Exercise 5 (Recursion and Object-oriented Programming)

Problem 1. (*Sum of Integers*) Implement the function `_sumOfInts()` in `sum_of_ints.py` that takes an integer n as argument and returns the sum $S(n) = 1 + 2 + 3 + \cdots + n$, computed recursively using the recurrence equation

$$S(n) = \begin{cases} 1 & \text{if } n = 1, \\ n + S(n-1) & \text{if } n > 1. \end{cases}$$

```
>_ ~/workspace/exercise5

$ python3 sum_of_ints.py 100
5050
```

Problem 2. (*Bit Counts*) Implement the functions `_zeros()` and `_ones()` in `bits.py` that take a bit string (ie, a string of zeros and ones) s as argument and return the number of zeros and ones in s, each computed recursively. The *number of zeros* in a bit string is 1 or 0 (if the first character is `'0'` or `'1'`) plus the *number of zeros* in the rest of the string; *number of zeros* in an empty string is 0 (base case). The *number of ones* in a bit string can be defined analogously.

```
>_ ~/workspace/exercise5

$ python3 bits.py 1010010010011110001011111
zeros = 11, ones = 14, total = 25
```

Problem 3. (*String Reversal*) Implement the function `_reverse()` in `reverse.py` that takes a string s as argument and returns the reverse of the string, computed recursively. The *reverse* of a string is the last character concatenated with the *reverse* of the string up to the last character; the *reverse* of an empty string is an empty string (base case).

```
>_ ~/workspace/exercise5

$ python3 reverse.py bolton
notlob
```

Problem 4. (*Palindrome*) Implement the function `_isPalindrome()` in `palindrome.py`, using recursion, such that it returns `True` if the argument s is a palindrome (ie, reads the same forwards and backwards), and `False` otherwise. You may assume that s is all lower case and doesn't include any whitespace characters. A string is a *palindrome* if the first character is the same as the last *and* the rest of the string is a *palindrome*; an empty string is a *palindrome* (base case).

```
>_ ~/workspace/exercise5

$ python3 palindrome.py bolton
False
$ python3 palindrome.py madam
True
```

Problem 5. (*Password Checker*) Implement the function `_isValid()` in `password_checker.py` that returns `True` if the given password string meets the following requirements, and `False` otherwise:

- Is at least eight characters long
- Contains at least one digit (0-9)
- Contains at least one uppercase letter
- Contains at least one lowercase letter
- Contains at least one character that is neither a letter nor a number

```
>_ ~/workspace/exercise5
$ python3 password_checker.py Abcde1fg
False
$ python3 password_checker.py Abcde1@g
True
```

Hint: use the `str` methods `isdigit()`, `isupper()`, `islower()`, and `isalnum()`.

Problem 6. (*2D Point*) Define a data type called `Point` in `point.py` that represents a point in 2D. The data type must support the following API:

point.Point

`Point(x, y)`	constructs a point `p` from the given `x` and `y` values
`p.distanceTo(q)`	returns the Euclidean distance between `p` and `q`
`str(p)`	returns a string representation of `p` as `'(x, y)'`

```
>_ ~/workspace/exercise5
$ python3 point.py 0 1 1 0
p1        = (0.0, 1.0)
p2        = (1.0, 0.0)
d(p1, p2) = 1.4142135623730951
```

Problem 7. (*1D Interval*) Define a data type called `Interval` in `interval.py` that represents a closed 1D interval. The data type must support the following API:

interval.Interval

`Interval(lbound, rbound)`	constructs an interval `i` given its lower and upper bounds
`i.lower()`	returns the lower bound of `i`
`i.upper()`	returns the upper bound of `i`
`i.contains(x)`	returns `True` if `i` contains the value `x`, and `False` otherwise
`i.intersects(j)`	returns `True` if `i` intersects interval `j`, and `False` otherwise
`str(i)`	returns a string representation of `i` as `'[lbound, rbound]'`

```
>_ ~/workspace/exercise5
$ python3 interval.py 3.14
0 1 0.5 1.5 1 2 1.5 2.5 2.5 3.5 3 4
[2.5, 3.5] contains 3.140000
[3.0, 4.0] contains 3.140000
[0.0, 1.0] intersects [0.5, 1.5]
[0.0, 1.0] intersects [1.0, 2.0]
[0.5, 1.5] intersects [1.0, 2.0]
[0.5, 1.5] intersects [1.5, 2.5]
[1.0, 2.0] intersects [1.5, 2.5]
[1.5, 2.5] intersects [2.5, 3.5]
[2.5, 3.5] intersects [3.0, 4.0]
```

Problem 8. (*Rectangle*) Define a data type called `Rectangle` in `rectangle.py` that represents a rectangle using 1D intervals (ie, `Interval` objects) to represent its x (width) and y (height) segments. The data type must support the following API:

rectangle.Rectangle

`Rectangle(xint, yint)`	constructs a rectangle `r` given its `x` and `y` segments, each an `Interval` object
`r.area()`	returns the area of rectangle `r`
`r.perimeter()`	returns the perimeter of rectangle `r`
`r.contains(x, y)`	returns `True` if `r` contains the point `(x, y)`, and `False` otherwise
`r.intersects(s)`	returns `True` if `r` intersects rectangle `s`, and `False` otherwise
`str(r)`	returns a string representation of `r` as `'[x1, x2] x [y1, y2]'`

```
>_ ~/workspace/exercise5

$ python3 rectangle.py 1.01 1.34
0 1 0 1 0.7 1.2 .9 1.5
Area([0.0, 1.0] x [0.0, 1.0]) = 1.000000
Perimeter([0.0, 1.0] x [0.0, 1.0]) = 4.000000
Area([0.7, 1.2] x [0.9, 1.5]) = 0.300000
Perimeter([0.7, 1.2] x [0.9, 1.5]) = 2.200000
[0.7, 1.2] x [0.9, 1.5] contains (1.010000, 1.340000)
[0.0, 1.0] x [0.0, 1.0] intersects [0.7, 1.2] x [0.9, 1.5]
```

Files to Submit

1. `sum_of_ints.py`
2. `bits.py`
3. `reverse.py`
4. `palindrome.py`
5. `password_checker.py`
6. `point.py`
7. `interval.py`
8. `rectangle.py`

Before you submit your files, make sure:

- You do not use concepts from sections beyond "Designing Data Types".
- Your code is adequately commented, follows good programming principles, and meets any specific requirements such as corner cases and running times.