Apuntes de Algoritmos y Estructura de Datos I

Federico Yulita

Segundo Cuatrimestre, 2019

Esta materia la cursé con Matías Lopez y Rosenfeld como profesor de las teóricas; en los talleres con Pablo Negri como JTP, Veronica Coy como Ayudante de Primera y Eric Brandwein y Maximiliano Martino como Ayudantes de Segunda; y en las prácticas con Pablo Brusco y Gabriela Di Piazza como JTPs, Cynthia Bonomi, Diego Raffo y Sebastián Vita como Ayudantes de Primera y Gabriel Budiño, Tomas Caballero y Marcelo Sancinetti como Ayudantes de Segunda. Si querés ver el resto de mis apuntes los podés encontrar en mi blog. Los ejercicios con un asterisco (*) son los que resolvimos en clase.

Índice

I Teórica **4**

1. Introducción a la Especificación de Problemas **5**
1.1. Contratos y Lógica Proposicional . . . 5
1.2. Secuencias . . . 8
1.3. Correctitud y Teorema del Invariante . . . 11
1.4. La Precondición Más Débil . . . 13

2. Testing Estructural **14**

3. Algoritmos Sobre Secuencias **15**
3.1. Algoritmos de Búsqueda . . . 15
3.1.1. Búsqueda Lineal . . . 15
3.1.2. Búsqueda Binaria . . . 16
3.2. Algoritmos de Ordenamiento . . . 17
3.2.1. Ordenamiento por Selección . . . 18
3.2.2. Ordenamiento por Inserción . . . 19
3.2.3. Dutch National Flag Problem . . . 20
3.3. Algoritmos Sobre Secuencias Ordenadas . . . 21
3.3.1. Apareo . . . 21

4. Algoritmos Sobre Strings **22**

II Práctica **24**

5. Guía 2: Introducción al Lenguaje de Especificación **25**
5.1. Ejercicio 4* . . . 25
5.1.1. (a) . . . 25
5.1.2. (c) . . . 25
5.1.3. (d) . . . 25
5.1.4. (e) . . . 25
5.1.5. (f) . . . 25
5.1.6. (h) . . . 25

5.1.7. (n) . . . 26
5.2. Ejercicio 6 . . . 26
5.2.1. (a)* . . . 26
5.2.2. (b) . . . 26
5.2.3. (c) . . . 26
5.2.4. (d)* . . . 26
5.2.5. (e)* . . . 26
5.2.6. (f) . . . 26

6. Guía 3: Especificación de Problemas **27**
6.1. Ejercicio 9* . . . 27
6.1.1. (e) . . . 27
6.2. Ejercicio 12* . . . 27
6.3. Ejercicio 14* . . . 27
6.3.1. (d) . . . 27
6.3.2. (e) . . . 27

7. Guía 4: Precondición Más Débil en SmallLang **28**
7.1. Ejercicio 2 . . . 28
7.1.1. (a) . . . 28
7.1.2. (b) . . . 28
7.1.3. (c) . . . 28
7.1.4. (d) . . . 28
7.1.5. (e) . . . 28
7.2. Ejercicio 4 . . . 28
7.2.1. (a) . . . 28
7.2.2. (d) . . . 29
7.2.3. (e) . . . 29
7.3. Ejercicio 6 . . . 29
7.3.1. (a) . . . 29
7.3.2. (b) . . . 29
7.3.3. (c) . . . 29
7.3.4. (d) . . . 30
7.4. Ejercicio 8 . . . 30
7.4.1. (a) . . . 30
7.4.2. (c) . . . 30
7.4.3. (d) . . . 31

8. Guía 5: Demostración de Corrección de Ciclos en SmallLang **31**
8.1. Ejercicio 2 . . . 31
8.2. Ejercicio 3 . . . 32
8.2.1. (a) . . . 32
8.2.2. (b) . . . 34
8.2.3. (c) . . . 34
8.2.4. (d) . . . 34
8.3. Ejercicio 4 . . . 34
8.3.1. (a) . . . 34
8.3.2. (b) . . . 34
8.4. Ejercicio 6 . . . 36
8.4.1. (a) . . . 36
8.4.2. (b) . . . 36
8.4.3. (c) . . . 37
8.5. Ejercicio 7 . . . 38
8.5.1. (a) . . . 38
8.5.2. (b) . . . 38

8.5.3. (c) . . . 39

9. Guía 6: Testing **39**
9.1. Ejercicio 3 . . . 40
9.2. Ejercicio 5 . . . 40
9.3. Ejercicio 6 . . . 41
9.4. Ejercicio 8 . . . 41
9.5. Ejercicio 9 . . . 42
9.6. Ejercicio 11 . . . 43
9.7. Ejercicio 14 . . . 44

10. Guía 7: Tiempo de Ejecución de Peor Caso de un Programa **44**
10.1. Ejercicio 2 . . . 44
10.1.1. (a) . . . 44
10.1.2. (b) . . . 45
10.1.3. (c) . . . 45
10.1.4. (d) . . . 45
10.1.5. (e) . . . 45
10.2. Ejercicio 5 . . . 45
10.3. Ejercicio 5 . . . 46
10.4. Ejercicio 9 . . . 46
10.5. Ejercicio 12 . . . 47

11. Guía 8: Búsqueda y Ordenamiento **48**
11.1. Ejercicio 4 . . . 48
11.1.1. (a) . . . 48
11.1.2. (b) . . . 48
11.1.3. (c) . . . 48
11.2. Ejercicio 5 . . . 49
11.2.1. (a) . . . 49
11.3. Ejercicio 6 . . . 50
11.3.1. (a) . . . 50
11.3.2. (b) . . . 51
11.3.3. (c) . . . 51
11.3.4. (d) . . . 51
11.3.5. (e) . . . 52
11.4. Ejercicio 8 . . . 52
11.5. Ejercicio 10 . . . 53
11.6. Ejercicio 12 . . . 53
11.7. Ejercicio 14 . . . 53

Parte I
Teórica

Sir Charles Antony Richard Hoare (1934)

1. Introducción a la Especificación de Problemas

1.1. Contratos y Lógica Proposicional

El objetivo de la materia es aprender a programar lenguajes imperativos . Vamos a describirlos en un lenguaje formal y estudiar algunos algoritmos para resolver problemas.

Definimos una **Computadora** es una máquina que procesa información automáticamente de acuerdo a un programa almacenado. Su función es procesar información de forma automática siguiendo un programa almacenado en su memoria. Un **Algoritmo** es una descripción de los pasos precisos para resolver un problema a partir de datos de entrada adecuados. El objetivo de un algoritmo es resolver algún problema dado. Un ejemplo de algoritmo es la Criba de Eratóstenes (276 AC - 194 AC) que devuelve todos los números primos entre 2 y n. Consiste en escribir todos los números entre 2 y n y, empezando desde el primer número en la lista, tachar todos los números divisibles por ese número. Se repite el proceso hasta haber pasado por todos los números no tachados y resulta que esos números son los primos.

Una **Especificación** es una descripción del problema a resolver. Vamos a buscar describir la especificación en un lenguje formal para implementar un algoritmo. El **Programa** finalmente va a ser la implementación del algoritmo que la computadora puede ejecutar para resolver el problema. La especificación es un **contrato** que da las propiedades de los datos de entrada y de la solución. Estos datos de entrada los llamamos **Parámetros** y son conocidos al momento de ejecutar el programa.

A los parámetros los vamos a clasificar dependiendo del **Tipo de Dato** que sea. El tipo de dato determina qué operaciones pueden utilizarse en el parámetro. Definimos un **Contrato** como un acuerdo entre el programador de una función y el usuario. Por ejemplo, si quiero programar un programa que calcule la raíz cuadrada de un número puedo tener un contrato con el usuario que establezca que el programa funciona para número reales positivos. Entonces, hago el programa asumiendo que el parámetro es un número real positivo. Esto es útil ya que simplifica el problema a resolver (o al menos lo separa en partes). Una **Precondición** establece una condición para los parámetros y la **Poscondición** asegura que los resultados son correctos.

Vamos a definir las especificaciones así:

```
proc nombre(parámetros){
    Pre{P}
    Post{Q}
}
```

Ejemplo

Para la raíz cuadrada:

```
proc raizCuadrada(in x : ℝ, out result : ℝ){
    Pre{x ⩾ 0}
    Post{result * result = x ∧ result ⩾ 0}
}
```

Para sumar:

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

Para hallar un número mayor:

```
proc cualquieraMayor(in x : ℤ, out result : ℤ){
```

```
        Pre{True}
        Post{result > x}
    }
```

Cuando se cumple el contrato para un programa entonces decimos que el programa es **Correcto**. Si el parámetro de entrada y el de salida es el mismo definimos un parámetro de tipo *inout*. Por ejemplo, si queremos incrementar un número por 1 podemos hacer esto:

```
proc incremento(inout a : ℤ){
    Pre{a = A_0}
    Post{a = A_0 + 1}
}
```

Definimos una variable auxiliar A_0 en la precondición para poder definir la poscondición.

Llamamos **Sobre-especificación** a una especificación con una poscondición más restrictiva de lo que necesita ser. Por ejemplo, para el ejemplo de *cualquieraMayor* podemos pedir que la poscondición sea $result = x + 1$. Funciona, pero es muy restrictivo. Llamamos **Sub-especificación** a una especificación con una precondición más restrictiva de lo que necesita ser.

Definimos las funciones *enum* que enumera los elementos de algún tipo de dato y *ord* que te dice la posición de algún dato. Por ejemplo:

```
enum Día{
    LUN, MAR, MIER, JUE, VIER, SAB, DOM
}
```

Entonces:

```
ord(LUN)=0
Día(2)=MIE
JUE<VIE
```

Usualmente vamos a usar los tipos de datos *Integer*, *Real*, *Bool* y *Char*. Definimos también el tipo *Tupla* como un tipo que tiene dos o más elementos del mismo tipo.

Vamos a usar **Funciones Auxiliares** y **Predicados Auxiliares** para simplificar el código. Las funciones las usamos así:

```
aux f(argumentos): tipo = expresión;
```

f es el nombre de la función. Para los predicados es parecido:

```
pred p(argumentos){f}
```

p es el nombre del predicado.

Ejemplo

```
aux suc(x : ℤ): ℤ = x + 1
pred esPar(n : ℤ){(n  mód 2 = 0)}
```

Vamos a usar también los operadores de lógica proposicional de matemática ($\wedge$, $\vee$, $\neg$, $\leftrightarrow$, etc). También vamos a usar $\perp$ para una proposición indefinida. Esto se llama **Semántica Trivaluada**. Vamos a leer de izquierda a derecha, así que si encontramos una proposición indefinida en alguna expresión entonces el resultado es indefinido, pero si no hay que seguir leyendo la expresión. Si en algún momento sabemos el resultado de la expresión sin necesidad de seguir leyendo entonces se establece el resultado. Por lo tanto, definimos las operaciones $\wedge_L$ (**Conditional AND**) y $\vee_L$ (**Conditional OR**) como los operadores $\wedge$ y $\vee$ respectivamente pero que nos salvan de evaluar proposiciones indefinidas. Por ejemplo:

- $p = T$, $q = \bot$: $p \wedge q = \bot$.
- $p = F$, $q = \bot$: $p \wedge q = \bot$.
- $p = T$, $q = \bot$: $p \wedge_L q = \bot$.
- $p = F$, $q = \bot$: $p \wedge_L q = F$.

Esta secuencialidad en la lógica es útil para interpretar expresiones como lo hace la computadora. De forma similar definimos el operador $\rightarrow_L$ (**Conditional IF**).

Ejemplo

```
proc piesimo(in i : ℤ, out result : ℤ){
    Pre{i > 0}
    Post{result = ⌊π * 10^i⌋  mód 10}
}
```

Veamos como hacer un predicado para un número primo usando este tipo de lógica proposicional:

```
pred esPrimo(n : ℤ){
    n > 1 ∧ (∀n' : ℤ) (1 < n' < n →_L n  mód n' ≠ 0)
}
```

Notemos también como hicimos para definir variables usando $\forall$. De la misma forma se hace para usar $\exists$ o $\exists!$. Entonces:

```
proc primo(in n : ℤ, out result : 𝔹){
    Pre{n > 1}
    Post{result = True ↔ esPrimo(n)}
}
```

Ejemplo

Especifiquemos un procedimiento que calcule el máximo común divisor entre dos números.

```
proc mcd(in n : ℤ, in m : ℤ, out result : ℤ){
    Pre{n ⩾ 1 ∧ m ⩾ 1}
    Post{n  mód result = 0 ∧ m
    mód result = 0 ∧ ¬ (∃p : ℤ) (p > result ∧ n  mód p = 0 ∧ m  mód p = 0)}
}
```

Ahora especifiquemos un semáforo. Vamos a definir dos variables de tipo *Bool* que representan el color de la luz. El semáforo pasa de rojo a rojo y amarillo, de rojo y amarillo a verde, de verde a amarillo y de amarillo a rojo.

```
pred esRojo(v, a, r : 𝔹){v = False ∧ a = False ∧ r = True}
pred esRojoAmarillo(v, a, r : 𝔹){v = False ∧ a = True ∧ r = True}
pred esAmarillo(v, a, r : 𝔹){v = False ∧ a = True ∧ r = False}
pred esVerde(v, a, r : 𝔹){v = True ∧ a = False ∧ r = False}
pred esValido(v, a, r : 𝔹){
    esRojo(v, a, r)
    ∨ esAmarillo(v, a, r)
```

```
        ∨esVerde(v, a, r)
        ∨esRojoAmarillo(v, a, r)
    }
    proc iniciar(out v, a, r : 𝔹){
        Pre{True}
        Post{v =True∧a=False∧r =False}
    }
    proc avanzar(inout v, a, r : 𝔹){
        Pre{esValido{v, a, r}∧v = V0 ∧ r = R0 ∧ a = A0}
        Post{
            (esRojo(V0, A0, R0)→esRojoAmarillo(v, a, r))
            ∧(esRojoAmarillo(V0, A0, R0)→esVerde(v, a, r))
            ∧(esVerde(V0, A0, R0)→esAmarillo(v, a, r))
            ∧(esAmarillo(V0, A0, R0)→esRojo(v, a, r))
        }
    }
```

Notemos que si, por ejemplo, el semáforo está en verde entonces todas las expresiones de las poscondición menos las tercera empiezan con un *False* y entonces si o si son *True*. Por lo tanto, la poscondición sólo va a ser verdadera si se cumple la tercera expresión.

1.2. Secuencias

Definimos una **Secuencia** como varios elementos del mismo tipo en cierto orden. Notamos $\text{Seq}\langle T\rangle$, donde T indica el tipo. Vamos a usar la función $length(a : \text{Seq}\langle T\rangle) : \mathbb{Z}$ para notar la longitud de la secuencia a (también notamos $a.length$ o $|a|$) y la **Indexación** $\text{Seq}\langle T\rangle\,[i : \mathbb{Z}] : T$ que indica el i-ésimo elemento de la secuencia (también notamos $a\,[i]$), si no se cumple $0 \leqslant i < |a|$ se indefine. Consideramos que dos secuencias son iguales si tienen la misma cantidad de elementos y si dada una posición el elemento en esa posición en ambas secuencias es el mismo para cualquier posición. También definimos las funciones $head(a : \text{Seq}\langle T\rangle) : T$ y $tail(a : \text{Seq}\langle T\rangle) : T$ que dan el primer elemento de una secuencia a y el resto de la secuencia respectivamente, si $|a| = 0$ se indefine. La función $addFirst(t : T, a : \text{Seq}\langle T\rangle) : \text{Seq}\langle T\rangle$ es una función que agrega un elemento t a una secuencia a del mismo tipo en la primera posición. Definimos también la **Concatenación** como $concat(a : \text{Seq}\langle T\rangle, b : \text{Seq}\langle T\rangle) : \text{Seq}\langle T\rangle$ que es una función que toma dos secuencias del mismo tipo y las junta una al final de la otra (también notamos $a + +b$). La función $subseq(a : \text{Seq}\langle T\rangle, d, h : \mathbb{Z}) : \text{Seq}\langle T\rangle$ que te da una secuencia conformada por los elementos de la secuencia a con los elementos entre las posiciones d (inclusive) y h (exclusive), si no se indefine. La función $setAt(a : \text{Seq}\langle T\rangle, i : \mathbb{Z}, val : T) : \text{Seq}\langle T\rangle$ que cambia el valor del i-ésimo elemento de la secuencia a a val. Las funciones entonces son:

- Logitud: $length(a : \text{Seq}\langle T\rangle) : \mathbb{Z}$ (notamos $|a|$).
- Indexación: $\text{Seq}\langle T\rangle\,[i : \mathbb{Z}] : T$ (notamos $a\,[i]$).
- Cabeza: $head(a : \text{Seq}\langle T\rangle) : T$.
- Cola: $tail(a : \text{Seq}\langle T\rangle) : T$.
- Añadir: $addFirst(t : T, a : \text{Seq}\langle T\rangle) : \text{Seq}\langle T\rangle$.
- Concatenación: $concat(a : Seq\langle T\rangle, b : \text{Seq}\langle T\rangle) : \text{Seq}\langle T\rangle$ (notamos $a + +b$).
- Subsecuencia: $subseq(a : \text{Seq}\langle T\rangle, d, h : \mathbb{Z}) : \text{Seq}\langle T\rangle$.
- Cambiar: $setAt(a : \text{Seq}\langle T\rangle, i : \mathbb{Z}, val : T) : \text{Seq}\langle T\rangle$.

