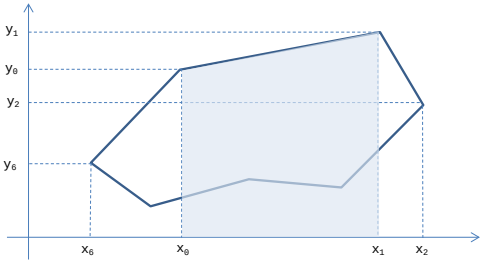
ISIS 1105 Diseño de algoritmos **Nombre:** _____
Semestre 2011-20 Parcial 2 **Sección:** _____
Octubre 20, 2011
Prof. Rodrigo Cardoso

1 **(30/100)** El perímetro de un polígono `Pol` está definido por la secuencia de vértices $\langle(x_0,y_0),(x_1,y_1),\ldots,(x_{n-1},y_{n-1}),(x_0,y_0)\rangle$. Suponga que la numeración de los vértices sigue el sentido del reloj y que no hay lados del polígono que se crucen. La figura muestra un ejemplo con n=7.



Una manera de calcular el área del polígono consiste en sumar las áreas trapezoidales formadas por vértices consecutivos. Recuerde que el área de un trapecio es "`(base mayor + base menor)` × `altura/2`". Las áreas de los trapecios pueden ser negativas.

Considere un algoritmo que responda a una especificación de la siguiente forma (en el contexto, los arreglos `x`, `y` guardan las coordenadas de los vértices de la secuencia que describe el polígono de n lados, n≥3):

```
[Ctx C: n:nat ∧ n≥3 ∧ ∧ x:[0..n-1]:real ∧ y:[0..n-1]:real
 ...
 {Inv P}
 {cota t }
 do  ...  od
 {R1}
 {R : a = "área del polígono Pol" }
]
```

1a (10/30) Defina una aserción `R1` que implique la poscondición `R` y que describa el final del procedimiento de sumas de áreas sugerido.

R1: $a = (+i|\ 0\le i<n : (y_{(i+1) \bmod n} + y_i)*(x_{(i+1) \bmod n} - x_i))/2$

[10/10]

Variante:
R1: $a = (+i|\ 0\le i<n-1 : (y_{i+1} + y_i)*(x_{i+1} - x_i))/2 + (y_0 + y_{n-1})*(x_0 - x_{n-1})/2$

[10/10]

1b (10/30) Defina un invariante `P` a partir de `R1`, usando la técnica de cambiar una constante por una variable. Defina una cota `t` correspondiente a su definición de `P`.

P: $a = (+i|\ 0\le i<k : (y_{(i+1) \bmod n} + y_i)*(x_{(i+1) \bmod n} + x_i))/2 \wedge 0\le k\le n$

[8/10]

t = n-k

[2/10]

Variante:

P: $a = (+i|\ 0 \le i < k : (y_{i+1} + y_i)*(x_{i+1} - x_i))/2 + (y_0 + y_{n-1})*(x_0 - x_{n-1})/2$

[8/10]

$t = n-k$

[2/10]

1c (10/30) Escriba el código correspondiente al invariante P y a la cota t establecidas.

[Ctx C: n:**nat** ∧ n≥3 ∧ ∧ x:[0..n-1]:**real** ∧ y:[0..n-1]:**real**

a,k:= 0,0;

[3/10]

{Inv P}
{cota t }

do k≠n

[3/10]

→ a,k:= a+(y[(i+1) mod n]+y[i])*(x[(i+1) mod n]-x[i])/2,k+1

[4/10]

od
{R1}
{R : a = "área del polígono Pol" }
]

Variante (dejar la división por 2 para el final; requiere un "samurai")

[Ctx C: n:**nat** ∧ n≥3 ∧ ∧ x:[0..n-1]:**real** ∧ y:[0..n-1]:**real**

a,k:= 0,0;

[3/10]

