# Algoritmos y Estructuras de Datos I

Primer Cuatrimestre 2018

Guía Práctica 6
**Testing**

*Departamento de Computación*
*Facultad de Ciencias Exactas y Naturales*
*Universidad de Buenos Aires*

**Ejercicio 1.** ★ Sea el siguiente programa:

```
int max(int x, int y) {
L1:  int result = 0;
L2:  if (x<y) {
L3:    result = y;
     } else {
L4:    result = x;
     }
L5: return result;
}
```

Y los siguientes casos de test:

- test1:
  - Entrada x = 0, y=0
  - Resultado esperado result=0
- test2:
  - Entrada x = 0, y=1
  - Resultado esperado result=1

1. Describir el diagrama de control de flujo (control-flow graph) del programa `max`.
2. Detallar qué líneas del programa cubre cada test

| Test | L1 | L2 | L3 | L4 | L5 |
|---|---|---|---|---|---|
| test1 | | | | | |
| test2 | | | | | |

3. Detallar qué decisiones (branches) del programa cubre cada test

| Test | L2-True | L2-False |
|---|---|---|
| test1 | | |
| test2 | | |

4. Decidir si la siguiente afirmación es verdadera o falsa: *"El test suite compuesto por test1 y test2 cubre el 100 % de las líneas del programa y el 100 % de las decisiones (branches) del programa"*

**Ejercicio 2.** ★ Sea la siguiente especificación del problema de retornar el mínimo elemento entre dos números enteros:

`proc min` (in x: $\mathbb{Z}$, in y: $\mathbb{Z}$, out result: $\mathbb{Z}$) {
`Pre` $\{True\}$
`Post` $\{(x < y \rightarrow result = x) \wedge (x \geq y \rightarrow result = y)\}$
}

Un programador ha escrito el siguiente programa para implementar la especificación descripta:

```
int min(int x, int y) {
L1:  int result = 0;
L2:  if (x<y) {
L3:    result = x;
     } else {
L4:    result = x;
     }
L5: return result;
}
```

Y el siguiente conjunto de casos de test (test suite):

- minA:
  - Entrada x=1,y=0
  - Salida esperada true
- minA:
  - Entrada x=0,y=1
  - Salida esperada true

1. Describir el diagrama de control de flujo (control-flow graph) del programa `min`.
2. ¿La ejecución del test suite resulta en la ejecución de todas las líneas del programa `min`?
3. ¿La ejecución del test suite resulta en la ejecución de todas las decisiones (branches) del programa?
4. ¿Es el test suite capaz de detectar el defecto de la implementación del problema de encontrar el mínimo?
5. Agregar nuevos casos de tests y/o modificar casos de tests existentes para que el test suite detecte el defecto.

**Ejercicio 3.** ★ Sea la siguiente especificación del problema de sumar y una posible implementación en lenguaje imperativo.

```
proc sumar (in x: ℤ, in y: ℤ, out result: ℤ) {
       Pre {True}
       Post {result = x + y}
}
```

```
int sumar(int x, int y) {
L1:  int result = 0;
L2:  result = result + x;
L3:  result = result + y;
L4:  return result;
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `sumar`.
2. Escribir un conjunto de casos de test (o "test suite") que ejecute todas las líneas del programa `sumar`.

**Ejercicio 4.** Sea la siguiente especificación del problema de restar y una posible implementación en lenguaje imperativo:

```
proc restar (in x: ℤ, in y: ℤ, out result: ℤ) {
       Pre {True}
       Post {result = x − y}
}
```

```
int restar(int x, int y) {
L1:  int result = 0;
L2:  result = result + x;
L3:  result = result + y;
L4:  return result;
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `restar`.
2. Escribir un conjunto de casos de test (o "test suite") que ejecute todas las líneas del programa `restar`.
3. La línea L3 del programa `restar` tiene un defecto, ¿es el test suite descripto en el punto anterior capaz de detectarlo? En caso contrario, modificar o agregar nuevos casos de test hasta lograr detectarlo.

**Ejercicio 5.** Sea la siguiente especificación del problema de `signo` y una posible implementación en lenguaje imperativo:

```
proc signo (in x: ℝ, out result: ℤ) {
       Pre {True}
       Post {(result = 0 ∧ x = 0) ∨ (result = −1 ∧ x < 0) ∨ (result = 1 ∧ x > 0)}
}
```

```
int signo(float x) {
L1:  int result = 0;
L2:  if (x<0) {
L3:    result = −1;
     } else if (x>0){
L4:    result = 1;
     }
L5:  return result;
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `signo`.
2. Escribir un test suite que ejecute todas las líneas del programa `signo`.
3. ¿El test suite del punto anterior ejecuta todas las posibles decisiones ("branches") del programa?

**Ejercicio 6.** Sea la siguiente especificación del problema de `signo` y una posible implementación en lenguaje imperativo:

```
proc signo (in x: ℝ, out result: ℤ) {
       Pre {True}
       Post {(r = 0 ∧ x = 0) ∨ (r = −1 ∧ x < 0) ∨ (r = 1 ∧ x > 0)}
}
```

```
int signo(float x) {
L1:  int result = 0;
L2:  if (x<0) {
L3:    result = −1;
     }
L4:  return result;
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `signo`.
2. Escribir un test suite que ejecute todas las líneas del programa `signo`.
3. Escribir un test suite que ejecute todas las posibles decisiones ("branches") del programa.
4. Escribir un test suite que ejecute todas las líneas del programa pero no ejecute todos las decisiones del programa.

