Taller 3 - Ingeniería de Software II

Ejercicio 1.

a.

```
(declare-const x Bool)
(declare-const y Bool)
(assert ( = (not(or x y)) (and (not x)(not y))))
(check-sat)
(get-model)

;OUTPUT:
sat
(
  (define-fun y () Bool
    false)
  (define-fun x () Bool
    false)
)
```

b.

```
(declare-const x Bool)
(declare-const y Bool)
(assert ( = (and x y) (not(or (not x)(not y)))))
(check-sat)
(get-model)

;OUTPUT:
sat
```

```
(
  (define-fun y () Bool
    false)
  (define-fun x () Bool
    false)
)
```

c.

```
(declare-const x Bool)
(declare-const y Bool)
(assert ( = (not(and x y)) (not(and (not x)(not y)))))
(check-sat)
(get-model)

;OUTPUT:
sat
(
  (define-fun y () Bool
    true)
  (define-fun x () Bool
    false)
)
```

Ejercicio 2.

a.

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ (* 3 x) (* 2 y)) 36))
(check-sat)
(get-model)
```

```
;OUTPUT:
sat
(model
  (define-fun y () Int
    0)
  (define-fun x () Int
    12)
)
```

b.

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ (* 5 x) (* 4 y)) 64))
(check-sat)
(get-model)

;OUTPUT:
sat
(model
  (define-fun y () Int
    1)
  (define-fun x () Int
    12)
)
```

c.

```
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (* x y) 64))
(check-sat)
(get-model)

;OUTPUT:
```

```
sat
(
  (define-fun x () Int
    64)
  (define-fun y () Int
    1)
)
```

Ejercicio 5. a.

Ejercicio 5				
#ITERACIÓN	INPUT CONCRETO	PATH-CONDITION	ESPECIFICACIÓN PARA Z3	POSIBLE RESULTADO
1	a = 0, b = 0, c = 0	C1	!C1	a = 1, b = 1, c = 1
2	a = 1, b = 1, c = 1	!C1 && !C2 && C3	!C1 && !C2 && !C3	a = 2, b = 3, c = 4
3	a = 2, b = 3, c = 4	!C1 && !C2 && !C3 && !C4	!C1 && !C2 && !C3 && C4	a = 2, b = 2, c = 3
4	a = 2, b = 2, c = 3	!C1 && !C2 && !C3 && C4	!C1 && !C2	a = 1, b = 1 , c = 2
5	a = 1, b = 1, c = 2	!C! && C2	FIN	

Ejercicio 6

a.

Ejercicio 6				
#ITERACIÓN	INPUT CONCRETO	PATH-CONDITION	ESPECIFICACIÓN PARA Z3	POSIBLE RESULTADO
1	k = 0	C1_0 && !C2_0 && C1_1 && !C2_1 && C1_2 && !C2_2 && !C1_3	C1_0 && !C2_0 && C1_1 && !C2_1 && C1_2 && C2_2	k = -3.0
2	k = -3.0	C1_0 && !C2_0 && C1_1 && !C2_1 && C1_2 && C2_2 && !C1_3	C1_0 && !C2_0 && C1_1 && !C2_1 && !C1_2	UNSAT
			C1_0 && !C2_0 && C1_1 && C2_1	k = -1.0
3	k = -1.0	C1_0 && !C2_0 && C1_1 && C2_1 && C1_2 && !C2_2 && !C1_3	C1_0 && !C2_0 && C1_1 && C2_1 && C1_2 && C2_2	UNSAT
			C1_0 && !C2_0 && C1_1 && C2_1 && !C1_2	UNSAT
			C1_0 && !C2_0 && !C1_1	UNSAT
			C1_0 && C2_0	k = -5.0
4	k = -5.0	C1_0 && C2_0 && C1_1 && !C2_1 && C1_2 && !C2_2 && !C1_3	C1_0 && C2_0 && C1_1 && !C2_1 && C1_2 && C2_2	UNSAT
			C1_0 && C2_0 && C1_1 && !C2_1 && !C1_2	UNSAT
			C1_0 && C2_0 && C1_1 && C2_1	UNSAT
			C1_0 && C2_0 && !C1_1	UNSAT
			!C1_0	UNSAT
FIN				

b. La idea del algoritmo de ejecución simbólica dinámica es lograr cubrir todos los branches. Mirando el árbol podemos ver que efectivamente se cubren todas las deciciones (sin considerar todo lo que sea UNSAT, puesto que no

existe input que haga ejecutar esas ramas). Luego, el branch coverage es del 100%.

Ejercicio 7. a.

Ejercicio 7				
#ITERACIÓN	INPUT CONCRETO	PATH-CONDITION	ESPECIFICACIÓN PARA Z3	POSIBLE RESULTADO
1	n = 0	!C1_0	C1_0	n = 1
2	n = 1	C1_0 && !C1_1	C1_0 && C1_1	n = 2
3	n = 2	C1_0 && C1_1&& !C1_2	FIN	