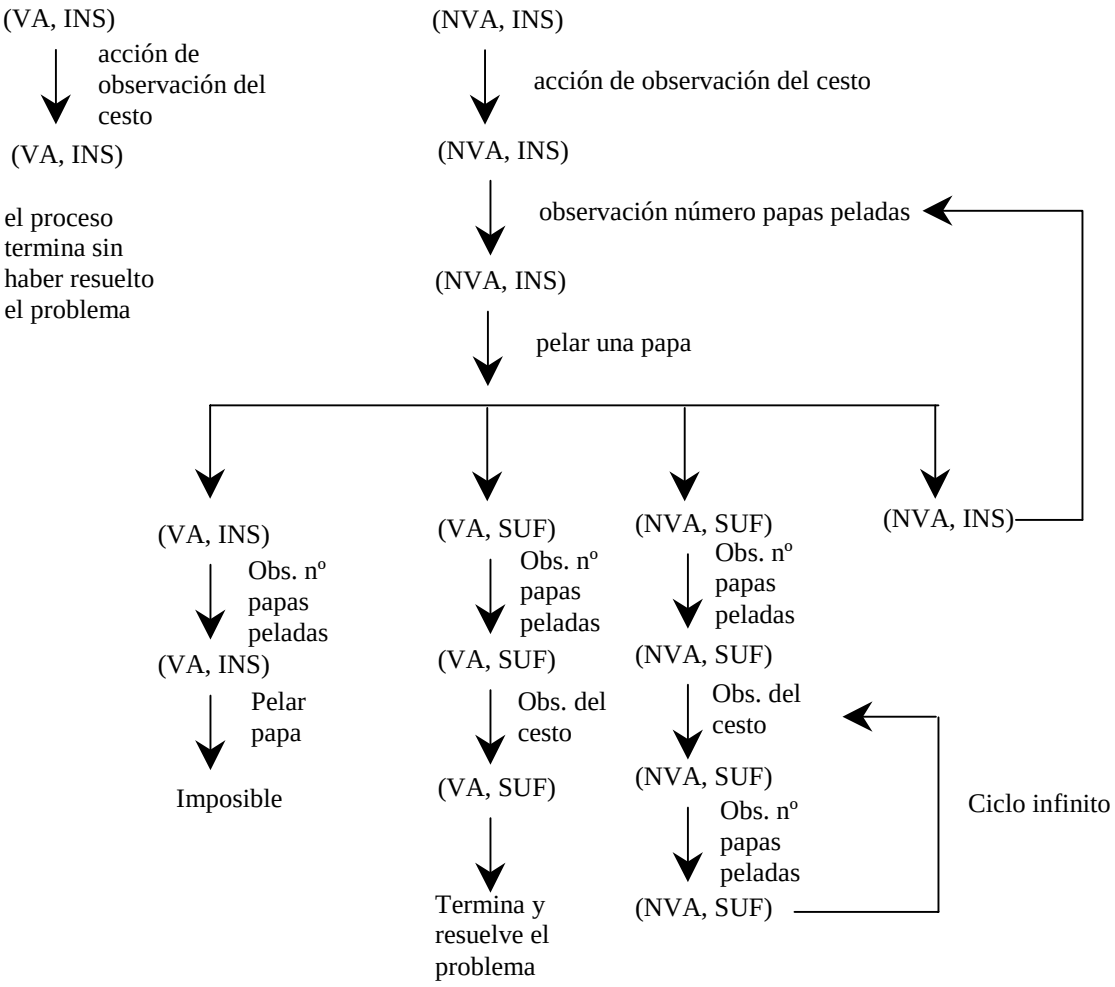

Figura 2

Descripción de los estados:

VA = cesto vacío,
INS = Número insuficiente de papas peladas

NVA = cesto no vacío
SUF = número suficiente de papas peladas

Estados iniciales posibles: (VA, INS) y (NVA, INS)

En las figura 2 vemos el proceso que genera el algoritmo partiendo respectivamente de los estados iniciales (VA, INS) y (NVA, INS)

Versión 5:

Mientras el número de papas no sea suficiente hacer lo siguiente:
Si el cesto no está vacío entonces pelar una papa
en caso contrario llenar el cesto con papas

