Q3. Show that any of the following modifications to Peterson's algorithm makes it incorrect:

a. A process in Peterson's algorithm sets the turn variable to itself instead of setting it to the other process. The remaining algorithm stays the same

In this scenario, a thread setting itself to have its turn would end allow it to skip the busy wait period of the algorithm, allowing for a case in which this thread enters a critical section while the other thread is accessing the critical section, creating a race condition, and violating the mutual exclusion aspect of Petersons algorithm.

b. A process sets the turn variable before setting the wantCS variable

In this scenario, Thread i setting the wantCS variable before the turn variable could cause Thread j to misinterpret the state of the system, and act as if it has the right to access the critical section, leading to a race condition. This also violates the Peterson algorithms ability to satisfy mutual exclusion, making the algorithm incorrect.

Q4. Prove that Petersons Algorithm is starvation free.
Proof by contradiction:

Assuming Code and functions from this implementation of Petersons from the book:

```
1  class PetersonAlgorithm implements Lock {
2      boolean wantCS[] = {false, false};
3      int turn = 1;
4      public void requestCS(int i) {
5          int j = 1 - i;
6          wantCS[i] = true;
7          turn = j;
8          while (wantCS[j] && (turn == j)) ;
9      }
10     public void releaseCS(int i) {
11         wantCS[i] = false;
12     }
13 }
```

Figure 2.6: Peterson's algorithm for mutual exclusion

Suppose for contradictions sake that thread i waits forever in the requestCS() method. Specifically that it runs in the while() loop forever waiting for either wantCS[j] to be false or turn to be equal to j. If thread J also got stuck in its while() loop, then it must have read J from turn. But since turn cannot be both i and J, this is a contradiction, and the hypothesis that thread J also got stuck in its requestCS() method is impossible. Thus, thread J must be able to enter its critical section, finish, and call the releaseCS() method. This will result in wantCS[j] becoming false, triggering Thread i to enter its own critical section. Hence, thread i must not wait forever at its while() loop, proving that Petersons Algorithm is Starvation free.