Challenge 1

Create a query with the following columns:

1. PurchaseOrderID, from **Purchasing.PurchaseOrderDetail**
2. PurchaseOrderDetailID, from **Purchasing.PurchaseOrderDetail**
3. OrderQty, from **Purchasing.PurchaseOrderDetail**
4. UnitPrice, from **Purchasing.PurchaseOrderDetail**
5. LineTotal, from **Purchasing.PurchaseOrderDetail**
6. OrderDate, from **Purchasing.PurchaseOrderHeader**
7. A derived column, aliased as "OrderSizeCategory", calculated via CASE logic as follows:
    - When OrderQty > 500, the logic should return "Large"
    - When OrderQty > 50 but <= 500, the logic should return "Medium"
    - Otherwise, the logic should return "Small"
8. The "Name" field from **Production.Product**, aliased as "ProductName"
9. The "Name" field from **Production.ProductSubcategory**, aliased as "Subcategory"; if this value is NULL, return the string "None" instead
10. The "Name" field from **Production.ProductCategory,** aliased as "Category"; if this value is NULL, return the string "None" instead

Only return rows where the order date occurred in December of ANY year. The MONTH function should provide a helpful shortcut here.

While I'll leave the exact details up to you, I've included a few pointers below on getting your joins to work correctly:

1. Every line item from Purchasing.PurchaseOrderDetail should have a corresponding order ID in Purchasing.PurchaseOrderHeader, so these tables can be joined such that a match is required on both sides of the join.
2. You can join Production.Product to Purchasing.PurchaseOrderDetail. Every line item in Purchasing.PurchaseOrderDetail should tie back to a product ID in Production.Product, so these tables can also be joined in such a way that a match is required in both sides of the join.
3. Try joining Production.ProductSubcategory to the Production.Product table. Not every product will have a subcategory, and we still want to see those that don't, so keep that in mind when deciding what kind of join to use.
4. Finally, join Production.ProductCategory to Production.ProductSubcategory. Again, not every product will have a category, but we still want to see those records in our output, so choose your join type accordingly.

Challenge 2

The Sales data in our AdventureWorks database is structured almost identically to our Purchasing data. It is so similar, in fact, that we can actually align columns from several of the Sales and Purchasing tables to create a unified dataset in which some rows pertain to Sales, and some to Purchasing. Note that we are talking about combining data by columns rather than by rows here – think UNION.

So with that said, your second challenge is to enhance your query from Challenge 1 by "stacking" it with the corresponding Sales data. That may seem daunting, but it is actually WAY easier than it sounds! It turns out that our two Purchasing tables from the Exercise 1 query map to an equivalent Sales table:

- Purchasing.PurchaseOrderDetail maps to Sales.SalesOrderDetail
- Purchasing.PurchaseOrderHeader maps to Sales.SalesOrderHeader

Further, the joins to the product tables work just the same.

So your steps will be:

1. Write an equivalent query to the one you wrote for Exercise 1, except that it pulls data for sales orders rather than purchasing orders.
2. "Stack" this query with your query from Exercise 1 by aligning the queries' respective columns.
3. Alias the two "ID" fields in your query as "OrderID" and "OrderDetailID", respectively. This way ,the field names do not specifically pertain to purchases or sales.
4. Add a string literal column called "OrderType" to both pieces of your query, that outputs a string denoting whether a given row has sales data or purchasing data. It should return "Sale" for the sales data rows, and "Purchase" for the purchasing data rows.
5. Sort the output of your query by order date, in descending order.

