UNED. Centro Asociado de VITORIA-GASTEIZ

Escuela Técnica de Informática

Programación II

Ejercicio ejemplo 1

Curso 1996/97

Los presentes apuntes han sido elaborados para su distribución gratuita a los alumnos de Programación II de Informática de Sistemas y Gestión del CENTRO ASOCIADO DE VITORIA-GASTEIZ de la Escuela Técnica de Informática de U.N.E.D.. Se ofrecen para uso personal de dichos alumnos pudiéndose reproducir y transmitir siempre que se mantenga esta nota y no se realice actividad de intercambio comercial de ningún tipo. En cualesquiera otras condiciones la copia, transmisión o almacenamiento por cualquier medio fotográfico, informático, telemático u otros requiere la autorización expresa del autor. Se agradecerá la comunicación de cualquier errata así como cualquier comentario o sugerencia.
(e-mail :**jqcJ jet.es**)

© Jerónimo Quesada 1997

1. INTRODUCCIÓN

En estos apuntes se desarrolla un ejercicio de diseño y verificación de algoritmos. Se trata de un ejercicio realmente simple. Se ha elegido este ejemplo como una primera aproximación a las técnicas de diseño y análisis recursivo e iterativo de algoritmos, de forma que se facilite el seguimiento de las técnicas a aplicar sin que la complejidad de los predicados utilizados en las especificaciones y demostraciones oculte las partes fundamentales de la asignatura. Pretende ser el primero de una colección de ejercicios que el autor tratará de ampliar en el futuro dentro de sus posibilidades. En el desarrollo del ejercicio se sigue el Cuadernillo del Alumno publicado en el Departamento de la Sede Central para la práctica de Programación II del curso 96/97 de forma que pueda ser utilizado como ejemplo-guía durante el desarrollo de la práctica. Se hará referencia continua al texto base de la asignatura: "Diseño de Programas", R.Peña Marí, Ed. Prentice-Hall y se utilizará la notación definida en éste texto. Para el ejemplo dado, y a partir del enunciado de problema:

a) Se especificará rigurosamente el algoritmo
b) Se diseñará la solución recursiva no final
c) Se diseñará directamente la solución recursiva final
d) Se obtendrá la solución recursiva final a partir de la no final por plegado-desplegado
e) Se diseñará directamente la solución iterativa
f) Se obtendrá la solución iterativa por transformación de la recursiva

2. ENUNCIADO

Dado un vector de enteros de dimensión N diseñar y verificar un algoritmo que calcule la suma de los elementos del vector.

3. ESPECIFICACIÓN

Especificar consiste en establecer de forma rigurosa lo que se espera del algoritmo y las condiciones que han de cumplir los datos de entrada para que la aplicación del algoritmo sea válida. En un primer paso se ha de establecer ***la interfaz*** del algoritmo con el resto del programa indicando las variables de entrada y salida. Siguiendo la notación de Peña Cap 2 la interfaz se establece por medio de una declaración simbólica de ***función*** o ***acción***. En este ejemplo concreto:

fun suma(v: vect) **dev** s: entero

Se trata de una función que toma como entrada un **vect** y devuelve un entero. Siempre que se utilice un tipo de datos distinto de los predeterminados: natural, entero, booleano, real … se ha de definir exactamente el nuevo tipo de datos. En este caso el tipo **vect** es:

tipo **vect**= vector [1..N] de enteros.

Queda definido el tipo **vect** como un vector de enteros con índices entre 1 y N siendo N una constante.

Para acabar de especificar completamente el algoritmo se han de establecer la ***postcondición*** (qué se espera del algoritmo) y la ***precondición*** (qué condiciones han de cumplir los datos de entrada). La postcondición es un predicado que establece de forma rigurosa las condiciones que cumplen las variables significativas tras la aplicación del algoritmo. En este ejemplo resulta claro que **s** ha de ser igual a la suma de todos los elementos del vector, luego, en principio, una postcondición puede ser:

$$R \equiv s = \sum_{\alpha=1}^{N} v[\alpha]$$

¿ Se han de incluir otras condiciones en la postcondición ?. Para este algoritmo sería lógico pedir que no se altere el contenido del vector. Siguiendo el criterio expuesto en Peña Cap 2, **v** es una variable de entrada (no de entrada/salida) y no podrá ser alterada en ningún caso (en las declaraciones de tipo **fun** los parámetros son variables sólo de entrada, en las declaraciones de tipo **accion** pueden ser de entrada, salida o entrada/salida). Por tanto R puede ser una postcondición adecuada, en ella intervienen todas las variables involucradas en el problema: **v** y **s**, además ha sido necesario utilizar una variable auxiliar (variable ligada) α para recorrer simbólicamente los elementos del vector entre sus índices mínimo y máximo.

En la precondición se han de establecer las condiciones que han de cumplir los parámetros de entrada. En este caso el único parámetro es **v** y al indicar que ha de ser del tipo definido vect quedan claras las condiciones, por lo tanto la precondición puede ser:

$Q \equiv cierto$

La especificación completa queda entonces:

$\{ \; Q \equiv cierto \; \}$

fun suma(v: vect) **dev** s: entero

$\{ \; R \equiv s = \sum_{\boldsymbol{a}=1}^{N} v[\boldsymbol{a}] \; \}$

4. DISEÑO RECURSIVO POR INMERSIÓN NO FINAL

4.1. Diseño

Para obtener una solución recursiva es casi siempre necesario realizar una inmersión, desde luego resulta imprescindible cuando se opera sobre estructuras de datos agregadas como vectores. ¿Que es una inmersión?. Básicamente consiste en especificar un algoritmo en el que o bien se ha debilitado la postcondición (lo que da lugar a inmersión de tipo no final) o bien se ha fortificado la precondición (lo que da lugar a inmersión de tipo final). En cualquier caso el nuevo algoritmo ha de ser capaz de resolver el problema originalmente planteado para unos valores particulares de los parámetros de entrada. Es decir, se trata de un algoritmo más general, que para unos valores particulares de los parámetros de entrada resuelve el problema original. Para obtener inmersiones la técnica más adecuada es la consistente en manipular la postcondición del algoritmo original. Ésta era:

$$R \equiv s = \sum_{\boldsymbol{a}=1}^{N} v[\boldsymbol{a}]$$

Para generalizarla la técnica básica consiste en introducir nuevas variable en la postcondición, normalmente en sustitución de constantes. En este caso se puede introducir una variable **i** en sustitución de la constante **N** y se tiene:

$$R \equiv s = \sum_{\boldsymbol{a}=1}^{i} v[\boldsymbol{a}] \wedge i = N$$

La postcondición original R se puede expresar entonces como:

$$R \equiv \underline{R_1 \wedge R_2} \;\; \text{siendo} \;\; R_1 \equiv s = \sum_{\boldsymbol{a}=1}^{i} v[\boldsymbol{a}] \;\; \text{y} \;\; R_2 \equiv i = N$$

La técnica de inmersión no final consiste en utilizar un predicado más débil que la postcondición original como nueva postcondición. En otras palabras, en la inmersión no final se "pide menos" a la función, se verá posteriormente que en la inmersión final la postcondición original se mantiene pero se "pide más" en la precondición, es decir se fortalece la precondición.

