Práctica de Programación II

Curso 1999 – 2000

I.T.I. SISTEMAS

ELCHE

José Manuel Carrillo Paños

ÍNDICE DE LA PRÁCTICA

2 **Documentación que hay que entregar** 3
2.1 Indice de la documentación necesaria 3
2.2 Comentarios a la documentación 4
3 **Enunciado de la práctica** 8
3.1 Enunciado: texto del problema 8
3.2 Comentarios al enunciado 8
3.3 Cuestiones sobre el enunciado 8
4 **Especificación** 8
4.1 Consideraciones sobre la especificación 8
4.2 Cuestiones sobre la especificación 9
4.3 Especificación formal de la función (trabajo del alumno) 9
5 **Diseño recursivo no final** 10
5.1 Especificación de la función inmersora y llamada inicial 10
5.2 Calculo de la inmersión y llamada inicial (trabajo del alumno) 10
5.3 Análisis por casos. 11
5.4 Análisis por casos de la función inmersora (trabajo del alumno) 11
5.5 Composición algorítmica (trabajo del alumno) 12
5.6 Verificación formal de la corrección. 13
5.6.1 Completitud de la alternativa (trabajo del alumno) 13
5.6.2 Satisfacción de la precondición para la llamada interna (trabajo del alumno) 13
5.6.3 Base de la inducción (trabajo del alumno) 14
5.6.4 Paso de inducción (trabajo del alumno) 14
5.6.5 Elección de una estructura de preorden bien fundado (trabajo del alumno). 14
5.6.6 Demostración del decrecimiento de los datos (trabajo del alumno). 14
5.7 Estudio del coste del algoritmo (trabajo del alumno). 15
6 **Transformación a recursivo final: Desplegado-plegado** 16
6.1 Arbol sintáctico de la función recursiva. (trabajo del alumno) 16
6.2 Substituciones aplicadas. (trabajo del alumno) 16
6.3 Desplegado y plegado. Llamada inicial. (trabajo del alumno) 16
6.4 Código de la función recursiva final. (trabajo del alumno) 18
7 **Diseño iterativo** 18
7.2 Derivación del invariante partiendo de la postcondición (trabajo del alumno) 18
7.3 Instrucción avanzar del bucle (trabajo del alumno) 19
7.4 Restablecimiento del invariante (trabajo del alumno) 20
7.5 Código del algoritmo iterativo (trabajo del alumno) 20
7.6 Estudio del coste. (trabajo del alumno) 21
7.7 Cuestiones sobre la función iterativa 22
8 **Implementación en Modula-2** 23
8.1 Código Modula-2 y juego de pruebas (trabajo del alumno). 23

2 Documentación que hay que entregar

2.1 Indice de la documentación necesaria

A modo de resumen, listamos a continuación la documentación que se debe entregar **al tutor, antes de la fecha que éste disponga**, para que la práctica se considere completa:

1. **Hoja de lectura óptica con <u>todos los datos personales cumplimentados.</u>**

2. **Enunciado del problema.**

3. **Especificación del algoritmo**.

4. **Algoritmo recursivo no final:**

   - Especificación de la función inmersora y llamada inicial.
   - Análisis por casos.
   - Composición algorítmica.
   - Verificación formal de la corrección.
   - Estudio del coste asintótico del algoritmo recursivo no final.

5. **Algoritmo recursivo final:**

   - árbol sintáctico de la función recursiva.
   - Substituciones aplicadas según la heurística de Kodratoff
   - Función desplegada y su equivalente plegada. Llamada inicial.
   - Código de la función recursiva final.

6. **Algoritmo iterativo:**

   - Derivación formal del algoritmo iterativo.
   - Código del algoritmo iterativo resultante por derivación directa.
   - Estudio del coste asintótico del algoritmo iterativo.

7. **Implementación:**

   Código Modula-2: Se entregarán los listados de los módulos impresos y un disco conteniendo los mismos listados junto con una versión compilada del programa.

8. **Juego de pruebas:** para cada prueba han de listarse:

   - Entrada de la prueba y salida prevista.
   - Salida obtenida.

9. **Estudio empírico del coste:**

   - Gráfica comparativa de los costes empíricos obtenidos para las ejecuciones del programa con diferentes juegos de datos.
   - Comentario de los datos así como de los resultados obtenidos.

2.2 Comentarios a la documentación

En este apartado se explican, más detalladamente, los puntos antes listados, se aclaran algunas cuestiones de formato y notación, y se hacen algunas indicaciones y precisiones sobre la documentación que se debe entregar.

1. **Hoja de lectura óptica (HLO) con todos los datos personales cumplimentados.**

   Las HLO las repartirán los Tutores en la primera sesión obligatoria de asistencia a las Prácticas. En ella hay que rellenar, de forma reconocible por la lectora óptica, los siguientes campos:

- **D.N.I**.
- **Código de carrera**: *I.T.Gestión*: 41; *I.T.Sistemas*: 40.
- **Código de Asignatura**: 108.
- **Convocatoria**: Junio. **Semana**: Primera.
- **Tipo de Examen**: A.

Todos los alumnos deberán cumplimentar los campos **Convocatoria, Semana y tipo de examen** con los datos antedichos, independientemente de cuando decidan examinarse.

La práctica no se considerará completa, y, por tanto, **el examen del alumno no se corregirá** si todos los datos consignados en la hoja de lectura óptica de la práctica no son correctos, o sea,

> si el programa de gestión de calificaciones rechaza la hoja de prácticas correspondiente al alumno, ya sea por datos incompletos o incorrectamente consignados, **el examen de dicho alumno no será calificado**.

2. **Enunciado del problema**.

   Ha de reproducirse copia del enunciado del problema y, si procede, una lista de modificaciones o extensiones del mismo junto con las justificaciones que las sustenten.

3. **Especificación del algoritmo.**

   Especificación inicial del algoritmo, antes de realizar ninguna inmersión. Debe incluir precondición, declaración o cabecera de la función y postcondición. Si bien la sintaxis concreta no es especialmente importante, el desarrollo de la practica deberá servir, también, al propósito de adquirir fluidez en su manejo, por lo que recomendamos se siga una de las propuestas en los textos de seguimiento de la Asignatura. De cualquier forma, se exige coherencia y consistencia en el uso de la sintaxis.

4. Algoritmo recursivo no final:

- Especificación de la función inmersora y llamada inicial.

La función requiere, habitualmente, una inmersión de diseño, cuya especificación ha de obtenerse a partir de la original, más la extensión de parámetros que sea necesaria. Además ha de anotarse la llamada inicial que hace que la función inmersora obtenga el resultado deseado. Para ello, habrán de

suministrarse valores iniciales adecuados para los parámetros nuevos (los propios de la función inmersora, que no estaban en la original).

- Análisis por casos.

De la función inmersora, comentados de manera que se pueda seguir el razonamiento sobre su validez. Nótese que, cuanto más precisos sean dichos argumentos, más fácil será construir la verificación de la función

- Composición algorítmica.

Código del algoritmo obtenido a partir del análisis por casos, junto con los comentarios que sean necesarios.

- Verificación formal de la corrección.

En la forma que aparece detallada en las referencias bibliográficas sugeridas para el estudio de la Asignatura, o sea:

I. Corrección parcial:
  (a) Completitud de la alternativa entre los casos (triviales y no triviales).
  (b) Satisfacción de la precondición para los parámetros de la llamada interna.
  (c) Base de la inducción (satisfacción de la postcondición para los casos triviales).
  (d) Paso de inducción (satisfacción de la postcondición para los casos recursivos, a partir de la hipótesis de inducción, esto es, la suposición de la satisfacción de dicha postcondición para los datos de la llamada interna).

