Teoema del invariante (Parte I)

Clase práctica AED I

Martín A. Miguel (*March*)
Rodrigo Castaño

1 de Junio de 2012

Introducción

- Hoy vamos a hacer un repaso de los conceptos y herramientas que usamos para demostrar la correctitud de programas en el paradigma imperativo.
- El enfoque principal va a estar alrededor del teorema del invariante, ya que es la herramienta que nos permite analizar ciclos. Los ciclos son una de las principales herramientas de este paradigma, como es la recursión en funcional.

Demostraciones

¿Qué significa demostrar la correctitud de un programa imperativo?

Queremos ver que todos los posibles estados al final de la función cumplen la postcondición. Un estado final de la función es el conjunto de valores que toman las variables que se encuentran en el ámbito de la función, luego de una ejecución de la misma con variables que cumplen la precondición.

¿Cómo exploramos TODOS los posibles estados finales?

No podemos ver qué resultado da la función para cada conjunto posible de entradas. De la misma forma que en álgebra no demostramos $x^2 > x, x \in \mathbb{N}$ viendo todos los valores posibles para x. En cambio, vamos a trabajar con las variables de forma abstracta, examinando las transformaciones que sufren a lo largo del código y viendo qué propiedades se mantienen.

Veamos que herramientas tenemos para hacer el análisis.

Vamos por partes decía Jack

Asignaciones y operaciones

El ejemplo más simple de código es cuando tenemos solamente variables, asignaciones y operaciones, con un único flujo de ejecución.

```
1  int foo (int x) {
2    int i = x + 1;
3    //estado e1; vale i@e1 == x@pre + 1
4    i = i + i;
5    //estado e2; vale i@e2 == i@e1 + i@e1
6    //implica i@e2 == (x@pre + 1) + (x@pre + 1)
7    //implica i@e2 == (x@pre + 1) * 2
8    return i;
9    //vale i == (x+1)*2
10 }
```

En este caso vemos como se van modificando las variables estado a estado y luego deducimos que relación hay entre el estado final y el inicial. Con estas observaciones luego podemos inferir si se cumple o no la postcondición.

Vamos por partes decía Jack

Funciones auxiliares

Cuando durante el curso del programa se llama a una función auxiliar, podemos abstraernos sobre qué pasa adentro de ella y atenernos a su especificación. Si nos aseguramos que el estado inmediatamente anterior a la llamada implica la precondición del problema resuelto por la función, podemos asumir que en el estado inmediatamente posterior se cumple la postcondición. Con esa información continuamos el seguimiento.

```
1  int foo(int x) {
2      int i = x + 1;
3      //estado pfunc; vale i@pfunc == x@pre + 1
4      int r = sumat(i);
5      //estado qfunc;
6      //vale r@qfunc == \sum[0..i@pfunc] && i@qfunc
           == i@pfunc
7      //implica r@qfunc == \sum[0..x@pre + 1]
8      return r;
9  }
```

Lamentablemente no siempre nos alcanza con concatenar operaciones entre variables y llamar auxiliares. Muchas veces debemos modificar el flujo de ejecución. Es en estos casos que lo que vimos hasta ahora no alcanza para realizar las demostraciones.

Sin embargo, en el espíritu de los computadores siempre se intenta reducir los problemas grandes a otros más chicos que ya conozcamos. Esto es lo que vamos a hacer.

Vamos por partes decía Jack

Condicionales

Los condicionales nos definen dos posibles flujos de ejecución según el valor de la guarda B. Para simplificar el problema, vamos a dividirlo. Por una parte tenemos dos conjuntos de acciones distintas: el bloque *then* y el bloque *else*.

¿Pero cómo unificamos las concecuencias de cada bloque?
Bueno, vamos a pensar el condicional como un problema aparte dentro de nuestro código. Si es un problema tiene que tener una precondición y una postcondición. Luego vamos a usar un formalismo particular para demostrar que si vale la precondición, vale la postcondición. Este formalismo se presenta en el ejemplo a continuación.

```
1   ...
2   //Pif
3   if (B) {
4     //vale Pif && B
5     bloque then
6     //vale H
7   } else {
8     //vale Pif && -B
9     bloque else
10    //vale G
11  }
12  //estado qif; vale H || G
13  //Qif
14  ...
```

Siendo H el resultado de un flujo de ejecución con precondiciones Pif y B, y G el resultado de un flujo de ejecución con precondiciones Pif y $\neg$B; queremos demostrar que H $\vee$ G $\Rightarrow$ Qif.