Ejemplo

Hagamos un predicado que sea verdadero sólo si una secuencia de enteros sólo posee enteros mayore a 5. Entonces:

```
pred seq_gt_five(s :Seq⟨ℤ⟩){
```
$$(\forall i : \mathbb{Z})\,(0 \leqslant i < |s| \rightarrow_L s\,[i] > 5)$$
```
}
```

Ahora hagamos un predicado que sea verdadero sólo si hay algún elemento en la secuencia que sea par y mayor que 5:

```
pred seq_has_elem_even_gt_five(s :Seq⟨ℤ⟩){
```
$$(\exists i : \mathbb{Z})\,(0 \leqslant i < |s| \wedge_L (s\,[i] \mod 2 = 0 \wedge s\,[i] > 5))$$
```
}
```

Ahora hagamos otro que sea verdadero sólo si la lista es vacía:

```
pred isEmpty(s :Seq⟨T⟩){
```
$$|s| = 0$$
```
}
```

Ahora otro que sea verdadero si el elemento e está en la secuencia:

```
pred has(s :Seq⟨T⟩, e : T){
```
$$(\exists i : \mathbb{Z})\,(0 \leqslant i < |s| \wedge_L s\,[i] = e)$$
```
}
```

Para este último predicado se suele notar $e \in s$. Ahora hagamos un predicado que dadas dos secuencias s_1 y s_2 sea verdadero si s_2 es s_1 pero con el elemento de la posición i igual a e. En caso de que $0 \leqslant i < |s_2|$ no se cumpla entonces el predicado sea falso sólo si las secuencias no son iguales.

```
pred isSetAt(s1, s2 :Seq⟨T⟩, i : ℤ, e : T){
```
$$(0 \leqslant i < |s_2| \wedge_L \mathit{setAt}(s_2, i, e) = s_1) \vee (\neg\,(0 \leqslant i < |s_2|) \wedge (s_1 = s_2))$$
```
}
```

Otra forma de hacerlo es:

```
pred isSetAt(s1, s2 :Seq⟨T⟩, i : ℤ, e : T){
```
$$(0 \leqslant i < |s_2| \rightarrow_L s_1 = \mathit{setAt}(s_2, i, e))$$
$$\wedge(\neg\,(0 \leqslant i < |s_2|) \rightarrow s_1 = s_2)$$
```
}
```

Vamos a usar sumatorias también que se notan usando la misma notación que en matemática:

$$\sum_{i=\text{from}}^{\text{to}} f\,(i)\,.$$

Veamos entonces como hacer un predicado que sea verdadero sólo si un número es primo:

```
pred esPrimo(n : ℤ){
```

$$n > 1 \wedge_L \left(\sum_{i=2}^{n-1} (\text{if } (n \mod i = 0) \text{ then } 1 \text{ else } 1 \text{ fi}) \right) = 0$$

}

Ejemplo

Escribamos una expresión que represente la sumatoria de los elementos pares de una secuencia s:

$$\sum_{i=0}^{|s|-1} (\text{if } (i \mod 2 = 0) \text{ then } s[i] \text{ else } 0 \text{ fi})$$

Lo mismo para la productoria:

$$\prod_{i=\text{from}}^{\text{to}} f(i).$$

Ejemplo

Definir un predicado que sea verdadero si una secuencia es una permutación de la otra:

aux $\#apariciones$(s :Seq$\langle T \rangle$, $e : T$): $\mathbb{Z}$ =

$$\sum_{i=0}^{|s|-1} (\text{if } s[i] = e \text{ then } 1 \text{ else } 0 \text{ fi})$$

pred esPermutacion(s_1, s_2 :Seq$\langle T \rangle$){

$(\forall e : T)\,(\#apariciones\,(s_1, 2) = \#apariciones\,(s_2, e))$

}

Veamos que con la función auxiliar definida usamos un método general para contar elementos de un conjunto A con ciertas propiedades P:

$$\sum_{i \in A} (\text{if } P(i) \text{ then } 1 \text{ else } 0 \text{ fi}).$$

Notamos esta expresión como $\#\{i \in A : P(i)\}$.

Vamos a notar los comentarios usando la notación:

```
/* Esto es un comentario */
```

Es útil para poder hacer comentarios en el código sin que se confunda con código real.

Las **Matrices** son secuencias de secuencias de igual longitud. Las notamos como Mat$\langle \mathbb{Z} \rangle$. Entonces:

aux *filas*(m :Mat$\langle \mathbb{Z} \rangle$): $\mathbb{Z} = |m|$;
aux *columnas*(m :Mat$\langle \mathbb{Z} \rangle$): $\mathbb{Z} =$ if $filas\,(m) > 0$ then $|m[0]|$ else 0 fi;
pred *esMatriz*(m :Seq$\langle$Seq $\langle \mathbb{Z} \rangle\rangle$){

$(\forall i : \mathbb{Z})\,(0 \leqslant i < filas\,(m) \rightarrow_L (|m[i]| > 0 \wedge (\forall j : \mathbb{Z})\,(0 \leqslant j < filas\,(m) \rightarrow_L |m[i]| = |m[j]|)))$

}

1.3. Correctitud y Teorema del Invariante

Definimos un **Estado** como el conjunto con el valor de las variables de un programa en algún momento. Entonces, la **Ejecución** de un programa es una sucesión de estados. Definimos **Asignación** como el cambio del estado de un programa. Entonces, llamamos a un programa S **Correcto** respecto a una especificación con una precondición $\mathbb{P}$ y una poscondición $\mathbb{Q}$ si siempre que el programa empieza con un estado que cumple $\mathbb{P}$ termina su ejecución y el estado final cumple $\mathbb{Q}$. Notamos a un programa correcto usando la **Tripla de Hoare** $\{\mathbb{P}\}\, S\, \{\mathbb{Q}\}$. Sea E_0 el estado inicial y E_f el estado final del programa S entonces $\{\mathbb{P}\}\, S\, \{\mathbb{Q}\} \leftrightarrow (\exists E_f) \wedge (E_0 \rightarrow \mathbb{P}) \wedge (E_f \rightarrow \mathbb{Q})$. Definimos un **Ciclo** a un programa que se ejecuta múltiples veces siempre y cuando se cumpla una condición que llamamos **Guarda**. Llamamos **Invariante** a alguna expresión que en un ciclo no cambia. Un predicado $\mathbb{I}$ es invariante de un ciclo con guarda $\mathbb{B}$ y cuerpo S si:

1. $\mathbb{I}$ vale antes de comenzar el ciclo.
2. Si vale $\mathbb{I} \wedge \mathbb{B}$ al comenzar una iteración arbitraria, entonces sigue valiendo I al finalizar el cuerpo del ciclo.

Entonces, sean se cumplen las condiciones entonces vale:

Teorema 1. *Sean $\mathbb{P}_\mathbb{C}$ la precondición del ciclo y $\mathbb{Q}_\mathbb{C}$ la poscondición del ciclo el programa es parcialmente correcto si se cumple:*

- $\mathbb{P}_\mathbb{C} \rightarrow \mathbb{I}$.
- $\{\mathbb{I} \wedge \mathbb{B}\}\, S\, \{\mathbb{I}\}$.
- $\mathbb{I} \wedge \neg\mathbb{B} \rightarrow \mathbb{Q}_\mathbb{C}$.

Este es el **Teorema del Invariante**. Si este teorema se cumple entonces el ciclo es sólo parcialmente correcto porque el teorema no verifica que el ciclo termine eventualmente.

Ejemplo

Consideremos la siguiente especificación y código (en C++):

proc *sumatoria* (in $n : \mathbb{Z}$, out $s : \mathbb{Z}$){

Pre{$(n \geqslant 0) \wedge (j = 1) \wedge (s = 0)$}

Post{

$$s = \sum_{k=1}^{n} k$$

}

}

```
int sumatoria(int n){
    while(j <= n){
        s = s + j;
        j = j + 1;
    }
    return s;
}
```

Busquemos la invariante $\mathbb{I}$ del programa. Notemos:

$$\mathbb{P} \equiv (n \geqslant 0) \wedge (j = 1) \wedge (s = 0)\,,\ \mathbb{Q} \equiv s = \sum_{k=1}^{n} k,\ \mathbb{B} \equiv j \leqslant n.$$

Por la tercera condición del teorema entonces necesitamos:

$$\neg\left(j \leqslant n\right) \wedge \mathbb{I} \equiv \left(j > n\right) \wedge \mathbb{I} \rightarrow s = \sum_{k=1}^{n} k.$$

Luego de una iteración del ciclo el valor de s se incrementa por j y el de j por 1. Veamos los estados luego de algunas iteraciones:

$$\{n = N_0, j = 1, s = 0\} \rightarrow \{n = N_0, j = 2, s = 1\} \rightarrow \{n = N_0, j = 3, s = 3\} \rightarrow \{n = N_0, j = 4, s = 6\} \rightarrow ...$$

Entonces, propongamos:

$$\mathbb{I} \equiv s = \sum_{k=1}^{j-1} k.$$

Notemos que esto tiene sentido ya que cumple con los valores de s en los estados que vimos. Sin embargo, notemos que para cumplir la segunda condición del teorema necesitamos pedir algo más, ya que:

$$\neg\left\{\left(j \leqslant n\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right)\right\} S \left\{s = \sum_{k=1}^{j-1} k\right\}.$$

Por ejemplo, si $j = -1$ entonces s, según el S, al principio vale 0 y al final vale -1 pero según esa expresión al final vale 0. Entonces, debemos reforzar el invariante agregando la condición de que $j \geqslant 0$:

$$\mathbb{I} \equiv \left(j \geqslant 0\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right).$$

Notemos que ahora esta invariante no cumple la tercera condición del teorema ya que:

$$\left(j \geqslant 0\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right) \wedge \left(j > n\right) \not\rightarrow s = \sum_{k=1}^{n} k.$$

Por ejemplo, si $j = n+2$ entonces la sumatoria suma un término de más. Entonces, reforcemos la invariante con esta nueva condición:

$$\mathbb{I} \equiv \left(0 \leqslant j \leqslant n+1\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right).$$

Entonces:

$$\left(n \geqslant 0\right) \wedge \left(j = 1\right) \wedge \left(s = 0\right) \rightarrow \left(0 \leqslant j \leqslant n+1\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right) \checkmark$$

$$\left\{\left(0 \leqslant j \leqslant n+1\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right) \wedge \left(j \leqslant n\right)\right\} S \left\{\left(0 \leqslant j \leqslant n+1\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right)\right\} \checkmark$$

$$\left(0 \leqslant j \leqslant n+1\right) \wedge \left(s = \sum_{k=1}^{j-1} k\right) \wedge \left(j > n\right) \rightarrow s = \sum_{k=1}^{n} k \checkmark$$

Por lo tanto, esa invariante es correcta. Como el programa cumple con el teorema del invariante entonces es parcialmente correcto.

1.4. La Precondición Más Débil

Usemos un lenguaje de programación más simple que C++ llamado *SmallLang*. Es bastante intuitivo y va a ser fácil de leer, sólo define asignación, condicionales, while-loops y skips. Consideremos el siguiente programa:

$$\{x \geqslant 4 \wedge y < -2\}\, x \stackrel{\text{def}}{:=} x + 1\, \{x \geqslant 5 \wedge y < 0\}$$

Notemos que el programa no cambia a la variable y y en la poscondición requerimos que $y < 0$ por lo tanto requerimos que en la precondición pidamos $y < 0$. Sin embargo, en esta precondición pedimos que $y < -2$. Esta condición cumple con lo que esperamos en la poscondición pero es más restrictiva (más *fuerte*), entonces vamos a buscar la precondición más *débil* dada una poscondición y algún código. Supongamos que tenemos un programa S con poscondición $\mathbb{Q}$ y un predicado que chequea si la precondición es la más debil (Notamos $\text{wp}(\boldsymbol{S}, \mathbb{Q})$). En este caso vale que:

$$\{x \geqslant 4 \wedge y < -2\} \rightarrow_L \text{wp}\left(x \stackrel{\text{def}}{:=} x + 1, \{x \geqslant 5 \wedge y < 0\}\right)$$

Es decir, sea $\mathbb{P}$ una precondición válida entonces $\mathbb{P} \rightarrow_L wp(\boldsymbol{S}, \mathbb{Q})$. Definimos a $S_1; S_2$ como un programa que es S_1 y luego S_2 sólo si S_1 y S_2 son programas. Definimos $\text{def}(E)$ como un predicado que indica las condiciones necesarias para que la expresión E esté definida. Siempre vamos a asumir que las variables de una expresión están definidas y que son del tipo de dato correcto. También, vamos a definir el predicado $\mathbb{Q}^x_E$ como el predicado que se obtiene de reemplazar en $\mathbb{Q}$ todas las apariciones libres de la variable x por E. Dadas estas definiciones definimos los siguientes axiomas:

1. $\text{wp}\left(x \stackrel{\text{def}}{:=} \boldsymbol{E}, \mathbb{Q}\right) \equiv \text{def}(E) \wedge_L \mathbb{Q}^x_E$.
2. $\text{wp}(\boldsymbol{skip}, \mathbb{Q}) \equiv \mathbb{Q}$.
3. $\text{wp}(\boldsymbol{S_1; S_2}, \mathbb{Q}) \equiv \text{wp}(\boldsymbol{S_1}, \text{wp}(\boldsymbol{S_2}, \mathbb{Q}))$.
4. $S = \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \implies \text{wp}(\boldsymbol{S}, \mathbb{Q}) \equiv \text{def}(B) \wedge_L ((B \wedge \text{wp}(\boldsymbol{S_1}, \mathbb{Q})) \vee (\neg B \wedge \text{wp}(\boldsymbol{S_2}, \mathbb{Q})))$.

De el último axioma podemos construir el siguiente teorema:

Teorema 2. *Sean S_1 y S_2 programas con un condicional B y $\mathbb{P}$ y $\mathbb{Q}$ la precondición y poscondición respectivamente entonces:*

$$(\mathbb{P} \rightarrow \text{def}(B)) \wedge (\{\mathbb{P} \wedge B\}\, S_1\, \{\mathbb{Q}\}) \wedge (\{\mathbb{P} \wedge \neg B\}\, S_2\, \{\mathbb{Q}\}) \rightarrow \{\mathbb{P}\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{\mathbb{Q}\}$$

Ahora veamos los ciclos. Supongamos que tenemos un ciclo:

$$\text{while } B \text{ do } S \text{ endwhile}$$

Definimos al predicado $H_k(\mathbb{Q})$ como el predicado que define el conjunto de estados a partir de los cuales la ejecución del ciclo termina en exactamente k iteraciones. Notemos que entonces si el ciclo realiza a lo sumo k iteraciones entonces:

$$\text{wp}(\text{while } B \text{ do } S \text{ endwhile}, \mathbb{Q}) \equiv \vee_{i=0}^{k} H_i(\mathbb{Q})$$

Se define entonces el quinto axioma:

5. $\text{wp}(\text{while } B \text{ do } S \text{ endwhile}, \mathbb{Q}) \equiv (\exists i \geqslant 0)(H_i(\mathbb{Q}))$

Con esto podemos ver el elemento que nos faltaba de los ciclos: su terminación.