II. Terminación de la función:
  (e) Elección de una estructura de preorden bien fundado.
  (f) Demostración del decrecimiento de los datos, en ese preorden, para las sucesivas llamadas recursivas.

- Estudio del coste del algoritmo.
  Lo que implica plantear la recurrencia adecuada, identificar los parámetros y **justificarlos oportunamente**, y aplicar la fórmula correspondiente para obtener su solución.

5. **Algoritmo recursivo final:**

- árbol sintáctico de la llamada recursiva, es decir, la expresión de la llamada recursiva interna diseñada para la primera función inmersora en forma de árbol sintáctico.

- Substituciones aplicadas según la heurística propuesta por Yves Kodratoff, explicada y aplicada en los ejemplos dados en [Peña93, Técnica de desplegado y plegado, pag(s) 81 a 87] o [Peña97, Técnica de desplegado y plegado, pag(s) 94 a 99], han de substituirse los términos del subárbol donde aparezca la llamada recursiva por variables (que serán nuevos parámetros para la función inmersora recursiva final), salvo en el caso de la propia llamada. Deben indicarse tales substituciones.

- Función desplegada y su equivalente plegada. Llamada inicial.
  O, lo que es lo mismo, realizar el desplegado y el plegado con los nuevos parámetros, explicarlo, y dar la llamada inicial de la función inmersora recursiva final que equivale a la función recursiva diseñada anteriormente.

- Código de la función recursiva final.
  Es decir, el obtenido por transformación a recursiva final de la función diseñada con anterioridad.
  Es importante hacer constar las nuevas precondición y postcondición junto al código de la función resultante, ya que estas diferirán de las de la función recursiva no final.

6. **Algoritmo iterativo:**

- Pasos efectuados para la derivación formal del algoritmo recursivo.
  Junto con todas aquellas justificaciones que convengan a su explicación
- Código del algoritmo iterativo resultante por derivación directa, **no por transformación del recursivo final**.
- Estudio del coste.
  Función de cota del bucle obtenido. Estudio del coste, con mención explícita a la forma en que se ha obtenido.

7. **Implementación**: El programa se codificará en Modula 2 y constara de 3 módulos.
   En todos los casos, el código en Modula-2 ha de estar lo mejor comentado posible.

- **Módulo** Pantalla:
  - Modulo de definición del tipo de datos pantalla para representar la pantalla del tetris.
    Para poder realizar un estudio empírico del coste el programa debe poder manejar pantallas de diversa longitud, por lo que el tipo de datos pantalla será un tipo estructurado conteniendo:
    * La matriz donde se almacenan realmente los valores de la pantalla. (Un vector bidimensional)
    * Dos valores naturales que indiquen las dimensiones efectivas de la matriz, definidas por restricciones sobre unas dimensiones máximas.
  - Procedimiento de carga de la matriz de datos desde un fichero
    La estructura del fichero será la siguiente:
    * Un natural que indique el tamaño en filas de la pantalla.
    * Un natural que indique el tamaño en columnas de la pantalla.
    * A continuación tantos naturales como hagan falta para codificar una pantalla de tetris con las medidas indicadas.

    Por ejemplo, supongamos un fichero con el contenido siguiente

    3 6

    0 0 0
    3 0 0
    3 0 3
    3 2 3
    2 2 3
    define una posición de Tetris de una pantalla de 3 columnas y 6 filas con 3 piezas, 2 de ellas iguales, y con 2 líneas a simplificar
- **Módulo** Matriz:
  Contiene el modulo de definición de la matriz y las funciones de comprobación de la posibilidad de simplificar dicha matriz. Estas funciones recibirán como parámetro una estructura de datos matriz y un valor f natural.
  Las funciones que habrá que implementar son:
  - función que codifique el **algoritmo recursivo no final**.
    Es decir, implementación en Modula-2 de la primera función inmersora. Ha de incluir, como comentarios, la precondición y la postcondición, así como la definición de la estructura de preorden bien fundado.
  - función que codifique el **algoritmo recursivo final**.
    Implementación en Modula-2 de la segunda función inmersora obtenida mediante la aplicación de la técnica de Desplegado/Plegado a la primera función. Habrá de incluirse en el código mediante el uso de comentarios la precondición y la postcondición de dicha función.

- función que codifique el **algoritmo iterativo**.

  Incluyendo como documentación los invariantes y la función de cota del bucle.

- **Modulo principal**:
  - El modulo principal importara el modulo de la definición de pantalla y de la matriz y realizara las actividades siguientes:
    * lectura del fichero
    * mostrar el fichero por pantalla al usuario
    * Preguntar al usuario por un numero de fila y comprobar que este está en un rango válido según la matriz leída
    * Invocar consecutivamente a cada una de las funciones que comprueban la simplificación de la fila. Esta invocación se realizara para cada función dentro de un bucle con un numero significativo de iteraciones a fin de que pueda medirse el tiempo que emplean en resolver la simplificación. Se leerá por tanto el reloj del sistema antes y después de cada uno de los 3 bucles. Al finalizar cada uno de ellos se escribirá el tiempo medido en pantalla.

8. **Juego de pruebas**:

Un juego de pruebas consiste en un conjunto de datos de entrada junto con la salida esperada para esos datos. esta se contrasta con la salida que realmente produce el programa. Los juegos de pruebas deberán estar lo mas cercanos posibles a la exhaustividad, probando cualesquiera condiciones potencialmente productoras de errores (como por ejemplo extremos de los intervalos de inmersión, intervalos vacíos, puntos de corte o condiciones no habituales).

Los casos de prueba obligatorios incluirán al menos 2 ficheros: tetris1.txt de 6x12 valores y tetris2.txt de 10x20

9. **Estudio empírico del coste**:

Con los datos sobre los tiempos invertidos en los cálculos, obtenidos a partir de las diferentes ejecuciones del programa, se pide que se construya una gráfica que estudie el coste empírico de los algoritmos programados. Ello implica que deben realizarse diversas ejecuciones con problemas de diferentes tamaños, ya que, lo que se pretende estudiar con la gráfica es el crecimiento del coste respecto al tamaño del problema.

Esta gráfica representara el tiempo empleado en la ejecución del algoritmo en función de la longitud del vector de entrada. Al contar con tres algoritmos (recursivo no final, recursivo e iterativo) que resuelven el problema, se deberá realizar una gráfica por algoritmo, para así poder realizar un estudio comparativo del coste real de los algoritmos implementados.

3 Enunciado de la práctica

3.1 Enunciado: texto del problema

En el conocido juego del **Tetris** se dispone de una pantalla por donde caen las piezas que se codifica en una matriz de **n** filas por **m** columnas definida como sigue:

t : **vector**[1..n; 1..m] **de nat**

Siendo la primera coordenada de la matriz el número de la fila y la segunda coordenada el número de la columna.

Cada posición almacena un número natural que representa el código del color de la pieza que ocupa dicha casilla. El valor 0 representara una casilla vacía.

Se pide diseñar una función completamente verificada que, dada una matriz *t* con las características descritas y un numero *f* entre 1 y *n* nos diga si la fila f-ésima esta completamente llena de piezas y, por tanto, representa una línea completada en el juego.

3.2 Comentarios al enunciado

Se trata de:

1. **Especificar** una función que resuelva el problema planteado.

2. **Verificar** formalmente la corrección del algoritmo obtenido y hallar su **coste**.