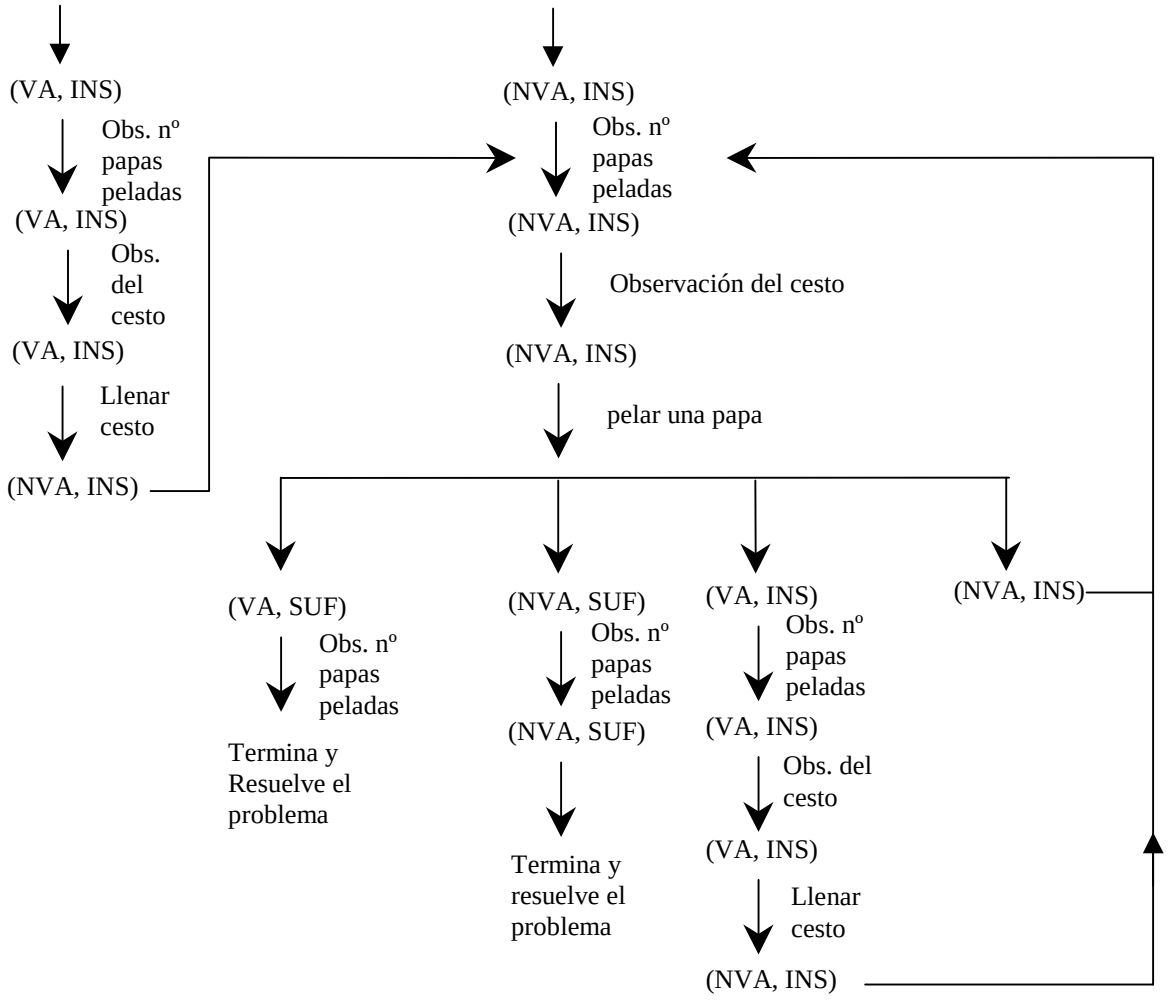

Figura 3

Esta versión es correcta, es decir, es un algoritmo que resuelve el problema, no existen acciones imposibles de realizar en la ejecución, ni cae en ciclo infinito y cuando termina se habrán pelado el número de papas requerido. El diagrama de estados de la figura 3 confirma este hecho.

Demostrar la correctitud del algoritmo anterior significa demostrar que partiendo de cualquier estado, al ejecutar el algoritmo, éste termina en un tiempo finito y cuando termina, este resuelve el problema (el estado final será SUF para el número de papas peladas).