**Ejercicio 7.** ★ Sea la siguiente especificación:

```
proc fabs (in x: ℤ, out result: ℤ) {
       Pre {True}
       Post {result = |x|}
}
```

Y la siguiente implementación:

```
int fabs(int x) {
L1:  if (x<0 ) {
L2:    return -x;
     } else {
L3:    return +x;
  }
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `fabs`.
2. Escribir un test suite que ejecute todas las líneas y todos los branches del programa.

**Ejercicio 8.** ★ Sea la siguiente especificación del problema de `mult10` y una posible implementación en lenguaje imperativo:

```
proc mult10 (in x: Z, out result: Z) {
      Pre {True}
      Post {result = x * 10}
}
```

```
int mult10(int x) {
L1:  int result = 0;
L2:  int count = 0;
L3:  while (count<10) {
L4:     result = result + x;
L5:     count = count + 1;
    }
L6:  return result;
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `mult10`.
2. Escribir un test suite que ejecute todas las líneas del programa `mult10`.
3. ¿El test suite anterior ejecuta todas las posibles decisiones ("branches") del programa?

**Ejercicio 9.** Sea la siguiente especificación del problema de `sumar` y una posible implementación en lenguaje imperativo:

```
proc sumar (in x: Z, in y: Z, out result: Z) {
      Pre {True}
      Post {result = x + y}
}
```

```
int sumar(int x, int y) {
L1:   int sumando = 0;
L2:   int abs_y = 0;
L3:   if (y<0) {
L4:     sumando = -1;
L5:     abs_y = -y;
      } else {
L7:     sumando = 1;
L8:     abs_y = y;
      }
L9:   int result = x;
L10:  int count = 0;
L11:  while (count < abs_y) {
L12:    result = result + sumando;
L13:    count = count + 1;
      }
L14:  return result;
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `sumar`.
2. Escribir un test suite que ejecute todas las líneas del programa `sumar`.
3. Escribir un test suite que ejecute todas las posibles decisiones ("branches") del programa.

**Ejercicio 10.** Sea el siguiente programa que computa el máximo común divisor entre dos enteros.

```
int mcd(int x, int y) {
L1:  assert(x >= 0 && y >= 0) // Pre: x e y tienen que ser no negativos
L2:  int tmp = 0;
L3:  while(y != 0) {
L4:    tmp = x %y;
L5:    x = y;
L6:    y = tmp;
  }
L7:  return x; // gcd
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `mcd`.
2. Escribir un test suite que ejecute todas las líneas y todos los branches del programa.

**Ejercicio 11.** ★ Sea el siguiente programa que retorna diferentes valores dependiendo si $a$, $b$ y $c$, definen lados de un triángulo inválido, equilátero, isósceles o escaleno.

```
int triangle(int a, int b, int c) {
L1:    if (a <= 0 || b <= 0 || c <= 0) {
L2:        return 4; // invalido
    }
L3:    if (! (a + b > c && a + c > b && b + c > a)) {
L4:        return 4; // invalido
    }
L5:    if (a == b && b == c) {
L6:        return 1; // equilatero
    }
L7:    if (a == b || b == c || a == c) {
L8:        return 2; // isosceles
    }
L9:    return 3;  // escaleno
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `triangle`.
2. Escribir un test suite que ejecute todas las líneas y todos los branches del programa.

**Ejercicio 12.** ★ Sea la siguiente especificación del problema de `multByAbs` y una posible implementación en lenguaje imperativo:

```
proc multByAbs (in x: Z, in y:Z, out result: Z) {
      Pre {True}
      Post {result = x * |y|}
}
```

```
int multByAbs(int x, int y) {
L1:  int abs_y = fabs(y); // ejercicio anterior
L2:  if (abs_y<0) {
L3:    return -1;
  } else {
L4:    int result = 0;
```

```
L5:     int i = 0;
L6:     while (i<abs_y) {
L7:       result = result + x;
L8:       i = i + 1;
      }
L9:     return result;
    }
}
```

1. Describir el diagrama de control de flujo (control-flow graph) del programa `multByAbs`.
2. Detallar qué líneas y branches del programa no pueden ser cubiertos por ningún caso de test. ¿A qué se debe?
3. Escribir el test suite que cubra todas las líneas y branches que puedan ser cubiertos.

**Ejercicio 13.** Sea la siguiente especificación del problema de `vaciarSecuencia` y una posible implementación en lenguaje imperativo:

