Ingeniería de Software II - Taller #4

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)
```

Z3 devolvió `sat`, sin mostrar ningún modelo.

b.

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

Z3 devolvió `sat`, sin mostrar ningún modelo.

c.

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

Z3 devolvió `sat`, con modelo `y = true, x = false`.

Ejercicio 2

a. $3x + 2y = 36$

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

El resultado fue `sat`, y el modelo fue `y = 0, x = 12`.

b. $5x + 4y = 64$

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

El resultado fue `sat`, y el modelo fue `y = 1, x = 12`.

c. x * y = 64

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

El resultado fue `sat`, y el modelo fue `y = 1, x = 64`.

Ejercicio 3

Una posible especificación para Z3 que calcule las expresiones pedidas sería:

```
(declare-const a1 Int)
(declare-const a2 Int)
(declare-const a3 Int)
(assert (= a1 (mod 16 2)))
(assert (= a2 (div 16 4)))
(assert (= a3 (rem 16 5)))
(check-sat)
(get-model)
```

La salida de Z3 con esta especificación fue:

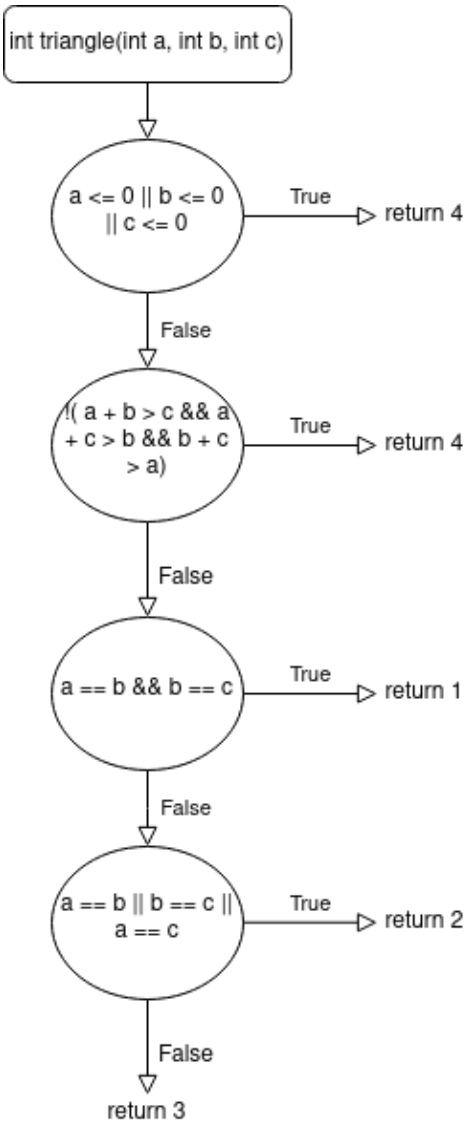
```
sat
(model
  (define-fun a3 () Int
    1)
  (define-fun a2 () Int
    4)
  (define-fun a1 () Int
    0)
)
```

, es decir que un modelo que verifica las fórmulas es con `a1 = 0, a2 = 4, y a3 = 1`.

Ejercicio 4

a.

El árbol de cómputo de la función `triangle` es el siguiente:



b.

Randoop generó 67 tests en 2 segundos, los cuales fueron más que suficientes para cubrir todas las líneas del método triangle. Esto podría deberse a que es un método pequeño con branches que son tomadas en muchos posibles casos, y Randoop tuvo la suerte de elegir bien los casos de test.

Para hacer una comparación entre Randoop y DSE, podríamos decir que Randoop probablemente produzca casos de test más rápidamente que DSE, por el hecho de que DSE necesita un demostrador de teoremas para funcionar. Por otro lado, cada caso que produzca DSE analizará un branch diferente del programa, y por lo tanto cada caso será más significativo y el testsuite será más reducido.

c.

Pondremos nombres a las condiciones de los `if` que componen el método `triangle` para lograr mayor síntesis.

