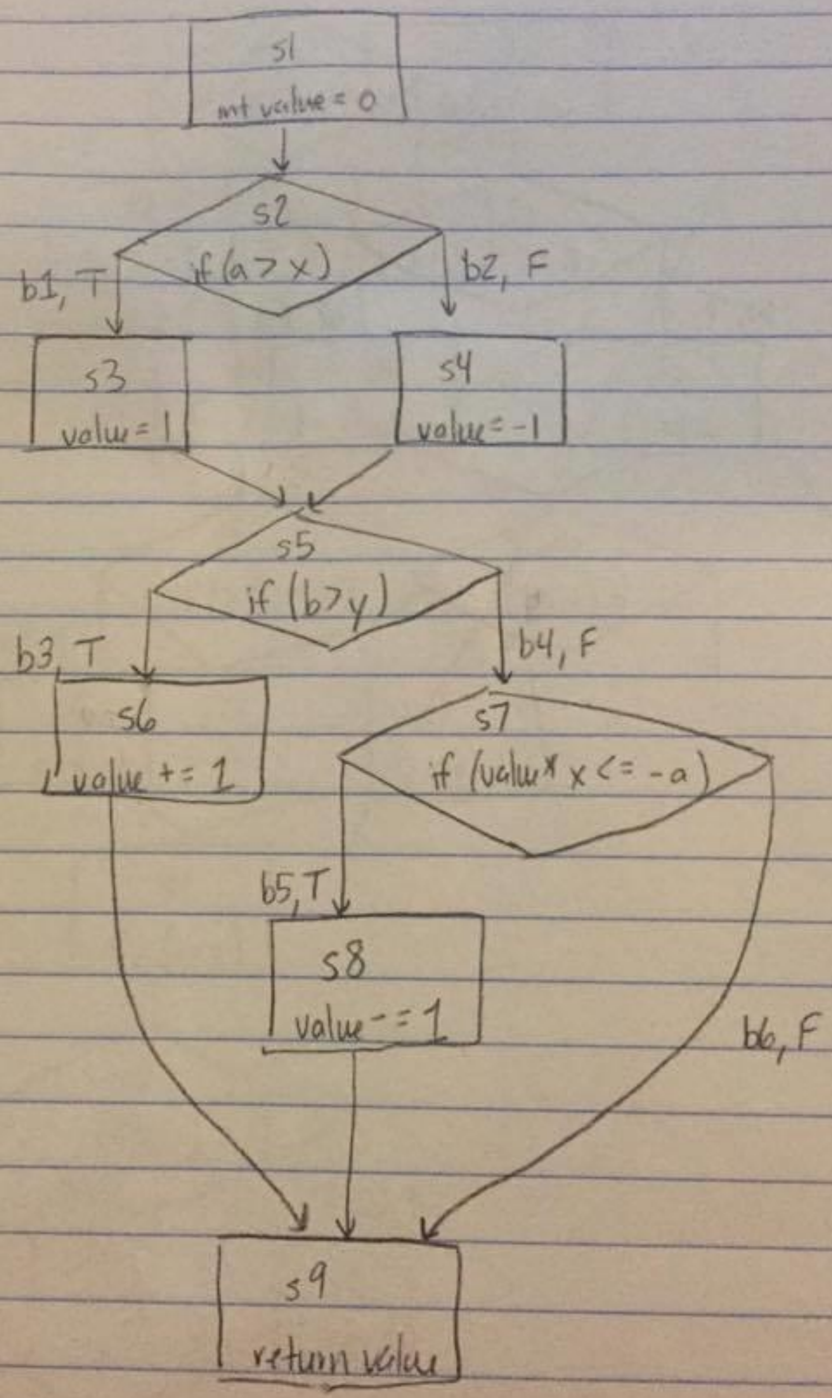
1 a)


