Introducción a la Programación
Algoritmos y Estructuras de Datos I

Parcial - 1er cuatrimestre 2023

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

1.1	1.2	2.1	2.2	3	4.1	4.2
~~1~~	1	~~2~~	2	2	1	1

Apellido LLOSA FERNANDEZ Nombre DANTE

LU 947/22 Turno NOCHE

Cant. de hojas entregadas (sin contar ésta) 4

El parcial se aprueba con 5 puntos. Entregar cada ejercicio en hoja separada. No se permite consultar ningún material durante el examen.

Corregió 66 ~~77~~ ~~88~~ Aprobado (10)

Ejercicio 1. 2 puntos

1. [1 punto]

   Completar en las siguientes especificaciones nombres adecuados para el problema a, los parámetros b y c, y las etiquetas x, y, z, u y w.

   `problema a` (in b: $seq\langle Char \times Char\rangle$, in c: $seq\langle Char\rangle$) : $seq\langle Char\rangle$ {
   - `requiere x:` $\{ (\forall i, j : \mathbb{Z})((0 \leq i < |b| \land 0 \leq j < |b| \land i \neq j) \rightarrow b[i]_0 \neq b[j]_0\}$
   - `requiere y:` $\{ (\forall i, j : \mathbb{Z})((0 \leq i < |b| \land 0 \leq j < |b| \land i \neq j) \rightarrow b[i]_1 \neq b[j]_1\}$
   - `requiere z:` $\{ (\forall i : \mathbb{Z})(0 \leq i < |c| \rightarrow (\exists j : \mathbb{Z})(0 \leq j < |b| \land b[j]_0 = c[i]))\}$
   - `asegura u:` $\{ |resultado| = |c| \}$
   - `asegura w:` $\{ (\forall i : \mathbb{Z})(0 \leq i < |c| \rightarrow (\exists j : \mathbb{Z})(0 \leq j < |b| \land b[j]_0 = c[i] \land b[j]_1 = resultado[i])) \}$

   }

2. [1 punto]

   Especificar el siguiente problema (se puede especificar de manera formal o semi-formal):

   Dados los inputs b: $seq\langle Char \times Char\rangle$, m: $seq\langle seq\langle Char\rangle\rangle$ y n: $seq\langle seq\langle Char\rangle\rangle$, retornar verdadero si n es igual al resultado de aplicar el `problema a` (del punto 1.1) a cada elemento de la secuencia m.

Ejercicio 2. 4 puntos

1. [2 puntos]

   Programar en Haskell una función que satisfaga la especificación del `problema a` del Ejercicio 1. Recordá escribir los tipos de los parámetros.

2. [2 puntos]

   Programar en Python una función que satisfaga la especificación del `problema a` del Ejercicio 1. Recordá escribir los tipos de los parámetros y variables que uses en tu implementación.

Ejercicio 3. 2 puntos

Sea la siguiente especificación del problema `aprobado` y una posible implementación en lenguaje imperativo:

`problema aprobado` (in notas: $seq(\mathbb{Z})$) : $\mathbb{Z}$ {
- `requiere:` $\{|notas| > 0\}$
- `requiere:` $\{(\forall i : \mathbb{Z})(0 \leq i < |notas| \rightarrow 0 \leq notas[i] \leq 10)\}$
- `asegura:` $\{result = 1 \leftrightarrow$ todos los elementos de *notas* son mayores o iguales a 4 y el promedio es mayor o igual a 7$\}$
- `asegura:` $\{result = 2 \leftrightarrow$ todos los elementos de *notas* son mayores o iguales a 4 y el promedio está entre 4 (inclusive) y 7$\}$
- `asegura:` $\{result = 3 \leftrightarrow$ alguno de los elementos de *notas* es menor a 4 o el promedio es menor a 4$\}$

}

```
def aprobado(notas: list[int]) -> int:
L1:   suma_notas: int = 0
L2:   i: int = 0
L3:   while i < len(notas):
L4:      if notas[i] < 4:
L5:         return 3
L6:      suma_notas = suma_notas + notas[i]
L7:      i = i + 1
L8:   if suma_notas >= 7 * len(notas):
L9:      return 2
L10:  else:
L11:     if suma_notas > 4 * len(notas):
L12:        return 2
L13:     else:
L14:        return 3
```

1. Dar el diagrama de control de flujo (control-flow graph) del programa `aprobado`.
2. Escribir un test suite que ejecute todas las líneas del programa `aprobado`.
3. Escribir un test suite que tenga un cubrimiento de **al menos** el 50 por ciento de decisiones ("branches") del programa.
4. Explicar cuál/es es/son el/los error/es en la implementación. ¿Los test suites de los puntos anteriores detectan algún defecto en la implementación? De no ser así, modificarlos para que lo hagan.

