This test covers: Chapters 3 (and past material).
Directions: Complete all questions. Partial credit will be given. Keep in mind there are many ways to answer most questions correctly.

The situation: You have been contracted by a startup e-commerce company to construct a database containing information about sales. Vital information includes the customer ID, customer name, customer address (single attribute), customer phone, item numbers, item quantities, item cost, transaction ID and transaction date. Beginning with the the relation:

Commerce(cID, cName, cAddress, cPhone, item, itemQuantity, itemCost, tID, tDate)

To simplify the scenario, will assume that descriptors are static, that is names, addresses, phone numbers, prices and transaction dates cannot be altered once set. Since this is *your* creation, in any of these questions below, if you are making what may be a controversial decision (something that could logically go more than one way), please explain.

1.) Given the scenario and the assumption of static descriptors, create a description for each attribute of `Commerce`.

cID - Customer ID number; unique to each person (integer).

cName - Customer name, may not be unique across all customers.

cAddress - Customer address, may not be unique across all customers.

cPhone - Customer phone number, may not be unique across all customers.

item - Item number of product (integer); Unique identifier.

itemQuantity - Number of items `item` purchased by customer in transaction `tID`.

itemCost - Cost of `item`.

tID - Transaction ID number; Unique integer.

tDate - Date of transaction `tID`.

2.) It should be immediately clear that `Commerce` has serious drawbacks. Describe issues including but not necessarily limited to anomalies.

- Customer's name/address/phone is repeated in many places.
- Item price is repeated each time an item is sold. In reality this would be probably correct, but under our assumption about static data entries it would not be ideal. Then again, if it is static, it would not be changed. A catch-22.
- `tDate` is repeated for each item in a given transaction.
- Given the key below: only one customer per transaction and an item may only be inserted once per transaction.

- Prices cannot change. A workaround would be to create a new item number for each price change, but that limits searching past transactions for a given item.
- Sales/discounts on items are not allowed without creating different item numbers.
- A customer cannot move or change phones. This requires much thought on implementation if it were to be allowed.
- Many more things that do not become apparent until implementation.

3.) Describe all keys of the relation `Commerce`.

This depends on how you look at the problem, but one such view is that for each transaction each `item` may only be ordered once (with multiples taken care of by `itemQuantity`). In this view, the only key would be (`tID`, `item`). Note customer ID could be implied by `tID`, so it would not need to be part of the key. This also assumes one customer per transaction which may or may not be valid.

4.) Describe all non-trivial functional dependencies of the relation `Commerce` (there are many, if $A \rightarrow B$ you do not need to include $(A, C) \rightarrow B$).

- `cID` $\rightarrow$ `cName`
- `cID` $\rightarrow$ `cAddress`
- `cID` $\rightarrow$ `cPhone`
- `item` $\rightarrow$ `itemCost`
- `tID` $\rightarrow$ `tDate`
- `tID` $\rightarrow$ `cID`
- `tID, item` $\rightarrow$ `itemQuantity`
- `cName,cAddress,cPhone` $\rightarrow$ `cID` — Could be argued either way, ignored here.
- `cID, tDate` $\rightarrow$ `tID` — Could be argued either way, ignored here.

5.) Find the closure of:

- `cName` = (`cName`) — Could include cID/address/phone depending on assumptions.
- `tDate` = (`tDate`)
- `tID` = (`cID, cName, cAddress, cPhone, tID, tDate`)

- (cID,item) = (cID, cName, cAddress, cPhone, item, itemCost)

6.) Find a minimal basis for `Commerce`.

- `cID` → `cName`
- `cID` → `cAddress`
- `cID` → `cPhone`
- `item` → `itemCost`
- `tID` → `tDate`
- `tID` → `cID`
- `tID, item` → `itemQuantity`

7.) Decompose `Commerce` into multiple relations where each satisfies Third Normal Form (3NF). Project functional dependencies onto to new relations.

- Load the first three together to get FD: `cID` → (`cName, cAddress, cPhone`). The left is not a superkey and the right does not contain members of a key. This splits off the relation:

  Customers(`cID, cName, cAddress, cPhone`)

- Split the next FD: `item` → `itemCost` to get:

  Items(`item, itemCost`)

- Split the FD: `tID` → `tDate` to get:

  Transactions(`tID, tDate`)

Recapping, `Commerce` becomes:

- **CommerceTemp**(`cID, item, itemQuantity, tID`)
- **Items**(`item, itemCost`)
- **Customers**(`cID, cName, cAddress, cPhone`)
- **Transactions**(`tID, tDate`)

One FD violation remains in `CommerceTemp`: `tID` → `cID`. This can be fixed by creating another relation with attributes `tID` and `cID` or better by moving cID into Transactions.

- **TransactionItems**(`item, itemQuantity, tID`)
  - (`tID, item`) → `itemQuantity`
- **Items**(`item, itemCost`)

  - `item` → `itemCost`

- **Customers**(`cID, cName, cAddress, cPhone`)
  - `cID` → `cName`
  - `cID` → `cAddress`
  - `cID` → `cPhone`
- **Transactions**(`tID, cID, tDate`)
  - `tID` → `tDate`
  - `tID` → `cID`

8.) Decompose `Commerce` into multiple relations where each satisfies Boyce-Codd Normal Form (BCNF).

DONE! BCNF is a stricter normal form than 3NF, but the lefthand side of all FD are keys or superkeys. This means that the above solution also satisfies BCNF.

9.) Compare and contrast the relations created by 3NF and BCNF.

- They are the same. There will be no anomalies and no loss of functional dependencies. This is typical of well-thought real-world examples.

10.) The benefit of the original relation is that all data is in a single relation. Once decomposed into BCNF, the data is spread over many relations. Write a query (using BCNF relations) that provides the data necessary to fill the item portion of an invoice. Each returned tuple should consist of an item number, item quantity, item price and an extended price.

Assume looking for transaction __TRANSACTION_NUMBER__:

```
    SELECT item, itemQuantity, itemPrice, itemPrice*itemQuantity AS extendedPrice
FROM TransactionItems NATURAL JOIN Transactions
WHERE tID = __TRANSACTION_NUMBER__;
```

11.) [BONUS: 5pts] Write a query that finds to the total price for a given transaction ID.Assume looking for transaction __TRANSACTION_NUMBER__:

```
    SELECT tID, sum(extendedPrice) AS totalPrice
FROM ( SELECT tID, itemPrice*itemQuantity AS extendedPrice
   FROM TransactionItems NATURAL JOIN Transactions
   WHERE tID = __TRANSACTION_NUMBER__; )
GROUP BY tID;
```

or

```
 SELECT tID, sum(itemPrice*itemQuantity) AS totalPrice
FROM TransactionItems NATURAL JOIN Transactions
WHERE tID = __TRANSACTION_NUMBER__
GROUP BY tID;
```