Mientras en el algoritmo original la postcondición exigía que **s** devolviera la suma de todos los elementos del vector, en la nueva postcondición "sólo" se exige que **s** contenga la suma de los **i** primeros elementos. La variable **i** habrá de ser introducida como parámetro de la nueva función. Es decir la interfaz de la función inmersora es:

fun suma1(v: vect,i: natural) **dev** s: entero

La nueva postcondición es R_1, en la nueva precondición se han de incluir las condiciones que ha de cumplir la variable incorporada. Desde luego **i** ha de ser un índice válido del vector luego en principio la especificación de suma1 podría ser:

$$\{ \; Q_1 \equiv 1 \le i \le N \; \}$$

fun suma(v: vect ;i:natural) **dev** s: entero

$$\{ \; R_1 \equiv s = \sum_{\alpha=1}^{i} v[\alpha] \; \}$$

Pero se ha de tener en cuenta que siempre se ha de intentar conseguir la precondición más débil posible. ¿ Es posible ampliar el rango admisible para **i** ? ¿ Qué ocurre si se admite **i=0** ? En la postcondición el rango del sumatorio se anula, de hecho trivialmente se cumple que para **i=0**, **s=0**. De esta forma se tendría una solución trivial para el algoritmo: cuando **i=0** basta devolver **s=0**. La obtención de una solución trivial es un requisito indispendable en el diseño de un algoritmo recursivo. Todo algoritmo recursivo se basa en dividir los casos a tratar entre triviales y no triviales. En los no triviales se realizan ejecuciones recursivas del propio algoritmo pero con el problema "disminuido" de forma que se acerque a las soluciones de tipo trivial.

En el análisis por casos del algoritmo el caso trivial puede ser, por tanto: para **i=0**, **s=0**. Se puede entonces especificar la función admitiendo **i=0** y además esbozar la solución como:

$$\{ \; Q_1 \equiv 0 \le i \le N \; \}$$

```
fun suma1(v: vect ; i:natural) dev s: entero
        caso i=0 → 0
        ÿ    i>0 → ?
        fcaso
ffun
```

$$\{ \; R_1 \equiv s = \sum_{\alpha=1}^{i} v[\alpha] \; \}$$

Queda por resolver el caso no trivial. Para resolver este caso se han de seguir los siguientes pasos:

a) Realizar una llamada recursiva a la función que se está diseñando pero con tamaño de problema reducido

b) Se ha de analizar qué condiciones cumple el resultado de esa llamada: cumplirá la postcondición de la función para el valor de parámetros utilizados en la llamada.

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

Paso a): ¿ Que significa reducir el tamaño del problema ?. En todo algoritmo recursivo se ha de definir una estructura de preorden bien fundado sobre los parámetros. Los casos triviales del algoritmo recursivo corresponderán a los minimales de este preorden (ver Peña apartado 3.2). La forma práctica de obtener un preorden bien fundado es definir una función de los parámetros con imagen en los enteros no negativos (o un subconjunto de éstos). Esta función tomará un valor mínimo para los casos triviales y ha de decrecer estrictamente de valor en la llamada recursiva. Abreviando para este caso esa función puede ser **T(v,i)=i** , el caso trivial corresponde a **i=0** y en cada llamada se ha de acercar **T(v,i)** al valor de caso trivial, esto se consigue disminuyendo **i** en una unidad en la llamada. De esta forma se obtiene:

$\{ \; Q_1 \equiv 0 \le i \le N \; \}$

```
fun suma1(v: vect ; i:natural) dev s: entero
          caso i=0 → 0
           ÿ  i>0 →  sea s'= suma1(v,i-1)
                         en   ?
          fcaso
ffun
```

$\{ \; R_1 \equiv s = \sum_{a=1}^{i} v[a] \; \}$

? ahora es la operación adicional que se ha de realizar tras la llamada recursiva.

Paso b): Tras la llamada recursiva se obtiene un valor s' que cumplirá postcondición para v e i-1, es decir:

$s' = \sum_{a=1}^{i-1} v[a]$

Paso c): A partir de s' se ha de devolver un valor s que cumpla:

$s = \sum_{a=1}^{i} v[a]$

Es inmediato que ha de ser s=s'+a[i] luego el diseño queda:

$\{ \; Q_1 \equiv 0 \le i \le N \; \}$

```
fun suma1(v: vect ; i:natural) dev s: entero
          caso i=0 → 0
           ÿ  i>0 →  sea s'= suma1(v,i-1)
                         en   s'+v[i]
          fcaso
ffun
```

$\{ \; R_1 \equiv s = \sum_{a=1}^{i} v[a] \; \}$

que puede expresarse simplemente como:

$\{ \; Q_1 \equiv 0 \le i \le N \; \}$

```
fun suma1(v: vect ; i:natural) dev s: entero
        caso i=0 → 0
         ÿ  i>0 →  suma1(v,i-1)+a[i]
        fcaso
ffun
```

$\{ \; R_1 \equiv s = \sum_{\alpha=1}^{i} v[\alpha] \; \}$

Expresado en palabras: para sumar **i** elementos de un vector, si **i** es **0** basta devolver el valor **0**, si **i** mayor que **0** se utiliza recursivamente la propia función para sumar los **i-1** primeros elementos y a esa suma se le suma el elemento **a[i].**

Para obtener el mismo resultado con **suma1** que el pretendido originalmente del algoritmo **suma** basta con ejecutar **suma1** para i=N, es decir: suma(v)=suma1(v,N)

Obsérvese como funciona el algoritmo para el caso concreto N=3:

a) La llamada suma1(v,3) provoca una nueva llamada: suma1(v,2)

b) La llamada suma1(v,2) provoca una nueva llamada: suma1(v,1)

c) La llamada suma1(v,1) provoca una nueva llamada: suma1(v,0)

d) La llamada suma1(v,0) alcanza el caso trivial y retorna inmediatamente devolviendo el valor 0.

e) Antes de retornar de la llamada suma1(v,1) se suma el valor v[1] al valor de retorno de la llamada anterior, es decir a 0. Se devuelve el resultado de la suma, es decir v[1]

f) Antes de retornar de la llamada suma1(v,2) se suma el valor v[2] al valor de retorno de la llamada anterior, es decir a v[1]. Se devuelve el resultado de la suma, es decir v[1]+v[2].

g) Antes de retornar de la llamada suma1(v,3) se suma el valor v[3] al valor de retorno de la llamada anterior, es decir a v[1]+v[2]. Se devuelve el resultado de esa suma, es decir v[1]+v[2]+v[3]. Éste es el valor devuelto por la llamada inicial (suma1(v,3)) y es la suma de los elementos del vector.

Esta técnica de análisis de algoritmos aplicándolos a un ejemplo de tamaño reducido puede ser muy útil en una primera verificación.

