240205 Programación

Prácticas de laboratorio

José Ramón González de Mendívil
mendivil@unavarra.es

Departamento de Estadística, Informática y Matemáticas
Universidad Pública de Navarra
Campus de Arrosadía
31006 Pamplona

20 de agosto de 2021

Prólogo

Introducción

El lenguaje C fué desarrollado en la segunda mitad de los años 70 por Brian Kernighan y Dennis Ritchie. Es un lenguaje de alto nivel que está estrechamente vinculado con el sistema operativo Unix. No obstante, el lenguaje incluye bastantes facilidades propias de los lenguajes de bajo nivel. El lenguaje C nos permite estructurar los programas mediante acciones y funciones, y además, manipular de forma eficiente contenidos de información elemental como si se utilizase un lenguaje ensamblador. El precio a pagar por tener un lenguaje orientado a la eficiencia es que su compilador tiene mecanismos de detección de errores más laxos que otros lenguajes de alto nivel. El programador en este lenguaje debe ser muy consciente de la codificación que produce puesto que, la detección de errores puede ser ardua y costosa. El lenguaje C es un buen lenguaje para programar y un mal lenguaje para aprender a programar. No debe alarmarse por esta afirmación; la realidad es la que es. Todo lenguaje está sujeto a una curva de aprendizaje que hay que superar. Lo importante es que se habitúe a que no siempre los conceptos teóricos se plasman tal cual en la práctica, y que debe desarrollar hábitos y emplear metodologías adecuadas para elaborar programas correctos en cualquier lenguaje. En esta asignatura se introducen aspectos básicos del lenguaje C que se irán ampliando paulatinamente en otras asignaturas más avanzadas de la titulación como es el caso de Estructura de Datos, Sistemas Operativos y Algorítmia.

En este manual de prácticas procederemos a describir cómo traducir los conceptos del lenguaje algorítmico al lenguaje C. El lenguaje algorítmico está orientado a facilitar el desarrollo y la demostración de soluciones correctas por lo que, siguiendo una metodología adecuada, obtendremos programas en C legibles, correctos y eficientes.

Orientación de las prácticas

De todas formas, de nada sirve tener un lenguaje de programación para construir programas, si antes no se ha hecho un esfuerzo en diseñar y demostrar que estos programas y, más importante, los algoritmos en los que están basados, son realmente correctos.

En esta parte de la asignatura, por un lado, se practica en el diseño de algoritmos correctos para un problema dado, siguiendo los métodos vistos en la parte teórica de la asignatura; y por otro lado, los algoritmos obtenidos se codifican y se prueban utilizando el lenguaje C.

Cuando se trata de codificar un algoritmo en C, se puede emplear cualquier entorno de programación disponible pero en esta asignatura utilizaremos el entorno que ofrece el

Sesión 1: Codificación
Sesión 2: Derivación I
Sesión 3: Derivación II
Sesión 4: Acciones y funciones
Sesión 5: Tablas I
Sesión 6: Tablas II
Sesión 7: Tablas III
Sesión 8: Recursividad
Sesión 9: Autoevaluación
Examen práctico

sistema operativo Linux disponible en el laboratorio, y el compilador `gcc` (en su versión 3.2 o superior).

En cuanto a la escritura de un programa en C, el programador puede hacerlo de muy distintas formas. Un aspecto muy importante a tener en cuenta en la escritura de un programa es que sea fácil de leer, de entender y de corregir. Por ello, en los equipos de programadores es muy común utilizar unas ciertas **normas de estilo** para facilitar este proceso. En esta asignatura se propone un conjunto de normas de estilo que se introducen en este documento. **Todos los programas que se realicen en esta asignatura tienen que respetar estas normas de estilo**.

Como puede observarse en la tabla anterior, las clases prácticas se han organizado en 9 sesiones y un examen práctico. Se realizarán dos entregas para evaluar la asistencia y la participación activa en clase. Las entregas se corrigen, evalúan y se devuelven al estudiante corregidas para que observe sus errores tanto de codificación como de aplicación de las normas de estilo. En la última Sesión 9 se realiza un ejemplo de examen de forma que el estudiante pueda autoevaluar sus conocimientos y le sirva de preparación para el examen final de prácticas.

Cada ejercicio entregado tendrá una nota entre 0 y 10 puntos. Los criterios de corrección de los programas se basan en el uso adecuado de las normas de estilo, el cumplimiento de la especificación propuesta, la adecuada codificación del algoritmo previamente propuesto y verificado, y que sea posible compilar y ejecutar correctamente el programa. En el documento 'Criterios de evaluación de los contenidos prácticos', que se encuentra a disposición de los estudiantes en `MiAulario`, se explica con detalle la manera de puntuar los ejercicios prácticos que se recogen en las *Entregas* y en el examen práctico.

Documentación

Los documentos relacionados con la parte práctica de la asignatura de '240205 Programación' se encuentran disponibles en el sitio de la asignatura en `MiAulario`. Estos documentos son:

1. *Criterios de evaluación de los contenidos prácticos.* Detalla los criterios utilizados para corregir los programas en C.

2. *Normas de estilo en la codificación de programas*. Resumen de las normas de estilo de codificación que se proponen en esta asignatura.

3. *Ejecución en C*. Un resumen muy básico de cómo se compila y ejecuta un programa en C. Presenta varios comandos básicos de Linux.

4. *Prácticas de laboratorio*. Este documento.

Sesión 1

Codificación de algoritmos en C

El objeto de esta práctica es conocer la estructura general de un programa en C y familiarizarnos con las estructuras de control de este lenguaje. También, nos iniciaremos en la prueba de programas y en el estudio de su eficiencia, mediante el desarrollo de pruebas de ejecución y el conteo del número de operaciones elementales realizadas.

1.1. Estructura típica de un programa en C

La estructura general de un programa sencillo en C es la siguiente:

```
declaración de importaciones
declaración de constantes
declaración de nuevos tipos de datos
declaración de funciones
int main(void)
{
    declaración de variables del programa
    instrucciones ejecutables
    return 0;
}
```

En esta estructura se reconocen los elementos esenciales de la construcción algorítmica: constantes, tipos, variables e instrucciones ejecutables.

La declaración de importaciones se utiliza para informar al compilador sobre la declaración de funciones externas al programa que se hallan disponibles en otros módulos (ficheros con extensión `.h`). La definición de constantes, nuevos tipos y funciones las iremos estudiando en secciones y capítulos posteriores. Un elemento especial en la estructura anterior es la función `main()` (función principal) que es la primera función que se ejecuta al invocar la ejecución del programa. Esta función contiene la declaración de variables y las intrucciones ejecutables del programa principal. Si el programa termina sin error, se alcanza la ejecución de la sentencia `return 0` [1]. Por convenio, si todo ha ido bien, con el valor 0 se indica al entorno del Sistema Operativo que la ejecución finalizó correctamente.

[1] Puede evitar esta sentencia si no se necesita informar al Sistema Operativo. Basta con utilizar `void main(void)`.

Las llaves {} se emplean para encerrar el cuerpo de la función principal.

Los algoritmos que desarrollamos tienen un nombre (normalmente relacionado con el problema que resuelven). Su estructura general es como sigue

algoritmo ⟨nombre de algoritmo⟩
constante ... **fconstante** *declaración de constantes*
tipo...**ftipo** *declaración de tipos*
declaraciones de acciones y funciones
declaración de variables de la especificación del problema
$\{Pre : \ldots\}$
var
declaración de variables auxiliares que utiliza el algoritmo
fvar
...
instrucciones ejecutables
...
$\{Post : \ldots\}$
falgoritmo

Cuando escribimos un programa en C que codifica un algoritmo, adoptaremos como **norma de estilo** que el ⟨nombre de algoritmo⟩ coincida con el nombre del fichero que contiene el código del programa en C que codifica dicho algoritmo. Los ficheros que contienen código en C tienen la extensión `.c`.

Como hemos indicado en el Prólogo, nuestro objetivo es codificar en C aquellos algoritmos que hemos demostrado que son correctos a la hora de resolver un cierto problema.

1.2. Comentarios

En el texto de un programa en C podemos incluir comentarios para explicar o informar de ciertos aspectos sobre el código al lector del mismo. Para añadir comentarios a un programa en C es suficiente con encerrar el comentario entre los símbolos `/*` y `*/`.

```
/* De esta manera el compilador
ignora estas líneas al leerlas */
```

Los comentarios entre `/*` y `*/` no se pueden anidar y pueden abarcar varias líneas de texto. También es posible comentar únicamente una línea poniendo al comienzo de dicha línea los símbolos `//`.

```
// De esta otra manera el compilador ignora esta línea al leerla
```

Como **norma de estilo** evitaremos utilizar comentarios superfluos o meras decoraciones del texto del programa.

1.3. Expresiones literales

El contenido de un fichero que contiene un código escrito en C es meramente un texto que debe seguir unas reglas de escritura asociadas en este caso al lenguaje C. Una expresión de un literal se refiere a la forma en la cual representamos los valores de números, carácteres o cadenas de símbolos.

Literales de Enteros. En el caso de números enteros los representamos en su forma habitual como una secuencia de dígitos, por ejemplo, `40`, `0`, `124567`, son enteros. En C hay otras formas de representar números enteros: en su codificación en octal o en hexadecimal. Por ejemplo, `0277` es el número 191 en base decimal codificado en octal (base 8). El literal `0xff` representa el número entero 255 codificado en hexadecimal (base 16).

Literales de Caracteres. El código ASCII establece una relación entre los símbolos que aparecen en el teclado con números. Por ejemplo, el carácter Q (q mayúscula) que aparece en el teclado tiene el valor decimal 81, o el carácter que representa al dígito 0 en el teclado tiene asociado el valor decimal 48, según este código. El código ASCII también establece la relación entre carácteres de control y su respectivo valor, por ejemplo, el carácter de control ESC (escape) tiene el valor decimal 27, o el salto de línea (line feed) el valor decimal 10. Cuando escribimos un literal de la forma `'a'`, el lenguaje C lo interpreta como un número entero: el valor ASCII del carácter 'a'. Así que incluso podemos escribir la expresion `'a'+1` cuyo valor es 97+1. Algunos caracteres de control también los podemos escribir entre comillas con un formato especial, por ejemplo, `'\n'` representa el valor 10 correspondiente al salto de línea. Estos caracteres especiales también se denominan *secuencias de escape*.

Literales de Reales. En el lenguaje algorítmico empleamos números reales[2]. Su codificación se realiza por medio de lo que se denomina *aritmética en coma flotante*. Por ese motivo, en C se suele denominar *float* a un número real. Para representar estos números empleamos el punto decimal, por ejemplo, `20.0` es un número real (float) y no se interpreta como un número entero aunque tengan el mismo valor numérico. El número `0.02` ($2{,}0 \times 10^{-2}$) se puede representar de forma alternativa como `2.0e-2`.

Literales de Cadenas. Una cadena (*string*) se representa como una secuencia de caracteres encerrados entre comillas dobles. Por ejemplo, `''una cadena\n''`. Debe darse cuenta que el espacio en blaco, también es un carácter (`' '` valor 32). En una cadena pueden aparecer secuencias de escape. Algunas secuencias de escape frecuentes se dan en la tabla 1.1.

Literales de Booleanos. En el lenguaje algorítmico disponemos de los literales *veradero* y *falso* para representar los dos valores que forman el tipo Booleano. Sin embargo, en el lenguaje C no se dispone de literales para este tipo. En realidad el valor *falso* se corresponde con el número entero `0` y el valor *verdadero* con cualquier número entero distinto de `0`. Esto puede dar lugar a algunos errores si intenta mezclar expresiones booleanas con expresiones numéricas.

[2]En realidad son números racionales, pero los lenguajes de programación no suelen hacer esa diferenciación matemática.

Secuencia	significado
\a	produce un aviso audible
\n	produce un salto de línea
\r	el cursor pasa a la primera posición de la línea actual (retorno de carro)
\t	tabulador
\\	muestra la barra invertida
\”	muestra la comilla doble
\?	muestra el interrogante

Cuadro 1.1: Algunas secuencias de escape.

1.4. Las directivas y el preprocesador

Siguiendo con la estructura general de un programa en C, encontramos dos elementos que están relacionados estrechamente con el compilador: la definición de constantes y la declaración de importaciones. Las líneas del fichero `.c` que comienzan con el carácter # son especiales y se denominan *directivas*. En realidad el compilador (traductor del lenguaje C al lenguaje del ordenador) no trata estas líneas. Hay otro programa antes del compilador de C que trata esas directivas. Ese programa es el Preprocesador. El Preprocesador de C es un programa independiente que se denomina `cpp`. Así, cuando hablamos del compilador `gcc`, en realidad, sabemos que es un programa que controla varios programas en el proceso de traducción de C al código de la máquina. A continuación vamos a ver dos directivas importantes:

- `#define`, que permite declarar constantes.
- `#include`, que permite incluir el contenido de un fichero `.h` para la importación de las declaraciones de funciones que se encuentran en una librería (módulo).

1.4.1. Constantes mediante la directiva #define

En el lenguaje algorítmico una constante es un identificador al cuál le asociamos un determinado valor. En el algoritmo usamos

constante...**fconstante**

para acotar la declaración de las constantes. Por ejemplo, podemos declarar las constantes,

constante PI = 3.14, E = 2.78 **fconstante**

En el lenguaje C, las constantes se definen mediante la directiva

```
#define CONSTANTE valor
```

Cada línea `#define` sólo puede contener el valor de una constante, por ejemplo;

```
#define PI 3.14159
#define E 2.7812
```

Podemos definir constantes con literales numéricos o con cualquier otro tipo de literales. Como **norma de estilo** las constantes siempre se expresan con letras mayúsculas y sólo las declararemos con la directiva anterior. No se permite utilizar números mágicos que actúen como constantes sin haber sido declarados como constantes.

Considera el siguiente programa `prueba.c` escrito en C:

```
#define PI 3.14
int main(void)
{
    int a;
    a = PI;
    return 0;
}
```

Si ejecutamos `$cpp -P prueba.c`, el preprocesador lo transforma en el siguiente,

```
int main(void)
{
    int a;
    a = 3.14;
    return 0;
}
```

donde ya no aparece la directiva y toda aparición de `PI` ha sido sustituida por su literal `3.14`. No se puede cometer el error de pensar que la directiva `#define` sirve como una declaración de variable. Veremos más adelante las declaraciones de variables.

1.4.2. Declaración de importaciones

Todos los lenguajes de programación suministran *librerías* que incluyen un gran número de funciones que son útiles para tratar diferentes problemas prácticos de programación. En el lenguaje C, estos módulos se integran en el programa mediante la directiva `#include`. Por ejemplo, la más común en C es aquella que sirve para manipular la entrada/salida estándar, es decir, el teclado y el terminal:

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

Para trabajar con algunas funciones matemáticas se puede utilizar el módulo `math.h`:

```
#include <math.h>
```

En este módulo se incluyen las declaraciones de algunas funciones matemáticas comunes (ver la tabla 1.2).

Para poder utilizar realmente estas funciones no basta con poner `#include <math.h>`, se necesita algo más. El fichero que incluimos `math.h` contiene una descripción de las

Función C	Función matemática
`sqrt(x)`	raíz cuadrada de `x`
`sin(x)`	seno de `x`
`cos(x)`	coseno de `x`
`atan(x)`	arco tangente de `x`
`exp(x)`	el número e^x
`exp10(x)`	el número 10^x
`log(x)`	el logaritmo en base e de `x`
`pow(x,y)`	`x` elevado a `y`
`fabs(x)`	valor absoluto de `x`

Cuadro 1.2: Alguna de las funciones en la librería `math.h`.

cabeceras de las funciones matemáticas pero no su contenido. El contenido de dichas funciones (ya compiladas) se encuentra en otro fichero `/usr/lib/libm.a`. El preprocesador y el compilador utilizan el fichero `math.h` para comprobar que se están utilizando correctamente las funciones. Una vez realizada esta comprobación se procede a generar el código máquina, y en ese momento, hay que enlazar los códigos máquina de las funciones que se utilizan con el código máquina del programa principal. El encargado de resolver estas referencias externas es un programa que se denomina *enlazador* (*linker*). Así que hay que indicar al compilador `gcc` dónde se encuentra el fichero con el código de las funciones. Hay que compilar utilizando

```
$gcc programa.c -o programa /usr/lib/libm.a
```

o en forma simplificada

```
$gcc programa.c -o programa -lm
```

El proceso más completo que describe la compilación de un programa en C es el siguiente:

`programa.c` → Preprocesador → Compilador → Enlazador → código ejecutable.

En la asignatura de Estructura de Datos se requerirá utilizar librerás creadas por los estudiantes[3].

[3]La realización de un programa en un lenguaje de programación requiere de varios conocimientos que se van adquiriendo a lo largo de la titulación. La teoría de Procesadores de Lenguajes se encarga del estudio de los programas que realizan las traducciones de unos lenguajes a otros lenguajes. Un programa cuando se ejecuta sobre una determinada máquina está bajo el control del Sistema Operativo que es el programa encargado de gestionar los recursos de la máquina y de los procesos y ficheros que se encuentran en la misma. Es justamente la asignatura de Sistemas Operativos la que se encarga de estudiar este tipo de programas y su diseño. Los detalles de la codificación de datos e instrucciones, su ejecución en un ordenador corresponde a la asignatura de Arquitectura de ordenadores.

1.5. Tipos de datos simples

Un tipo de datos representa por una parte, los valores que pueden tomar las variables y expresiones del tipo, y por otra parte, las operaciones que se pueden utilizar con dicho tipo. En el caso de los tipos de datos simples, en el lenguaje algorítmico hemos presentado en teoría los que se indican en la tabla 1.3.

Nombre del tipo	Descripción
entero	representa el conjunto de números enteros
subrango de enteros	$A..B$ representa los enteros entre las constantes A y B, $A \leq B$
real	representa el conjunto de número racionales
caracter	representa el conjunto de caracteres imprimibles y de control
booleano	representa el conjunto cuyos dos únicos valores son *verdadero* y *falso*
enumerado	los valores del tipo se definen por el usuario

Cuadro 1.3: Tipos simples en el lenguaje algorítmico

En el lenguaje C sólo hay tres tipos de datos simples: enteros, flotantes y enumerados. Lo que sucede es que tienen diferententes variantes para poder ajustarse al tira y afloja entre rango/precisión y ocupación de memoria. En este curso es suficiente considerar los tipos simples de C indicados en la tabla 1.4.

Nombre del tipo	Número de bytes	Rango
`int`	4 bytes	$[-2^{31}, 2^{31}]$ complemento a 2
`unsigned int`	4 bytes	$[0, 2^{32} - 1]$ magnitud sin signo
`float`	4 bytes	max valor absoluto `3.40282347e38` coma flotante min valor absoluto `1.17549435e-38`
`char`	1 byte	$[-127, 128]$ complemento a 2
`unsigned char`	1 byte	$[0, 255]$ magnitud sin signo

Cuadro 1.4: Algunos tipos simples en el lenguaje C.

En el lenguaje C los valores enteros se han empleado habitualmente para codificar los valores booleanos. Como hemos indicado con anterioridad, el valor `0` representa el valor lógico *falso* y cualquier otro valor representa el valor lógico *verdadero*.

El tipo `char` aunque tenga un nombre que sugiere el tipo *caracter* designa un subconjunto de enteros. Lo que sucede es que es frecuente usar variables de tipo `char` para almacenar el valor ASCII correspondiente a un determinado carácter. Si `a` es una variable de tipo `char`, escribir la asignación `a= '0'` es equivalente a la asignación `a= 48`, pues el valor ASCII del literal `'0'` es `48`.

En el lenguaje C no se dispone del tipo subrango. C está pensado para ser eficiente y el tipo subrango requiere comprobar que los valores de las variables del tipo subrango no se salen del rango definido. Esto ralentiza la ejecución del programa. C prima la eficiencia respecto a la seguridad. No tener este tipo de datos obliga a una precisión extraordinaria en cuanto a la corrección de los programas escritos en C, sobre todo, cuando se trata de programas que manipulan grandes espacios de memoria ocupados por tablas. Postpondremos el estudio del tipo de datos estructurado mediante tablas para más adelante. En la siguiente sección veremos el tipo enumerado para introducir la definición de nuevos tipos

de datos simples.

Como **norma general** el estudiante deberá aplicar los tipos simples de C que mejor se ajusten a la solución del problema que esté tratando.

1.6. Definición de nuevos tipos de datos

En el lenguaje algorítmico podemos definir nuevos tipos de datos o bien dar un nombre (que nos parezca significativo) a un tipo de datos conocido. El tipo más común que puede crearse y definirse por el programador es el tipo *enumerado*. En dicho tipo los valores del tipo se crean por el programador. Por ejemplo, supongamos que queremos definir el estado de funcionamiento de un sistema de calefacción programable. El programador puede observar que el sistema se encuentra en cada momento en el estado encendido, apagado, o error. En el algoritmo puede emplear la siguiente declaración para crear justmente estos valores:

tipo *estado_sistema* = {ENCENDIDO, APAGADO, ERROR} **ftipo**

En C puedes hacer algo parecido, pero se debe dar un valor numérico a dichos elementos[4]. Esto es así porque en realidad en C el tipo enumerado es una agrupación de la declaración de términos constantes. La declaración anterior se codifica en C de la siguiente manera:

```
typedef enum {ENCENDIDO = 1, APAGADO, ERROR } estado_sistema;
```

Otro ejemplo común del tipo enumerado es el siguiente. Como C no tiene ninguna implementación primitiva del tipo booleano, para poder declarar variables de dicho tipo se debería definir previamente. Podemos emplear la siguiente declaración:

```
typedef enum {FALSO = 0, VERDADERO = 1} boolean;
```

Como **norma de estilo**, los nombres de los tipos que se declaran mediante `typedef` se escriben con letras minúsculas y deben ser nombres significativos para el programador.

Hasta esta sección hemos visto los elementos que componen, en un programa en C, la declaración de importaciones, la definición de constantes y la definición de nuevos tipos de datos. En lo que sigue nos centraremos en la declaración de variables y las sentencias e instrucciones ejecutables. La definición de funciones y su uso requerirá un capítulo aparte.

1.7. Declaración de variables

Los identificadores que utilizamos para construir los nombres que damos a los diferentes elementos (costantes, nuevos tipos, variables) son cadenas de hasta 32 caracteres alfa-

[4]Si no das valores numéricos C establece que el primero es 0 y los siguientes toman valores consecutivos 1, 2,..., etc. En realidad los valores que puedes definir pueden ser secuenciales o no.

numéricos (letras o números) y pueden contener el guión bajo _ para unir ciertas palabras. En general, todo identificador normalmente comienza con una letra aunque la norma indica que no puede comenzar con un dígito. Obviamente, hay algunos nombres que no se pueden utilizar porque están reservados para el propio lenguaje C: `auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unisgned, void, volatile y while`.

Como **norma de estilo** se utilizarán las letras minúsculas para designar los nombres de las variables y se reserva el uso de mayúsculas para nombres de constantes.

De esta manera, para un algoritmo que comience así

.....
x: *real*
a: *entero*
$\{Pre : a = A \ \wedge \ x = X\}$
var
    c: *caracter*
    es_final: *booleano*
fvar
.....

el código C correspondiente será

```
typedef enum {FALSO = 0, CIERTO = 1} boolean;
int main(void)
{
    float x;
    int a;
    char c;
    boolean es_final;
    .....
    return 0;
}
```

Se puede declarar una serie de variables del mismo tipo en una sola sentencia separando sus identificadores por comas, por ejemplo,

```
float x, y, z;
int a, b, c;
```

Observa que cada declaración termina con `;`. Una variable una vez declarada ya no puede cambiar de tipo.

El lenguaje C permite combinar la declaración de una variable y la asignación de un valor inicial. Sabemos que acceder a variables no inicializadas puede provocar errores en la ejecución. La inicialización de las variables es parte del diseño de la solución y su inicialización se realizará en el punto indicado por el algoritmo.

Como **norma de estilo** la declaración de variables sólo incluye su declaración y es conveniente que las variables del mismo tipo esten agrupadas, y las de tipos distintos vayan en líneas diferentes. Al pasar de un algoritmo, ya diseñado, a su codificación en C, no se cambiarán los nombres de las variables (o los parámetros de las funciones) declaradas por otros nombres nuevos, a no ser que sea estrictamente necesario.

1.7.1. Variables y sus direcciones en memoria

Todo elemento que utiliza un programa debe ser previamente declarado. Al declarar una variable le damos un *nombre* y definimos su *tipo*. El contenido de una variable en cada 'momento' es su *valor*. Este *valor* pertenece al conjunto de datos que define su *tipo*. Decimos 'momento' porque la ejecución de ciertas instrucciones, en un determinado momento de la ejecución del programa, puede hacer que este valor de la variable cambie. El valor de la variable es entonces el último valor 'escrito' en dicha variable. En un ordenador una variable se materializa porque ocupa un espacio físico real, así que podemos escribir o leer el *contenido* de este espacio. La memoria del ordenador se encarga de dotar de espacio a las variables. La abstracción que hacemos de la memoria del ordenador es símplemente verla como una sucesión de celdas numeradas. Los números de cada celda son su *dirección*. La cuestión es si podemos saber o no la dirección de memoria donde se almacena una variable. La respuesta es afirmativa ya que el lenguaje C dispone de un operador que te permite conocer la dirección de la variable. Por ejemplo, si la variable es `x` su dirección en memoria es `&x`. El operador `&` permite entonces conocer la dirección de memoria de una variable[5].

