Práctica de Programación II

Curso 97-98

Jesús Rasines F.
Medina de Pomar
I.T.I Sistemas

Enunciado del problema.

Se desea un algoritmo recursivo, completamente verificado, tal que dado un vector ***v: vector[1..n] de bool****, determine si v contiene una alternancia de valores booleanos, siendo los valores situados en posiciones pares del vector, falso y aquellos situados en posiciones impares verdadero.*

Consideraciones al enunciado.

Cuando tratamos un vector con un sólo elemento o bien un vector nulo, no está definida la solución que debe devolver el algoritmo recursivo. Pueden tomarse varias decisiones respecto a la forma de tratarlo; dependiendo de ésta, la solución ofrecida por el algoritmo en el caso trivial será distinta. Este planteamiento lo veremos desarrollado en el análisis por casos.

Especificación del problema.

Precondición: el algoritmo va a trabajar sobre un vector cuyo rango es [1..n], por lo tanto podemos asegurar que $N \geq 1$

Postcondición: en el vector se alternan necesariamente posiciones pares y posiciones impares, los valores que deben tener son mutuamente excluyentes por lo que bastará con ver para cada elemento del vector si su valor es cierto o falso dependiendo de su índice de acuerdo con la condición requerida, lo que logramos mediante la siguiente operación lógica de comparación: $v[\alpha] \neq (\alpha \text{ MOD } 2 = 0)$, siendo α el natural que indica la posición del elemento actual en el vector de booleanos.

Si α es par, entonces el resto de la división por 2 es cero, luego la comparación $\alpha \text{ MOD } 2 = 0$, debe dar como resultado cierto. Después comparamos el contenido del vector en la posición señalada por α, con el resultado anterior, cierto, y si el vector contiene falso en esa posición el resultado será cierto. El tratamiento de α impar es análogo.

{Q ≡ N ≥ 1 }
fun alt (**v**:vector[1..N] de bool) **dev b**:bool
{R ≡ b =∇ α ∈ [1..N]. v[α] ≠ (α MOD 2 = 0) }

Diseño del algoritmo recursivo.

Especificación de la inmersión.

Para realizar el diseño recursivo necesitamos hacer, en cada llamada a la función, que el tamaño del problema inicial vaya decreciendo. Para lograr éste objetivo, en cada llamada de la función vamos procesando solo un elemento del vector y volvemos a llamar recursivamente a la función. Incorporamos un nuevo parámetro inmersor, ***i,*** que será el que defina un tamaño menor de problema.

La nueva especificación de la función será:

{Q ≡ 1 ≤ i ≤ N }
fun ialt (**v**:vector[1..N] de bool, **i**:natural) **dev b**:bool
{R ≡ b =∇ α ∈ [1..i]. v[α] ≠ (α MOD 2 = 0) }

Llamada inicial: **ialt(v,n)**

Análisis por casos.

Si el vector sólo tiene un elemento, *v[1]*, es imposible que exista en él una alternancia de valores positivos y negativos, por lo que en cualquier caso debería devolver *falso* como resultado de aplicar el algoritmo a un vector con un sólo elemento. Si utilizamos el cuantificador universal en la Postcondición, como parece sugerir el enunciado ya que se trata de ciertas restricciones que deben cumplir todos los elementos del vector, tendremos que tratar el caso ***n=1,*** esto es, un vector con un elemento como una excepción. No podemos usar falso como resultado en el caso trivial, esto es, cuando ***i=1*** puesto que al evaluar el último elemento independientemente de la longitud de éste, y del resultado evaluar el resto de las celdas del vector, siempre obtendremos como resultado falso puesto que:

$$falso \ \grave{U} \ cierto \equiv falso$$
$$falso \ \grave{U} \ falso \equiv falso$$

Otra forma más sencilla de tratar el vector "mono-elemento" sería devolver *cierto* en el caso trivial. Puesto que *cierto* es el elemento neutro del cuantificador universal ∀, utilizado en la Postcondición dada en el algoritmo propuesto, cuando el rango del cuantificador se anula, el resultado también se correspondería con el caso de evaluar el vector de un sólo elemento pues:

$$cierto \ \grave{U} \ cierto \equiv cierto$$
$$cierto \ \grave{U} \ falso \equiv falso$$

También podemos tratar el caso trivial, de forma que el algoritmo devolviese en el caso trivial el valor del elemento que trata el algoritmo en ese momento: ***v[1]***

Si v[1] es cierto, entonces cumple con la condición requerida en el enunciado del algoritmo, esto es, tener en las posiciones impares el valor cierto, además se corresponde con el elemento neutro del cuantificador universal por lo que el resultado devuelto por el algoritmo sería correcto. La otra posibilidad es que v[1] sea falso, entonces el algoritmo siempre devolvería *falso*, pues en una posición impar tendríamos el valor falso. Para un vector nulo no podemos determinar que solución debe devolver el algoritmo en este caso. Puesto que tratamos con un vector "que no existe" por lo tanto no lo aceptamos.