{Inv P}
{cota t }

do k≠n

[3/10]

→ a,k:= a+(y[(i+1) mod n]+y[i])*(x[(i+1) mod n]-x[i]),k+1

[3/10]

od

{R2: $2*a = (+i|\ 0 \le i < n : (y_{(i+1) \bmod n} + y_i)*(x_{(i+1) \bmod n} - x_i))$}

a:= a/2

[1/10]

{R1}
{R : a = "área del polígono Pol" }
]

Variante (no usar módulo al definir el invariante)

[Ctx C: n:**nat** ∧ n≥3 ∧ ∧ x:[0..n-1]:**real** ∧ y:[0..n-1]:**real**

```
a,k:= (y[0]+y[n-1])*(x[0]-x[n-1])/2,0;
```
[3/10]

```
{Inv P: a = (+i| 0≤i<k : (y_{i+1}+y_i)*(x_{i+1}-x_i))/2 + (y_0+y_{n-1})*(x_0-x_{n-1})/2}
{cota t: n-1-k }

do k≠n-1
```
[3/10]

```
     → a,k:= a+(y[i+1]+y[i])*(x[i+1]-x[i])/2,k+1
```
[4/10]

```
od

{R1}
{R : a = "área del polígono Pol" }
]
```

2 **(30/100)** Un *heap* sobre **nat** es un árbol binario que tiene la propiedad de *forma* (todas las hojas con diferencia de nivel de `0` o `1`, las más profundas a la izquierda), y la propiedad de *orden* (todo nodo contiene un número natural menor que todos los elementos que están bajo él).

Se sabe que un *heap* de n elementos se puede representar con un arreglo `b[1..n]` que guarda el árbol por niveles (raíz en `b[1]`, segundo nivel en `b[2]` y `b[3]` , …). Suponga que así se ha hecho y que exactamente uno de los elementos del arreglo contiene un valor que daña la propiedad de orden del *heap*. Se dice que b representa un *heap* defectuoso.

El siguiente ejemplo de un *heap* defectuoso muestra que el valor `4`, en la posición `2` del arreglo, tiene como hijo izquierdo un valor `3`, que viola la propiedad de orden del *heap*.

Dado un *heap* defectuoso, se quiere señalar una posición que corresponda a un nodo que tenga un descendiente inmediato que viole la propiedad de orden.
N.B. Documente su programa con aserciones, explicaciones de paradigmas, etc.
Se calificarán mejor las soluciones más eficientes.

2a (20/30) Escriba una solución para el problema

```
[ Ctx C: n:nat ∧ "b:[1..n] representa un heap defectuoso"
 {Pos R: "Un hijo de b[r] viola la propiedad de orden"}
]
```

Búsqueda lineal con certeza:

Espacio de búsqueda : `1..n`
[5/20]

Función sucesor : `suc: 1..n-1 → 1..n`
`suc.r = r+1, 1≤r<n`
[5/20]

Predicado de búsqueda: `b[r]≥b[2r] ∨ b[r]≥b[2r+1]`
[5/20]

```
 [ Ctx C: n:nat ∧ "b:[1..n] representa un heap defectuoso"

   r:= 1;
   do b[r]<b[2*r] ∧  b[r]<b[2*r+1] → r:= r+1 od

   {Pos R: " Un hijo de b[r] viola la propiedad de orden"}
]
```

[5/20]

La evaluación del predicado siempre está bien definida si se efectúa de izquierda a derecha (**cor**), debido a la forma del *heap*. El valor problema debe tener hijos. Si tiene dos, no hay problema de definición de la expresión; si tiene uno, es un hijo izquierdo y debe ser el que causa el desorden (porque se revisa por niveles).