Una dirección de memoria es de algún tipo de datos, ya que se puede usar: su tipo es `unsigned int`. Puedes programar y ejecutar este programa en C para ilustrar lo indicado:

```
#include <stdio.h>
int main(void)
{
    int a, b;
    a = 0;
    b = a + 8;
    printf("Direccion de a: \n", (unsigned int)&a);
    printf("Direccion de b: \n", (unsigned int)&b);
    return 0;
}
```

Cuando escribimos en C una instrucción de asignación como `x = b + 9*c` debemos ser conscientes de que en la parte derecha de la asignación se accede a los valores de las variables `b` y `c` para calcular el valor de la expresión `b + 9*c`. El valor de la expresión se copia en la dirección de memoria asociada a la variable (asignada) `x`. Así, al terminar su

[5]La cuestión es para qué todo esto... en algunas ocasiones sólo nos interesa conocer el valor de una variable o de una expresión para poder utilizarlo posteriormente, pero en otras ocasiones, nos interesa cambiar una variable por otra como si fuesen la misma variable. Para conseguir este efecto podemos remplazar la dirección de la primera variable por la segunda y así, todo efecto que se imagine sobre la primera variable está siendo realizado sobre la segunda variable. El nombre técnico que podemos dar a estas formas de uso de las variables es: uso de variable *por valor* o uso de variable *por referencia*. La *referencia* de una variable también es el nombre que recibe la dirección de memoria de la variable.

ejecución el valor de la variable `x` cambia pero no cambian los valores de las variables `b` y `c`.

1.8. Entrada y presentación de datos

En el lenguaje algorítmico la Precondición establece las condiciones de los valores iniciales de las variables que intervienen en la especificación del problema. El predicado Postcondición establece las condiciones que deben cumplir las variables que intervienen en el problema al final de la ejecución de cualquier algoritmo que resuelve dicho problema. Obviamente, en el lenguaje algorítmico no se indica de dónde se toman los valores iniciales de las variables del problema, ni tampoco cómo se presentan los valores finales de las variables. A la hora de codificar el algoritmo en C debemos preocuparnos por estos detalles.

La salida de datos en C la realizamos presentando éstos en la pantalla utilizando la función `printf(.)`. Dicha función permite escribir en la pantalla cadenas de caracteres. Para intercalar los valores de las expresiones o variables en la presentación en la pantalla se utilizan marcas de formato como las indicadas en la tabla 1.5.

Tipo	marca de formato
`int`	%d
`unsigned int`	%u
`float`	%f
`float`	%e (científica)
`char`	%hhd
`unsigned char`	%hhu
`char`	%c (como carácter)
`string`	%s (cadena de caracteres)

Cuadro 1.5: Algunas marcas de formato.

Un ejemplo sencillo. Suponer `int a;float b;`

```
printf("Sea un entero: %d, y un real: %f \n", a+1, 3.4*b);
```

En la pantalla se muestra:

```
Sea un entero: 2, y un real: 3.4
```

si el valor de `a` es 1 y el valor de `b` es 1.0, cuando se ejecuta dicha sentencia.

Como **norma de estilo** no se permite que la cadena que se utiliza como argumento en `printf()` ocupe más de una línea en el texto del programa.

La entrada de datos la realizamos utilizando el teclado del terminal. La función `scanf()` disponible en `stdio.h` permite leer datos por el teclado y construir los valores para la variable objetivo que queremos actualizar su valor. Si deseas leer por el teclado el

valor para una variable entera `x` puedes utilizar:

```
scanf("%d", &x);
```

La función requiere una cadena con la marca de formato y la dirección de memoria correspondiente a la variable cuyo valor se desea actualizar. Se requiere acudir a un manual o libro de texto sobre C más completo, para obtener los detalles sobre el funcionamiento de las funciones presentadas en esta sección.

1.9. La instrucción de asignación

La instrucción básica del lenguaje algorítmico es la asignación:

variable := expresión

En el lenguaje algorítmico empleamos diferentes operadores para construir las expresiones cuyo valor se asigna a la variable en la instrucción de asignación. Tanto la variable asignada como la expresión deben ser del mismo tipo de datos. En las tablas siguientes indicamos los operadores empleados en el lenguaje algorítmico y su codificación en C.

Operador	Significado	Operador en C
Operadores aritméticos		
`+`	suma	`+`
`*`	producto	*
`-`	diferencia (o cambio de signo)	`-`
`div`	división entera	`/`
`mod`	resto de la división entera	`%`
`/`	división real	`/`
Operadores lógicos		
$\wedge$	conjunción lógica (Y lógica)	`&&`
$\vee$	disjunción lógica (O lógica)	||
$\neg$	negación lógica	`!`
$\equiv$	igualdad lógica	`==`
Operadores relacionales		
$=$	igual que	`==`
$\neq$	distinto que	`!=`
$<$	menor que	$<$
$>$	mayor que	$>$
$\leq$	menor o igual que	$<=$
$\geq$	mayor o igual que	$>=$

Cuadro 1.6: Operadores

Su definición y reglas de funcionamiento son análogas a las mostradas para sus equivalentes en la notación algorítmica.

La instrucción de asignación en C presenta el aspecto

```
variable = expresión;
```

De nuevo la `expresión` debe tomar un valor del tipo de la variable. La sentencia de asignación (como prácticamente todas las sentencias en C) debe terminar en `;`. Es conveniente dejar un espacio entre el signo de asignación en C, `=` , y la `expresión` para evitar algunos efectos no deseados con algunos operadores. En C no se debe confundir la asignación `=` con la igualdad `==`.

Es conveniente compilar un programa en C con la opción `gcc -Wall ...` con el objetivo de detectar algunos errores semánticos.

El lenguaje C dispone de un grupo más amplio de operadores y de formatos para la instrucción de asignación: operadores para manipulación de bits, distintas formas de operadores de asignación, operador de tamaño `sizeof()`, operación de coversión de tipos, operador condicional, y operadores de incremento/decremento. Aunque algunos de estos operadores puedan resultar de interés, no son objeto de este primer contacto con C. Por tanto, como **norma de estilo** sólo se utilizarán los operadores indicados anteriormente salvo en el caso de que alguna práctica requirera el uso de alguna facilidad de C que sea indicada por el profesor.

1.10. La composición secuencial

En el lenguaje algorítmico empleamos el punto y coma ';' como constructor de la composición secuencial. En el lenguaje C, la manera de secuenciar intrucciones consiste en escribirlas consecutivamente ya que el `;` en C es un terminador de sentencias y **no** un separador de sentencias (que indica la sentencia que se va ejecutar a continuación).

Como **norma de estilo** para aumentar la legibilidad de los programas se deben escribir instrucciones diferentes en líneas diferentes y respetar el uso habitual de la indentación.

1.11. La composición alternativa

La traducción a C de la estructura algorítmica

si $\langle$condición$_1\rangle \longrightarrow \langle$instrucción$_1\rangle$
[] $\langle$condición$_2\rangle \longrightarrow \langle$instrucción$_2\rangle$
...
[] $\langle$condición$_{n-1}\rangle \longrightarrow \langle$instrucción$_{n-1}\rangle$
[] $\langle$condición$_n\rangle \longrightarrow \langle$instrucción$_n\rangle$
fsi

se realiza mediante la instrucción que sigue con el formato siguiente (**norma de estilo**)

```
if (condición1)
{
    ⟨instrucción1⟩
}
else if (condición2)
{
```

```
    ⟨instrucción2⟩
}
    ...
else if (condición_{n-1})
{
    ⟨instrucción_{n-1}⟩
}
else
{
    ⟨instrucción_n⟩
}
```

Como los algoritmos que escribimos en clase corresponden a la construcción alternativa exclusiva, la última condición es la negación de todas las demás, entonces recurrimos a la palabra `else`.

Si para algún i la secuencia de instrucciones es vacía (continuar) podemos prescindir de ella. En el caso $n = 2$ la instrucción se reduce a

```
if (condición1)
{
    ⟨instrucción1⟩
}
else
{
    ⟨instrucción2⟩
}
```

y si la secuencia de instrucciones 2 es vacía entonces la codificación queda de la forma

```
if (condición1)
{
    ⟨instrucción1⟩
}
```

Cuando la secuencia de instrucciones representada anteriormente por ⟨instrucción$_i$⟩ contiene únicamente una instrucción no es obligatorio encerrarla entre llaves. Pero siempre es obligatorio encerrar entre paréntesis las condiciones (condición$_i$) por muy sencillas que sean éstas. En todos los casos se emplea la indentación para indicar la composición de instrucciones.

1.12. La composición iterativa

En el lenguaje C aparecen tres formas de composición iterativa: la instrucción `while` que traduce el ya conocido **mientras**; el `do...while` cuya semántica implica la ejecución del cuerpo de instrucciones del bucle al menos una vez; y el `for` que es otra manera de organizar el **mientras**. Para traducir a C

mientras ⟨condición⟩ **hacer**
    ⟨secuencia de instrucciones⟩
fmientras

escribimos

```
while (condición)
{
    ⟨secuencia de instrucciones⟩
}
```

Nuevamente los paréntesis son obligatorios y las llaves pueden surpimirse en el caso de que la secuencia de instrucciones sólo contenga una instrucción.

El `do...while` tiene la siguiente forma

```
do
{
    ⟨secuencia de instrucciones⟩
} while (condición);
```

La diferencia entre ambas es que en el caso del `do...while` la condición de continuación se evalúa después de ejecutar la secuencia de instrucciones, y por lo tanto, la ejecuta como mínimo una vez.

En el lenguaje C se dispone también del bucle `for`. Tiene la siguiente estructura:

```
for (inicializacion; condicion; incremento)
{
    ⟨secuencia de instrucciones⟩
}
```

Es equivalente a este fragmento de código:

```
⟨inicializacion⟩;
while (condicion)
{
    ⟨secuencia de instrucciones⟩;
    ⟨incremento⟩;
}
```

1.13. Actividades a realizar

Con el fin de tomar contacto con las construcciones del lenguaje C vistas, se propone la codificación de varios algoritmos. La última actividad consiste en comparar la eficiencia de dos de los algoritmos propuestos.

1.13.1. Composición alternativa

Según las **normas de estilo**, todos los programas deben ejecutarse tantas veces como el usuario desee sin salir del propio programa. Con este fin, se debe utilizar una estructura `do ... while` en la que la condición de terminación utilice una variable de tipo carácter para dar la orden de salida del programa.

Para empezar vamos a ver un ejemplo de cómo codificar en lenguaje C el siguiente algoritmo para calcular el máximo de dos números reales.

algoritmo maximo;
a, b: *real*
m: *real*
$\{Pre : a = A \wedge b = B\}$
si a $\geq$ b $\longrightarrow$ m := a
[] a $<$ b $\longrightarrow$ m := b
fsi
$\{Post : m = max(A, B)\}$
falgoritmo

El programa en C siguiendo las normas de estilo exigidas en esta asignatura sería el siguiente.

```
1) // Programa: maximo
2) // nombre del autor
3) // fecha de realizacion
4) /* Este programa calcula el mayor
5) de dos numeros reales */
6) #include <stdio.h>
7) int main (void)
8) {
9)    float a,b,m;
11)   char res;
12)   printf("Program: maximo \n");
13)   printf("nombre del autor \n");
14)   printf("fecha de realizacion \n");
15)   printf("Programa que calcula el mayor ... \n");
16)   printf(" ... de dos numeros reales. \n");
17)   do
18)   {
19)        printf("El programa trabaja con numeros reales.\n");
20)        printf("Dame el primer numero real: \n");
21)        scanf("%f",&a);
22)        printf("Dame el segundo numero real: \n");
23)        scanf("%f",&b);
24)        if (a>=b)
25)             m = a;
26)        else
27)             m = b;
28)        printf("El mayor numero de los dos es:%f\n",m);
29)        printf("Deseas continuar? s/n: ");
30)        scanf(" %c",&res);
31)   } while(res == 's' || res == 'S');
32)   return 0;
33) }
```

Todos los programas escritos deben seguir las siguientes **normas de estilo**:

Mediante comentarios en el formato indicado en las líneas 1 a 5 se incluye, el nombre del programa, que debe coincidir con el nombre del fichero `.c`, el nombre del autor, la

fecha de realización y una breve descripción de lo que hace el programa. La línea 6 incluye los módulos a utilizar, normalmente `<stdio.h>`. Las líneas 9 a 11 presentan la declaración de variables. La estructura `do...while` (líneas 17, 31) se utiliza para poder repetir el programa cuantas veces sea necesario. Previamente y sólo la primera vez, se presenta en la pantalla la misma información sobre el autor, fecha de realización y descripción del programa (líneas 12 a 16). En el fichero original no deben quedar rotas las líneas de los `printf()`. Las líneas 19 a 23 realizan la toma de datos que debe ser breve, precisa y clara para el usuario cuando la utilice. Debe informar del tipo de datos que debe de introducir así como sus condiciones. La traducción de las instrucciones del algoritmo se dan en las líneas 24 a 27 según el formato de traducción dado en las secciones anteriores de este documento. La línea 28 presenta la solución que debe ser de nuevo breve, precisa y clara para el usuario. Las líneas 29 a 31 corresponden a la decisión de continuar o no con la ejecución. En todos los programas que se desarrollen es necesario indicar de manera precisa cómo se introducen los datos e informar de los resultados obtenidos. Procure no dejar líneas vaciás en el programa a no ser que trate de separar alguna sección de la estructura general del programa.

Para controlar la presentación del valor de una variable `m` de tipo real, se utilizan los formatos de salida que proporciona las función `printf()`. Consulte la bibliografía.

Ejercicio 1. Ordenación de 3 números enteros

Codifica en lenguaje C el siguiente algoritmo para ordenar tres números de menor a mayor.

algoritmo ordena3;
x, y, z: *entero*
p, s, t: *entero*
$\{Pre : x = X \wedge y = Y \wedge z = Z\}$
si x $\leq$ y $\longrightarrow$
si y $<$ z $\longrightarrow$ p:= x; s:= y; t:= z
[] (y $\geq$ z) $\wedge$ (z $\geq$ x) $\longrightarrow$ p:= x; s:= z; t:= y
[] z $<$ x $\longrightarrow$ p:= z; s:= x; t:= y
fsi
[] x $>$ y $\longrightarrow$
si x $<$ z $\longrightarrow$ p:= y; s:= x; t:= z
[] (y $\leq$ z) $\wedge$ (z $\leq$ x) $\longrightarrow$ p:= y; s:= z; t:= x
[] z $<$ y $\longrightarrow$ p:= z; s:= y; t:= x
fsi
fsi
$\{Post : (p, s, t) \in \text{Perm}(X, Y, Z) \wedge p \leq s \leq t\}$
falgoritmo

Idea seis casos de prueba tales que entre todos consigan ejecutar todas las instrucciones del programa.

Ejercicio 2. Otra ordenación de 3 números enteros

Codifica en lenguaje C el siguiente algoritmo, que sigue una estrategia distinta que el anterior, para ordenar tres números de menor a mayor.

algoritmo ordena3b;
x, y, z: *entero*
p, s, t: *entero*
$\{Pre \equiv x = X \wedge y = Y \wedge z = Z\}$
var
    aux: *entero*;
fvar
    p:= x; s:= y; t:= z;
    si
        p > s $\longrightarrow$ aux:= p; p:= s; s:= aux
      [] p $\leq$ s $\longrightarrow$ **continuar**
    fsi;
    si
        p > t $\longrightarrow$ aux:= p; p:= t; t:= aux
      [] p $\leq$ t $\longrightarrow$ **continuar**
    fsi;
    si
        s > t $\longrightarrow$ aux:= s; s:= t; t:= aux
      [] s $\leq$ t $\longrightarrow$ **continuar**
    fsi
$\{Post \equiv (p, s, t) \in \text{Perm}(X, Y, Z) \wedge p \leq s \leq t\}$
falgoritmo

Idea casos de prueba tales que entre todos consigan ejecutar todas las instrucciones del programa. ¿Cuántos casos son necesarios?

Nota: Es interesante realizar la derivación de los dos algoritmos anteriores ya que siguen estrategias de diseño bastante diferentes. El primero es un diseño a fuerza bruta mientras que el segundo se basa en una descomposición más inteligente de la postcondición.

1.13.2. Composición iterativa

Cuando se trata de estudiar la eficiencia de un programa es necesario desarrollar pruebas de ejecución para contar el número de operaciones elementales realizadas. En los algoritmos que se basan en la composición iterativa se suele contar el número de iteraciones que realiza el programa.

Por ejemplo, consideramos el siguiente algoritmo para la multiplicación eficaz de dos números enteros.

algoritmo prodef;
x, y: *entero*
p: *entero*
$\{Pre : x = X \wedge X \geq 0 \wedge y = Y \wedge Y \geq 0\}$
    p := 0;
$\{Inv : X \cdot Y = p + x \cdot y \wedge y \geq 0\}$
    mientras y > 0 **hacer**
        si y **mod** $2 = 1 \longrightarrow$ p:= p + x; y:= y - 1
          [] y **mod** $2 = 0 \longrightarrow$ x:= x + x; y := y **div** 2
        fsi
    fmientras
$\{Post : p = X \cdot Y\}$
falgoritmo

La codificación en C ateniéndonos a las normas de estilo se muestra a continuación. Obsérvese que cualquier programa que se realice en este laboratorio debe advertir en la toma de datos de las limitaciones que deben cumplir éstos, normalmente según especifique la precondición o el enunciado del ejercicio, para que la ejecución y el resultado sean correctos.

```
// Program: prodef
// nombre del autor
// fecha de realización
/* Este programa realiza el producto eficaz
de dos numeros enteros positivos*/
#include <stdio.h>
int main (void)
{
   int x, y, p, cont;
   char res;
   printf("Program: prodef\n");
   printf("nombre del autor \n");
   printf("fecha de realización \n");
   printf("Este programa realiza el producto eficaz ... \n");
   printf("de dos numeros enteros positivos.\n");
   do
   {
      printf("El programa trabaja con numeros enteros.\n");
      printf("Dame el primer numero entero positivo: \n");
      scanf("%d",&x);
      printf("Dame el segundo numero entero positivo: \n");
```

```
        scanf("%d",&y);
        cont = 0;
        p = 0;
        while (y > 0)
        {
           if (y % 2 == 1)
           {
              p = p + x;
              y = y - 1;
           }
           else
           {
              x = x + x;
              y = y / 2;
           }
           cont = cont + 1;
        }
        printf("El producto eficaz de los dos es: %d\n",p);
        printf("El numero de iteraciones realizadas es: %d\n",cont);
        printf("Deseas continuar? s/n: ");
        scanf(" %c",&res);
     } while(res == 's' || res == 'S');
     return 0;
}
```

Para contar el número de veces que se repite el bucle del algoritmo simplemente utilizamos la variable `cont` previamente declarada e inicializada a `0`. Al final del programa, mostramos junto con el resultado el número de iteraciones realizadas.

Ejercicio 3. Máximo común divisor de dos números enteros positivos

Codifica en lenguaje C el siguiente algoritmo para el cálculo del máximo común divisor de dos números enteros positivos. El programa debe advertir en la toma de datos de las limitaciones que deben cumplir éstos para que la ejecución y el resultado sean correctos.

algoritmo mcd;
x, y: *entero*
$\{Pre : x = X \wedge X > 0 \wedge y = Y \wedge Y > 0\}$
$\{Inv : \text{mcd}(x, y) = \text{mcd}(X, Y) \wedge x > 0 \wedge y > 0\}$
    mientras x $\neq$ y **hacer**
        si x > y $\longrightarrow$ x:= x - y
          [] x < y $\longrightarrow$ y:= y - x
        fsi
    fmientras
$\{Post : x = \text{mcd}(X, Y)\}$
falgoritmo

Ejercicio 4. Máximo común divisor eficaz

Codifica en lenguaje C el siguiente algoritmo para el cálculo eficaz del máximo común divisor de dos números enteros positivos. El programa debe advertir en la toma de datos

de las limitaciones que deben cumplir estos para que la ejecución y el resultado sean correctos.

algoritmo mcdef;
x, y: *entero*
$\{Pre : x = X \wedge X > 0 \wedge y = Y \wedge Y > 0\}$
var aux: *entero*; **fvar**
    si x $\geq$ y $\longrightarrow$ **continuar**
      [] x $<$ y $\longrightarrow$ aux:= y; y:= x; x:= aux
    fsi;
$\{Inv : \text{mcd}(X, Y) = \text{mcd}(x, y) \wedge x \geq y > 0\}$
    mientras x **mod** y $\neq 0$ **hacer**
        aux:= y;
        y:= x **mod** y;
        x:= aux
    fmientras
$\{Post : y = \text{mcd}(X, Y)\}$
falgoritmo

Nota: Piensa si es realmente necesario exigir en el invariante la condición $x \geq y > 0$. ¿Ofrece alguna ventaja la alternativa inicial si no es necesaria dicha condición?.

Ejercicio 5. Comparación de algoritmos para el cálculo del máximo común divisor

Compara la eficacia de los dos algoritmos anteriores. Con este fin has de elaborar un programa con nombre `mcdcomp`, que pida una sola vez los números enteros positivos para el cálculo del m.c.d. y aplique sobre esos valores ambos algoritmos de m.c.d.. El programa debe mostrar por pantalla el resultado y el número de iteraciones que ha realizado cada algoritmo.

Nota: Imprime los programas que has realizado una vez que hayas comprobado que funcionan correctamente. Revisa las normas de estilo y corrige el programa si es necesario. Realiza capturas de pantalla de la ejecución en algunos casos. Genera los pdfs de los ficheros para guardarlos con los programas originales. Comprueba que has realizado todas las cuestiones que se plantean en los ejercicios. Si tienes alguna duda consulta al profesor en las prácticas o en el horario de tutorías.

Nota: Si encuentras algún error en este documento indícaselo al profesor de prácticas o escribe a `mendivil@unavarra.es`

Sesión 2

Derivación de algoritmos I. Sustitución de expresiones

El objeto de esta práctica es trabajar con la derivación formal de algoritmos y adquirir soltura en la codificación en C de los algorítmos obtenidos.

2.1. Actividades a realizar

En esta sesión vamos a derivar varios algoritmos utilizando invariantes construidos mendiante la sustitución de expresiones de la postcondición por nuevas variables. En algunos casos será necesario hacer reforzamientos de los invariantes. Finalmente, codificaremos en C los algoritmos obtenidos.

Ejercicio 1. Una suma agujereada

Dado el predicado

$$I : s = \sum_{0 \le i \le k} x^{\frac{1}{2}i(i+1)} \wedge\ n = N\ \wedge\ x = X \wedge\ p = x^{\frac{1}{2}k(k+1)} \wedge\ q = x^k \wedge\ 0 \le k \le n$$

y la especificación

n: *entero*;
x, s: *real*;
$\{Pre : n = N\ \wedge\ N \ge 0\ \wedge\ x = X\ \wedge\ X > 0\}$
suma_agujereada
$\{Post : s = \sum_{0 \le i \le n} x^{\frac{1}{2}i(i+1)}\ \wedge\ n = N\ \wedge\ x = X\}$

Construye un algoritmo que resuelva el problema planteado por la especificación anterior. El algoritmo ha de emplear una única composición iterativa cuyo invariante ha de ser el predicado I.

Ejercicio 1.A. Derivación del algoritmo

Antes de comenzar es conveniente plantear un ejemplo sencillo para comprender la especificación del algoritmo. Si el algoritmo comienza en un estado inicial con los valores 4 y 5,0 en las variables n y x repectivamente, debe terminar en un estado tal que $n = 4$,

$x = 5{,}0$ (estas variables no cambian su valor inicial) y la variable s tiene que tomar el valor $s = \sum_{0 \leq i \leq n} x^{\frac{1}{2}i(i+1)} = 5{,}0^0 + 5{,}0^1 + 5{,}0^3 + 5{,}0^6 + 5{,}0^{10}$. La suma que se refleja en la variable s está 'agujereada' porque no contiene a todas las potencias de x de forma consecutiva, como se puede observar en el ejemplo.

• *Propuesta de invariante.* En el ejercicio nos dan como predicado invariante el predicado I que, además de contener expresiones para las variables s, n y x, también tiene definidas las variables k, p y q. Las variables p y q son de tipo real y la variable k es de tipo entero. El predicado invariante se ha obtenido a partir de la postcondición *Post* sustituyendo la variable n por la nueva variable k, cuyo rango de valores es $0 \leq k \leq n$. Las expresiones que contienen las variables p y q son reforzamientos que sirven para ayudar a obtener la solución de forma más simple.

• *Cálculo de la condición de continuación.* Se debe cumplir $I \wedge \neg B \Rightarrow Post$. En este caso, si añadimos a I la condición $k = n$, esto asegura que la postcondición se cumple. Entonces, $\neg B : k = n$, luego $B : k \neq n$. Como el invariante I asegura que $0 \leq k \leq n$, podemos cambiar la desigualdad anterior y escribir $B : k < n$.

