Example Inputs/Outputs

Test for 1NF,2NF,3NF, and BCNF

```
y
test1
1,2,3,4,5,6,7,8,9,10
1,2 3
y
10
y
1->4 3->5 6->7 8->2
B
done
n
```

Test for 1NF-3NF and no detected MVD for 4NF

```
y
test2
1,2,3,4,5,6,7,8,9,10
1,2 3
y
10
y
1->4 3->5 6->7 8->2
4
d,a,d,f,a,10,11,12,13,14
g,a,d,h,b,16,17,18,19,20
g,c,d,h,c,21,22,23,24,25
f,a,d,e,c,25,26,27,28,28
g,e,d,e,b,29,30,31,32,33
x,x,d,e,c,34,35,36,37,38
done
n
```

Test for successful detection of 4NF MVD

```
y
test3
1,2,3,4,5
1
n
y
1->4 1->5
4
d,a,d,f,a
```

g,a,d,h,b
g,c,d,h,c
f,a,d,e,c
g,e,d,e,b
x,x,d,e,c
done
n

Test for 5NF join dependency detection

y
test4
OrderID,CustomerID,DrinkID,Milk
OrderID,CustomerID,DrinkID,Milk
n
n
5
1001,1,1,ND
1001,1,1,D
1002,1,2,D
1003,2,3,ND
1003,2,3,D
1003,2,4,ND
done
n

Given 1-4NF sample input

y
CoffeeShopData
OrderID,Date,PromocodeUsed,TotalCost,TotalDrinkCost,TotalFoodCost,CustomerID,CustomerName,DrinkID,DrinkName,DrinkSize,DrinkQuantity,Milk,DrinkIngredient,DrinkAllergen,FoodID,FoodName,FoodQuantity,FoodIngredient,FoodAllergen
OrderID,DrinkID,FoodID
y
PromocodeUsed,DrinkIngredient,DrinkAllergen,FoodIngredient,FoodAllergen
y
OrderID->PromocodeUsed DrinkID->DrinkIngredient DrinkID->DrinkAllergen
FoodID->FoodIngredient FoodID->FoodAllergen
OrderID->Date,TotalCost,TotalDrinkCost,TotalFoodCost,CustomerID,CustomerName
OrderID,DrinkID->DrinkSize,DrinkQuantity,Milk OrderID,FoodID->FoodQuantity
CustomerID->CustomerName DrinkID->DrinkName FoodID->FoodName
3
1001,6/30/2024,NONE,$7.25,$7.25,$0.00,1,Alice Brown,1,Caffe Latte,Grande,1,ND,Espresso|Oat Milk,Oat,0,NULL,0,NONE,NONE

1002,6/30/2026,SUMMERFUN,$9.98,$5.99,$3.99,2,David Miller,2,Iced Caramel Macchiato,Tall,2,D,Espresso|Vanilla Syrup|Milk|Ice,Dairy|Nuts,3,Blueberry Muffin,1,Flour|Sugar|Blueberries|Eggs,Wheat|Egg
1002,6/30/2026,SUMMERFUN,$9.98,$5.99,$3.99,2,David Miller,3,Iced Matcha Latte,Grande,1,ND,Matcha|Coconut Milk|Ice,Nuts,3,Blueberry Muffin,1,Flour|Sugar|Blueberries|Eggs,Wheat|Egg
1003,6/29/2024,SUMMERFUN|JUNEVIP,$115.00,$115.00,$0.00,3,Emily Garcia,4,Vanilla Bean Frappuccino,Venti,8,ND,Coffee|Ice|Vanilla Syrup|Soy Milk,Nuts|Soy,0,NULL,0,NONE,NONE
done
n

*The program does not detect the expected 4NF MVD in this sample input, since the given MVD does not meet the formal requirements to be detected as an MVD.

Input Parser
To allow for more tables to be created from a single table provided at the start, a dictionary is created that holds all tables, and it is updated with changes to tables, as well as new tables throughout the normalizing process. Inputs are taken in the below order:

-The program asks the user if they want to input a relation. This part of the input exists so that the user can have multiple relations normalized in a row without closing the program if they decide to. After the inputs below are received and the relation is normalized, the user is asked if they want to input another relation to decide if the program should repeat or if it should exit.

