1.) a.) Consider the following sequence of events.

```
Inc(): read tally to register (reads 0)
Dec(): read tally to register (reads 0)
Dec(): perform decrement loop
Dec(): write tally (writes -50)
Inc(): perform increment loop (tally register reads 0)
Inc(): write tally (writes 50)
```

This order of execution saves each increment, but overwrites each decrement since the threads don't affect eachother's registers, thus the maximum value for tally is 50. A similar argument shows that the minimum value for tally is -50.

b.) Suppose the two threads run exactly like the previous sequence, except each `Dec()` is replaced with the second `Inc()`. In this case, each increment performed by the second `Inc()` would be overwritten by the first thread, and thus the minimum value for tally is 50. If there is no interference between the threads, then each increment would be counted for each thread, and thus the maximum value for tally is 100.

2.) Let S be a semaphore with a value of 1 and let P_1 and P_2 be processes. If `wait()` was non-atomic, then the following could happen:

```
P1 reads a value of 1 from S
P2 reads a value of 1 from S
P1 modifies S and enters the critical section
P2 modifies S and enters the critical section
```

After this program runs, both P_1 and P_2 will be in the critical section simultaneously, thus violating mutual exclusion.

3.) The following psuedocode can be used to synchronize the mechanics:

Mechanic 1:

```
while true:
  wait(A)
  wait(B)
  wait(C)
  fix()
  signal(A,B,C) (shorthand for signal(A); signal(B); signal(C))
  take_break()
```

Mechanic 2:

```
while true:
  wait(A)
  wait(C)
  fix()
  signal(A,C)
  take_break()
```

Mechanic 3:

```
while true:
  wait(B)
  wait(C)
  fix()
  signal(B,C)
  take_break()
```

Assuming wait operations must be performed sequentially as written, it is impossible for this program to deadlock because it is impossible for any of the mechanics to claim resource C and not finish their repair, thus no mechanic can claim C and then get stuck waiting for another resource.

4.) a.) Adding both the available and taken resources, we find that the total number of available resources is 6, 7, 12, and 12 for R_1, R_2, R_3, and R_4 respectively.

b.) The remaining needs of the processes are given in the table:

	R_1	R_2	R_3	R_4
P_1	0	0	0	0
P_2	0	7	5	0
P_3	6	6	2	2
P_4	2	0	0	2
P_5	0	3	2	0

c.) The system is in a safe state because the execution order $\{P_1, P_4, P_5, P_2, P_3\}$ successfully runs each process.

d.) As previously mentioned, the system is in a safe state, and thus the system is not deadlocked.

e.) There is a high potential for processes 2 and 3 to become deadlocked with eachother. Since their collective demand for resources 1 and 2 are greater than the total available, it is possible to allocate resources in such a way that a deadlock occurs.

f.) After the new request, P_3 would need 7 units of R_2. Performing the execution order from (c.), we would be left with P_3 as the last process, in which case there are clearly enough resources for it to run. However, if the request was granted immediately, then the system would deadlock, as after you execute P_1, P_4, and P_5, the system would only have 6 units of R_2 left instead of 7, and thus neither P_2 nor P_3 would be able to run next, thus causing a deadlock.

5.) Each philosopher is a separate instance of the following program:

```
while true:
  wait(CHOPSTICKS[position], CHOPSTICKS[position + 1 modulo 5])
  eat_rice()
  signal(CHOPSTICKS[position], CHOPSTICKS[position + 1 modulo 5])
  sleep()
```

Since each philosopher is forced to wait for both chopsticks to be ready, it is impossible for a philosopher to pick up only a single chopstick then get stuck waiting, thus preventing deadlocks. One potential downside to this solution is that a philosopher might, by chance, have to spend more time waiting for the chopsticks than if they were allowed to pick up one chopstick at a time.

6.) a.) The minimum value for M is 7, each process can claim up to 2 resources without completing, but as soon as any of them claim a third, it runs and subsequently frees the resources, preventing a deadlock.

b.) Let T be the total needs of the processes, and suppose each process claims one less than its need, then the total number of claimed resources C would be given by $C = T - N < M + N - N = M$, thus $C + N < M$, thus we know that in this state there is still at least one available resource. However, since each process only requires one more resource, any process that claims the remaining resource will run and subsequently free its resources, thus a deadlock cannot occur.