• *Cálculo de las instrucciones de inicio.* El algoritmo tiene que terminar cuando $k = n$. Como en el invariante siempre se cumple $0 \leq k \leq n$, podemos buscar las instrucciones más sencillas de inicio, por ejemplo, haciendo $k = 0$. Observa que las variables n y x no cambian su valor inicial, por lo que será necesario obtener los valores iniciales para las variables p y q que hagan que se cumpla el invariante I antes de comenzar el bucle. Si sustituimos en I la variable k por su valor inicial 0 entonces

$$s = \sum_{0 \leq i \leq 0} x^{\frac{1}{2}i(i+1)} = x^0 \wedge\ p = x^0 \wedge\ q = x^0 \wedge\ 0 \leq 0 \leq n$$

es decir, los valores iniciales son $s = 1$, $p = 1$ y $q = 1$. Para obtener estos valores tenemos que realizar las siguientes instrucciones de asignación:

k:= 0; s:= 1.0; p:= 1.0; q:= 1.0;

De esta manera podemos comprobar que se cumple $Pre \Rightarrow (((I)^q_{1,0})^p_{1,0})^s_{1,0})^k_0$.

$$\begin{aligned} &Pre : n = N \ \wedge\ N \geq 0 \ \wedge\ x = X \ \wedge\ X > 0 \\ &(((I)^q_{1,0})^p_{1,0})^s_{1,0})^k_0 : 1{,}0 = \sum_{0 \leq i \leq 0} x^{\frac{1}{2}i(i+1)} \wedge\ n = N \ \wedge\ x = X \ \wedge \\ &\qquad\qquad \wedge\ 1{,}0 = x^{\frac{1}{2}0(0+1)} \wedge\ 1{,}0 = x^0 \wedge\ 0 \leq 0 \leq n \end{aligned}$$

Suponemos que las condiciones en la precondición son verdaderas. Empezamos por el final. Se cumple $0 \leq 0 \leq n$ ya que en la precondición se cumple $n = N \ \wedge\ N \geq 0$. Se cumple $1{,}0 = x^0$, ya que el valor de x^0 es uno para cualquier x mayor que cero. Por la misma razón se cumple $1{,}0 = x^{\frac{1}{2}0(0+1)}$. Se cumplen $n = N \wedge x = X$ porque se cumple la precondición. Finalmente, se cumple $1{,}0 = \sum_{0 \leq i \leq 0} x^{\frac{1}{2}i(i+1)}$, ya que el sumatorio sólo tiene un elemento, el $x^{\frac{1}{2}0(0+1)}$, correspondiente a $i = 0$ y, como ya hemos visto, su valor es 1,0 luego esta última igualdad también se cumple.

Hasta el momento tenemos construido parte del algoritmo

algoritmo suma_agujereada;
n: *entero*;
x, s: *real*;
$\{Pre : n = N \ \wedge \ N \geq 0 \ \wedge \ x = X \ \wedge \ X > 0\}$
var
k: *entero*;
p, q: *real*;
fvar
k:= 0; s:= 1.0; p:= 1.0; q:= 1.0;
$\{I\}$ $\{cota...\}$
mientras k $\neq$ n **hacer**
'restablecer'
'avanzar'
fmientras
$\{Post : s = \sum_{0 \leq i \leq n} x^{\frac{1}{2}i(i+1)} \ \wedge \ n = N \ \wedge \ x = X\}$
falgoritmo

• *Cálculo de las instrucciones de avanzar*. La condición de terminación es $k = n$. La variable n no cambia su valor inicial N a lo largo de la ejecución y la variable k comienza la ejecución del bucle con el valor 0. Por lo tanto, el valor de la variable k tiene que crecer hasta alcanzar el valor de n para poder terminar la ejecución del algoritmo. Proponemos como instrucción de avanzar k:= k +1.

• *Cálculo de las instrucciones de restablecer*. Se tiene que cumplir

$$\{I \wedge B\} \text{ 'restablecer' } \{(I)^k_{k+1}\}$$

· Primero, escribimos estos predicados
$I \wedge B : s = \sum_{0 \leq i \leq k} x^{\frac{1}{2}i(i+1)} \wedge \ n = N \wedge x = X \wedge p = x^{\frac{1}{2}k(k+1)} \wedge q = x^k \wedge 0 \leq k < n$
$(I)^k_{k+1} : s = \sum_{0 \leq i \leq k+1} x^{\frac{1}{2}i(i+1)} \wedge n = N \wedge x = X \wedge p = x^{\frac{1}{2}(k+1)(k+2)} \wedge q = x^{k+1} \wedge 0 \leq k+1 \leq n$

· Segundo, comprobamos lo que se cumple en $(I)^k_{k+1}$ suponiendo que se cumple $I \wedge B$.

Se cumplen $n = N \wedge x = X$ en $(I)^k_{k+1}$ siempre que se cumplen en $I \wedge B$. Se cumple $0 \leq k+1 \leq n$ ya que en $I \wedge B$ se cumple $0 \leq k < n$.

· Tercero, relacionamos las expresiones de las variables que cambian su valor en $(I)^k_{k+1}$ teniendo en cuenta el valor que tenían antes en $I \wedge B$:

Para la variable s: $\sum_{0 \leq i \leq k+1} x^{\frac{1}{2}i(i+1)} = \sum_{0 \leq i \leq k} x^{\frac{1}{2}i(i+1)} + x^{\frac{1}{2}(k+1)(k+2)}$

El nuevo valor de la variable s es el que tenía antes (en $I \wedge B$) más el nuevo valor (en $(I)^k_{k+1}$) de la variable p, es decir, s:= s+ p.

Para la variable p: $x^{\frac{1}{2}(k+1)(k+2)} = x^{\frac{1}{2}k(k+1)}x^{k+1}$

El nuevo valor de la variable p es el que tenía antes (en $I \wedge B$) por el nuevo valor (en $(I)^k_{k+1}$) de la variable q, es decir, p:= p * q.

Para la variable q: $x^{k+1} = x^k x$.

El nuevo valor de la variable q es el que tenía antes (en $I \wedge B$) por el valor (en $I \wedge B$) de la variable x, es decir, q:= q * x.

Llegados a este punto, sabemos que las instrucciones de restablecer son las siguientes, en el orden que se indica a continuación:

// inicialmente n, x, p, q, s tienen el valor expresado en $I \wedge B$
q:= q * x; //q tiene ahora el valor expresado en $(I)^k_{k+1}$
p:= p * q; //p tiene ahora el valor expresado en $(I)^k_{k+1}$
s:= s + p; //s tiene ahora el valor expresado en $(I)^k_{k+1}$
//n y x mantienen su valor (tal como expresa el invariante I)

• *Diseño del algoritmo.* Con la derivación realizada anteriormente, una vez conocidas las intrucciones, podemos escribir el algoritmo siguiendo la estructura de la solución para algoritmos iterativos.

algoritmo suma_agujereada;
n: *entero*;
x, s: *real*;
$\{Pre : n = N \ \wedge \ N \geq 0 \ \wedge \ x = X \ \wedge \ X > 0\}$
var
k: *entero*;
p, q: *real*;
fvar
k:= 0; s:= 1; p:= 1; q:= 1;
{Invariante I} {*cota...*}
mientras k $\neq$ n **hacer**
q:= q * x;
p:= p * q;
s:= s + p;
k:= k + 1
fmientras
$\{Post : s = \sum_{0 \leq i \leq n} x^{\frac{1}{2}i(i+1)} \ \wedge \ n = N \ \wedge \ x = X\}$
falgoritmo

• *Cálculo de la función de cota.* La existencia de la función de cota garantiza que el algoritmo termina. Normalmente para obtener la función de cota nos fijamos en la condición de continuación y en las instrucciones del cuerpo del bucle que afectan a las variables de esta condición. De las variables n y k que toman parte en la condición de continuación, $B : k \neq n$, la variable n no cambia su valor, tal y como queda reflejado en el invariante I, y la variable k crece en cada paso del bucle. Así, en cada paso del bucle decrece el valor $n - k$. Se puede proponer como función de cota, $cota = n - k$. Vamos a comprobar que realmente es una buena función de cota demostrando las siguientes condiciones:

a) $I \wedge B \Rightarrow n - k > 0$
b) $\{Inv \wedge B \wedge n - k = T\}$ 'restablecer'; $k := k + 1$ $\{n - k < T\}$

· Primera, suponemos que se cumple el antecedente $I \wedge B$ vamos a comprobar si entonces también se cumple el consecuente. Escribimos de nuevo el predicado

$$I \wedge B : s = \sum_{0 \leq i \leq k} x^{\frac{1}{2}i(i+1)} \wedge \ n = N \wedge x = X \wedge p = x^{\frac{1}{2}k(k+1)} \wedge q = x^k \wedge 0 \leq k < n$$

Si $I \wedge B$ es verdadero en el estado en cuestión entonces también lo es $k < n$ y restando k a los dos lados de la desigualdad obtenemos $0 < n - k$. Luego la primera condición se cumple.

· Segunda, escribimos los predicados resultantes después de la sustitución de las instrucciones del cuerpo del bucle

$$Inv \wedge B \wedge n - k = T \Rightarrow n - (k + 1) < T$$

Suponiendo que el antecedente es verdadero, entonces $n - k = T$ y sustituyendo la expresión $n - k$ en el consecuente queda $T - 1 < T$, lo que obviamente, es cierto.

• *Simulación.* Realiza una simulación de la ejecución del algoritmo con el mismo ejemplo inicial para comprobar su buen funcionamiento.

Ejercicio 1.B. Codificación en C

Escribe un programa en C con nombre `suma_agujereada` que codifique el algoritmo anterior. Realiza algunas pruebas de ejecución del programa para diferentes valores de las variables de entrada x y n, y comprueba el resultado en la variable de salida s. Cuando x =2.0, ¿Cuál es el mayor valor inicial de la variable n que sigue ofreciendo un resultado correcto en la variable s?.

Ejercicio 2. Suma de divisores de un número natural

Deriva y codifica en C un algoritmo que satisfaga la siguiente especificación.

n, s: *entero*;
$\{Pre : n = N \ \wedge \ N \geq 1\}$
suma_divisores
$\{Post : s = (\sum_{\substack{1 \leq d \leq n \ \wedge \\ (n \ mod \ d) = 0}} d) \ \wedge \ n = N\}$

Ejercicio 2.A. Derivación del algoritmo

El algoritmo calcula en la variable s la suma de los divisores del valor inicial $N \geq 1$ de la variable n. Por ejemplo, si n comienza con el valor 6, la variable s debe contener al final la suma $s = \ 1 \ + \ 2 \ + \ 3 + \ 6$ y la variable n no cambia su valor, $n = 6$. En la expresión de la suma en la postcondición, el dominio de recorrido de la variable ligada d está determinado por dos condiciones $1 \leq d \leq n \ \wedge \ (n \ mod \ d) = 0$. En el ejemplo anterior $1 \leq d \leq 6$ y además, los valores de d que realmente se suman son divisores de 6, $(6 \ mod \ d) = 0$.

• *Propuesta de invariante.* Cuando en la postcondición el cálculo a realizar se expresa, como en este caso, mediante un cuantificador, el problema se resuelve en la mayor parte de los casos, construyendo un invariante mediante la sustitución de una constante por una

nueva variable. En el predicado $Post \equiv s = (\sum_{\substack{1 \leq d \leq n \ \wedge \\ (n \ mod \ d)=0}} d) \ \wedge \ n = N$ tenemos que 1 es una constante y que la variable n 'actúa como una constante' ya que no cambia su valor en el algoritmo (la postcondición exige $n = N$). Vamos a construir el invariante sustituyendo 1 por una nueva variable k de tipo entero. También debemos definir el rango de valores para la nueva variable k. Como k sustituye a 1 en el dominio de la suma debe cumplir $1 \leq k \leq n$.

$$Inv : s = (\sum_{\substack{k \leq d \leq n \ \wedge \\ (n \ mod \ d)=0}} d) \ \wedge \ 1 \leq k \leq n \ \wedge \ n = N$$

• *Cálculo de la condición de continuación.* Se debe cumplir $Inv \wedge \neg B \Rightarrow Post$. En este caso, cuando $k = 1$, se cumple que $Inv \wedge (k = 1) \Rightarrow Post$. Por lo tanto $\neg B : k = 1$ y $B : k \neq 1$. También podemos poner $B : 1 < k$ teniendo en cuenta que en el invariante se cumple $1 \leq k \leq n$.

• *Cálculo de las instrucciones de inicio.* El algoritmo tiene que terminar cuando $k = 1$. Como en el invariante siempre se cumple $1 \leq k \leq n$ podemos buscar las instrucciones más sencillas de inicio haciendo que $k = n$.

Calcula cuál es el valor inicial para la variable s.

```
k:= n;
s:= v_inicial;
```

Tienes que comprobar que se cumple $Pre \Rightarrow ((Inv)^{s}_{v\_inicial})^{k}_{n}$.

• *Cálculo de las instrucciones de avanzar.* La condición de terminación es $k = 1$. La variable k, comienza la ejecución del bucle con el valor n, por lo tanto, el valor de la variable k tiene que decrecer en cada paso del bucle hasta alcanzar el valor 1 para poder terminar la ejecución del algoritmo. Proponemos como instrucción de avanzar k:= k - 1.

• *Cálculo de las instrucciones de restablecer.* Se tiene que cumplir

$$\{Inv \wedge B\} \text{ 'restablecer' } \{(Inv)^{k}_{k-1}\}$$

· Primero, escribimos los predicados.

$$Inv \wedge B : s = (\sum_{\substack{k \leq d \leq n \ \wedge \\ (n \ mod \ d)=0}} d) \ \wedge \ 1 < k \leq n \ \wedge \ n = N$$

$$(Inv)^{k}_{k-1} : s = (\sum_{\substack{k-1 \leq d \leq n \ \wedge \\ (n \ mod \ d)=0}} d) \ \wedge \ 1 \leq k-1 \leq n \ \wedge \ n = N$$

· Segundo, comprobamos lo que se cumple en $(Inv)^{k}_{k-1}$ teniendo en cuenta $Inv \wedge B$.

$$1 < k \leq n \ \wedge \ n = N \Rightarrow 1 \leq k-1 \leq n \ \wedge \ n = N$$

· Tercero, relacionamos las expresiones de las variables que cambian su valor en $(Inv)_{k-1}^{k}$ teniendo en cuenta el valor que tenían antes en $Inv \wedge B$.

Para la variable s: $(\sum_{\substack{k-1\leq d\leq n \ \wedge \\ (n \ mod \ d)=0}} d) = (\sum_{\substack{k\leq d\leq n \ \wedge \\ (n \ mod \ d)=0}} d) + (k-1)$; si $(k-1)$ es divisor de n.
El nuevo valor de la variable s es el que tenía más (k-1) si $(k-1)$ es divisor de n.

s:= s + (k-1) (**si** $(k-1)$ es divisor de n)

En el otro caso, es decir, si $(k-1)$ no es divisor de n

$$(\sum_{\substack{k-1\leq d\leq n \ \wedge \\ (n \ mod \ d)=0}} d) = (\sum_{\substack{k\leq d\leq n \ \wedge \\ (n \ mod \ d)=0}} d)$$

Por tanto, el nuevo valor de la variable s es el mismo que tenía

s:= s; si $(k-1)$ (**si** $(k-1)$ **no** es divisor de n)

La instrucción de 'restablecer' se construye mediante una alternativa exclusiva que diferencia ambos casos.

• *Diseño del algoritmo.* El algoritmo resultante es el siguiente

algoritmo suma_divisores;
n, s: *entero*;
$\{Pre : n = N \ \wedge \ N \geq 1\}$
var
    k: *entero*;
fvar
    k:= n;
    s:= val_ini;
$\{Inv\}$ $\{cota...\}$
    mientras 1 < k **hacer**
        si n mod (k-1)= 0 → s:= s + (k-1)
          [] n mod (k-1)≠ 0 → **continuar**
        fsi;
        k:= k -1
    fmientras
$\{Post : s = (\sum_{\substack{1\leq d\leq n \ \wedge \\ (n \ mod \ d)=0}} d) \ \wedge \ n = N\}$
falgoritmo

• *Cálculo de la función de cota.* En este caso la función de cota es simplemente $cota = k$. Comprueba que se cumplen

a) $I \wedge B \Rightarrow k > 0$
b) $\{Inv \wedge B \wedge k = T\}$ 'restablecer'; $k := k - 1$ $\{k < T\}$

• *Simulación.* Realiza una simulación de la ejecución del algoritmo con el ejemplo dado al principio del apartado para comprobar su buen funcionamiento.

Ejercicio 2.B. Codificación en C

Escribe un programa en C con nombre `suma_divisores` que codifique el algoritmo anterior. Realiza algunas pruebas de ejecución del programa para diferentes valores de la variable de entrada n y comprueba el resultado en la variable de salida s.

Modifica el programa anterior para construir otro programa con nombre `numero_perfecto` que, dado un número natural $N \geq 1$ como entrada, escriba en pantalla un mensaje indicando si el número es perfecto o no lo es. Recuerda que un número natural es 'perfecto' si es igual a la suma de sus divisores menos él mismo, por ejemplo, 6 es perfecto ya que $6 = 1 + 2 + 3$.

Ejercicio 4. Coeficientes binomiales

El número de subconjuntos de m elementos que se pueden hacer en un conjunto de cardinal n, $\binom{n}{m}$, verifica la igualdad

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$$

Dada la especificación

n, m, q: *entero*;
$\{Pre : n = N \wedge\ m = M \wedge\ N \geq M \geq 0\}$
coeficientes_binomial
$\{Post : q = \binom{n}{m} \wedge\ n = N \wedge\ m = M\}$

Deriva un algoritmo basado en un único bucle que resuelva el problema. Se debe utilizar como invariante para el bucle el predicado siguiente

$$Inv : q = \frac{n!}{i! \cdot (n-i)!} \wedge\ 0 \leq i \leq m \wedge\ n = N \wedge\ m = M$$

Da una función de cota. Estudia su complejidad computacional.

Observa que el invariante Inv para el desarrollo del algoritmo anterior se basa en la sustitución de m por una nueva variable, en este caso i, con $0 \leq i \leq m$. Realiza otra propuesta de invariante, por ejemplo, sustituyendo en la postcondición n por una nueva variable k. Deriva el algoritmo correspondiente.

a) Escribe un programa en C de nombre `coef_binomial` tal que solicite dos números enteros positivos N y M tales que $N \geq M \geq 0$. En caso contrario el programa debe avisar al usuario de que ha relizado una introducción de datos incorrecta. A continuación el programa escribe por pantalla el valor del correspondiente coeficiente binomial $\binom{N}{M}$. Realiza las pruebas correspondientes para diferentes valores de N y M y comprueba los resultados obtenidos.

b) Escribe un nuevo programa en C de nombre `coef_binomiales` tal que solicite un número entero positivo N y escriba por pantalla todos los coeficientes binomiales $\binom{N}{0}$, $\binom{N}{1}$, hasta $\binom{N}{N}$. Aunque no hayas estudiado las funciones en C, intenta construir la solución utilizando una función que codifique el programa anterior `coef_binomial`. Piensa en las ventajas que tiene su uso en el programa solicitado.

Sesión 3

Derivación de algoritmos II. Elección de una conjunción

El objeto de esta práctica es trabajar la derivación formal de algoritmos a partir de nuevos métodos de construcción de invariantes y seguir ejercitando la codificación en C de los algorítmos obtenidos.

3.1. Actividades a realizar

En esta sesión, vamos a derivar varios algoritmos utilizando invariantes construidos mendiante la técnica de elección de una conjunción. También se aprenderá a utilizar pasos intermedios para resolver un problema: En algunas ocasiones, para resolver un problema definido sobre unas variables con las cuales se expresan la precondición y postcondición del algoritmo, hay que resolver antes un paso intermedio para obtener la solución final. En los dos ejercicios propuestos hay que seguir esta estrategia. Finalmente, codificaremos en C los algoritmos obtenidos.

Ejercicio 1. Inversa de la función de Cantor

El matemático alemán George Cantor (San Petersburgo 1845 - Halle 1918) llamó numerables a aquellos conjuntos cuyos elementos pueden ser puestos en correspondencia, uno a uno, con los números del conjunto de enteros positivos, lo que equivale a poderlos contar. En el año 1874 mostró que el conjunto de los números racionales es numerable. Para ello diseñó un algoritmo que a cada número racional le hace corresponder un número entero. Este proceso es reversible, por lo que así quedó demostrado de forma constructiva que los números racionales pueden ser emparejados biunívocamente con los números enteros. El proceso que diseño Cantor se puede simplificar para demostrar que el conjunto $\mathcal{N}^2$ está en correspondencia biyectiva con el conjunto $\mathcal{N}$. La biyección entre ambos conjuntos se expresa del modo siguiente

$$\begin{array}{rccl} J\colon & \mathcal{N}\times\mathcal{N} & \longrightarrow & \mathcal{N} \\ & (m,n) & \longmapsto & J(m,n) := \frac{1}{2}\cdot(m+n)\cdot(m+n+1)+m \end{array}$$

$$\begin{array}{rccl} J^{-1}\colon & \mathcal{N} & \longrightarrow & \mathcal{N}\times\mathcal{N} \\ & r & \longmapsto & J^{-1}(r) := (K(r),L(r)) \end{array}$$

donde K y L son a su vez funciones que tienen por definición

$$\begin{array}{rcl} K(r) & := & r - \frac{1}{2} \cdot s \cdot (s+1) \\ L(r) & := & s - K(r) \end{array}$$

En las definiciones anteriores, s es un número natural que depende de r del modo que expresan las desigualdades

$$\frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2)$$

Dada la especificación

r, m, n: *entero*;
$\{Pre : r = R \ \wedge \ R \geq 0\}$
    inversa_de_Cantor
$\{Post : m = K(r) \ \wedge \ n = L(r) \ \wedge \ r = R\}$

deriva un algoritmo que la cumpla y codifica el algoritmo en C.

Ejercicio 1.A. Derivación del algoritmo

Al analizar con un ejemplo el problema, nos damos cuenta de que antes de calcular $K(r)$ y $L(r)$ para un valor de r dado es necesario calcular el valor de s que satisface la desigualdad $\frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2)$. Podemos entonces plantear el siguiente problema donde el cálculo del valor para s se realiza antes que los cálculos para las variables m y n respectivamente.

algoritmo inversa_de_Cantor;
r, m, n: *entero*;
$\{Pre : r = R \ \wedge \ R \geq 0\}$
var s: *entero*; **fvar**
    ...
$\{Q : \frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2) \ \wedge \ r = R\}$
    ...
$\{Post : m = K(r) \ \wedge \ n = L(r) \ \wedge \ r = R\}$
falgoritmo

Una vez resuelto el subproblema con precondición Pre y postcondición Q, para la variable auxiliar s, resolver el subproblema con precondición Q y postcondición $Post$ es trivial, teniendo en cuenta las definiciones de $K(r)$ y $L(r)$ dadas en el enunciado del ejercicio. Sólo se necesitan dos instrucciones de asignación que debes calcular

algoritmo inversa_de_Cantor;
r, m, n: *entero*;
$\{Pre : r = R \ \wedge \ R \geq 0\}$
var s: *entero*; **fvar**
    ...
$\{Q : \frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2) \ \wedge \ r = R\}$
    m:=.....;
    n:=.....;
$\{Post : m = K(r) \ \wedge \ n = L(r) \ \wedge \ r = R\}$
falgoritmo

Para resolver el primer subproblema tenemos que plantear un invariante. El siguiente desarrollo corresponde a dicho subproblema.

• *Propuesta de invariante.* Si tomas, por ejemplo, $r = 4$ sabes que el valor de s que satisface $\frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2)$ es $s = 2$, ya que $3 \leq 4 < 6$.

En el predicado $Q : \frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2) \ \wedge \ r = R$ puedes elegir $r < \frac{1}{2} \cdot (s+1) \cdot (s+2)$ como condición de terminación $\neg B$ y el resto como predicado invariante Inv

$$Inv : \tfrac{1}{2} \cdot s \cdot (s+1) \leq r \ \wedge \ r = R$$
$$\neg B : r < \tfrac{1}{2} \cdot (s+1) \cdot (s+2)$$

De esta manera se cumple directamente $Inv \wedge \neg B \Rightarrow Q$ ya que $[[Inv \wedge \neg B \equiv Q]]$. Esto es así, siempre que se utiliza el método de construcción de invariantes que denominamos de elección de una conjunción.

• *Cálculo de la condición de continuación.* La condición de continuación tiene que programarse correctamente. Así, como r y s son variables de tipo entero, no podemos utilizar la expresión $\frac{1}{2}$ ya que esta expresión es de tipo real. Por tanto, la condición de continuación es $B : r \geq ((s+1) * (s+2)$ **div** $2)$.

Observa que no hay error, ya que el producto $(s+1) \cdot (s+2)$ siempre es divisible por dos y por lo tanto, $\frac{1}{2} \cdot (s+1) \cdot (s+2) = (s+1) \cdot (s+2)$ **div** 2.

• *Cálculo de las instrucciones de inicio.* En la precondición $r = R \wedge R \geq 0$ y el invariante es $\frac{1}{2} \cdot s \cdot (s+1) \leq r \ \wedge \ r = R$. El valor más sencillo para la variable s que hace que inicialmente se cumpla el invariante es simplemente $s = 0$. Por lo tanto, la instrucción de inicio es $s := 0$.

