Exercises, Section 9.2

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 9.1.

   Solution (Instructor only):

   Δ1 *The statement will always be* **reached**. *Since we replace one variable with another, a test will* **infect** *if the variables have different values. The infection will* **propagate** *if we skip the body of the* `if` *statement.*

   R: $True$

   I: $A \neq B$

   P: $\neg(B < A) \equiv A \geq B$

   Δ2 *The statement will always be* **reached**. *Since we change the relational operator, a test will* **infect** *if the entire predicate gives a different result. Since the infection will force a different path, the infection will always* **propagate**.

   R: $True$

   I: $(B < A) \neq (B > A) \equiv A \neq B$

   P: $True$

   Δ3 *The statement will always be* **reached**. *Since we replace one variable with another, a test will* **infect** *if the variables have different values. But, the value of* `A` *was assigned to* `minVal` *in the previous statement, so they will always have the same value. Therefore, the mutant is* **equivalent**. **Propagation** *is not relevant for an equivalent mutant.*

   R: $True$

   I: $A \neq minVal \equiv False \rightarrow equivalent$

   P: N/A

   Δ4 *The statement is* **reached** *if the predicate is true. A* `Bomb()` *mutant raises an immediate runtime exception, so it always* **infects**. *Likewise,* `Bomb()` *mutants always* **propagate**.

   R: $B < A$

   I: $True$

   P: $True$

   Δ5 *The statement is* **reached** *if the predicate is true. Since we replace one variable with another, a test will* **infect** *if the variables have different values. Since* `minVal` *has been given a different value, the infection will always* **propagate**.

   R: $B < A$

   I: $A \neq B$

   P: $True$

   Δ6 *The statement is* **reached** *if the predicate is true. A* **failOnZero()** *mutant raises an immediate runtime exception if the expression is zero. failonZero() mutants always* **propagate**.

R: $B < A$
I: $B = 0$
P: $True$

2. Answer questions (a) through (d) for the mutant on line 5 in the method `findVal()`.

   (a) If possible, find test inputs that do **not reach** the mutant.
   Solution:
   `findVal`*: The mutant is always reached, even if* `x = null`*.*

   (b) If possible, find test inputs that satisfy reachability but **not infection** for the mutant.
   Solution:
   `findVal`*: Infection always occurs, even if* `x = null`*, because* `i` *always has the wrong value after initialization in the loop.*

   (c) If possible, find test inputs that satisfy infection, but **not propagation** for the mutant.
   Solution:
   `findVal`*: As long as the last occurrence of* `val` *isn't at* `numbers[0]`*, the correct output is returned. Examples are:* `(numbers, val) = ([1, 1], 1)` *or* `([-1, 1], 1)` *or* `(null, 0)`*.*

   (d) If possible, find test inputs that strongly **kill** the mutants.
   Solution:
   `findVal`*: Any input with* `val` *only in* `numbers[0]` *works. An example is:* `(numbers, val) = ([1, 0], 1)`

```
/**
 * Find last index of element
 *
 * @param numbers array to search
 * @param val value to look for
 * @return last index of val in numbers; -1 if absent
 * @throws NullPointerException if numbers is null
 */
1.  public static int findVal(int numbers[], int val)
2.  {
3.     int findVal = -1;
4.
5.     for (int i=0; i<numbers.length; i++)
5'. // for (int i=(0+1); i<numbers.length; i++)
6.        if (numbers [i] == val)
7.           findVal = i;
8.     return (findVal);
9.  }
```

3. Answer questions (a) through (d) for the mutant on line 6 in the method `sum()`.

   (a) If possible, find test inputs that do **not** reach the mutant.
   Solution (Instructor only):
   `sum`*: If* `x` *is null or the empty array, ie* `x = null` *or* `[]`*, then the mutant is never reached.*

   (b) If possible, find test inputs that satisfy reachability but **not infection** for the mutant.
   Solution (Instructor only):
   `sum`*: Any input with all zeroes will reach but not infect. Examples are:* `x = [0]` *or* `[0, 0]`*.*

(c) If possible, find test inputs that satisfy infection, but **not propagation** for the mutant.

Solution (Instructor only):
`sum`*: Any input with nonzero entries, but with a sum of zero, is fine. Examples are:* `x = [1, -1]` *or* `[1, -3, 2]`.

(d) If possible, find test inputs that strongly **kill** the mutants.

Solution (Instructor only):
`sum`*: Any input with a nonzero sum works. An example is:* `x = [1, 2, 3]`

```
/**
 * Sum values in an array
 *
 * @param x array to sum
 *
 * @return sum of values in x
 * @throws NullPointerException if x is null
 */
1. public static int sum(int[] x)
2. {
3.    int s = 0;
4.    for (int i=0; i < x.length; i++) }
5.    {
6.       s = s + x[i];
6'.   // s = s - x[i]; //AOR
7.    }
8.    return s;
9.  }
```

4. Refer to the `patternIndex()` method in the `PatternIndex` program in Chapter 7. Consider Mutant A and Mutant B given below. Implementations are available on the book website in files `PatternIndexA.java` and `PatternIndexB.java`.

```
while (isPat == false && isub + patternLen - 1 < subjectLen) // Original
while (isPat == false && isub + patternLen - 0 < subjectLen) // Mutant A

isPat = false;  // Original  (Inside the loops, not the declaration)
isPat = true;   // Mutant B
```

Answer the following questions for each mutant.

Solution (Instructor only):
Note that it is possible for the students to come up with different correct answers. We try to give the general description of a test input to kill the mutant, then an example test.

(a) If possible, design test inputs that do **not reach** the mutants.

Solution (Instructor only):
Mutant A: `subject == null` *or* `pattern == null`
While this is not possible through the command line interface, it certainly is possible in a JUnit test. See the file ***`PatternIndexTest.java`*** *on the book website for an example.*

Mutant B: Three possibilities: (1) `subject == pattern`*; (2)* ***`pattern`*** *is longer than* ***`subject`*** *(3) First character in* ***`pattern`*** *is not in* ***`subject`***
examples: `PatternIndexB X X`

PatternIndexB a abc
PatternIndexB cat hat

(b) If possible, design test inputs that satisfy reachability but **not infection** for the mutants.

Solution (Instructor only):
Mutant A: The pattern needs to be longer than the subject
example: `PatternIndexA x xyz`

Mutant B: Not possible to satisfy reachability and not infection for this mutant.

(c) If possible, design test inputs that satisfy reachability and infection, but **not propagation** for the mutants.

Solution (Instructor only):
Mutant A: The subject must be as long as or longer than the pattern, and the pattern is not in the subject.
example: `PatternIndexA yet yes`

Mutant B: The subject and pattern are the same length, the first characters match, but not all the characters match.
example: `PatternIndexB yet yes`

(d) If possible, design test inputs that **strongly kill** the mutants.

Solution (Instructor only):
Mutant A: `subject == pattern`
example: `PatternIndexA X X`

Mutant B: The pattern is a substring of the subject, and a partial match occurs first.
example: `PatternIndexB ohyeahyes yes`

Fixed thanks to Xin Meng, Spring 2012.

5. Why does it make sense to remove ineffective test cases?

Solution (Instructor only):
They do not contribute to coverage, and thus are needed in a coverage-based test approach. However, it is important to realize that the concept of coverage is not perfect and the tests we remove could find additional faults in the software under test.

6. Define 12 mutants for the following method `cal()` using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for `cal()` were created?