-The program asks the user to input attributes on one line in the form “a1,a2,a3,a4”. The input that the program receives is then split into values for a tuple based on where the commas are placed. The tuple that is created now contains the attributes for the program, and it is stored in a new table in the dictionary.

-Next, the program asks for candidate keys on one line in the format “a1,a2 a3” where parts of a key are separated by commas, and different keys are separated by a space. After submitting a set of keys as input, they are checked for validity, which requires the parts in each key are attributes added in the previous step. If the input is not valid, the program notifies the user and asks them to submit a fixed input.

-The program then asks the user if there are any non-atomic attributes in the relation, since sometimes they may not be present and therefore would not need to be entered (‘y’ = Yes, ‘n’ = No). If the user says that there are non-atomic attributes present, then the program asks for a valid subset of the existing attributes that are non-atomic in the form “a1,a2,a3”. The input is split by the commas, and the multivalued attributes are stored to be used when normalizing to 1NF form later.

-The program then asks the user if there are any functional dependencies in the relation, since sometimes they may not be present as there may only be keys in the relation, and therefore

would not need to be entered (‘y’ = Yes, ‘n’ = No). Functional dependencies are taken in the form a1,a2->a3,a4 a5->a6 with attributes on one side of the functional dependency being separated with commas, the left side of the functional dependency is separated from the rights side by ‘->’, and different functional dependencies are separated with a space. If any functional dependencies are found to be invalid with the available attributes, then the user is asked to fix the input to enter only valid functional dependencies. If the whole input is valid, it is then taken and split into parts as described above, with the left and right sides of the functional dependencies each stored in their own list.

-The program then asks the user for which form they plan to reach, with the valid options as shown below:
Input : Form to reach
1 : 1NF
2 : 2NF
3 : 3NF
B : BCNF
4 : 4NF
5 : 5NF

-The user can also input data instances, although they are only required if the user decides to normalize to 4NF or 5NF form. Data instances are only expected when the user wants to reach 4NF or higher because they are not necessary to normalize a relation from forms 1NF-BCNF, but are for 4NF and 5NF. Data instance inputs are expected to be received in the form:
1001,1,1,ND
1001,1,1,D
1002,1,2,D
1003,2,3,ND
1003,2,3,D
1003,2,4,ND
done
Each line of the input represents one tuple of data, with each piece of data being split by a comma. Tuples of input are connected to attributes in the order they were originally entered (ex: attributes 1,2,3,4 would connect to data tuple a,b,c,d as 1:a, 2:b, 3:c, and 4:d). Inputs are also expected to be the same length as the number of attributes, and invalid inputs are discarded (ex: for attributes 1,2,3, all data inputs must be 3 data pieces long, such as a,b,c). Data inputs that include non-atomic attributes can also be include in the form a|b|c where each piece of data in a group for an attribute is split with a “|”. Since the inputs for data tuples are taken line by line, the user can continue to submit tuples until they submit ‘done’ on its own line, which notifies the program that the user has finished adding data tuples.

Normalizer
1NF
For 1NF normalization, non-atomic attributes (decided by user) are moved from the original table into their own tables. In this program, all attributes from candidate keys are included in the

new relation in order to preserve the data integrity of the key FDs. For other FDs, if they don't involve non-atomic attributes, they are kept as part of the original table, while if they do involve non-atomic attributes, they are moved to the new table. All attributes that make up an FD that is moved to a new table are also brought to the table, so that the full FD can be preserved for data integrity.
-If data instance tuples are present, and multivalued attributes are present, then the data associated with the non-atomic attributes is checked, and any instances of non-atomic data are split into their own data tuples (a,b|c,d -> a,b,c a,c,d)

2NF
For 2NF form, the program reviews all functional dependencies in each table that currently exists. If a dependency is detected as a partial dependency (left side of the FD is only part of the left side of the FD), then a new table is created for the partial dependency with only necessary attributes. The right side attribute of the FD is then removed from the original table, and any FDs that involve it are moved to the new table as well. The partial functional dependency is also removed from the original table when the new table is created.