s1
int value = 0
s2
if (a > x)
b1, T
b2, F
s3
value = 1
s4
value = -1
s5
if (b > y)
b3, T
b4, F
s6
value += 1
s7
if (value * x <= -a)
b5, T
b6, F
s8
value -= 1
s9
return value

```
/* 1b - We want our tests to have full statement coverage, which means that
 * s1, s2, ..., s9 have to be executed
 */

/* This test should enter both if statements and should
 * cover s1, s2, s3, s5, s6, s9.
 * In order for it to enter these statements, a>x and b>y
 * The return value will be 1 + 1 = 2
 * This test covers the path b1,b3
 */
@Test
public void test1() {
   assertEquals(2, coverage(1, 1, 0, 0));
}

/* This test should enter the else then else-if statements and should
 * cover s1, s2, s4, s5, s7, s8, s9
 * In order for it to eter these statements a<=x, b<=y, and
 * value (which should be -1 based on these assumptions) * x <= -a
 * The return value will be -1 - 1 = -2
 * This path covers the path b2,b4,b5
 */
@Test
public void test2() {
        assertEquals(-2, coverage(0, 0, 0, 0));
}

/* 1c - We want our tests to achieve full branch coverage, which means
 * that, in our diagram, b1, ..., b6 should all be executed.
 * Looking at the above tests, we see that the following branches have already
 * been tested: b1, b2, b3, b4, b5.
 * This means that we need to test b6: the case where the second if/else-if staement
 * isn't satisfied. Therefore, we need a test where b<=y and value*x > -a.
 * For this test, we will also use a>x so value will be 1.
 * This path covers the path b1,b4,b6
 */
@Test
public void test3() {
        assertEquals(1, coverage(1, 0, 0, 0));
}

/* 1d - We want our tests to achieve full feasible path coverage. There are a
 * couple of places where our code branches off. Our code can take b1 or b2.
 * In addition, it can take b3 or [b4,b5] or [b4,b6].
 * This means that all possible paths include:
 *              1. b1,b3 - covered by test1
```