4.2. Verificación

Tras el diseño se ha de verificar que el algoritmo es correcto. A veces la verificación descubre errores o simplemente mejoras potenciales y se ha de rediseñar, el proceso diseño-verificacón es un proceso iterativo que ha de conducir a un diseño correcto. Para comprobar la corrección de un algoritmo recursivo se han de verificar cada uno de los puntos siguientes (ver Tabla 3.2 en Peña):

1. **Completitud de la alternativa**: $Q(\overline{x}) \Rightarrow B_t(\overline{x}) \vee B_{nt}(\overline{x})$ .Se ha de comprobar que todos los parámetros que cumplen la precondición son contemplados en alguno de los casos triviales o no triviales. En este caso la concreción de Q, Bt y Bnt es $Q \equiv 0 \leq i \leq N \quad B_t \equiv i = 0 \quad B_{nt} \equiv i > 0$ y es inmediato que $0 \leq i \leq N \Rightarrow i = 0 \vee i > 0$ En palabras: para todos los valores de i admitidos por precondición existe un caso que los trata

2. **Satisfacción de la precondición para la llamada interna**: $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \Rightarrow Q(s(\overline{x}))$. En otras palabras se ha de comprobar que la llamada recursiva se realiza con parámetros que cumplen precondición. En el ejemplo tratado la función **s** que realiza la transformación de parámetros antes de la llamada recursiva es tal que transforma el par **s,i** en el par **s,i-1**, es decir **s(s,i)=s,i-1**. Sólo es necesario tener en cuenta **i** ya que es la única variable contemplada en la precondición y en la condición de caso no trivial. La precondición aplicada a **i** es $Q \equiv 0 \leq i \leq N$, aplicada a **i-1** es $Q_i^{i-1} \equiv 0 \leq i-1 \leq N$ y como $B_{nt} = i > 0$ se tiene: $0 \leq i \leq N \wedge i > 0 \equiv 0 < i \leq N \Rightarrow 0 \leq i-1 \leq N$ En palabras: los parámetros que se utilizan en la llamada interna cumplen la precondición general de la función. Obsérvese que esto es debido a que se garantiza que la llamada interna sólo se realiza si **i>0** y entonces se tiene que **i-1≥ 0**.

3. **Base de la inducción**: $Q(\overline{x}) \wedge B_t(\overline{x}) \Rightarrow R(\overline{x}, triv(\overline{x}))$. Se trata de comprobar que para el caso trivial y con parámetros que cumplan precondición la solución que se devuelve cumple postcondición. En el caso trivial es **i=0** y se devuelve **s=0** y es evidente que $0 = \sum_{\mathbf{a}=1}^{0} v[\mathbf{a}]$, ya que el dominio del sumatorio se anula, rigurosamente se tiene: $0 \leq i \leq N \wedge i = 0 \equiv i = 0 \Rightarrow 0 = \sum_{\mathbf{a}=1}^{0} v[\mathbf{a}]$. En palabras: en el caso trivial el valor devuelto ha de cumplir postcondición

4. **Paso de inducción**: $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \wedge R(s(\overline{x}), \overline{y}') \Rightarrow R(\overline{x}, c(\overline{y}', \overline{x}))$. Este suele ser el punto de verificación más complicado en el caso de inmersión no final y suele no entenderse bien. Básicamente y expresado en palabras quiere decir: para parámetros que cumplan precondición, y para caso no trivial, suponiendo que el valor devuelto por la llamada recursiva con parámetros modificados cumple postcondición entonces el valor devuelto por la función tras la operación adicional ha de cumplir también postcondición. En este ejemplo la llamada se realiza disminuyendo i, es decir la función s transforma **i** a **i-1**, y si suponiendo que dicha llamada cumple postcondición el valor devuelto es la suma de los **i-1** primer elementos del vector. En la operación adicional a dicho valor devuelto se le suma el elemento **v[i]**, entonces se obtiene la suma de los **i** primeros elementos del vector, que es la postcondición deseada. En forma rigurosa:
$$0 \leq i \leq N \wedge i > 0 \wedge s' = \sum_{\mathbf{a}=1}^{i-1} v[\mathbf{a}] \Rightarrow s = s' + v[i] = \sum_{\mathbf{a}=1}^{i-1} v[\mathbf{a}] + v[i] = \sum_{\mathbf{a}=1}^{i} v[\mathbf{a}]$$

5. **Elección de la estructura de preorden bien fundado**. Se trata de encontrar una función de los parámetros de entrada en los enteros no negativos. Si se toma la función: $t(v,i) = i$,es evidente que siempre toma valor no negativo puesto que por precondición $0 \leq i$, la función **t** define entonces un preorden bien fundado en el conjunto producto de vectores de tipo vect por naturales, éste es el conjunto dominio de los parámetros de entrada

6. **Demostración de decrecimiento de los datos**. Se ha de demostrar que en cada llamada recursiva la operación previa que se realiza sobre los parámetros de entrada hace decrecer estrictamente los datos en relación al preorden bien fundado previamente definido. De otra forma: que la función t(v,i) toma valores estrictamente decrecientes en cada llamada. Es importante que el decrecimiento sea estricto, en este ejemplo en la llamada recursiva se toma i disminuido en una unidad y es evidente que:

   $t(v,i-1)= i-1 < i=t(v,i)$

Un repaso a las condiciones que ha de cumplir un algoritmo recursivo para que sea correcto:

1) todos los valores de parámetros admitidos por precondición han de dar lugar a una operación contemplada en los casos triviales o no triviales, es decir, no es admisible que un valor de parámetro de entrada no satisfaga ninguna de las condiciones de los casos triviales o no triviales.

2) Para parámetros de entrada que cumplan precondición y además una condición de caso no trivial, la llamada recursiva se ha de realizar con valores de parámetros que a su vez cumplan precondición.

3) Para parámetros de entrada que cumplan precondición y condición de caso trivial el valor devuelto por el algoritmo ha de cumplir postcondición de forma trivial.

4) Para parámetros que cumplan precondición y además condición de caso no trivial, si se supone que la llamada recursiva devuelve valores que cumplen postcondición para el valor asignado en los parámetros en dicha llamada recursiva, entonces, tras la operación adicional, el valor devuelto ha de cumplir la postcondición de la función .

5) Ha de poder definirse un preorden bien fundado sobre los parámetros, la forma práctica consiste en definir una función de los parámetros en los enteros no negativos.

6) En cada llamada recursiva se ha de producir un decrecimiento estricto de los parámetros para el preorden bien fundado definido, o en forma equivalente, la función que determina el preorden bien fundado ha de disminuir estrictamente de valor en cada llamada.

Estos puntos de la demostración ha de llegar a comprenderse, dominarse y aprenderse de memoria. Se ha de ser capaz de expresarlos con palabras y con predicados.

4.3. Coste

Tras diseñar un algoritmo se ha de estudiar su coste. Para el caso de algoritmos recursivos el estudio de coste, al nivel de la asignatura, consiste en la "sabia" aplicación de las recurrencias 1.2 y 1.4 de Peña Cap 1, de soluciones 1.3 y 1.5 respectivamente. En este ejemplo la reducción de coste se realiza por resta de 1 al valor previo. La operación de caso trivial es de coste constante (consiste simplemente en devolver el valor cero) y la operación adicional es también de coste constante (una suma). La recurrencia 1.2 queda:

$$T(n) = \begin{cases} c & si\ n < 1 \\ T(n-1) + c & si\ n \geq 1 \end{cases}$$

Dicho en palabras: la suma de n elementos de un vector es una operación de coste constante para cero elementos y de coste suma del coste de sumar n-1 elementos más el de una operación de coste constante para n>1. Comparando esta recurrencia con la de 1.2 del texto se tiene para los parámetros a,b y k de dicha ecuación los valores 1, 1 y 0 respectivamente, por lo que la solución es (aplicando 1.3 de Peña):

$T(n) \in \theta(n)$. Es decir, se trata de un algoritmo de coste lineal.

