Question 1.

The purpose of a condition variable is to allow a thread to block itself until a specified data reaches a predefined state. It also allows threads to atomically release a lock and enter a sleeping state (wait). A condition variable is needed for functions like pthread_cond_wait and pthread_cond_signal / broadcast, which functions we used excessively in this assignment.

Question 2.

The pthread_cond_wait function unlocks the mutex, therefore, it cannot possibly unlock a mutex which isn't locked. Hence, it is always wrong to call pthread_cond_wait on an already unlocked mutex. The whole purpose of the wait function is so that it can block a thread and atomically release the lock, which allows another thread to take the mutex, and then the other thread should then signal this thread to unblock given specific conditions.

Question 3.

One plausible scenario where it is reasonable to use the timedwait function, is when there is a possibility of error in another thread such that, the other thread will never send a signal back.

For example, the other thread is performing a mathematical calculation which involves limits, and if the limit of the particular function doesn't exist then the thread will never send a signal back nor unlock due to an infinite loop. Another example is if the other thread runs an algorithm that is non-polynomial time, hence, it would take very long to execute until a signal is sent back to the waiting thread. Therefore, a timed wait function would be very appropriate.

Justification
In the real world it is often required to calculate limits of particular functions, and in many cases, these limits never terminate. Therefore, a timedwait condition is very useful and appropriate in saving processing time and computer memory. Additionally, if the function continues to run for a long time, there is a chance a lot of memory will be wasted. Once again, the same can be said about the example with a non-poly time algorithm.

Question 4.

- As seen in my code, there are two stages to each student and teacher routine. The first stage is the distribution stage. This is where the teacher assigns each student thread a group ID. To ensure that the students have their own ID, I have malloced space in the heap and have passed in the reference to this part in memory through the thread routine parameter. The second stage is where the students get called into the lab room by group IDs. Below answers all the questions about synchronisation issues on a high level. The reason I have decided to explain it on a high level is because the specifics are clearly seen in my code.

The first stage on a high level:

- The teacher thread will keep waiting until all students have taken the distributed IDs
- While the teacher thread is waiting, the student threads will update their group ID (which is the malloced memory mentioned above) and will increment a global variable which keeps track of how many students have been allocated to a group
- Once this global counter reaches K, another global variable which keeps track of the number of groups increments
- The last student then signals the teacher that all the IDs have been received and assigned, and resets the students in group counter to 0, ready for the next group
- The teacher then waits again until all students have taken the distributed IDs
- This repeats until the total number of groups counter reaches N/K + 1 (i.e. all the students have been assigned a group ID)

Between the first and second stage:

- Students that have collected IDs wait until all the students have all collected IDs before continuing to the second stage
- In the first stage there is a global counter which counts the total number of students collecting IDs
- So, when this counter equals N then that means the last student has just collected their ID, when this happens the thread sends a signal to the teacher, telling the teacher that all students have collected their IDs

The second stage on a high level:

- The students keep waiting until their group ID has been called
- Once the teacher calls out a group ID, teacher thread keeps waiting until all students in the particular group enters the lab
- Each student thread will wait right after they enter the lab (waiting for the teacher to let them leave)
- Once the last student in the group enters the lab, the last student will generate a random time between T/2 and T, set it as the global variable, and then reset some global counters and lastly, signal the teacher that all members have entered the lab
- To ensure synchronisation for the printing statements, the teacher will then signal for all students to leave the lab after the teacher routine has slept for the randomly generated time
- Right before they leave, they will all print out that they have completed the lab

Question 5.

Testing methods:

- I wrote a bunch of I/O unit tests for specific scenarios, these scenarios included variations of different inputs by the user. Below outline them all:
    - When N, K or T is invalid (i.e. negative or zero)
    - When K is larger than N
    - Different combinations of N, K ratios (T will always be valid as long as it isn't negative)
        - N = 5, K = 1
        - N = 5, K = 2
        - N = 5, K = 3
        - N = 5, K = 4
        - N = 5, K = 5
        - N = 147, K = 23
        - N = 100, K = 3
        - N = 6, K = 2
        - N = 41, K = 3

Debugging:

- It was very difficult when I only thought about what happened by purely reading the code so,
- the first thing I tried was to use print statements before wait and signal statements, this worked for most of my bugs. For example, when the program didn't continue running, there was a high chance that a thread is blocked and is waiting to be woken up, so I would put a print statement before every signal of that particular condition variable. The same was done with the wait conditions. I would also print the return value of pthread_cond_wait, and if it didn't equal 0, then a problem had occurred. Therefore, printing was in fact a very effective way of debugging, however, when it no longer worked, I had to resort to
- using a variation of parallel stacks which showed call stack information for all the threads. This was very useful because I was able to visualise (using task view) and see exactly where the problem occurred.