Challenge 3

Create a query with the following columns:

11. BusinessEntityID, from **Person.Person**
12. PersonType, from **Person.Person**
13. A derived column, aliased as "FullName", that combines the first, last, and middle names from **Person.Person.**
    - There should be exactly one space between each of the names.
    - If "MiddleName" is NULL and you try to "add" it to the other two names, the result will be NULL, which isn't what you want.
    - You could use ISNULL to return an empty string if the middle name is NULL, but then you'd end up with an extra space between first and last name – a space we *would* have needed *if* we had a middle name to work with.
    - So what we really need is to apply conditional, IF/THEN type logic; if middle name is NULL, we just need a space between first name and last name. If not, then we need a space, the middle name, and then another space. See if you can accomplish this with a CASE statement.
14. The "AddressLine1" field from **Person.Address**; alias this as "Address".
15. The "City" field from **Person.Address**
16. The "PostalCode" field from **Person.Address**
17. The "Name" field from **Person.StateProvince**; alias this as "State".
18. The "Name" field from **Person.CountryRegion**; alias this as "Country".

Only return rows where person type (from Person.Person) is "SP", OR the postal code begins with a "9" AND the postal code is exactly 5 characters long AND the country (i.e., "Name" from Person.CountryRegion) is "United States".

You will probably find it useful to group your criteria with parentheses, and the LEFT and LEN text functions may come in handy for applying the criteria to postal code.

Again, here are a few pointers on getting your joins to work correctly:

1. You won't be able to join Person.Address directly to Person.Person; instead, try joining Person.BusinessEntityAddress to Person.Person, then joining Person.Address to Person.BusinessEntityAddress.
2. You can join Person.StateProvince to Person.Address.
3. Likewise, you can join Person.CountryRegion to Person.StateProvince.
4. You can stick with INNER joins for all of your joins.

Challenge 4

Enhance your query from Exercise 3 as follows:

1. Join in the HumanResources.Employee table to Person.Person on BusinessEntityID. Note that many people in the Person.Person table are not employees, and we don't want to limit our output to just employees, so choose your join type accordingly.
2. Add the "JobTitle" field from HumanResources.Employee to our output. If it is NULL (as it will be for people in our Person.Person table who are not employees, return "None".
3. Add a derived column, aliased as "JobCategory", that returns different categories based on the value in the "JobTitle" column as follows:
    - If the job title contains the words "Manager", "President", or "Executive", return "**Management**". Applying wildcards with LIKE could be a helpful approach here.
    - If the job title contains the word "Engineer", return "**Engineering**".
    - If the job title contains the word "Production", return "**Production**".
    - If the job title contains the word "Marketing", return "**Marketing**".
    - If the job title is NULL, return "NA".
    - If the job title is one of the following *exact strings* (NOT patterns), return "**Human Resources**": "Recruiter", "Benefits Specialist", OR "Human Resources Administrative Assistant". You could use a series of ORs here, but the IN keyword could be a nice shortcut.
    - As a default case when none of the other conditions are true, return "**Other**".

Challenge 5

Select the number of days remaining until the end of the current month; that is, the difference in days between the current date and the last day of the current month.

Your solution should be *dynamic*: it should work no matter what day, month, or year you run it, which means it needs to calculate the end of the current month based on the current date.

This can be a little tricky, so here are some steps you can follow to break the problem down into manageable pieces:

1. First, GET today's DATE.
2. Calculate the first day of the current month by plugging the YEAR and MONTH of today's date into the DATEFROMPARTS function, along with the number of the first day of the month (HINT – this is just as obvious as it seems!)
3. Use DATEADD to "add" a month to the value we derived in step 2 – now we have the first day of *next* month.
4. Now we can use DATEADD one more time, to subtract one day from the first day of next month…which is the *last* day of *this* month!
5. Finally, we can use DATEDIFF to calculate the difference in DAYs between the current date and the end-of-month date.

Your solution is basically going to be a series of nested functions, in which the output of the "inner" functions is used as input to the outer functions. In other words, each step above is going to involve at least one function call, that uses the result of the previous step as input.

This sort of thing can get pretty gnarly pretty quickly, so you may find it useful to put each "step" on a new line of code as you're writing it out, and to test your calculation at each intermediate step. This will also help to ensure that you include a closing parenthesis for each function call, which is easy to lose track of once you start "nesting" lots of functions.