Comprueba que se cumple $Pre \Rightarrow (Inv)^s_0$.

• *Cálculo de las instrucciones de avanzar.* La condición de continuación es $r \geq \frac{1}{2} \cdot (s+1) \cdot (s+2)$, el valor de r no cambia $r = R$. Para que se cumpla la condición de salida del bucle $r < \frac{1}{2} \cdot (s+1) \cdot (s+2)$, la variable s tiene que aumentar su valor. Podemos proponer la instrucción $s := s + 1$.

• *Cálculo de las instrucciones de restablecer.* Se tiene que cumplir

$$\{Inv \wedge B\} \text{ 'restablecer' } \{(Inv)^s_{s+1}\}$$

No es necesario ninguna instrucción de restablecer porque se cumple $Inv \wedge B \Rightarrow (Inv)^s_{s+1}$. Comprueba este hecho.

• *Diseño del algoritmo.*

algoritmo inversa_de_Cantor;
r, m, n: *entero*;
$\{Pre : r = R \ \wedge \ R \geq 0\}$
var s: *entero*; **fvar**
    s:=0;
$\{Inv\}$ $\{cota...\}$
    mientras r $\geq$ ((s+1)*(s+2) **div** 2) **hacer**
        s:= s + 1
    fmientras;
$\{Q : \frac{1}{2} \cdot s \cdot (s+1) \leq r < \frac{1}{2} \cdot (s+1) \cdot (s+2) \ \wedge \ r = R\}$
    m:=.....;
    n:=.....
$\{Post : m = K(r) \ \wedge \ n = L(r) \ \wedge \ r = R\}$
falgoritmo

- *Cálculo de la función de cota.* Propón una función de cota que cumpla

    a) $Inv \wedge B \Rightarrow cota > 0$
    b) $\{Inv \wedge B \wedge cota = T\}$ 'restablecer'; $s := s + 1$ $\{cota < T\}$

Ejercicio 1.B. Codificación en C

Escribe un programa en C con nombre `inversa_de_Cantor` que codifique el algoritmo anterior. El programa lee desde el teclado un valor para la variable r. Después de escribir los resultados contenidos en las variables m y n, también debe escribir el resultado de la función $J(m, n)$ dada en el enunciado del ejercicio. Realiza distintas pruebas de ejecución para diferentes valores de r y comprueba que el resultado de $J(m, n)$ es exactamente r.

Ejercicio 2. Función dientes de sierra

Definida sobre $\mathcal{R}$ la función

$$f(x) = \frac{3}{5} \cdot (x - 10 \cdot k), \text{ con } k \in \mathcal{Z} \text{ tal que } x \in [10 \cdot k - 5, 10 \cdot k + 5),$$

y dada la especificación

x, y: *real*;
$\{Pre : x = X\}$
    dientes_de_sierra
$\{Post : y = f(x) \ \wedge \ x = X\}$

deriva un algoritmo que resuelva el problema planteado por ésta y codifica dicho algoritmo en C.

Ejercicio 2.A. Derivación del algoritmo

Antes de comenzar analizamos la función $f(x)$ para ver cómo se calcula para un valor concreto de x. Dado, x =8.5, $f(8.5) = \frac{3}{5} \cdot (8.5 - 10 \cdot k)$, pero k es un número entero tal que $x \in [10 \cdot k - 5, 10 \cdot k + 5)$, es decir, $10 \cdot k - 5 \leq x < 10 \cdot k + 5$. En el ejemplo anterior, para x =8.5, el valor de k es $k = 1$, ya que, $5 \leq 8.5 < 15$. Una vez calculado $k = 1$, finalmente

podemos calcular $f(8.5) = \frac{3}{5} \cdot (8.5 - 10 \cdot 1) = -\frac{9}{10}$.

La función $f(x)$ cuando

$k = -1$, es la recta $y = \frac{3}{5}\ (x + 10)$ para $-15 \leq x < -5$
$k = 0$, es la recta $y = \frac{3}{5}\ x$ para $-5 \leq x < 5$
$k = 1$, es la recta $y = \frac{3}{5}\ (x - 10)$ para $5 \leq x < 15$

Si representas graficamente la función verás que tiene un aspecto de dientes de sierra. La conclusión del ejemplo es que para calcular $f(x)$ para un x dado, primero tenemos que calcular el valor de k que cumple $10 \cdot k - 5 \leq x < 10 \cdot k + 5$, y segundo tenemos que aplicar directamente la fórmula $f(x) = \frac{3}{5} \cdot (x - 10 \cdot k)$. Como hemos hecho en el ejercicio 1, vamos a identificar los dos subproblemas que forman la secuencia de la solución.

algoritmo dientes_de_sierra;
x, y: *real*;
$\{Pre : x = X\}$
var k: *entero*; **fvar**
    ...
$\{Q : x = X \wedge 10 \cdot k - 5 \leq x < 10 \cdot k + 5\}$
    y:= (3/5) * (x-10 * k)
$\{Post : y = f(x)\ \wedge\ x = X\}$
falgoritmo

Observa que una vez resuelto el primer subproblema, la última instrucción de asignación da el resultado correcto, ya que se cumple que $Q \Rightarrow (Post)^{y}_{(3/5)*(x-10*k)}$. Comprueba este hecho antes de continuar.

En lo que sigue vamos a derivar el algoritmo para el primer subproblema con precondición Pre y postcondición Q.

• *Propuesta de invariante.* Dado $x = X$, tenemos que encontrar el valor de k que cumple $10 \cdot k - 5 \leq x < 10 \cdot k + 5$. Este tipo de problemas se pueden resolver con un bucle donde el invariante se construye mediante el 'método de elección de una conjunción'. Si escribimos $Q : x = X \wedge 10 \cdot k - 5 \leq x\ \wedge x < 10 \cdot k + 5$ podemos proponer

$Inv : x = X \wedge 10 \cdot k - 5 \leq x$
$\neg B : x < 10 \cdot k + 5$

De esta forma se cumple directamente $Inv \wedge \neg B \Rightarrow Q$.

• *Cálculo de la condición de continuación.* En este caso la conjunción no seleccionada $\neg B : x < 10 \cdot k + 5$ y por tanto $B : x \geq 10 \cdot k + 5$.

• *Cálculo de las instrucciones de inicio.* En el invariante $x = X$ tiene su valor inicial, pero hay que iniciar la variable k. En el invariante sólo sabemos que $10 \cdot k - 5 \leq x$. En la precondición sólo se indica $x = X$ y no tenemos más información. Así que podríamos poner cualquier valor inicial para k. Vamos a suponer que iniciamos $k = 0$ mediante la instrucción de asignación k:= 0. Entonces se tiene que cumplir $Pre \Rightarrow (Inv)^{k}_{0}$, es decir se tendría que cumplir $x = X \Rightarrow x = X \wedge 10 \cdot 0 - 5 \leq x$ (pero no es así). Observa que

$x = X$ no implica que $-5 < x$, ya que X puede ser cualquier valor. Esto mismo nos va a pasar sea cual fuere el valor inicial que elijamos para k. Para que tenga sentido la iniciación con $k = 0$ tengo que obligar a que se cumpla antes $-5 < x$, de forma que $Pre \wedge -5 < x \Rightarrow (Inv)_0^k$ sea una implicación verdadera. Lo que ésto nos está indicando es que la propuesta del invariante y la condición de continuación son válidas para los valores de x que cumplen $-5 < x$. No queda más remedio que construir una composición alternativa, de manera que la estructura general del algoritmo es la siguiente

algoritmo dientes_de_sierra;
x, y: *real*;
$\{Pre : x = X\}$
var k: *entero*; **fvar**
  si -5 ≤ x → k:= 0;
    $\{Inv\}$ {cota ...}
    mientras x ≥ 10 * k + 5 **hacer**
      'avanzar'
    fmientras
  [] -5 > x → .....
  fsi;
$\{Q : x = X \wedge 10 \cdot k - 5 \leq x < 10 \cdot k + 5\}$
  y:= (3/5) * (x-10 * k)
$\{Post : y = f(x) \ \wedge \ x = X\}$
falgoritmo

La composición alternativa es exclusiva y está bien definida.

• *Cálculo de las instrucciones de avanzar.* Para salir del bucle que estamos construyendo, dado que la condición de continuación es $x \geq 10 \cdot k + 5$, se tiene que cumplir lo contrario $x < 10 \cdot k + 5$. Como x no cambia su valor y la parte $10 \cdot k + 5$ es positiva cuando k comienza desde 0, entonces el valor de k debe crecer. Proponemos como instrucción de avanzar k:= k +1.

• *Cálculo de las instrucciones de restablecer.* No son necesarias instrucciones de restablecer ya que se cumple directamente que $Inv \wedge B \Rightarrow (Inv)_{k+1}^k$.

$$x = X \wedge 10 \cdot k - 5 \leq x \wedge x \geq 10 \cdot k + 5 \Rightarrow x = X \wedge 10 \cdot (k + 1) - 5 \leq x$$

• *Diseño del algoritmo.*

algoritmo dientes_de_sierra;
x, y: *real*;
$\{Pre : x = X\}$
var k: *entero*; **fvar**
  si -5 ≤ x → k:= 0;
    $\{Inv\}$ $\{cota = x - 10 \cdot k\}$
    mientras x ≥ 10 * k + 5 **hacer**
      k:= k+1
    fmientras
  [] -5 > x → .....
  fsi;

$\{Q : x = X \wedge 10 \cdot k - 5 \leq x < 10 \cdot k + 5\}$
    y:= (3/5) * (x-10 * k)
$\{Post : y = f(x) \ \wedge \ x = X\}$
falgoritmo

• *Cálculo de la función de cota.* Comprueba que $cota = x - 10 \cdot k$ es una función de cota correcta para este primer bucle.

a) $Inv \wedge B \Rightarrow cota > 0$
b) $\{Inv \wedge B \wedge cota = T\}\ k := k + 1\ \{cota < T\}$

Debes construir el bucle para la otra rama de la alternativa, cuando $x < -5$. Como ayuda, el invariante para esta parte es

$Inv' : x = X \wedge x < 10 \cdot k + 5$

Completa el algoritmo.

algoritmo dientes_de_sierra;
x, y: *real*;
$\{Pre : x = X\}$
var k: *entero*; **fvar**
    si -5 ≤ x → k:= 0;
        $\{Inv\}$ $\{cota = x - 10 \cdot k\}$
        mientras x ≥ 10 * k + 5 **hacer**
            k:= k+1
        fmientras
    [] -5 > x → k:= .....;
        $\{Inv'\}$ $\{cota...\}$
        mientras 'condición de continuación' **hacer**
            'avanzar'
        fmientras
    fsi;
$\{Q : x = X \wedge 10 \cdot k - 5 \leq x < 10 \cdot k + 5\}$
    y:= (3/5) * (x-10 * k)
$\{Post : y = f(x) \ \wedge \ x = X\}$
falgoritmo

Ejercicio 2.B. Codificación en C

Escribe un programa en C con nombre `dientes_de_sierra` que codifique el algoritmo anterior. Realiza distintas pruebas de ejecución para diferentes valores de x, comprueba que los resultados sean correctos.

Sesión 4

Acciones y funciones

El objeto de esta práctica es mostrar las posibilidades que ofrece C para codificar algoritmos que puedan ser reutilizados. La parametrización de algoritmos y su encapsulación mediante acciones y funciones permite resolver problemas complejos mediante el uso apropiado del diseño descendente. Explicaremos en esta sesión como codificar acciones y funciones en el lenguaje C.

4.1. Acciones y funciones

A medida que avanzamos en la construcción de algoritmos y de los programas que codifican dichos algoritmos, se hace patente la necesidad de encontrar formas de reducir la complejidad de ciertos problemas. Una técnica es construir la solución como resultado de refinamientos sucesivos.

En cada paso de refinamiento, el diseñador divide el problema a resolver en un número de subproblemas de menor complejidad. Mediante este proceso es posible llegar a problemas lo suficientemente sencillos para ser resueltos directamente. Entonces se combinan las soluciones de los subproblemas para resolver el problema que los comprende y así, se van obteniendo soluciones hasta llegar al nivel de mayor complejidad.

Los algoritmos que se introducen en cada nivel de refinamiento pueden aparecer codificados en forma de pequeños subprogramas denominados **acciones** y **funciones**. La diferencia entre estas dos formas es que una acción puede devolver múltiples valores o ninguno, mientras que la función devuelve al menos un valor.

Las acciones y funciones deben estar declaradas antes de utilizarse en el cuerpo principal del algoritmo o en otras acciones o funciones.

4.1.1. Acciones

Una acción se puede entender como un algoritmo que está diseñado de tal forma que puede ser utilizado por cualquier otro algoritmo u otra acción. Por ejemplo, supongamos que tenemos el algoritmo de intercambio de dos variables.

algoritmo intercambio;
a, b : *entero*;
$\{P : a = A \wedge b = B\}$
var aux: *entero* **fvar**
    aux:= a;

a:= b;
b:= aux
$\{Q : a = B \wedge b = A\}$
falgoritmo

Si observas detenidamente el algoritmo *ordena3b* (Ejercicio 2, Sesión 1), el intercambio se ha empleado tres veces con diferentes variables y cada vez se ha tenido que modificar para ajustarse a las variables que se usaban en cada alternativa

intercambiar p por s
intercambiar s por t
intercambiar p por t

Incluso en este caso sencillo podemos utilizar el concepto de acción. Primero observamos la especificación del algoritmo *intercambio*: la variable a y la variable b se utilizan tanto para contener un dato inicial (son de entrada) y para recoger el resultado final (son de salida). Por lo tanto van a ser considerados como parámetros de entrada/salida (**ent/sal**). Una vez realizada esta identificación procedemos a presentar la declaración de la acción. La declaración incluye el nombre de la acción, los parámetros formales (indicando si son de entrada **ent**, de salida **sal**, o de entrada/salida **ent/sal**) y su tipo. Como en todo algoritmo incluimos la precondición y la postcondición.

acción intercambio(**ent/sal** a, b : *entero*);
$\{P : a = A \wedge b = B\}$
$\{Q : a = B \wedge b = A\}$
facción

La declaración de la acción es suficiente para saber qué hace y cómo otros algoritmos la pueden utilizar. Este hecho es independiente de cómo resuelvas el problema *intercambio* especificado por los predicados P y Q. Puedes utilizar cualquiera que satisfaga la especificación de la acción. Escribimos de nuevo el algoritmo *ordena3b* pero usando la acción *intercambio* para que veas el efecto.

algoritmo ordena3b;
acción intercambio(**ent/sal** a, b : *entero*);
$\{P : a = A \wedge b = B\}$
var aux: *entero* **fvar**
aux:= a;
a:= b;
b:= aux
$\{Q : a = B \wedge b = A\}$
facción
//
x, y, z: *entero*
p, s, t: *entero*
$\{Pre : x = X \wedge y = Y \wedge z = Z\}$
p:= x; s:= y; t:= z;
si p > s $\longrightarrow$ intercambio(p,s)
[] p $\leq$ s $\longrightarrow$ **continuar**
fsi;

si p > t ⟶ intercambio(p,t)
[] p ≤ t ⟶ **continuar**
fsi;
si s > t ⟶ intercambio(s,t)
[] s ≤ t ⟶ **continuar**
fsi
$\{Post : (p, s, t) \in \text{Perm}(X, Y, Z) \wedge p \leq s \leq t\}$
falgoritmo

Cuando en el algoritmo principal se invoca la ejecución de, por ejemplo,

intercambio(p,s)

se produce:

(i) Una asociación de los argumentos actuales con los parámetros formales en la invocación de acción. En este caso, las variables p y s se asocian con los parámetros formales a y b respectivamente. Esto es debido al orden en que aparecen los parámetros formales. Por otra parte, para que no haya errores, el número de argumentos tiene que coincidir siempre con el número de parámetros formales, y el tipo de cada argumento que se asocia con un parámetro debe ser exáctamente el mismo.

(ii) La comunicación entre argumentos y parámetros al inicio de la invocación de la acción. El valor de cada argumento es asignado a su correspondiente parámetro en el momento de la llamada a la acción. En el ejemplo, el valor de p se pasa a a y el de s se pasa a b, ya que a y b son parámetros de entrada (también de salida). A partir de ahí, se ejecutan las instrucciones que forman parte del cuerpo de la acción.

(iii) La comunicación entre los parámetros y los argumentos al final de la ejecución de las instrucciones de la acción. El valor final de cada parámetro de salida se pasa al argumento correspondiente. En el ejemplo, el valor de a se pasa a p y el de b se pasa a s.

Discusión. Un lenguaje de programación debe determinar de forma precisa la manera en que los cambios efectuados sobre los valores de los parámetros formales en el cuerpo de la acción invocada afectan a los argumentos (también algunas veces denominados parámetros reales) de la acción principal:
- *Parámetros de entrada*. Si un parámetro formal de una acción se declara de entrada, el argumento que se asocia al parámetro en la invocación no verá alterado su valor al terminar la ejecución de la acción.
- *Parámetros de salida*. Si el parámetro se declara de salida, el argumento asociado al parámetro se verá afectado porque recoge el valor final del parámetro al terminar la ejecución de la acción.
- *Parámetros de entrada/salida*. Si el parámetro es de entrada/salida, el argumento se verá afectado por recoger el resultado final pero, además, su valor inicial será utilizado como parte del cálculo en la acción.

De lo anterior se deduce que un argumento para un parámetro de salida o de entrada/salida sólo puede ser una variable del mismo tipo que el parámetro. En cambio, el argumento para un parámetro de entrada puede ser una expresión del mismo tipo que el parámetro. Algunos lenguajes de programación consideran un error la posibilidad que en el código de la acción se modifique el valor de un parámetro de entrada. Por otro lado, se considera un buen estilo de programación no modificar el valor de los parámetros (sólo)

de entrada, aunque ello no tenga ningún efecto sobre los argumentos asociados, fuera de la acción. El lenguaje algorítmico que utilizamos está pensado para facilitar la escritura de algoritmos y estudiar su corrección, no para ser ejecutado por un ordenador. Por este motivo, en ocasiones tenemos que limitar algunos usos que otros lenguajes permiten 'a la ligera'.

Como **norma de esta asignatura**, en la escritura del código de las acciones y funciones con el lenguaje algorítmico (y por tanto, en su codificación en C) **no permitimos** la modificación de los parámentros de entrada (basta con evitar instrucciones de asignación que afecten a estos parámetros de entrada).

El algoritmo *ordena3b*, que acabamos de ver, también puede escribirse en forma de acción. En este caso las variables x, y y z actuarán como parámetros de entrada: se usan sus valores iniciales en los cálculos y no se ven alteradas por la ejecución de las instrucciones del algoritmo. Las variables p, s, y t actuarán como parámetros de salida, ya que contendrán el resultado según la especificación. Teniendo en cuenta la discusión anterior, la declaración de esta acción es

acción ordena3b(**ent** x, y, z : *entero*; **sal** p, s, t : *entero*);
$\{P' : x = X \wedge y = Y \wedge z = Z\}$
$\{Q' : x = X \wedge y = Y \wedge z = Z \wedge (p, s, t) \in \text{Perm}(X, Y, Z) \wedge p \leq s \leq t\}$
facción

Observa que hemos repetido $x = X \wedge y = Y \wedge z = Z$ en la precondición P' y en la postcondición Q' para que quede claro que actúan sólo como parámetros de entrada. En este curso no haremos uso de reglas de inferecia para acciones que permitan demostrar la corrección de los algoritmos que las usan, pero sí haremos demostraciones de corrección de algoritmos que usan funciones por lo que la siguiente sección es importante a ese respecto.

4.1.2. Funciones

Una función es una acción que tiene uno o varios parámetros de entrada y un único parámetro de salida[1] [2]. La funciones tienen una forma muy relacionada con las funciones que estamos habituados a utilizar en matemáticas y son un mecanismo adecuado y rápido para incrementar el repertorio de instrucciones de un lenguaje de programación. Como ejemplo, damos la declaración de una función para calcular el máximo común divisor.

función max_com_div(x, y : *entero*) **dev** mcd: *entero*;
$\{P : x = X \wedge y = Y \wedge X > 0 \wedge Y > 0\}$
$\{Q : x = X \wedge y = Y \wedge mcd = MCD(X, Y)\}$
    dev mcd
ffunción

Por la propia forma en la que está expresada la declaración de la función, se asume que x e y son parámetros de entrada y que el resultado se devuelve en el parámetro de

[1] En el tema de funciones recursivas permitiremos que las funciones tengan más de un parámentro de salida.

[2] El nombre del parámetro de salida debe ser distinto a cualquiera de los de entrada. En otro caso el parámetro sería de entrada/salida y deberíamos utilizar una acción, no una función.

salida *mcd*. Como en la declaración queda clararamente reflejada la clasificación de los parámetros, no escribimos **ent**, ni **sal**. En el lenguaje algorítmico, las funciones **sólo se utilizan** en combinación con instrucciones de asignación. Por ejemplo,

r:= max_com_div(25, 10*z);

Como **norma de esta asignatura**, en el lenguaje algorítmico **no está permitido** combinar funciones con otras operaciones o funciones en la misma asignación con objeto de facilitar el estudio de la corrección de los algoritmos que las usan. Por ejemplo, la siguiente instrucción de asignación no está permitida en esta asignatura

r:= max_com_div(25, 10*z) + r;

En el siguiente Ejercicio 1 explicamos otras cuestiones importantes relacionadas con las funciones y su corrección.

Ejercicio 1. Serie exponencial

Deriva un algoritmo correcto para el cálculo de una serie exponencial siguiendo la siguiente especificación

n, s: *entero*
$\{Pre : n = N \ \wedge \ N \geq 0\}$
    serie_exponencial
$\{Post : s = \sum_{0<i\leq n} i^i \ \wedge \ n = N\}$

En la resolución del problema planteado, vamos a ver que es necesario resolver también la siguiente función

función potencia (a, b: *entero*) **dev** c: *entero*
$\{P : a = A \ \wedge \ A > 0 \ \wedge \ b = B \ \wedge \ B \geq 0\}$
    ...
$\{Q : c = a^b \ \wedge a = A \ \wedge \ b = B\}$
    dev c
ffunción

Notas sobre funciones

(1) Observa que en la especificación reforzamos la idea de que los parámetros (formales) a y b son parámetros de entrada poniendo tanto en la precondición P como en la postcondición Q las expresiones $a = A$ y $b = B$. Por tanto, cuando escribas el código de la función no utilices instrucciones de asignación del tipo $a :=$ ... ó $b :=$ .... En el caso de que el algoritmo empleado necesite ir modificando el valor de alguna de esas variables, cópialo antes en una variable auxiliar para realizar la instrucción de asignación. De esta forma, si haces al principio, por ejemplo, $aux\_a := a$, ya tienes libertad para modificar aux_a respetando $a = A$ a lo largo de todo el código.

(2) A la hora de escribir la especificación de la función, los parámetros formales de entrada se representan como variables, en el caso anterior 'a, b: *entero*'. No obstante, dadas

las características de los parámetros de entrada, puedes emplear argumentos que sean expresiones del mismo tipo que los parámetros. Por ejemplo, si vas a utilizar la función potencia() en un algoritmo que tiene una declaración de variables p, j, k, s : *entero* y, en un determinado punto del algoritmo, necesitas calcular en la variable p el valor de la potencia $(j+2)^{(3k)}$ basta con que realices la asignación: $p := potencia(j+2, 3*k)$. El parámetro formal a recoge el valor de la expresión $(j+2)$ y el parámetro b el valor de la expresión $3*k$. Ambas expresiones constituyen los argumentos para la función y son evaluadas antes de la llamada a la función. El parámetro formal c de la función *potencia* sirve para que el valor que adquiere después de ejecutarse la función sea copiado en la variable p sobre la que se realiza la asignación. De esta manera, como en la postcondición de la función $c = a^b$ y tanto a como b no cambian su valor inicial, sabes que cuando la función termina la asignación dará como resultado $p = (j+2)^{(3k)}$.

(3) Siguiendo el ejemplo anterior, supongamos que tenemos el siguiente predicado antes de la llamada a la función en la descripción del algoritmo

$\{j = J \wedge k = K \wedge 0 \leq k < j \wedge p = j^k \wedge s = j \cdot k\}$
p:= potencia(j+2, 3*k)

Después de la ejecución de la función, podemos deducir cuál será el predicado resultante

$\{j = J \wedge k = K \wedge 0 \leq k < j \wedge p = j^k \wedge s = j \cdot k\}$
p:= potencia(j+2, 3*k)
$\{j = J \wedge k = K \wedge 0 \leq k < j \wedge p = (j+2)^{(3k)} \wedge s = j \cdot k\}$

Esta deducción sólo es posible porque estamos asumiendo que: (i) las variables que forman parte de las expresiones, que son argumentos de entrada, nunca cambiarán su valor por efecto de la función; y (ii) la función no tiene efectos colaterales sobre otras variables del algoritmo u acción principal que la invoca. En el ejemplo, la variable s no se ha utilizado (ní se puede utilizar) en el código de la función, modificando su valor de forma oculta, aunque el lenguaje de programación que se utiliza lo hubiera permitido.