Si logramos demostrar Qif, se cumple la postcondición de nuestro if y podemos trabajar el condicional como si fuera una función auxiliar, tal como vimos antes.

Queda claro que reducimos el problema en problemas más chicos, algunos ya conocidos.

Ahora si,
sin más demoras,
hay que enfrentarse a los...

Ciclos

En los ciclos es se vuelve más complicado comprender como se transforman las variables en cada iteración y cuál va a ser el estado del programa una vez que este termina (si es que lo hace).

Para resolver este problema vamos a arrancar usando la misma idea que con los condicionales: vamos a definir una precondición y una postcondición para el ciclo, e intentar probarlos.

Además, para mantener el espíritu de la computación, vamos a separar el problema de terminación del de correctitud. En otras palabras, vamos a demostrar por un lado que el ciclo termina, y por el otro que hace lo que queremos.

Ciclos

Terminación

Para atacar el problema de la terminación le vamos a pedir prestado un concepto a nuestro amigo el paradigma funcional: la función variante.

La función variante de ciclos nos permite medir que tan cerca estamos de llegar al final del mismo. Para ello además definimos una cota para esta función variante. La cota será el equivalente al caso base de la recursión.

Si sabemos que la función variante disminuye estríctamente en cada iteración y $(v \leq c) \Rightarrow \neg B$, sabemos que el ciclo termina.

Ciclos

Correctitud

Ahora, para la correctitud vamos a necesitar maquinaria más pesada, y para ello vamos a llamar al padre político de nuestra ciencia: la matemática. La maquinaria: el teorema de inducción.

El teorema de inducción nos permite probar una propiedad $P(n)$ para todos los naturales n. La idea detrás del teorema de inducción es que la propiedad relaciona a un n con un $k < n$. En particular, solemos probar $P(n) \Rightarrow P(n+1)$.

Bueno, si n es el número de iteración, existe una relación muy clara entre la iteración n y la $n+1$. ¡Así que podemos usar la misma idea!

Ciclos

Correctitud

Cuando demostramos que el estado inmediato interior al ciclo implica el invariante, es análogo a probar $P(0)$. Cuando vemos que invariante se mantiene, es análogo a probar $P(n) \Rightarrow P(n+1)$.

Por inducción, tenemos una propiedad que vale sobre el ciclo SIEMPRE. Esto no es algo menor, porque acabamos de resumir toda la mecánica del ciclo en una única condición.

Con esto hecho, ver que el ciclo es correcto es tan fácil como ver que cuando no se cumple la guarda, sabiendo que el invariante vale, se cumple la postcondición (i.e.: $(I \wedge \neg B) \Rightarrow Q_c$).

Ciclos

El teorema del invariante

Resumiendo

Necesitamos definir:

- Precondición del ciclo (P_c): son aquellas cosas que deben valer para que el ciclo tenga sentido.
- Poscondición del ciclo (Q_c): condición que esperamos cumplan las variables al finalizar el ciclo.
- Invariante del ciclo (I): la propiedad general que queremos probar del ciclo. Funciona como una abstracción del mismo.
- Guarda (B): condición que define si se continúa la ejecución del ciclo.
- Función variante(v): nuestra forma de medir que tan cerca estamos de terminar el ciclo.
- Cota (c): nuestra forma de saber, a partir de v que el ciclo terminó.

Luego debemos demostrar:

- $P_c \Rightarrow I$
- v decrece estrictamente en cada iteración.
- $(I \wedge v \leq c) \Rightarrow \neg B$
- $\neg B \wedge I \Rightarrow Q_c$ El invariante I se mantiene.