Para este problema en particular, el diagrama de estados nos permite verificar la correctitud de nuestro problema y esto se debe a que el número posible de estados es muy reducido y podemos hacer una corrida completa para todos los estados iniciales posibles. Este no será el caso general, y no podremos utilizar el diagrama de estados (o corrida en frío) para verificar la correctitud del algoritmo pues el espacio de estados será, en general, infinito.

Parte de la verificación de la correctitud debe ser “el algoritmo termina”, o sea, que el algoritmo no debe caer en ciclos infinitos. Vemos que el diagrama de estados, en este caso, nos permite comprobar esto, ya que todo ciclo pasa por la acción “pelar papa” y como el número de papas que se quiere pelar es finito, estos ciclos serán finitos. Y cuando el algoritmo termina, se vé que termina en un estado que resuelve el problema.

Versión 6:

En la vesión anterior hay un problema de *estilo*: la acción que se repite no siempre es “pelar papa”, pero lo lógico es pelar una papa cada vez que se observa que hay un número insuficiente de papas.

Una versión “mejor”:

Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
    Si el cesto está vacío entonces llenar el cesto con papas
    Pelar una papa

Las acciones que se repiten (el cuerpo del mientras) tienen un resultado previsible cada vez que se ejecutan: independientemente del estado inicial de la acción, al concluir ésta, se ha pelado una papa. En la versión (5) al concluir la acción que se repite no sabremos si se ha pelado una papa o no.

Ejercicio: Dibuje el diagrama de estados de la versión (6) y verifique que el algoritmo es correcto.

Estudie de la misma manera las siguientes versiones:

a) Mientras el número de papas peladas sea insuficiente hacer lo siguiente:
    Mientras el cesto no esté vacío hacer:
        Pelar una papa

b) Mientras el número de papas peladas sea insuficiente o el cesto no esté vacío hacer:
    Pelar una papa

c) Si el cesto está vacío entonces llenarlo
   Pelar una papa
   Mientras el número de papas peladas sea insuficiente hacer:
    Si el cesto no está vacío entonces pelar una papa
    Si el cesto está vacío entonces llenar el cesto y pelar una papa

   ¿En esta versión las acciones que se repiten tienen siempre como estado final haber pelado exactamente una papa?. ¿Cree usted que esta versión es poco elegante? ¿Por qué?.

Segundo ejemplo (importancia de las especificaciones y del razonamiento sobre programas):

Supongamos que hemos escrito un programa muy largo que, entre otras cosas, calcula, como resultados intermedios, el cociente **q** y el resto **r** de dividir un entero no negativo **x** entre un entero positivo **y**. Por ejemplo, para **x**=7 e **y**=2, el programa calcula **q**=3 y **r**=1.

El trozo de programa que calcula el cociente y el resto es muy simple, pues consiste en el proceso de sustraer tantas veces **y** a una copia de **x,** guardando el número de sustracciones que se hayan efectuado, hasta que una sustracción más resulte en un número negativo. El trozo de programa, tal y como aparece en el programa completo es el siguiente (los puntos "..." son para indicar que el trozo de programa está inserto en otro programa más grande):

```
...
r := x;  q:=0;
mientras r>y hacer
        comienzo  r:=r-y;  q:= q+1  fin;
...
```

Comencemos a depurar el programa.

Respecto al cálculo del cociente y el resto, sabemos claramente que inicialmente el divisor no debe ser negativo y que después de concluir la ejecución del trozo de programa, las variables deberían satisfacer:

$$x = y*q + r$$

De acuerdo a lo anterior, agregamos algunas instrucciones para imprimir algunas variables con la finalidad de revisar los cálculos:

```
 ...
escribir ('dividendo x =', x, 'divisor y =', y);
r := x;  q:=0;
mientras r>y hacer
        comienzo  r:=r-y;  q:= q+1  fin;
escribir ( ' y*q + r = ', y*q + r);
...
```

Desafortunadamente, como este trozo de programa se ejecuta muchas veces en una corrida del programa, la salida es muy voluminosa. En realidad queremos conocer los valores de las variables sólo si ocurre un error. Supongamos que nuestro lenguaje de programación permite evaluar expresiones lógicas cuando estas aparecen entre llaves, y si la evaluación de la expresión lógica resulta falsa entonces el programa se detiene e imprime el estado de las variables en ese momento; si la evaluación de la expresión es verdadera, entonces el programa continúa su ejecución normalmente. Estas expresiones lógicas son llamadas *aserciones,* debido a que estamos afirmando que ellas serán verdaderas en el instante justo después de la ejecución de la instrucción anterior a ella.