3NF
For 3NF form, the program reviews all functional dependencies in each table that currently exists. If a dependency is detected as a transitive dependency (FD X->Y doesn't meet the requirements of X being a superkey or Y being a prime attribute), then the dependency is detected as a violation. A new table is then created that includes X and Y, and the original table has Y and its functional dependency removed.

BCNF
For BCNF form normalization, the program reviews all functional dependencies in each table that currently exists. Any dependencies X->Y where X is not a superkey are considered violations of BCNF form. When this occurs, a new table is created that includes attributes X and Y, and the original table has Y and related info removed.

4NF - Extra Credit Automatic Identification
For 4NF and above, data instances are requested for the user in order to determine whether or not a multivalued dependency is present. The data instances are asked for before the relations begin to get normalized, and they are stored until 1NF-BCNF forms are completed. If 4NF form is selected, a process is completed in order to determine if and where a multivalued dependency is present.
-The process that I created
-If a multivalued dependency is found, normalization of the original table to 4NF then occurs. For a MVD X ->> Y|Z, one table would be created that includes all attributes and FDs from X and Y, while the other would include all attributes and FDs from X and Z. The original table is then deleted.
The MVD detection code is designed to find MVDs that meet the following requirements:

-$t_1[X] = t_2[X] = t_3[X] = t_4[X]$

-$t_1[Y] = t_3[Y]$ and $t_2[Y] = t_4[Y]$
-$t_2[Z] = t_3[Z]$ and $t_1[Z] = t_4[Z]$

-The process first gets all data tuples related to the current table by using the get_connected function. The process then has three stages of searching to detect MVDs.
-In the first stage, count_column_duplicates is used to check each column for instances where 4 or more pieces of data are equal in a column. Once the duplicates are identified, gather_tuples_by_duplicate is used to gather each group of matching tuples for the second stage
-In the second stage, count_column_duplicates is used again to check each column (except for the one used in the first stage) for instances where 2 or more pieces of data are equal in a column. Next, gather_tuples_by_duplicate is used twice to gather instances where two sets of two or more duplicates are discovered in a column.
-Next, in the third stage, count_column_duplicates is used to find more sets of at least two matching data pieces (in columns not used in the first and second stage). If a set of 2 or more duplicated data pieces are found connected to a new column in two of the tuple groups gathered in the second stage (matches_found), then an MVD is detected. Once an MVD is detected, the search ends and the 4NF normalization process begins.

-The MVD detection was validated to detect MVDs through test inputs such as the ones given above. I made sure to test the detection with a variety of inputs, to ensure that all expected MVDs were found, and that no MVDs were identified incorrectly. When designing the algorithm, it was challenging to create a process that could go through every possible arrangement of tuples to determine any area in the table that a MVD may have occurred. It was also challenging to keep the complexity as low as possible so that it could be executed quickly. The assumption is made that only MVDs that meet the formal requirements above are detected, which has the drawback of excluding certain MVDs such as the one suggested in the sample 1NF-4NF output, since there are not enough equal tuples to detect the MVD. The process also has the drawback of becoming slower to execute on larger data sets as well as when a table has more attributes. This drawback means that for very large tables/data sets, the process may take a significant amount of time to complete MVD detection.

5NF
For 5NF normalization, the program reviews available data instances to detect if any non-trivial join dependencies exist. To complete this process, the table generates a set of all possible ways it could be decomposed into two tables. Each set of tables then has its tuples retrieved, and the tuples are joined on a selected attribute to recreate the original table. If no data is changed or lost when the table is reconstructed, a join dependency is detected and the table is then normalized to 5NF form by splitting into the two smaller tables.

Final Relation Generator
For output I decided to create a textual representation of the tables based on the normalization process. The output includes the name for each table, its attribute list, and primary keys/candidate keys. Once all tables are normalized to the form selected by the user, they are

then each printed in the terminal as follows:

==========

Table: {Name of table}

Attributes: {List of attributes}

Data Instances: ←(Only included if data instances present)

(datatuple1)

(datatuple2)

…

(lastdatatuple)

Candidate Keys: {List of all candidate keys present}

==========

(one row of space to separate each table)