5. DISEÑO RECURSIVO POR INMERSIÓN FINAL

5.1. Diseño

En el apartado anterior se ha obtenido una solución recursiva no final, es decir tras la llamada recursiva es necesario realizar una operación adicional antes de devolver el resultado. A partir de la solución no final puede obtenerse una solución final aplicando la técnica de plegado y desplegado, cuando es posible. En este apartado, sin embargo, se tratará de hallar directamente la solución recursiva final. La especificación de la función a diseñar era:

$\{\ Q \equiv cierto\ \}$

fun suma(v: vect) **dev** s: entero

$\{\ R \equiv s = \sum_{\alpha=1}^{N} v[\alpha]\ \}$

y la postcondición podía expresarse como:

$R \equiv \underline{R_1 \wedge R_2}$ siendo $R_1 \equiv s = \sum_{\alpha=1}^{i} v[\alpha]$ y $R_2 \equiv i = N$

En la inmersión no final del apartado anterior se debilitaba la postcondición, se tomaba como nueva postcondición el predicado R_1, más debil que el original R. Para obtener una inmersión de tipo final la técnica consiste en mantener la misma postcondición R pero fortalecer la precondición. En la precondición se pedirá que parte del trabajo "se dé hecho" básicamente consiste en incluir el predicado R_1 para una nueva variable **r** (en sustitución de la variable de salida **s**) e **i**. O expresado en palabras pedir que en un parámetro adicional **r** se pase la suma de los **i** primeros elementos del vector. La nueva especificación es:

$\{\ Q_2 \equiv 0 \le i \le N \wedge r = \sum_{\alpha=1}^{i} v[\alpha]\ \}$

fun suma2(v: vect; r: entero; i: natural) **dev** s: entero

$\{\ R \equiv s = \sum_{\alpha=1}^{N} v[\alpha]\ \}$

Obsérvese que la postcondición se mantiene. Sin embargo se han incluido dos parámetros adicionales **r** e **i**. En la precondición se pide que **r** contenga la suma de los **i** primeros elementos del vector. ¿ Cómo utilizar esta función para resolver el problema original ? Basta realizar la llamada con **i=0** y **r=0** ya que en este caso se cumple precondición trivialmente (en **r** se pasa la suma de los **0** primeros elementos del vector que es **0**). Es decir:

suma(v)=suma2(v,0,0)

En el análisis por casos el primer paso es buscar casos triviales. Si la llamada se hace con **i=N** se tiene que **r** contendrá la suma de los **N** elementos del vector, pero esto es lo exigido en la postcondición. Luego en este caso bastará con devolver **r** como solución. Si **i<N** se tendrán casos no triviales. Un primer esbozo de la solución es, por tanto,:

$\{ \; Q_2 \equiv 0 \le i \le N \wedge r = \sum_{\mathbf{a}=1}^{i} v[\mathbf{a}] \; \}$

fun suma2(v: vect; r: entero; i: natural) **dev** s: entero
caso i=N → r
ÿ i<N → suma2(v,r',i')
fcaso
ffun

$\{ \; R \equiv s = \sum_{\mathbf{a}=1}^{N} v[\mathbf{a}] \; \}$

Se ha tenido en cuenta que en el caso no trivial no habrá operaciones adicionales tras la llamada recursiva puesto que se trata de una recursividad final. En este diseño el punto crítico del diseño es la elección de los parámetros de la llamada recursiva **r'** e **i'** (v sigue siendo el vector original). Se han de tomar de forma que se cumplan dos condiciones: a) que la llamada se haga cumpliendo precondición, b) que se disminuya el tamaño del problema por aproximación de los parámetros a los valores de caso trivial.

Para cumplir b) y puesto que el caso trivial corresponde a **i=N** se puede incrementar **i**, es decir **i'=i+1**.

Para cumplir a) se ha de tener en **r'** la suma de los **i'=i+1** primeros elementos del vector, esto se consigue haciendo **i'=r+v[i+1]**, ya que en **r** se tenía la suma de los **i** primeros elementos del vector. El diseño de suma2 queda entonces:

$\{ \; Q_2 \equiv 0 \le i \le N \wedge r = \sum_{\mathbf{a}=1}^{i} v[\mathbf{a}] \; \}$

fun suma2(v: vect; r: entero; i: natural) **dev** s: entero
caso i=N → r
ÿ i<N → suma2(v,r+v[i+1],i+1)
fcaso

ffun

$\{ \; R \equiv s = \sum_{\mathbf{a}=1}^{N} v[\mathbf{a}] \; \}$

Obsérvese como funciona el algoritmo para el caso concreto N=3:

a) La llamada suma2(v,0,0) provoca una nueva llamada: suma2(v,1,r) con r=v[1]

b) La llamada suma2(v,1,r) provoca una nueva llamada: suma2(v,2,r') con r'=v[1]+v[2]

c) La llamada suma2(v,2,r) provoca una nueva llamada: suma2(v,3,r'') con r''=v[1]+v[2]+v[3]. En esta llamada se alcanza el caso trivial (i=N) y se devuelve r'' como valor de retorno

d) La llamada suma2(v,2,r’) devuelve el valor de retorno que le ha devuelto la llamada anterior, es decir r’’

e) La llamada suma1(v,1,r) devuelve el valor de retorno que le ha devuelto la llamada anterior, es decir r’’

f) La llamada suma1(v,0,0) devuelve el valor de retorno que le ha devuelto la llamada anterior, es decir r’’=v[1]+v[2]+v[3] que es la suma de los elementos del vector.

Obsérvese que, mientras en la inmersión no final el problema se va resolviendo en la operación adicional tras la llamada recursiva, en este caso el problema se va resolviendo en la operación previa a la llamada recursiva necesaria para cumplir la postcondición.

5.2. Verificación

Se realizarán los pasos de la verificación sin extenderse en repetir explicaciones ya dadas en el apartado correspondiente a la solución no final. Obsérvese que si en el caso de inmersión no final el punto más complicado de la demostración suele ser el 4 (paso de inducción) en este caso de inmersión final el punto más complejo suele ser el 2 (demostrar que se satisface precondición en la llamada interna)

1. **Completitud de la alternativa**: $Q(\overline{x}) \Rightarrow B_t(\overline{x}) \vee B_{nt}(\overline{x})$ .Se ha de comprobar que todos los parámetros que cumplen la son contemplados en alguno de los casos triviales o no triviales. En este caso la concreción de Q, Bt y Bnt es $Q \equiv 0 \leq i \leq N \wedge r = \sum_{\alpha=1}^{i} v[i];\quad B_t \equiv i = N \quad B_{nt} \equiv i < N$ y es inmediato que $Q \Rightarrow i = N \vee i < N$

2. **Satisfacción de la precondición para la llamada interna**: $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \Rightarrow Q(s(\overline{x}))$. Este es el punto más importante a verificar en el caso de solución recursiva final. En este ejemplo queda: $Q \equiv 0 \leq i \leq N \wedge r = \sum_{\alpha=1}^{i} v[i] \wedge i < N \Rightarrow 0 \leq i+1 \leq N \wedge r + v[i] = \sum_{\alpha=1}^{i+1} v[i]$. Expresado en palabras: si en r se tiene la suma de los i primeros elementos del vector y se tiene que i<N, en la llamada ejecutada con i+1 en lugar de i se cumple que el tercer parámetro contendrá la suma de los i+1 primer elementos del vector.