Teorema 3. *Sea $\mathbb{V}$ el producto cartesiano de los dominios de las variables del programa y sea $\mathbb{I}$ un invariante del ciclo while B do S endwhile si existe una función $f : \mathbb{V} \rightarrow \mathbb{Z}$ / $(\{\mathbb{I} \wedge \mathbb{B} \wedge V_0 = f\}\, S\, \{f < V_0\}, V_0 \in \mathbb{V}) \wedge (\mathbb{I} \wedge f \leq 0 \rightarrow \neg\mathbb{B})$ entonces la ejecución del ciclo siempre termina.*

Llamamos a la función f del teorema la **Función Variante**.

Teorema 4. $(\{\mathbb{P}_\mathbb{C}\}\, while\, B\ do\ S\ endwhile\, \{\mathbb{Q}_\mathbb{C}\}) \rightarrow (\mathbb{P}_\mathbb{C} \rightarrow wp(while\ B\ do\ S\ endwhile, \mathbb{Q}_\mathbb{C}))$.

2. Testing Estructural

Llamamos **Testing** al proceso de verificar si cierto programa satisface los requerimientos y si el comportamiento esperado es el comportamiento real. El objetivo de hacer testing es de encontrar defectos en el programa. Llamamos **Test Input** a una asignación concreta de valores de los parámetros de entrada del programa que se va a testear. Definimos **Test Case** a un programa que utiliza cierto test input para ejecutar en el programa a testear y que chequea si los datos de salida del programa son los esperados. Llamamos **Test Suite** a un conjunto de test cases. Todo programa debe ser una correcta implementación de la especificación, si es así entonces el programa debe aprobar todo test que pueda hacérsele. Los test inputs deben cumplir con la precondición del programa y el test case debe chequear que el output cumpla con la poscondición del programa. De esta forma, un test no puede demostrar que un programa sea correcto ya que no es exhaustivo, solo puede llegar a demostrar que un programa no es correcto hallando cierto test input que falle. La mayor dificultad de hallar un test adecuado es hallar un test suite que sea lo suficientemente grande como para abarcar el dominio y maximizar la probabilidad de hallar un error pero lo suficientemente pequeño como para minimizar el costo de testing.

Existen dos tipos de criterios para seleccionar test inputs: los **Tests de Caja Negra** se generan analizando la especificación sin considerar la implementación y los **Tests de Caja Blanca** se generan considerando la implementación. Los tests de caja negra se llaman **Tests Funcionales** y los de caja blanca se llaman **Tests Estructurales**.

El **Control Flow-Graph** (CFG) de un programa es una representación gráfica estática de un programa - es decir - no depende de las variables de entrada. Se utilizan para desarrollar test suites para distintos programas. Veamos un ejemplo:

Ejemplo

Especificación

proc *valorAbsoluto* (in $x : \mathbb{Z}$, out $result : \mathbb{Z}$){

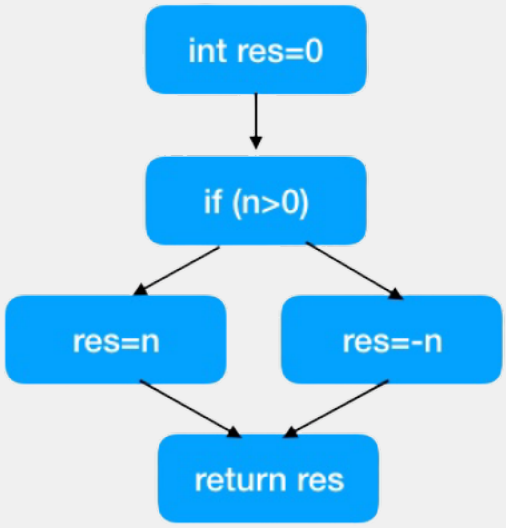
```
    Pre{True}
    Post{result = |x|}
```

}

Programa

```
int valorAbsoluto(int n){
    int res = 0;
    if(n > 0){
        res = n;
    } else{
        res = -n;
    }
    return res;
}
```

Control Flow-Graph



Un CFG contiene nodos donde se ejecuta alguna acción del código y arcos, que representan el flujo del código. Notemos como un condicional bifurca el flujo del código. Un test suite debe cubrir todos los arcos presentes en el CFG al menos una vez. Si se cubren todos lo arcos entonces se cubren todos los nodos, pero no al revés. Este es el **Criterio de Adecuación**. También, cada condición básica (cada fórmula no divisible) debe ser evaluada como verdadero y como falso al menos una vez.

3. Algoritmos Sobre Secuencias

3.1. Algoritmos de Búsqueda

Consideremos el problema de querer encontrar la posición de un elemento en una secuencia. Tomemos la función:

```
proc contiene(in s :Seq⟨ℤ⟩, in x : ℤ, out result : 𝔹){
    Pre{True}
    Post{result = True ↔ (∃j : ℤ) (0 ⩽ j < |s| ∧_L s[j] = x)}
}
```

Veamos cómo implementar esta función en un programa.

3.1.1. Búsqueda Lineal

Consideremos el siguiente programa en C++:

```
bool contiene(vector<int> &s, int x) {
    int j = 0;
    while (j < s.size() && s[j] != x) {
        j = j + 1;
    }
    return j < s.size();
}
```

Este programa cumple con la especificación con el siguiente invariante:

$$\mathbb{I} \equiv (0 \leqslant j \leqslant |s|) \wedge_L (\forall k : \mathbb{Z})\,(0 \leqslant k < j \rightarrow_L s\,[k] \neq x)$$

y con la siguiente función variante:

$$f_V\,(s, j) = |s| - j$$

Puede demostrarse que el programa termina y es correcto.

Notemos que para este programa el tiempo máximo ejecución sucede cuando el elemento buscado no está en la secuencia. En ese caso el ciclo entero debe correrse $|s|$ veces. Tomemos c_i, $i \in \{1, 2, 3, 4\}$ como el tiempo que tarda en ejecutarse cada línea del ciclo y tomemos $n = |s|$. Notemos que c_i son constantes ya que no dependen del estado del programa. Entonces, notemos que la primera línea sólo debe ejecutarse una vez, la segunda línea se ejecuta como máximo $n + 1$ veces, la tercer línea se ejecuta como máximo n veces y la última línea se ejecuta sólo una vez al final. Entonces, el tiempo máximo de ejecución es:

$$T\,(n) = c_1 + (n + 1)\,c_2 + nc_3 + c_4$$

El **Tiempo de Ejecución** de un programa es una función $T : \mathbb{N} \rightarrow \mathbb{R}_{\geqslant 0}$ que representa el tiempo máximo que tarda un programa en ejecutarse dada una entrada de tamaño n. Sean $f, g : \mathbb{N} \rightarrow \mathbb{R}$ notamos $f \in \mathcal{O}\,(g) \iff \exists\, c \in \mathbb{R},\, n_0 \in \mathbb{N} \,/\, f\,(n) \leqslant cg\,(n)\ \forall n \geqslant n_0$. Esta notación se llama **Notación O Grande**. Entonces, en el caso del tiempo de ejecución del programa tenemos que $T \in \mathcal{O}\,(n)$. Llamamos a este tiempo de ejecución **Lineal**. Análogamente se nombran otros tipos de tiempos de ejecución cuando son $\mathcal{O}\left(n^2\right)$, $\mathcal{O}\left(n^3\right)$, $\mathcal{O}\left(n^k\right)$, $\mathcal{O}\left(k^n\right)$ como cuadrático, cúbico, polinomial y exponencial respectivamente. Mientras menor sea el orden del tiempo de ejecución de un programa decimos que es más **Eficiente**.

3.1.2. Búsqueda Binaria

Consideremos ahora que la lista está ordenada:

```
proc contieneOrdenada(in s :Seq⟨ℤ⟩, in x : ℤ, out result : 𝔹){
    Pre{ordenado(s)}
    Post{result = True ↔ (∃j : ℤ) (0 ⩽ j < |s| ∧_L s[j] = x)}
}
pred ordenado(s :Seq⟨ℤ⟩){
    (∀j : ℤ) (0 ⩽ j < |s| − 1 →_L s[j] ⩽ s[j + 1])
}
```

Entonces, notemos ahora que el programa puede mejorarse. Antes recorríamos los elementos de la secuencia desde el primero hasta encontrar el elemento que buscamos. Si el elemento no estaba en la secuencia entonces había que recorrer toda la secuencia para saberlo. Ahora, podemos explotar propiedades del ordenamiento. Si en cierta posición k sabemos que $s\,[k] > x$ entonces sabemos que debemos buscar por los valores anteriores en la secuencia y si $s\,[k] < x$ debemos buscar por los valores siguientes en la secuencia. Haciendo esto múltiples veces podemos encontrar el elemento mucho más rápido o llegar a la conclusión de que el elemento no está en la secuencia. También, podemos chequear los casos triviales. Por ejemplo, si $|s| = 0$ entonces obviamente el elemento no está en la secuencia. Lo mismo si $x > s\,[|s| - 1]$ o $x < s\,[0]$. También, si $|s| = 1$ entonces sólo hay que chequear si $s\,[0] = x$.

El programa es:

```
bool contieneOrdenada(vector<int> &s, int x) {
    // casos triviales
    if (s.size() == 0) {
        return false;
    } else if (s.size() == 1) {
        return s[0] == x;
```

```
    } else if (x <= s[0]) {
        return s[0] == x;
    }else if (x >= s[s.size() - 1]) {
        return s[s.size() - 1] == x;
    } else {
        // casos no triviales
        int low = 0;
        int high = s.size() - 1;
        while (low + 1 < high) {
            int mid = (low + high) / 2;
            if (s[mid] <= x) {
              low = mid;
            } else {
              high = mid;
            }
        }
        return s[low] == x;
    }
}
```

Para este ciclo el invariante es:

$$\mathbb{I} \equiv (0 \leqslant low < high < |s|) \wedge_L (s\,[low] \leqslant x < s\,[high])$$

y la función variante es:

$$f_V\,(high, low) = high - low - 1$$

Puede demostrarse que el programa es correcto.

Veamos si este programa es más eficiente que el anterior. En cada iteración del ciclo tenemos que la cantidad de elementos por revisar de la secuencia es $\frac{1}{2}\,(n-1)$. Entonces, en el peor de los casos el ciclo corre t veces y queda sólo un elemento en la secuencia. Entonces:

$$\frac{n-1}{2^t} = 1$$

$$\implies t = \log_2\,(n-1)$$

$$\implies T \in \mathcal{O}\,(\log_2\,(n-1))$$

El tiempo de ejecución escala logarítmicamente, lo cual es mucho mejor que lineal. Habría que ver qué tan eficiente es ordenar una lista para poder comparar mejor.

3.2. Algoritmos de Ordenamiento

Consideremos el problema de tener que ordenar de menor a mayor una secuencia de enteros. La especificación es:

```
proc ordenar(inout s :Seq⟨ℤ⟩){
    Pre{s = S_0}
    Post{mismos (s, S_0) ∧ ordenado (s)}
}
pred mismos (s, t :Seq⟨ℤ⟩){
```

$(\forall e : \mathbb{Z})\,(\texttt{\#apariciones}\,(s, e) = \texttt{\#apariciones}\,(t, e))$

}

aux *#apariciones* (s :Seq⟨ℤ⟩, e : ℤ): ℤ =

$$\sum_{i=0}^{|s|-1} (\texttt{if}\ s[i] = e\ \texttt{then}\ 1\ \texttt{else}\ 0\ \texttt{fi});$$

pred *ordenado* (s :Seq⟨ℤ⟩){

$(\forall j : \mathbb{Z})\,(0 \leqslant j < |s| - 1 \rightarrow_L s[j] \leqslant s[j+1])$

}

Vamos a usar un programa que va a alterar la secuencia a través de intercambios de las posiciones de los elementos de la secuencia. Definamos entonces a estos intercambios mediante la siguiente especificación:

proc *swap* (inout s :Seq⟨ℤ⟩, in $i, j : \mathbb{Z}$){

Pre{$(0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|) \wedge (s = S_0)$}
Post{$(s[i] = S_0[j]) \wedge (s[j] = S_0[i]) \wedge (\forall k : \mathbb{Z})\,((0 \leqslant k < |s|) \wedge (k \neq i) \wedge (k \neq j) \rightarrow_L s[k] = S_0[k])$}

}

Existen muchos algoritmos de ordenamiento. Veamos algunos.

3.2.1. Ordenamiento por Selección

Este algoritmo consiste en buscar al mínimo elemento de la secuencia y intercambiar su posición con el primer elemento. Luego, repetir usando la secuencia desde la segundo posición y así hasta que la secuencia esté ordenada. Especifiquemos la función que busca el menor elemento entre las posiciones d (inclusive) y h (exclusive):

proc *findMinPosition* (in s :Seq⟨ℤ⟩, in $d, h : \mathbb{Z}$, out $min : \mathbb{Z}$){

Pre{$0 \leqslant d < h \leqslant |s|$}
Post{$(d \leqslant min < h) \wedge_L (\forall i : \mathbb{Z})\,(d \leqslant i < h \rightarrow_L s[min] \leqslant s[i])$}

}

El programa de ordenamiento entonces es:

```
int findMinPosition(vector<int> &s, int d, int h) {
    int min = d;
    for (int i = d + 1; i < h; i++) {
        if (s[i] <= s[min]) {
           min = i;
        }
    }
    return min;
}

void swap(vector<int> &s, int i, int j) {
    int aux = s[i];
    s[i] = s[j];
    s[j] = aux;
}
void selectionSort(vector<int> &s){
```

```
    if (s.size() != 0) {
        for (int i = 0; i < s.size() - 1; i++) {
            int minPos = findMinPosition(s, i, s.size());
            swap(s, i, minPos);
        }
    }
}
```

Puede demostrarse la correctitud de este programa usando el invariante:

$$\mathbb{I} \equiv \text{mismos}\,(s, S_0) \wedge$$
$$((0 \leqslant i \leqslant |s|) \wedge_L \text{ordenado}\,(\text{subseq}\,(s, 0, i)) \wedge (\forall j, k : \mathbb{Z})\,(((0 \leqslant j < i) \wedge (i \leqslant k < |s|)) \rightarrow_L s\,[j] \leqslant s\,[k]))$$

y la función variante:

$$f_V\,(s, i) = |s| - i$$

Se pueden deducir los siguientes tiempos de ejecución:

$$T_{findMinPosition} \in \mathcal{O}\,(n)$$

$$\implies T_{selectionSort} \in \mathcal{O}\left(n^2\right)$$

3.2.2. Ordenamiento por Inserción

Este algoritmo de ordenamiento consiste en tomar un elemento en cierta posición y, asumiendo que todos los elementos anteriores están ordenados entre sí, insertar este elemento en el lugar que le corresponda en el orden de los elementos anteriores. Si se empieza por el primer elemento (que trivialemente está ordenado con respecto a la secuencia que sólo contiene el primer elemento - es decir - sí mismo) y se recorre la secuencia en el orden que está entonces se pueden ordenar todos los elementos. El programa para este algoritmo es:

```
void insert(vector<int> &s, int i) {
    for (int j = i; j > 0 && s[j] < s[j - 1]; j--) {
        swap(s, j, j - 1);
    }
}

void insertionSort(vector<int> &s) {
    for (int i = 0; i < s.size(); i++) {
        insert(s, i);
    }
}
```

Para este algoritmo el tiempo de ejecución es $T_{insert} \in \mathcal{O}\,(n)$ y $T_{insertSort} \in \mathcal{O}\left(n^2\right)$. Por lo tanto, no es más eficiente que el algoritmo de ordenamiento por selección. Existen más algoritmos de ordenamiento que son más eficientes, como *MergeSort*, *Counting Sort* y *Radix Sort*. Estos algoritmos tienen tiempo de ejecución de $\mathcal{O}\,(n \log\,(n))$, $\mathcal{O}\,(n)$ y $\mathcal{O}\,(1)$ respectivamente. Luego hay otros algoritmos como *Bubble Sort* y *Quicksort* que son comunes pero son $\mathcal{O}\left(n^2\right)$. Se pueden ver animaciones de estos algoritmos de ordenamiento aquí.

3.2.3. Dutch National Flag Problem

Dada una secuencia de colores rojo, blanco y azul queremos ordenarla para que estén en el orden de la bandera holandesa (rojo, luego blanco y luego azul). Para simplificar, tomemos el color rojo como el número 0, el blanco como 1 y el azul como 2. Esto nos permite trabajar con una secuencia de enteros donde tenemos que ordenar sus elementos de menor a mayor. Este problema se parece mucho a los de ordenamiento anteriores, sin embargo notemos que en este caso sabemos que sólo pueden haber tres distintos elementos en la secuencia: 0, 1 y 2. Esto nos va a ayudar a reducir la complejidad del programa.