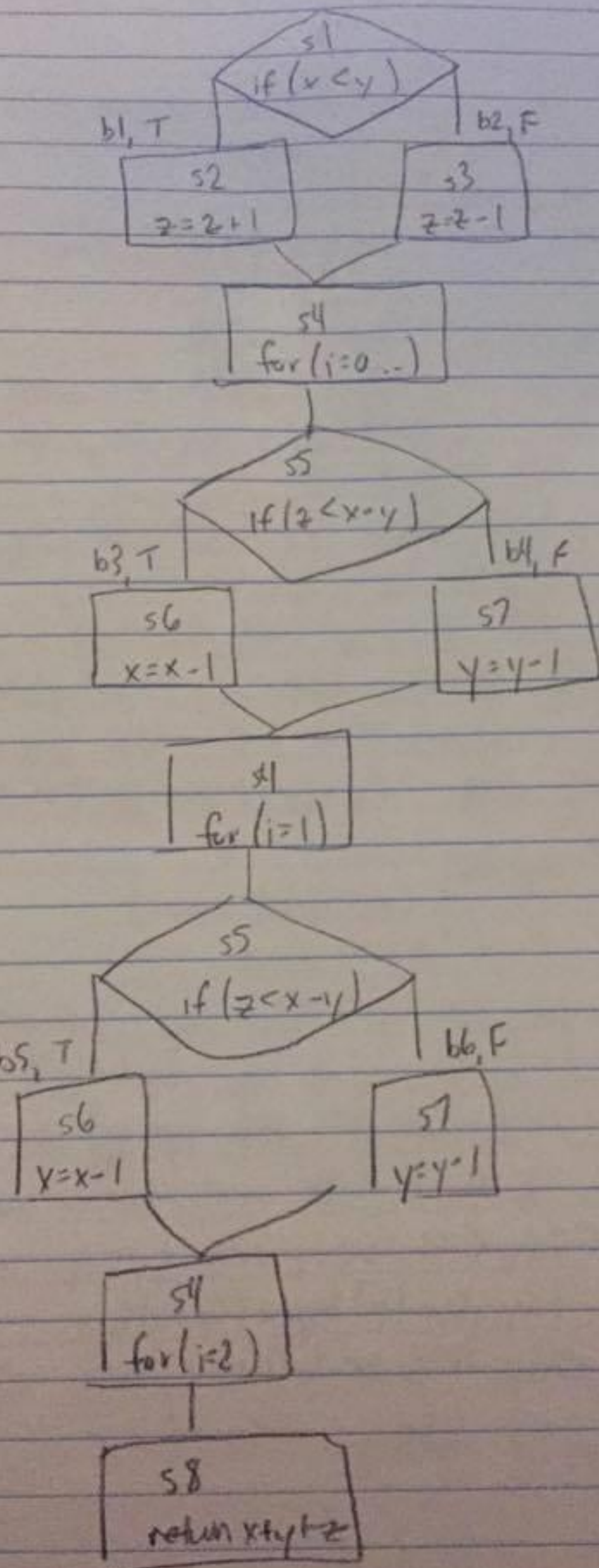
```
 *                2. b1,b4,b5
 *                3. b1,b4,b6 - covered by test3
 *                4. b2,b3
 *                5. b2,b4,b5 - covered by test2
 *                6. b2,b4,b6
 * We will then attempt to create tests to cover the remaining paths
 * starting from the top of the list and moving to the bottom.
 */

/* We need a test that covers b1,b4,b5, which means that we need
 * a>x (implies value=1) as well as (value*x <= -a) and b<=y. This path makes
 * value=1 so (value*x) will simplify to just x. The second condition then
 * becomes x<=-a
 */
@Test
public void test4() {
        assertEquals(0, coverage(-1, 0, -2, 1));
}

/* We need atest that covers b2,b3, which means that we need
 * a<=x and b>y.
 */
@Test
public void test5() {
        assertEquals(0, coverage(0, 1, 1, 0));
}

/* 1d and 1e
 * We need a test that covers b2,b4,b6, which means that we need
 * a<=x, b<=y, (value*x > -a). From these assumptions, we see that
 * value at the point of the second condition will be equal to -1.
 * Therefore the condition will become (-x > -a). Multiplying the
 * inequality by -1 yields (x<a). However, the first condition states
 * that (x>=a). We see there is no way to fulfill both of these conditions
 * so this is an infeasible path.
```

2.


