Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel$\rightarrow$Restart) and then **run all cells** (in the menubar, select Cell$\rightarrow$Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

```
In [1]: NAME = ""
        COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 5

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form.
- This homework is due on **Tuesday, 19 April 2022** by 11:59pm.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

  `# YOUR CODE HERE`

and you should not modify any other place.
- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  ```
  assert x == 2
  ```

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.

- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

We have seen in the chapter on classes how to implement complex numbers:

In [2]:
```python
import math

class Complex(object):

    def __init__(self, r, i):
        self.r = r # Real part
        self.i = i # Imaginary part

    def __add__(self, other):
        return Complex(self.r + other.r, self.i + other.i)

    def __sub__(self, other):
        return Complex(self.r - other.r, self.i - other.i)

    def __mul__(self, other):
        return Complex((self.r * other.r - self.i * other.i),
                       (self.r * other.i + self.i * other.r))

    @property
    def modulus_square(self):
        return self.r * self.r + self.i * self.i

    @property
    def modulus(self):
        return math.sqrt(self.modulus_square)

    def inverse(self):
        m = self.modulus_square # to cache it
        return Complex(self.r / m, - self.i / m)

    def __truediv__(self, other):
        return self * other.inverse()

    def __repr__(self):
        """This defines how to print a complex number."""
        if self.i < 0:
            return "{}-{}i".format(self.r, -self.i)
        return "{}+{}i".format(self.r, self.i)

    def __eq__(self, other):
        """We even define equality"""
        return self.r == other.r and self.i == other.i
```

There are several ideas above:

- To implement the mathematical operations `+`, `-`, `*`, `/`, between complex numbers, we implement the methods `__add__`, `__sub__`, `__mul__`, `__truediv__`. You can

find more information in the documentation for the Python data model.
- Similarly, to define equality we define `__eq__`, and to define `<` we define `__lt__`.

We will now do something similar to define fractions.

Problem 1: Implement Fractions

- List item
- List item

We want to define a class `Fraction` to represent a fraction, with integers as numerator and denominator.
Similarly to the `Complex` class above, you need to implement the methods necessary to define `+`, `-`, `*`, `/` among fractions, as well as equality.

You will represent fractions in *normal form*, such that:

- numerator and denumerator which do not have common factors (common divisors), except for 1 (of course),
- the denominator is positive.

For example, when you create a fraction via:

```
r = Fraction(8, 6)
```

and then ask for the denominator,

```
r.numerator
```

the result will be 4, and

```
r.denominator
```

will be 3.

To remove the common factors from a fraction m/n, simply compute the greatest common divisor $d = gcd(m, n)$ via Euclid's algorithm (see the chapter on recursion), and reduce the fraction to $(m/d)/(n/d)$.

We advise you to reduce a fraction into normal form directly into the constructor of the class (the `__init__` method).

Here is the code for `Fraction`; we leave the interesting bits for you to do.

In [3]:
```
# Definition of Fraction class

# Here is the gcd function, as it may well be useful to you.
```

```python
def gcd(m, n):
    # This is the "without loss of generality" part.
    m, n = (m, n) if m > n else (n, m)
    m, n = abs(m), abs(n)
    return m if n == 0 else gcd(m % n, n)

class Fraction(object):

    def __init__(self, numerator, denominator):
        assert isinstance(numerator, int)
        assert isinstance(denominator, int)
        assert denominator != 0
        # YOUR CODE HERE
        d = gcd(numerator, denominator)
        self.numerator = int(numerator/d)
        self.denominator = int(denominator/d)
        if denominator < 0:
          self.numerator = -self.numerator
          self.denominator = -self.denominator


    def __repr__(self):
        """Pretty print a fraction."""
        return "{}/{}".format(self.numerator, self.denominator)

    ## Here, implement the methods for +, -, *, /, =, and <.
    ## Done quite at leisure, with spaces and all, this can be done in about
    ## 25 lines of code.
    # YOUR CODE HERE
    def eqdenom(self, other):
      numerator, numerator2 = self.numerator * other.denominator, other.numerator *
      denominator, denominator2 = self.denominator * other.denominator, other.denom
      return (numerator, numerator2, denominator)

    def __add__(self, other):
      if self.denominator == other.denominator:
        numerator = self.numerator + other.numerator
        return Fraction(numerator, self.denominator)
      else:
        numerator, numerator2, denominator = self.eqdenom(other)
        numerator = numerator + numerator2
        return Fraction(numerator, denominator)

    def __sub__(self, other):
      if self.denominator == other.denominator:
        numerator = self.numerator - other.numerator
        return Fraction(numerator, self.denominator)
      else:
        numerator, numerator2, denominator = self.eqdenom(other)
        numerator = numerator - numerator2
        return Fraction(numerator, denominator)

    def __mul__(self, other):
      numerator = self.numerator * other.numerator
      denominator = self.denominator * other.denominator
      return Fraction(numerator, denominator)
    def __truediv__(self, other):
      f1 = self
      return f1 * Fraction(other.denominator, other.numerator)

    def __eq__(self, other):
      return True if self.numerator == other.numerator and self.denominator == othe

    def __lt__(self, other):
      if self.denominator == other.denominator:
        return True if self.numerator < other.numerator else False
      else:
        numerator, numerator2, denominator = self.eqdenom(other)
        return True if numerator < numerator2 else False
```