3. **Base de la inducción**: $Q(\overline{x}) \wedge B_t(\overline{x}) \Rightarrow R(\overline{x}, triv(\overline{x}))$. Se trata de comprobar que para el caso trivial y con parámetros que cumplan precondición la solución que se devuelve cumple postcondición. En el caso trivial es i=N y se devuelve r directamente. En este caso por precondición r contiene la suma de los i=N elementos del vector. Luego se cumple la postcondición

4. **Paso de inducción**: $Q(\overline{x}) \wedge B_{nt}(\overline{x}) \wedge R(s(\overline{x}), \overline{y}') \Rightarrow R(\overline{x}, c(\overline{y}', \overline{x}))$. En el caso de recursividad final aquí prácticamente no hay nada que demostrar: si la llamada interna devuelve un valor que cumple postcondición y puesto que esta postcondición no depende de las variables que varían en la llamada (i y r) y además es ese valor el que directamente devuelve la función, se tendrá que cumple postcondición. Para este ejemplo: el valor devuelto en la llamada interna es la suma de los N elementos del vector y ese valor es el que se devuelve y por tanto el valor devuelto cumple postcondición (contiene la suma de los N elementos del vector)

5. **Elección de la estructura de preorden bien fundado**. Se trata de encontrar una función de los parámetros de entrada en los enteros no negativos. Si se toma la función: $t(v,i,r) = N - i$ es evidente que siempre toma valor no negativo puesto que por precondición $i \leq N$, la función **t** define entonces un preorden bien fundado en el conjunto producto de vectores de tipo vect por naturales y por enteros, éste es el conjunto dominio de los parámetros de entrada. Obsérvese que ahora los minimales corresponden a las triplas **(v,i,r)** en las que **i=N**, ya que en este caso **t(v,i,r)** toma el valor **0**.

6. **Demostración de decrecimiento de los datos**. Se ha de demostrar que en cada llamada recursiva la operación previa que se realiza sobre los parámetros de entrada hace decrecer estrictamente los datos en relación al preorden bien fundado previamente definido. De otra forma: que la función **t(v,i,r)** toma valores estrictamente decrecientes en cada llamada. Es importante que el decrecimiento sea estricto, en este ejemplo en la llamada recursiva se toma **i** incrementado en una unidad y a **r** se le suma **v[i+1]**, es evidente que:

   t(v,i+1,r+v[i+1])= N-(i+1) < N-i=t(v,i,r)

Este ejemplo demuestra que "disminuir el tamaño del problema" puede consistir en incrementar el valor de un parámetro. **Lo importante es el hecho de que los parámetros se aproximen a los valores de caso trivial**.

6. TRANSFORMACIÓN POR PLEGADO-DESPLEGADO

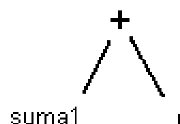
La técnica de plegado-desplegado permite, cuando es posible aplicarla, obtener una solución recursiva final a partir de una no final. Se trata de un proceso bastante automático que se comprenderá fácilmente tras realizar varios ejemplos. En este ejercicio la solución recursiva no final era:

{ $Q_1 \equiv 0 \leq i \leq N$ }

fun suma1(v: vect ; i:natural) **dev** s: entero
    caso i=0 → 0
    ÿ i>0 → suma1(v,i-1)+v[i]
    fcaso
ffun

{ $R_1 \equiv s = \sum_{\alpha=1}^{i} v[\alpha]$ }

En primer lugar se ha de dibujar el árbol sintáctico de la operación adicional, en este ejemplo se trata de una suma, en ella se mantendrá el sumando correspondiente a la llamada de función y el otro sumando se sustituirá por una variable adicional r. El árbol sintáctico en este ejemplo es, pues:



De aquí se deduce que la nueva función, que se llamará suma3 será de la forma: **suma3(v,i,r)= suma1(v,i)+r**

Para que pueda aplicarse la técnica de plegado-desplegado la operación que enlaza **suma1** con la variable adicional ha de tener elemento neutro y ser asociativa. La suma tiene elemento neutro (el 0) y es asociativa, por lo tanto es posible aplicar la técnica en este caso. La llamada a realizar en suma3 para que resuelva el problema que resolvía suma1 será la que utilice como valor de la variable adicional el neutro de la operación, es decir **suma1(v,N)=suma3(v,N,0).** Para desarrollar el plegado-desplegado basta seguir una serie de pasos bastante automáticos:

1) Sustituir en **suma3(v,i,r)=suma1(v,i)+r** la función suma1 por su desarrollo, es decir:

$$suma3(v,i,r)=\left\{\begin{array}{c} caso\, i=0 \rightarrow 0 \\ caso\, i>0 \rightarrow suma1(v,i-1)+v[i] \end{array}\right\}+r$$

2) Trasladar la operación adicional a cada uno de los casos (desplegado):

$$suma3(v,i,r)=\left\{\begin{array}{c} caso\, i=0 \rightarrow 0+r \\ caso\, i>0 \rightarrow (suma1(v,i-1)+v[i])+r \end{array}\right\}$$

3) Aplicando la propiedad de neutro en el caso trivial y la asociativa en el no trivial transformar a:

$$suma3(v,i,r)=\left\{\begin{array}{c} caso\, i=0 \rightarrow r \\ caso\, i>0 \rightarrow suma1(v,i-1)+(v[i]+r) \end{array}\right\}$$

4) Teniendo en cuenta que la definición de suma3 es **suma3(v,i,x)=suma1(v,i)+x**, se puede sustituir la expresión de caso no trivial por su equivalente en términos de suma3 (operación de plegado), aquí x es (v[i]+r):

$$suma3(v,i,r)=\left\{\begin{array}{c} caso\, i=0 \rightarrow r \\ caso\, i>0 \rightarrow suma3(v,i-1,v[i]+r) \end{array}\right\}$$

Y se ha obtenido una solución recursiva final para el problema, ésta solución es:

```
fun suma3(v: vect ; i:natural; r:entero) dev s: entero
        caso i=0 → r
        ÿ  i>0 →  suma3(v,i-1,r+v[i])

ffun
```

La postcondición de esta función será la de suma1 para i=N es decir:

$$R_3 \equiv s=\sum_{\alpha=1}^{N} v[\alpha]$$

En la precondición se han de incluir condiciones adecuadas para r, éstas condiciones pueden deducirse el hecho de que suma3(v,i,r)=suma1(v,i)+r que implica r=suma3(v,i,r)-suma1(v,i) y sustituyendo en suma3 y suma1 por los predicados de sus respectivas postcondiciones:

$$r = \sum_{a=1}^{N} v[a] - \sum_{a=1}^{i} v[a] = \sum_{a=i+1}^{N} v[a]$$

La precondición será entonces:

$$Q_3 \equiv 0 \leq i \leq N \wedge r = \sum_{a=i+1}^{N} v[a]$$

r contendrá en cada llamada la suma de los N-(i+1) últimos elementos del vector, en la llamada inicial suma3(v,N,0) se cumple trivialmente ya que al ser **i+1=N+1>N** el dominio del sumatorio se anula y básta pasar **0** como valor de **r**.

Obsérvese que ésta es una solución recursiva final distinta a la obtenida por desarrollo directo. Se podría obtener ésta misma solución por desarrollo directo si en lugar de generalizar la postcondición de la función suma() original por sustitución de la constante N se hubiera generalizado por inclusión de la variable adicional en la definición del inicio del sumatorio, es decir si la postcondición R original se hubiera expresado como:

$$R \equiv \underline{R_1 \wedge R_2} \text{ siendo } R_1 \equiv s = \sum_{a=i+1}^{N} v[a] \text{ y } R_2 \equiv i = 0$$

Puede realizarse este desarrollo como ejercicio. Se deja también como ejercicio la verificación de suma3 y la aplicación sobre un ejemplo de tamaño reducido en forma análoga a como se ha hecho en los apartados anteriores. Como puede comprobarse, para un algoritmo dado, existen múltiples soluciones recursivas obtenibles por inmersión , en función de cuántas y cómo se introduzcan las variables adicionales en la postcondición.