Ejercicio 1

Dada la siguiente especificación y código, demostrar que el segundo cumple el primero.

$$
\begin{array}{l}
\texttt{problema } \mathit{sumal}(l : [\texttt{Int}]) = r : \texttt{Int}\{ \\
\quad \texttt{asegura } r == \texttt{sumaSublista}(l,\ 0,\ \texttt{longitud}(l)) \\
\quad \texttt{aux sumaSublista}(\texttt{Lista}\langle\texttt{Int}\rangle\ l, \texttt{Int}\ x, \texttt{Int}\ y) : \mathit{Int} = \\
\quad \texttt{suma}\ [\ \texttt{iesimo}(l, i)\ |\ i \leftarrow [x..y)\ ] \\
\}
\end{array}
$$

```
1  int sumal(Lista<int> l)
2  {
3      int s=0, i =0;
4      while (i < l.longitudL())
5      {
6          s = s + l.iesimoL(i);
7          i = i + 1;
8      }
9      return s;
10 }
```

Ejercicio 1

```
1  int sumal(Lista<int> l)
2  {
3      int s=0, i =0;
4      //estado pc1; vale s@pc1 == 0 && i@pc1 == 0
5      while (i < l.longitudL())
6      {
7          // estado e1: vale I && B
8          s = s + l.iesimoL(i);
9          // estado e2: vale i@e2 == i@e1  && s@e2 == s@e1 + iesimo
               (l, i@e1)
10         i = i + 1;
11         //~ estado e3: vale i@e3 == i@e2 + 1 && s@e3 == s@e2
12     }
13     //~ estado fc: vale Qc
14     return s;
15     //~ estado fin: vale s == s@fc && vale res == s
16 }
```

Ejercicio 1

```
1  int sumal(Lista<int> l)
2  {
3      int s=0, i =0;
4      //estado pc1; vale s@pc1 == 0 && i@pc1 == 0
5      //Pc : i == 0 && s == 0
6      while (i < l.longitudL())
7      {
8          //I: 0 <= i <= longitud(l) && s == sumaSublista(l,0,i)
9
10         // estado e1: vale I && B
11         s = s + l.iesimoL(i);
12         // estado e2: vale i@e2 == i@e1  && s@e2 == s@e1 + iesimo
               (l, i@e1)
13         //~ implica s@e2 == sumaSublista(l,0,i@e1+1)
14         i = i + 1;
15         //~ estado e3: vale i@e3 == i@e2 + 1 && s@e3 == s@e2
16         //~ implica s@e3 == sumaSublista(l,0,i@e1+1)
17         //~ implica i@e3 == i@e1 + 1 && s@e3 == sumaSublista(l,0,
               i@e3)
18     } //v: longitud(l) - i ; c == 0
19     //Qc: s == sumaSublista(l, 0, longitud(l))
20     //~ estado fc: vale Qc
21     return s;
22     //~ estado fin: vale s == s@fc && vale res == s
23     //~ implica res == sumaSublista(l, 0, longitud(l))
24     //~ implica Q
25 }
```

Ejercicio 1

pc1 $\Rightarrow$ Pc

Vale i@pc1 $= 0 \wedge$ s@pc1 $= 0$

Ejercicio 1

Pc $\Rightarrow$ I

Por Pc: vale $i = 0$	
a) Luego	implica $i \geq 0$
b) Por invariante de Lista$\langle$Int$\rangle$:	implica longitud(l) ≥ 0,
c) Como i $= 0$:	implica longitud(l) $\geq i$
Por a) y c)	implica $0 \leq i \leq$ longitud(l)
vale sumaSublista(l, 0, 0) $= 0$	
Por i=0:	implica sumaSublista(l, 0, i) $= 0$
Por s=0:	implica sumaSublista(l, 0, i) $= s$
Por todo lo anterior,	implica I

Ejercicio 1

$(I \wedge \neg B) \Rightarrow Qc$

a) Por ¬B:	implica i $\geq$ longitud(l)
b) Por I:	implica i $\leq$ longitud(l)
c) Por a) y b)	implica i = longitud(l)
d) Por I:	implica s = sumaSublista(l, 0, i)
Por c) y d)	implica s = sumaSublista(l,0,longitud(l))
	implica Qc

Ejercicio 1

I se mantiene

i) Por los implica de e3:	implica i@e3 = i@e1 + 1 ∧ s@e3 = sumaSublista(l,0,i@e3)
a) Por B:	implica i@e1 < longitud(l)
b) Por I:	implica i@e1 ≥ 0
Por a) y propiedades de naturales:	
c)	implica i@e1 + 1 <= longitud(l)
d) Por b)	implica i@e1 + 1 ≥ 0
e) Por c) y d)	implica 0 <= i@e3 <= longitud(l)
Por i) y e)	implica 0 <= i@e3 <= longitud(l) ∧ s@e3 = sumaSublista(l,0,i@e3) implica I