In [4]:
```
## Here is an example.  Feel free also to use this cell to test your code.

f = Fraction(5, 7)
f2 = Fraction(5, 6)
print(f < f2)
```

```
True
```

Here are some tests.

In [5]:
```
## Tests for creating a fraction.

## First, let us check that you correctly put the fraction into normal form,
## without common factor between numerator and denominator, and with a
## positive denominator.

f = Fraction(8, 6)
assert f.numerator == 4 and f.denominator == 3

f = Fraction(-8, 6)
assert f.numerator == -4 and f.denominator == 3

f = Fraction(8, -6)
assert f.numerator == -4 and f.denominator == 3

f = Fraction(-8, -6)
assert f.numerator == 4 and f.denominator == 3

f = Fraction(0, 10)
assert f.numerator == 0 and f.denominator == 1
```

In [6]:
```
### 5 points: hidden tests for fraction creation.
```

In [7]:
```
## tests for fraction operations.

f = Fraction(8, 6) + Fraction(25, 20)
assert f.numerator == 31 and f.denominator == 12
assert f == Fraction(31, 12)
assert f == Fraction(62, 24)
assert Fraction(6, 4) + Fraction(-8, 6) == Fraction(6, 4) - Fraction(8, 6)
assert not (Fraction(6, 4) + Fraction(-8, 6) == Fraction(6, 5) - Fraction(8, 6))
```

In [8]:
```
## 10 points: Hidden tests for fraction operations.
```

In [9]:
```
## more tests for fractions operations.

assert Fraction(3, 2) * Fraction(2, 3) == Fraction(1, 1)
assert Fraction(3, 2) / Fraction(2, 3) == Fraction(9, 4)
assert Fraction(3, 2) / Fraction(6, 4) == Fraction(1, 1)
assert Fraction(32, 16) == Fraction(2, 1)
assert not Fraction(33, 16) == Fraction(4, 2)
```

In [10]:
```
## 10 points: More hidden tests for fraction operations.
```

In [11]:
```
## tests for fraction comparison.

assert Fraction(5, 7) < Fraction(5, 6)
assert Fraction(-3, 2) < Fraction(0, 3)
```

In [12]:
```
## 10 points: hidden tests for fraction comparisons.
```

In [13]:
```
## Let's check you leave things unchanged.

a = Fraction(7, 8)
b = Fraction(-4, 5)
a + b
a / b
a < b
a * b
assert a == Fraction(7, 8)
assert b == Fraction(-4, 5)
```

In [14]:
```
## And finally, some random tests.

import random
for _ in range(1000):
    a = Fraction(random.randint(-200, 200), random.randint(1, 100))
    b = Fraction(random.randint(-200, 200), random.randint(1, 100))
    c = Fraction(random.randint(-200, 200), random.randint(1, 100))
    assert Fraction(-1, 1000) < (a - b) * (a - b)
    assert (a - b) * (a + b) == a * a - b * b
    z = Fraction(0, 1) # Zero, as a fraction.
    if not ((a == z) or (b == z) or (c == z)):
        assert (a / b) * b == (a / c) * c
        assert (a / b) * (a / c) == (a * a) / (b * c)
        assert (a / b) / (b / c) == (a * c) / (b * b)
        assert (a * a * b * c) / (a * c) == a * b
```

Question 2: An Int class

To define the value 7, you can write `Fraction(14, 2)` or `Fraction(7, 1)` (it's the same), but this is a bit inconvenient. Write a subclass `Int` of `Fraction`, so that `Int(7)` generates a fraction with value 7.

In [15]:
```
class Int(Fraction):

    # YOUR CODE HERE
    def __init__(self, num):
      self.numerator = num
      self.denominator = 1
```

In [16]:
```
## You can test your solution here.

print(Int(7))
```

```
7/1
```

And now for some tests.

In [17]:
```
## tests for the int class.

assert Int(3) / Int(2) == Fraction(3, 2)
assert Int(3) * Int(4) / (Int(5) + Int(2)) == Fraction(12, 7)
assert Int(3) * Int(4) / (Int(5) + Int(1)) == Fraction(2, 1)
```

In [18]:
```
## 10 points: hidden tests for the Int class
```