Ejercicio 4. 2 puntos

1. [1 punto] Suponga las siguientes dos especificaciones de los problemas p1 y p2:

```
problema p1(x:Int)=res:Int {
  requiere A;
  asegura C;
  }
```

```
problema p2(x:Int)=res:Int {
  requiere B;
  asegura C;
  }
```

   Si A es más fuerte que B, ¿Es cierto que todo algoritmo que satisface la especificación p1 también satisface la especificación p2? ¿Y al revés?, es decir, ¿Es cierto que todo algoritmo que satisface la especificación p2 también satisface la especificación p1? Justifique.

2. [1 punto] ¿Es posible que haya un test suite con 100 % de cubrimiento de nodos que todos los test pasen pero que igual el programa tenga un bug? Justifique.

NOTA ①

(1) a = convertirCódigo
b = claveDeConversión
c = código
} NOTA ②: entendí el problema como descriptivo en mensaje

{ X = Todos los primeros elementos son diferentes
{ Y = Todos los segundos elementos son diferentes
NOTA ②: estos dos requiere nos permiten asegurar que todo código tiene un solo "post-código" (convertido) posible y viceversa (todo código convertido es la conversión de un único código

Z = El código ~~puede contener~~ está conformado por caracteres de la clave
} NOTA ③: como $c \in Z[i][o]$ puede convertirse

V = el código y la conversión tienen la misma cantidad de elementos
V = Cada elemento de res corresponde a un pre-elemento del código, usando la clave.

(1) (2) problema verificarConversiones (in clave: seq<Char x Char>, in códigos: seq<seq<Char>>, in conversiones: seq<seq<Char>>):
Bool {
ACÁ ES MENOR, NO MONOTONA
requiere: {(∀i ∈ Z), (0 ≤ i ≤ |códigos|), mismaLongitud(codigos[i], conversiones[i])}
requiere: {mismaLongitudes(codigos, conversiones)}
asegura { res = True <-> (∀i ∈ Z), (0 ≤ i ≤ |codigos|), (mismaLista(conversión[i], convertirCódigo(código[i]))) }

(2)

pede mismaLongitud X, Y { X e Y tienen la misma cantidad de elementos -> True }

pede ~~mismoElemento~~ mismosElementos X, Y { todos los elementos de X e Y son iguales y están en el mismo orden }

Hoja 2

2 (1)

```haskell
convertirCódigo :: [(Char, Char)] -> [Char] -> [Char]
convertirCodigo clave [] = []
convertirCódigo clave (x:xs) = (equivalencia x) : convertirCódigo clave xs
```

falta paréntesis

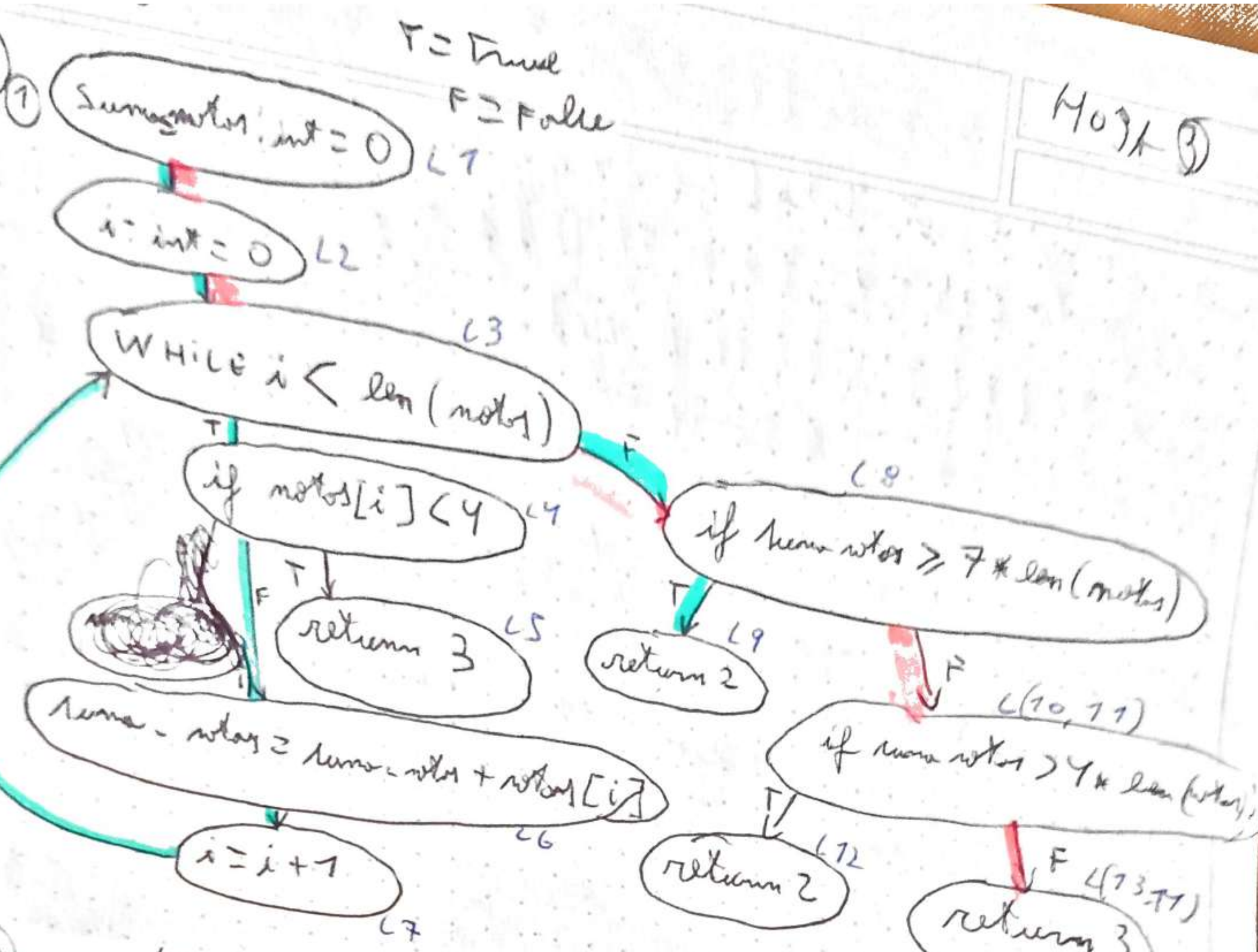
```haskell
equivalencia :: [(Char, Char)] -> Char -> Char
~~[illegible]~~
~~[illegible]~~
equivalencia ((pre, post):claves) x | x == pre = post
                                    | otherwise = equivalencia claves x
```