s1
if (x<y)
b1, T
b2, F
s2
z=z+1
s3
z=z-1
s4
for (i=0 ..)
s5
if (z<x-y)
b3, T
b4, F
s6
x=x-1
s7
y=y-1
s4
for (i=1)
s5
if (z<x-y)
b5, T
b6, F
s6
x=x-1
s7
y=y-1
s4
for (i=2)
s8
return x+y+z

```
public static int compute(int x, int y, int z) {
        if (x < y) {                                    // S1, B1
                z = z + 1;                                      // S2
        }
        else {                                                  // B2
                z = z - 1;                                      // S3
        }

        // Loop unrolled (repeats twice)
        /* Enter loop */                                // S4
        if (z < x - y) {                        // S5, B3
                x = x - 1;                                      // S6
        }
        else {                                                  // B4
                y = y - 1;                                      // S7
        }

        /* Next iteration */                    // S4
        if (z < x - y) {                        // S5, B5
                x = x - 1;                                      // S6
        }
        else {                                                  // B6
                y = y - 1;                                      // S7
        }

        return x + y + z;                               // S8
}

/* First we will come up with the symbolic execution after S3 executes.
 * To do this, we will come up with the effect of S2 executing which becomes
 * (x<y) AND (z=z+1). Next, the effect of S3 executing is (x>=y) AND (z=z-1).
 * Therefore, the symbolic execution is simply the OR of these two possibilieis:
 * ((x<y) AND (z=z+1)) OR ((x>=y) AND (z=z-1)).
 *
 * Next we need the symbolic execution of each iteration of the loop.
 * We do the same process as above. The effect of S6 executing becomes
 * (z<x-y) AND (x=x-1). The effect of S7 executing becomes (z>=x-y) AND (y=y-1).
 * Therefore, the symbolic execution becomes the OR of these:
 * ((z<x-y) AND (x=x-1)) OR ((z>=x-y) AND (y=y-1))
 *
 * The next iteration will have the same symbolic execution.
 *
 * We should note, however, that later branch conditions are affected
 * by prior assignments.
 */
```

```
/* Next, we need to come up with different paths of execution.
 * Each path of exeuction can take (B1 OR B2) AND (B3 OR B4) AND (B5 OR B6).
 * We will note here that S4 will be always be executed at the beginning
 * of a loop iteration and S8 will always be executed at the end of the function.
 * S1 is always executed because the expression needs to be evaluated.
 * If B1 is taken, then S2 executes. Otherwise B2 is taken and S3 executes.
 * This logic follows for the other branches.
 * S5 is always executed at the beginning of an iteration.
 * If B3 or B5 is taken, then S6 is executed.
 * Otherwise, B4 or B6 is taken and S7 is executed.
 *
 * Next we will list the possible branches that can be taken:
 *              1. B1,B3,B5 (TTT)
 *              2. B1,B3,B6 (TTF)
 *              3. B1,B4,B5 (TFT)
 *              4. B1,B4,B6 (TFF)
 *              5. B2,B3,B5 (FTT)
 *              6. B2,B3,B6 (FTF)
 *              7. B2,B4,B5 (FFT)
 *              8. B2,B4,B6 (FFF)
 *
 * Associating each of the branches with the exeuctions we get the
 * following statement executions:
 *              1. S1,S2,S4,S5,S6,S4,S5,S6,S4,S8
 *              2. S1,S2,S4,S5,S6,S4,S5,S7,S4,S8