Ejercicio 1

La función variante disminuye

v: longitud(l) - i ; c = 0
Por los implica de e3:

implica i@e3 = i@e1 + 1 $\wedge$ s@e3 = sumaSublista(l,0,i@e3)
vale longitud(l) - i@e1 - 1 $<$ longitud(l) - i@e1
implica longitud(l) - i@e3 $<$ longitud(l) - i@e1
implica v@e3 $<$ v@e1

Ejercicio 1

$(I \wedge v \leq c) \Rightarrow \neg B$

v: longitud(I) - i ; c = 0

Por v <= c:

vale longitud(i) - i <= 0
implica longitud(i) <= i I I
implica ¬(longitud(i) > i)
implica ¬B

Ejercicio 2

Dada la siguiente especificación y código, demostrar que el segundo cumple el primero.

$$\begin{array}{l}\texttt{problema } \mathit{fact}(x : \texttt{Int}) = r : \texttt{Int}\{ \\ \quad \texttt{requiere } x \geq 0 \\ \quad \texttt{asegura } r == \prod[1..x] \\ \}\end{array}$$

```
1  int fact(int x) {
2    int i = 1;
3    int f = 1;
4    if (x > 0) {
5         while (i < x) {
6           i = i + 1;
7           f = f * i;
8         }
9    }
10   return f;
11 }
```

Ejercicio 2

Agregamos los estados importantes y los cambios de estados locales. Esto es, de un paso al siguiente.

```
1  int fact(int x) {
2    int i = 1;
3    int f = 1;
4    //estado pif; vale f@pif == 1 && i@pif == 1 && x@pif == x@pre
5    if (x > 0) {
6        //estado iif; vale x@iif == x@pif && x@pif > 0 && f@iif == f@pif && i@iif == i@pif
7        while (i < x) {
8          //estado e1; vale I && B
9          i = i + 1;
10         //estado e2; vale i@e2 == i@e1 + 1 && f@e2 == f@e1 && x@e2 == x@e1
11         f = f * i;
12         //estado e3; vale f@e3 == f@e2 * i@e2 && i@e3 == i@e2 && x@e3 == x@e2
13       }
14       //estado fc; vale Qc
15   }
16   //estado qif; vale (x > 0 => f@qif == f@fc && i@qif == i@fc && x@qif == x@fc)
17   //         &&  vale (- (x > 0) => f@qif == pif && i@qif == i@pif && x@qif == x@pif)
18   return f;
19 }
```

Ejercicio 2

Ahora agregamos aquellas proposiciones que nos son interesantes:

- Precondición del if: **Pif**
- Postcondición del if: **Qif**
- Precondición del ciclo: **Pc**
- Postcondición del ciclo: **Qc**
- Invariante de ciclo: **I**
- Función variante y cota: **v**, **c**

```
1  int fact(int x) {
2    int i = 1;
3    int f = 1;
4    //estado pif; vale f@aif == 1 && f@aif == 1 && x@pif == x@pre
5    //Pif: x = x@pre && x >= 0
6    if (x > 0) {
7        //estado iif; vale x@iif == x@pif && x@pif > 0 && f@iif == f@pif && i@iif == i@pif
8        //Pc : i == 1 && f == 1 && x >= 1 && x == x@pre
9        while (i < x) { //I: 1 <= i <= x && f == \prod[1..i] && x >= 1 && x == x@pre
10         //estado e1; vale I && B
11         i = i + 1;
12         //estado e2; vale i@e2 == i@e1 + 1 && f@e2 == f@e1 && x@e2 == x@e1
13         f = f * i;
14         //estado e3; vale f@e3 == f@e2 * i@e2 && i@e3 == i@e2 && x@e3 == x@e2
15       } //v: i - x; c == 0
16       //Qc: f == \prod[1..x] && x == x@pre
17       //estado fc; vale Qc
18   }
19   //estado qif; vale (x > 0 => f@qif == f@fc && i@qif == i@fc && x@qif == x@fc)
20   //         ||  vale (- (x > 0) => f@qif == pif && i@qif == i@pif && x@qif == x@pif)
21   //Qif: f@qif == \prod[1..x] && x@qif == x@pre
22   return f;
23   //Q: f == \prod[1..x]
24 }
```

Ejercicio 2

Pc $\Rightarrow$ I

Ahora demostramos que el invariante vale al entrar en el ciclo.

