1

- A：原子性
  - 事务是不可分的原子，其中的操作要么都做，要么都不做
  - 反例：A要转账给B，金额为100元，那么要么A账户减100并且B账户加100元全部都做，要不然就都不做，即rollback。

    若只让A账户-100，而B账户不变，则违反了原子性
- C：一致性
  - 事务的执行保证数据库从一个一致状态转到另一个一致状态
  - 反例：A要转账给B，金额为100元，那么当A账户-100和B账户+100都完成时，才会commit。若仅做了A-100的操作后系统崩溃，恢复系统后A仍然-100，而B账户没+100，那么就违反了一致性 （正确的做法是A-100的修改也不会保存到数据库）
- I：隔离性
  - 多个事务一起执行时相互独立
  - 反例：A给B转账100元的同时C给A转账100元，这两个事务必须互不干扰。否则，可能存在A账户最终账户余额增加100的情况
- D：持久性
  - 事务一旦成功提交，就在数据库永久保存
  - 反例：commit并写回磁盘后的数据在系统崩溃后，再修复数据库后发现发生了变化或丢失了。

2

- A中嵌套了B，那么若B成功提交后，A执行出现错误需要回滚，此时B已经提交的修改已经写入磁盘了，这样A的回滚达不到期望的效果，违反数据库的原子性
- A中嵌套了B，A在读了数据a后并操作后，只放在内存里，但还未写回磁盘，接着B又读数据a，此时读的是内存里的数据，是脏读，可能会有一些未知错误。

3

(1)

- Undo列表：{T2,T3}
- Redo列表：{T1}
- Undo

```
T3:E=50
T2:D=40
T2:C=30
```

- Redo

```
T1:A=40
T1:B=60
T1:A=75
```

- Write log

```
<Abort，T2>
<Abort，T3>
```

- 结果：A=75, B=60, C=30, D=40, E=50, F=unknown, G=unknown

(2)

- Undo 列表： {T3}
- Redo 列表：{T1,T2}
- Undo

```
T3:E=50
```

- Redo

```
T1:A=40
T1:B=60
T1:A=75
T2:C=50
T2:D=80
T2:D=65
T2:C=75
```

- Write log

```
<Abort,T3>
```

- 结果 A=75, B=60, C=75, D=65, E=50, F=unknown, G=unknown

(3)

- Undo 列表： {T4}
- Redo 为空，checkpoint后不需要再关注 {T1，T2，T3}
- Undo

```
T4:G=70
T4:F=60
```

- Wirte log

```
<Abort,T4>
```

- 结果 A=75, B=60, C=75, D=65, E=90, F=60, G=70（F和G是旧值)

4

- 证明：2PL里，lock操作先于unlock操作。以并发事务T1和T2为例（可扩展到更多的并发事务）
  若事务T1写数据A，T2读或写A ； 或者T1读数据A，T2写A
  以上两种情况下，是没法串行化的。

设T1，T2潜在冲突公共对象为$A_1,\ldots,A_n$，其中
$X_1=\{A_1,\ldots,A_j\}$是事务T_1写数据A，T_2读或写A

$X_2=\{A_{j+1},\ldots,A_n\}$是$T1$读数据$A$，$T2$写$A$

此时T_1要用**X锁q**，而T_2用**S锁或X锁p**。假设**q**先执行，则T_1获得锁，T_2等待。根据2PL的原理，T_1在全部获得X_1与X_2中的对象的锁后才会开始释放锁。若此时T_2获得了A_i的锁，那么会发生死锁，在冲突图中形成环，即违背2PL。否则只有当T_1全部执行完后，X_1,X_2的事务的锁才会释放，让等待的T_2开始执行。所以T_1,T_2可以顺序串行执行，它们的调度是冲突可串的，是可串化调度。

当先执行**p**时，同理易得。

所以原命题得证

5

t	T_1	T_2
1	lock(A)	
2	read(A,t)	
3	t ← t+1	
4	write(A,t)	lock(A)
5	unlock(A)	wait
6	rolleback	read(A,t)

满足2PL，但由于最后T_1Rollback，导致T_2对A的Read是脏读

6