Con todas estas consideraciones y dado que sea cual sea el modo que elijamos se trata más de una discusión semántica que de diseño, cualquier convención adoptada al diseñar el algoritmo será válida. Tratamos pues, como más aceptable la última posibilidad expuesta, por lo tanto tendremos:

Caso directo o trivial: Se alcanza cuando el parámetro de inmersión vale 1, esto es cuando hemos tratado todos los elementos del vector comenzando el recorrido por el último, o bien cuando el vector tratado sólo tiene un elemento. En ambas situaciones el resultado que debe devolver el algoritmo es el propio contenido del vector: v[1].

Caso recursivo o no trivial: Son casos recursivos todos aquellos en los que i >1. La forma de tratar el problema para obtener subproblemas de menor tamaño es comprobar en cada llamada recursiva a la función, si el elemento del vector v, señalado por i cumple la condición v[i] ≠ (i MOD 2 = 0), y de nuevo se llama recursivamente a la función ialt con un elemento menos.

Composición algorítmica.

El algoritmo de que resuelve el problema de forma recursiva es el siguiente:

{Q ≡ 1 ≤ i ≤ N }
fun ialt (**v**:vector[1..N] de bool, **i**:natural) **dev b**:bool
  caso i = 1 ⟶ v[1];
    i > 1 ⟶ (v[i] ≠ (i MOD 2 = 0)) ∧ ialt(v, i-1);
  fcaso
ffun
{R ≡ b =∇ α ∈ [1..i]. v[α] ≠ (α MOD 2 = 0) }

Verificación formal del algoritmo recursivo.

Completitud de la alternativa.
Se trata de ver que el conjunto de protecciones cubre todos los casos posibles, formalmente:
$Q(x) \Rightarrow B_t(x) \vee B_{nt}(x)$.

$Q(x) \equiv 1 \leq i \leq n$
$B_t(x) \equiv i = 1$
$B_{nt}(x) \equiv i > 1$
Y por lo tanto tendremos: $1 \leq i \leq n \Rightarrow i = 1 \vee i > 1$

Satisfacción de la precondición por la llamada recursiva.
Esto quiere decir que la función debe llamarse en estados que cumplan la precondición, expresado formalmente debe demostrarse que: $Q(x) \wedge B_{nt}(x) \Rightarrow Q(s(x))$.

$Q(x) \equiv 1 \leq i \leq n$
$s(x) \equiv i-1$
$Q(s(x)) \equiv 1 \leq i-1 \leq n$

Entonces: $1 \leq i \leq n \wedge i > 1 \Rightarrow 1 \leq i-1 \leq n$, ya que necesariamente $i \geq 2$ para la llamada recursiva.

Base de inducción.
Debemos demostrar que la postcondición se satisface para los casos triviales, en términos formales:
$Q(x) \wedge B_t(x) \Rightarrow R(x,triv(x))$.

$Q(x) \wedge B_t(x) \equiv 1 \leq i \leq n \wedge i = 1 \equiv i = 1$
$R(x,triv(x)) \equiv v[1]$
$b = \nabla \alpha \in [1..1].\ v[\alpha] \neq (\alpha \text{ MOD } 2 = 0) \equiv b = v[1] \neq (1 \text{ MOD } 2 = 0)$.