 *              3. S1,S2,S4,S5,S7,S4,S5,S6,S4,S8
 *              4. S1,S2,S4,S5,S7,S4,S5,S7,S4,S8
 *              5. S1,S3,S4,S5,S6,S4,S5,S6,S4,S8
 *              6. S1,S3,S4,S5,S6,S4,S5,S7,S4,S8
 *              7. S1,S3,S4,S5,S7,S4,S5,S6,S4,S8
 *              8. S1,S4,S4,S5,S7,S4,S5,S7,S4,S8
 *        |     |     |
 *
 * Now let's look at the te test cases given in Part A.
 *
 * 1.   x = 1 AND y = 2 AND z = -4
 *      First branch: (x<y) -> (1<2) TRUE
 *              z incremented to -3 -> x=1, y=2, z=-3
 *              Second branch: (z<x-y) -> (-3<1-2=-1) TRUE
 *              x decremented to 0 -> x=0, y=2, z=-3
 *              Third branch: (z<x-y) -> (-3<0-2=-2) TRUE
 *              x decremented to -1 -> x=-1, y=2, z=-3
 *              Return -1 + 2 + -3
 *              TTT -> Path 1 executed.
 *
 * 2.   x = 2 AND y = 3 AND z = -1
```

```
*              First branch: (x<y) -> (2<3) TRUE
*              z incremented to 0 -> x=2, y=3, z=0
*              Second branch: (z<x-y) -> (0<2-3=-1) FALSE
*              y decremented to 2 -> x=2, y=2, z=0
*              Third branch: (z<x-y) -> (0<2-2=0) FALSE
*              y decremented to 1 -> x=2, y=1, z=0
*              Return 2 + 1 + 0
*              TFF -> Path 4 executed
*
* 3.    x = 4 AND y = 6 AND z = -3
*              First branch: (x<y) -> (4<6) TRUE
*              z incremented to -2 -> x=4, y=6, z=-2
*              Second branch: (z<x-y) -> (-2<4-6=-2) FALSE
*              y decremented to 5 -> x=4, y=5, z=-2
*              Third branch: (z<x-y) -> (-2<4-5=-1) TRUE
*              x decremented to 3 -> x=3, y=5, z=-2
*              Return 3 + 5 + -2
*              TFT -> Path 3 executed
*
* 4.    x = 3 AND y = 1 AND z = 2
*              First branch: (x<y) -> (3<1) FALSE
*              z decremented to 1 -> x=3, y=1, z=1
*              Second branch (z<x-y) -> (1<3-1=2) TRUE
*              x decremented to 2 -> x=2, y=1, z=1
*              Third branch (z<x-y) -> (1<2-1=1) FALSE
*              y decremented to 0 -> x=2, y=0, z=1
*              Return 2 + 0 + 1
*              FTF -> Path 6 executed
*
* 5.    x = 2 AND y = 5 AND z = -5
*              First branch: (x<y) -> (2<5) TRUE
*              z incremented to -4 -> x=2, y=5, z=-4
*              Second branch: (z<x-y) -> (-4<2-5=3) TRUE
*              x decremented to 1 -> x=1, y=5, z=-4
*              Third branch: (z<x-y) -> (-4<1-5=-4) FALSE
*              y decremented to 4 -> x=1, y=4, z=-4
*              Return 1 + 4 + -4
*              TTF -> Path 2 executed
*
* 6.    x = 3 AND y = 2 AND z = 2
*              First branch: (x<y) -> (3<2) FALSE
*              z decremented to 1 -> x=3, y=2, z=1
*              Second branch: (z<x-y) -> (1<3-2=1) FALSE
*              y decremented to 1 -> x=3, y=1, z=1
*              Third branch: (z<x-y) -> (1<3-1=2) TRUE
*              x decremented -> x=2, y=1, z=1
```

```
*               Return 2 + 1 + 1
*               FFT -> Path 7 executed
*
* 7.    x = 1 AND y = 2 AND z = -2
*               First branch: (x<y) -> (1<2) TRUE
*               z incremented to -1 -> x=1, y=2, z=-1
*               Second branch: (z<x-y) -> (-1<1-2=-1) FALSE
*               y decremented to 1 -> x=1, y=1, z=-1
*               Third branch: (z<x-y) -> (-1<1-1=0) TRUE
*               x decremented to 0 -> x=0, y=1, z=-1
*               Return 0 + 1 + -1
*               TFT -> Path 3 executed
*
* 8.    x = 1 AND y = 2 AND z = -3