$$\begin{aligned}
&\text{Vale } Pc : i = 1 \wedge f = 1 \wedge x \geq 1 \wedge x = x@pre \\
&\text{Implica } i \geq 1 \wedge f = 1 \wedge x \geq 1 \wedge x = x@pre \\
&\qquad \text{Porque } i = 1 \Rightarrow i \geq 1 \\
&\text{Implica } i \geq 1 \wedge f = \prod[1..i] \wedge x \geq 1 \wedge x = x@pre \\
&\qquad \text{Porque } i = 1 \Rightarrow [1..i] = [1.,1] \Rightarrow 1 = \prod[1..i] \Rightarrow f = \prod[1..i] \\
&\text{Implica } i \geq 1 \wedge f = \prod[1..i] \wedge x \geq 1 \wedge x = x@pre \wedge i \leq x \\
&\qquad \text{Porque } x \geq 1 \Rightarrow x \geq i \\
&\text{Implica } I
\end{aligned}$$

Ejercicio 2

$(I \wedge \neg B) \Rightarrow Qc$

Luego demostraremos que el invariante y la negación de la guarda implican la postcondición del ciclo.

Vale I; Vale $\neg B$

Vale $1 \leq i \leq x \wedge f = \prod[1..i] \wedge x = x@pre \wedge \neg(i < x)$

Implica $1 \leq i \leq x \wedge f = \prod[1..i] \wedge x = x@pre \wedge x \geq i$

Por la negación del operador $<$

Implica $i = x \wedge f = \prod[1..i] \wedge x = x@pre$

Porque $i \leq x \wedge x \leq i \Rightarrow i = x$

Implica $i = x \wedge f = \prod[1..x] \wedge x = x@pre$

Porque $i = x \Rightarrow [1..i] = [1..x] \Rightarrow \prod[1..i] = \prod[1..x] \Rightarrow f = \prod[1..x]$

Implica Qc

Ejercicio 2

I se mantiene

Queremos ver que I@e1 $\Rightarrow$ I@e3.

Vale $f@e3 = f@e2 * i@e2 \land i@e3 = i@e2 \land x@e3 = x@e2 \land I@e1 \land B@e1$

Implica $f@e3 = f@e1 * (i@e1 + 1) \land i@e3 = i@e1 + 1 \land x@e3 = x@e1 \land x@e3 \geq 1$

Por sustitución

Implica $f@e3 = \prod[1..i@e1] * (i@e1 + 1) \land i@e3 = i@e1 + 1 \land x@e3 = x@pre \land x@e3 \geq 1$

Por sustitución

Implica $f@e3 = \prod[1..(i@e1 + 1)] \land i@e3 = i@e1 + 1 \land x@e3 = x@pre \land x@e3 \geq 1$

Por propiedad de la productoria

Implica $f@e3 = \prod[1..i@e3] \land i@e3 = i@e1 + 1 \land x@e3 = x@pre \land x@e3 \geq 1$

Por sustitución

Implica $f@e3 = \prod[1..i@e3] \land i@e3 = i@e1 + 1 \land 1 \leq i@e3 \land x@e3 = x@pre \land x@e3 \geq 1$

Porque$i@e1 \geq 1 \Rightarrow i@e1 + 1 \geq 1 + 1 \geq 1 \Rightarrow i@e3 \geq 1$

Implica $f@e3 = \prod[1..i@e3] \land i@e3 = i@e1 + 1$

$\land 1 \leq i@e3 \land i@e3 \leq x \land x@e3 = x@prex@e3 \geq 1$

Porque $i@e1 < x@e1 \Rightarrow i@e1 + 1 < x@e1 + 1 \Rightarrow i@e1 + 1 \leq x@e1$(con x e i Int)

$\Rightarrow i@e3 \leq x@e1 \Rightarrow i@e3 \leq x@e3$

Porque $x@e1 = x@pre \land x@e3 = x@pre$

Implica $I@e3$

Ejercicio 2

La función variante disminuye

Por lo visto anteriormente tenemos:

$$
\begin{aligned}
\text{Vale } & x@e3 = x@e1 \wedge i@e3 = i@e1 + 1 \\
\text{Implica } & v@e3 = x@e3 - i@e3 = x@e1 - (i@e1 + 1) = \\
& x@e1 - i@e1 - 1 < x@e1 - i@e1 = v@e1 \\
\text{Implica } & v@e3 < v@e1
\end{aligned}
$$