2 (2)

```python
def convertirCodigo(clave: list[tuple[Char, Char]],
        codigo : list[Char]) -> list[Char]:
    conversión: list[Char] = []
    for letra in codigo:
        for par in clave:
            if par[0] == letra:
                conversión.append(par[1])
    return conversión
```

3

③ ①



③ ② y ③ - ① test_aprobado:

redundante ↑ ~~Caso promoción { input: [7, 9, 8, 7] → output: 1 }~~

Caso promociónBorde { input: [7, 7, 7] → OUTPUT: 1 }

Caso aprobado { input [7, 4, 5, 6] → OUTPUT: 2 }

Caso aprobadoBorde { input [4, 4] → OUTPUT: 2 }

Caso reprobado { INPUT: [3, 2, 4] → OUTPUT: 3 }

Caso notaMala { INPUT [7, 8, 6, 2] → OUTPUT: 3 }

Caso listaVacía { INPUT [ ] → OUTPUT: 3 }

↑ lista vacía

(2) promociónBorde ejecuta [1,2,3,4,6,7,8,9], falta ejecutar 5, que lo cubre notaMala [1,2,3,4,5,6,7]. 11, 11 y 12 los cubre aprobado [1,2,3,4,6,7,8,10,11,12] y 13 y 14 los cubre ausente [1,2,3,8,10,11,13,14].

(3) ~~el [illegible], este caso con su [illegible]~~ marcados el cubrimiento de promocionBorde y ausente en el grafico, podemos ver que de 12 conexiones entre nodos abarcan Cuidado los branches no cuenta todas las conexiones, solo las "bifurcaciones"

(4) hay 2 errores en la implementación

- L9: abarca los casos que, segun especificación deberían retornar "1"

corrección: "return 1"

esto lo detecta el caso "PromociónBorde" del test suite.

- L11: abarca los casos [illegible] que según la especificación deberían retornar "2 ó 3", segun la suma de las notas,

corrección: "IF SUMA_NOTAS >= 4 * len(NOTAS):"

agregamos el caso en que el alumno se saco 4 en todo.

Esto lo detecta el caso "aprobadoBorde" del test suite,

(4) (1) Cuando decimos que (A) es más fuerte que (B), decimos que A → B es TAUTOLOGÍA (siempre V), es decir que cuando sea cumple A, SIEMPRE CUBRIMOS B,

A fines prácticos, (A) limita más los entradas que (B) para que [redacted] (P1), VA A SER UNA SOBRE ESPECIFICACIÓN DE (P2), YA QUE RECIBE MENOS VALORES POSIBLES.

Si tomamos a (P) como posible especificación de (P)

POR LO TANTO, UN ALGORITMO DISEÑADO PARA (P2), VA A CUBRIR Y SERVIR PARA (P1), SOLO QUE VA A PODER RECIBIR VALORES QUE NO SON NECESARIOS NI RELEVANTES PARA P1.

