3. (a) Table 1 shows a case that an undo/redo log is mistakenly maintained by an undo log:

Step	Action	t	M-A	D-A	Log
1					⟨ START ⟩
2	READ(A,t)	8	8	8	
3	t := t * 2	16	8	8	
4	WRITE(A,t)	16	16	8	⟨ T, A, 8 ⟩
5	FLUSH LOG				
6	OUTPUT(A)	16	16	16	
7					⟨ COMMIT ⟩
8	FLUSH LOG				

Table 1: Log is mistakenly maintained as an undo log.

(Notes for table 1- M-A: Value of A in main memeory; M-B: Value of A in Disk) In this case, atomocity and durability could be violated. Suppose a crash occured after COMMIT, and COMMIT record has been written on the disk. The recovery manager will start scaning the log from bottom and COMMIT record is found. Then, the recovery manager will redo the next actions till it finds the START record. Therefore the record ⟨T, A, 8⟩ gets read, and will be copied to the A's cache slot and flushed to the disk. Eventually, 8 will appear on the disk as updated A's value. Obviously, this is not right because actual updated data sould be 16. Thus, we say the atomicity is violated because the wrong data makes the database element look like it has not been updated, i.e., the transaction is not complete; durability is also violated because once the transaction is committed, the updated element should persist in the database, however, in this case the updated element is alternated (redone) back to the same as before.

(b) Table 2 shows a case that an undo/redo log is mistakenly maintained by a redo log:

Step	Action	t	M-A	D-A	Log
1					⟨ START ⟩
2	READ(A,t)	8	8	8	
3	t := t * 2	16	8	8	
4	WRITE(A,t)	16	16	8	⟨ T, A, 16 ⟩
7					⟨ COMMIT ⟩
5	FLUSH LOG				
6	OUTPUT(A)	16	16	16	
8	FLUSH LOG				

Table 2: Log is mistakenly maintained as an undo log.

(Notes for table 2- M-A: Value of A in main memeory; M-B: Value of A in Disk)

In this case, atomicity could be violated. Suppose a crash occured before the COMMIT. The recovery manager will start examing the log from bottom. Since COMMIT record will not be found, the recovery manager considers the transaction incomplete. Then, the recovery manager will undo the actions it reads till it finds the START record. The next record found will be ⟨T, A, 16⟩. Therefore, the value 16 will be copied to the A's cache slot and flushed to the disk. Eventually, 16 will appear on the disk as updated A's value. However, since the transaction was not complete, A is supposed to stay the same as before which is 8, in order to show the user that the transaction never happens. Thus, in this case, DBMS fails to preserve the atomicity.

4. By looking at the 4 RM procedures (RM-Read, RM-Write, RM-Commit, and RM-Abort), we can see that RM-Read and RM-Write only perform actions on the cache, which is in the main memory. (We assume that it is up to INPUT and OUTPUT to save the contents from memory to disk.) RM-Commit saves the log and the commit list to disk, and RM-Abort saves the abort list to disk. Here, we assume that the commit and abort lists are flushed to the secondary storage after each operation on them. In reality, this may not be true because the system may save those list in memory and then flush them to the disk after it reaches some size. But either way, it does not impact our argument.

   When a system crashes and then reboots, the RM-Restart procedure can only look at what was on the secondary storage, because whatever was in the main memory cannot survive the reboot. Have said this, we know that a crash during RM-Read or RM-Write has no effects on the RM-Restart.

   If we further look into the steps in RM-Restart, we know that it only look at the log entries as well as the commit list, therefore, RM-Abort has no effects either since it deals with the abort list only.

   Therefore, the only possible place where such violation may occur is the RM-Commit procedure. RM-Commit has two steps:

```
1. Flush log to secondary storage
2. Add i to the commit list and acknowledge RM-Commit
```

   If the crash occurs while RM is in the middle of step 1, i.e. flushing log to secondary storage. Since it is not atomic, we can suppose that some log entries made to the disk, 1 log entry made to the disk only half way, and the rest never made to the disk.

   For entries that made to the disk, the RM-Restart will see them. Since none of them is in the commit list (RM-Commit crashed before it reached step 2), RM-Restart will try to undo all of them by restoring their previous values. This is fine.

   For entries that never made to the disk, the RM-Restart will not even know about them. And on the disk, they still pertain their previous values (assume no OUTPUT action was performed), so that is fine as well.

As of the entry that only made to the disk half-way, there could be some problems. It is fine if the entry made all the way to its previous value, and only the new value did not get saved. Then we can still restore it. However, if the previous value did not make to the disk, then RM-Restart may read some garbage values and think of that as the previous values of that particular item. So it will try to "restore" using the garbage values, and can thus violate Durability - the value of a committed transaction get modified without doing another valid committed transaction.

But this problem can be easily solved using an error detection system such as CRC. The RM-Restart will notice that the last entry has an invalid CRC, therefore, it may choose to ignore it as if it never made to the disk. Then everything will be fine.

Now, let us suppose the crash occurs while RM-Commit is saving the commit list. In this case, it either makes to the disk or it does not (a garbage value of half-way save would be regarded as not on disk because the original transaction number does not match this one). Since this happens after the log was completely saved on the disk, we can either redo the transaction if it made to the disk, or undo it if it had not. Atomicity is not likely to be violated.