```
proc vaciarSecuencia (inout s: seq⟨ℤ⟩) {
       Pre {S_0 = s}
       Post {|s| = |S_0| ∧
          (∀j : ℤ)(0 ≤ j < |s| →_L s[j] = 0)}
}
```

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

1. Escribir el diagrama de control de flujo (control-flow graph) del programa `vaciarSecuencia`.
2. Escribir un test suite que cubra todos las líneas de programa (observar que un `for` contiene 3 líneas distintas)
3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

**Ejercicio 14.** Sea la siguiente especificación del problema de `existeElemento` y una posible implementación en lenguaje imperativo:

```
proc existeElemento (in s: seq⟨ℤ⟩, in e:ℤ, out result: Bool) {
       Pre {True}
       Post {result = True ↔ (∃j : ℤ)(0 ≤ j < |s| ∧_L s[j] = e)}
}
```

```
bool existeElemento(vector<int> s, int e) {
L1:         bool result = false;
L2,L3,L4:   for (int i=0; i<s.size();i++) {
L5:           if (s[i]==e) {
L6:             result = true;
L7:             break;
              }
            }
L8:           return result;
}
```

1. Escribir el diagrama de control de flujo (control-flow graph) del programa `existeElemento`.
2. Escribir un test suite que cubra todos las líneas de programa (observar que un `for` contiene 3 líneas distintas)

3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

**Ejercicio 15.** Sea la siguiente especificación del problema de `cantidadDePrimos` y una posible implementación en lenguaje imperativo:

proc `cantidadDePrimos` (in n: $\mathbb{Z}$, out result: $\mathbb{Z}$) {
  Pre $\{n \geq 0\}$
  Post $\{result = \sum_{i=2}^{i \leq n} \big(\text{if } ((\forall j : \mathbb{Z})((1 < j < i) \rightarrow i \bmod j \neq 0)) \text{ then 1 else 0 fi}\big)\}$
}

```
int cantidadDePrimos(int n) {
L1:          int result = 0;
L2,L3,L4:    for (int i=0; i<=n;i++) {
L5:            bool inc = esPrimo(i);
L6:            if (inc==true) {
L7:              result++;
               }
             }
L8:            return result;
}
/* procedimiento auxiliar */
bool esPrimo(int n) {
L9:          int result = true;
L10,L11,L12:  for (int i=2 ; i <n; i++) {
L13:           if (n %i == 0) {
L14:             result = false;
L15:             break;
               }
             }
L16:           return result;
}
```

1. Escribir los diagramas de control de flujo (control-flow graph) para `cantidadDePrimos` y la función auxiliar `esPrimo`.
2. Escribir un test suite que cubra todos las líneas de programa del programa `cantidadDePrimos` y `esPrimo`.
3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.

**Ejercicio 16.** Sea la siguiente especificación del problema de `esSubsecuencia` y una posible implementación en lenguaje imperativo:

proc `esSubsecuencia` (in s: $seq\langle\mathbb{Z}\rangle$, in r: $seq\langle\mathbb{Z}\rangle$,out result: Bool) {
  Pre $\{True\}$
  Post $\{result = True \leftrightarrow |r| \leq |s|$
    $\wedge_L (\exists i : \mathbb{Z})((0 \leq i < |s| \wedge i + |r| < |s|) \wedge_L (\forall j : \mathbb{Z})(0 \leq j < |r| \rightarrow_L s[i + j] = r[j]))\}$
}

```
1  bool esSubsecuencia(vector<int> s, vector<int> r) {
2    bool result = false;
3    int ultimoIndice = s.size() - r.size();
4    for (int i = 0; i< ultimoIndice; i++) {
5
6      /* obtener una subsecuencia de s */
7      vector<int> subseq = subsecuencia(s,i,r.size());
8
9      /* chequear si lasubsecuencia es igual a  r */
10     bool sonIguales = iguales(subseq, r);
```

```
11      if (sonIguales==true) {
12        result = true;
13        break;
14      }
15    }
16    return result;
17  }
18  /* procedimiento auxiliar subsecuencia*/
19  vector<int> subsecuencia(vector<int> s, int desde, int longitud) {
20    vector<int> rv;
21    int hasta = desde+longitud;
22    for (int i=desde;i<hasta ;i++) {
23      int elem = s[i];
24      rv.push_back(elem);
25    }
26    return rv;
27  }
28  /* procedimiento auxiliar iguales*/
29  bool iguales(vector<int> a, vector<int> b) {
30    bool result = true;
31    if (a.size()==b.size()) {
32      for (int i=0;i<a.size() ;i++) {
33        if (a[i]!=b[i]) {
34          result = false;
35          break;
36        }
37      }
38    } else {
39      result = false;
40    }
41    return result;
42  }
```

1. Escribir los diagramas de control de flujo (control-flow graph) para `esSubsecuencia` y las funcones auxiliares `subsecuencia` e `iguales`.

2. Escribir un test suite que cubra todos las líneas de programa *ejecutables* de todos los procedimientos. Observar que un `for` contiene 3 líneas distintas.

3. En caso que el test suite del punto anterior no cubriera todo los branches del programa, extenderlo de modo que logre cubrirlos.