```
alguno_no_positivo := a0 <= 0 || b0 <= 0 || c0 <= 0
```

```
lado_muy_largo := !(a0 + b0 > c0 && a0 + c0 > b0 && b0 + c0 > a0)
equilátero := a0 == b0 && b0 == c0
isósceles := a0 == b0 || b0 == c0 || a0 == c0
```

En vez de hacer una tabla, muestro las iteraciones en una lista, porque las especificaciones para Z3 llegan a ser muy largas y no entrarían en una celda.

Iteraciones:

1.

- **Input concreto:** `a=0`, `b=0`, `c=0`
- **Condición de Ruta:** `alguno_no_positivo`
- **Especificación para Z3:**

```
(assert (not (or (or (<= a0 0) (<= b0 0)) (<= c0 0))))
```

- **Resultado Z3:** `a0=1`, `b0=1`, `c0=1`

2.

- **Input concreto:** `a=1`, `b=1`, `c=1`
- **Condición de Ruta:** `!alguno_no_positivo && !lado_muy_largo && equilátero`
- **Especificación para Z3:**

```
(assert (not (or (or (<= a0 0) (<= b0 0)) (<= c0 0))))
(assert (and (and (> (+ a0 b0) c0) (> (+ a0 c0) b0)) (> (+ b0 c0) a0)))
(assert (not (and (= a0 b0) (= b c0))))
```

- **Resultado Z3:** `a0=2`, `b0=3`, `c0=4`

3.

- **Input concreto:** `a=2`, `b=3`, `c=4`
- **Condición de Ruta:** `!alguno_no_positivo && !lado_muy_largo && !equilátero && !isósceles`
- **Especificación para Z3:**

```
(assert (not (or (or (<= a0 0) (<= b0 0)) (<= c0 0))))
(assert (and (and (> (+ a0 b0) c0) (> (+ a0 c0) b0)) (> (+ b0 c0) a0)))
(assert (not (and (= a0 b0) (= b0 c0))))
(assert (or (or (= a0 b0) (= b0 c0)) (= a0 c0)))
```

- **Resultado Z3:** `a0=2`, `b0=1`, `c0=2`

4.

- **Input concreto:** `a=2`, `b=1`, `c=2`
- **Condición de Ruta:** `!alguno_no_positivo && !lado_muy_largo && !equilátero && isósceles`
- **Especificación para Z3:**