7. DISEÑO ITERATIVO DIRECTO

7.1. Diseño

El diseño iterativo se inicia de la mismo forma que el recursivo: generalizando la postcondición. Siguiendo el mismo proceso que en el diseño recursivo se tiene:

$$R \equiv \underline{R_1 \wedge R_2} \text{ siendo } R_1 \equiv s = \sum_{a=1}^{i} v[a] \text{ y } R_2 \equiv i = N$$

Pero ahora los términos R1 y R2 se utiliza de otra forma: uno de ellos servirá, junto con condiciones para la variable auxiliar i, como invariante del bucle, el otro, negado, como condición de finalización del bucle, si se toma P=R1 como invariante y R2 negado como condición de finalización se tiene un primer diseño dado por:

```
fun sumaiter(v:vect) dev s: entero
    Inicializar;
    mientras i≠N hacer { P ≡ s = Σ_{a=1}^{i} v[a] ∧ 0 ≤ i ≤ N }
        Restaurar;
        Avanzar;
    fmientras
ffun
```

$\{ \ R \equiv s = \sum_{\alpha=1}^{N} v[\alpha] \ \}$

Se han de diseñar los términos *Inicializar, Avanzar* y *Restaurar*

El término Inicializar supone declarar las dos variables implicadas en el invariante: **s** e **i**, y asignarles valores que hagan que antes del iniciar el bucle se cumpla ya el predicado invariante P. Esto se consigue fácilmente asignando a **s** el valor 0 y a **i** también el valor 0, la inicialización queda, por tanto:

```
var i: natura, s: entero
i:=0;
s:=0;
```

En el término *Avanzar* se han de modificar las variables necesarias para que se acerque el bucle a su condición de finalización, evidentemente, y como se ha partido de **i=0** y la condición de finalización es **i=N**, esto se consigue incrementando **i**, por ejemplo, así *Avanzar* queda:

```
i:=i+1;
```

Y, por fin, el que resulta ser el paso más complicado, normalmente, del diseño: la operación Restaurar. ¿ En qué consiste ?. Tras realizar la operación Avanzar se ha de cumplir el invariante para el nuevo valor de **i**, es decir se ha de cumplir que $P_i^{i+1} \equiv s = \sum_{\alpha=1}^{i+1} v[\alpha] \wedge 0 \le i+1 \le N$ es cierto, o sea **s** contiene la suma de los **i+1** primeros elementos del vector. Antes de la operación restaurar se cumple el invariante para **i**, es decir **s** contiene la suma de los **i** primeros elementos del vector, la operación Restaurar ha de hacer que el invariante que era cierto para **i** pase a ser cierto para **i+1**. En este caso se deduce que dicha operación ha de consistir en sumar a **s** el término **v[i+1]**, de forma que **s** vuelva a contener la suma de los **i** primeros elementos del vector para el nuevo valor de **i**. En forma rigurosa la deducción consiste deducir cual es la operación Restablecer sabiendo que tras dicha operación se ha de cumplir el invariante para **i+1** y antes se cumple para **i,** para el término fundamental del invariante (la condición para **i+1** viene garantizada por la condición de bucle) se tiene:

$\{ P1 \equiv s = \sum_{\alpha=1}^{i} v[\alpha] \}$

Restablecer

$\{ P1_i^{i+1} \equiv s = \sum_{\alpha=1}^{i+1} v[\alpha] \}$

Puesto que se tiene: $P1_i^{i+1} \equiv s = \sum_{\alpha=1}^{i+1} v[\alpha] = \sum_{\alpha=1}^{i} v[\alpha] + v[i+1]$

es evidente que a **s** se le habrá de sumar el término **v[i+1]**, es decir Restablecer es:

s:=s+v[i+1];

con lo que el diseño queda, finalmente:

{Q≡Cierto}
fun sumaiter(v:vect) dev s: entero
    var s:entero;i:natural;
    s:=0;
    i:=0;
    mientras i≠N hacer $\{ P \equiv s = \sum_{\alpha=1}^{i} v[\alpha] \wedge 0 \leq i \leq N \}$
        s:=s+v[i+1];
        i:=i+1;
    fmientras
ffun

$\{ R \equiv s = \sum_{\alpha=1}^{N} v[\alpha] \}$

Es evidente que a poco experiencia que se tenga en programación este bucle se podía haber diseñado "a sentimiento" sin seguir un proceso riguroso como el expuesto, pero es necesario el entrenamiento en la técnica para cuando haya de abordarse ejercicios no tan triviales.

Es interesante señalar el paralelismo entre diseño recursivo final e iterativo:

En ambos casos se reescribe la postocondición como una conjunción de predicados incluyendo una nueva variable: i

En el diseño recursivo se toma uno de los dos predicados (R1) como condición adicional en la precondición, en el iterativo como invariante.

El otro predicado (R2) se toma en el caso recursivo como condición de caso trivial y en el iterativo su negación se toma como condición de bucle, es decir R2 es la condición de finalización de bucle.

En el caso recursivo cuando se cumple R2 como además se cumple R1 por precondición, se cumplirá la conjunción de R1 y R2, es decir la postcondición.

En el caso iterativo cuando se cumple R2 (fin de bucle) como además se cumple siempre R1 (invariante) se cumplirá la conjunción de ambos (postcondición).

En el caso recursivo final la llamada inicial hace que se cumpla trivialmente la precondición (suma2(v,0,0)) ya que la suma de cero elementos del vector es cero.

En el caso iterativo la inicialización asigna a s y a i el valor 0 y por tanto hace que se cumpla trivialmente el invariante.

7.2. Verificación

En la verificación de un algoritmo iterativo se han de comprobar cada uno de los puntos de la Figura 4.2 de Peña Cap 4. En concreto:

0) Fijar invariante y función limitadora. En este caso el invariante se ha obtenido en el proceso de diseño. La función limitadora ha de ser una función que tome valores no negativos y que decrezca estrictamente en cada paso del bucle. Para este caso una función apropiada es: t(v,s,i)=N-i (obsérvese el paralelismo con la función que determina el preorden bien fundado en el caso recursivo)

1) Verificar que si se cumple invariante y condición de fin de bucle entonces se cumple postcondición ($P \wedge \neg B \Rightarrow R$). Aquí B es el predicado inverso de R2. Evidentemente se tiene: $s = \sum_{\mathbf{a}=1}^{i} v[\mathbf{a}\} \wedge i = N \Rightarrow s = \sum_{\mathbf{a}=1}^{N} v[\mathbf{a}]$

2) Verificar que tras inicialización (y teniendo en cuenta la precondición) se cumple el invariante ({Q} Inic {P}). En este caso la inicialización consiste en asignar tanto a s como a i el valor 0 y se cumple: $0 = \sum_{\mathbf{a}=1}^{0} v[\mathbf{a}]$, ya que el dominio del sumatorio se anula.

3) Verificar que, teniendo en cuenta que a la entrada del bucle se cumple el invariante y la condición de bucle, tras las operaciones interiores al bucle se ha de cumplir el invariante para los nuevos valores de las variables. La condición 0≤i+1≤N se cumple por la condición de bucle que garantiza que i<N. Puesto que en el bucle se asigna a s el valor s+v[i+1] y a i el valor i+1 se ha de cumplir: $P_{i,s}^{i+1,s+v[i+1]} \equiv s + v[i+1] = \sum_{\mathbf{a}=1}^{i+1} v[\mathbf{a}]$. Esto es evidentemente cierto. Antes de las operaciones interiores **s** contiene la suma de los **i** primeros elementos del vector. Tras las operaciones interiores **i** pasa a valer **i+1** y **s** pasa a contener la suma de los **i+1** primeros elementos del vector.