Paso de inducción.
Hay que probar ahora que la postcondición también se cumple para los casos recursivos.
$Q(x) \wedge B_{nt}(x) \wedge R(s(x),y') \Rightarrow R(x,c(y',x))$

$Q(x) \wedge B_{nt}(x) \equiv 1 < i \leq n$
$R(s(x),y') \equiv b' = \nabla \alpha \in [1..i-1].\ v[\alpha] \neq (\alpha \text{ MOD } 2 = 0)$
$R(x,c(y',x)) \equiv b = \nabla \alpha \in [1..i].\ v[\alpha] \neq (\alpha \text{ MOD } 2 = 0) =$
$\nabla \alpha \in [1..i-1].\ v[\alpha] \neq (\alpha \text{ MOD } 2 = 0) \wedge v[i] \neq (i \text{ MOD } 2 = 0)$

Elección de una estructura de preorden bien fundado.
Hay que encontrar t: $D_{T1} \rightarrow Z$, de modo que $Q(x) \Rightarrow t(x) \geq 0$

$Q(x) \equiv 1 \leq i \leq n \Rightarrow i > 0$ Por lo tanto bastará con **t(x) = i.**

Demostración del decrecimiento de los datos.
Para el preorden que hemos definido, hay que ver que el tamaño de los datos decrece para cada llamada recursiva. Equivale a demostrar: $Q(x) \wedge B_{nt}(x) \Rightarrow t(s(x)) < t(x)$.
$t(s(x)) \equiv i-1$
$t(x) \equiv i$
$1 < i \leq n \Rightarrow i-1 < i$.

Estudio del coste del algoritmo.

Dado que el tamaño de los datos decrece por sustracción, la ecuación de recurrencia que se plantea para resolver el problema es la siguiente:

Con los parámetros; *a=1* puesto que Sólo hay una llamada recursiva cada vez, *b=1* que es lo que decrece el vector en cada llamada recursiva. Además el coste de lo que no son llamadas recursivas es constante por lo tanto cn^k=cte, o lo que es igual: *k=0.*

$$T(n) = \begin{cases} cn^k & si \quad 0 \le n < b \\ aT(n-b) + cn^k & si \quad n \ge b \end{cases}$$

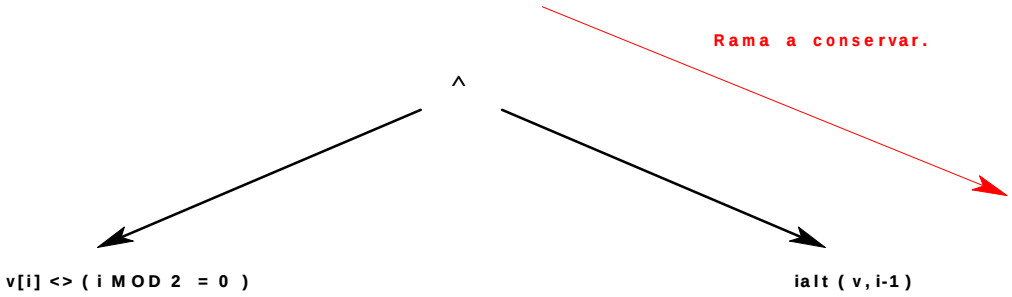
Y la solución de la recurrencia es:

$$T(n) \in \begin{cases} \Theta(n^k) & si \quad a < 1 \\ \Theta(n^{k+1}) & si \quad a = 1 \\ \Theta(a^{n \; div \; b}) & si \quad a > 1 \end{cases}$$

Aplicando los datos que hemos obtenido, a saber: a=1, b=1, k=0. El coste del algoritmo está indicado por la expresión: $T(n) \in \Theta(n)$. Por lo tanto **el coste del algoritmo recursivo es lineal.**

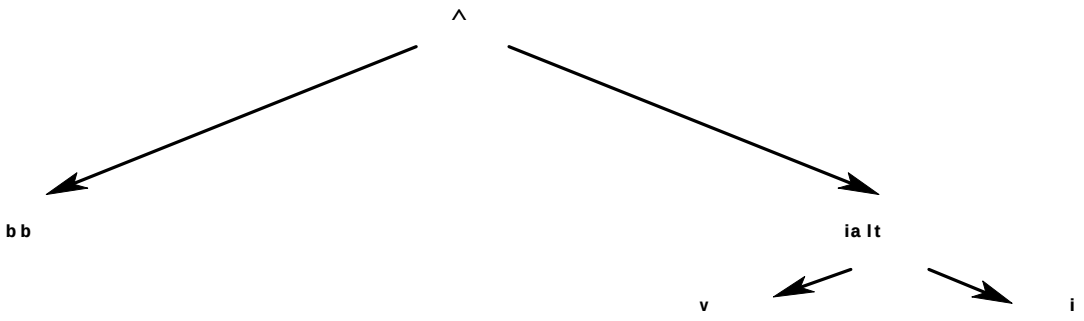
Transformación a "recursivo final".