La especificación del problema es:

```
proc dutchNationalFlag(inout s :Seq⟨ℤ⟩){
    Pre{s = S_0 ∧ (∀e : ℤ) (e ∈ s ↔ (e = 0 ∨ e = 1 ∨ e = 2))}
    Post{mismos (s, S_0) ∧ ordenado (s)}
}
```

Para resolver este problema podemos recorrer la secuencia y contar las veces que aparece cada color, almacenando estas cantidades en una secuencia de tres números. Luego, cambiamos la secuencia por una que tenga el color rojo la cantidad de veces que haya sido encontrada en la secuencia, luego lo mismo para el blanco y luego lo mismo para el azul. La implementación es la siguiente:

```
#define RED 0
#define WHITE 1
#define BLUE 2
using flag = vector<int>;
vector<int> fillColorCount(flag &s) {
    vector<int> colorCount(3, 0);
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == RED) {
            colorCount[RED]++;
        } else if (s[i] == WHITE) {
            colorCount[WHITE]++;
        } else if (s[i] == BLUE) {
            colorCount[BLUE]++;
        }
    }
    return colorCount;
}

void populate(flag &s, vector<int> &colorCount) {
    for (int i = 0; i < s.size(); i++) {
        if (colorCount[RED] > 0) {
            s[i] = RED;
            colorCount[RED]--;
        } else if (colorCount[WHITE] > 0) {
            s[i] = WHITE;
            colorCount[WHITE]--;
        } else {
            s[i] = BLUE;
            colorCount[BLUE]--;
        }
```

```
    }
}
void dutchNationalFlag(flag &s) {
    vector<int> colorCount = fillColorCount(s);
    populate(s, colorCount);
}
```

Notemos que este programa recorre la secuencia dos veces, así que el tiempo de ejecución es $T(n) \in \mathcal{O}(n)$. Esto se debe a que sabemos qué elementos pueden estar en la secuencia. Proponiendo un invariante puede demostrarse que este programa es correcto.

3.3. Algoritmos Sobre Secuencias Ordenadas

3.3.1. Apareo

Consideremos que tenemos dos secuencias ordenadas y queremos unirlas en una única secuencia ordenada. La especificación del problema es:

```
proc merge(in a, b :Seq⟨ℤ⟩, out c :Seq⟨ℤ⟩){
    Pre{ordenado(a) ∧ ordenado(b)}
    Post{ordenado(c) ∧ mismos(c, a++b)}
}
```

La implementación de la especificación es:

```
vector<int> merge(vector<int> &a, vector<int> &b) {
    vector<int> c(a.size() + b.size());
    int i = 0; // Para recorrer a
    int j = 0; // Para recorrer b
    for(int k = 0; k < c.size(); k++) {
        if(j >= b.size() || (i < a.size() && a[i] < b[j])) {
            c[k] = a[i];
            i++;
        } else {
            c[k] = b[j];
            j++;
        }
    }
    return c;
}
```

Lo que el programa hace es ir recorriendo las secuencias a y b y compara cada elemento. Luego, inserta el menor elemento en la secuencia c y pasa al siguiente elemento de la secuencia de donde insertó el elemento. Hace esto hasta terminar alguna secuencia y luego inserta el resto de los elementos de la secuencia que sobra. Al final, la secuencia c tiene todos los elementos de a y b ordenados. El tiempo de ejecución de este algoritmo es $\mathcal{O}(n)$.

4. Algoritmos Sobre Strings

Llamamos un **String** a una secuencia de caracteres ($\text{Seq}\,\langle\text{Char}\rangle$). Lo que vamos a querer hacer es buscar cierto patrón de caracteres dentro de cierta string. Especifiquemos este problema:

```
proc contiene(in text, pattern :Seq⟨Char⟩, out result : 𝔹){
    Pre{True}
    Post{result = True ↔ (∃i : ℤ) ((0 ⩽ i ⩽ |text| − |pattern|) ∧_L subseq(text, i, i + |pattern|) = pattern)}
}
```

Lo que podemos hacer es buscar donde está en el texto el primer caracter del patrón y de ahí comparar el resto de los caracteres para ver si es o no. Especifiquemos la función auxiliar:

```
proc matches(in text, pattern :Seq⟨Char⟩, in i : ℤ, out result : 𝔹){
    Pre{(0 ⩽ i < |text| − |pattern|) ∧ (|pattern| ⩽ |text|)}
    Post{resutl = True ↔ subseq(text, i, i + |pattern|) = pattern}
}
```

La implementación es:

```
bool matches(string &t, string &p, int i){
    int k = 0;
    while (k < p.size() && t[i + k] == p[k]) {
        k++;
    }
    return k == p.size();
}
```

Esta función es $\mathcal{O}(p)$, donde $p = |pattern|$. Implementemos entonces la función contiene:

```
bool contiene(string &t, string &p){
    bool res = false;
    if (t.size() != 0) {
        for (int i = 0; i <= t.size() - p.size(); i++) {
            if (matches(t, p, i)) {
              res = true;
            }
        }
    }
    return res;
}
```

El tiempo de ejecución es de $\mathcal{O}(tp)$, donde $t = |text|$.

Veamos ahora el **Algoritmo de Knuth, Morris y Pratt**. Este es un algoritmo que resuelve la misma especificación en menor tiempo. Consiste en usar dos índices, l y r, que cumplen:

$$\begin{cases} 0 \leqslant l \leqslant r \leqslant t \\ \text{subseq}\,(text, l, r) = \text{subseq}\,(pattern, 0, r - l) \\ pattern \text{ no está en subseq}\,(text, 0, r) \end{cases}$$

Consideremos distintos casos:

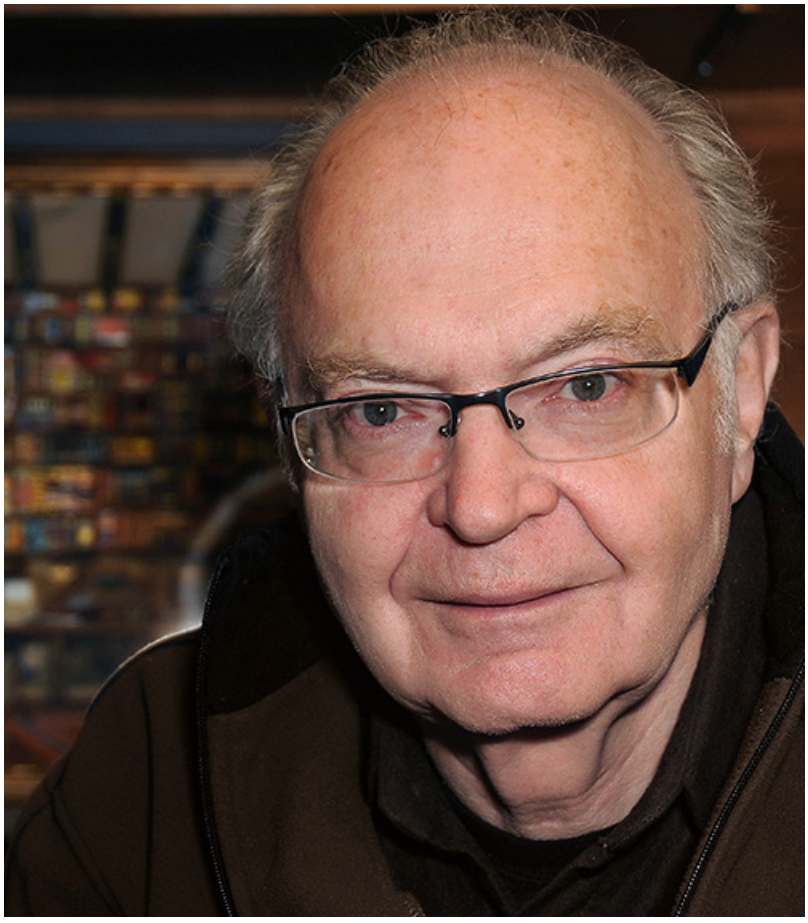
- $r - l = p$: Hallaste el string.
- $r - l < p$:
  - $text[r] = pattern[r - l]$: Hallamos una nueva coincidencia y entonces hay que avanzar r.
  - $(text[r] \neq pattern[r - l]) \wedge (l = r)$: No tenemos un prefijo de *pattern*, así que hay que avanzar r y l.
  - $(text[r] \neq pattern[r - l]) \wedge (l < r)$: Avanzamos l lo máximo posible. Llamamos **Bifijo** a una cadena de caracteres que es prefijo y sufijo de cierto string. Es decir, un bifijo de un string es un string menor que está al principio del string y al final del string (ver ejemplos en las diapositivas de la clase 11). Entonces, si $\pi(r - l)$ es el largo del máximo bifijo del string entre l y r, movemos l por $r - \pi(r - l)$ lugares. En esta nueva posición sabemos que el patrón empieza igual. El tiempo de ejecución de $\pi(r - l) \in \mathcal{O}(p)$.

La implementación de este algoritmo es:

```
vector<int> calcularPi(string &p) {
    vector<int> pi(p.size());
    int i = 0;
    for (int  j = 1; j < p.size(); j++) {
        while (i > 0 && p[i] != p[j]) {
            i = pi[i - 1];
        }
        if (p[i] == p[j]){
            i++;
        }
        pi[j] = i;
    }
    return pi;
}
bool contieneKMP(string &t, string &p) {
    vector<int> pi = calcularPi(p);
    int l = 0;
    int r = 0;
    while (r < t.size() && r - l < p.size()) {
        if (t[r] == p[r - l]) {
            r++;
        } else if ( l == r) {
            r++;
            l++;
        } else {
            l = r - pi[r - l];
        }
    }
    return r - l == p.size();
}
```

El tiempo de ejecución de este algoritmo es de $\mathcal{O}(t + p)$.

Parte II
Práctica



Donald Ervin Knuth (1938)

5. Guía 2: Introducción al Lenguaje de Especificación

5.1. Ejercicio 4*

5.1.1. (a)

```
pred estaAcotada(s:Seq⟨ℤ⟩){
```

$$(\forall x : \mathbb{Z})\,(x \in s \rightarrow 1 \leqslant x \leqslant 100)$$

```
}
```

5.1.2. (c)

```
/* Un prefijo es una subsecuencia que esta al principio */
pred esPrefijo(l, s:Seq⟨ℤ⟩){
```

$$|l| \leqslant |s| \wedge_L (\forall i : \mathbb{Z})\,(0 \leqslant i < |l| \rightarrow_L s\,[i] = l\,[i])$$

```
}
/* Otra solucion */
pred esPrefijo_2(l, s:Seq⟨ℤ⟩){
```

$$(\exists t{:}\texttt{Seq}\langle\mathbb{Z}\rangle)(s = l + +t)$$

```
}
/* Otra solucion */
pred esPrefijo_2(l, s:Seq⟨ℤ⟩){
```

$$(\exists i : \mathbb{Z})(0 \leqslant j \leqslant |s| \wedge_L l = \texttt{subseq}(s, 0, j))$$

```
}
```

5.1.3. (d)

```
pred estaOrdenada(s:Seq⟨ℤ⟩){
```

$$(\forall i : \mathbb{Z})\,(0 \leqslant i < |s| - 1 \rightarrow_L s\,[i] \leqslant s\,[i+1])$$

```
}
```

5.1.4. (e)

```
pred todosPrimos(s:Seq⟨ℤ⟩){
```

$$(\forall x : \mathbb{Z})(x \in s \rightarrow \textit{esPrimo}\,(x))$$

```
}
```

5.1.5. (f)

```
pred primosEnPosicionesPares(s:Seq⟨ℤ⟩){
```

$$(\forall i : \mathbb{Z})(0 \leqslant i < |s| \wedge_L \textit{esPrimo}\,(s\,[i]) \rightarrow i \mod 2 = 0)$$

```
}
```

5.1.6. (h)

```
pred hayUnoParQueDivideAlResto(s:Seq⟨ℤ⟩){
```

$$(\exists x : \mathbb{Z})\,(x \in s \wedge x \mod 2 = 0 \wedge x \neq 0 \wedge_L (\forall y : \mathbb{Z})\,(y \in s \rightarrow y \mod x = 0))$$

```
}
```

5.1.7. (n)

pred *esPermutacionOrdenada* (s, t:Seq$\langle\mathbb{Z}\rangle$){

$esPermutacion\,(s,t) \wedge estaOrdenada\,(s)$

}

5.2. Ejercicio 6

5.2.1. (a)*

pred *ej_seis_a* (s:Seq$\langle\mathbb{Z}\rangle$){

$(\forall i : \mathbb{Z})\,(0 \leqslant i < |s| \wedge_L P\,(s\,[i]) \rightarrow_L Q\,(s\,[i]))$

}

/* Otra solucion */

pred *ej_seis_a_2* (s:Seq$\langle\mathbb{Z}\rangle$){

$(\forall i : \mathbb{Z})\,(i \in s \wedge P\,(i) \rightarrow Q\,(i))$

}

5.2.2. (b)

pred *ej_seis_b* (s:Seq$\langle\mathbb{Z}\rangle$){

$(\forall x : \mathbb{Z})(x \in s \wedge P\,(x) \rightarrow \neg Q\,(x))$

}

5.2.3. (c)

pred *ej_seis_c* (s:Seq$\langle\mathbb{Z}\rangle$){

$(\forall i : \mathbb{Z})\,(0 \leqslant i < |s| \wedge_L (i \mod 2 = 0 \wedge P\,(s\,[i])) \rightarrow \neg Q\,(s\,[i]))$

}

5.2.4. (d)*

pred *ej_seis_d* (s:Seq$\langle\mathbb{Z}\rangle$){

$(\forall i : \mathbb{Z})\,(0 \leqslant i < |s| \wedge_L (P\,(s\,[i]) \wedge Q\,(i)) \rightarrow_L s\,[i] \mod 2 = 0)$

}

5.2.5. (e)*

pred *ej_seis_e* (s:Seq$\langle\mathbb{Z}\rangle$){

$(\exists x : \mathbb{Z})\,(x \in s \wedge \neg P\,(x)) \rightarrow (\forall y : \mathbb{Z})\,(y \in s \rightarrow \neg Q\,(y))$

}

5.2.6. (f)

pred *ej_seis_f* (s:Seq$\langle\mathbb{Z}\rangle$){

$((\exists x : \mathbb{Z})\,(x : s \wedge \neg P\,(x)) \rightarrow (\forall y : \mathbb{Z})\,(y \in s \wedge \neg Q\,(y))) \wedge$
$((\forall x : \mathbb{Z})\,(x \in s \wedge P\,(x)) \rightarrow (\exists y, z : \mathbb{Z})\,(y, z \in s \wedge y \neq z \wedge Q\,(y) \wedge Q\,(z)))$

}

6. Guía 3: Especificación de Problemas

6.1. Ejercicio 9*

6.1.1. (e)

proc *duplicPosImp* (in $s : \langle \mathbb{R} \rangle$, out $l : \langle \mathbb{R} \rangle$){

Pre{True}
Post{$|s| = |l| \wedge_L$
$(\forall i : \mathbb{Z})\,((0 \leqslant i < |s| \wedge i \mod 2 = 1 \rightarrow_L l[i] = 2s[i]) \wedge (0 \leqslant i < |s| \wedge i \mod 2 = 0 \rightarrow_L l[i] = s[i]))$}

}

6.2. Ejercicio 12*

proc *enBinario* (in $n : \mathbb{Z}$, out $s : \langle \mathbb{Z} \rangle$){

Pre{$n > 0$}
Post{

$$|s| > 0 \wedge_L (\forall x : \mathbb{Z})\,(x \in s \rightarrow (x = 0 \vee x = 1)) \wedge_L \sum_{i=0}^{|s|-1} s[i]\, 2^{|b|-i-1} = n$$

}

}

6.3. Ejercicio 14*

6.3.1. (d)

proc *factores* (in $n : \langle \mathbb{Z} \rangle$, out res :Seq$\langle \mathbb{Z} \times \mathbb{Z} \rangle$){