3. **Transformar** el algoritmo en otro *recursivo final* mediante la técnica de **desplegado y plegado** posterior del algoritmo inicial.

4. **Diseñar formalmente** un algoritmo iterativo que resuelva el problema.

5. **Implementar** en Modula-2 cada uno de los algoritmos obtenidos, así como un programa principal que los llame y que permita estimar (empíricamente) el crecimiento del coste temporal respecto al del tamaño del problema.

3.3 Cuestiones sobre el enunciado

1. Piense intuitivamente como resolvería a mano el problema y exponga claramente sus conclusiones.

4 Especificación

4.1 Consideraciones sobre la especificación

Para la construcción de la especificación del problema han de tenerse en cuenta los siguientes aspectos:

- El primer paso puede ser declarar la función, esto es, darle un nombre y declarar los parámetros que recibe y el resultado que devuelve, nombrándolos y decidiendo a que tipos de datos pertenecen.

- La postcondición expresa el resultado que se desea alcanzar, por lo que este será el siguiente paso. Se trata de expresarla de forma precisa.

- Ténganse en cuenta, al expresar la postcondición, todos los casos posibles, con especial atención a los *rangos vacíos* y a los casos extremos, ya que la sintaxis, usada imprecisamente, puede dar lugar a equívocos y a definiciones incorrectas. Recuérdese que la postcondición debe expresar una cierta relación entre las variables de salida y las de entrada.

- Una vez decidida la postcondición, la precondición debe restringir los casos potencialmente erróneos, limitando el dominio de aplicación de los datos de entrada.

- Debido a la naturaleza del problema la función que deberíamos obtener **no nos va a permitir** realizar directamente un diseño recursivo, por lo cual deberemos especificar otra función que sí nos lo permita (es importante convencerse de este hecho que puede presentarse en multitud de ocasiones).

4.2 Cuestiones sobre la especificación

1. Si en la postcondición de una función no se hace referencia alguna a las variables de entrada, ¿que podemos asegurar de dicha función?

4.3 Especificación formal de la función (trabajo del alumno)

La ***interfaz*** del algoritmo con el resto del programa indica las variables de entrada y de salida, nombrándolas y decidiendo el tipo de datos al que pertenecen. En este caso tendremos dos parámetros de entrada: uno de tipo vector de dos dimensiones (*t: vect[1..n,1..m] de nat*) y, el otro, un número que indique la fila que se quiere analizar (*f: nat*). A la salida la función deberá indicar si la fila *f-ésima* está completamente llena de piezas y por tanto representa una línea completada en el juego. Esto significa que el resultado de la función es un booleano (*b:bool*) que devolverá el valor "*cierto*" cuando se cumpla la condición impuesta (línea completada), y "*falso*" en caso contrario

La interfaz puede ser: **fun** linea (t: vect[1..n,1..m] de nat; f: nat) **dev** b: bool

La ***postcondición*** deberá indicar claramente si:

a) todos los elementos de la fila *f-ésima* de la matriz *t* son distintos de cero, en cuyo caso la condición del enunciado se cumple y por tanto la función devolverá el valor "*cierto*"
b) o, por el contrario, existe uno o varios elementos de la fila *f-ésima* de la matriz *t* iguales a 0, en cuyo caso no se cumple la condición "*linea completada*" y, por tanto, la función devolverá el valor "*falso*"

Para verificar dichas condiciones se puede utilizar:

a) el cuantificador universal "$\forall$" de la forma $\{R \equiv b = \forall\alpha\in\{1..m\}.\ t[f,\alpha] \neq 0\}$
b) o, el cuantificador existencial "$\exists$" $\{R \equiv b = \neg(\exists\alpha\in\{1..m\}.\ t[f,\alpha] = 0)\}$.

Vamos a tomar como precondición $\{R \equiv b = \forall\alpha\in\{1..m\}.\ t[f,\alpha] > 0\}$ ya que, al tratarse de naturales $t[f, \alpha]\neq 0$ se puede escribir como $t[f,\alpha] > 0$

En cualquier caso, R puede ser una postcondición adecuada ya que en ella intervienen todas las variables involucradas en el problema. Además ha sido necesario utilizar una variable auxiliar (*ligada*) α para recorrer simbólicamente los elementos de la fila *f-ésima* de la matriz *t* entre sus índices mínimo y máximo.

Respecto a la ***precondición***, ¿qué condiciones han de cumplir los datos a la entrada?
El propio enunciado del problema nos dice que ***1 £f £n*** , marcando los límites mínimo y máximo de la variable f.
La cuestión que nos planteamos es si se debe exigir alguna condición de dominio para los valores *n* y *m* que determinan las dimensiones de la matriz *t*. ¿Qué pasará *m = 0*?. Observando la postcondición, (en cualquiera de las variantes propuestas), el valor de *b* será "*cierto*" (véase Peña93 pág. 36 "Convenio sobre cuantificadores"). ¿Y para *n = 0*?. Esta posibilidad queda excluida ya que al imponer a la entrada la condición *1 £f £n ≡ 1 £f Ù f£n* → *1 £ n*
En consecuencia, las llamadas con *n = 0* son *incorrectas*, mientras que las llamadas con *m = 0* son correctas y devuelven el valor "*cierto*"

La especificación del algoritmo podría ser:

{Q ≡ 1 ≤ f ≤ n}
fun linea (t: vect[1..n,1..m] de nat; f: nat) **dev** b: bool
{R ≡ b = ∀α∈{1..m}. t[f,α] > 0}

---

5 Diseño recursivo no final

5.1 Especificación de la función inmersora y llamada inicial.

Para diseñar una función recursiva necesitamos contar con algún parámetro de entrada sobre el que se pueda establecer un subproblema, en función de cuya solución podamos resolver el problema original. Ello implica que el tamaño del subproblema **debe ser menor que el del problema**.
La especificación de la función que acabamos de terminar recibe como parámetros de entrada un vector de enteros, lo que significa que necesitaremos algún nuevo parámetro que nos permita definir subproblemas de menor tamaño a partir del problema original.
La función, que definiremos a partir de la originalmente especificada, y que incorporara al nuevo parámetro, será una *función inmersora* de la función que hemos especificado. Eso quiere decir que, además de la nueva especificación, debemos obtener una *llamada inicial*, asignando un valor adecuado al parámetro inmersor, de forma que, el resultado de la función inmersora sea el mismo que hemos definido para la original. Por cierto, utilizaremos una notación uniforme de cara a una mejor comprensión, de manera que, si a la función original le dábamos el nombre (por ejemplo) de *linea*, a la inmersora la llamaremos *ilinea.*
En la especificación hemos de tener especial cuidado en acotar el rango del nuevo parámetro, para evitar comportamientos erróneos o no deseados.

5.2 Calculo de la inmersión y llamada inicial (trabajo del alumno)

Podemos "*manipular*" la postcondición original introduciendo una nueva variable en sustitución de una constante para *debilitarla*.