4) Verificar que la función limitadora se mantiene no negativa durante la ejecución del bucle. Esto es inmediato: **N-i** se mantiene no negativo pues durante el bucle **i** es menor o igual que **N**

5) Verificar que en cada paso de bucle la función limitadora decrece estrictamente, esto es también inmediato pues **N-(i+1)<N-i**

Al igual que en el caso recursivo estos puntos de la verificación han de aprenderse de memoria y llegar a dominar su aplicación con predicados.

7.3. Coste

En el caso de los algoritmos iterativos la técnica a emplear para determinar el conste asintótico es la aplicación de las reglas prácticas para cálculo de eficiencia explicadas en Peña Ap 1.4.

El coste de las operaciones internas al bucle es el coste de dos operaciones consecutivas de coste constante (dos sumas) y por tanto es de coste constante. En otras palabras el coste de las operaciones S internas al bucle es:

$$\Theta s = \Theta(1) + \Theta(1) = \Theta(1+1) = \Theta(max(1,1)) = \Theta(1)$$

El número de iteraciones depende de N, y por la regla del producto, el coste total es:

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

8. TRANSFORMACIÓN RECURSIVO-ITERATIVA

En este apartado se realizará la transformación de las soluciones recursivas final y no final obtenidas en los apartados anteriores a iterativa. Para ello se aplicarán las equivalencias expuestas en Peña fig 3.14 y Peña fig. 3.15 respectivamente.

8.1. Transformación de recursivad final

La solución recursiva final era:

$$\{\ Q_2 \equiv 0 \le i \le N \wedge r = \sum_{\mathbf{a}=1}^{i} v[\mathbf{a}]\ \}$$

```
fun suma2(v:  vect; r: entero; i: natural) dev s: entero
        caso i=N → r
        ÿ    i<N →  suma2( v,r+v[i+1],i+1)
        fcaso
ffun
```

$$\{\ R \equiv s = \sum_{\mathbf{a}=1}^{N} v[\mathbf{a}]\ \}$$

y se ejecutaba con llamada inicial suma2(v,0,0)

Su equivalente iterativa dará lugar a un bucle, en el que:

a) La inicialización de variables consistirá en asignar a las variables involucradas los mismos valores que se utilizan en la llamada inicial en la función recursiva. (i:=0; r:=0 para este ejemplo).

b) La condición de bucle será la de caso no trivial en el algoritmo recursivo (i<N en el ejemplo)

c) Se devolverá tras el bucle el mismo valor que se devolvía en el caso trivial en el algoritmo recursivo (r en el ejemplo)

d) 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: $P(\bar{x},\bar{xini}) \equiv Q(\bar{x}) \wedge f(\bar{xini}) = f(\bar{x})$. En este caso: $P(r,i,0,0) \equiv 0 \le i \le N \wedge r = \sum_{a=1}^{i} v[a] \wedge \quad suma2(v,0,0) = suma2(v,i,r)$ pero los predicados suma2(v,0,0) y suma2(v,i,r) son iguales ya que se trata de la postcondición de suma2 por tanto se puede eliminar de ambos miembros de la igualdad obteniéndose: $P(r,i,0,0) \equiv 0 \le i \le N \wedge r = \sum_{a=1}^{i} v[a] = cierto$, en forma más simple y puesto que el invariante es un predicado que se presupone ha de tomar el valor cierto al inicio de cada vuelta de bucle, se tiene que el invariante es: $P \equiv 0 \le i \le N \wedge r = \sum_{a=1}^{i} v[a]$. Obsérvese la técnica para obtener el invariante del bucle equivalente a un algoritmo recursivo final: escribir el predicado dado por la igualdad entre: a) la conjunción formada por la precondición y la postcondición para valores iniciales y b) la postcondición para valores generícos (ver Peña ecuación 3.19). En la práctica conduce al predicado dado por la precondición de la función recursiva ya que la postcondición del algoritmo recursivo no depende de los parámetros

e) La precondición de la función iterativa será el predicado utilizado en la precondición de la recursiva en el que se sustituyan las variables genéricas por sus valores iniciales es decir: $Q_2(\bar{xini}) = 0 \le i \le N \wedge r = \sum_{a=1}^{i} v[a] \Big|_{0,0}^{i,r} \equiv 0 \le 0 \le N \wedge 0 = \sum_{a=1}^{0} v[a] \equiv cierto$, luego el predicado cierto es la precondición del algoritmo iterativo.

A partir de los puntos anteriores se puede escribir el bucle iterativo como:

```
{ cierto }
fun sumaiter1(v: vect) dev s:entero
        var i:natural; r:entero;
        i:=0;
        r:=0;
        mientras i<N hacer { P ≡ 0 ≤ i ≤ N ∧ r = Σ_{a=1}^{i} v[a] }
                r:=r+v[i+1];
                i:=i+1;
        fmientras
        dev r;
ffun
```

$\{ \ R \equiv s = \sum_{a=1}^{N} v[a] \ \}$

8.2. *Transformación de recursividad no final*

La solución recursiva no final era:

$\{ \ Q_1 \equiv 0 \leq i \leq N \ \}$
fun suma1(v: vect ; i:natural) **dev** s: entero
    caso i=0 → 0
    ÿ i>0 → suma1(v,i-1)+a[i]

ffun
$\{ \ R_1 \equiv s = \sum_{a=1}^{i} v[a] \ \}$

Su transformación a algoritmo iterativo equivalente dará lugar a dos bucles. Antes se inicializan las variables con los mismos valores utilizados en la la llamada recursiva, en este caso la única variable a inicializar es i (aunque siempre habrán de declararse variables para utilizar en la devolución de resultados, en este caso s). La inicialización queda (la llamada recursiva era suma1(v,N)):

i:=N;

El primer bucle consiste en realizar la operación de modificación de parámetros realizada en la llamada recursiva utilizando como condición de bucle la de caso no trivial de la función recursiva, es decir:

mientras i>0 **hacer**
    i:=i-1;
fmientras

Antes del segundo bucle se ha de inicializar la variable que contenga finalmente el resultado con el valor de caso trivial de la función recursiva. En este caso:

s:=0;

El segundo bucle consiste en aplicar sucesivamente:

a) La función inversa de la aplicada en la modificación de parámetros en la llamada recursiva, en este caso la operación es la que asigna a i el valor i-1, y su inversa será la que asigne a i el valor i+1 ya que (i-1)+1=i.

b) La operación adicional del algoritmo recursivo sobre las variables de salida y entrada. En este caso consiste en sumar a s el valor v[i].

El bucle finaliza cuando las variables alcanzan de nuevo sus valores iniciales, en este caso cuando i alcanza de nuevo el valor N.

```
 mientras i≠ N hacer
          i:=i+1;
          s:=s+v[i];
 fmientras
```

Finalmente se devolverá el resultado en s.

El algoritmo iterativo completo queda:

```
{ cierto }
fun sumaiter1(v: vect) dev s:entero
          var i:natural; s:entero;
          i:=N;
          mientras i>0 hacer {P1}
                    i:=i-1;
          fmientras
          s:=0;
          mientras i≠ N hacer {P2
                    i:=i+1;
                    s:=s+v[i];
          fmientras
          dev s;
ffun
```

$$\{\ R \equiv s = \sum_{\alpha=1}^{N} v[\alpha]\ \}$$

Obsérvese que la precondición y postcondición son las mismas del algoritmo recursivo.