Pre{$n > 1$}
Post{

$$(\forall i : \mathbb{Z})(0 \leqslant i < |s| \rightarrow_L (\textit{esTuplaValida}(s[i]) \wedge (i \neq |s| - 1 \rightarrow_L s[i]_0 < s[i+1]_0)))$$
$$\wedge \prod_{j=0}^{|s|-1} s[j]_0^{s[j]_1} = n$$

}

}
pred *esTuplaValida* ($x : \mathbb{Z} \times \mathbb{Z}$){

$\textit{esPrimo}(x_0) \wedge x_1 > 0$

}
pred *esPrimo* ($n : \mathbb{Z}$){

$$n > 1 \wedge \left(\forall n' : \mathbb{Z}\right)\left(1 < n' < n \rightarrow_L n \mod n' \neq 0\right)$$

}

6.3.2. (e)

proc *maxDif* (in s :Seq$\langle \mathbb{Z} \rangle$, out $r : \mathbb{Z}$){

Pre{$|s| > 1$}
Post{$(\exists x_1, x_2 : \mathbb{Z})\,((x_1 \in s) \wedge (x_2 \in s) \wedge (r = x_1 - x_2) \wedge \neg\,(\exists y_1, y_2 : \mathbb{Z})\,((y_1 \in s) \wedge (y_2 \in s) \wedge (r < y_1 - y_2)))$}

}

7. Guía 4: Precondición Más Débil en SmallLang

7.1. Ejercicio 2

7.1.1. (a)

$$\begin{aligned}\text{wp}\left(a \overset{\text{def}}{:=} a+1, a \geqslant 0\right) &\equiv \text{def}\left(a+1\right) \wedge_L \left(a+1 \geqslant 0\right) \\ &\equiv a \geqslant -1\end{aligned}$$

7.1.2. (b)

$$\begin{aligned}\text{wp}\left(a \overset{\text{def}}{:=} a/b, a \geqslant 0\right) &\equiv \text{def}\left(\frac{a}{b}\right) \wedge_L \left(\frac{a}{b} \geqslant 0\right) \\ &\equiv (a \geqslant 0 \wedge b > 0) \vee (a \leqslant 0 \wedge b < 0)\end{aligned}$$

7.1.3. (c)

$$\begin{aligned}\text{wp}\left(a \overset{\text{def}}{:=} A\left[i\right], a \geqslant 0\right) &\equiv \text{def}\left(A\left[i\right]\right) \wedge_L \left(A\left[i\right] \geqslant 0\right) \\ &\equiv A\left[i\right] \geqslant 0\end{aligned}$$

7.1.4. (d)

$$\begin{aligned}\text{wp}\left(a \overset{\text{def}}{:=} b * b, a \geqslant 0\right) &\equiv \text{def}\left(b * b\right) \wedge_L \left(b * b \geqslant 0\right) \\ &\equiv b^2 \geqslant 0 \\ &\equiv \text{True}\end{aligned}$$

7.1.5. (e)

$$\begin{aligned}\text{wp}\left(b \overset{\text{def}}{:=} b+1, a \geqslant 0\right) &\equiv \text{def}\left(b+1\right) \wedge_L \left(a \geqslant 0\right) \\ &\equiv a \geqslant 0\end{aligned}$$

7.2. Ejercicio 4

$$\mathbb{Q} \equiv (\forall j : \mathbb{Z})\left(0 \leqslant j < |A| \rightarrow_L A\left[j\right] \geqslant 0\right)$$

7.2.1. (a)

$$\begin{aligned}\text{wp}\left(A\left[i\right] \overset{\text{def}}{:=} 0, \mathbb{Q}\right) \equiv &\left(\left(\text{def}\left(i\right) \wedge_L 0 \leqslant i < |A|\right) \wedge \text{def}\left(0\right)\right) \wedge_L \\ &\left(\left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \wedge j \neq i \rightarrow_L A\left[j\right] \geqslant 0\right) \wedge \left(0 \geqslant 0\right)\right) \\ \equiv &\left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \wedge j \neq i \rightarrow_L A\left[j\right] \geqslant 0\right)\end{aligned}$$

7.2.2. (d)

$$\begin{aligned}\text{wp}\left(A\left[i\right] \overset{\text{def}}{:=} 2 * A\left[i\right], \mathbb{Q}\right) &\equiv \left(\left(\text{def}\left(i\right) \wedge_L 0 \leqslant i < |A|\right) \wedge \text{def}\left(2 * A\left[i\right]\right)\right) \wedge_L \\ &\quad \left(\left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \wedge j \neq i \rightarrow_L A\left[j\right] \geqslant 0\right) \wedge \left(2A\left[i\right] \geqslant 0\right)\right) \\ &\equiv \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \wedge j \neq i \rightarrow_L A\left[j\right] \geqslant 0\right) \wedge \left(A\left[i\right] \geqslant 0\right) \\ &\equiv \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \rightarrow_L A\left[j\right] \geqslant 0\right)\end{aligned}$$

7.2.3. (e)

$$\begin{aligned}\text{wp}\left(A\left[i\right] \overset{\text{def}}{:=} A\left[i-1\right], \mathbb{Q}\right) &\equiv \left(\left(\text{def}\left(i\right) \wedge_L 0 \leqslant i < |A|\right) \wedge \left(\text{def}\left(i-1\right) \wedge_L 0 \leqslant i-1 < |A|\right)\right) \wedge_L \\ &\quad \left(\left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \wedge j \neq i \rightarrow_L A\left[j\right] \geqslant 0\right) \wedge \left(A\left[i-1\right] \geqslant 0\right)\right) \\ &\equiv \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j < |A| \wedge j \neq i \rightarrow_L A\left[j\right] \geqslant 0\right)\end{aligned}$$

7.3. Ejercicio 6

7.3.1. (a)

Programa

```
a := a + 2;
```

Demostración

$$\begin{aligned}\text{wp}\left(S, \mathbb{Q}\right) &\equiv \text{def}\left(a+2\right) \wedge_L \left(a+2 = a+2\right) \\ &\equiv \text{True}\end{aligned}$$

$$\mathbb{P} \rightarrow \text{wp}\left(S, \mathbb{Q}\right) \checkmark$$

7.3.2. (b)

Programa

```
b := a + 3;
```

Demostración

$$\begin{aligned}\text{wp}\left(S, \mathbb{Q}\right) &\equiv \text{def}\left(a+3\right) \wedge_L \left(a+3 = a+3\right) \\ &\equiv \text{True}\end{aligned}$$

$$\mathbb{P} \rightarrow \text{wp}\left(S, \mathbb{Q}\right) \checkmark$$

7.3.3. (c)

Programa

```
c := a + b;
```

Demostración

$$\begin{aligned}\text{wp}\left(S, \mathbb{Q}\right) &\equiv \text{def}\left(a+b\right) \wedge_L \left(a+b = a+b\right) \\ &\equiv \text{True} \checkmark\end{aligned}$$

7.3.4. (d)

Programa

```
result := 2 * a[i];
```

Demostración

$$\begin{aligned}\text{wp}\left(S, \mathbb{Q}\right) &\equiv \left(\text{def}\left(i\right) \wedge_L 0 \leqslant i < |a|\right) \wedge_L \left(2 * a\left[i\right] = 2 * a\left[i\right]\right) \\ &\equiv \text{True}\end{aligned}$$

7.4. Ejercicio 8

7.4.1. (a)

Programa

```
a := a + s[i];
```

Demostración

$$\begin{aligned}\text{wp}\left(S, a = \sum_{j=0}^{i} s\left[j\right]\right) \equiv &\left(\left(\text{def}\left(i\right) \wedge_L 0 \leqslant i < |s|\right) \wedge_L \left(\text{def}\left(a\right) \wedge_L a = \sum_{j=0}^{i-1} s\left[j\right]\right)\right) \wedge_L \\ &\left(\sum_{j=0}^{i-1} s\left[j\right] + s\left[i\right] = \sum_{j=0}^{i} s\left[j\right]\right) \\ \equiv &\ 0 \leqslant i < |s| \wedge_L a = \sum_{j=0}^{i-1} s\left[j\right] \ \checkmark\end{aligned}$$

7.4.2. (c)

Programa

```
if (s[i] >= 0) then
    res := true;
else
    res := false;
endif
```

Demostración

$$\begin{aligned}\text{wp}\left(S, \mathbb{Q}\right) \equiv &\left(\text{def}\left(s\right) \wedge_L \left(\text{def}\left(i\right) \wedge_L 0 \leqslant i < s\left[i\right]\right)\right) \wedge_L \\ &\left(\left(s\left[i\right] \geqslant 0 \wedge \text{wp}\left(res \stackrel{\text{def}}{:=} \text{True}, \mathbb{Q}\right)\right) \vee \left(\neg\left(s\left[i\right] \geqslant 0\right) \wedge \text{wp}\left(res \stackrel{\text{def}}{:=} \text{False}, \mathbb{Q}\right)\right)\right) \\ \equiv &\left(0 \leqslant i < s\left[i\right]\right) \wedge_L \left(s\left[i\right] \geqslant 0 \wedge \text{True} = \text{True} \leftrightarrow \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j \leqslant i \rightarrow_L s\left[j\right] \geqslant 0\right)\right) \\ &\vee \left(0 \leqslant i < s\left[i\right]\right) \wedge_L \left(s\left[i\right] < 0 \wedge \text{False} = \text{True} \leftrightarrow \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j \leqslant i \rightarrow_L s\left[j\right] \geqslant 0\right)\right) \\ \equiv &\left(0 \leqslant i < s\left[i\right]\right) \wedge_L \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j \leqslant i \rightarrow_L s\left[j\right] \geqslant 0\right) \\ &\vee \left(0 \leqslant i < s\left[i\right]\right) \wedge_L \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j \leqslant i \rightarrow_L s\left[j\right] > 0\right) \\ \equiv &\left(0 \leqslant i < s\left[i\right]\right) \wedge_L \left(\forall j : \mathbb{Z}\right)\left(0 \leqslant j \leqslant i \rightarrow_L s\left[j\right] > 0\right) \ \checkmark\end{aligned}$$

7.4.3. (d)

Programa

```
if (s[i] != 0) then
    a := a + 1;
else
    skip;
endif
```

Demostración

$$\begin{aligned}\mathrm{wp}\left(S, \mathbb{Q}\right) &= \left(\mathrm{def}\left(s\right) \wedge_L \left(\mathrm{def}\left(i\right) \wedge_L 0 \leqslant i < |s|\right)\right) \wedge_L \\ &\quad \left(\left(s\left[i\right] \neq 0 \wedge \mathrm{wp}\left(S_1, \mathbb{Q}\right)\right) \vee \left(\neg\left(s\left[i\right] \neq 0\right) \wedge \mathrm{wp}\left(S_2, \mathbb{Q}\right)\right)\right) \\ &\equiv \left(\left(0 \leqslant i < |s|\right) \wedge_L \left(s\left[i\right] \neq 0 \wedge a + 1 = \sum_{j=0}^{i} \tilde{S}\right)\right) \\ &\quad \vee \left(\left(0 \leqslant i < |s|\right) \wedge_L \left(s\left[i\right] = 0 \wedge a = \sum_{j=0}^{i} \tilde{S}\right)\right) \\ &\equiv \left(0 \leqslant i < |s|\right) \wedge_L \left(a = \sum_{j=0}^{i-1} \left(\text{if } s\left[j\right] \neq 0 \text{ then } 1 \text{ else } 0 \text{ fi}\right)\right) \checkmark\end{aligned}$$

8. Guía 5: Demostración de Corrección de Ciclos en SmallLang

8.1. Ejercicio 2

$$\begin{cases} \mathbb{P}_{\mathbb{C}} \equiv n \geqslant 0 \\ \mathbb{Q}_{\mathbb{C}} \equiv result = \sum_{j=0}^{n-1} \left(\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}\right) \\ \mathbb{B} \equiv i < n \\ \mathbb{I} \equiv \left(0 \leqslant i \leqslant n+1\right) \wedge \left(i \mod 2 = 0\right) \wedge \left(result = \sum_{j=0}^{i-1} \left(\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}\right)\right) \end{cases}$$

$\mathbb{P}_{\mathbb{C}} \to \mathbb{I}$

$$n \geqslant 0 \to \left(\left(0 \leqslant i \leqslant n+1\right) \wedge \left(i \mod 2 = 0\right) \wedge \left(result = \sum_{j=0}^{i-1} \left(\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}\right)\right)\right)$$

$n \geqslant 0 \to 0 \leqslant n+1$ ✓

$\{\mathbb{I} \wedge \mathbb{B}\}\, S\, \{\mathbb{I}\}$

$$\begin{aligned}\mathbb{I} \wedge \mathbb{B} &\equiv \left(0 \leqslant i \leqslant n+1\right) \wedge \left(i \mod 2 = 0\right) \wedge \left(result = \sum_{j=0}^{i-1} \left(\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}\right)\right) \wedge \left(i < n\right) \\ &\equiv \left(0 \leqslant i < n\right) \wedge \left(i \mod 2 = 0\right) \wedge \left(result = \sum_{j=0}^{i-1} \left(\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}\right)\right)\end{aligned}$$

El ciclo lo único que hace es sumarle i a $result$ y sumarle 2 a i. Llamemos C al ciclo, C_1 a la acción de sumarle i a $result$ y C_2 a la acción de sumarle 2 a i. Entonces:

$$
\begin{aligned}
\text{wp}\,(C, \mathbb{I}) &\equiv \text{wp}\,(C_1, \text{wp}\,(C_2, \mathbb{I})) \\
&\equiv \text{def}\,(i) \wedge_L \left(\text{wp}\,(C_2, \mathbb{I})^{result}_{result+i}\right) \\
&\equiv \text{wp}\,(C_2, \mathbb{I})^{result}_{result+i} \\
&\equiv \mathbb{I}^{result,\, i}_{result+i,\, i+2} \\
&\equiv (0 \leqslant i+2 \leqslant n+1) \wedge ((i+2) \mod 2 = 0) \wedge \left(result + i = \sum_{j=0}^{i-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \\
&\equiv (0 \leqslant i < n) \wedge (i \mod 2 = 0) \wedge \left(result = \sum_{j=0}^{i-2} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \\
i \mod 2 = 0 \rightarrow \quad &\equiv (0 \leqslant i < n) \wedge (i \mod 2 = 0) \wedge \left(result = \sum_{j=0}^{i-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \\
&\equiv \mathbb{I} \wedge \mathbb{B} \checkmark
\end{aligned}
$$

$\mathbb{I} \wedge \neg\mathbb{B} \rightarrow \mathbb{Q}_C$

$$
\begin{aligned}
\mathbb{I} \wedge \neg\mathbb{B} &\equiv (0 \leqslant i \leqslant n+1) \wedge (i \mod 2 = 0) \wedge \left(result = \sum_{j=0}^{i-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \wedge \neg\,(i < n) \\
&\equiv (0 \leqslant i \leqslant n+1) \wedge (i \mod 2 = 0) \wedge \left(result = \sum_{j=0}^{i-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \wedge (i \geqslant n) \\
&\equiv (n \leqslant i \leqslant n+1) \wedge (i \mod 2 = 0) \wedge \left(result = \sum_{j=0}^{i-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \\
&\equiv \text{if } n \mod 2 = 0 \text{ then } result = \sum_{j=0}^{n-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}) \text{ else} \\
&\quad result = \sum_{j=0}^{n} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}) \text{ fi} \\
&\equiv result = \sum_{j=0}^{n-1} (\text{if } j \mod 2 = 0 \text{ then } j \text{ else } 0 \text{ fi}) \\
&\equiv \mathbb{Q}_C \checkmark
\end{aligned}
$$

8.2. Ejercicio 3

8.2.1. (a)

Programa

```
i := 0;
result := 1;
while(i < n) do
    result := result * m;
    i := i + 1;
```

```
endwhile
```

Demostración

$$\begin{cases} \mathbb{P}_{\mathbb{C}} \equiv (n \geqslant 0) \land \neg (n = 0 \land m = 0) \land (i = 0) \land (result = 1) \\ \mathbb{Q}_{\mathbb{C}} \equiv result = m^n \\ \mathbb{B} \equiv i < n \\ \mathbb{I} \equiv (0 \leqslant i \leqslant n) \land \left(result = m^i\right) \end{cases}$$

$\mathbb{P}_{\mathbb{C}} \to \mathbb{I}$

$$\begin{aligned} (n \geqslant 0) \land \neg (n = 0 \land m = 0) \land (i = 0) \land (result = 1) &\to (0 \leqslant i \leqslant n) \land \left(result = m^i\right) \\ &\to (n \geqslant 0) \land (result = 1) \ \checkmark \end{aligned}$$

Veamos la tabla de estados:

i	$result$
0	1
1	m
2	m^2
n	m^n

Notemos que el invariate vale siempre menos en la primera fila cuando $m = 0$. De hecho acá creo que hay un error con el invariante que nos dieron porque si $m = 0$ entonces como el invariante admite la posibilidad de $i = 0$ al hacer m^i puede indefinirse. Es decir, así como lo dieron el invariante puede estar indefinido. Podría cambiar el programa y poner un condicional que si $m = 0$ entonces defina $i = 1$ pero eso complica un montón el programa sólo para ajustar a un invariante que es incómodo.