POR EL CONTRARIO, UN ALGORITMO DISEÑADO PARA (P1) NO VA A ABARCAR TODOS LOS CASOS PARA CUMPLIR LA ESPECIFICACIÓN DE (P2)

(2)

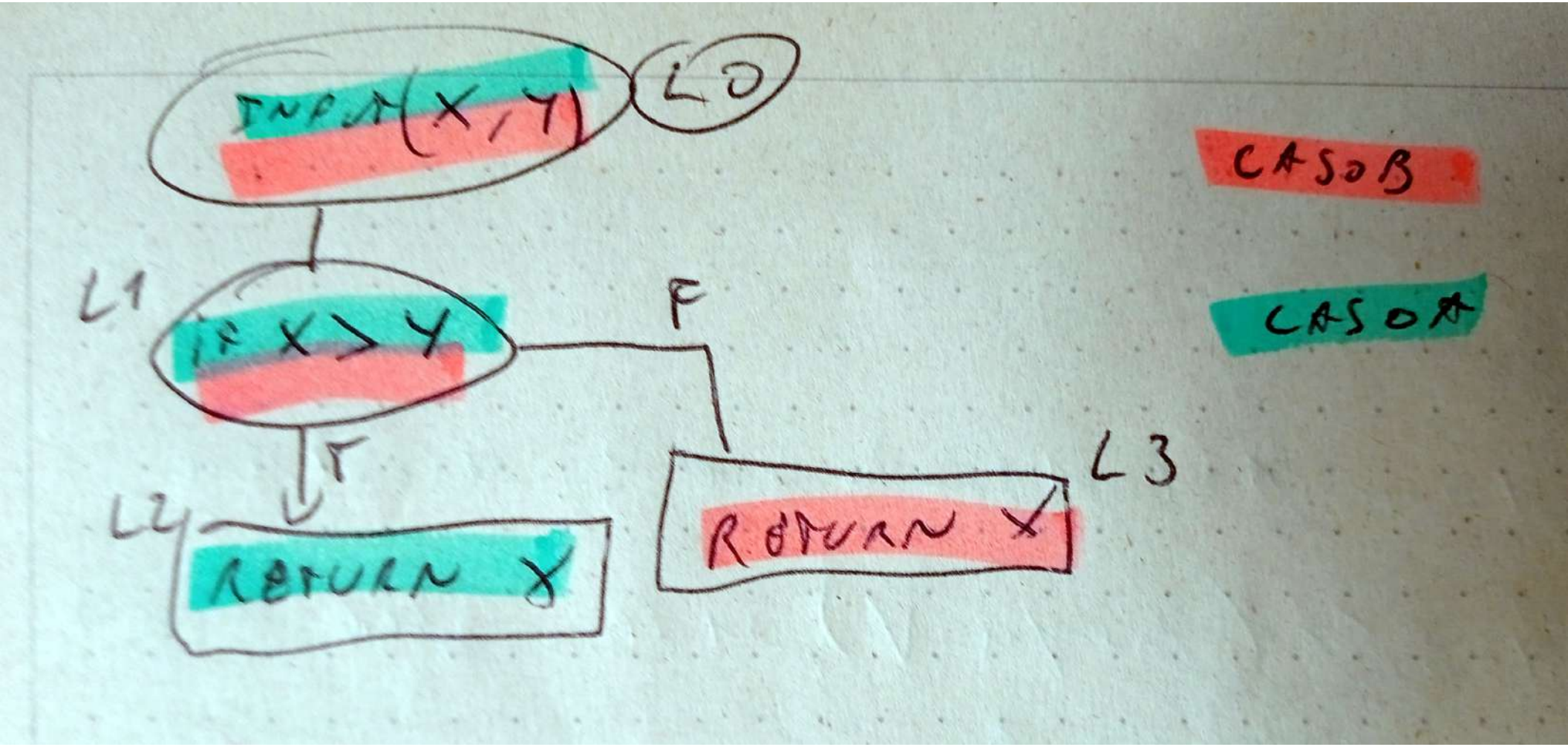
(4) (2) QUE UN TEST CUBRA TODOS LOS NODOS NO SIGNIFICA QUE CUBRA TODAS LAS BRANCHES LÓGICAS.

EJ

```
def MAX(X, Y) -> INT:
L1  IF X > Y
L2      RETURN X
L3  RETURN Y
```

TEST SUITE:	
A: IN[0,0] → OUT: 0	→ L1, L3
B: IN[1,0] → OUT: 1	→ L1, L2

AMBOS TESTS SE CUMPLEN, Y CUBRE TODOS LOS NODOS PERO EL PROGRAMA TIENE UN BUG: SI Y ES MAYOR, IGUAL DEVUELVE X!


INPUT (X, Y)
L0
L1
IF X > Y
F
T
L2
RETURN X
L3
RETURN Y
CASO B
CASO A