A pesar de la ineficiencia que significa insertar aserciones, pues aumenta el número de cálculos que debe hacer el programa, decidimos incorporarlas pues es preferible que el programa detecte un error cuando sea utilizado por otros a que arroje resultados erróneos.

Por lo tanto incorporamos las aserciones al programa, y eliminamos las instrucciones de escritura:

```
  ...
{ y > 0 }
r := x;  q:=0;
mientras r > y hacer
        comienzo  r := r-y;  q := q+1  fin;
{  x = y*q + r }
...
```

De esta forma, nos ahorramos largos listados (papel impreso) en el proceso de depuración.

En una corrida de prueba, el chequeo de aserciones detecta un error debido a que **y** es 0 justo antes del cálculo del cociente y el resto. Nos lleva 4 horas encontrar y corregir el error en el cálculo de **y**. Luego, más adelante, tardamos un día en seguirle la pista a un error que no fue detectado en las aserciones, y determinamos que el cálculo del cociente y el resto fue:

$x = 6, y = 3, q = 1, r = 3$

Como vemos, las aserciones son verdaderas para estos valores. El problema se debe a que el resto ha debido ser menor estricto que el divisor y no lo es. Detectamos que la condición de continuación del proceso iterativo (el **mientras**) ha debido ser ($r \geq y$) en lugar de ($r > y$). Si hubiésemos sido lo suficientemente cuidadosos en colocar en la aserción del resultado otras propiedades que debe cumplir el cociente y el resto, es decir, si hubiésemos utilizado la aserción más *fuerte* $(x = y*q + r) \wedge (r < y)$, nos habríamos ahorrado un día de trabajo. Corregimos el error insertando la aserción más fuerte:

```
  ...
{ y > 0 }
r := x;  q:=0;
mientras r  ≥ y hacer
        comienzo  r :=  r - y;  q := q+1  fin;
{  (x = y*q + r)∧(r < y) }
...
```

Las cosas marchan bien por un tiempo, pero un día obtenemos una salida incomprensible. Resulta que el algoritmo del cociente y resto produjo un resto negativo r = -2. Y nos damos cuenta que r es negativo debido a que inicialmente x era –2. Hubo otro error en el cálculo de los datos de entrada del algoritmo de cociente y resto, pues se supone que x no puede ser

negativo. Hemos podido ahorrar tiempo en detectar este error si nuevamente hubiésemos sido cuidadosos en fortalecer suficientemente las aserciones. Una vez más , arreglamos el error y reforzamos las aserciones:

```
  ...
{ ( x ≥ 0) ∧ (y > 0) }
r := x;  q:=0;
mientras r  ≥ y hacer
        comienzo  r:=r-y;  q:= q+1  fin;
{  (x = y*q + r)∧( 0 ≤ r < y) }
...
```

Parte del problema que hemos confrontado se debe a que no hemos sido lo suficientemente cuidadosos en especificar lo que el trozo de programa debe hacer. Hemos debido comenzar por escribir la aserción inicial (x ≥ 0) ∧ (y > 0) y la aserción final (x = y*q + r)∧(0 ≤ r < y) antes de escribir el trozo de programa, ya que son estas aserciones las que conforman la definición de cociente y de resto.

Pero, ¿qué podemos decir del error que cometimos en la condición del proceso iterativo?. ¿Lo hemos podido prevenir desde un comienzo?. ¿Existe alguna manera de probar, a partir del programa y las aserciones, que las aserciones son verdaderas en cada punto de la ejecución del programa donde ellas aparecen?. Veamos lo que podemos hacer.