$\{\mathbb{I} \land \mathbb{B}\}\, S\, \{\mathbb{I}\}$

$$\begin{aligned} \mathbb{I} \land \mathbb{B} &\equiv (0 \leqslant i \leqslant n) \land \left(result = m^i\right) \land (i < n) \\ &\equiv (0 \leqslant i < n) \land \left(result = m^i\right) \end{aligned}$$

$$\begin{aligned} \text{wp}\,(S, \mathbb{I}) &\equiv \text{wp}\,(S_1, \text{wp}\,(S_2, \mathbb{I})) \\ &\equiv \text{def}\,(m) \land_L \text{wp}\,(S_2, \mathbb{I})^{result}_{result*m} \\ &\equiv \text{wp}\,(S_2, \mathbb{I})^{result}_{result*m} \\ &\equiv \mathbb{I}^{result,\, i}_{result*m,\, i+1} \\ &\equiv (0 \leqslant i + 1 \leqslant n) \land \left(result * m = m^{i+1}\right) \\ &\equiv (0 \leqslant i < n) \land \left(result = m^i\right) \\ &\equiv \mathbb{I} \land \mathbb{B} \ \checkmark \end{aligned}$$

$\mathbb{I} \land \neg\mathbb{B} \to \mathbb{Q}_{\mathbb{C}}$

$$\begin{aligned} \mathbb{I} \land \neg\mathbb{B} &\equiv (0 \leqslant i \leqslant n) \land \left(result = m^i\right) \land \neg (i < n) \\ &\equiv (0 \leqslant i \leqslant n) \land \left(result = m^i\right) \land (i \geqslant n) \\ &\equiv (i = n) \land \left(result = m^i\right) \\ &\equiv result = m^n \\ &\equiv \mathbb{Q}_{\mathbb{C}} \ \checkmark \end{aligned}$$

8.2.2. (b)

Falla la primera parte del teorema que es $\mathbb{P}_\mathbb{C} \to \mathbb{I}$. Notemos:

$$(n \geqslant 0) \land \neg (n = 0 \land m = 0) \land (i = 0) \land (result = 0) \to (0 \leqslant i \leqslant n) \land \left(result = m^i\right)$$
$$\to (n \geqslant 0) \land (0 = 1) \; ABS!$$

8.2.3. (c)

Es correcta ya que el valor de i no afecta al valor de *result* así que se pueden asignar dentro del ciclo en cualquier orden. La demostración de que es correcta la implementación es la misma que antes.

8.2.4. (d)

Para que la implementación sea correcta la precondición debe reforzarse pidiendo que $n \geqslant 2$.

8.3. Ejercicio 4

8.3.1. (a)

```
i := 1;
result := 0;
while(i < n + 1) do
    if(n% i = 0) then
        result := result + i;
        i := i + 1;
    else
        i := i + 1;
    endif
endwhile
```

8.3.2. (b)

Veamos que el invariante cumpla con las tres condiciones del teorema (1).

$$\begin{cases} \mathbb{P}_\mathbb{C} \equiv (n \geqslant 1) \land (i = 1) \land (result = 0) \\ \mathbb{Q}_\mathbb{C} \equiv result = \sum_{j=1}^{n} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \\ \mathbb{B} \equiv i \leqslant n \\ \mathbb{I} \equiv (1 \leqslant i \leqslant n) \land \left(result = \sum_{j=1}^{i} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi})\right) \end{cases}$$

$\mathbb{P}_\mathbb{C} \to \mathbb{I}$

$$(n \geqslant 1) \land (i = 1) \land (result = 0) \to (1 \leqslant i \leqslant n) \land \left(result = \sum_{j=1}^{i} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right)$$
$$\to (n \geqslant 1) \land \left(0 = \sum_{j=1}^{1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right)$$
$$\equiv (n \geqslant 1) \land (0 = 1) \; ABS!$$

El problema es que en el programa que hice i llega hasta $n+1$ y *result* empieza siendo 0, no 1. Tomemos entonces el invariante:

$$\mathbb{I} \equiv (1 \leqslant i \leqslant n+1) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right)$$

Entonces:

$$\begin{aligned}(n \geqslant 1) \wedge (i = 1) \wedge (result = 0) &\to (1 \leqslant i \leqslant n+1) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \\ &\to (n \geqslant 0) \wedge \left(0 = \sum_{j=1}^{0} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \\ &\to (n \geqslant 1) \ \checkmark\end{aligned}$$

$\{\mathbb{I} \wedge \mathbb{B}\}\, S\, \{\mathbb{I}\}$

$$\begin{aligned}\mathbb{I} \wedge \mathbb{B} &\equiv (1 \leqslant i \leqslant n+1) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \wedge (i \leqslant n) \\ &\equiv (1 \leqslant i \leqslant n) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right)\end{aligned}$$

$$\begin{aligned}\text{wp}\,(S, \mathbb{I}) &\equiv \left(\text{def}\,(i) \wedge_L \left(\mathbb{B}_{\text{if}} \wedge \text{wp}\left(result \stackrel{\text{def}}{:=} result + i, \mathbb{I} \right) \right) \right) \vee \\ &\quad (\text{def}\,(i) \wedge_L (\neg \mathbb{B}_{\text{if}} \wedge \text{wp}\,(\text{skip}, \mathbb{I}))) \\ &\equiv \left((n \mod i = 0) \wedge \text{wp}\left(result \stackrel{\text{def}}{:=} result + i;\ i \stackrel{\text{def}}{:=} i+1, \mathbb{I} \right) \right) \vee \left((n \mod i \neq 0) \wedge \text{wp}\left(i \stackrel{\text{def}}{:=} i+1, \mathbb{I} \right) \right) \\ &\equiv \left((n \mod i = 0) \wedge \left(\text{def}\,(i) \wedge_L \mathbb{I}^{result,\, i}_{result+i,\, i+1} \right) \right) \vee \left((n \mod i \neq 0) \wedge \left(\text{def}\,(i) \wedge_L \mathbb{I}^{i}_{i+1} \right) \right) \\ &\equiv \left((n \mod i = 0) \wedge (1 \leqslant i+1 \leqslant n+1) \wedge \left(result + i = \sum_{j=1}^{i} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \right) \vee \\ &\quad \left((n \mod i \neq 0) \wedge (1 \leqslant i+1 \leqslant n+1) \wedge \left(result = \sum_{j=1}^{i} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \right) \\ &\equiv (1 \leqslant i \leqslant n) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \\ &\equiv \mathbb{I} \ \checkmark\end{aligned}$$

$\mathbb{I} \wedge \neg \mathbb{B} \to \mathbb{Q}_{\mathbb{C}}$

$$\begin{aligned}
\mathbb{I} \wedge \neg \mathbb{B} &\equiv (1 \leqslant i \leqslant n+1) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \wedge \neg (i \leqslant n) \\
&\equiv (1 \leqslant i \leqslant n+1) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \wedge (i > n) \\
&\equiv (i = n+1) \wedge \left(result = \sum_{j=1}^{i-1} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \right) \\
&\equiv result = \sum_{j=1}^{n} (\text{if } n \mod j = 0 \text{ then } j \text{ else } 0 \text{ fi}) \\
&\equiv \mathbb{Q}_{\mathbb{C}} \checkmark
\end{aligned}$$

8.4. Ejercicio 6

8.4.1. (a)

$$\begin{cases} \mathbb{P}_{\mathbb{C}} \equiv (|s| \geqslant 1) \wedge (i = 0) \wedge (j = 1) \\ \mathbb{Q}_{\mathbb{C}} \equiv (0 \leqslant i < |s|) \wedge_L (\forall j : \mathbb{Z}) \, (0 \leqslant j < |s| \to_L s\,[j] \leqslant s\,[i]) \end{cases}$$

8.4.2. (b)

$$\begin{cases} \mathbb{P}_{\mathbb{C}} \equiv (|s| \geqslant 1) \wedge (i = 0) \wedge (j = 1) \\ \mathbb{Q}_{\mathbb{C}} \equiv (0 \leqslant i < |s|) \wedge_L (\forall j : \mathbb{Z}) \, (0 \leqslant j < |s| \to_L s\,[j] \leqslant s\,[i]) \\ \mathbb{B} \equiv j < |s| \\ \mathbb{I} \equiv ((0 \leqslant i < |s|) \wedge (1 \leqslant j \leqslant |s|)) \wedge_L (\forall k : \mathbb{Z}) \, (0 \leqslant k < j \to_L s\,[k] \leqslant s\,[i]) \end{cases}$$

$\mathbb{P}_{\mathbb{C}} \to \mathbb{I}$

$$\begin{aligned}
(|s| \geqslant 1) \wedge (i = 0) \wedge (j = 1) &\to ((0 \leqslant i < |s|) \wedge (1 \leqslant j \leqslant |s|)) \wedge_L (\forall k : \mathbb{Z}) \, (0 \leqslant k < j \to_L s\,[k] \leqslant s\,[i]) \\
&\to (|s| \geqslant 1) \wedge_L (\forall k : \mathbb{Z}) \, (0 \leqslant k < 1 \to_L s\,[k] \leqslant s\,[0]) \\
&\to (|s| \geqslant 1) \wedge_L (s\,[0] \leqslant s\,[0]) \\
&\to (|s| \geqslant 1) \checkmark
\end{aligned}$$

$\{\mathbb{I} \wedge \mathbb{B}\}\, S\, \{\mathbb{I}\}$

$$\begin{aligned}
\mathbb{I} \wedge \mathbb{B} &\equiv ((0 \leqslant i < |s|) \wedge (1 \leqslant j \leqslant |s|)) \wedge_L (\forall k : \mathbb{Z}) \, (0 \leqslant k < j \to_L s\,[k] \leqslant s\,[i]) \wedge (j < |s|) \\
&\equiv ((0 \leqslant i < |s|) \wedge (1 \leqslant j < |s|)) \wedge_L (\forall k : \mathbb{Z}) \, (0 \leqslant k < j \to_L s\,[k] \leqslant s\,[i])
\end{aligned}$$

$$
\begin{aligned}
\mathrm{wp}\left(S, \mathbb{I}\right) &\equiv \mathrm{wp}\left(\text{if...}, \mathrm{wp}\left(j \overset{\text{def}}{:=} j+1, \mathbb{I}\right)\right) \\
&\equiv \mathrm{wp}\left(\text{if...}, \mathrm{def}\left(j\right) \wedge_L \mathbb{I}_{j+1}^{j}\right) \\
&\equiv \mathrm{wp}\left(\text{if...}, \mathbb{I}_{j+1}^{j}\right) \\
&\equiv \left(\mathrm{def}\left(j\right) \wedge_L \left(\mathbb{B}_{\text{if}} \wedge \mathrm{wp}\left(i \overset{\text{def}}{:=} j, \mathbb{I}_{j+1}^{j}\right)\right)\right) \vee \\
&\left(\mathrm{def}\left(j\right) \wedge_L \left(\neg \mathbb{B}_{\text{if}} \wedge \mathrm{wp}\left(\text{skip}, \mathbb{I}_{j+1}^{j}\right)\right)\right) \\
&\equiv \left(\left(j < |s|\right) \wedge \left(\mathrm{def}\left(i\right) \wedge_L \mathbb{I}_{j+1,\, j}^{j,\, i}\right)\right) \vee \left(\left(j \geqslant |s|\right) \wedge \mathbb{I}_{j+1}^{j}\right) \\
&\equiv \left(\left(j < |s|\right) \wedge \left(\left(0 \leqslant j < |s|\right) \wedge \left(1 \leqslant j+1 \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j+1 \rightarrow_L s\left[k\right] \leqslant s\left[j\right]\right)\right) \vee \\
&\left(\left(j \geqslant |s|\right) \wedge \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j+1 \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j+1 \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right)\right) \\
&\equiv \left(\left(0 \leqslant j < |s|\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k \leqslant j \rightarrow_L s\left[k\right] \leqslant s\left[j\right]\right)\right) \vee \\
&\left(\mathrm{False} \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j+1 \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right)\right) \\
&\equiv \left(\left(0 \leqslant i < |s|\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k \leqslant j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right)\right) \vee \mathrm{False} \\
&\equiv \left(0 \leqslant i < |s|\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k \leqslant j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \\
&\equiv \mathbb{I} \wedge \mathbb{B} \checkmark
\end{aligned}
$$

$\mathbb{I} \wedge \neg \mathbb{B} \rightarrow \mathbb{Q}_{\mathbb{C}}$

$$
\begin{aligned}
\mathbb{I} \wedge \neg \mathbb{B} &\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \wedge \neg \left(j < |s|\right) \\
&\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \wedge \left(j \geqslant |s|\right) \\
&\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(j = |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \\
&\equiv \left(0 \leqslant i < |s|\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < |s| \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \\
&\equiv \left(0 \leqslant i < |s|\right) \wedge_L \left(\forall j : \mathbb{Z}\right) \left(0 \leqslant j < |s| \rightarrow_L s\left[j\right] \leqslant s\left[i\right]\right) \\
&\equiv \mathbb{Q}_{\mathbb{C}} \checkmark
\end{aligned}
$$

8.4.3. (c)

Tomemos:

$$f\left(s, j\right) = |s| - j$$

Notemos:

$$
\begin{aligned}
\mathbb{I} \wedge \left(f\left(s, j\right) \leqslant 0\right) &\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \wedge \left(|s| - j \leqslant 0\right) \\
&\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \wedge \left(j \geqslant |s|\right) \\
&\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(j = |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < |s| \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \\
&\rightarrow \neg \mathbb{B} \checkmark
\end{aligned}
$$

Veamos ahora que esta función variante decrece con cada iteración del ciclo. Quiero ver que:

$$\left\{\mathbb{I} \wedge \mathbb{B} \wedge \left(V_0 = f\right)\right\} S \left\{f < V_0\right\}$$

Notemos:

$$
\begin{aligned}
\mathbb{I} \wedge \mathbb{B} \wedge \left(V_0 = f\right) &\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j \leqslant |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \wedge \left(j < |s|\right) \wedge \left(V_0 = |s| - j\right) \\
&\equiv \left(\left(0 \leqslant i < |s|\right) \wedge \left(1 \leqslant j < |s|\right)\right) \wedge_L \left(\forall k : \mathbb{Z}\right) \left(0 \leqslant k < j \rightarrow_L s\left[k\right] \leqslant s\left[i\right]\right) \wedge \left(V_0 = |s| - j\right)
\end{aligned}
$$

$$
\begin{aligned}
\operatorname{wp}(S, f < V_0) &\equiv \operatorname{wp}\left(\text{if...}, \operatorname{wp}\left(j \overset{\text{def}}{:=} j+1, f < V_0\right)\right) \\
&\equiv \operatorname{wp}(\text{if...}, ((\operatorname{def}(j) \wedge \operatorname{def}(s)) \wedge_L (V_0 = |s| - j)) \wedge_L |s| - (j+1) < V_0) \\
&\equiv \operatorname{wp}(\text{if...}, (V_0 = |s| - j) \wedge_L (|s| - (j+1) < V_0)) \\
&\equiv ((V_0 = |s| - j) \wedge ((0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|))) \wedge_L \\
&\left(\left((s[j] > s[i]) \wedge \operatorname{wp}\left(i \overset{\text{def}}{:=} j, |s| - (j+1) < V_0\right)\right) \vee ((s[j] \leqslant s[i]) \wedge \operatorname{wp}(\text{skip}, |s| - (j+1) < V_0))\right) \\
&\equiv ((V_0 = |s| - j) \wedge (0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|)) \wedge_L \\
&(((s[j] > s[i]) \wedge |s| - (j+1) < V_0) \vee ((s[j] \leqslant s[i]) \wedge |s| - (j+1) < V_0)) \\
&\equiv ((V_0 = |s| - j) \wedge (0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|)) \wedge_L (|s| - (j+1) < V_0) \\
&\equiv ((V_0 = |s| - j) \wedge (0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|)) \wedge_L (|s| - (j+1) < |s| - j) \\
&\equiv ((V_0 = |s| - j) \wedge (0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|)) \wedge_L (-1 < 0) \\
&\equiv (V_0 = |s| - j) \wedge (0 \leqslant i < |s|) \wedge (0 \leqslant j < |s|) \\
&\to ((0 \leqslant i < |s|) \wedge (1 \leqslant j < |s|)) \wedge_L (\forall k : \mathbb{Z})\,(0 \leqslant k < j \to_L s[k] \leqslant s[i]) \wedge (V_0 = |s| - j) \ \checkmark
\end{aligned}
$$