Arbol sintáctico de la función recursiva.



Resulta evidente el camino que debemos seguir hasta llegar a la llamada a la función recursiva, que no es otro que la rama derecha; la rama izquierda de la que hemos desarrollado todo el subárbol, sustituiremos toda ella por un nuevo parámetro de inmersión que llamaremos bb.

El nuevo árbol que se obtiene es el siguiente:



Sustituciones aplicadas.

Hemos añadido un parámetro más, *bb*, que llevara precalculado el valor del resultado, por lo tanto la generalización que hacemos para la función será: *iialt(v, i, bb).*

Por la definición de ialt tenemos que: **iialt(v, i, bb) = bb ∧ ialt(v, i)**

Caso trivial: ialt = v[1] ® bb Ù v[1]

Caso recursivo: ialt = v[i] ¹ (i mod 2 =0) Ù ialt (v, i-1) ® bb Ù v[i] ¹ (i mod 2 =0) Ù ialt (v, i-1)

Desplegado y plegado.

Desplegado.

Aplicando la definición de la función ialt se obtiene:

iialt(v, i, bb) = bb ∧ v[i] ≠ (i mod 2 =0) ∧ ialt (v, i-1).

Plegado.

*S*e obtiene, aplicando la propiedad asociativa de la operación lógica,*Ù,* en la expresión de la función desplegada, y después aplicando la definición de la propia función iialt; con lo cual se llega a:

iialt(v, i-1, v[i] ≠ (i mod 2 =0) ∧ bb).

Llamada inicial a la nueva función recursiva final es**: iialt(v, n, cierto).**

De esta forma se cumple que iialt(v, n, cierto) = ialt (v, n).

Código del algoritmo utilizando la función recursiva final.

```
{Q ≡ 1 ≤ i ≤ N }
fun iialt (v:vector[1..N] de bool, i:natural, bb: bool) dev b:bool
        caso    i = 1 ⟶ bb ∧ v[1];
                i > 1 ⟶ iialt(v, i-1, v[i] ≠ (i mod 2 =0) ∧ bb).
        fcaso
ffun
{R ≡ b =∇ α ∈ [1..i]. v[α] ≠ ( α MOD 2 = 0 ) }
```

Transformación del algoritmo recursivo final, en iterativo.

Código del algoritmo iterativo.

```
Q ≡ 1 ≤ iini ≤ Nini }
fun iialt-it (vini:vector[1..N] de bool, iini:natural, bbini: bool) dev b:bool
        var     v:vector[1..N] de bool;
                i:natural;
                bb: bool;
        fvar
        v:=vini;
        i:=iini;
        bb:=bbini;

        mientras  i > 1 hacer
                bb:= v[i] ≠ (i mod 2 =0) ∧ bb);
                i:=i-1;
        fmientras
        dev  bb ∧ v[1];
ffun
{R ≡ b =∇ α ∈ [1..i]. v[α] ≠ ( α MOD 2 = 0 ) }
```

Invariante y estudio del coste.

Invariante: Según se indica en el libro de Peña, el invariante para un bucle obtenido por transformación de un algoritmo recursivo final a iterativo es de la forma: **P≡Q ∧ f(xini).** De este modo tendremos que el invariante para el bucle será: **P ≡ 1 ≤ i ≤ N ∧ iialt-it (vini, iini, bbini) = iialt(v,i,bb).**

Otro invariante podemos obtenerlo por debilitamiento de la postcondición, de esta forma el invariante propuesto será: **P ≡ b =∇ α ∈ [1..i]. v[α] ≠ (α MOD 2 = 0) ∧ 1 ≤ i ≤ N**

Función de cota: Se trata de conseguir una función t(i), puesto que i es el índice que hace avanzar el bucle, además este está limitado por n, una posible función podría ser: **t(i) =n-i**, o bien , simplemente **t(i)=i**, puesto que i siempre es menor o igual que n.

Coste del algoritmo iterativo.

El bucle del tipo ***mientras B hacer S fmientras*** tiene un coste que está en $O(f_B, s(n)\cdot \times f_{iter}(n))$.

El coste de evaluar la condición B, de finalización del bucle tiene coste constante, $f_B \in O(1)$.

