UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructuras de Datos I
Primer parcial – Segundo cuatrimestre de 2021

Ejercicio 1. El funcionamiento del siguiente programa se basa en el hecho de que la suma de los primeros n números naturales verifica la igualdad $1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$:

$P_c : \{n \geq 0 \land s = (n * (n + 1)) \text{ div } 2 \land t = n\}$

```
while (s > 0) do
  s := s - t;
  t := t - 1
endwhile
```

$Q_c : \{s = 0 \land t = 0\}$

proponer un invariante I para el ciclo y demostrar que se cumplen los siguientes puntos del teorema del invariante:

a) $(I \land \neg B) \Rightarrow Q_c$

b) $\{I \land B\}$ ⟨cuerpo del ciclo⟩ $\{I\}$

Ejercicio 2. El método de compresión por *run-length encoding* se basa en representar una **palabra** (secuencia de caracteres) a través de un **código**. Un código es una lista de pares, cada uno de los cuales contiene un fragmento de la palabra, acompañado del número de veces que dicho fragmento debe repetirse. Por ejemplo, la palabra `"axaxaxblablacaxax"` se puede representar con el código `[("ax",3),("bla",2),("c",1),("ax",2)]`. En general puede haber varios códigos para una misma palabra. Por ejemplo, la palabra `"banana"` se puede representar con los cuatro códigos siguientes, entre otros:

```
      [("banana",1)]  [("b",1),("an",2),("a",1)]
[("ba",1),("na",2)]  [("b",1),("a",1),("n",1),("a",1),("n",1),("a",1)]
```

Definimos el tipo Palabra como un renombre de $seq\langle \text{Char} \rangle$ y el tipo Código como un renombre de $seq\langle \text{Palabra} \times \mathbb{Z} \rangle$. Se pide:

a) Especificar el predicado `pred esDescompresión(`*cod* : `Código,` *pal* : `Palabra)` que es verdadero si *pal* es la palabra que resulta de descomprimir el código *cod*.

b) Especificar el problema `proc comprimir(in` *pal* : `Palabra, out` *cod* : `Código)` que dada una palabra devuelve algún código que la representa. El código resultante no debe contener pares que contengan fragmentos vacíos (ej. `("",3)`) ni repeticiones nulas (ej. `("ax",0)`).

c) Especificar el problema `proc optimizarCódigo(inout` *cod* : `Código)` que, dado un código, lo modifica para que siga representando la misma palabra pero de tal modo que el código modificado sea **óptimo**. Un código c es óptimo cuando cualquier otro código que represente la misma palabra cuesta *al menos* lo mismo que cuesta c. El costo se computa de acuerdo con algún criterio (irrelevante a los efectos de este ejercicio). Se puede suponer ya definida una función auxiliar que determina el costo de un código dado, que es siempre un número entero positivo:

`aux costo(`*cod* : `Código) : ` $\mathbb{Z}$

Ejercicio 3. Varias listas de enteros **no nulos** pueden combinarse para formar una sola lista de enteros, usando al 0 como separador. Por ejemplo, las listas "chicas" $[1,2,3]$, $[4,5]$ y $[6,7,8,9]$ se combinan para formar la lista "grande" $[1,2,3,0,4,5,0,6,7,8,9]$. Dada una posición k de la lista grande, se quieren recuperar dos índices: un índice i que indica a cuál de las listas chicas corresponde esa posición, y otro índice j que indica cuál es el índice dentro de la lista chica. Como parte de la solución a este problema se cuenta con el siguiente programa S en SmallLang y la siguiente especificación:

```
if (s[k] = 0) then
  i := i + 1;
  j := 0
else
  j := j + 1
endif
```

```
proc avanzar (in s: seq⟨ℤ⟩, in k : ℤ, inout i: ℤ, inout j: ℤ) {
      Pre {k + 1 < |s| ∧_L posicionesCorrespondientes(s, k, i, j)}
      Post {posicionesCorrespondientes(s, k + 1, i, j)}
}
pred posicionesCorrespondientes (s: seq⟨ℤ⟩, k : ℤ, i : ℤ, j : ℤ) {
          (0 ≤ k < |s| ∧ 0 ≤ j ≤ k)
     ∧_L  cantApariciones(subseq(s, 0, k − j), 0) = i
     ∧    (k − j = 0 ∨_L s[k − j − 1] = 0)
     ∧    cantApariciones(subseq(s, k − j, k), 0) = 0
}
aux cantApariciones (s: seq⟨T⟩, x : T) : ℤ =
   Σ_{t=0}^{|s|−1} if s[t] = x then 1 else 0 fi;
```

a) Calcular la precondición más débil del programa S con respecto a la postcondición de la especificación: *wp(S; Post)*.

b) Demostrar que el programa es correcto con respecto a la especificación propuesta.

Ejercicio 4.
Considerar el siguiente programa y su especificación (donde la función auxiliar `cantApariciones` se define igual que en el ejercicio 3):

```
     void f(vector <int> s, vector <int>& c){
L1:    int i = 0;
L2:    while (i < c.size()) {
L3:      c[i] = 0;
L4:      i = i + 1;
       }

L5:    int j = 0;
L6:    while (j < s.size()) {
L7:      int k = s[j];
L8:      if (c[k] == 3) {
L9:        c[k] = 0;
         } else {
L10:       c[k] = c[k] + 1;
         }
L11:     j = j + 1;
       }
L12:   return;
     }
```

```
proc f (in s: seq⟨ℤ⟩, inout c: seq⟨ℤ⟩) {
       Pre {
         C_0 = c
       ∧ (∀j ∈ ℤ)(0 ≤ j < |s| ⟶_L 0 ≤ s[j] < |c|)
       }
       Post {
          |c| = |C_0|
       ∧_L (∀i ∈ ℤ)(0 ≤ i < |c| ⟶_L
             c[i] = cantApariciones(s, i) mod 3)}
}
```

Cada caso de test propuesto debe contener la entrada y el resultado esperado.

a) Describir el diagrama de control de flujo (*control-flow graph*) del programa.

b) Escribir un conjunto de casos de test (o *test suite*) que cubra todas las sentencias. Mostrar qué líneas cubre cada test. Este conjunto de tests ¿cubre todas las decisiones? (Justificar).

c) Escribir un *test* que encuentre el defecto presente en el código (una entrada que cumple la precondición pero tal que el resultado de ejecutar el código no cumple la postcondición).

d) ¿Es posible escribir para este programa un *test suite* que cubra todas las decisiones pero que no encuentre el defecto en el código? En caso afirmativo, escribir el test suite; en caso negativo, justificarlo.