En programación estructurada no se permiten efectos colaterales en las funciones ni en las acciones, por eso ponemos la declaración de las funciones y acciones antes de las variables que se utilizan como argumentos en dichas funciones y acciones. Cualquier compilador que se precie debe 'avisar' de que hay una 'referencia hacía atrás': una variable se ha utilizado antes de ser declarada.

(4) Teniendo en cuenta lo discutido anteriormente y con objeto de facilitar las demostraciones y comprobaciones con los algoritmos que usan funciones escribiremos la especificación de las funciones de la forma que se muestra en el ejemplo.

función potencia (a, b: *entero*) **dev** c: *entero*
$\{P \equiv a > 0 \ \wedge \ b \geq 0\}$
$\{Q \equiv c = a^b\}$
    dev c
ffunción

Asumiremos y garantizaremos a partir de ahora que:

- Los parámetros de entrada no serán modificados en el código de la función.
- En el código de la función no se utilizarán variables declaradas[3] fuera del ámbito de la función con objeto de evitar efectos colaterales.
- Las variables que forman parte de las expresiones, que son los argumentos asociados a los parámetros de entrada, nunca cambiarán su valor por efecto de la ejecución de la función.

(5) Sobre la verificación de los algoritmos que usan funciones: regla de inferencia de funciones. La declaración de la función contiene toda la información que necesitas, si se ha descrito correctamente la precondición y la postcondición de la misma, en nuestro caso los predicados P y Q respectivamente. El algoritmo que utiliza dicha función puede requerir, para ser correcto, que parte de su código cumpla las condiciones dadas por otros predicados. Analicemos el siguiente ejemplo.

```
algoritmo ejemplo;
    función potencia (a, b: entero) dev c: entero
    {P : a > 0 ∧ b ≥ 0}
    {Q : c = a^b}
        dev c
    ffunción
p, j, k, s: entero
{Pre}
    .....
    {R}
    p:= potencia(Exp1, Exp2)
                (* Exp1, Exp2, expresiones que forman los argumentos *)
    {S}
    .....
{Post}
falgoritmo
```

Debes observar que los predicados R y S expresan relaciones sobre las variables del algoritmo. Para que comenzando en un estado que cumpla R, después de ejecutar la instrucción de asignación que llama a la función, se llegue a un estado que cumpla S se tiene que verificar:

(i) $R \Rightarrow P[a \leftarrow Exp1, b \leftarrow Exp2]$, es decir, se tienen que cumplir las condiciones de la precondición de la función sobre los argumentos de entrada. La sustitución múltiple anterior se hace de manera simultánea.

(ii) Se tiene que cumplir que la implementación de la función es correcta: Cumple con su especificación y termina!

(iii) Sea R' la parte de R que no incluye condiciones sobre la variable p entonces se tiene que cumplir $R' \wedge Q[c \leftarrow p, a \leftarrow Exp1, b \leftarrow Exp2] \Rightarrow S$, es decir lo que no cambia en R por efecto de la función más los resultados obtenidos por la función forman parte del estado descrito por S.

[3]En general, se recomienda que los identificadores de las variables declaradas localmente en el cuerpo de la función sean diferentes a los identificadores de las variables declaradas fuera del ámbito de la función.

Ejercicio 1.A. Derivación del algoritmo para la serie exponencial

Recuperamos la especificación del problema del cálculo de una serie exponencial

n, s: *entero*
$\{Pre : n = N \ \wedge \ N \geq 0\}$
serie_exponencial
$\{Post : s = \sum_{0<i\leq n} i^i \ \wedge \ n = N\}$

Se trata de derivar un algoritmo correcto para este problema utilizando la función *potencia* previamente especificada.

• *Propuesta de invariante.* El método a emplear para obtener el invariante del algoritmo principal es el de sustitución de la variable n (que actúa como una constante ya que su valor no cambia en el algoritmo) por una nueva variable j (de tipo entero). Así el invariante sería

$$Inv : s = \sum_{0<i\leq j} i^i \ \wedge \ 0 \leq j \leq n \ \wedge \ n = N.$$

• *Cálculo de las instrucciones de inicio.* Puedes observar que el primer valor para j es 0, ya que j sustituye a n y 0 es un (primer) valor posible para dicha variable. Entonces, el dominio del sumatorio es vacío y por lo tanto habrá que iniciar s a 0.

• *Cálculo de la condición de continuación.* La condición de continuación es $B : j \neq n$, ya que $Inv \wedge \neg B \Rightarrow Post$.

• *Cálculo de las instrucciones de avanzar.* Si j comienza el bucle con el valor 0 y el bucle debe terminar cuando $j = n$, siendo n en general un número entero positivo, la decisión para avanzar es $j := j + 1$.

• *Cálculo de las instrucciones de restablecer.* En el cálculo del bucle, al avanzar con la instrucción $j := j + 1$, será necesario restablecer el valor de la variable s con el valor de la expresión $(j + 1)^{(j+1)}$. Como esta expresión no se puede calcular directamente, podemos reforzar el invariante introduciendo una nueva variable entera z que sea igual a dicha expresión (en este caso no podemos utilizar la expresión previa j^j). De manera que, una nueva propuesta de invariante puede ser

$$Inv' : s = \sum_{0<i\leq j} i^i \ \wedge \ 0 \leq j \leq n \ \wedge \ z = (j+1)^{(j+1)} \ \wedge \ n = N.$$

En la revisión del programa con el nuevo invariante Inv', al avanzar con $j := j + 1$, deberás resolver otro problema que es el cálculo de $z = (j + 2)^{(j+2)}$. El problema que encontramos aquí, es que no podemos restablecer el valor de z con alguna expresión sencilla formada por su valor previo y el valor de j.

La conclusión es que el reforzamiento que hemos introducido es en valde, ya que debemos calcular explícitamente con un algoritmo $(j+1)^{(j+1)}$ en el caso de que utilicemos como invariante Inv, el primero que hemos propuesto, o bien, $(j + 2)^{(j+2)}$ si seguimos con el invariante Inv' que tiene dicho reforzamiento. Para no tener que escribir el código del algoritmo que calcula cualquiera de esas potencias dentro del código del algoritmo

principal, podemos emplear la función *potencia* que ya tenemos especificada. También es importante darse cuenta de que en este caso no es obligatorio reforzar el invariante Inv, aunque sí vamos a necesitar una variable auxiliar para obtener el valor de $(j+1)^{(j+1)}$ con objeto de restablecer el valor de la variable s.

Completa el algoritmo que está esbozado a continuación y realiza las comprobaciones que se indican. En el algoritmo 'val_ini_j' y 'val_ini_s' son las expresiones de los valores iniciales para las variables j y s respectivamente. En la función potencia, 'exp_a' y 'exp_b' son las expresiones para los argumentos de entrada de la función (en la discusión anterior están las soluciones).

algoritmo serie_exponencial;
  función potencia (a, b: entero) **dev** c: entero
  $\{P : a > 0 \ \wedge \ b \geq 0\}$
    ...
  $\{Q : c = a^b\}$
    dev c
  ffunción
n, s: *entero*
$\{Pre : n = N \ \wedge \ N \geq 0\}$
var j, aux: *entero* **fvar**
  j:= 'val_ini_j';
  s:= 'val_ini_s';
$\{Inv\}$ $\{cota = n - j\}$
  mientras j $\neq$ n **hacer**
    aux:= potencia('exp_a', 'exp_b');
    s:= s + aux;
    j:= j + 1
  fmientras
$\{Post : s = \sum_{0<i\leq n} i^i \ \wedge \ n = N\}$
falgoritmo

Comprueba:

- $Pre \Rightarrow ((Inv)^s_{val\_ini\_s})^j_{val\_ini\_j}$
- $Inv \wedge \neg B \Rightarrow Post$
- $Inv \wedge B \Rightarrow P[a \leftarrow exp\_a, b \leftarrow exp\_b]$
- $Inv \wedge B \wedge Q[c \leftarrow aux, a \leftarrow exp\_a, b \leftarrow exp\_b] \Rightarrow ((Inv)^j_{j+1})^s_{s+aux}$
- $Inv \wedge B \Rightarrow cota > 0$
- $Inv \wedge B \wedge cota = T \Rightarrow (cota < T)^j_{j+1}$

El algoritmo de la serie exponencial puede transformarse fácilmente en una función para poder utilizarse en otro algoritmo. Completa el ejercicio construyendo la función *serie_exponencial* a partir del algoritmo construido. Identifica en la especificación las variables que actuarán como parámetros de entrada y salida, y completa la declaración

función serie_exponencial(.....) **dev** ....: .....
$\{P' : .....\}$
$\{Q' : .....\}$
    dev .....
ffunción

4.2. Codificación de acciones y funciones en C

Hasta el momento hemos analizado las acciones y funciones desde el punto de vista del lenguaje algorítmico, en esta sección vamos a estudiar la definición y el uso de funciones en C.

En el lenguaje C no hay palabras reservadas para indicar la declaración de acciones o funciones como en otros lenguajes como Python o Pascal. Por otra parte, es común decir que C sólo admite la declaración de funciones. El aspecto de la declaración de una función en C es bastante simple

```
tipo identificador(parametros)
{
    cuerpo de la funcion
}
```

El `identificador` es el nombre de la función y debe ser un identificador válido como los que se emplean para dar nombres a las variables. El tipo de la función indica de qué tipo de datos es el valor que devuelve la función. Entre los paréntesis aparece la declaración de los parámetros separados por comas. El cuerpo de la función debe ir encerrado entre llaves. Aquí encontraremos la declaración de variables locales a la función y las instrucciones a ejecutar. Veamos un ejemplo:

funcion sumatorio(a, b : *entero*) **dev** s: *entero*;
$\{P : 0 \leq a \leq b\}$
var i: entero **fvar**
    i:= a;
    s:= a;
    $\{Inv : s = \sum k : a \leq k \leq i : k \wedge a \leq i \leq b\}$
    mientras $i \neq b$ **hacer**
        s:= s + (i+1);
        i:= i + 1
    fmientras;
    dev s
$\{Q : s = \sum_{k=a}^{b} k\}$
ffuncion

La codificación en C de la función anterior es la siguiente:

```
int sumatorio(int a, int b)
{
    int i, s;
    i = a;
    s = a;
    while (i != b)
    {
        s = s + (i + 1);
        i = i + 1;
    }
    return s;
}
```

La última sentencia `return s;` es la sentencia que devuelve el valor contenida en la expresión (en este caso la variable `s`). La ejecución de esta sentencia finaliza la ejecución de la función.

Las variables que se declaran en el cuerpo de la función se crean cuando se invoca la ejecución de la función desde otra función (acuerdate que la primera función que se ejecuta es la función `main(void)`) y desaparecen cuando termina la ejecución de la función.

Una función que no tiene parámetros se declara de la misma forma pero utilizando el término `void`. Es posible que una función no devuelva nada y por tanto, el tipo de retorno de la función también es `void`.

En el lenguaje algorítmico los parámetros se clasifican como de entrada, salida, entrada/salida. En la función en C se hace implicita esta clasificación según cómo sea la declaración de parámetros.

En el ejemplo anterior los parámetros `a` y `b` son de entrada y C te **asegura** de que si modificas su valor en el cuerpo de la función, el argumento (parámetro real) que usa la función no se ve alterado después de terminar la ejecución de la función. Aunque nosotros no permitimos modificaciones en el cuerpo de la función de los parámetros de entrada en el lenguaje algorítmico para asegurar la corrección, el lenguaje C sí te lo permite.

La cuestión es, cómo se indica en C que los parámetros son de salida o de entrada/salida. Básicamente hay que indicar que el parámetro es una dirección de memoria que apunta a un determinado tipo de datos. Recordemos que si declaramos una variable `int x`, `&x` nos devuelve su dirección. Esta dirección, `&x`, apunta a un número del tipo `int`. C dispone de una operación que te permite acceder al contenido de una dirección, esta operación es el asterisco `*`[4]. Así pues, el valor contenido en la dirección `&x`, es `*&x`, y por lo tanto, `*&x == x`. Cuando escribimos una declaración de la forma `int * p`, estamos indicando que el parámetro `p` contiene una dirección que apunta a un tipo `int`. Por ese motivo se suele decir que `p` es un *apuntador*.

En el siguiente ejemplo codificamos en C la acción intercambio para que analices lo expuesto anteriormente.

[4]No debes confundir el uso de `*` como operador de acceso al contenido de una dirección con la operación de multiplicación ya que ambas usan el mismo símbolo.

accion intercambio(**ent/sal** a, b : *entero*);
$\{P : a = A \wedge b = B\}$
var aux: entero **fvar**
    aux:= a;
    a:= b;
    b:= aux
$\{Q : a = B \wedge b = A\}$
faccion

Un programa en C que codifica la acción anterior se muestra a continuación.

```
#include <stdio.h>
//
void intercambio(int *a, int *b)
{
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
    return;
}
//
int main(void)
{
    int x;
    int y;
    y = 21;
    x = 12;
    intercambio(&x, &y);
    printf("x es%d e y es%d \n", x, y);
    return 0;
}
```

Debes observar que la variable `a` de tipo entero de la acción, al ser de entrada/salida la codificamos como `int *a` para reflejar en la función de C ese hecho. Ahora `a` es un apuntador. En la codificación de las instrucciones del cuerpo de la función en C debemos escribir `*a` para acceder al contenido apuntado por `a`. Lo mismo podemos decir para la variable `b` de la acción.

Como hemos visto, las cosas son un poco más sencillas en el caso de codificar funciones del lenguaje algorítmico a C. Veamos otro ejemplo simple de codificación de una función. Supongamos que se quiere codificar en C la siguiente función

función val_abs(x: *entero*) **dev** y: *entero*;
$\{P$: verdadero$\}$
    si x $\geq 0 \longrightarrow$ y:= x
      [] x $< 0 \longrightarrow$ y:= -x
    fsi;
$\{Q : y = |x|\}$
    dev y
ffunción

Recuerda que x es un parámetro de entrada y que no realizamos instrucciones de asignación sobre él. La codificación en C de esta función es la siguiente,

```
#include <stdio.h>
//
int val_abs(int x)
{
    int y;
    if (x >= 0)
        y = x;
    else
        y = -x;
    return y;
}
//
int main(void)
{
    ....
    return 0;
}
```

Como se puede ver, se ha declarado la variable local `y` dentro de la función que hace las veces del parámetro de salida. Además la instrucción (**dev** y) en el lenguaje algorítmico simplemente se codifica como la sentencia `return y;` en C. Todo lo que hemos explicado en este capítulo es válido para cualquier tipo de datos simple. En las sesiones de prácticas dedicadas a la estructura de datos del tipo tabla volveremos sobre la forma de definición de parámetros y paso de argumentos con variables de este tipo de datos.

4.3. Actividades a realizar

Ejercicio 1. Lista de exponenciales

Escribe un programa en C denominado `lista_expo`. El programa codifica la función *serie_exponencial* dada en el ejercicio 1.A en forma de función con el nombre `serie_exponencial` y la función *potencia* como otra función con nombre `potencia`. El programa pide al usuario un número positivo N, y muestra en pantalla el valor de cada serie exponencial desde 0 hasta N.

Ejercicio 2. Lista de Cantor

Codifica por medio de una acción el algoritmo dado en el ejercicio 1 de la Sesión 3, `accion inversa_de_Cantor(...)`. Si observas con detenimiento el algoritmo, éste tiene un parámetro de entrada y dos parámetros de salida. Así la declaración de la acción vendrá dada por

```
accion inversa_de_Cantor(e/ r: entero; sal/ m, n: entero);
```

Codifica un programa en C denominado `lista_de_Cantor`. El programa solicita un número entero R mayor o igual a 1 y escribe la lista de las inversas de Cantor de los números desde 1 hasta R según el formato que se indica:

```
........
Introduce un numero R mayor o igual a uno: 6
La inversa de Cantor para 1 es, K(1)=... y L(1)=...
La inversa de Cantor para 2 es, K(2)=... y L(2)=...
...........
La inversa de Cantor para 6 es, K(6)=... y L(6)=...
...........
```

Nota: Por claridad, en todos los casos anteriores los parámetros formales que utilicéis en la declaración de las acciones y funciones deben tener el mismo nombre que las variables que se utilizan en las especificaciones de los algoritmos correspondientes. Como siempre, si tienes alguna duda consulta al profesor.

Nota: A medida que avanzáis en la codificación de algoritmos en C y en el conocimiento de este lenguaje de programación encontráis que ciertas características de C permiten escribir códigos más sintéticos y que ciertas cuestiones sobre las normas de estilo de codificación son demasiado estrictas y no utilizan todos los elementos del lenguaje. Por ejemplo, es más simple poner `i++` en algunas sentencias en vez de `i = i + 1` o que una instrucción como `s = s + a` pueda escribirse como `s += a`. Incluso utilizando el hecho de que C no cambia el argumento de un parámetro de entrada, las funciones presentan códigos más simples. Por ejemplo, la función valor absoluto la podríamos haber escrito como sigue:

```
#include <stdio.h>
//
int val_abs(int x)
{
    if (x < 0)
        x = -x;
    return x;
}
//
int main(void)
{
    ....
    return 0;
}
```

No hay inconveniente en que después de haber codificado según las normas utilices las facilidades propias de C para generar otra versión del programa. En ese caso, en las entregas de prácticas podéis entregar **las dos versiones** del mismo programa.

Ejercicio 3. Cálculo de aproximaciones de π

El matemático inglés del siglo XVII John Wallis descubrió la siguiente fórmula para calcular el valor de la constante π

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

donde los puntos suspensivos del numerador y del denominador abrevian un producto que es necesariamente infinito. La fórmula de Wallis

$$\frac{\pi}{4} = \prod_{j \geq 1} \frac{4 \cdot j \cdot (j+1)}{(2 \cdot j + 1)^2} = \prod_{j \geq 1} (1 - \frac{1}{(2 \cdot j + 1)^2})$$

se puede emplear para obtener aproximaciones de π si el producto infinito que en ella aparece se trunca y queda formado por un número finito de factores. Dada la especificación

n: *entero*;
p: *real*;
$\{Pre : n = N \ \wedge \ N \geq 0\}$
aproximacion_de_pi
$\{Post : p = 4 \cdot \prod_{1 \leq j \leq n} (1 - \frac{1}{(2 \cdot j + 1)^2}) \ \wedge \ n = N\}$

deriva un algoritmo que la satisfaga.

Ejercicio 3.A. Derivación del algoritmo

• *Propuesta de invariante.* A partir de la postcondición

$$Post : p = 4 \cdot \prod_{1 \leq j \leq n} (1 - \frac{1}{(2 \cdot j + 1)^2}) \ \wedge \ n = N$$

podemos sustituir la variable n por una nueva variable k de tipo entero, obteniendo el invariante

$$Inv : p = 4 \cdot \prod_{1 \leq j \leq k} (1 - \frac{1}{(2 \cdot j + 1)^2}) \ \wedge \ 0 \leq k \leq n \ \wedge \ n = N$$

• *Cálculo de la condición de continuación.* Se debe cumplir $Inv \wedge \neg B \Rightarrow Post$. En este caso $\neg B : k = n$, luego $B : k \neq n$.

• *Cálculo de las instrucciones de inicio.* Como k debe terminar siendo n; por su rango $0 \leq k \leq n$ puede comenzar en $k = 0$. En ese caso el domino del producto es el dominio vacío y el valor de p será 4. Luego $k := 0$; $p := 4$.

Antes de seguir comprueba $Pre \Rightarrow ((Inv)_4^p)_0^k$.

• *Cálculo de las instrucciones de avanzar.* Como k comienza en 0, termina en n, y el producto incluye a todos los términos, entonces la propuesta de avanzar es $k := k + 1$.

• *Cálculo de las instrucciones de restablecer.* Se debe cumplir

$$\{Inv \wedge B\} \text{ 'restablecer' } \{(Inv)_{k+1}^k\}$$

Calcular los predicados

$Inv \wedge B$: ...
$(Inv)_{k+1}^k$: ...

Comprueba primero lo que se cumple en $(Inv)_{k+1}^{k}$ deducido de lo que ya se cumple en $Inv \wedge B$. Identifica la variable que cambia su valor y relaciona las expresiones para obtener la instrucción que se necesita.

• *Cálculo de la función de cota.* En este caso es sencillo, basta con proponer $cota = n - k$. Debes comprobar

a) $Inv \wedge B \Rightarrow n - k > 0$
b) $Inv \wedge B \wedge n - k = T \Rightarrow (n - k < T)_{k+1}^{k}$

• *Diseño de la función.* A partir de la derivación anterior completa el diseño del algoritmo en forma de una función de nombre aproximacion_de_pi.

función aproximacion_de_pi (.....) **dev** p: *real*
{$Pre \equiv$ .....}
    ...
{$Post \equiv$ .....}
    dev p
ffunción

Ejercicio 3.B. Codificación en C

Escribe un programa en C con nombre `lista_de_pi`. El programa codifica en C la función `aproximacion_de_pi` obtenida en el apartado anterior. El programa solicita al usuario un número R mayor que cero y presenta por la pantalla una lista de las $R + 1$ primeras aproximaciones de pi.

```
.........
Introduce un numero entero mayor o igual a cero: 5
Aproximacion 0 de pi:...
Aproximacion 1 de pi:...
.......................
Aproximacion 5 de pi:...
............
```

Ejercicio 4. Cálculo de los números de Catalan

Los números de Catalan (E. Charles Catalan, 1814-1894) se utilizan para contar el número de casos distintos que ocurren en algunos tipos de problemas combinatorios. Por ejemplo, el número de Catalan $C(n)$ indica el número de expresiones que contienen n pares de paréntesis correctamente colocados. Así, con $n = 3$ pares de paréntesis puedo formar las $C(3) = 5$ expresiones ((())), ()(()), ()()(), (())() y (()()). La fórmula recursiva para calcular estos números viene dada de la siguiente manera

$$C(n) = \begin{cases} 1 & \text{si } n = 0, \\ \dfrac{2(2n-1)}{n+1}\, C(n-1), & \text{si } n > 0. \end{cases}$$

Dada la ecuación recursiva anterior diseña un algoritmo que resuelva el problema siguiente

c, n: *entero*;
$\{Pre : n = N \ \wedge \ N \geq 0\}$
numero_catalan
$\{Post : c = C(N)\}$

Ejercicio 4.A. Derivación del algoritmo

Antes de comenzar el diseño del algoritmo es importante ver con un ejemplo cómo actúa la definición recursiva anterior. Tienes que tener en cuenta que todo lo que necesitas para diseñar el algoritmo se encuentra en dicha definición: la condición de terminación, las instrucciones de inicio, cómo elegir la instrucción de avanzar y prácticamente resueltas las instrucciones de restablecer.

Por ejemplo, si elejimos $n = 4$ el cálculo para $C(4)$ es

$$C(4) = \frac{2(2*4-1)}{4+1} \quad C(3) = \frac{14}{5}\ \frac{2(2*3-1)}{3+1} \quad C(2) = \frac{14}{5}\ \frac{10}{4}\ \frac{2(2*2-1)}{2+1}$$
$$C(1) = \frac{14}{5}\ \frac{10}{4}\ \frac{6}{3}\ \frac{2(2*1-1)}{1+1} \quad C(0) = \frac{14}{5}\ \frac{10}{4}\ \frac{6}{3}\ \frac{2}{2}\ 1 = 14$$

El resultado es un número natural.

• *Propuesta de invariante.* La función es recursiva lineal no-final que utiliza como operación base únicamente la operación de multiplicación ($\times$). El invariante utiliza la misma variable de salida c para ir acumulando los cálculos hasta el final. La idea es: lo que quiero obtener al final $C(N)$ es igual a lo que voy acumulando en la variable c multiplicado ($\times$) por lo que me queda de calcular del resto de la función $C(n)$ sobre la variable n. En el invariante hay que añadir las condiciones de la función sobre la variable n, es decir $n \geq 0$. En resumen, el invariante propuesto es

$$Inv : C(N) = c \times C(n) \wedge n \geq 0.$$

• *Cálculo de la condición de continuación.* se debe cumplir $Inv \wedge \neg B \Rightarrow Post$. Basta mirar la definición de la función para saber que el cálculo termina cuando $n = 0$. En este caso $\neg B : n = 0$, luego $B : n \neq 0$.

• *Cálculo de las instrucciones de inicio.* El invariante tiene que ser cierto antes de entrar en el bucle, como $C(N) = c \times C(n)$ en el Inv y al principio, en la precondición, se toma el valor inicial de n, es decir $n = N$, entonces $C(N) = c \times C(N)$. Esto sólo puede ser cierto si la variable c toma el valor inicial 1, luego $c := 1$.

Antes de seguir comprueba $Pre \Rightarrow (Inv)^c_1$.

• *Cálculo de las instrucciones de avanzar.* Como n comienza en N y el cálculo termina cuando $n = 0$, entonces la propuesta de avanzar es $n := n - 1$.

Por otro lado, basta mirar en la definición de la función. $C(n) = \frac{2(2n-1)}{n+1}\ C(n-1)$ cuando $n > 0$. La función termina su cálculo porque n va disminuyendo mediante $n - 1$.