Variante:

```
[ Ctx C: n:nat ∧ "b:[1..n] representa un heap defectuoso"

  r:= 1;
```

[2/20]

```
  {Inv P: 1≤r≤n ∧ (∀k| 1≤k<r : b[k]<b[2k] ∧  b[k]<b[2k+1])}
```

[7/20]

```
  {Cota t: n-r}
```

[2/20]

```
  do b[r]<b[2*r] ∧  b[r]<b[2*r+1]
```

[7/20]

```
        → r:= r+1
```

[2/20]

```
  od

  {Pos R: " Un hijo de b[r] viola la propiedad de orden"}
]
```

2b (10/30) Estime las complejidades espacial y temporal de su solución. Operación básica: comparación. Explique su respuesta.

`S(n) = O(1)` // Variable r

[5/10]

`T(n) = O(n)` // 2 comparaciones por iteración; máximo n iteraciones.

[5/10]

3 **(40/100)** Un autómata celular está diseñado para encontrar el máximo valor guardado en un arreglo `b[0..n-1]:nat`, `n>0`.

El autómata funciona cambiando los valores del arreglo a partir de un tiempo `0` (en el tiempo `0` el arreglo contiene los valores iniciales). Para calcular el valor de `b[i]` en el tiempo `t>0`, el autómata deja allí el máximo de `b[i]` y de sus vecinos a la izquierda y a la derecha, en el tiempo `t-1`. Note que `b[0]` no tiene vecino izquierdo y `b[n-1]` no tiene vecino derecho. El autómata se detiene si todos los valores del arreglo son iguales, precisamente, la respuesta.

El ejemplo muestra un cálculo que se detiene cuando `t=4`, cuando se quiere encontrar el mayor de los 6 valores: `3,16,10,16,17,12`:

t \ b	0	1	2	3	4	5
0	3	16	10	16	17	12
1	16	16	16	17	17	17
2	16	16	17	17	17	17
3	16	17	17	17	17	17
4	17	17	17	17	17	17

3a (30/40) Diseñe un algoritmo de programación dinámica para calcular el tiempo que demora el autómata en dar su respuesta. Use como lenguaje del problema la siguiente notación:
`x(t,i)` ≈ "valor de `b[i]` en el tiempo `t`", `0≤i<n,0≤t`.

Hay que encontrar el mínimo `t` tal que

```
(∀i| 0≤i<n : x(t,0)=x(t,i))
```

ya que, si esto pasa, será el mínimo `t` para el que:

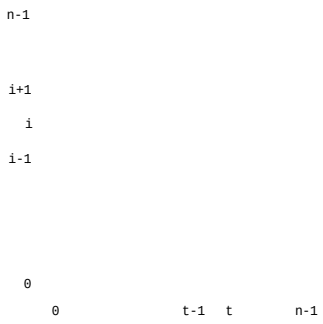
```
(∀i| 0≤i<n : b[0]=b[i]).
```

Recurrencia

```
x(t,i)= b[i]                                          , 0≤i<n,0=t
      = x(t-1,0) max x(t-1,1)                         , 0<t
      = x(t-1,i-1) max x(t-1,i) max x(t-1,i+1)        , 0<i<n-1,0<t
      = x(t-1,n-2) max x(t-1,n-1)                     , 0<t
```

[10/30]

Diagrama de necesidades



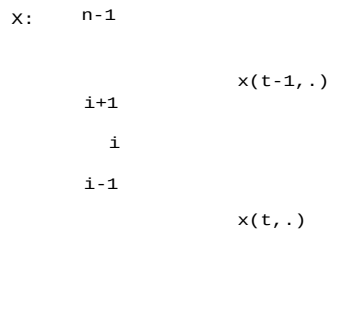
El diagrama de necesidades muestra una dependencia entre los valores del tiempo `t` y los del `t-1`. En los extremos hay una dependencia menos.