Como antes del mientras se cumple x = r y q = 0, vemos que también se cumple parte del resultado antes del mientras:

(1) x = y*q + r

Además , por las asignaciones que se hacen en el cuerpo del mientras, podemos concluir que si (1) es verdad antes de la ejecución del cuerpo del mientras entonces es verdad después de su ejecución. Así (1) será verdad antes y después de la ejecución de *cada* iteración del mientras. Insertemos (1) en los lugares que corresponde y hagamos las aserciones tan fuertes como sea posible:

```
  ...
{ ( x ≥ 0) ∧ (y > 0) }
r := x;  q:=0;
{ (0 ≤ r) ∧ (0 < y) ∧ (x = y*q + r) }
mientras r ≥ y hacer
        comienzo
        { (0 ≤ r) ∧ (0 < y ≤ r) ∧ (x = y*q + r) }
        r := r-y;  q := q+1
        { (0 ≤ r) ∧ (0 < y) ∧ (x = y*q + r) }
        fin;
{  (x = y*q + r) ∧ (0 ≤ r < y) }
...
```

¿Cómo podemos probar que la condición es correcta?. Cuando el mientras termina la condición debe ser falsa y queremos que $r < y$, por lo que el complemento, $r \geq y$ debe ser la condición correcta del mientras.

Del ejemplo anterior se desprende que si supiéramos encontrar las aserciones más fuertes posibles y aprendiéramos a razonar cuidadosamente sobre aserciones y programas, no cometeríamos tantos errores, sabríamos que nuestro programa es correcto y no necesitaríamos depurar programas (sólo correríamos casos de prueba para aumentar la confianza en un programa que sabemos está correcto; encontrar un error sería la excepción y no la regla). Por lo que los días gastados en correr casos de prueba, examinar listados y buscar errores, podrían ser invertidos en otras actividades.

Tercer ejemplo (razonar en términos de propiedades del problema y no por casos de prueba):

Para mostrar lo efectivo que puede ser adoptar una metodología como la que proponemos, ilustremos el siguiente ejemplo:

Supongamos que tenemos un recipiente de café que contiene algunos granos negros y otros blancos y que el siguiente proceso se repite tantas veces como sea posible:

Se toma al azar dos granos del recipiente. Si tienen el mismo color se botan fuera y colocamos otro grano negro en el recipiente (hay suficientes granos negros disponibles para hacer esto). Si tienen color diferente, se coloca el grano blanco de vuelta en el recipiente y se bota el grano negro.

La ejecución de este proceso reduce en uno el número de granos en el recipiente. La repetición de este proceso terminará con un grano en el recipiente ya que no podremos tomar dos granos para continuar el proceso. La pregunta es: ¿qué puede decirse del color del grano final en función del número de granos blancos y negros inicialmente en el recipiente? Piense un poco....

Como vemos, no ayuda mucho intentar con casos de prueba. No ayuda mucho ver qué pasa cuando hay dos granos de color distinto, o dos blancos y uno negro, etc.; aunque examinar casos pequeños permite obtener una mejor comprensión del problema.

En lugar de esto, proceda como sigue: quizás exista una *propiedad sencilla de los granos en el recipiente que permanece cierta una vez que tomamos dos granos y colocamos el grano que corresponde en el recipiente,* y que junto con el hecho de que sólo quedará un grano, pueda dar la respuesta. Como la propiedad siempre permanecerá cierta, nosotros la llamamos *un invariante*.

Tratemos de buscar tal propiedad. Suponga que cuando terminamos tenemos un grano negro. ¿Qué propiedad es verdad cuando terminamos que podríamos generalizar, quizás, para que sea nuestro invariante? Una sería “existe un número impar de granos negros en el recipiente”, así, quizás la propiedad de permanecer impar el número de granos negros sea

cierta. Sin embargo, este no es el caso, pues el número de granos negros cambia de par a impar o de impar a par con cada ejecución de un paso del proceso. Pero también habría cero granos blancos cuando terminamos; es posible que la propiedad de permanecer par el número de granos blancos sea cierta. Y, en efecto, cada ejecución del proceso termina con dos granos blancos de menos en el recipiente o con la misma cantidad de granos blancos. Por lo tanto, el último grano es negro si inicialmente hay un número par de granos blancos; de otra forma, el último grano será blanco.

La solución al problema anterior es sumamente simple, pero extremadamente difícil de encontrar viendo sólo casos de prueba. El problema es más fácil de resolver si buscamos propiedades que permanecen ciertas. Una vez encontradas, estas propiedades nos permiten ver de una manera sencilla que hemos encontrado una solución. Este problema nos permite ver también el siguiente principio: hay que conocer y entender en detalle el enunciado del problema y las propiedades de los objetos que van a ser manipulados por el programa que resuelve el problema.