Assignment 1

Question 1.

(a) A software engineer wants to increase the reliability of the software. Which three methods or techniques an engineer can adopt for this purpose?

1. N-Version Programming
2. Recovery Blocks
3. Triple Modular Redundancy

(b) How many minimum redundant components are required in Triple Modular Redundancy (TMR) technique to tolerate simultaneous failures of ‘n’ components?
Total modules = 2n + 1
Min redundant modules = 2n

(c) Why is the N-version programming not effective to build a highly reliable software?
Because it is more expensive and complex for a similar result to other methods.

(d) Why is the ‘recovery blocks’ method only suitable for providing fault-tolerance to real-time tasks having large laxity i.e., larger deadlines than their computation times?
It is only suitable in this case because it accommodates the re-execution in the worst case scenario.

Question 2.

(a) Explain why the later try blocks in the ‘recovery blocks method’ should contain only skeletal code.
Because as you get closer to the end, You want more and more for there to be no faults. if a task has to fall back, it is better for it to fall back to something that will execute faster to keep within the deadline. Lastly, the later blocks will be much less used, and it is better not to spend much time developing them much beyond functional.

Question 3.

(a)As a software engineer, you investigated that computations of the critical component of the software were correct until a particular time instant (let’s say $t4$), however these computations went wrong at $t4$ due to data corruption. More specifically, the computations were correct at time instants $t1$, $t2$ and $t3$. Now which technique do you think is the most appropriate to achieve fault-tolerance in this scenario? Explain your strategy using a timeline with proper annotations.
I would use checkpoint and rollback. Since the data was fine up until t4, the most recent checkpoint would be at t3 and rolling back to t3 would not be too cumbersome. The timeline would be that at T1 verify state and save checkpoint, T2 verify state and save checkpoint, T3 verify state and save checkpoint, T4 verify state and run into fault, rollback the system to the state saved at t3, If the state at t4 is verified correctly it will rerun, if it happens again roll back to the state saved at t2 or t1.

This plan is good because it allows for the fastest rolling back if its a one off error, while still allowing rollbacks to older checkpoints. The downside is that saving a checkpoint after each task can fill up the ram. If there was a large number of tasks, you could make a checkpoint every 2 or 3 tasks, but that risks the system going over the deadline.