• *Cálculo de las instrucciones de restablecer.* Se debe cumplir

$$\{Inv \wedge B\} \text{ 'restablecer' } \{(Inv)^n_{n-1}\}$$

Siendo los predicados

$$Inv \wedge B : C(N) = c \times C(n) \wedge n \geq 0 \wedge n \neq 0$$
$$(Inv)^n_{n-1} : C(N) = c \times C(n-1) \wedge n - 1 \geq 0$$

Comprueba primero lo que se cumple en $(Inv)^n_{n-1}$ y qué se deduce de $Inv \wedge B$. Identifica la variable que cambia su valor y relaciona las expresiones para obtener la instrucción que se necesita.

En el caso de las funciones recursivas para obtener la solución tienes que **desplegar** la función en la parte $Inv \wedge B$. Primero, obseva que $n \geq 0 \wedge n \neq 0$ implica $n > 0$. Entonces, en $Inv \wedge B$ tienes $C(N) = c \times C(n) \wedge n > 0$. Aquí es donde despliegas el valor de $C(n)$ según su definición.

$$Inv \wedge B : C(N) = c \times \frac{2(2n-1)}{n+1}\ C(n-1) \wedge n > 0$$

comparando con $(Inv)^n_{n-1} : C(N) = c \times C(n-1) \wedge n - 1 \geq 0$, la variable c tiene que acumular la expresión $c \times \dfrac{2(2n-1)}{n+1}$, luego la instrucción de restablecer debería ser $c := c \times \dfrac{2(2n-1)}{n+1}$, pero, teniendo en cuenta que c es una variable de tipo entero, finalmente $c := (c * 2 * (2n-1))$ **div** $(n+1)$.

• *Cálculo de la función de cota.* En este caso basta con proponer cota=n. Debes comprobar

a) $Inv \wedge B \Rightarrow n > 0$
b) $\{Inv \wedge B \wedge n = T\}$ 'restablecer'; $n := n - 1$ $\{n < T\}$

Reflexión: Todo parece haber ido bien, pero la instrucción de restablecer está mal ya que no se ha comprobado que (c*2*(2n-1)) sea divisible por (n+1). Esta es la condición para sustituir la expresión $c \times \dfrac{2(2n-1)}{n+1}$ calculada, por la expresión $(c*2*(2n-1))$ **div** $(n+1)$ en la asignación a c. De hecho, si programas el algoritmo veras que no funciona porque la división no siempre es exacta. La función matemática te asegura que lo es al final. Para corregir este defecto basta con que observes que la variable c acumula una fracción, que siempre puede ser descrita con dos variables, una que represente el numerador *num* y otra que represente el denominador *den*. Así que podemos reescribir el invariante de la siguiente forma

$$Inv' : C(N) = \frac{num}{den} \times C(n) \wedge n \geq 0$$

Ahora, no aparece c en el Inv' luego $Inv' \wedge \neg B$ no implica la postcondición. Se necesita un tratamiento final después de terminar el bucle para calcular c. La instrucción necesaria al término del bucle es sencilla, simplemente c:= *num* **div** *den*.

• *Diseño del algoritmo.* A partir de la derivación anterior, completa el diseño del algoritmo

algoritmo numero_catalan;
n, c: *entero*;
$\{Pre : n = N \wedge N \geq 0\}$
var
    num, den: *entero*

fvar
num:= .....;
den:= .....;
$\{Inv' : C(N) = \frac{num}{den} \times C(n) \wedge n \geq 0\}$ {cota = n}
mientras n $\neq$ 0 **hacer**
num:= .....;
den:= .....;
n := n - 1
fmientras;
c:= num div den
$\{Post : c = C(N)\}$
falgoritmo

• *Diseño de la función.* A partir de la derivación anterior completa el diseño del algoritmo en forma de una función de nombre *numero_catalan*. Recuerda que se deben respetar las notas sobre funciones que dimos en la Sesión 4.

Ejercicio 4.B. Codificación en C

Escribe un programa en C, de nombre `catalanes`, tal que codifique la función `numero_catalan` obtenida en el apartado anterior. El programa pide al usuario un número R mayor o igual a cero y escribe por la pantalla los valores de los R primeros números de Catalan.

Sesión 5

Diseño de algoritmos iterativos: Tablas I. Búsquedas y recorridos

Los tipos básicos (booleano, carácter, entero y real), los enumerados y los subrangos son tipos de datos que no tienen estructura. En las clases de teoría se ha introducido el constructor de tipo **tabla** y la forma de declarar tablas en lenguaje algorítmico. En esta práctica vamos a estudiar cómo se declaran y se usan este tipo de datos en lenguaje C. También vamos a trabajar con la derivación de algoritmos iterativos que utilizan este tipo de estructuras.

5.1. Tablas en C

5.1.1. Declaración y utilización de tablas

Una tabla en C consiste en un número fijo de componentes (definidas cuando se declara la tabla), todos del mismo tipo, al que se llama *tipo base* o también *tipo del componente*. Una de las ventajas que presenta esta estructura es que cada componente puede denotarse explícitamente y es posible acceder a él directamente, mediante el nombre de la tabla seguido por el denominado *índice* entre corchetes. Los índices sirven para denotar el componente al cual nos estamos refiriendo en una determinada sentencia del programa. Además, el tiempo que se necesita para seleccionar o acceder a un determinado componente de la estructura no depende del valor del índice. Es por esta última e importante cualidad, por lo que las tablas se suelen denominar *estructuras de acceso directo*.

En el lenguaje algorítmico una declaración completa de una variable de tipo tabla incluye normalmente la declaración de una constante, la declaración de un subrango para establecer el dominio de los índices de la tabla y la declaración de un tipo tabla para luego poder establecer más cómodamente la declaración de variables. Por ejemplo,

constante N = 25, M = 30 **fconstante**
tipo Filas = 1.. N, Columnas = 1 .. M **ftipo**
tipo Vector = **tabla** [Filas] **de** entero **ftipo**
tipo Matriz = **tabla** [Filas, Columnas] **de** entero **ftipo**
// declaracion de variables
i: Filas
j: Columnas
v, z: Vector
m : Matriz

Para acceder a la décima posición de la variable `v` (que es una `tabla` de 25 posiciones de tipo entero indexadas de 1 a 25) basta con indicarlo de la forma

```
v[10]
```

Para acceder a la i_ésima fila y j_ésima columna de la variable m lo indicamos de la forma

```
m[i,j]
```

Si se desea realizar una asignación de un nuevo valor a una determinada posición de la matriz, lo expresamos mediante una asignación, como por ejemplo

```
m[i,j]:= 456
```

Como las variables son de tipo subrango y las tablas están acotadas, cualquier acceso a elementos no declarados de las tablas nos obliga a considerarlo un error. Obviamente, el código algorítmico lo ejecutamos nosotros mismos y el error lo generamos nosotros mismos. Algunos lenguajes de programación como Pascal reflejan la declaración anterior fielmente a nuestro entendimiento. Sin embargo, el lenguaje C, por razones de eficiencia, tiene un comportamiento más libre: no tiene subrangos, ni mira si te sales de los tamaños declarados en las tablas.

La codificación en C de la declaración anterior sólo respeta el tamaño de las tablas. La codificación sería

```
#define N 25
#define M 30
//
typedef int Vector[N];
typedef int Matriz[N][M];
//
void main(void)
{
    int i,j;
    Vector v, z;
    Matriz m;
    .....
}
```

El vector `v` comprende los elementos `v[0], v[1],...,v[N-1]`, es decir los índices para acceder al vector `v` quedan implícitos en la declaración y comienzan siempre con valor `0`. Como el tamaño es `N`, el último ínidice (para no salirte del vector) es `N-1`. Lo mismo sucede en la matriz declarada. Los ídices de las filas van desde `0` hasta `N-1` y los de las columnas desde `0` hasta `M-1`. Una vez creados, los valores que contienen estas tablas son completamente arbitrarios.

Podemos preguntarnos cuál es la razón para que C tenga este comportamiento con los índices:

La dirección de un elemento del vector `v`, por ejemplo, `v[6]` es `&v[6]`.
La dirección del primer elemento del vector `v` es `&v[0]`.
Se cumple que `&v[6] == &v[0] + 6* sizeof(int)`

El cálculo de la dirección que ocupa un elemento (que depende del tipo de elemento, `sizeof(int)` te devuelve el número de bytes que ocupa en la memoria del ordenador los datos de tipo entero, en este caso 4 bytes) es un cálculo simple.

A este respecto podemos preguntarnos qué es realmente una variable declarada como sigue `int p[N];`. Por una parte `p[N]` declara una tabla de `N` posiciones, pero, ¿qué representa la variable `p` (sin los corchetes)?: pues no es más que un apuntador cuyo contenido es la dirección del primer elemento de la tabla, `p == &p[0]` (o también, `*p == p[0]`).

No podemos permitir, que la libertad que nos ofrece C con los apuntadores, nos lleve a realizar accesos ilícitos a la memoria, es decir, acceder a una posición de memoria reservada para otros propósitos en nuestro código. C no comprueba si los índices están dentro del rango permitido para que la ejecución sea más rápida. Por este motivo, el diseño y la codificación de algoritmos con tablas debe ser muy preciso y sin errores.

Para asignar los valores inciales de una tabla debes indicar al usuario del programa el elemento al que estás accediendo. Aquí tienes un ejemplo:

```
#include <stdio.h>
#define N 10
//
typedef int Vector[N];
//
void main(void)
{
    int i;
    Vector v;
//
    for(i = 0; i <N; i++)
    {
        printf("Introduce un valor entero en v[%d]: \n", i);
        scanf("%d", &v[i]);
    }
    .....
    return;
}
```

Resulta de interés definir tipos para las tablas en C, por ejemplo, `typedef int Vector[N];` puesto que explícitamente incluyen el tamaño de la tabla.

De lo indicado anteriormente, se deduce que cuando declaramos en una función en C un parámetro de tipo tabla, éste parámetro es del tipo entrada/salida desde el punto de vista del lenguaje algorítmico. Por ese motivo, si el parámetro en el lenguaje algorítmico es sólo de entrada no debemos efectuar instrucciones de asignación sobre él cuando escribimos el

cuerpo de la función. Un ejemplo de paso de parámetros del tipo tabla en C se muestra a continuación.

```
#include <stdio.h>
//
#define N 10
//
typedef enum {FAlSE = 0, TRUE = 1} boolean;
typedef int Vector[N];
//
void incrementa(Vector a)
{
    int i;
    for (i = 0; i < N; i = i+1)
        a[i] = a[i] + 1;
    return;
}
boolean existe_cero(Vector a)
{
    int i;
    boolean b;
    i = 0;
    while((i<(N-1)) && (a[i]!= 0))
        i = i + 1;
    b = (a[i] == 0);
    return b;
}
//
void main(void)
{
    int i;
    Vector v;
    boolean b;
//
    for(i = 0; i < N; i = i+1)
    {
        printf("Introduce un valor entero en v[%d]: \n", i);
        scanf("%d", &v[i]);
    }
    printf("El vector v:( ");
    for (i = 0; i < N; i = i+1)
        printf("%d ", v[i]);
    printf(")\n");
    incrementa(v);
    printf("El vector v incrementado:( ");
    for (i = 0; i < N; i = i+1)
        printf("%d ", v[i]);
    printf(")\n");
    b = existe_cero(v);
    if (b)
```

```
            printf("El vector v contiene algun elemento nulo\n");
        return;
    }
```

Observa que en `incrementa(Vector a)`, el parámentro `a` es de entrada/salida, y en `existe_cero(Vector a)`, el parámentro `a` es sólo de entrada.

5.2. Actividades a realizar

Para ejercitarnos con la utilización de tablas vamos a realizar varios programas, a partir de la derivación de varios algoritmos iterativos que utilizan esta estructura de datos.

Ejercicio 1. Elementos no repetidos

Diseña un algoritmo que resuelva el siguiente problema, siendo N una constante entera positiva,

t: **tabla** [0..N] **de** *entero*;
r: *booleano*;
$\{Pre : t = T\}$
elementos_no_repetidos
$\{Post : t = T \wedge\ r \equiv (\forall k : 0 \leq k < N : (\forall l : k < l \leq N : t[k] \neq t[l]))\}$

Ejercicio 1.A. Derivación del algoritmo

Algunos ejercicios con tablas tienen la apariencia de poder ser resueltos fácilmente mediante un esquema de recorrido. Eso puede ser cierto en el caso de que aparezca en la postcondición del problema el cuantificador universal. Sin embargo, en algunos casos, incluso cuando aparece el cuantificador universal, se trata de problemas de búsqueda y, por lo tanto, la manera más correcta es obtener la solución mediante un esquema de búsqueda. Habrá que analizar detenidamente en cada caso si se trata de un problema de búsqueda o de recorrido para aplicar el esquema adecuado.

Observa que en el problema planteado, se devuelve en la variable booleana r el valor verdadero si todos los elementos de la tabla son diferentes, y el valor falso si hay dos elementos iguales en posiciones diferentes. Por tanto, este problema se puede transformar en el problema de 'buscar dos elementos en la tabla, en posiciones distintas, tales que ambos sean iguales'. Es decir, a pesar del cuantificador universal que aparece en la postcondición, éste es un problema de búsqueda.

En general, los problemas de búsqueda tienen una postcondición de la forma $t = T \wedge\ b \equiv (\exists k : 0 \leq k \leq N : A(k,t))$. Lo que vamos a hacer, es transformar la postcondición del problema para encontrar esta estructura con el cuantificador existencial. Si tenemos en cuenta la propiedad $[[(\forall k : C(k)) \equiv \neg(\exists k : \neg C(k))]]$, podemos poner

$$Post : t = T \wedge\ r \equiv (\forall k : 0 \leq k < N : (\forall l : k < l \leq N : t[k] \neq t[l]))$$
$$Post : t = T \wedge\ r \equiv \neg(\exists k : 0 \leq k < N : (\exists l : k < l \leq N : t[k] = t[l]))$$
$$Post : t = T \wedge\ r \equiv \neg(\exists k : 0 \leq k \leq N-1 : (\exists l : k+1 \leq l \leq N : t[k] = t[l]))$$

Si ahora definimos $A(k,t) : (\exists l : k+1 \leq l \leq N : t[k] = t[l])$, tenemos identificada la estructura de un problema de búsqueda

$$Post : t = T \wedge \ r \equiv \neg(\exists k : 0 \leq k \leq N-1 : A(k,t))$$

Para obtener la solución planteamos un esquema de búsqueda en tablas, utilizando como variable de búsqueda la variable x. El dominio de la búsqueda en este caso va desde 0 hasta $N-1$.

t: **tabla** [0..N] **de** *entero*;
r: *booleano*;
$\{Pre : t = T\}$
var x:*entero* **fvar**
    x:= 0; (* inicio de la búsqueda *)
$\{Inv : (\forall k : 0 \leq k < x : \neg A(k,t)) \ \wedge \ 0 \leq x \leq N-1 \ \wedge \ t = T\}$
$\{cota = N - x\}$
    mientras $\neg A(x,t)$ $\wedge$ x<(N-1) **hacer**
        x:= x+1
    fmientras;
    r:= **no** $A(x,t)$ (* éxito de la búsqueda *)
$\{Post : t = T \wedge \ r \equiv \neg(\exists k : 0 \leq k \leq N-1 : A(k,t))\}$

Por la teoría, ya sabemos que el planteamiento anterior es correcto. El problema que tenemos ahora es que $A(x,t)$ no se puede programar en la condición de continuación[1]. Utilizamos el método de reforzamiento del invariante. Así, añadimos una nueva variable al invariante, en este caso z, que sea igual a la expresión que no podemos programar.

$$Inv' : (\forall k : 0 \leq k < x : \neg A(k,t)) \ \wedge \ 0 \leq x \leq N-1 \ \wedge \ t = T \ \wedge \ z = A(x,t)$$

donde $A(x,t) : (\exists l : x+1 \leq l \leq N : t[x] = t[l])$.

Al repasar la solución anterior con el nuevo invariante Inv' tenemos que calcular una nueva instrucción de inicio y una nueva instrucción de restablecer para la variable z, de forma que se cumpla

$$Pre \Rightarrow ((Inv')^z_{val\_ini\_z})^x_0$$
$$Inv' \wedge B \Rightarrow ((Inv')^z_{restablecer\_z})^x_{x+1}$$

Si hacemos los cálculos, ésto nos da $z := A(0,t)$ y $z := A(x+1,t)$ respectivamente. La expresión booleana $A(0,t)$ no toma un valor sencillo. De la misma forma, $A(x+1,t)$ no se puede calcular de forma simple teniendo en cuenta su valor previo $A(x,t)$.

Una forma elegante de continuar la resolución del ejercicio es escribir el código de una función que calcule en general el valor de $A(x,t)$, siendo x un parámentro de entrada de tipo entero y t un parámetro de entrada del tipo de la tabla que estamos considerando. Por simplificar, la declaración de la función sería:

```
funcion calcula_A(x: entero, t: tablaON) dev b:booleano
```

Hasta aquí, el algoritmo nos queda con la siguiente estructura:

[1]Cuando en la condición de continuación aparece una expresión que no se puede programar es obligatorio reforzar el invariante con justamente, la expresión que no se puede programar.

algoritmo elementos_no_repetidos;
constante N (> 0) **fconstante**
tipo tabla0N= **tabla** [0..N] **de** *entero* **ftipo**
    funcion calcula_A(x:*entero*, t: tabla0N) **dev** b:*booleano*
    $\{P : 0 \leq x \leq N-1\}$
    $\{Q : b \equiv (\exists l : x+1 \leq l \leq N : t[x] = t[l])\}$
    ffuncion
t: tabla0N;
r: *booleano*;
$\{Pre : t = T\}$
var x:*entero*, z:*booleano* **fvar**
    z:= calcula_A(0,t);
    x:= 0;
$\{Inv' : (\forall k : 0 \leq k < x : \neg A(k,t)) \ \wedge \ 0 \leq x \leq N-1 \ \wedge \ t = T \ \wedge \ z = A(x,t)\}$
$\{cota = N - x\}$
    mientras $\neg z \wedge$ x<(N-1) **hacer**
        z:= calcula_A(x+1,t);
        x:= x+1
    fmientras;
    r:= $\neg z$
$\{Post : t = T \wedge \ r \equiv \neg(\exists k : 0 \leq k \leq N-1 : A(k,t))\}$

Observa que, como en el invariante $z = A(x,t)$, hemos modificado adecuadamente, tanto la programación de la condición de continuación como el tratamiento final.

Completa el algoritmo anterior dando una codificación de la función `calcula_A(...)`. La solución se obtiene siguiendo de nuevo el esquema de búsqueda en tablas.

Ejercicio 1.B. Codificación en C

Escribe un programa en C, de nombre `repeticiones`, tal que codifique el algoritmo del apartado anterior como una función `elementos_no_repetidos`. El programa debe pedir al usuario valores enteros para crear una tabla de $N+1$ elementos (con $N = 6$). A continuación, debe mostrar por pantalla el contenido de la tabla en una sola línea y después el mensaje 'No existen elementos repetidos' en caso de que no haya ningún elemento repetido en la tabla, o en caso contrario el mensaje 'Existe algún elemento repetido en la tabla'.

Ejercicio 2. Cuenta impares duplicados

El siguiente ejercicio es un ejemplo concreto de cómo debe proceder un estudiante en su estudio individual sobre algunos de los ejercicios de tablas.

Enunciado del problema (*Cuenta impares duplicados*): Diseñe un algoritmo iterativo que cuente el número de valores impares almacenados en la tabla t que también aparecen en la tabla h, de acuerdo con la especificación siguiente:

t: **tabla** [1..N] **de** *entero*;
h: **tabla** [1..M] **de** *entero*;
r: *entero*;

$\{Pre : t = T \wedge h = H\}$
cuenta_nones_duplicados
$\{Post : t = T \wedge h = H \wedge r = (\#k{:}\,1..N{:}\,t[k] \bmod 2 = 1 \wedge (\exists\, l{:}\,1..M{:}\,t[k] = h[l]))\}$

El algoritmo se ha de construir utilizando los esquemas algorítmicos conocidos. Se deben seguir los siguientes pasos:

1. Identificar el problema (o sus subproblemas) según alguno de los esquemas conocidos (recorrido, búsqueda, etc.);
2. Construir el algoritmo y las funciones que codifiquen los subproblemas siguiendo los esquemas correspondientes; y,
3. Escribir los invariantes y las funciones de cota para el (o los) bucle(s) que aparezcan en la solución, a partir de los invariantes de los esquemas aplicados.

Solución: En la variable r se tiene que obtener el número de veces que un valor impar de la tabla $t[]$ se encuentra "*duplicado*" en la tabla $h[]$.

Veámoslo con un ejemplo, sea la tabla t:

1	*2*	*3*	*4*	*5*
3	0	-2	7	4

y la tabla h:

1	*2*	*3*	*4*	*5*	*6*	*7*
3	1	3	6	3	8	2

Obtenemos que $r = 1$ ya que, de todos los impares de t ($t[1] = 3$ y $t[4] = 7$), sólo el valor 3 se encuentra en h y no importa el número de veces que aparezca repetido (la pertenecia a h se da a través de un existe $\exists$...). Del mismo modo, podemos comprobar que el valor 7 no está en h y, por tanto, el valor de r no se incrementa.

Por un lado, tenemos que recorrer todos los elementos de la tabla t de 1 a N, estamos hablando de un problema de recorrido. Luego, por cada elemento de t, comprobar si es impar y ver, si además, está en la tabla h. Si esto es así, cada vez que ocurra, r irá incrementando su valor de uno en uno (# es el operador que representa el recuento de una expresión boleana).

Escribimos la postcondición así:

$\{Post : t = T \wedge h = H \wedge r = \#k{:}\,1 \leq k \leq N{:}\,B(t[k], h)\}$

donde $B(t[k], h)$ se define como

$B(t[k], h) : t[k] \bmod 2 = 1 \wedge (\exists\, l{:}\,1 \leq l \leq M{:}\,t[k] = h[l])$.

Siguiendo el esquema de recorrido, para definir el invariante, necesitamos una variable de recorrido que vaya apuntando a cada elemento de t, a la que denotamos con i de tipo

entero. Si re-escribimos la postcondición de acuerdo al esquema de recorrido estándar tenemos que

$$\{Post : t = T \ \wedge \ h = H \ \wedge \ r = (\#k{:}\, 1 \leq k < (\mathbf{N} + \mathbf{1}){:}\, B(t[k], h))\}$$

Así, procedemos a la sustitución de la constante $(N + 1)$ por la variable i, obteniendo el invariante:

$$\{Inv : t = T \ \wedge \ h = H \ \wedge \ r = (\#k{:}\, 1 \leq k < i{:}\, B(t[k], h)) \ \wedge \ 1 \leq i \leq N + 1\}$$

A continuación procedemos a escribir el esquema de recorrido, de acuerdo a las condiciones de nuestra especificación:

t: **tabla** $[1..N]$ **de** *entero*;
h: **tabla** $[1..M]$ **de** *entero*;
r: *entero*;
$\{Pre : t = T \ \wedge \ h = H\}$
var i: *entero* **fvar**
$i := 1$;
tratamiento inicial
$\{Inv : t = T \ \wedge \ h = H \ \wedge \ r = (\#k{:}\, 1 \leq k < i{:}\, B(t[k], h)) \ \wedge 1 \leq i \leq N + 1\}$
$\{cota = N + 1 - i\}$
mientras $i \neq N + 1$ **hacer**
  tratar el elemento i_ésimo;
  $i := i + 1$
fmientras;
$\{Post : t = T \ \wedge \ h = H \ \wedge \ r = (\#k{:}\, 1..N{:}\, t[k] \bmod 2 = 1 \ \wedge (\exists\, l{:}\, 1..M{:}\, t[k] = h[l]))\}$

Vamos a calcular las instrucciones que nos faltan siguiendo el esquema de derivación.

- *"tratamiento inicial"*. Cuando $i = 1$, la variable r en el invariante es $r = (\#k{:}\, 1 \leq k < 1{:}\, B(t[k], h))$. Si nos fijamos en el dominio de la variable ligada k, $1 \leq k < 1$, es vacío ($\emptyset$), luego $r = 0$ y, por tanto $r \leftarrow 0$.

