Question 1:

i)

1 <= x <= 31, x is an integer.

This works as the only valid equivalency class as it covers the full input range of this function that meets the specification. I also denoted that the input must be an integer, as a number like 5.4, while in the valid range, would not be considered a valid input by the function.

ii)

x < 1
x > 31
1 <= x <= 31, x is not an integer
x is a char

These cover all of the inputs that would be received as invalid by the specification. First, any number that is out of bounds should be considered invalid. Also, having numbers within the valid range, but are in the form of a decimal (5.4), should not be counted as valid by the function. Lastly, we want to make sure that only numerical data types are accepted, and not letters and special characters, as those would be invalid.

iii)

Condition Less than 1		Condition Greater than 31	
Below boundary 0	Above boundary 1	Below boundary 31	Above boundary 32

Question 2:

i)

7 <= x <= 10, x is number of character

This works as the only valid equivalency class as it covers the full input range of this function that meets the specification. Note that the class is the number of characters rather than what those characters should actually be. This is because the specification doesn't specify which characters are valid, only that the length is in between 7 and 10.

ii)

x < 7, x is number of character
x > 10, x is number of character
x <= 0, x is number of character

This accounts for the invalid inputs for number of characters, making sure that number of characters that were out of bounds of the specification were not accounted for. I also added a test to make sure that the number 0 and values below it somehow couldn't be entered in, just to be extra safe.

iii)

Condition Less than 7		Condition Greater than 10	
Below boundary 6	Above boundary 7	Below boundary 10	Above boundary 11

Question 3:

i)

16 <= x < 60, x is an integer
65 < x <= 70, x is an integer

These are the two bounds that fit the specifications for valid inputs. It covers the full range excluding the years 60-65, and also makes sure that the input is an integer. This is to make sure that an input such as 16.5 won't be accepted, even if it is in the valid range. It is also important that the only data types to be accepted are integers, not letters or floats.

ii)

x < 16, x is an integer
60 <= x <= 65, x is an integer
x > 70, x is an integer
x is anything but an integer

These partition classes cover everything that is out of the scope of the specifications. This includes out of bounds on the main range 16-70, and also tests the exception range that is 60-65. Again, we test that the function fails if anything but an integer is given.

iii)

Condition Less than 16		Condition Greater than 60		Condition Less than 65		Condition Greater than 70	
Below boundary 15	Above boundary 16	Below boundary 59	Above boundary 60	Below boundary 65	Above boundary 66	Below boundary 70	Above boundary 71

Question 4:
Unit testing can be both white box and back box! If the unit tests are written before the code is, then it's a black box test, if I write the test afterwards, then it's a white box test since it's at that point based on the implementation. For example, in my twitter social media platform, I could create a unit test for the number of characters that can go into a tweet. I know the specifications state that a tweet can only have 280 max characters, so before I even write the code I could write tests for that requirement. In terms of a whitebox test, once I write the code for the tweet character limit interaction, I can test more detailed edge cases based on how the code actually implements the checks. This allows me to perform more edge cases with different inputs that I may know will fail or not.

Question 5:
1) I would use multiple different testing frameworks to perform unit tests for my application in several areas. For the actual sending of tweets, I could use unittest to assert that the implementation is sound, such as making sure that malformed tweets can't get processed or that a message has a limit of 280 characters max. I could also use unit testing to test the authentication system with different sets of valid usernames and passwords to see if it can pull from the database correctly.

2) For black box testing, I would be testing the specifications of the system without knowledge of the implementation. I could use something like selenium to test certain UI functionality that wouldn't otherwise be testable through code based unit tests. These could range anywhere from making sure that certain buttons activate the events they are meant to trigger, to ensuring comments get loaded in the correct order. I could also use blackbox testing to test that the authentication system's password policy works by testing the different input boundaries.

3) Since the system is so large, it would help with synchronous development on the same code base. Most version control systems also have some sort of unit tests built in that you can run on pull requests to make sure that what you are merging actually meets the requirements and functionality of what the end product should be. Also, using a software engineering methodology can help drive development forward by making the requirements and objectives clear to each person working on the project. Something like agile with it's scrum methodology can help with building a system that might have changing requirements as development progresses.