*               First branch: (x<y) -> (1<2) TRUE
*               z incremented to -2 -> x=1, y=2, z=-2
*               Second branch: (z<x-y) -> (-2<1-2=-1) TRUE
*               x decremented to 0 -> x=0, y=2, z=-2
*               Third branch: (z<x-y) -> (-2<0-2=-2) FALSE
*               y decremented to 1 -> x=0, y=1, z=-2
*               Return 0 + 1 + -2
*               TTF -> Path 2 executed
*
* 9.    x = 1 AND y = 2 AND z = 0
*               First branch: (x<y) -> (1<2) TRUE
*               z incremented to 1 -> x=1, y=2, z=1
*               Second branch: (z<x-y) -> (1<1-2=-1) FALSE
*               y decremented to 1 -> x=1, y=1, z=1
*               Third branch: (z<x-y) -> (1<1-1=0) FALSE
*               y decremented to 0 -> x=1, y=0, z=1
*               Return 1 + 0 + 1
*               TFF -> Path 4 executed
*
* 10.   x = 2 AND y = 1 AND z = 1
*               First branch (x<y) -> (2<1) FALSE
*               z decremented to 0 -> x=2, y=1, z=0
*               Second branch (z<x-y) -> (0<2-1=1) TRUE
*               x decremented to 1 -> x=1, y=1, z=0
*               Third branch (z<x-y) -> (0<1-1=0) FALSE
*               y decremented to 0 -> x=1, y=0, z=0
*               Return 1 + 0 + 0
*               FTF -> Path 6 executed
*
* Paths 5 and 8 are not executed by the test cases.
*
* Path 5 (FTT) -        (x>=y) AND (z=z-1) AND (z<x-y) AND (x=x-1)
```

```
*                     AND (z<x-y)  AND (x=x-1)  =>
*                                               (x>=y)  AND (z-1<x-y)  AND (z-1<(x-1)-y)  =>
*                                               (x>=y)  AND (z-1<x-y)  AND (z<x-y)  =>
*                                               (x>=y)  AND (z<x-y)  => C
*
* Path 8 (FFF) -         (x>=y)  AND (z=z-1)  AND (z>=x-y)  AND (y=y-1)
*                                               AND (z>=x-y)  AND (y=y-1)  =>
*                                               (x>=y)  AND (z-1>=x-y)  AND (z-1>=x-(y-1))  =>
*                                               (x>=y)  AND (z-1>=x-y)  AND (z-2>=x-y)  =>
*                                               (x>=y)  AND (z>=x-y+2)  => B
```

```
private int getSomeElement(MyObject o, int x) {
    int n;

                        wp(if guard then if-block else else-block, P)
    int[] s;            ≡ (guard AND wp(if-block, P)) OR (NOT guard AND wp(else-block, P))
                        ≡ (guard AND Q6) OR (NOT guard AND Q4)
    if (o != null) {    ≡ (o!=null AND o.arr!=null AND o.f >= 0 AND o.f < o.arr.length())
                          OR (o==null AND x >= 1/3 AND x < 3/4)

        n = o.f;  -> wp(n = o.f, Q5) ≡ (o.f >= 0 AND o.f < o.arr.length()) } Q6
                                      o.arr != null

        s = o.arr;
                  wp(s = o.arr, Q1) ≡ (n >= 0 AND n < o.arr.length()) } Q5
    }

    else {    wp(x = 2*x+1, Q3) ≡ (2x+1) >= 5/3 AND (2x+1) < 5/2
        x = 2*x+1;          ≡ x >= 1/3 AND x < 3/4            } Q4

        n = 3*x - 5;  wp(n = 3*x-5, Q2)   x >= 5/3
                        ≡ (x >= 0 AND (3*x-5) >= 0 AND 3*x-5 < x)
        s = new int[x];     ≡ x >= 5/3 AND x < 5/2            } Q3

    }       wp(s = new int[x], Q1)
                ≡ x >= 0 AND Q1 } Q2
    return s[n];   wp(s[n], true)
                ≡ (n >= 0 AND n < s.length() AND true)
                   \________________________/
                              Q1
}
```

```
class MyObject {
    int f;
    int [] arr;
}
```