- *"tratar el elemento i_ésimo"*. Su tratamiento en el bucle va a depender de si se cumple $B(t[i], h)$. Si denotamos al nuevo valor de r como r_{new} y al viejo r_{old}, respectivamente, podemos definir el nuevo valor de r como:

  - $r_{new} \leftarrow r_{old} + 1$ si $B(t[i], h)$ es verdadero
  - $r_{new} \leftarrow r_{old}$ si $B(t[i], h)$ es falso

De todo ello, nos quedaría el siguiente algoritmo:

t: **tabla** $[1..N]$ **de** *entero*;
h: **tabla** $[1..M]$ **de** *entero*;
r: *entero*;
$\{Pre : t = T \wedge h = H\}$
var i: *entero* **fvar**
$i := 1$;
$r := 0$;
$\{Inv : t = T \wedge h = H \wedge r = (\#k{:}\, 1 \leq k < i{:}\, B(t[k], h)) \wedge 1 \leq i \leq N + 1\}$
$\{cota = N + 1 - i\}$
mientras $i \neq N + 1$ **hacer**
  si $B(t[i], h) \rightarrow r := r + 1$
  $[]\ \neg B(t[i], h) \rightarrow$ continuar
  fsi;
  $i := i + 1$
fmientras;
$\{Post : t = T \wedge h = H \wedge r = (\#k{:}\, 1..N{:}\, t[k] \bmod 2 = 1 \wedge (\exists\, l{:}\, 1..M{:}\, t[k] = h[l]))\}$

Nos queda por responder a la pregunta de cuál es el predicado booleano $B(t[i], h)$. Su expresión es la siguiente:

$$B(t[i], h) : t[i] \bmod 2 = 1 \wedge (\exists\, l{:}\, 1 \leq l \leq M{:}\, t[i] = h[l])$$

De esta expresión podemos programar $t[i] \bmod 2 = 1$ pero no podemos programar $\exists\, l{:}\, 1 \leq l \leq M{:}\, t[i] = h[l]$. Podríamos reforzar el invariante con una nueva variable b tal que $b \equiv \exists\, l{:}\, 1 \leq l \leq M{:}\, t[i] = h[l]$ si se pudiera calcular el nuevo valor de b a partir de uno anterior pero no es el caso. Luego lo más intuitivo en este punto es aplicar el diseño descendente definiendo una función:

funcion esta_en(h: **tabla** $[1..M]$ **de** *entero*, z: *entero*) **dev** b: *booleano*;
  $\{P :$ verdadero$\}$
  $\{Q : b \equiv \exists\, l{:}\, 1 \leq l \leq M{:}\, z = h[l]\}$
ffunción

El problema a resolver por la función es buscar en h si el valor de z está o no en la tabla. Este problema se corresponde a una búsqueda lineal simple.

algoritmo cuenta_nones_duplicados;
  función esta_en(h: **tabla** $[1..M]$ **de** *entero*, z: *entero*) **dev** b: *booleano*;
    $\{P$: verdadero$\}$
     completar ....
    dev b
    $\{Q : b \equiv \exists\, l{:}\, 1 \leq l \leq M{:}\, z = h[l]\}$
  ffunción
t: **tabla** $[1..N]$ **de** *entero*;
h: **tabla** $[1..M]$ **de** *entero*;
r: *entero*;
$\{Pre : t = T \wedge h = H\}$
var i: *entero*, b: *booleano* **fvar**
  $r := 0$;
  $\{Inv : t = T \wedge h = H \wedge r = (\#k{:}\, 1 \leq k < i{:}\, t[k] \bmod 2 = 1 \wedge$
      $\wedge(\exists\, l{:}\, 1 \leq l \leq M{:}\, t[k] = h[l])) \wedge 1 \leq i \leq N + 1\}$
  $\{cota \equiv N + 1 - i\}$
  para $(i := 1;\ i < N + 1;\ i := i + 1)$ **hacer**
    $b :=$ esta_en$(h, t[i])$;
    si $(t[i] \bmod 2 = 1 \wedge b) \rightarrow r := r + 1$
    [] $(t[i] \bmod 2 \neq 1 \vee \neg b) \rightarrow$ continuar
    fsi
  fpara
$\{Post : t = T \wedge h = H \wedge r = (\#k{:}\, 1..N{:}\, t[k] \bmod 2 = 1 \wedge$
    $\wedge(\exists\, l{:}\, 1..M{:}\, t[k] = h[l]))\}$
falgoritmo

Completa el código de la función. En la solución final se ha utilizado la estructura *para...hacer*, así se deja claro que se trata de un problema de recorrido que incluye en cada paso una búsqueda.

Para terminar, codifica en C el algoritmo anterior, `cuenta_nones_duplicados`, y realiza las pruebas correspondientes con diferentes tablas.

Nota Importante: a la hora de codificar en C el algoritmo anterior tienes que tener en cuenta que el tamaño de las tablas en la declaración del algoritmo son N y M, y que los índices van de $1..N$ y de $1..M$ respectivamente. Quizás lo mas cómodo sea declarar en C dos tablas de tamaño $N + 1$ y $M + 1$ para que sus índices vayan desde $0..N$ y $0..M$ respectivamente. De esta forma, lo único que cambiaría en la codificación del algortimo anterior es que no vas a usar ni el elemento $t[0]$ ni $h[0]$. **Esto no se debe hacer porque es una mala práctica de codifcación**. En ocasiones te encontrarás diseños que debes adaptar a las características del lenguaje de codificación que estás utilizando. Deberás adaptar el algoritmo anterior para que los índices de recorrido de las tablas se ajusten a su correcta codificación en C. Tened siempre mucho cuidado con los límites de las tablas que se utilicen. Los recorridos no deben sobrepasar el espacio que ocupan.

Ejercicio 3. Cuenta divisores

Construye un algoritmo que, dadas dos tablas de N posiciones, indique el número de posiciones de la primera que contienen valores que son divisores de algún elemento contenido en la segunda

h, g: **tabla** [0..N-1] **de** *entero*;
s: *entero*;
$\{Pre : h = H \wedge g = G \wedge \forall i : 0..N-1 : h[i] \neq 0\}$
cuenta_divisores
$\{Post : h = H \wedge g = G \wedge s = (\#i : 0..N-1 : (\exists j : 0..N-1 : g[j] \bmod h[i] = 0))\}$

El algoritmo se ha de construir utilizando los esquemas algorítmicos conocidos. Se deben seguir los pasos dados en el ejercicio 2 de esta Sesión.

Transforma el algoritmo anterior en una función, `funcion cuentadivisores(...) dev s: entero`. Escribe un programa en C, `cuenta_divisores`, que codifique en C la función anterior y la utilice en un programa que solicita dos tablas, las presenta en la pantalla en una línea cada una, y escribe el resultado obtenido mediante la función.

Ejercicio 4. Matriz simétrica

Especifique un problema algorítmico que indique si dada una matriz cuadrada de dimensión $N \times N$ es simétrica o no. Diseñe el algoritmo correspondiente y codifique en C dicho algoritmo.

Sesión 6

Diseño de algoritmos iterativos: Tablas II

El objeto de esta práctica es trabajar algunas derivaciones de algoritmos iterativos que utilizan tablas. Para profundizar en el uso de acciones y funciones se codificarán los algoritmos como acciones o funciones que se utilizarán en un programa en C.

6.1. Actividades a realizar

Ejercicio 1. Chequeo de ortogonalidad

Dadas dos tablas $t[0..N]$ y $h[0..N]$, siendo N una constante entera positiva, que se supone representan dos vectores en un espacio $N+1$-dimensional, se pide construir un algoritmo basado en un único bucle que calcule si ambos vectores son ortogonales, según la especificación siguiente

t, h: **tabla** [0..N] **de** *entero*;
b: *booleano*;
$\{Pre : t = T \ \wedge \ h = H\}$
son_ortogonales
$\{Post : t = T \ \wedge \ h = H \ \wedge \ b \equiv (\sum_{i=0}^{N} t[i] \times h[i] = 0)\}$

Derive la solución para el problema anterior. Escribe un programa en C, de nombre `ortogonalidad`, tal que codifique el algoritmo obtenido como una función `son_ortogonales`. El programa pedirá al usuario valores para crear dos tablas de $N+1$ elementos de tipo entero (con $N = 4$) y comparará ambas tablas para ver si son ortogonales o no. Finalmente, el programa tiene que mostrar por pantalla el contenido de ambas tablas, cada una en una sola línea, y a continuación el mensaje 'Las tablas son ortogonales' o bien 'Las tablas NO son ortogonales' en función de si son ortogonales o no.

Ejercicio 2. Partición de tablas

Sea N una constante entera positiva. Construye un algoritmo que satisfaga la especificación

x: *real*;
t: **tabla** [0..N] **de** *real*;
i: 0..N+1;
$\{Pre : x = X \wedge\ t = T\}$
partición_tabla
$\{Post : x = X \wedge\ t \in Perm(T) \wedge\ (\forall k : 0 \leq k < i : t[k] < x) \wedge$
$\wedge\ (\forall k : i \leq k \leq N : t[k] \geq x) \wedge 0 \leq i \leq N + 1\}$

El algoritmo propuesto como solución ha de constar de un único bucle cuyo invariante ha de ser el siguiente predicado

$Inv : x = X \wedge\ t \in Perm(T) \wedge\ 0 \leq i \leq j \leq N + 1 \wedge$
$\wedge\ (\forall k : 0 \leq k < i : t[k] < x) \wedge\ (\forall k : j \leq k \leq N : t[k] \geq x)$

Ejercicio 2.A. Derivación del algoritmo

Consideremos una tabla t, con $N = 5$, formada por los seis valores siguientes: (3.4, 7.6, 2.5, -1.0, 8.4, 0.3). Sea x =3.8. Analizando la postcondición, vemos que el algoritmo debe calcular un valor para i tal que todas las posiciones de la tabla con índices menores estrictamente que i contienen valores estrictamente menores que x y todas las posiciones de la tabla con índices mayores o iguales que i contienen valores mayores o iguales que x. La tabla queda dividida en dos partes $0..i-1$ y $i..N$. Además, como $t \in Perm(T)$, los valores en la tabla resultante son una permutación de los valores originales, es decir, se pueden reordenar pero no crear o destruir los valores originales. Teniendo en cuenta esto, para el ejemplo dado, el resultado para x =3.8 es $i = 4$ y t podrÃa ser (3.4, 2.5, -1.0, 0.3, 7.6, 8.4). Observa, que si el valor inicial de x es mayor que todos los elementos de la tabla, el resultado en la variable i habría sido $i = 6$, y que en el caso de que el valor inicial de x hubiese sido menor o igual a todos los elementos de la tabla, el valor en la variable i habría sido $i = 0$. En cualquier caso, se cumple que $0 \leq i \leq N + 1$.

En el ejercicio, nos indican cuál es el invariante a emplear. La nueva variable j incluida en el invariante también cumple $0 \leq j \leq N + 1$.

• *Cálculo de la condición de continuación.* $Inv \wedge \neg B \Rightarrow Post$. En este caso $\neg B : i = j$, luego $B : i \neq j$.

• *Cálculo de las instrucciones de inicio.* Buscamos las instrucciones de inicio más sencillas, aquellas que hacen que Inv sea trivialmente cierto al principio. Como en la precondición se cumple $x = X \wedge t = T$, entonces en Inv es cierto que $x = X \wedge t \in Perm(T)$. Podemos dar a i y a j valores que hagan que el dominio de k en $0 \leq k < i$ y en $j \leq k \leq N$ se haga vacío. Estos valores tienen que verificar que $0 \leq i \leq j \leq N + 1$.

$i := 'val\_ini\_i'$;
$j := 'val\_ini\_j'$;

Calcula dichos valores. Antes de seguir comprueba $Pre \Rightarrow ((Inv)^{j}_{val\_ini\_j})^{i}_{val\_ini\_i}$.

• *Cálculo de las instrucciones de avanzar.* Como $0 \leq i \leq j \leq N+1$ y la condición de continuación es $B : i \neq j$, entonces se cumple $0 \leq i < j \leq N+1$ cuando se entra a ejecutar las instrucciones del cuerpo del bucle. Para que se alcance la terminación $i = j$, la i debe crecer o la j decrecer. Así, se tienen varias posibilidades:

· avanzar con i, $i := i + 1$
· avanzar con j, $j := j - 1$
· avanzar con ambas variables a la vez, $i := i + 1$; $j := j - 1$.

• *Cálculo de las instrucciones de restablecer.* Vamos a suponer que avanzamos sólo con i, entonces escribimos los predicados

$Inv \wedge B : x = X \wedge\ t \in Perm(T) \wedge\ 0 \leq i < j \leq N+1\ \wedge\ (\forall k : 0 \leq k < i : t[k] < x) \wedge\ (\forall k : j \leq k \leq N : t[k] \geq x)$ (hemos puesto $i < j$ teniendo en cuenta que $B : i \neq j$)

$(Inv)^{i}_{i+1} : x = X \wedge\ t \in Perm(T) \wedge\ 0 \leq (i+1) \leq j \leq N+1\ \wedge\ (\forall k : 0 \leq k < (i+1) : t[k] < x) \wedge\ (\forall k : j \leq k \leq N : t[k] \geq x)$

A partir de $Inv \wedge B$ observamos que en $(Inv)^{i}_{i+1}$ todas las condiciones se cumplen excepto que $(\forall k : 0 \leq k < i : t[k] < x)$ no tiene por qué implicar $(\forall k : 0 \leq k < (i+1) : t[k] < x)$, más en detalle $(\forall k : 0 \leq k < (i+1) : t[k] < x) \equiv (\forall k : 0 \leq k < i : t[k] < x) \wedge t[i] < x$.

Sólo podemos asegurar que $Inv \wedge B$ implica $(Inv)^{i}_{i+1}$ si antes se cumple $t[i] < x$, entonces $Inv \wedge B \wedge t[i] < x \Rightarrow (Inv)^{i}_{i+1}$.

En otras palabras podemos avanzar con $i := i + 1$ si se cumple la condición $t[i] < x$. Esto nos lleva a una alternativa en el cuerpo de las instrucciones que forman el bucle.

Al analizar el caso $j := j - 1$ (hazlo como ejercicio) se llega a una conclusión parecida: podemos avanzar con $j := j - 1$ sólo si se cumple la condición $t[j-1] \geq x$.

Entonces tenemos cuatro casos posibles que hay que analizar formando una alternativa exclusiva:

si $t[i] < x \wedge t[j-1] \geq x \rightarrow i := i + 1; j := j - 1$
[] $t[i] < x \wedge t[j-1] < x \rightarrow i := i + 1$
[] $t[i] \geq x \wedge t[j-1] \geq x \rightarrow j := j - 1$
[] $t[i] \geq x \wedge t[j-1] < x \rightarrow$ 'restablecer'; $i := i + 1; j := j - 1$
fsi

En los tres primeros casos sabemos cómo avanzar y además podemos comprobar que no es necesario ninguna instrucción de restablecer. Pero en el último caso hay que hacer algo para que realmente se cumpla

$\{Inv \wedge B \wedge t[i] \geq x \wedge t[j-1] < x\}$ 'restablecer' $\{((Inv)^{j}_{j-1})^{i}_{i+1}\}$

Calcula estas instrucciones de restablecer. Basta con que te fijes que antes de 'restablecer' se cumple $t[i] \geq x \wedge t[j-1] < x$, que después de 'restablecer' se cumple $t[i] < x \wedge t[j-1] \geq x$ y que los valores en la tabla no pueden ser modificados ya que $t \in Perm(T)$.

• *Cálculo de la función de cota.* En este caso basta con proponer $cota = j - i$. Debes comprobar que

a) $Inv \wedge B \Rightarrow j - i > 0$
b) $\{Inv \wedge B \wedge j - i = T\}$ 'restablecer'; 'avanzar' $\{j - i < T\}$

en todas las posibilidades de las instrucciones de avanzar.

• *Diseño del algoritmo.* Completa el diseño del algoritmo siguiendo los cálculos anteriores.

Ejercicio 2.B. Codificación en C

Escribe un programa en C, de nombre `particion`, tal que codifique el algoritmo del apartado anterior mediante una acción con nombre `particion_tabla` que devuelva la posición que crea la partición y la tabla reorganizada. El programa debe pedir al usuario valores reales para crear una tabla de $N+1$ elementos (con $N = 9$), un valor para x y debe mostrar por pantalla los valores de tabla original y los valores de las dos particiones que se crean según el algoritmo codificado por la función anterior, cada conjunto de valores en una misma línea.

Sesión 7

Diseño de algoritmos iterativos: Tablas III

El objetivo de esta práctica es la realización de dos algoritmos sobre tablas clásicos. Un intento de obtener las soluciones sin un método adecuado produce algoritmos difíciles de entender y en muchas ocasiones incorrectos. En la construcción de algoritmos iterativos como los de esta sesión, se muestran claramente las ventajas del método de derivación a partir de invariantes propuesto en la asignatura.

7.1. Actividades a realizar

Para profundizar en la derivación de algoritmos iterativos que manejan tablas vamos a derivar los siguientes algoritmos y codificarlos como acciones y funciones en programas que hacen uso de ellos. Una vez más, para los distintos ejercicios propuestos, N es una constante entera positiva.

Ejercicio 1. Máximo de sumas de segmentos

Sea N una constate positiva. Diseñe un algoritmo basado en **un único bucle** que resuelva el siguiente problema

t: **tabla** [0..N] **de** *entero*;
r: *entero*;
$\{Pre : t = T\}$
segmento_máximo
$$\{Post : t = T \wedge\ r = \max\left\{\sum_{i \le k \le j} t[k] : 0 \le i \le j \le N\right\}\}$$

Ejercicio 1.A. Derivación del algoritmo

Antes de comenzar planteamos un ejemplo para comprender la especificación del problema. Tenemos una tabla de enteros, por ejemplo, $(-1, 2, 3, -2, 0)$. El índice de esta tabla va de 0 a N con N=4 (5 elementos). Según la postcondición, en la variable r tenemos que obtener el máximo del conjunto $\left\{\sum_{i \le k \le j} t[k] : 0 \le i \le j \le N\right\}$. Por ejemplo, si $i = 0$ y $j = 3$ obtenemos el valor de la suma $\sum_{0 \le k \le 3} t[k] = 2$; si $i = 1$ y $j = 2$ obtenemos el valor $\sum_{1 \le k \le 2} t[k] = 5$; y si $i = 2$ y $j = 4$, el valor es $\sum_{2 \le k \le 4} t[k] = 1$. En este ejemplo,

está claro que el valor de r se corresponde con el valor $\sum_{1\leq k\leq 2} t[k] = 5$, ya que este es el segmento de la tabla ($i = 1$ y $j = 2$) que tiene la máxima suma posible.

En general, para un N dado, en el conjunto $\left\{\sum_{i\leq k\leq j} t[k] : 0 \leq i \leq j \leq N\right\}$ podemos tener en total hasta $\frac{(N+1)(N+2)}{2}$ valores diferentes, ya que el recorrido de las variables i y j es $0 \leq i \leq j \leq N$. Así para $i = 0$, $0 \leq j \leq N$, para $i = 1$, $1 \leq j \leq N$, hasta que para $i = N$, $N \leq j \leq N$. Por tanto tenemos la serie $(N + 1) + N + (N - 1) + ... + 1$ cuyo valor es el dado anteriormente.

Como el problema planteado es un problema de recorrido, se puede proponer una solución (poco meditada) de la siguiente manera

```
algoritmo segmento_maximo;
t: tabla [0..N] de entero;
r: entero;

var
    a, b: 0..N;
    z: entero
fvar
    r:= 0;
    para a := 0 hasta N hacer
        z:= 0;
        para b := a hasta N hacer
            z:= z + t[b];
            r:= max(r,z)
        fpara
    fpara

falgoritmo
```

Esta solución es correcta sólamente en el caso en el que los valores de las sumas de los segmentos de la tabla fuesen mayores o iguales que 0. La pregunta es si podemos o no resolver este problema con un único bucle, como pide el ejercicio, en el caso general, esto es con cualesquiera valores en los elementos de la tabla.

La solución no es trivial y tiene algunos detalles técnicos que deben resolverse adecuadamente. Primero, dado que es un problema de recorrido podemos plantearnos utilizar directamente las ideas del esquema de recorrido. En este caso, la postcondición la podemos escribir como

$$Post : t = T \wedge\ r = \max\left\{\sum_{i\leq k\leq j} t[k] : 0 \leq i \leq j < N + 1\right\}$$

La idea de utilizar la constante $N + 1$ (en vez de N) se introdujo cuando se analizó el esquema de recorrido[1]. Esto nos permite obtener instrucciones de inicio más sencillas. Si sustituimos $N+1$ por una nueva variable x, la expresión $\max\left\{\sum_{i\leq k\leq j} t[k] : 0 \leq i \leq j < x\right\}$ tiene un dominio vacío cuando $x = 0$. El problema es que el máximo sobre un domino vacío tiene formalmente como elemento neutro $-\infty$, pero no es posible hacer la asignación inicial $r := -\infty$. En los apuntes de clase se indica que es conveniente trabajar con el cuantificador

[1]El esquema de recorrido en tablas es bśicamente construir el invariante por sustitución de constantes por variables en el cuantificador. En un recorrido estándar se pide que el cuantificador valga el elemento neutro de su operación al inicio del bucle.

máximo con dominos no vacíos. En conclusión, a pesar de tratarse de un problema de recorrido no podemos utilizar la constante $N + 1$ en la sustitución. Mantendremos la constante N.

• *Propuesta de invariante.* Trabajaremos con la postcondición original, utilizando para obtener el invariante el método de sustitución de constantes por variables. En este caso, para la postcondición dada, el invariante queda

$$Inv : t = T \wedge\ r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x\right\} \wedge 0 \le x \le N$$

La constante N ha sido sustituida por la nueva variable x.

• *Cálculo de la condición de continuación.* Se debe cumplir $Inv \wedge \neg B \Rightarrow Post$. En este caso $\neg B : x = N$, luego $B : x \neq N$. Como en el Inv se cumple $0 \le x \le N$ podemos también escribir $B : x < N$.

• *Cálculo de las instrucciones de inicio.* Como x debe terminar siendo N, por su rango $0 \le x \le N$ puede comenzar en $x = 0$. En este caso, si $x = 0$ entonces

$$r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le 0\right\} = \max\left\{\sum_{0\le k\le 0} t[k]\right\} = t[0]$$

Las instrucciones de inicio son x:= 0; r:= t[0].

• *Cálculo de las instrucciones de avanzar.* Como x comienza en 0, termina en N, la propuesta de avanzar es $x := x + 1$.

• *Cálculo de las instrucciones de restablecer.* $\{Inv \wedge B\}$ 'restablecer' $\{(Inv)^x_{x+1}\}$

$$Inv \wedge B : t = T \wedge\ r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x\right\} \wedge 0 \le x \le N \wedge x < N$$

$$(Inv)^x_{x+1} : t = T \wedge\ r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x+1\right\} \wedge 0 \le x+1 \le N$$

Como se cumple $t = T \wedge 0 \le x \le N \wedge x < N$ en $Inv \wedge B$, también se cumple $t = T \wedge 0 \le x+1 \le N$ en $(Inv)^x_{x+1}$. La variable r cambia su valor y hay que relacionar de alguna forma el nuevo valor de r dado por la expresión $\max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x+1\right\}$ con el valor de r anterior que es $\max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x\right\}$. Separando el dominio $0 \le i \le j \le x+1$ en dos dominios disjuntos $0 \le i \le j \le x$ y $0 \le i \le x+1$, cuando j toma su último valor $j = x + 1$, entonces se cumple

$$\max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x+1\right\} =$$
$$\max\left\{\max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x\right\}, \max\left\{\sum_{i\le k\le x+1} t[k] : 0 \le i \le x+1\right\}\right\}.$$

Entonces, el nuevo valor para r es el máximo entre su valor anterior y el máximo de los valores en el conjunto $\{\sum_{i\le k\le x+1} t[k] : 0 \le i \le x+1\}$.

$$r := \max\left\{r, \max\left\{\sum_{i\le k\le x+1} t[k] : 0 \le i \le x+1\right\}\right\}$$

Como nos piden resolver el problema con sólo un bucle, podemos intentar reforzar el invariante porque no podemos programar $\max\left\{\sum_{i\le k\le x+1} t[k] : 0 \le i \le x+1\right\}$, pero hay que tener mucho cuidado. Si escribes el nuevo invariante utilizando la expresión anterior en una nueva variable entera z

$$t = T \wedge\ r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x\right\} \wedge 0 \le x \le N \wedge$$
$$\wedge\ z = \max\left\{\sum_{i\le k\le x+1} t[k] : 0 \le i \le x+1\right\}$$

Este invariante no es un predicado válido porque como $0 \le x \le N$, cuando $x = N$ en la expresión asociada a z se puede acceder al elemento $t[N+1]$, ¡que no existe en la declaración!. Así que podemos intentar la solución con el siguiente invariante

$$Inv' : t = T \wedge\ r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le x\right\} \wedge 0 \le x \le N \wedge$$
$$z = \max\left\{\sum_{i\le k\le x} t[k] : 0 \le i \le x\right\}$$

donde la expresión para la variable z está definida con respecto a x en vez de con respecto a $x+1$.

Según lo que hemos visto, la estructura de la solución es la siguiente

algoritmo segmento_maximo;
t: **tabla** [0..N] **de** *entero*;
r: *entero*;
$\{Pre \equiv t = T\}$
var
    x: 0..N;
    z: *entero*
fvar
    x:= 0;
    r:= t[0];
    z:= .....;
$\{Inv'....\}$ $\{cota...\}$
    mientras x < N **hacer**
        z:= .....;
        r:= max(r,z);
        x:= x+1
    fmientras
$\{Post \equiv t = T \wedge\ r = \max\left\{\sum_{i\le k\le j} t[k] : 0 \le i \le j \le N\right\}\}$
falgoritmo

Trabajo pendiente:

1. Tienes que calcular el valor inicial para z cuando $x = 0$
   $z = \max\left\{\sum_{i\le k\le x} t[k] : 0 \le i \le x\right\} = \max\left\{\sum_{i\le k\le 0} t[k] : 0 \le i \le 0\right\} = ...$
   luego $z := ....$

2. Tienes que restablecer el nuevo valor de z que aparece en $(Inv')^{x}_{x+1}$, a partir de su valor anterior en el $Inv' \wedge B$ antes de utilizarlo con r en la instrucción $r := max(r, z)$.