Tenemos un bucle con dos operaciones "complejas" en su interior cuyo coste es dependiente de la máquina y constante en cada iteración del bucle lo que se traduce en una constante multiplicativa al implementar el mismo algoritmo en computadores diferentes; el coste de s(n) es: $O(1)$.

El bucle se repite un total de n-1 veces para recorrer el vector comprobando si cada elemento de este, cumple la propiedad buscada. Por tanto el coste de $f_{iter}(n)$, será n-1.
El coste del bucle es $O(n-1) \subset O(n)$.

El resto de las operaciones previas y posteriores al bucle son asignaciones y devolver el resultado calculado cuyo coste es constante aplicando la regla de la suma a toda la secuencia de instrucciones que compone el algoritmo, obtenemos que: $O(\max(1,n))=. O(n)$.

Finalmente el coste del algoritmo *está en el orden de n,* esto es: $\mathbf{T(n) \in \Theta(n)}$.

Al igual que el algoritmo recursivo del que hemos obtenido el iterativo por transformación el coste depende linealmente del tamaño de los datos de entrada del problema.

Algoritmo codificado en Modula-2

Modulo de definición de las funciones auxiliares.

```
DEFINITION MODULE Funciones;
FROM InOut IMPORT   ReadString, Read, ReadInt,
                    WriteInt, WriteString, WriteLn,Write;

CONST max=32767;

VAR long,indice:INTEGER;

   PROCEDURE ialt (VAR v:ARRAY OF BOOLEAN; i:INTEGER):BOOLEAN;

   PROCEDURE iialt_it (VAR v:ARRAY OF BOOLEAN; i_ini:INTEGER; bb_ini:BOOLEAN)

   PROCEDURE LeeVector (VAR v:ARRAY OF BOOLEAN);

   PROCEDURE EscribeVector (VAR v:ARRAY OF BOOLEAN; VAR numero:INTEGER);

END Funciones.
```

Módulo de implementación de las funciones auxiliares.

```
IMPLEMENTATION MODULE Funciones;

FROM InOut IMPORT   ReadString, Read, ReadInt,
                    WriteInt, WriteString, WriteLn,Write;



PROCEDURE ialt(VAR v:ARRAY OF BOOLEAN; i:INTEGER):BOOLEAN;
(* Función recursiva no final que resuelve el problema *)

BEGIN

   IF i>1 THEN
      RETURN (v[i]#(i MOD 2=0)) AND ialt(v,i-1);
   ELSE
      RETURN v[1];
   END;

END ialt;



PROCEDURE iialt_it(VAR v:ARRAY OF BOOLEAN;i_ini:INTEGER; bb_ini:BOOLEAN):BOOLE
(* Función iterativa para resolver el problema *)

VAR i:INTEGER;
   bb: BOOLEAN;

BEGIN
   i:=i_ini;
   bb:=bb_ini;

   WHILE i>1 DO
      bb:=(v[i]#(i MOD 2=0)) AND bb;
      i:=i-1;
   END;
   RETURN v[1] AND bb;

END iialt_it;
```

```
PROCEDURE LeeVector (VAR v:ARRAY OF BOOLEAN);
(* Procedimiento para leer un vector elemento a elemento introducido desde el teclado *)

   VAR k:CHAR;
      j:INTEGER;
      bool:BOOLEAN;

   BEGIN
      j:=1;
      WriteString("teclea el elemento ");WriteLn;
      LOOP
         WriteString("v[");WriteInt(j,0);WriteString("]=");
         Read(k);Write(k);WriteLn;
         IF k="." THEN EXIT END;
         IF (k="v") OR (k="V") THEN bool:=TRUE END;
         IF (k="f") OR (k="F") THEN bool:=FALSE END;
         v[j]:=bool;
         long:=j; INC(j);
      END;

END LeeVector;

PROCEDURE EscribeVector (VAR v:ARRAY OF BOOLEAN; VAR numero:INTEGER);
(* Procedimiento para escribir el vector introducido *)

VAR m: INTEGER;

BEGIN
   FOR m:=1 TO numero DO
      IF v[m]=TRUE THEN Write("V") ELSE Write("F") END;
   END;
END EscribeVector;

END Funciones.
```