Escribimos: $\underbrace{b = [\forall\alpha\in\{1..m\}.\ t[f,\alpha] > 0]}_{R} \equiv \underbrace{b =([\forall\alpha\in\{1..i\}.\ t[f,\alpha] > 0]}_{R_1} \wedge \underbrace{(i = m))}_{R_2}$

La postcondición original R se puede expresar entonces como $R \equiv R_1 \wedge R_2$ siendo:

$\{R_1 \equiv b = \forall\alpha\in\{1..i\}.\ t[f,\alpha] > 0\}$ y $\{R_2 \equiv (i = m)\}$

Tomando ***R_1*** como ***postcondición*** de la nueva función *ilinea*, la interfaz de la misma requerirá la inclusión del nuevo parámetro "*i*". La ***interfaz*** de la función inmersora podría ser:

fun ilinea (t: vect[1..n,1..m] de nat; f: nat; i:nat) **dev** b: bool

La nueva ***precondición*** Q_1 debe completarse con las condiciones que debe cumplir la nueva variable "*i*". En principio podríamos escribir $\{Q_1 \equiv 1 \le f \le n \wedge 1 \le i \le m\}$, pero debemos plantearnos: ¿es posible ampliar el rango de *i*? ¿Será correcta la llamada a *ilinea* para el caso "*i = 0*"? Si *i = 0,* el valor de ***b*** será "*cierto*" por los mismo motivos expuestos anteriormente, luego podemos admitir el valor *i = 0*

Hasta ahora tenemos:

$\{Q_1 \equiv 1 \le f \le n \wedge 0 \le i \le m\}$
fun ilinea (t: vect[1..n,1..m] de nat; f: nat; i:nat) **dev** b: bool
$\{R_1 \equiv b = \forall\alpha\in\{1..i\}.\ t[f,\alpha] > 0\}$

que será la especificación formal de la nueva función *ilinea* .

Para obtener el mismo resultado con *ilinea* que el pretendido con la función original *linea*, bastará con ejecutar *ilinea* para *i = m*, es decir *linea(t, f) = ilinea(t, f, m).* De esta forma la nueva función "recorrerá" todos los elementos de la fila f-ésima de la matriz t

Por tanto, la llamada inicial será: *linea(t, f) = ilinea(t, f, m)*

5.3 Análisis por casos.

El análisis por casos de una función recursiva consiste en obtener una clasificación, dependiente de los parámetros de entrada, que permita discernir para que valores de estos existe una solución inmediata (que no requiere ulterior descomposición recursiva), y para cuales se necesita apoyar la solución en curso en mas invocaciones recursivas de la propia función.
En cada caso, se trata de obtener una *protección* (o condición booleana que se debe cumplir para que se proceda a devolver la parte derecha de dicha expresión) y el resultado a devolver si esta se abre. Ha de tenerse en cuenta, además, que la disyunción de *todas* las alternativas debe evaluarse a **cierto**, para evitar que haya casos sin tratar.
En la practica que nos ocupa, el caso *trivial* (el no recursivo) puede tratarse del análisis de un vector vacío. El caso recursivo requerirá la evaluación de un predicado lógico en función de lo obtenido en una llamada recursiva.

5.4 Análisis por casos de la función inmersora (trabajo del alumno)

Planteamos dos casos posibles según el valor de la variable *i* :

- Si *i = 0* → *b = cierto* *(caso trivial)*
- Si *i > 0* → *?* *(caso no trivial)*

Para resolver el caso *no trivial* y aunque después se verificará formalmente el diseño, deberemos tener en cuenta las siguientes consideraciones:

a) Deberemos realizar una llamada recursiva a la función que estamos diseñando pero con un tamaño de problema reducido. En nuestro caso se podría considerar la llamada recursiva *ilinea (t, f, i-1)*.

b) Analizar qué condiciones cumple el resultado de esta llamada (cumplirá la postcondición de la función para el valor de los parámetros utilizados en la llamada R_1^{i-1}).

   Sea *b' = ilinea (t, f, i-1) =* **"aÎ{1..***i-1***}.** *t[f,***a***] > 0*

c) Realizar las operaciones adicionales necesarias para obtener un resultado que cumpla la postcondición para los parámetros originales a partir del valor devuelto por la llamada recursiva

   Sea *b = ilinea (t, f, i) =* **"aÎ{1..***i***}.** *t[f,***a***] > 0*

   Puesto que *b'* comprueba que los *i-1* primeros elementos de la fila *f-ésima* de la matriz *t* sean distintos de cero, y *b* hace lo mismo con los *i* primeros elementos de dicha fila, para alcanzar el mismo resultado bastará comprobar si el elemento *i* es distinto de *0* ; es decir, *b= b'* **Ù***(t[f, i] > 0)*

   Luego para el caso *recursivo (no trivial)* se puede escribir

   <u>*i > 0* → *ilinea(t, f, i-1)* **Ù***(t[f, i] > 0)*</u>

d) Asegurarnos informalmente de que la reducción aplicada conduce a casos más pequeños que necesariamente han de terminarse en el caso *trivial*. En este caso es evidente que *i* irá decreciendo hasta llegar a *i = 0*

e) Asegurarnos de que entre el caso *trivial* y el *no trivial* se cubren todos los estados previstos en la precondición. En este caso también se cumple.

Ya estamos en condiciones de escribir la *inmersión no final*

5.5 Composición algorítmica (trabajo del alumno).

La composición algorítmica no es mas que la expresión, utilizando una sintaxis algorítmica, del trabajo que hemos realizado hasta el momento, es decir, la especificación formal y el análisis por casos. En ella deben pues incluirse la precondición, la declaración o cabecera de la función, la alternativa de las protecciones con sus resultados, dada por el análisis por casos, y la postcondición.
Si bien, como ya aludamos en el comentario introductorio a la documentación requerida (véase el apartado 2.2 en la pagina 7), la sintaxis concreta no es especialmente relevante, sí es menester fijar y utilizar una sola forma de notación, y hacerlo de manera consistente y coherente. Así pues, utilizaremos la sintaxis descrita en [Peña93] o [Peña97], con la que el alumno deberá estar familiarizado antes de abordar la realización de este apartado.
La composición algorítmica, que estamos a punto de escribir, constituye el código de la función, que, en adelante, nos servirá de documentación y de base para la verificación formal de la corrección.
En el espacio dejado a este propósito a continuación, transcríbase la composición algorítmica de la función *ilinea*.

El diseño recursivo por inmersión no final quedará:

$\{Q_1 \equiv \ (1 \leq f \leq n) \ \wedge \ (0 \leq i \leq m)\}$
fun ilinea (t: vect[1..n,1..m] de nat; f, i: nat) **dev** b: bool
  caso i = 0 → cierto
  [] i > 0 → ilinea(t, f, i-1) ∧ (t[f,i] > 0)
  fcaso
ffun
$\{R_1 \equiv \ b = \forall\alpha\in\{1..i\}.\ t(f,\alpha) > 0)\}$

5.6 Verificación formal de la corrección.

Una vez obtenido el código de la función *ilinea*, hemos de verificar, formalmente, su corrección. Para simplificar las explicaciones copiaremos, a continuación, la forma general de una función recursiva lineal, tal como se describe en [Peña93, Figura 3.2, pag(s) 53] o [Peña97, Figura 3.2, pag(s) 61]:

$\{Q(\overline{x})\}$
fun $f(\overline{x} : T_1)$ **dev** $(\overline{y} : T_2) \equiv$
  caso $B_t(\overline{x}) \rightarrow triv(\overline{x})$
  [] $B_{nt}(\overline{x}) \rightarrow c(f(s(\overline{x})), \overline{x})$
  fcaso
ffun
$\{R(\overline{x}, \overline{y})\}$

Figura 1: Forma abstracta de una función recursiva lineal

5.6.1 Completitud de la alternativa (trabajo del alumno)

Como la función ha de estar bien definida en su dominio, hay que demostrar que el conjunto de protecciones que hemos obtenido en el análisis (véase 5.3) cubre *todos* los casos posibles, para lo que hay que contar con la precondición, ya que esta da el conjunto de estados que son pertinentes a la función.