   $Inv' \wedge B \equiv ....z = \max\left\{\sum_{i\le k\le x} t[k] : 0 \le i \le x\right\}$
   'restablecer z'
   $....z = \max\left\{\sum_{i\le k\le x+1} t[k] : 0 \le i \le x+1\right\}$

• *Cálculo de la función de cota.* En este caso, basta con proponer $cota = N - x$. Debes comprobar que

a) $Inv \wedge B \Rightarrow N - x > 0$
b) $\{Inv \wedge B \wedge N - x = T\}$ 'restablecer'; x:= x+1 $\{N - x < T\}$

Finalmente, calcula el coste computacional del algoritmo obtenido.

Ejercicio 1.B. Codificación en C

Escribe un programa en C, de nombre `segmentos`, tal que codifique el algoritmo anterior como una función `segmento_maximo`. El programa solicitará al usuario valores enteros para crear dos tablas de $N+1$ elementos (con $N = 6$) y calculará el máximo de las sumas de los segmentos de cada una. Finalmente, el programa tendrá que mostrar por pantalla las sumas máximas obtenidas para cada tabla junto con un mensaje de la forma 'La tabla X es la que contiene la suma mayor' que indique cuál de las dos tablas tiene la suma máxima mayor.

Ejercicio 2. Cuenta parejas con distinto signo

Diseñe un algoritmo basado en **un único bucle** que resuelva el siguiente problema

t: **tabla** [0..N] **de** *entero*;
r: *entero*;
$\{Pre : t = T\}$
    cuenta_parejas
$\{Post : t = T \wedge\ r = \#(i,j) : 1 \leq i < j \leq N : (t[i] \leq 0 \wedge\ t[j] \geq 0)\}$

Escriba un programa en C, de nombre `contador`, tal que codifique el algoritmo anterior como una función `cuenta_parejas`. El programa debe pedir al usuario valores enteros para crear una tabla de $N+1$ elementos (con $N = 9$) y calcular el número de parejas utilizando la función anterior. Finalmente, mostrará por pantalla el contenido de la tabla en una sola línea y el número de parejas encontradas en la tabla.

Sesión 8

Recursividad

El objeto de esta práctica es mostrar las posibilidades que ofrece C para codificar algoritmos en forma de acciones y funciones recursivas.

8.1. C y recursividad

El lenguaje de programación C es imperativo, en consecuencia, la estructura de control por excelencia es la repetición. No obstante, a diferencia de otros lenguajes de estas características también ofrece la posibilidad de programar recursivamente (existen lenguajes como LISP en los que sólo se programa recursivamente). La manera de codificar funciones o acciones recursivas no presenta diferencias reseñables frente al pseudocódigo, por lo que no destacamos ninguna particularidad en este sentido. Sólo un comentario sobre la ejecución: para comprender el mecanismo de ejecución de una función o acción recursiva, es preciso imaginar que cada llamada a una acción recursiva no es más que una llamada a una copia de sí misma. Cada una de estas copias se comporta como una acción independiente, ya que las variables locales y los parámetros correspondientes existen de modo diferenciado en cada llamada.

8.2. Actividades a realizar

Para entrenarnos en el uso de la recursividad vamos a codificar algunos algoritmos. En cada caso, además del resultado pedido, se deberá proporcionar el número de llamadas recursivas que ha generado el algoritmo.

Ejercicio 1. División de dos números naturales

Para simplificar la presentación de las funciones recursivas permitimos el uso de las asignaciones múltiples y también vamos a admitir que una función pueda devolver varios resultados[1]. A la hora de codificar en C funciones con más de un resultado, éstas se deberán convertir en acciones en el algoritmo correspondiente y serán codificadas en C mediante funciones que representan dichas acciones (repasar la Sesión 4). Vamos a presentar un ejemplo utilizando el problema de la división entera de números enteros positivos. Además el ejemplo tiene interés porque se muestra que es posible obtener distintas soluciones

[1] Repasa las notas sobre las funciones dadas en la sesión 4

mediante funciones recursivas con distinta eficiencia. Tenemos la siguiente declaración de la función división.

función division(a,b:*entero*) **dev** q, r: *entero*;
$\{P : a \geq 0 \wedge b > 0\}$
$\{Q : a = q \cdot b + r \wedge 0 \leq r < b\}$
ffunción

Esta función tiene que resolver el problema (P, Q) para los datos $\langle a, b\rangle$ (a es el dividendo y b el divisor). Imagina que ya has resuelto este mismo problema (P, Q) pero con los datos $\langle a - b, b\rangle$ cuando $a >= b$. En este caso, se cumple

$$a - b = q' \cdot b + r' \wedge 0 \leq r' < b$$

Para obtener a partir de $\langle q', r'\rangle$ la solución $\langle q, r\rangle$ para los datos originales $\langle a, b\rangle$ basta observar que

$$a = q' \cdot b + r' + b = (q' + 1) \cdot b + r' \text{ y } 0 \leq r' < b$$

luego $\langle q, r\rangle = \langle q' + 1, r'\rangle$

Cuando se cumple $a < b$ ya sabemos que la solución es $\langle q, r\rangle = \langle 0, a\rangle$.

El análisis anterior nos permite escribir la forma recursiva de la operación de división

Sea $\langle q, r\rangle =$ division(a, b):
- $\langle q, r\rangle = \langle 0, a\rangle$ si $a < b$ (*caso trivial*)
- $\langle q', r'\rangle =$ division$(a - b, b)$ si $a \geq b$ (*llamada recursiva*)
  $\langle q, r\rangle = \langle q' + 1, r'\rangle$

La codificación de esta función en el lenguaje algorítmico es la siguiente

función division(a,b:*entero*) **dev** q,r:*entero*;
$\{P : a \geq 0 \wedge b > 0\}$
var q', r':*entero* **fvar**
  si a < b $\longrightarrow$ ⟨q,r⟩:= ⟨0,a⟩
  [] a ≥ b $\longrightarrow$ ⟨q',r'⟩:= division(a-b,b);
  ⟨q,r⟩:=⟨q'+1,r'⟩
  fsi;
$\{Q : a = q \cdot b + r \wedge 0 \leq r < b\}$
  dev q, r
ffunción

Esta función termina porque si comienza con $a \geq b$ y en cada nueva llamada los nuevos parámetros son $\langle a - b, b\rangle$ se llegará a una llamada k-ésima en que se cumpla $a - kb < b$. En esta llamada se resuelve el caso trivial y esta última llamada conduce a que todas las llamadas anteriores terminen su ejecución. Observa que al final $a - q \cdot b = r$ con $0 \leq r < b$. Luego el número de llamadas recursivas realizadas es a div b. Podemos concluir que el tiempo de ejecución de esta solución es proporcional a a div b.

Otra forma de obtener el tiempo de ejecución consiste en relacionarlo con el 'tamaño del problema' y analizar cómo actúa la función recursiva. En este caso sea $n = a$ div b el

tamaño del problema, y sea $T(n)$ el tiempo de ejecución. Según el código de la función $T(0) = K_1$ (el tiempo del caso trivial) y $T(n) = T(n-1) + K_2$ (el tiempo de la otra rama alternativa: tiempo de la llamada recursiva $T(n-1)$[2] más K_2 el tiempo de ejecución de las asignaciones). Al resolver esta recurrencia: $T(n) = T(n-1) + K_2 = T(n-2) + 2K_2 = ... = nK_2 + K_1$. El tiempo de ejecución es lineal, proporcional a n.

Como el tiempo de ejecución es proporcional a a div b está claro que cuanto mayor sea el divisor menor será el tiempo de ejecución. Esta idea se puede aplicar para intentar encontrar una solución eficiente al problema de la división. Considera que has obtenido $\langle q', r' \rangle =$ division$(a, 2b)$ cuando $a \geq b$. Entonces se cumple

$$a = q' \cdot 2b + r' \wedge 0 \leq r' < 2b$$

Para obtener a partir de $\langle q', r' \rangle$ la solución $\langle q, r \rangle$ para los datos originales $\langle a, b \rangle$ basta observar que si $r' < b$ entonces $q = 2q'$ y $r = r'$. Pero si $b \leq r' < 2b$ podemos hacer lo siguiente

$$a = q' \cdot 2b + b + r' - b \wedge 0 \leq r' - b < b$$

En este caso obtenemos $q = 2q' + 1$ y $r = r' - b$. El análisis anterior nos permite escribir otra forma recursiva de la operación división

Sea $\langle q, r \rangle =$ division(a, b):
- $\langle q, r \rangle = \langle 0, a \rangle$ si $a < b$ (*caso trivial*)
- Sea $\langle q', r' \rangle =$ division$(a, 2b)$ si $a \geq b$ (*llamada recursiva*)
  $\langle q, r \rangle = \langle 2q', r' \rangle$ si $r' < b$
  $\langle q, r \rangle = \langle 2q' + 1, r' - b \rangle$ si $r' \geq b$

La codificación de esta función en el lenguaje algorítmico es la siguiente

```
función division(a,b:entero)dev q,r:entero;
{P : a ≥ 0 ∧ b > 0}
var q', r':entero fvar
    si   a < b ⟶ ⟨q,r⟩:= ⟨0,a⟩
      [] a ≥ b ⟶ ⟨q',r'⟩:= division(a,2*b);
                 si   r' ≥ b ⟶ ⟨q,r⟩:= ⟨2*q'+1,r'-b⟩
                   [] r' < b ⟶ ⟨q,r⟩:= ⟨2*q',r'⟩
                 fsi
    fsi;
{Q : a = q · b + r ∧ 0 ≤ r < b}
    dev q, r
ffunción
```

Calcula el tiempo de ejecución de la función anterior, $T(n)$, donde el tamaño del problema es $n = a$ div b. Codifica en lenguaje C un programa de nombre `divide_eficiente` que implemente la función anterior mediante una acción `division`, para obtener el cociente y resto de la división de dos enteros positivos. Realiza diferentes pruebas y comprueba los resultados obtenidos.

[2] $n - 1 = (a - b)$ div b

Ejercicio 2. Las Torres de Hanoi

Se dice que después de crear el mundo Dios puso en la tierra tres barras hechas de diamantes y 64 anillos de oro. Estos anillos son todos ellos de distintos tamaños. En la creación se dispusieron los anillos en la primera barra por orden de tamaños, con el mayor en la parte inferior y el menor en la parte superior. También creó Dios un monasterio junto a las barras. La tarea de los monjes durante su vida es trasladar todos los anillos a la segunda barra. La única operación permitida es trasladar un solo anillo de una barra a otra de tal manera que nunca se coloque un anillo encima de otro que sea menor. Cuando los monjes hayan finalizado su tarea, dice la leyenda, acabará el mundo. Probablemente sea la profecía que ofrece más consuelo con respecto al fin del mundo, porque si los monjes se las arreglasen para mover un anillo por segundo, trabajando día y noche sin descansar ni cometer errores, ¡su trabajo no habría terminado 500.000 millones de años después de que empezaran!. ¡Son más de 25 veces la edad estimada del Universo!.

Una solución al problema para el caso general de una torre de n piezas puede darse en la forma siguiente: pasar una torre de n piezas de la barra A a la barra B utilizando la barra C de forma auxiliar, consiste en (1) pasar una torre de n-1 piezas de la barra A a la C (utilizando la B de auxiliar), (2) pasar la pieza n de la barra A a la B, y (3) pasar una torre de n-1 piezas de la barra C a la B (utilizando la A de auxiliar). En lenguaje algoritmico

```
acción colocar(ent n: entero; ent i, j, k: char);
{Pre : n > 0 ∧ (i, j, k) ∈ perm('A','B','C')}
    si   n = 1 ⟶ 'mover disco n del poste i al j'
      [] n > 1 ⟶ colocar(n-1, i, k, j);
                 'mover disco n del poste i al j';
                 colocar(n-1, k, j, i)
    fsi
{Post : pasada la torre de n piezas del poste i al poste j con k poste auxiliar}
facción;
```

Calcula primero el tiempo de ejecución de la acción anterior, $T(n)$, donde el tamaño del problema n es el número de piezas a mover. Escribe en C un programa de nombre `hanoi` que implemente la acción anterior en forma de acción con nombre `colocar`, para obtener la secuencia de movimientos que conducen a las n piezas de la barra A a la barra B. Prueba el programa con valores pequeños de n.

Ejercicio 3. Búsqueda dicotómica

La formulación recursiva de la búsqueda dicotómica es bastante sencilla. Dada una tabla t de N elementos, con posiciones $1..N$, cuyos elementos están ordenados de manera creciente (puedes suponer que son números enteros) y dado un número x, se trata de indicar en qué posición habría que introducirlo en la tabla para mantener la propiedad de ordenación de la misma. La idea es comparar x con el valor de la tabla que ocupa la 'posición media' $t[m]$. Si es mayor o igual, $x \geq t[m]$, buscamos de nuevo a partir de la posición media en la parte derecha de la tabla, y si es menor, $x < t[m]$, buscamos de nuevo a partir de la posición media en la parte izquierda de la tabla. El proceso continúa hasta que encontramos la condición $t[p] \leq x < t[p+1]$ siendo p una posición que cumple

$0 \leq p < N + 1$. En esta formulación del problema se admite que la tabla t contiene dos posiciones ficticias, $t[0]$ y $t[N + 1]$, cuyos valores son $t[0] = -\infty$ y $t[N + 1] = +\infty$. Si el resultado es $p = 0$ simplemente significa que x es menor que todos los elementos de la tabla. Si $p = N$, x es mayor o igual al último elemento de la tabla. Esta forma de especificar la búsqueda dicotómica permite expresar la postcondición de una manera simple. No obstante, para mantener la corrección del algoritmo, es necesario que nunca se acceda a las posiciones 0 y $N + 1$, que no existen.

```
función buscar(t: tabla [1..N] de entero; i, j, x: entero)dev p: entero;
{Pre : 0 ≤ i < j ≤ N + 1 ∧ t[i] ≤ x < t[j]}
var m:entero fvar
    si   j = i+1 ⟶ p:=i
      [] j ≠ i+1 ⟶ m := (i+j) div 2;
                   si   x ≥ t[m] ⟶ p:= buscar(t, m, j, x)
                     [] x < t[m] ⟶ p:= buscar(t, i, m, x)
                   fsi
    fsi;
{Post : 0 ≤ p < N + 1 ∧ t[p] ≤ x < t[p + 1]}
    dev p
ffunción
```

a) Demuestra que la función anterior es correcta.

b) La primera vez que se llama a la función para obtener el resultado sobre toda la tabla t, el valor inicial de las variables i y j tienen que ser 0 y $N + 1$ respectivamente. Suponer $N = 10$. Escribe en C un programa de nombre `busqueda_dicotomica` que implemente la función `buscar`. El programa construye primero una tabla con valores ordenados de menor a mayor, luego pide un número y a continuación indica si ese número está en la tabla o no está en la tabla. En cualquier caso, finalmente escribe un mensaje con las posiciones entre las que debería encontrarse el número solicitado en la tabla.

Ejercicio 4. Suma de divisores de un número natural

Construye una función recursiva para la especificación siguiente

```
función suma_divisores(n: entero) dev s: entero;
{Pre : n ≥ 1}
{Post : s = Σ_{1≤d≤n, d|n} d}
```

Ejercicio 4.A. Diseño de la solución

Este ejercicio se realizó de manera iterativa en la Sesión 2 de prácticas. Aquí vamos a realizar una versión recursiva, lo que permitirá ver las diferencias entre los métodos de diseño iterativos y recursivos presentados en la asignatura. Para resolver este problema utilizaremos el método de inmersión de especificaciones por reforzamiento de la precondición. La especificación de la función inmersora tiene la misma postcondición que la función original. Deberemos calcular nuevos parámetros de entrada y la nueva precondición con el método mencionado, tal como se presenta a continuación.

Especificación de la función inmersora

función i_suma_divisores(n: *entero*,....) **dev** s: *entero*;
$\{P'\}$
$\{Q' : s = \sum_{\substack{1 \leq d \leq n \\ d|n}} d\}$
ffunción

Para calcular la precondición P':

1. Cambiamos la variable de salida s de la postcondición por una nueva variable p, que será parámetro de entrada en la función inmersora, y añadimos la precondición original.

$$p = \sum_{\substack{1 \leq d \leq n \\ d|n}} d \wedge n \geq 1$$

2. Debilitamos este predicado mediante las técnicas conocidas: sustitución de constantes por variables, expresiones por variables, eliminación de una conjunción. En este caso sustituimos la variable n por una nueva variable k e indicamos el rango de valores para k.

$$p = \sum_{\substack{1 \leq d \leq k \\ d|n}} d \wedge n \geq 1 \wedge 1 \leq k \leq n \equiv p = \sum_{\substack{1 \leq d \leq k \\ d|n}} d \wedge 1 \leq k \leq n$$

La variable k también será un parámetro de entrada de la función inmersora. Elegimos este predicado como la precondición de la función inmersora y escribimos toda la estructura de la solución.

$$P' : p = \sum_{\substack{1 \leq d \leq k \\ d|n}} d \wedge 1 \leq k \leq n$$

función suma_divisores(n: *entero*) **dev** s: *entero*;
$\{Pre : n \geq 1\}$
    función i_suma_divisores(n, p, k:*entero*) **dev** s: *entero*;
    $\{P' : p = \sum_{\substack{1 \leq d \leq k \\ d|n}} d \wedge 1 \leq k \leq n\}$
        si 'caso trivial' $\longrightarrow$ s:= 'exp_caso_trivial'
        [] 'caso recursivo' $\longrightarrow$ s:= i_suma_divisores(n, 'exp_p', 'exp_k')
        fsi;
    $\{Q'$: $s = \sum_{\substack{1 \leq d \leq n \\ d|n}} d\}$
        dev s
    ffunción
    s:= i_suma_divisores(n, 'val_ini_p', 'val_ini_k');
$\{Post$: $s = \sum_{\substack{1 \leq d \leq n \\ d|n}} d\}$
    dev s
ffunción

3. La conjunción $1 \leq k \leq n$ nos ayuda a definir las partes que no están completadas en el diseño.

Caso trivial. Si $k = n$ y se cumple la precodición P', tenemos que obtener, por el caso trivial de la alternativa, la siguiente implicación

$$P' \wedge k = n \Rightarrow (Q')^s_{exp\_caso\_trivial}$$
$$p = \sum_{\substack{1 \leq d \leq n \\ d|n}} d \wedge 1 \leq n \leq n \Rightarrow exp\_caso\_trivial = \sum_{\substack{1 \leq d \leq n \\ d|n}} d$$

Por tanto, p es justamente la $exp\_caso\_trivial$.

Caso recursivo. Si $k \neq n$ y se cumple la precondición P', tenemos que obtener por el caso recursivo de la alternativa, la siguiente implicación

$$P' \wedge k \neq n \Rightarrow P'[n \leftarrow n, p \leftarrow exp\_p, k \leftarrow exp\_k]$$

Es decir, se debe verificar la precondición de la llamada a la función i_suma_divisores(n, 'exp_p', 'exp_k') con los nuevos argumentos dados por sus correspondientes expresiones. La sustitución $P'[n \leftarrow n, p \leftarrow exp\_p, k \leftarrow exp\_k]$ es una sustitución múltiple (las sustituciones de las variables por las nuevas expresiones se hacen a la vez).

Una de las ventajas de este método es que nos permite razonar 'casi' como si lo hiciésemos de manera iterativa. Primero identificamos cómo avanzar al caso trivial y luego ajustamos el resto de expresiones para que se cumpla la precondición. En este caso k tiene que alcanzar el valor n para que se cumpla la condición del caso trivial. Podemos pensar en 'avanzar' con exp_k siendo $k + 1$. Escribimos ahora los dos predicados de la implicación anterior teniendo en cuenta la forma de alcanzar el caso trivial.

$$P' \wedge k \neq n \equiv p = \sum_{\substack{1 \leq d \leq k \\ d|n}} d \wedge 1 \leq k < n$$
$$P'[n \leftarrow n, p \leftarrow exp\_p, k \leftarrow k + 1] \equiv exp\_p = \sum_{\substack{1 \leq d \leq k+1 \\ d|n}} d \wedge 1 \leq k + 1 \leq n$$

Vemos que se cumple $1 \leq k < n \Rightarrow 1 \leq k + 1 \leq n$. Por otra parte,

$$exp\_p = \sum_{\substack{1 \leq d \leq k+1 \\ d|n}} d = \sum_{\substack{1 \leq d \leq k \\ d|n}} d + (k+1), \text{ si } n \text{ div } (k+1) = 0$$

o

$$exp\_p = \sum_{\substack{1 \leq d \leq k+1 \\ d|n}} d = \sum_{\substack{1 \leq d \leq k \\ d|n}} d, \text{ si } n \text{ div } (k+1) \neq 0.$$

Como en el antecedente $p = \sum_{\substack{1 \leq d \leq k \\ d|n}} d$ entonces

$$exp\_p = p + (k+1) \text{ si } n \text{ div } (k+1) = 0$$

o

$exp\_p = p$ si n div $(k+1) \neq 0$.

El caso recursivo es de nuevo una alternativa que diferencia la 'exp_p' que se debe utilizar en la llamada recursiva.

La primera llamada. En la primera llamada a la función inmersora debemos calcular los valores iniciales 'val_ini_p' y 'val_ini_k'. En la primera llamada se tiene que cumplir

$$Pre \Rightarrow P'[n \leftarrow n, p \leftarrow val\_ini\_p, k \leftarrow val\_ini\_k]$$

Como k puede comenzar en 1, probamos con $val\_ini\_k = 1$. Escribimos

$$P'[n \leftarrow n, p \leftarrow val\_ini\_p, k \leftarrow 1] \equiv val\_ini\_p = \sum_{\substack{1 \leq d \leq 1 \\ d|n}} d \wedge 1 \leq 1 \leq n$$

Por tanto, para que la implicación anterior sea cierta es necesario que val_ini_p sea 1.

La **solución** queda de la siguiente forma

```
función suma_divisores(n: entero) dev s: entero;
{Pre : n ≥ 1}
    función i_suma_divisores(n, p, k:entero)dev s: entero;
    {P' : p = Σ_{1≤d≤k, d|n} d ∧ 1 ≤ k ≤ n}
        si   k = n ⟶ s:= p
           [] k ≠ n ⟶
                si   n mod (k+1)= 0 ⟶ s:= i_suma_divisores(n, p + (k+1), (k+1))
                   [] n mod (k+1)≠ 0 ⟶ s:= i_suma_divisores(n, p, (k+1))
                fsi
        fsi;
    {Q': s = Σ_{1≤d≤n, d|n} d}
        dev s
    ffunción
    s:= i_suma_divisores(n, 1, 1);
{Post: s = Σ_{1≤d≤n, d|n} d}
    dev s
ffunción
```

Ejercicio 4.B. Codificación en C

Escribe un programa en C, de nombre `lista_suma_divisores`, que implemente la función anterior con el nombre `suma_divisores`. El programa solicita al usuario un número M mayor o igual a uno. En el caso de que $M < 1$ el programa escribe un mensaje de error. El programa escribe en pantalla la lista:

```
La suma de los divisores de 1 es ...
La suma de los divisores de 2 es ...
...
La suma de los divisores de M es ...
```

donde los puntos suspensivos indican el valor correspondiente al obtenido mediante la función que se ha codificado.

Ejercicio 5. Suma de factoriales

Construye una función recursiva para la especificación siguiente

función suma_de_factoriales(n: *entero*) **dev** s: *entero*;
$\{Pre : n \geq 0\}$
$\{Post\text{: } s = \sum_{0 \leq i \leq n} i!\}$

Utiliza el método de inmersión de funciones por reforzamiento de la precondición. Para ayudarte en la realización del ejercicio se propone la siguiente precondición de la función inmersora.

$$P' : p = \sum_{0 \leq i \leq k} i! \wedge 0 \leq k \leq n \wedge z = k!$$

Ejercicio 5.A. Diseño de la solución

El esquema de la solución es

función suma_de_factoriales(n: *entero*) **dev** s: *entero*;
$\{Pre : n \geq 0\}$
  función i_suma_de_factoriales(n, p, k, z:*entero*) **dev** s: *entero*;
  $\{P' : p = \sum_{0 \leq i \leq k} i! \wedge 0 \leq k \leq n \wedge z = k!\}$
    si 'caso trivial' $\longrightarrow$ s:= 'exp_caso_trivial'
    [] 'caso recursivo' $\longrightarrow$ s := i_suma_de_factoriales(n, 'exp_p', 'exp_k', 'exp_z')
    fsi;
  $\{Q'\text{: } s = \sum_{0 \leq i \leq n} i!\}$
    dev s
  ffunción
  s:= i_suma_de_factoriales(n, 'val_ini_p', 'val_ini_k', 'val_ini_z');
$\{Post\text{: } s = \sum_{0 \leq i \leq n} i!\}$
  dev s
ffunción

Calcula y completa las expresiones indicadas.

Ejercicio 3.B. Codificación en C

Escribe un programa en C, de nombre `lista_suma_factoriales`, que implemente la función `suma_de_factoriales`. El programa solicita al usuario un número M mayor o igual que cero. En el caso de que $M < 0$ el programa escribe un mensaje de error. El programa escribe en pantalla la lista:

```
La suma de los factoriales de 0 es ...
La suma de los factoriales de 1 es ...
...
La suma de los factoriales de M es ...
```

donde los puntos suspensivos indican el valor correspondiente al obtenido mediante la función que se ha codificado.