[10/30]

Estructura de datos e Invariante

El diagrama de necesidades sugiere tener un arreglo `X[0..n-1]` que guarde los valores de la función `x`. Una variable booleana `z` puede llevar la verdad de que se haya llegado a un mismo valor en todo el arreglo. La variable `y` se usa para recordar el valor `x(t-1,i-1)`, para poder calcular `x(t,i)` como el autómata lo hace[1]. El invariante sería:

`Inv:`



$\wedge$ z $\equiv$ ($\forall$j| 1$\leq$j<n: x(t-1,0)=x(t-1,j)) $\wedge$ 0<t
$\wedge$ (i>0 $\Rightarrow$ y=x(t-1,i-1))

[10/30]

Variante

Dos arreglos: `Xant,X[0..n-1]: nat`

`Inv: 0<t` $\wedge$ `0`$\leq$`i`$\leq$`n` $\wedge$ `Xant[0..n-1] = x(t-1,0..n-1)` $\wedge$ `X[0..i-1]=x(t,0..i-1)`
$\wedge$ z $\equiv$ ($\forall$j| 1$\leq$j<n: x(t-1,0)=x(t-1,j))

[7/30]

3b (10/40) Estime las complejidades espacial y temporal de su solución. Operación básica: asignación.

`S(n,b) = θ(n)` // Un arreglo y algunas variables.

[5/20]

Variante

`S(n,b) = θ(n)` // Dos arreglos y algunas variables.

[3/20]

`T(n,b) = θ(n²)`

Para esta última afirmación puede argumentarse que el procedimiento simula el accionar del autómata y el ciclo descrito se puede medir con la cota

[1] Si en la fórmula que calcula `x(t,i)`se usara `x(t,i-1)` en vez de `x(t-1,i)`, el cálculo estaría bien, pero el procedimiento no imitaría fielmente lo que hace el autómata e, incluso, podría llegar a calcular el máximo en menos iteraciones.

u = "No. de elementos diferentes del máximo"

Si la cota es positiva, hay valores diferentes del máximo. Al menos uno de ellos debe ser vecino de un valor igual al máximo; por tanto, en el siguiente paso el valor no igual al máximo debe cambiarse por el máximo, de modo que la cota baja efectivamente en 1, cada vez. Como al principio todos los valores pueden ser diferentes, la cota puede valer inicialmente n-1 y, en el peor caso, rebajar 1 cada vez. esto es, en n-1 pasos el autómata debe parar.

En cada paso, debe calcularse el vector X y chequear la estabilidad del resultado. Este cálculo es $\theta(n)$.

[5/10]

Un algoritmo solución, a manera de ilustración (un solo arreglo):

```
funct estable (a[0..n-1]:nat): bool
{Pos: estable ≡ (∀j| 0<j<n: a[0]=a[j])}
⌈ j,jcent:= 1,n;
  do j≠jcent      →    if a[0]=a[j]      then j:= j+1
                                         else jcent:= j
                       fi
  od;
  estable:= (j=n)
⌋

[Ctx: b[0..n-1]:nat ∧ b=B
 X:= b;
 i,t:= 0,1;
 z:= estable(b);

 {Inv: 0≤i<n ∧ 0<t ∧ X[0..i-1]=x(t,.) ∧ X[i..n-1]=x(t-1,.)
             ∧ z ≡ (∀j| 0<j<n: x(t-1,0)=x(t-1,j))
             ∧ (i>0 ⇒ y=x(t-1,i-1))                        }

do ¬z →     if    i=0   →    y:= x[0];
                             x[0]:= y max x[1]

            [] 0<i<n-1  →    y1:= x[i];
                             x[i]:= y max y1 max x[i+1];
                             y:= y1

            []    i=n-1 →    x[n-1]:= y max x[n-1]

            fi;

            if    i≠n-1      then  i:= i+1
                             else  i,t:= 0,t+1;
                                   z:= estable(X)
            fi
 od;

 {R1: (∀j| 0<j<n: x(t-1,0)=x(t-1,j))

 t:= t-1
{Pos: "El autómata para en el tiempo t"}
]
```