Esto es lo mismo, y siguiendo la notación de la figura 1, hay que demostrar que $Q(\overline{x}) \Rightarrow B_t(\overline{x}) \vee B_{nt}(\overline{x})$, que es el siguiente trabajo que hay que llevar a cabo.

- En este caso es evidente que *[(1 £f £n) Ù (0 ≤ i £m)] Þ (i = 0) Ú(i > 0)*

5.6.2 Satisfacción de la precondición para la llamada interna (trabajo del alumno)

La función ha de invocarse siempre en estados que satisfagan su precondición, o sea, hay que demostrar que $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \Rightarrow Q(s(\overline{x}))$.

- La precondición aplicada a *i* es *[(1£f£n)Ù(0≤ i£m)]* y para *i-1* es *[(1£f£n)Ù(0≤ i-1£m)]*. Tendremos que comprobar que *[(1£f £n)Ù(0≤ i £m)]Ù(i > 0) Þ [(1£f£n)Ù(0≤ i-1£m)]*, que se reduce a comprobar que *(0 ≤ i £m)Ù(i > 0)Þ0£i-1£m*, es decir *1≤ i£m Þ0£i-1 £m* lo que resulta evidente.

5.6.3 Base de la inducción (trabajo del alumno)

Esto es, ha de demostrarse la satisfacción de la postcondición para los casos triviales, o sea, $Q(\overline{x}) \wedge B_t(\overline{x}) \Rightarrow R(\overline{x}, triv(\overline{x}))$.

- En el caso trivial, se devuelve el valor *cierto* y es evidente que *b = ["***aÎ{1..i}.** *v(f,***a***) > 0]* para el caso *i = 0* toma el valor *b = cierto*, en consecuencia podemos afirmar que el valor devuelto por la función en el caso trivial cumple la postcondición

---

5.6.4 Paso de inducción (trabajo del alumno)

Hay que probar la satisfacción de la postcondición para los casos recursivos, a partir de la hipótesis de inducción, esto es, de la suposición de la satisfacción de dicha postcondición para los datos de la llamada interna. Formalmente, hay que demostrar que $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \wedge R(s(\overline{x}), \overline{y}') \Rightarrow R(\overline{x}, c(\overline{y}', \overline{x}))$

En este ejemplo, la llamada se realiza disminuyendo el valor de *i*, es decir *s* transforma *i* a *i - 1* y suponiendo que dicha llamada cumple la postcondición, el valor devuelto supone la comprobación de que los *i-1* primeros elementos de la fila *f-ésima* son distintos de *cero*. En la operación adicional se comprueba si el elemento *t[f,i]* es también distinto de *0*, con lo que se obtiene la postcondición aplicada a *i*, ya que ésta lo que comprueba es precisamente que los *i* primeros elementos de la fila *f-ésima* de la matriz *t* sean distintos de *cero*.

Formalmente:

$R(s(\overline{x}), \overline{y}') = b' =$ *["***aÎ{1..***i-1***}.** *t[f,***a***] > 0]*

$Q(\overline{x}) \wedge B_{nt}(\overline{x}) =$ *[(1 £f £n)***Ù***(0 £i £m)]***Ù***(i > 0)* **Þ** *(1£f£n)***Ù***(1£i£m)*

$R(\overline{x}, c(\overline{y}', \overline{x}))$ ¿≡? *["***aÎ{1..i}.** *t(f,***a***) > 0] = b = b'* **Ù** *(t[f,i] > 0) ya que*

$c(\overline{y}', \overline{x}) = b'$ **Ù** *(t[f,i] > 0)*

b' **Ù** *(t[f,i] > 0) = [(t[f, 1] > 0)* **Ù***(t[f, 2] > 0)* **Ù***……..***Ù** *(t[f, i-1] > 0)])* **Ù** *(t[f,i] > 0)*

*= "***aÎ{1..i}.** *t[f,***a***] > 0 = b*

5.6.5 Elección de una estructura de preorden bien fundado (trabajo del alumno).

Hay que encontrar $h : D_{T1} \rightarrow Z$ tal que $Q(\overline{x}) \Rightarrow h(\overline{x}) \geq 0$

$Q(\overline{x})$ ≡ *[(1£f£n)* **Ù***(0£i£m)]*

h(t,f,i) = i

} ⇒ *i ³0*

5.6.6 Demostración del decrecimiento de los datos (trabajo del alumno).

En ese preorden, para las sucesivas llamadas recursivas. Esto equivale a demostrar que $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \Rightarrow h(s(\overline{x})) < h(\overline{x})$

$Q(\overline{x}) \wedge B_{nt}(\overline{x}) =$ *[(1 £f £n)* **Ù***(0 £i £m)]* **Ù***(i > 0)* **Þ** *h(t,f,i-1)= i-1* **£** *i = h(t,f,i)*

Una vez verificada formalmente la corrección de la función propuesta, aplicando la Máxima37 (Peña93 pag 64) podríamos transformar el código de la función *ilinea*

```
{Q1.1 ≡  (1 ≤ f ≤ n)  ∧  (0 ≤ i ≤ m)}
fun ilinea (t: vect[1..n,1..m] de nat; f, i: nat) dev b: bool
        caso    i = 0  → cierto
          []    i > 0  → sea r = t[f,i] en
                            caso r = 0 → falso
                              []   r > 0 → ilinea(t, f, i-1)
                            fcaso
        fcaso
ffun
{R1.1 ≡ b = ∀α∈ {1..i}. t[f,α] > 0)}
```

Con esta nueva propuesta tenemos la ventaja de que en el caso no trivial lo primero que hacemos es comprobar si t[f,i] = 0 y en ese caso se interrumpe el procedimiento recursivo y se devuelve inmediatamente el valor falso sin necesidad de seguir recorriendo el resto de elementos de la fila. Además hemos transformado la función en recursiva final ya que no existe operación adicional, (no es necesaria).

5.7 Estudio del coste del algoritmo (trabajo del alumno).

Esto implica plantear una recurrencia a partir del tamaño del problema y la forma de decrecimiento de este en el p.b.f., identificar los parámetros y resolverla, tal como se muestra en [Peña93, Resolución de recurrencias, pag(s) 14 a 17] o [Peña97, Resolución de recurrencias, pag(s) 16 a 20].

- El tamaño del problema viene dado por *m* que representa el número de columnas de la matriz En este caso el tamaño del problema decrece por substracción siendo pues las fórmulas a aplicar:

$$T(n) = \begin{cases} cn^k & \text{si} \quad n < 1 \\ aT(n-b) + cn^k & \text{si} \quad n \geq 1 \end{cases} \qquad y \qquad T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \, div \, b}) & \text{si } a > 1 \end{cases}$$

Como el coste del caso trivial es constante (consiste en devolver el valor *cierto*), $k = 0$. Además $b = 1$ ya que a cada nueva invocación de la función el tamaño de los datos decrece en una unidad. Y $a = 1$ ya que sólo se produce una llamada interna a la función.

$$T(m) = \begin{cases} c & \text{si} \quad m < 1 \\ T(m-1) + c & \text{si} \quad m \geq 1 \end{cases} \qquad y \qquad T(m) \in \Theta(m)$$