```
(assert (not (or (or (<= a0 0) (<= b0 0)) (<= c0 0))))
(assert (not (and (and (> (+ a0 b0) c0) (> (+ a0 c0) b0)) (> (+ b0 c0) a0))))
```

- **Resultado Z3:** `a0=1`, `b0=1`, `c0=2`

5.

- **Input concreto:** `a=1`, `b=1`, `c=2`
- **Condición de Ruta:** `!alguno_no_positivo && lado_muy_largo`
- **Especificación para Z3:** Ninguna
- **Resultado Z3:** Ninguno

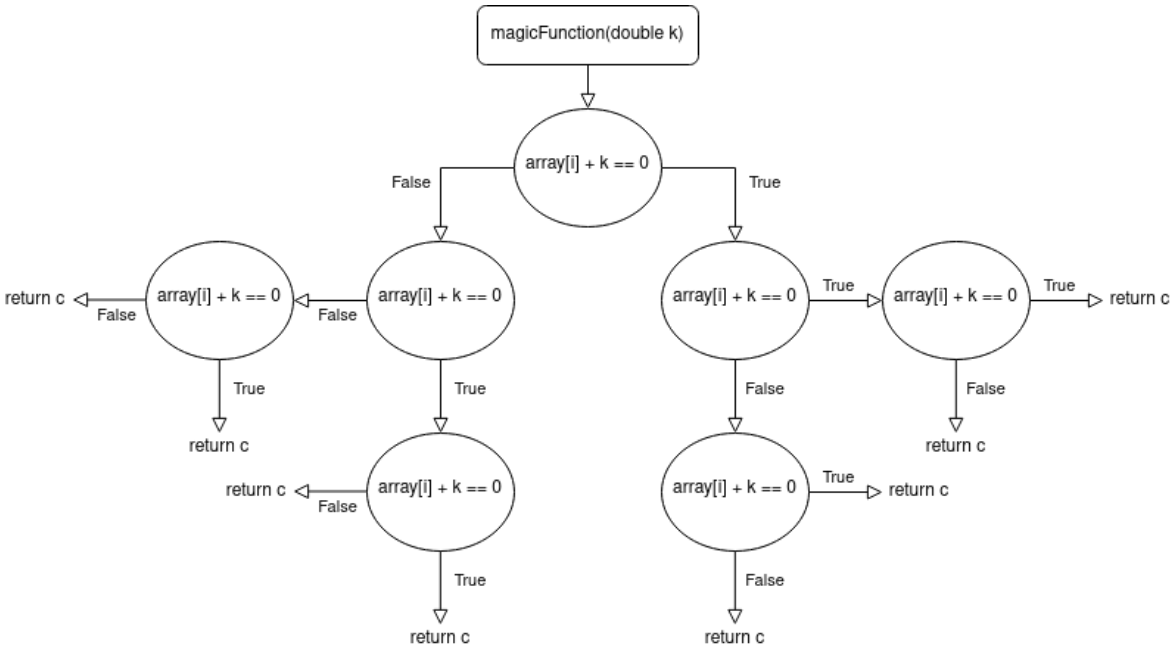
d.

Este test suite alcanzará un line coverage del 100%, ya que en ningún momento hubo un resultado `unsat` o `unknown` de parte de Z3, lo cual garantiza que, como el algoritmo finalizó, cubrió todas las ramas del árbol de cómputo.

Ejercicio 5

a.

El árbol de cómputo de la función `magicFunction` es el siguiente:



b.

De nuevo, la cobertura fue del 100%. Esto nos está indicando que Randoop es bastante bueno para generar tests, por lo menos para estas funciones pequeñas.

c.

Vamos a llamar `Q(i)` a la condición `array[i] + k0 == 0`.

Iteraciones:

1.

- **Input Concreto:** `k=0.0`

- **Condición de Ruta:** `!Q(0) && !Q(1) && !Q(2)`
- **Especificación para Z3:**

```
(assert (not (= (+ 5.0 k0) 0)))
(assert (not (= (+ 1.0 k0) 0)))
(assert (= (+ 3.0 k0) 0))
```

- **Resultado Z3:** `k0=-3.0`

2.

- **Input Concreto:** `k=-3.0`
- **Condición de Ruta:** `!Q(0) && !Q(1) && Q(2)`
- **Especificación para Z3:**

```
(assert (not (= (+ 5.0 k0) 0)))
(assert (= (+ 1.0 k0) 0))
```

- **Resultado Z3:** `k0=-1.0`

3.

- **Input Concreto:** `k=-1.0`
- **Condición de Ruta:** `!Q(0) && Q(1) && !Q(2)`
- **Especificación para Z3:**

```
(assert (not (= (+ 5.0 k0) 0)))
(assert (= (+ 1.0 k0) 0))
(assert (= (+ 3.0 k0) 0))
```

- **Resultado Z3:** `unsat`
- **Especificación para Z3:**

```
(assert (= (+ 5.0 k0) 0))
(assert (not (= (+ 1.0 k0) 0)))
(assert (not (= (+ 3.0 k0) 0)))
```

- **Resultado Z3:** `k0=-5.0`

4.

- **Input Concreto:** `k=-5.0`
- **Condición de Ruta:** `Q(0) && !Q(1) && !Q(2)`
- **Especificación para Z3:**

```
(assert (= (+ 5.0 k0) 0))
(assert (not (= (+ 1.0 k0) 0)))
(assert (= (+ 3.0 k0) 0))
```

- **Resultado Z3:** `unsat`
- **Especificación para Z3:**

```
(assert (= (+ 5.0 k0) 0))
(assert (= (+ 1.0 k0) 0))
```

- **Resultado Z3:** `unsat`

d.

El line coverage es de nuevo 100%. En este caso, para tener un line coverage de 100% sólo es necesario que haya por lo menos un test para el cual la condición del `if` sea verdadera, lo cual se cumple con el test generado en la segunda iteración.