a) From above, o.arr != null AND o.f >= 0 AND o.f < o.arr.length (A)

b) From above $x>=0$ AND $(3*x-5)>=0$ AND $\underbrace{(3*x-5)<x}_{2x-5<0}$

Substitute $x \rightarrow 2x+1$

↳ $(2*x+1)>=0$ AND

$(3*(2*x+1)-5>=0 \Rightarrow 6*x-2>=0$

$(2*(2*x+1)-5)>0 \Rightarrow 4*x-3<0$ (C)

c) Weakest precondition: $x>=0$ AND $n>=0$ AND $n<x$

✓ A $n>0$ AND $n<x$ AND $x>=0$

↳ not weakest b/c n=0 cases

X B $n<x$ AND $x>=0$

↳ not precondition, e.g. $x=1$ $n=-1$

X C $n>=0$ AND $n<x$ AND $x>=0$

↳ this is the weakest precondition

? D $n>=0$ AND $n<x$ AND $x>0$

↳ this is not weakest b/c $x=0$ (although after simplifying some of the logic, we see this is not a valid case, so it may be considered a weakest precondition in that sense)

✓ E $n>1$ AND $n<x$ AND $x>=0$

↳ not weakest for when $n=0,1$

d) (o != null AND o.arr != null AND o.f >= 0 AND o.f < o.arr.length) OR (o == null AND $(2*x+1)>=0$ AND $(6*x-2)>=0$ AND $(4*x-3)<0$)

(B)

4 a) Show invariant held initially P = {x1 > 0 AND x2 > 0}

wp(T, J) where T = {y1 := x1; y2 := x2}

J = {y1 > 0 AND y2 > 0 AND gcd(x1, x2) = gcd(y1, y2)}

$wp(T, J) = \{\underbrace{x1 > 0 \text{ AND } x2 > 0}_{\text{Precondition for problem provides this (P)}} \text{ AND } \underbrace{gcd(x1, x2) = gcd(x1, x2)}_{\text{This is always true}}\}$

This is always true

Precondition for problem provides this (P) ∴ P ⇒ wp(T, J) ✓

b) Show invariant maintained

wp(S, J) where S = {if (y1 > y2) then y1 := y1 − y2 else y2 := y2 − y1}

J = {y1 > 0 AND y2 > 0 AND gcd(x1, x2) = gcd(y1, y2)}

wp(S, J) ≡ {(y1 > y2 AND wp(y1 := y1 − y2, J))

OR (y1 <= y2 AND wp(y2 = y2 − y1, J))}

1\. ≡ {(y1 > y2 AND y1 − y2 > 0 AND y2 > 0 AND gcd(x1, x2) = gcd(y1 − y2, y2))

2\. OR (y1 <= y2 AND y1 > 0 AND y2 − y1 > 0 AND gcd(x1, x2) = gcd(y1, y2 − y1))}

B ∧ J where B = NOT(y1 = y2)

$B \wedge J \equiv \{NOT(y1 = y2) \text{ AND } \underbrace{y1 > 0 \text{ AND } y2 > 0}_{\text{precondition for using hints}} \text{ AND } gcd(x1, x2) = gcd(y1, y2)\}$

↓

2 cases: 1. (y1 > y2) $\xrightarrow{\text{from hints}}$ gcd(y1, y2) = gcd(y1 − y2, y2)

2\. (y1 < y2) ⟶ gcd(y1, y2) = gcd(y1, y2 − y1)

Therefore, can break B ∧ J down into:

1\. {(y1 > y2 AND y1 > 0 AND y2 > 0 AND gcd(x1, x2) = gcd(y1, y2) = gcd(y1 − y2, y2))

2\. OR (y2 > y1 AND y1 > 0 AND y2 > 0 AND gcd(y1, y2) = gcd(x1, x2) = gcd(y1, y2 − y1))

Now, we can show B ∧ J ⇒ J

1. y1 > y2 AND y1 > 0 AND y2 > 0 implies y1 − y2 > 0
   gcd(x1, x2) = gcd(y1 − y2, y2)
2. y2 > y1 AND y1 > 0 AND y2 > 0 implies y2 − y1 > 0
   gcd(x1, x2) = gcd(y1, y2 − y1)

c) Show invariant is sufficient

$\{J \text{ AND } (\text{NOT } B)\} \Rightarrow Q$ where

$J = \{y1 > 0 \text{ AND } y2 > 0 \text{ AND } gcd(x1, x2) = gcd(y1, y2)\}$

$(\text{NOT } B) = (y1 = y2)$

$Q = y1 = gcd(x1, x2)$

$J \text{ AND } (\text{NOT } B) \equiv$

$\{y1 > 0 \text{ AND } y2 > 0 \text{ AND } gcd(x1, x2) = gcd(y1, y2) \text{ AND } y1 = y2\}$

From $y1 = y2 \Rightarrow gcd(x1, x2) = gcd(y1, y1)$

substituting $y1 = y2$

From the hints: $gcd(y1, y1) = y1 \therefore$

substituting: $gcd(x1, x2) = y1 = Q$ ✓