8.5. Ejercicio 7

8.5.1. (a)

$$
\begin{cases}
\mathbb{P}_{\mathbb{C}} \equiv (|s| = |r|) \wedge (i = 0) \\
\mathbb{Q}_{\mathbb{C}} \equiv (|s| = |r|) \wedge (\forall j : \mathbb{Z})\,(0 \leqslant j < |s| \to_L s[j] = r[j])
\end{cases}
$$

8.5.2. (b)

$$
\begin{cases}
\mathbb{P}_{\mathbb{C}} \equiv (|s| = |r|) \wedge (i = 0) \\
\mathbb{Q}_{\mathbb{C}} \equiv (|s| = |r|) \wedge (\forall j : \mathbb{Z})\,(0 \leqslant j < |s| \to_L s[j] = r[j]) \\
\mathbb{B} \equiv i < |s| \\
\mathbb{I} \equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \to_L s[j] = r[j])
\end{cases}
$$

$\mathbb{P}_{\mathbb{C}} \to \mathbb{I}$

$$
\begin{aligned}
(|s| = |r|) \wedge (i = 0) &\to (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \to_L s[j] = r[j]) \\
&\to (|s| = |r|) \ \checkmark
\end{aligned}
$$

$\{\mathbb{I} \wedge \mathbb{B}\}\, S\, \{\mathbb{I}\}$

$$
\begin{aligned}
\mathbb{I} \wedge \mathbb{B} &\equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \to_L s[j] = r[j]) \wedge (i < |s|) \\
&\equiv (|s| = |r|) \wedge (0 \leqslant i < |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \to_L s[j] = r[j])
\end{aligned}
$$

$$
\begin{aligned}
\operatorname{wp}(S, \mathbb{I}) &\equiv \operatorname{wp}\left(r[i] \overset{\text{def}}{:=} s[i], \operatorname{wp}\left(i \overset{\text{def}}{:=} i+1, \mathbb{I}\right)\right) \\
&\equiv \operatorname{wp}\left(r[i] \overset{\text{def}}{:=} s[i], \operatorname{def}(i) \wedge_L \mathbb{I}^{i}_{i+1}\right) \\
&\equiv ((\operatorname{def}(r) \wedge \operatorname{def}(s)) \wedge_L (0 \leqslant i < |s|) \wedge (0 \leqslant i < |r|)) \wedge_L \mathbb{I}^{r[i],\, i}_{s[i],\, i+1} \\
&\equiv ((0 \leqslant i < |s|) \wedge (0 \leqslant i < |r|)) \wedge_L \\
&(|s| = |r|) \wedge (0 \leqslant i+1 \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i+1 \to_L s[j] = r[j]) \\
&\equiv (|s| = |r|) \wedge (-1 \leqslant i < |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j \leqslant i \to_L s[j] = r[j]) \\
&\to \mathbb{I} \wedge \mathbb{B} \ \checkmark
\end{aligned}
$$

$(\mathbb{I} \wedge \neg\mathbb{B}) \rightarrow \mathbb{Q}_\mathbb{C}$

$$\begin{aligned}
\mathbb{I} \wedge \neg\mathbb{B} &\equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \wedge \neg\,(i < |s|) \\
&\equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \wedge (i \geqslant |s|) \\
&\equiv (|s| = |r|) \wedge (i = |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \\
&\equiv (|s| = |r|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < |s| \rightarrow_L s\,[j] = r\,[j]) \\
&\equiv \mathbb{Q}_\mathbb{C}\ \checkmark
\end{aligned}$$

8.5.3. (c)

Propongo:

$$f\,(s, i) = |s| - i$$

$\mathbb{I} \wedge (f \leqslant 0) \rightarrow \neg\mathbb{B}$

$$\begin{aligned}
\mathbb{I} \wedge (f \leqslant 0) &\equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \wedge (|s| - i \leqslant 0) \\
&\equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \wedge (i \geqslant |s|) \\
&\equiv \mathbb{I} \wedge \neg\mathbb{B} \\
&\rightarrow \neg\mathbb{B}\ \checkmark
\end{aligned}$$

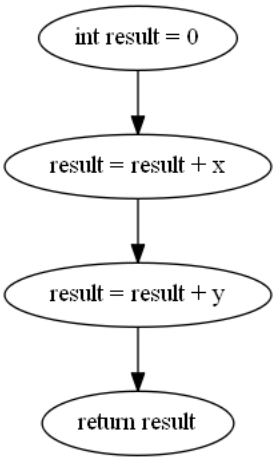
$\{\mathbb{I} \wedge \mathbb{B} \wedge (V_0 = |s| - i)\}\ S\ \{f < V_0\}$

$$\begin{aligned}
\mathbb{I} \wedge \mathbb{B} \wedge (V_0 = |s| - i) &\equiv (|s| = |r|) \wedge (0 \leqslant i \leqslant |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \wedge (i < |s|) \wedge (V_0 = |s| - i) \\
&\equiv (|s| = |r|) \wedge (0 \leqslant i < |s|) \wedge_L (\forall j : \mathbb{Z})\,(0 \leqslant j < i \rightarrow_L s\,[j] = r\,[j]) \wedge (|s| - i = V_0)
\end{aligned}$$

$$\begin{aligned}
\mathrm{wp}\,(S, f < V_0) &\equiv \mathrm{wp}\left(r\,[i] \overset{\text{def}}{:=} s\,[i], \mathrm{wp}\left(i \overset{\text{def}}{:=} i + 1, f < V_0\right)\right) \\
&\equiv \mathrm{wp}\left(r\,[i] \overset{\text{def}}{:=} s\,[i], (\mathrm{def}\,(s) \wedge \mathrm{def}\,(i)) \wedge_L (V_0 = |s| - i) \wedge (|s| - (i+1) < |s| - i)\right) \\
&\equiv \mathrm{wp}\left(r\,[i] \overset{\text{def}}{:=} s\,[i], V_0 = |s| - i\right) \\
&\equiv (\mathrm{def}\,(r) \wedge \mathrm{def}\,(s)) \wedge_L ((0 \leqslant i < |s|) \wedge (0 \leqslant i < |r|)) \wedge (V_0 = |s| - i) \\
&\equiv (0 \leqslant i < |s|) \wedge (0 \leqslant i < |r|) \wedge (V_0 = |s| - i) \\
&\rightarrow \mathbb{I} \wedge \mathbb{B} \wedge (V_0 = |s| - i)\ \checkmark
\end{aligned}$$

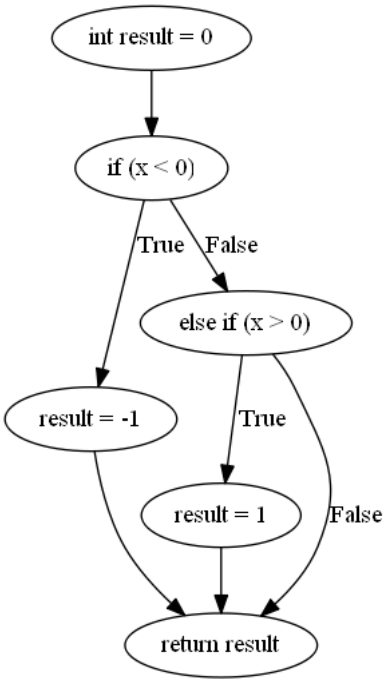
9. Guía 6: Testing

9.1. Ejercicio 3



Como el programa es lineal entonces cualquier test case recorre todos los nodos. Así que un posible test suite es el test case $(x = 1, y = 2; result = 3)$.

9.2. Ejercicio 5

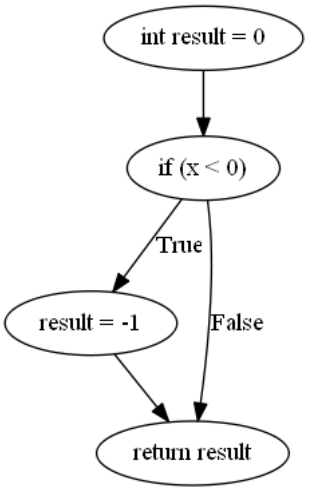


Un posible test suite es:

- $(x = 0; result = 0)$.
- $(x = 1; result = 1)$.
- $(x = -1; result = -1)$.

Este test suite ejecuta todas las líneas y todos los posibles branches del programa.

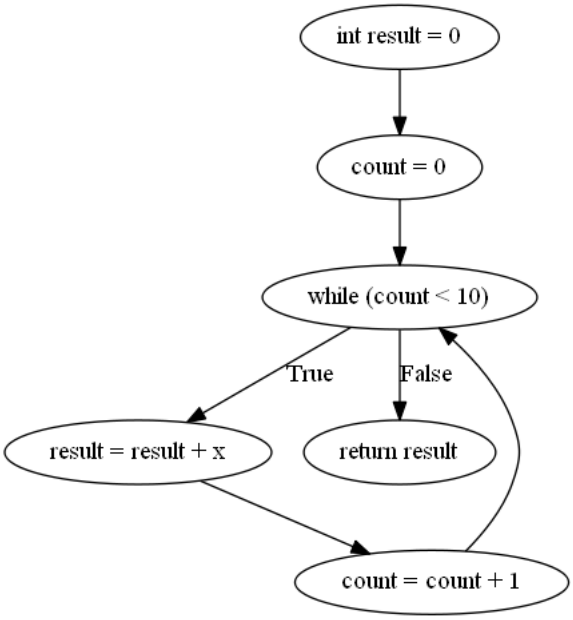
9.3. Ejercicio 6



Un posible test suite es:

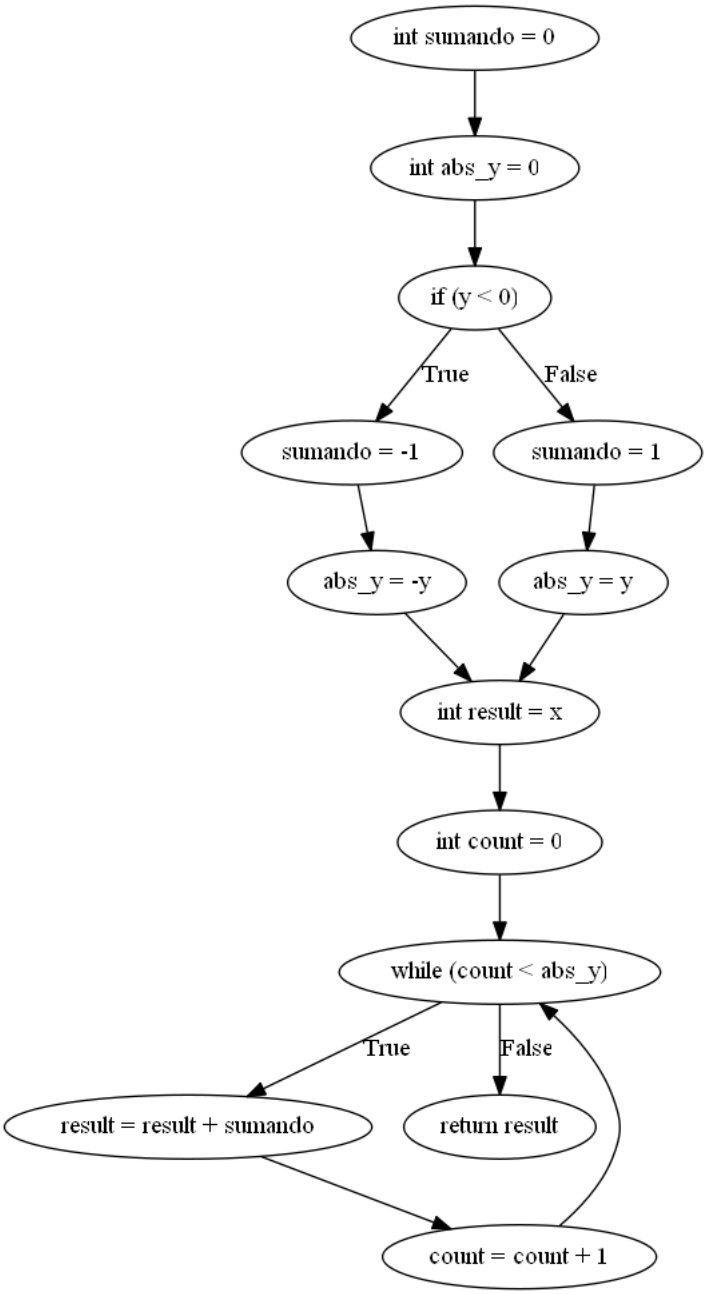
- $(x = -1; result = -1)$.
- $(x = 0; result = 0)$.

9.4. Ejercicio 8



Un posible test suite es $(x = 1; result = 10)$.

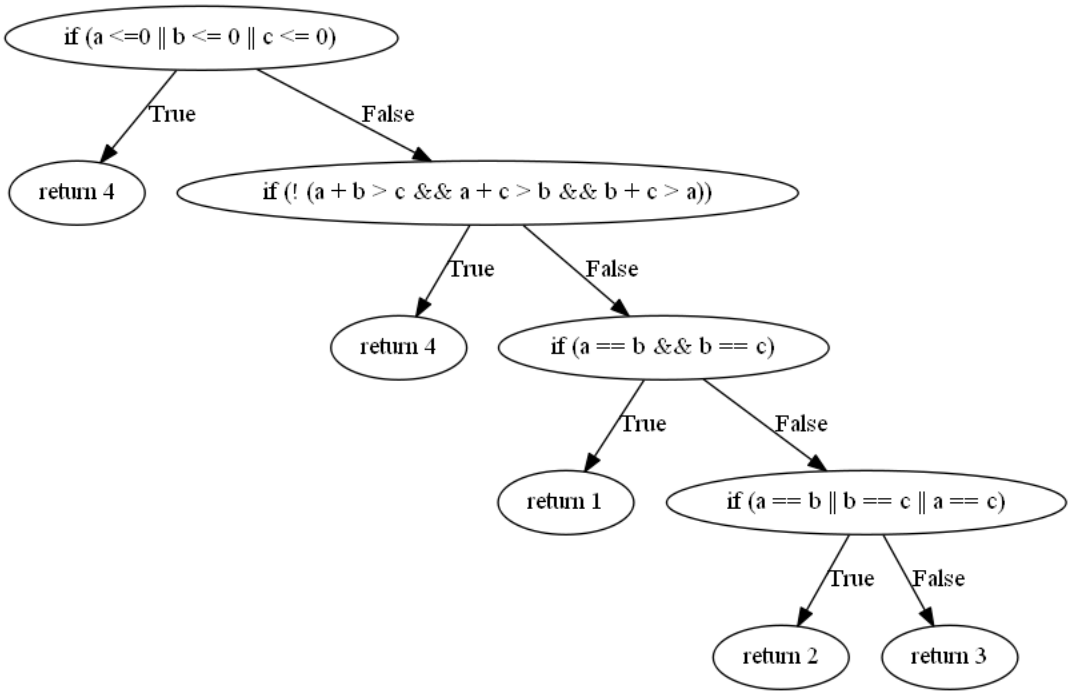
9.5. Ejercicio 9



Un posible test suite es:

- $(x = 1, y = -1; result = 0)$.
- $(x = 1, y = 1; result = 2)$.