Programa Principal.

```
MODULE Alternados;

FROM InOut IMPORT   ReadString, Read, WriteInt, WriteString, WriteCard,
WriteLn, Write;

FROM Funciones IMPORT LeeVector, EscribeVector,
                  ialt, iialt_it, long, max;

IMPORT TimeDate;


(* CONST max=32767 *)

VAR z,zz,l,tiempo_total:INTEGER;
   c,d:BOOLEAN;
   w:ARRAY [1..max] OF BOOLEAN;
   inicial,final:TimeDate.Time;



PROCEDURE VerResultado(x:BOOLEAN);

(* Este procedimiento muestra el resultado de evaluar el vector  *)
BEGIN
   IF x=FALSE THEN
      WriteString("falso");
   ELSE
      WriteString("CIERTO");
   END;
END VerResultado;
```

```
PROCEDURE Diferencia_de_tiempos(VAR comienzo,fin:TimeDate.Time):CARDINAL;

(* Este procedimiento calcula la diferecia de tiempo expresada en milisegundos,
entre dos instantes de tiempo consecutivos, tomados del reloj del sistema con
las limitaciones que ello conlleva; en MSDOS se actualiza a través de la INT 8,
a una frecuencia de 18.2 Hz. La precisión sólo llega a 55 milésimas. *)

VAR tiempo:CARDINAL;

BEGIN

   tiempo:=((fin.minute)*60000)+fin.millisec -
         ((comienzo.minute)*60000) - comienzo.millisec;
   RETURN tiempo;
END Diferencia_de_tiempos;




(* ============================================================================
                              PROGRAMA   PRINCIPAL
============================================================================*)

BEGIN

LeeVector(w);   (* Leemos un vector desde el teclado, elemento a elemento, y lo almacenamos en w *)

IF long=0 THEN RETURN END; WriteLn;

(*Procesa el vector, SOLO SI NO ES NULO. Aunque en la precondición se especifica que i>=1 debido al modo de leerlo
(vector abierto, importando después la  longitud) aparentemente se procesaria un vector nulo y el resultado devuelto
aunque el programa dice falso, puede ser cualquier cosa, pues la postcondición sólo garantiza que el resultado es correcto
para aquellos valores de entrada  que cumplen la precondición y el valor i=0 no lo cumple   *)

WriteString(" Longitud del vector "); WriteInt(long,4);WriteLn;WriteLn;

TimeDate.GetTime(inicial);
FOR z:=1 TO 200 DO; FOR zz:=1 TO 20000 DO;

   c:=ialt(w,long);     (*Proceso del vector, recursivo no final *)

END; END;
TimeDate.GetTime(final);

WriteString("Utilizando la función recursiva, se tardan. ");
WriteCard(Diferencia_de_tiempos(inicial,final),10);WriteLn;WriteLn;

WriteString("Es ");   VerResultado(c);
WriteString(" que el vector cumpla la propiedad requerida ");WriteLn;WriteLn;

(*Tomamos el tiempo antes de procesar el vector.
  Lo procesamos 4.000.000 de veces para obtener un resultado apreciable.
  Tomamos el tiempo tras procesarlo.
  Calculamos la diferencia de tiempos y la mostramos.   Mostramos el resultado de la comprobación *)
(* El mismo proceso lo repetimos para la función iterativa *)

TimeDate.GetTime(inicial);
FOR z:=1 TO 200 DO; FOR zz:=1 TO 20000 DO;

   d:=iialt_it(w,long,TRUE);  (* Proceso del vector iterativo *)

END; END;
TimeDate.GetTime(final);

WriteString("Utilizando la función iterativa, se tardan. ");
WriteCard(Diferencia_de_tiempos(inicial,final),10);WriteLn;WriteLn;

WriteString("Es ");   VerResultado(d);
WriteString(" que el vector cumpla la propiedad requerida ");WriteLn;WriteLn;

END Alternados.
```