Ejercicio 2

$(I \wedge v \leq c) \Rightarrow \neg B$

Vale $I \wedge v \leq c$

Implica $1 \leq i \leq x \wedge x = x@pre \wedge x \geq 1 \wedge f = \prod[1..i] \wedge x \neg i \leq 0$

Implica $x \leq i$

Porque $x - i \leq 0 \iff x \leq i$

Implica $\neg\neg(x \leq i)$

Porque la doble negación se cancela

Implica $\neg(x > i)$

Por negación de $\leq$

Implica $\neg B$

Queda entonces demostrado que en fc vale Qc y que el ciclo termina. Veamos el resto de la demostración.

Ejercicio 2

iif $\Rightarrow$ Pc

Ya está demostrado el ciclo. Ahora podemos trabajar únicamente con la especificación del ciclo. Esto es, Pc y Qc. Primero tenemos que ver que se cumple la precondición del ciclo. Para esto exigimos que el estado inmediato antes del ciclo la implique.

Vale $x@iif = x@pif \land x@pif > 0 \land f@iif = f@pif \land i@iif = i@pif$

Implica $x@iif = x@pre \land x@pif > 0 \land f@iif = 1 \land i@iif = 1$

Por sustitución

Implica $x@iif = x@pre \land x@pif \geq 1 \land f@iif = 1 \land i@iif = 1$

Porque $x > 0 \Rightarrow x \geq 0 + 1$(con x Int) $\Rightarrow x \geq 1$

Implica Pc

Ejercicio 2

qif $\Rightarrow$ Qif

Ahora que sabemos que se cumple la postcondición del ciclo, vale Qc en el estado fc. Queremos inferir Qif. Luego sale de forma inmediata que se cumple la postcondición de la función.

$$\text{Vale } (x > 0 \Rightarrow f@qif = f@fc \wedge i@qif = i@fc \wedge x@qif = x@fc)$$
$$\wedge (\neg(x > 0) \Rightarrow f@qif = pif \wedge i@qif = i@pif \wedge x@qif = x@pif)$$

$$\text{Implica } (x > 0 \Rightarrow f@qif = \prod[1..x]) \wedge (\neg(x > 0) \Rightarrow f@qif = 1)$$

Por sustitución

$$\text{Implica } f = \prod[1..x]$$

Porque si $x > 0$ vale $f@qif = \prod[1..x]$

Si no lo es, $x@qif = x@pre \wedge x@pre \geq 0 \wedge x@qif \leq 0$

lo cual implica $x@qif = 0$,

por lo que $[1..x] = [1.,0] = []$

entonces $\prod[1..x] = \prod[] = 1$

Entonces $f@qif = 1 \Rightarrow f@qif = \prod[1..x]$

La vida real

En la facultad, en el grupo de investigación LAFHIS, se desarrolló como tesis de licenciatura una herramienta de verificación formal de software automática. En otras palabras, se encarga de ver si un código cumple la especificación sin tener que hacer testing, sino que utiliza los mismos conceptos vistos en la materia. Esta herramienta utiliza un lenguaje de programación imperativo simplificado, y notaciones para explicar precondición, postcondición, invariante de ciclo y función variante.

Este trabajo fue la tesis de licenciatura de un estudiante de la carrera. El trabajo puede verse en `http://lafhis.dc.uba.ar/budapest/Welcome.html`. Hoy en día se está desarrollando una versión web que permite jugar con la herramienta. La herramienta online se encuentra en `http://lafhis-server.exp.dc.uba.ar/budapest/#pestTab`.

La vida real

```
 max(a[], r)
:? |a| > 0 //Esta es la precondición del problema
:! (forall k from 0 to |a|-1: a[r] >= a[k]) //Esta es la poscondición del mismo
:* r //Esta es una clasula modifica
{
        local i <- 1
        r <- 0
        while i < |a|
        // El siguiente es el invariante de ciclo
        :?! 1 <= i && i <= |a| && 0 <= r && r < |a| &&
                (forall k from 0 to i-1: a[r] >= a[k])
        :# |a| - i // Esta es la función variante con cota 0
        do {
                if (a[i] > a[r]) then
                        r <- i

                i <- i + 1
        }
}
```

Ejemplo de código para budapest