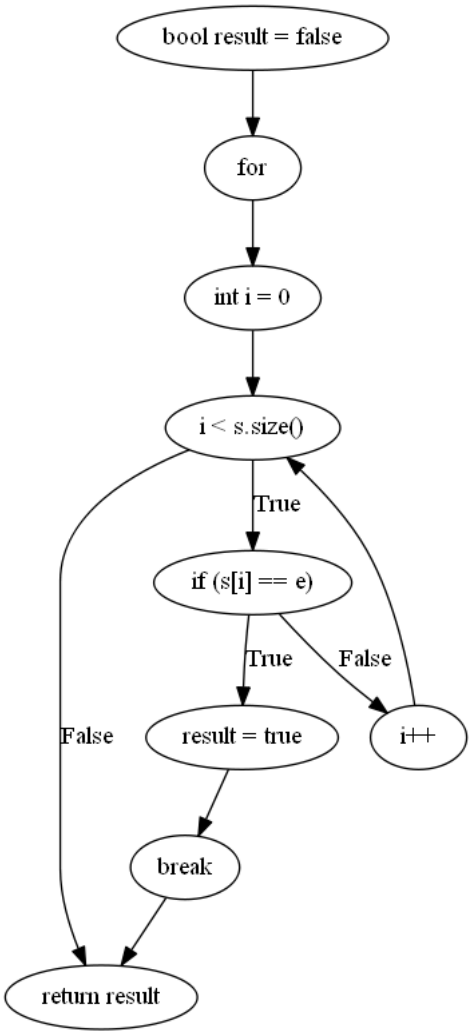
9.6. Ejercicio 11



Un posible test suite es:

- $(a = 0, b = 1, c = 0; 4)$.
- $(a = 1, b = 2, c = 3; 4)$.
- $(a = 1, b = 1, c = 1; 1)$.
- $(a = 1, b = 1, c = 2; 2)$.
- $(a = 4, b = 5, c = 2; 3)$.

9.7. Ejercicio 14



Un test suite posible es:

- $(s = [0, 1, 2], e = 1; \text{true})$.
- $(s = [0, 1, 2], e = 3; \text{false})$.

10. Guía 7: Tiempo de Ejecución de Peor Caso de un Programa

10.1. Ejercicio 2

10.1.1. (a)

Siempre ignorando las líneas que toman tiempo unitario notemos que el ciclo se ejecuta $\frac{s}{2} + 1$ veces. Entonces:

$$T(s) \propto \frac{s}{2} + 1$$

$$\implies T(s) \in \mathcal{O}(s)$$

10.1.2. (b)

El ciclo se ejecuta 10000 veces, siempre. Entonces:

$$T(s) \in \mathcal{O}(1)$$

10.1.3. (c)

El primer ciclo se ejecuta n veces. De esas n veces, 100 veces el condicional es verdadero y por lo tanto se ejecuta el segundo ciclo, que se ejecuta n veces cada vez. Es decir:

$$T(s) \propto \begin{cases} 100 \cdot s + (n - 100) & s \geqslant 100 \\ s \cdot s & s \leqslant 100 \end{cases}$$

Como debemos considerar el peor caso entonces consideremos $s \geqslant 100$. Entonces:

$$T(s) \propto 101s - 100$$

$$\implies T(s) \in \mathcal{O}(s)$$

Sólo para $s \leqslant 100$ el tiempo de ejecución es de $\mathcal{O}(s^2)$ pero aún en ese caso el peor tiempo de ejecución en $100^2 \leqslant 101s - 100 \, \forall s \geqslant 100$.

10.1.4. (d)

Ambos ciclos se ejecutan max_iter veces. max_iter es 1000 como mínimo. Entonces:

$$T(s) \propto \begin{cases} 1000 \cdot 1000 & s \geqslant 1000 \\ s \cdot s & s \leqslant 1000 \end{cases}$$

$$\implies T(s) \in \mathcal{O}(1)$$

10.1.5. (e)

La primera línea tiene un tiempo de ejecución de $\mathcal{O}(s_1 + s_2)$. El primer ciclo se ejecuta s_1 veces y el segundo se ejecuta s_2 veces, así que ambos ciclos tienen un tiempo de ejecución de $\mathcal{O}(s_1 + s_2)$. Entonces:

$$T(s_1, s_2) \in \mathcal{O}(s_1 + s_2)$$

10.2. Ejercicio 5

El programa lo que hace es ir recorriendo la secuencia y fijarse cuál es la mayor subsecuencia que tiene todos los número iguales dentro de la secuencia (denominado como "meseta" por el programador). La especificación del programa sería:

```
proc mesetaMasLarga(in s :Seq⟨ℤ⟩, out maxMeseta : ℤ){
    Pre{True}
    Post{(∀i, j : ℤ) (esMeseta (i, j, s) →_L maxMeseta ⩾ j − i) ∧ ¬ (∃i, j : ℤ) (esMeseta (i, j, s) ∧ maxMeseta < j − i)}
}
pred esMeseta(i : ℤ, j : ℤ, s :Seq⟨ℤ⟩){
    (0 ⩽ i < |s|) ∧ (0 ⩽ j ⩽ |s|) ∧ (i < j) ∧ (j ≠ |s| →_L s [i] ≠ s [j]) ∧_L
    (∀k : ℤ) (i ⩽ k < j →_L s [k] = s [i])
}
```

El primer ciclo se ejecuta una vez para cada meseta del ciclo y el segundo se ejecuta para cada largo de meseta. Ya que la suma de los largos de las mesetas es igual al largo de la secuencia entonces sabemos que en total los ciclos se van a ejecutar s veces. Depende de cada ciclo la cantidad exacta de veces que se ejecuta cada ciclo, pero en total ambos se ejecutan s. Entonces:

$$T(s) \in \mathcal{O}(s)$$

Si quisiéramos hacer el programa usando sólo un ciclo podemos hacerlo de la siguiente forma:

```
int mesetaMasLarga(vector<int> s) {
    int i = 0;
    int maxMeseta = 0;
    int meseta = 1;
    while (i < s.size() - 1) {
        if (s[i] == s[i + 1]) {
            meseta = meseta + 1;
            i = i + 1;
        } else {
            if (maxMeseta < meseta) {
              maxMeseta = meseta;
              meseta = 1;
            } else {
              meseta = 1;
            }
        }
    }
    return maxMeseta;
}
```

10.3. Ejercicio 5

El programa hacerAlgo ejecuta 100 veces el programa contarApariciones, así que:

$$T(s) \propto 100\tilde{T}(s)$$

donde $\tilde{T}(s)$ es el tiempo de ejecución de contarApariciones. El ciclo en contarApariciones se ejecuta s veces así que:

$$T(s) \propto 100s$$

$$\implies T(s) \in \mathcal{O}(s)$$

10.4. Ejercicio 9

```
bool esTriangular(vector<vector<int>> A){
    bool res = true;
    if (A.size() < 2 || A.size() != A[0].size()){
        res = false;
    } else {
        for (int i = 1; i < A.size(); i++) {
            for (int j = 0; j < i; j++) {
```

```
                if (A[i][j] != 0) {
                  res = false;
                }
              }
            }
        }
        return res;
    }
    int detTriangular(vector<vector<int>> A) {
        int det = 1;
        for (int i = 0; i < A.size(); i++) {
            det = det * A[i][i]
        }
        return det;
    }
```

Si la matriz es de $n \times n$ entonces hay n^2 elementos en la matriz. Tomemos $m = n^2$, T_1 como el tiempo de ejecución de esTriangular y T_2 como el tiempo de ejecución de detTriangular. El primer ciclo de esTriangular se ejecuta $n-1$ veces y el segundo depende de en qué fila de la matriz esté. Entonces:

$$\begin{aligned} T_1 &\propto \sum_{i=1}^{n} i \\ &= \frac{n\,(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n \\ &= \frac{1}{2}m + \frac{1}{2}\sqrt{m} \end{aligned}$$

$$\implies T_1 \in \mathcal{O}\,(m)$$

El ciclo de detTriangular se ejecuta n veces siempre. Entonces:

$$T_2 \in \mathcal{O}\left(\sqrt{m}\right)$$

10.5. Ejercicio 12

```
int maxSubseqImpar(vector<int> s) {
    int res = 0;
    int largo = 0;
    for (int i = 0; i < s.size(); i++){
        if (s[i]% 2 == 1) {
            largo = largo + 1;
        } else {
            if (largo > res) {
              res = largo;
            }
            largo = 0;
        }
    }
    return res;
}
```

11. Guía 8: Búsqueda y Ordenamiento

11.1. Ejercicio 4

11.1.1. (a)

```
int sumaBinaria(vector<int> s) {
    int result = 0;
    for (int i = 0; i < s.size(); i++) {
        result = result + s[i];
    }
}
```

11.1.2. (b)

```
int sumaBinariaOrdenada(vector<int> s) {
    int ans;
    if (s.size() == 0) {
        ans = 0;
    } else if (s.size() == 1) {
        ans = s[0];
    } else if (s[s.size() - 1] == 0) {
        ans = 0;
    } else if (s[0] == 1) {
        ans = s.size();
    } else {
        int low = 0;
        int high = s.size() - 1;
        while (low + 1 < high) {
            int mid = (low + high) / 2;
            if (s[mid] == 0) {
              low = mid;
            } else {
              high = mid;
            }
        }
        ans = s.size() - (low + 1);
    }
    return ans;
}
```

Este programa usa el algoritmo de búsqueda en secuencia ordenada. El tiempo de ejecución es de $T(s) \in \mathcal{O}(\log_2(s-1))$.

11.1.3. (c)

```
int sumaBinariaOrdenada(vector<int> s) {
    int ans;
    if (s.size() == 0) {
        ans = 0;
```

```
    } else if (s.size() == 1) {
        ans = s[0];
    } else if (s[s.size() - 1] == 15) {
        ans = 15 * s.size();
    } else if (s[0] == 22) {
        ans = 22 * s.size();
    } else {
        int low = 0;
        int high = s.size() - 1;
        while (low + 1 < high) {
            int mid = (low + high) / 2;
            if (s[mid] == 15) {
              low = mid;
            } else {
              high = mid;
            }
        }
        ans = (low + 1) * 15 + (s.size() - (low + 1)) * 22;
    }
    return ans;
}
```

Este programa es análogo al anterior.

11.2. Ejercicio 5

11.2.1. (a)

```
int busquedaFila(vector<vector<int>> s, int e) {
    int ans;
    if (s.size() == 1) {
        ans = 0;
    } else {
        int low = 0;
        int high = s.size() - 1;
        while (low + 1 < high) {
            int mid = (low + high) / 2;
            if (s[mid][0] <= e) {
              low = mid;
            } else {
              high = mid;
            }
        }
        ans = low;
    }
    return ans;
}
int busquedaColumna(vector<int> s, int e) {
```

```
    int ans;
    if (s.size() == 0 || s.size() == 1) {
        ans = 0;
    } else if (s[s.size() - 1] < e) {
        ans = 0;
    } else {
        int low = 0;
        int high = s.size() - 1;
        while (low + 1 < high) {
            int mid = (low + high) / 2;
            if (s[mid][0] <= e) {
              low = mid;
            } else {
              high = mid;
            }
        }
        ans = low;
    }
}
int contarAparicionesSinRepe(vector<vector<int>> s, int e) {
    int ans;
    if (s.size() == 0){
        ans = 0;
    } else if (s[0].size() == 0) {
        ans = 0;
    } else {
        int f = busquedaFila(s, e);
        int c = busquedaColumna(s[f], e);
        if (s[f][c] == e) {
            ans = 1;
        } else {
            ans = 0;
        }
    }
    return ans;
}
```

11.3. Ejercicio 6

11.3.1. (a)

```
int contarElemento(vector<int> v, int e) {
    int ans = 0;
    for (int i = 0; i < v.size(); i++) {
        if (v[i] == e) {
            ans = ans + 1;
        }
```

```
    }
    return ans;
}
```

$$T(v) \in \mathcal{O}(v)$$

11.3.2. (b)

```
int maximaDiferencia(vector<int> v, int d) {
    int max = v[0];
    int min = v[0];
    for (int i = 0; i < v.size(); i++) {
        if (v[i] > max) {
            max = v[i];
        }
        if (v[i] < min) {
            min = v[i];
        }
    }
    return max - min;
}
```

$$T(v) \in \mathcal{O}(v)$$

11.3.3. (c)

```
int moda(vector<int> v) {
    int moda = v[0];
    int veces = contarElemento(v[0]);
    for (int i = 1; i < v.size(); i++) {
        if (contarElemento(v, v[i]) > veces) {
            moda = v[i];
            veces = contarElemento(v, v[i]);
        }
    }
    return moda;
}
```

$$T(v) \in \mathcal{O}(v^2)$$

11.3.4. (d)

```
vector<int> fillNumberCount(vector<int> &v, int r1, int r2) {
    vector<int> numberCount(r2 - r1 + 1, 0);
    for (int i = 0; i < numberCount.size(); i++) {
        numberCount[i] = contarElemento(v, i + r1);
    }
    return numberCount;
}
void populate(vector<int> &v, vector<int> &numberCount) {
```

```
    int i = 0;
    int j = 0;
    while (i < v.size()) {
        while (j < numberCount.size()) {
            if (numberCount[j] > 0) {
              s[i] = j + r1;
              i++;
            } else {
              j++;
            }
        }
    }
}
void ordenarNumerosAcotados(vector<int> &v) {
    vector<int> numberCount = fillNumberCount(v);
    populate(v, numberCount);
}
```

No creo que pueda hacerse en tiempo lineal usando el algoritmo anterior ya que no considera la cantidad de elementos distintos que hay en la secuencia. Este algoritmo es una adaptación del algoritmo de Dutch National Flag.

$$T(v) \in \mathcal{O}(v)$$

11.3.5. (e)

El primer algoritmo se convierte en un algoritmo de búsqueda sobre una secuencia ordenada, así que es de $\mathcal{O}(\log_2(v-1))$. El segundo es de $\mathcal{O}(1)$, ya que es simplemente hacer $v[|v|-1]-v[0]$. El tercero no creo que se pueda mejorar, no se me ocurre cómo.

11.4. Ejercicio 8

La estrategia es usar el algoritmo del NationalDutchFlag:

```
vector<int> fillLetterCount(string &s) {
    vector<int> letterCount(112956, 0);
    /* Voy a asumir que en Unicode los caracteres son enteros de 0 a 112956
     * (probablemente no sea asi pero bue). */
    for (int i = 0; i < s.size(); i++) {
        for (int j = 0; i < letterCount.size(); j++) {
            if (ord(s[i]) == j) {
              letterCount[j]++;
            }
        }
    }
    return letterCount;
}

void populate(string &s, vector<int> &letterCount) {
```

```
    int i = 0;
    int j = 0;
    while (i < s.size()) {
        while (j < letterCount.size()) {
            if (letterCount[j] > 0) {
              s[i] = uni(j);
              i++;
            } else {
              j++;
            }
        }
    }
}
void ordenarCaracteres(string &s) {
    vector<int> letterCount = fillLetterCount(s);
    populate(s, letterCount);
}
```

11.5. Ejercicio 10

Ver el proyecto de C++ en la carpeta “Algoritmos Utiles” en el material de la materia.

11.6. Ejercicio 12

Ver el proyecto de C++ en la carpeta “Algoritmos Utiles” en el material de la materia.

11.7. Ejercicio 14

Ver el proyecto de C++ en la carpeta “Algoritmos Utiles” en el material de la materia.

Índice alfabético

Algoritmo, 5
Algoritmo de Knuth, Morris y Pratt, 22
Asignación, 11

Bifijo, 23

Ciclo, 11
Computadora, 5
Concatenación, 8
Conditional AND, 6
Conditional IF, 7
Conditional OR, 6
Contrato, 5
Control Flow-Graph, 14
Correcto, 11
Criterio de Adecuación, 15

Eficiencia, 16
Ejecución, 11
Especificación, 5
Estado, 11

Función Auxiliar, 6
Función Variante, 13

Guarda, 11

Indexación, 8
Invariante, 11

Matriz, 10

Notación O Grande, 16

Parámetro, 5
Poscondición, 5
Precondición, 5
Predicado Auxiliar, 6
Programa, 5
Programa Correcto, 6

Secuencia, 8
Semántica Trivaluada, 6
Sobre-especificación, 6
String, 22
Sub-especificación, 6

Teorema del Invariante, 11
Test Case, 14
Test de Caja Negra, 14
Test Estructural, 14
Test Funcional, 14
Test Input, 14
Test Suite, 14
Testing, 14
Tests de Caja Blanca, 14
Tiempo de Ejecución, 16
Tiempo de Ejecución Lineal, 16
Tipos de Datos, 5
Tripla de Hoare, 11