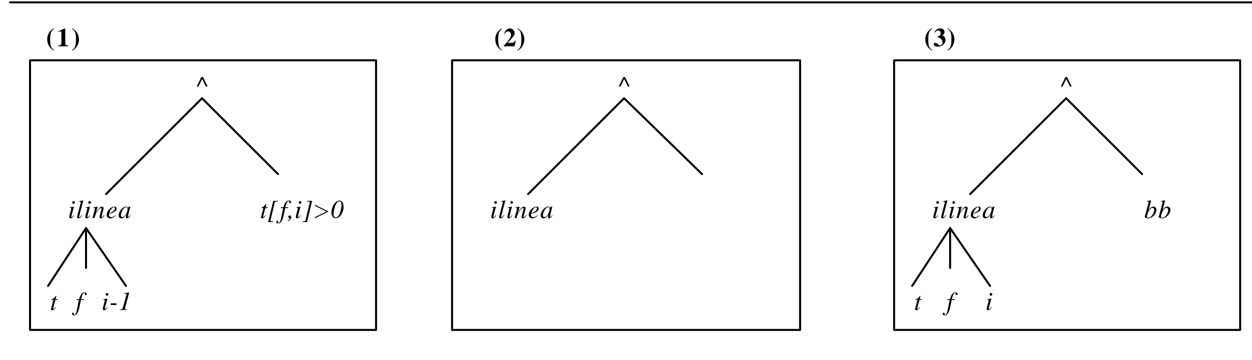
Es decir, se trata de un algoritmo de coste lineal

---

6 Transformación a recursivo final: Desplegado-plegado

6.1 Arbol sintáctico de la función recursiva. (trabajo del alumno)

Dibújese el árbol sintáctico de la llamada recursiva interior de la función no final *ilinea* (rama del caso no trivial).



6.2 Substituciones aplicadas. (trabajo del alumno)

Según la heurística de Kodratoff, explicada y aplicada en los ejemplos dados en [Peña93, Técnica de desplegado y plegado, pag(s) 81 a 87] o [Peña97, Técnica de desplegado y plegado, pag(s) 94 a 99]. Detallar las substituciones aplicadas.

- La substitución aplicada es *bb = (t[f,i] > 0)*
  bb es una variable de tipo booleano que tomará el valor *cierto* cuando el valor del elemento *i* de la fila *f-ésima* de la matriz *t* sea distinto de *0* y *falso* si *t[f,i] = 0.*

  La nueva función *iilinea* deberá cumplir: ***iilinea(t, f, i, bb) = ilinea(t, f, i) Ùbb***
  y para *bb=cierto* tendremos *iilinea(t, f, i, cierto) = ilinea(t, f, i)*

6.3 Desplegado y plegado. Llamada inicial. (trabajo del alumno)

En este apartado han de listarse todas las transformaciones que permiten desplegar y posteriormente plegar la función *ilinea* en la nueva función, recursiva final, *iilinea* (inmersión de *ilinea*). Ha de darse, además la llamada inicial que hace que *iilinea(?) = ilinea(?)*, análogamente al punto 5.1, donde se calculaba la llamada inicial para que *ilinea(?) = linea(?)*
Es necesario recalcar en la importancia de modificar la precondición y la postcondición, ya que al realizar el desplegado y posterior plegado se añadirá a la función un nuevo parámetro, por lo cual la precondición y la postcondición deberán reflejarlo de forma adecuada.

- Para poder aplicar la técnica del *desplegado-plegado* es necesario que la operación adicional en la función de combinación (c según *figura1* del apartado 5.6) tenga las propiedades de *elemento neutro y asociativa.*
  La operación lógica ∧ tiene como elemento neutro el valor *cierto* y sabemos que cumple la propiedad *asociativa (P ∧ Q) ∧ R º P ∧ (Q ÙR)*

- Para desarrollar la técnica de desplegado y plegado basta seguir los siguientes pasos:
    - Sustituir en *iilinea(t, f, i, bb) = ilinea(t, f, i)* $\wedge$ *bb* la función *ilinea* por su desarrollo trasladando la operación adicional a cada uno de los casos

      iilinea (t, f, i, bb) = *caso i = 0* → *cierto* $\wedge$ *bb*
      caso i > 0 → *[ilinea(t, f, i-1)* $\wedge$ *(t[f, i] > 0)]* $\wedge$ *bb*

    - Aplicamos la propiedad *elemento neutro* en el caso *trivial* y *asociativa* en el no trivial:

      iilinea (t, f, i, bb) = *caso i = 0* → *bb*
      caso i > 0 → *ilinea(t, f, i-1)* $\wedge$ *[(t[f, i] >0)* $\wedge$ *bb]*

    - Aplicando la definición dada para *iilinea* tenemos *iilinea(t, f, i, bb') = ilinea(t, f, i)* $\wedge$ *bb'* al caso *no trivial* para *i = i - 1* y *bb' = ((t[f, i] > 0)* $\wedge$ *bb)*, con lo cual tendremos:

      iilinea (t, f, i, bb) = *caso i = 0* → *bb*
      caso i > 0 → *iilinea(t, f, i-1, [(t[f, i] > 0)* $\wedge$ *bb])*

    - Obtenemos una solución recursiva final para el problema:

```
fun iilinea (t: vect[1..n, 1..m] de nat; f, i: nat; bb: bool) dev b: bool
        caso   i = 0  →  bb
         []    i > 0  →  iilinea(t, f, i-1, [(t[f, i] > 0) ∧ bb])
        fcaso
ffun
```

- La ***postcondición*** de esta función será la de *ilinea* para *i = m*, es decir, la inicial:

  $\{R_2 \equiv b = \forall \alpha \in \{1..m\}.\ t[f,\alpha] > 0\}$

- En la ***precondición*** se han de incluir las condiciones adecuadas para la nueva variable *bb*. Estas condiciones pueden deducirse del hecho de que *iilinea(t, f, i, bb) = ilinea(t, f, i)* $\wedge$ *bb*. Comparando las postcondiciones de *ilinea* e *iilinea:*

  $\{R_1 \equiv b = \forall \alpha \in \{1..i\}.\ t[f,\alpha] > 0)]\}$ y $\{R_2 \equiv b = \forall \alpha \in \{1..m\}.\ t[f,\alpha] > 0)]\}$

  se deduce que, para que el resultado devuelto por la llamada *iilinea(t, f, i, bb)* sea igual al devuelto por *ilinea(t, f, i)* $\wedge$ *bb* tendrá que ser $bb = \forall \alpha \in \{i+1..m\}.\ t[f,\alpha] > 0$

  NOTA: el parámetro *bb* contendrá en cada llamada el resultado (*cierto o falso*) de comprobar que los *(m – i)-últimos* elementos de la fila *f-ésima* de *t* son distintos de *0*.

  La precondición será: $\{Q_2 \equiv (1 \le f \le n) \wedge (0 \le i \le m) \wedge (bb = \forall \alpha \in \{i+1..m\}.\ t[f,\alpha] > 0)\}$

  Como *iilinea(t,f,i,cierto) = ilinea(t,f,i)* y, por otra parte sabemos que *ilinea(t,f,m) = linea(t,f)*, la ***llamada inicial*** para la función *iilinea* será *linea(t, f) = iilinea(t, f, m, cierto).*

  Con esta llamada se cumple la precondición ya que al ser *m + 1 > m*, *bb* será *cierto*

6.4 Código de la función recursiva final. (trabajo del alumno)

Y, por fin, aquí se debe transcribir el código de la función recursiva final *ilinea*, completo con la precondición y postcondición.