En este ejemplo es evidente que el primer bucle puede ser eliminado asignando simplemente a i el valor que toma al finalizar el bucle, es decir i:=0. Esto ocurrirá en todos los casos en los que la operación de modificación de parámetros en la llamada recursiva sea muy simple.

En este caso existe operación inversa de la aplicada en la modificación de parámetros en la llamada recursiva, pero esto no siempre ocurre. En algunos casos la función inversa no puede expresarse por una operación matemática. En estos casos los valores obtenidos en el primer bucle en cada operación de modificación han de ser acumulados en una estructura tipo pila. En el segundo bucle la operación inversa se sustituye por la extracción de esos valores desde la pila (Fig 3.16 Peña).

Quedan por establecer los invariantes P1 y P2 de los bucles. P1 es conceptualmente complejo. Se obtiene a partir de la expresión 3.20 de Peña como:

$$P_1(i,N) \equiv 0 \le i \le N \wedge \exists k \in N.(i = N - k) \wedge \forall k' \in \{0..k-1\}.N - k' > 0 \wedge 0 \le N - k' \le N$$

Expresado en palabras: i se mantiene mayor o igual que 0 y menor o igual que N, además existe un natural k tal que i es igual al resultado de de aplicar k veces la operación interna del bucle a i (en este caso restar k veces 1 o sea restar k) y además para todo valor k' entre 0 y k-1 el resultado de aplicar la operación interna de bucle al valor inicial N k' veces ha de cumplir la condición de caso no trivial del algoritmo

recursivo, y además ese valor ha de cumplir precondición del algoritmo recursivo. Conviene repasar este ejemplo siguiendo la explicación del texto

El invariante P2 es aún más complejo ya que es la conjución de P1 con la postcondición para los valores genéricos de las variables. En este caso:

$$P_2(i, N, s) \equiv P_1(i, N) \wedge s = \sum_{\boldsymbol{a}=1}^{i} v[\boldsymbol{a}]$$

9. CODIFICACIÓN EN MODULA 2

Se presentan a continuación los listados correspondientes a la codificación en MODULA-2 de algunos de los algoritmos recursivos e iterativos desarrollados. Se incluye un programa principal que permite realizar ejecuciones para distintos tamaños de problema evaluando el tiempo de ejecución.

Listado 1. Codificación de los algoritmos recursivos e iterativos desarrollados

```
(* Ejercicio1 de guia para prácticas de Programación II  Curso 96/97 *)

MODULE Ejercicio1;

FROM InOut  IMPORT  WriteString, WriteCard, WriteLn, Read;
FROM TimeDate IMPORT Time,GetTime;

CONST
   N = 100;

TYPE
   vect = ARRAY [1..N] OF INTEGER;

VAR
   VPrueba :vect;

PROCEDURE SumaRecur1(VAR v:vect;i: CARDINAL): INTEGER;
 BEGIN
   IF i=0 THEN
      RETURN 0
   ELSE
      RETURN v[i]+SumaRecur1(v,i-1)
   END;
 END SumaRecur1;

PROCEDURE SumaRecur2(VAR v:vect;i: CARDINAL;r: INTEGER): INTEGER;
 BEGIN
   IF i=N THEN
      RETURN r
   ELSE
      RETURN SumaRecur2(v,i+1,r+v[i+1])
   END;
 END SumaRecur2;

PROCEDURE SumaIter(VAR v:vect): INTEGER;
 VAR i:CARDINAL;s:INTEGER;
 BEGIN
   i:=0;
   s:=0;
   WHILE i<N DO
     s:=s+v[i+1];
     i:=i+1;
   END;
   RETURN s
 END SumaIter;

PROCEDURE SumaIter1(VAR v:vect;n: CARDINAL): INTEGER;
 VAR i:CARDINAL;s:INTEGER;
 BEGIN
   i:=0;
   s:=0;
   WHILE i<n DO
     s:=s+v[i+1];
     i:=i+1;
   END;
   RETURN s
 END SumaIter1;
```

```
(* PROGRAMA PRINCIPAL *)

VAR i,k,t: CARDINAL; t1,t2: Time;s:INTEGER;c: CHAR;

BEGIN
 (* Se rellena primero el vector con valores apropiados *)

 FOR i:=1 TO N DO
    VPrueba[i]:=i;
 END;

 (* Se verifica el funcionamiento correcto de cada una de las funciones, el
  resultado ha de valer N(N+1)/2= 100*101/2= 5050 *)

 WriteLn;
 WriteString("Resultado de SumaRecur1:");
 WriteCard(SumaRecur1(VPrueba,N),5);

 WriteLn;
 WriteString("Resultado de SumaRecur2:");
 WriteCard(SumaRecur2(VPrueba,0,0),5);

 WriteLn;
 WriteString("Resultado de SumaIter:");
 WriteCard(SumaIter(VPrueba),5);
(* Se generan tablas con medidas de tiempo de ejecución para distintos tamaños
  de problema, puesto que el tiempo de ejecución es corto se mide el tiempo para
  la repetición de 2000 operaciones de suma, para poder hacer llamadas
  a la función iterativa con distintos tamaños de problema se utiliza una
  versión modificada (SumaIter1) que permite indicar el número de elementos
  del vector a sumar *)
 WriteLn;WriteLn;
 WriteString(" TIEMPOS DE EJECUCION PARA SumaRecur1");
 WriteLn;
 WriteString("   i      t (dec. s)");
 WriteLn;
 WriteString("---------------");
 WriteLn;
 FOR i:=0 TO N BY 20 DO
   GetTime(t1);
   FOR k:=1 TO 10000 DO
      s:=SumaRecur1(VPrueba,i);
   END;
   GetTime(t2);
   IF t2.millisec>= t1.millisec THEN
       t:=t2.millisec-t1.millisec;
   ELSE
       t:=(t2.millisec+60000)-t1.millisec;
   END;
   WriteCard(i,6);
   WriteCard(t DIV 100,6);
   WriteLn;
 END;
 WriteLn;WriteLn;
 WriteString(" TIEMPOS DE EJECUCION PARA SumaIter1");
 WriteLn;
 WriteString("   i      t (dec s)");
 WriteLn;
 WriteString("---------------");
```

```
  WriteLn;
  FOR i:=0 TO N BY 20 DO
    GetTime(t1);
    FOR k:=1 TO 10000 DO
      s:=SumaIter1(VPrueba,i);
    END;
    GetTime(t2);
    IF t2.millisec>= t1.millisec THEN
       t:=t2.millisec-t1.millisec;
    ELSE
       t:=(t2.millisec+60000)-t1.millisec;
    END;
    WriteCard(i,6);
    WriteCard(t DIV 100,6);
    WriteLn;
  END;

END Ejercicio1.
```

Para la medida de tiempos de utiliza el tipo definido Time y la función GetTime() que se encuentran definidos en el módulo TimeDate de la biblioteca del compilador FST-Modula2 que es el normalmente utilizado en las prácticas. Para otros compiladores es muy probable que existan funciones de biblioteca análogas. El listado del módulo TIMEDATE.DEF es:

```
DEFINITION MODULE TimeDate;

(* (C) Copyright 1987,1988 Fitted Software Tools. All rights reserved. *)


TYPE
   Time =  RECORD
            day, minute, millisec :CARDINAL;
            (*
              day = ((Year - 1900) * 16 + Month) * 32 + DayOfTheMonth
              minute = minutes since midnight
              millisec = milliseconds past the minute
            *)
          END;

PROCEDURE GetTime( VAR time :Time );
(*
   get the system time
*)

PROCEDURE SetTime( time :Time );
(*
   set the system time
*)

END TimeDate.
```