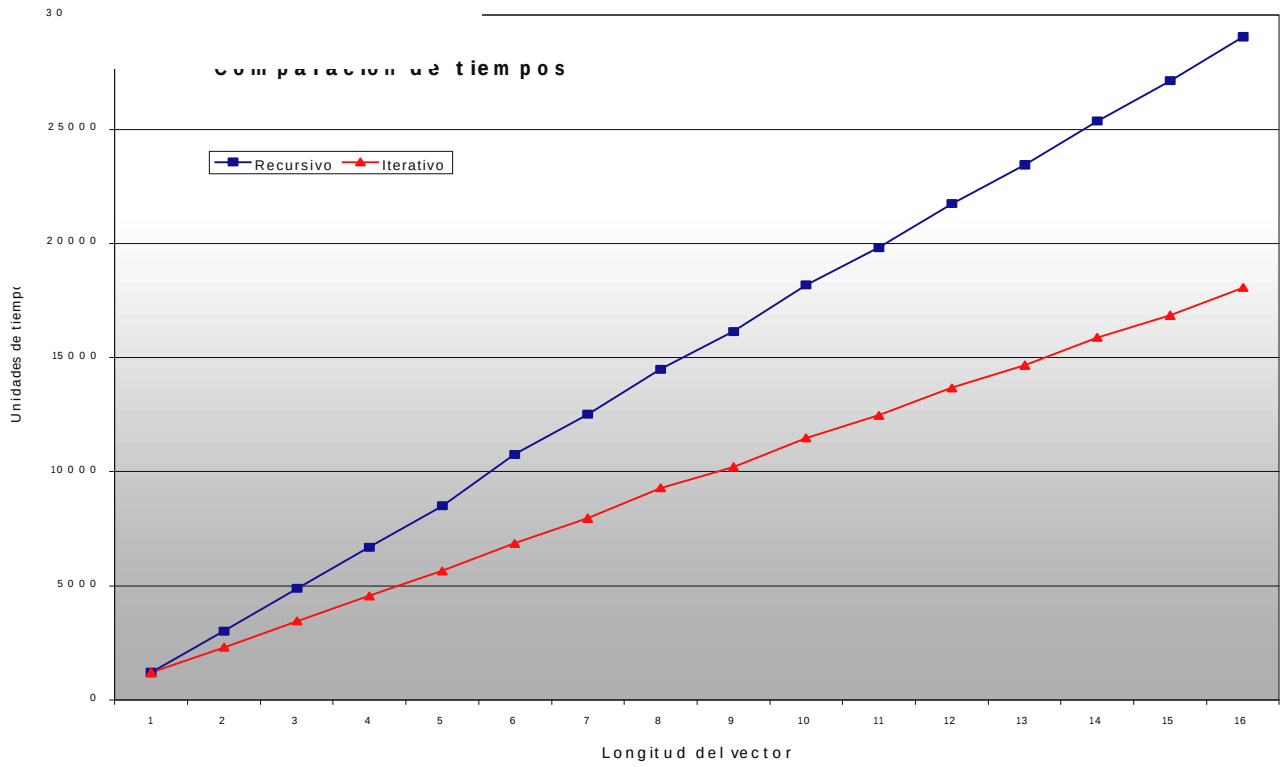
Comparación de resultados experimentales.

Como era lógico esperar, las versiones codificadas de ambos algoritmos, siguen un comportamiento lineal, sus tiempos de ejecución crecen a medida que lo hace el tamaño del vector que están tratando.

El algoritmo iterativo, es ligeramente más rápido que el recursivo (no final), debido a que lo hemos obtenido por transformación de un algoritmo recursivo final, con lo que nos ahorramos los pasos de apilar y desapilar resultados parciales que consumen un tiempo extra en el algoritmo recursivo no final.

Los resultados experimentales tras la ejecución de un juego de pruebas, han sido los siguientes:

Longitud del vector	Recursivo	Iterativo
1	1210	1210
2	3020	2310
3	4890	3460
4	6700	4560
5	8510	5660
6	10770	6870
7	12520	7970
8	14500	9290
9	16150	10210
10	18180	11480
11	19830	12470
12	21750	13680
13	23450	14670
14	25370	15880
15	27130	16860
16	29060	18070



Juego de pruebas de ejecución, para verificar el funcionamiento correcto.

Vector de prueba.	Respuesta esperada.	Respuesta obtenida.
F	FALSO	FALSO
V	CIERTO	CIERTO
FF	FALSO	FALSO
FV	FALSO	FALSO
VF	CIERTO	CIERTO
VV	FALSO	FALSO
FFF	FALSO	FALSO
FFV	FALSO	FALSO
FVF	FALSO	FALSO
FVV	FALSO	FALSO
VFF	FALSO	FALSO
VFV	CIERTO	CIERTO
VVF	FALSO	FALSO
VVV	FALSO	FALSO
VFVFVFVFVF	CIERTO	CIERTO
FFFFFFFFFF	FALSO	FALSO
VFFFFFFFFF	FALSO	FALSO