Q3. TACC Username for Shamma Kabir: skabir1
George Doykan: gdoykan

Q4.

a. If two processes p0 and p1 sets the turn variable to itself then a scenario such as this could occur:
p0: wantCS[0] = true
turn = 0
while (wantCS[1] && turn =1)
entersCS
p1 starts after p0 exits the busy wait and entersCS
p1: wantCS[1] = true
turn = 1
while (wantCS[0] && turn =0) → because turn = 1, p1 doesn't wait and entersCS
entersCS
therefore, we can see that both processes enter the critical section at the same time because neither of the processes wait making the algorithm incorrect because it violates mutual exclusion.

b. if two processes p0 and p1 set the turn before setting the wantCS variable then suppose a scenario such as this occurs:
p0 : sets turn = 1 and then it switches to p1
p1 : turn = 0
wantCS[1] = true
wait (wantCS[0] && turn = 0) → doesn't wait, because wantCS[0] is FALSE, p0 hasn't set it yet, so p1 enters the critical section
p0 : wantCS[0] = true
wait (wantCS[1] && turn = 1) → doesn't wait, because turn != 1 (p1 changed it when it interrupted p0 earlier), so p0 enters the critical section
therefore, both processes enters the critical section which would make the Peterson algorithm incorrect because it violates mutual exclusion.

Q5. Let's consider this problem with two processes, s.t p0 accesses turn0 and p1 accesses turn1.

```
p0: 1. wantCS[0] = true
    2. turn0 = turn1
    3. while (wantCS[1] && (turn0 == turn1)) //wait
    4. enterCS
    5. wantCS[0] = false

p1: 1. wantCS[1] = true
    2. turn1 = 1 – turn0
    3. while (wantCS[0] && (turn0 != turn1)) //wait
    4. enterCS
    5. wantCS[1] = false
```

We can see that this algorithm does not starve any processes, is dead-lock free, and satisfies the mutex property. It is starvation free because if p0 waits while p1 is in the critical section, then as soon as p1 exits, p0 will leave the wait because wantCS[1] will be false (vice versa). The algorithm is free of deadlocks because the two different while loop conditions is (turn0==turn1)

or (turn0 != turn1) you know that they're not going to be in the while loop at the same time, waiting. The algorithm also satisfies mutex because we are just checking (turn0==turn1) or (turn0 != turn1).

Q6. While all the numbers are initially 0, consider processes p0 and p1 are choosing their numbers concurrently (so there are two processes, N=2). Without choosing, the pseudo-code would look a little like this:

```
requestCS():
for (int j = 0; j < N; j++) {
        if (number[j] > number[i])
                number[i] = number[j];
number[i]++;
 for (int j = 0; j < N; j++)
        while(number[j]!=0 && (number[j]<number[i]||(number[j]=number[i]&&j<i));
        //busy wait

  CriticalSection();
```

Let's p0 is on its second iteration of the first for loop in the requestCS method but before p0 updates its number, p1 interrupts and goes through the requestCS. Before it goes back to p0, p1 enters the second for loop and enters the critical section because according to p1 everything in the numbers array is 0, so it doesn't need to wait. Now we go back to p0, p0 updates its number and now number[0] == number[1]. Now, when p0 enters the second for loop it does not wait because even though number[0] == number[1], BUT 0 < 1, so it can enter the critical section. Now both processes are in the critical section, so the bakery algorithm does not work with the absence of *choosing* variables.