- El código de la función recursiva final quedará:

  $\{Q2 \equiv (1 \leq f \leq n) \wedge (0 \leq i \leq m) \wedge (bb = \forall \alpha \in \{i+1..m\}.\ t(f,\alpha) > 0)\}$

  fun *iilinea (t: vect[1..n, 1..m] de nat; f, i: nat; bb: bool)* **dev** *b: bool*

  caso *i = 0* → *bb*

  [] *i > 0* → *iilinea(t, f, i-1, (t[f, i]>0)∧bb)*

  fcaso

  ffun

  $\{R_2 \equiv b = \forall \alpha \in \{1..m\}.\ t(f,\alpha) > 0)]\}$

- De manera análoga a como hemos comentado en el caso de la función recursiva no final, ahora también podríamos plantear otro desarrollo del código para la función *iilinea*

  $\{Q_2 \equiv (1 \leq f \leq n) \wedge (0 \leq i \leq m) \wedge (bb = \forall \alpha \in \{i+1..m\}.\ t[f,\alpha] > 0)]\}$
  fun *iilinea (t: vect[1..n, 1..m] de nat; f, i: nat; bb: bool)* **dev** *b: bool*
  caso *i = 0* → *bb*
  [] *i > 0* → caso *(t[f, i] = 0)* → *falso*
  [] *(t[f, i] > 0)* → *iilinea(t, f, i-1, bb])*
  fcaso
  fcaso
  ffun
  $\{R_2 \equiv b = \forall \alpha \in \{1..m\}.\ t(f,\alpha) > 0)]\}$

---

7 Diseño iterativo

7.1 Cuestiones preliminares sobre el diseño iterativo

A continuación se va a proceder a diseñar un nuevo algoritmo que resuelva el mismo problema, pero en este caso de forma iterativa.

Partiremos de la postcondición de la función especificada en 4.3, y derivaremos el invariante del bucle iterativo (consultar [Peña93, derivación del invariante a partir de la postcondición, pag(s) 117 y 118] o [Peña97, derivación del invariante a partir de la postcondición, pag(s) 133 y 134] y [Balc93, derivación directa de bucles, pag(s) 227 y ss.]).

7.2 Derivación del invariante partiendo de la postcondición (trabajo del alumno)

Escríbase aquí el invariante del bucle, junto con las explicaciones necesarias para explicar su derivación formal. Pista: se puede probar debilitando la postcondición.

- Partimos de $b = \forall \alpha \in \{1..m\}.\ t[f,\alpha] > 0 \equiv b = (\forall \alpha \in \{1..i\}.\ t[f,\alpha] > 0) \wedge (i = m))$ al igual que hicimos para el cálculo de la función recursiva no final, siendo $R \equiv R_1 \wedge R_2$ con $R_1 \equiv b = \forall \alpha \in \{1..i\}.\ t[f,\alpha] > 0)$ y $R_2 \equiv (i = m)$, pero ahora los términos R_1 y R_2 los utilizamos de forma diferente: uno de ellos servirá como <u>*invariante del bucle*</u>, y el otro, *negado*, como *condición de finalización* del mismo, de forma que . $P \wedge \neg B \Rightarrow R$

Probaremos con $P \equiv R1$ y $B \equiv R2$, tendremos entonces un primer diseño iterativo dado por:

fun *linea_iter(t:vect[1..n, 1..m] de nat; f: nat)* **dev** *b: bool*
Inicializar;
mientras $i \neq m$ **hacer** $\{P \equiv b = (\forall \alpha \in \{1..i\}.\ t(f,\alpha) > 0)\}$
Restaurar;
Avanzar;
fmientras
ffun
$\{R \equiv b = (\forall \alpha \in \{1..m\}.\ t(f,\alpha) > 0)\}$

Pero tengamos en cuenta que el invariante se satisface antes de la primera iteración, después de cada una de ellas y después de la última, es decir, a la terminación del bucle. Si el bucle termina, lo hace satisfaciendo el invariante *P* y, además, la negación de la condición *B*.
Para no *"perdernos"*, volveremos a recordar que lo que en nuestro caso garantiza el invariante es que la variable *b* representará el valor (*cierto / falso*) obtenido como resultado de comprobar si los *i-primeros* elementos de la fila *f-ésima* de *t* son distintos de cero. Cuando se compruebe que hemos alcanzado el último elemento de la fila $i = m$ la función devolverá el valor que en ese momento tenga *b*.

7.3 Instrucción avanzar del bucle (trabajo del alumno)

Una vez obtenido el invariante del bucle procederemos a derivar su cuerpo, comenzando por una instrucción avanzar que haga progresar el bucle hacia su terminación (consultar [Peña93, derivación de bucles a partir de invariantes, pag(s) 116] o [Peña97, derivación de bucles a partir de invariantes, pag(s) 132] y [Balc93, derivación directa de bucles, pag(s) 227 y ss.]).

- Antes de abordar el término *Avanzar* vamos a tratar el término *Inicializar.* Necesitamos unas instrucciones *Inicializar* que hagan válido *{Q} Inic {P}.* Esto supone declarar las dos variables que intervienen en el invariante (*i* , *b*) asignándoles valores que hagan que antes de iniciar el bucle se cumpla ya el predicado invariante *P*. Estos valores iniciales son los que hacen nulos los dominios de los cuantificadores. Así pues, asignamos el valor *0* a la variable *i.* Entonces resulta que $b = (\forall \alpha \in \{1..i\}.\ t(f,\alpha) > 0)]$ será *cierto*, por lo que asignaremos ese valor inicial para la variable *b* a fin de que se cumpla *P*.

Por tanto: **var** *i: nat, b: bool*
i:= 0;
b:= cierto

- En el término *Avanzar* se han de modificar las variables necesarias para que se acerque el bucle a su condición de finalización. En nuestro caso, dado que *i* se inicializa con el valor *0* y la condición de finalización es $i = m$, bastará con incrementar el valor de *i*, así *Avanzar* quedará:

Avanzar $\equiv i := i + 1$

7.4 Restablecimiento del invariante (trabajo del alumno)

Esta instrucción de avanzar hará que el invariante que habíamos calculado haya perdido su propiedad de invarianza, por ello será necesario incluir otra instrucción (o secuencia de instrucciones) que nos permita restablecer dicha propiedad (consultar [Peña93, derivación de bucles a partir de invariantes, pag(s) 116] o [Peña97, derivación de bucles a partir de invariantes, pag(s) 132] y [Balc93, derivación directa de bucles, pag(s) 227 y ss.]).

- Antes de la operación *Avanzar*, se cumple el invariante para *i* es decir que *b* indicará si los *i-primeros* elementos de la fila *f-ésima* de *t* son distintos de *0*, por tanto será:

  $$b = \forall \alpha \in \{1..i\}. t(f,\alpha) > 0 \equiv (t[f,1]>0) \wedge (t[f,2]>0) \wedge (t[f,3]>0) \wedge ... \wedge (t[f,i]>0)$$

  Para $i + 1$ el invariante impone que $b' = (\forall \alpha \in \{1..i + 1\}. t(f,\alpha)>0)$ que en forma desarrollada y aplicando la propiedad asociativa equivale a:

  $$b' = (\forall \alpha \in \{1..i + 1\}. t(f,\alpha) > 0) \equiv [(t[f,1]>0) \wedge (t[f,2]>0) \wedge ... \wedge (t[f,i]>0)] \wedge (t[f,i+1]>0)$$

  podemos concluir que $b' = b \wedge (t[f,i+1]>0)$

  De la expresión anterior obtenemos que la asignación necesaria para restaurar la propiedad invariante del predicado *P* será:

  $$Restaruar \equiv b := b \wedge (t[f,i+1]>0)$$

7.5 Código del algoritmo iterativo (trabajo del alumno)

A continuación se deberá escribir el código completo del algoritmo iterativo junto con su precondición y postcondición.

