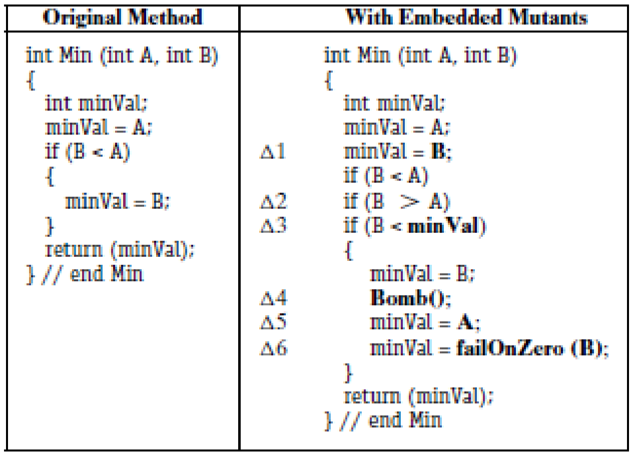
Natnael Kebede

1001149004

Section 5.2: Problems 1, and 2 (only for method sum()).



Original Method

```
int Min (int A, int B)
{
   int minVal;
   minVal = A;
   if (B < A)
   {
      minVal = B;
   }
   return (minVal);
} // end Min
```

With Embedded Mutants

```
        int Min (int A, int B)
        {
           int minVal;
           minVal = A;
Δ1         minVal = B;
           if (B < A)
Δ2         if (B  > A)
Δ3         if (B < minVal)
           {
              minVal = B;
Δ4            Bomb();
Δ5            minVal = A;
Δ6            minVal = failOnZero (B);
           }
           return (minVal);
        } // end Min
```

Figure 5.1. Method Min and six mutants.

1) Δ2

   Reachability: True, the statement will always be reached.

   Infection: (B < A) ≠ (B > A) additionally, A ≠ B

   Propagation: True, the infection will lead to a different path and will always propagate.

   Δ4

   Reachability: the statement is only reached when the predicate B < A is true.

   Infection: True because Bomb() is a mutant that will result in runtime exception.

   Propagation: True, because Bomb() is a mutants that will always propagate.

   Δ5

   Reachability: the statement is only reached when the predicate B < A is true.

   Infection: A ≠ B

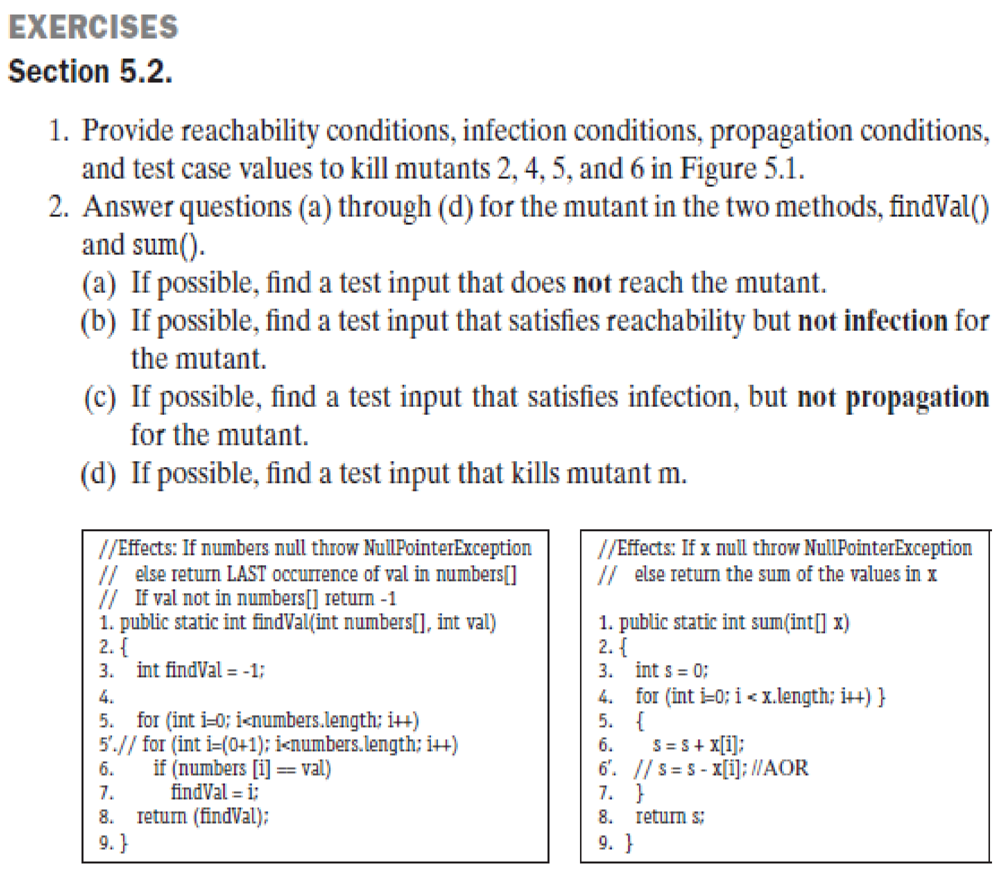
   Propagation: True, because minVal has been given a different value and the infection will always propagate.

   Δ6

   Reachability: the statement is only reached when the predicate B < A is true.

   Infection: failOnZero() results in a runtime exception when B = 0

   Propagation: True, because failOnZero() is a mutants that will always propagate.



EXERCISES

Section 5.2.

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 5.1.
2. Answer questions (a) through (d) for the mutant in the two methods, findVal() and sum().
   (a) If possible, find a test input that does **not** reach the mutant.
   (b) If possible, find a test input that satisfies reachability but **not infection** for the mutant.
   (c) If possible, find a test input that satisfies infection, but **not propagation** for the mutant.
   (d) If possible, find a test input that kills mutant m.

```
//Effects: If numbers null throw NullPointerException
//   else return LAST occurrence of val in numbers[]
//   If val not in numbers[] return -1
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. }
```

```
//Effects: If x null throw NullPointerException
//   else return the sum of the values in x

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. }
```

2 a) If the test input for x = NULL or [] the mutant will not be reached

b) If we consider a test input where x = [0, 0] then it satisfies reachability but not infection for the mutant.

c) If we consider a test input where x = [1, 3, -4] and other test inputs where each element is non-zero and the sum of the elements is a zero, it satisfies infection but not propagation for the mutant.

d) If we consider a test input where x = [1, 3, 4] and other test inputs where the sum of the elements isn't zero, it will strongly kill the mutant.