- El diseño quedará finalmente:

  $\{Q \equiv 1 \le f \le n\}$

  fun *linea_iter(t:vect[1..n, 1..m] de nat; f: nat)* **dev** *b: bool*

  var *i: nat, b: bool*
  i:= 0;
  b:= cierto

  mientras $i \neq m$ **hacer** $\{P \equiv (b = \forall \alpha \in \{1..i\}. t(f,\alpha) > 0)$

  $b := b \wedge (t[f,i+1]>0)$;

  $i := i + 1$;

  fmientras

  ffun

  $\{R \equiv b = (\forall \alpha \in \{1..m\}. t(f,\alpha) > 0)\}$

Al igual que hicimos con las funciones recursivas, para el caso iterativo podríamos plantearnos una reducción en el número de iteraciones. En el caso de que *b* tome el valor *falso* en alguna de las iteraciones ya no haría falta continuar con más repeticiones ya que evidentemente el resultado de la función será *falso*.

La solución que se propone a continuación realizará un número menor de iteraciones salvo para los casos en que la fila esté completada o sea el último elemento de la fila el único con valor *0*

```
{Q º 1£f £n}
fun linea_iter(t:vect[1..n, 1..m] de nat; f: nat) dev b: bool
        var  i: nat, b: bool
                i:= 0;
                b:= cierto
        mientras  (i ¹ m) ∧ (b= cierto) hacer{P º (b = " aÎ {1..i}. t(f,a) > 0)}
                b:= b Ù(t[f ,i+1]>0);
                i := i + 1;
        fmientras
ffun
{R ≡ b = (" aÎ {1..m}. t(f,a) > 0)}
```

7.6 Estudio del coste. (trabajo del alumno)

En este apartado se deberá escribir (a partir del invariante) la función de cota del bucle y el coste del algoritmo iterativo obtenido en el apartado anterior (esto ultimo, segun aparece en [Peña93, Reglas practicas para el calculo de la eficiencia, pag(s) 11 y ss.] o [Peña97, Reglas practicas para el calculo de la eficiencia, pag(s) 12 y ss.]). Se deberá razonar dicho coste según las anteriores reglas.

Para abordar el estudio del coste de la función iterativa recurriremos a las reglas prácticas para el cálculo de eficiencia explicadas en Peña93 apartado 1.4

- El coste de las operaciones internas al bucle es el coste de dos operaciones consecutivas de coste constante (+ y ∧) y por lo tanto el coste es constante
- El número de iteraciones depende de *m* y por la regla del producto, el coste total es:

$$\Theta = \Theta(1) * \Theta(m) = \Theta(m * 1) = \Theta(m)$$

7.7 Cuestiones sobre la función iterativa

1. Transforme la función recursiva final *iilinea* en una función iterativa *iilinea.* Compárese con la función obtenida en el apartado 7.5

La solución recursiva final era:

$\{Q_2 \equiv (1 \leq f \leq n) \wedge (0 \leq i \leq m) \wedge bb = (\forall \alpha \in \{i+1..m\}. t(f,\alpha) > 0)\}$

fun *iilinea (t: vect[1..n, 1..m] de nat; f, i: nat; bb: bool)* **dev** *b: bool*

caso $i = 0 \rightarrow bb$

$i > 0 \rightarrow iilinea(t, f, i\text{-}1, [(t[f, i] > 0) \wedge bb])$

fcaso

ffun

$\{R_2 \equiv b = (\forall \alpha \in \{1..m\}. t(f,\alpha) > 0)\}$

Su equivalencia iterativa dará lugar a un bucle en el que:

a) La inicialización de las variables consistirá en asignar a las variables involucradas los mismo valores que se utilizan en la llamada inicial en la función recursiva. En nuestro caso $i = m$ y $bb = cierto$

b) La condición del bucle será la del caso no trivial en el caso recursivo: $i > 0$

c) El invariante del bucle será el predicado dado por la igualdad entre la conjunción de la precondición del algoritmo recursivo y de la función aplicada a los valores iniciales y la propia función para valores genéricos, es decir:

$$P(\overline{x}, \overline{x}_{ini}) \equiv Q(\overline{x}) \wedge f(\overline{x}_{ini}) = f(\overline{x})$$

En la práctica conduce al predicado dado por la precondición de la función recursiva ya que la postcondición del algoritmo recursivo final no depende de los parámetros, luego en nuestro caso:

$$\{P \equiv (0 \leq i \leq m) \wedge bb = (\forall \alpha \in \{i+1..m\}. t(f,\alpha) > 0)\}$$

d) Se devolverá tras el bucle el mismo valor que se devolvía en el caso trivial en el algoritmo recursivo: *bb*

e) La precondición de la función iterativa será el predicado utilizado en la precondición de la recursiva en el que se sustituyen las variables genéricas por sus valores iniciales, es decir:

$$Q_2(\overline{x}_{ini}) = (1 \leq f \leq n) \wedge (0 \leq m \leq m) \wedge cierto = (1 \leq f \leq n)$$

Luego la precondición del algoritmo iterativo será $\{Q_{iiter} \equiv (1 \leq f \leq n) \wedge (0 \leq m)\}$

El algoritmo iterativo a partir del recursivo final será:

```
{Qiiter ≡ (1 ≤ f ≤ n) ∧ (0 ≤ m)}

fun iilinea_iter(t: vect[1..n,1..m] de nat; f: nat) dev b:bool
        var  i: nat; bb: bool;
        i:= m;
        bb:= cierto;
        mientras  i> 0 hacer {P ≡ (0 ≤ i ≤ m) ∧ bb=(∀α ∈ {i+1..m}. t(f,α) > 0)}
                bb:= bb ∧ t[f, i];
                i:= i – 1;
        fmientras
        dev bb;
ffun
{Riiter ≡ b= (∀α ∈ {1..m}. t(f,α) > 0)}
```

---

2. ¿Cuál es el coste del algoritmo iterativo, obtenido en el anterior punto? Compárese con el coste del algoritmo recursivo obtenido en 5.7 y con el del algoritmo iterativo obtenido en 7.6.

- El coste será el mismo que el obtenido en los algoritmos anteriores, es decir de tipo ***lineal***

Si se comparan los bucles obtenidos por derivación y el obtenido en el punto anterior se observa que se dan las mismas condiciones para el cálculo del coste

---

8 Implementación en Modula-2

8.1 Código Modula-2 y juego de pruebas (trabajo del alumno).

A partir de este punto, debe añadirse el código, en Modula-2 de las funciones *ilinea, iilinea y linea-it*, así como un procedimiento de recogida de datos, un programa principal, un juego de pruebas, y, por último, un estudio comparativo del coste, incluyendo una gráfica al efecto, como se describe en la documentación (véanse los puntos 7, 8, y 9 del apartado 2.2).

REFERENCES

[Balc93] J.L. Balcazar. Programación Metódica. McGraw-Hill, 1993.

[CCM + 93] J. Castro, F. Cucker, X. Messeguer, A. Rubio, L. Solano y B. Valles. Curso de Programación. McGraw-Hill, 1993.

[Col99] J.I. Mayorga Toledano , M. Rodrguez Artacho y F. López Ostenero Colección de problemas de Programación II. 1999.

[Peña93] R. Peña. Diseño de Programas: formalismo y abstracción. Prentice Hall, 1993.

[Peña97] R. Peña. Diseño de Programas: formalismo y abstracción, 2 a Edicion . Prentice Hall, 1997.

NOTA: *No publico el código de mi práctica porque considero que a pesar de funcionar correctamente, lo de programar no es lo mío, no obstante si alguien lo quiere me lo puede pedir a la siguiente dirección: puuboo@terra.es*