Problem 1 (5points, 1 point each)

1.     Classify each public method of RatNum as a creator, observer, producer or mutator.

Creators: RatNum constructors. valueOf
Observers: isNan, isNegative, isPositive, compareTo, doubleValue, intValue, floatValue, longValue, hashCode, equals
Producers: negate, add, sub, mul, div.
Mutators: None. RatNum is immutable.

GRADING: Small errors (e.g., wrongly classified a producer as a mutator) take -0.5. If mostly correct, give 0.5 points.

2.    add, sub, mul, and div all require that "arg != null". This is because all of the methods access fields of 'arg' without checking if 'arg' is null first. But the methods also access fields of 'this' without checking for null; why is "this != null" absent from the requires-clause for the methods?

Because if the method is running, then the receiver "this" is guaranteed to be non-null. Had the method been called on a null receiver, e.g., x.mul(y) where x is null, Java would have thrown a runtime exception and never descending into mul.

GRADING: Either correct or incorrect.

3.     Why is RatNum.valueOf(String) a static method? What alternative to static methods would allow one to accomplish the same goal of generating a RatNum from an input String?

valueOf is static because it makes no use of concrete state of a "this" RatNum. It is a class method and can be called without an instance variable. Similarly to valueOf(String) in all Boxed primitive classes in Java. One alternative is a constructor that takes a String.

GRADING: Give partial credit for reasonable answers. If they just say so it can be called without an instance, but don't give an alternatives, deduct 0.5 points.

4.    add, sub, mul, and div all end with a statement of the form return new RatNum ( numerExpr , denomExpr);.
      Imagine an implementation of the same function except the last statement is:

```
this.numer = numerExpr;
this.denom = denomExpr;
return this;
```

    For this question, pretend that the this.numer and this.denom fields are not declared as final so that these assignments compile properly. How would the above changes fail to meet the specifications of the function (Hint: take a look at the @requires and @modifies statements, or lack thereof.) and fail to meet the specifications of the RatNum class?

The spec requires that nothing is modified. (The RatNum is immutable.)
This change would violate the immutability requirement.

GRADE: Either right or wrong.

5.  Calls to checkRep are supposed to catch violations in the classes' invariants. In general, it is recommended that one call checkRep at the beginning and end of

every method. In the case of RatNum, why is it sufficient to call checkRep only at the end of the constructors? (Hint: could a method ever modify a RatNum such that it violates its representation invariant? Could a method change a RatNum at all? How are changes to instances of RatNum prevented?)

Because the RatNum methods, except for the constructors, are immutable. In general it's a good idea to do checkRep even in immutable methods (to make sure there is no accidental mutation of "this"). But in this case it's pointless because the methods are short and it is clear they do not mutate "this".

GRADE: They should something about RatNum being immutable or that the fields are final and can't be altered after construction.

PROBLEM 2: (4 points, 2 points each)

1. We have chosen the array representation of a polynomial: RatNum[] coeffs, where coeffs[i] stores the coefficient of the term of exponent i. An alternative data representation is the list-of-terms representation: List<Term> terms, where each Term object stores the term's RatNum coefficient and integer exponent. The beauty of the ADT methodology is that we can switch from one representation to the other without affecting the clients of our RatPoly! Briefly list the advantages and disadvantages of the array representation versus the list-of-terms representation.

Short answer: arrays allow for very easy implementation of polynomial arithmetic. list-of-terms is a lot more memory efficient.

GRADING: Give points for reasonable arguments. Should mention efficiency of operations vs storage.

2. Where did you include calls to checkRep (at the beginning of methods, the end of methods, the beginning of constructors, the end of constructors, some combination)? Why?

Short answer: at the end of constructors. For the same reason as RatNums. RatPoly is immutable. This question is duplicated in part 1. They don’t have to answer it twice.

GRADING: Give points if they put the checkRep() calls somewhere reasonable: end of constructor, in mutators. Eyc.

Problem 3: (5 points)

Pseudocode (2 points)
This pseudocode is from https://en.wikipedia.org/wiki/Polynomial_long_division and https://math.stackexchange.com/questions/215734/pseudo-code-for-polynomial-long-division#215792

```
# precondition degree(d) < degree(this) && d != 0
function this / d
  if this == 0
     return 0
  q = 0
  r = this
  # this doesn’t have to be included in pseudocode
  # LI: this = d × q + r &&
        (r != 0 || r == 0) &&
        (degree(r) >= degree(d) || degree(r) < degree(d))

  while r != 0 AND degree(r) >= degree(d):
     t = lead(r)/lead(d)     # Divide the leading terms
     q = q + t
     r = r - t * d
  return q
  # postcondition: this = q*d + r
```

Grading: give points for reasonable attempts. It should be pseudocode. Take off a point if they just pasted their Java code. They don’t have to state pre or postconditions, or LI here.

Partial correctness: (3 points)

based on the version above

```
LI: this = d × q + r &&
        (r != 0 || r == 0) &&
        (degree(r) >= degree(d) || degree(r) < degree(d))
```

Expressing the loop conditions in the LI is tricky because for example r == 0 at the last iteration, but this = d × q + r must hold at exit. If they don't include anything about loop conditions, don't deduct points. Concentrate on how they express this part: this = d × q + r.

Give points if the invariant makes some sort of sense. When in doubt, ask me.

```
Base: q == 0, r == this;
      r = 0 + this = this

      degree(r) = degree(this)
      degree(d) < degree(this) by precondition
      therefore degree(d) <= degree(r)

      if r == 0 -> this == 0 and we already had a trivial result q = 0 and done,
therefor r != 0.
```

Induction:
Assume: this = q(k)*p + r(k); q(k) is the value of q at step k
step k+1
q(k+1)*p = (q(k)+t)*p
r(k+1) = r(k) - t*p
q(k+1)*p + r(k+1) = q(k)*p + t *p + r(k) - p*t = q(k)*p + r(k) = this

r(k) != 0 or we would exited, so either r(k+1) != 0 or r(k+1) == 0 and this is our last iteration.
degree(r(k)) >= degree(this) or we would have exited, so at
iteration k+1, either degree(r(k+1)) >= degree(this) or degree(r(k+1)) < degree(this) and we're done.

Simplified version of induction leaving off loop conditions:

Base case:
q == 0, r == this;
r = 0 + this = this

Induction:
Assume: this = q(k)*p + r(k); q(k) is the value of q at step k
step k+1
q(k+1)*p = (q(k)+t)*p
r(k+1) = r(k) - t*p

q(k+1)*p + r(k+1) = q(k)*p + t *p + r(k) - p*t = q(k)*p + r(k) = this

Grading: This can be tricky because students can come up with many different Lis. Give points for reasonable attempts. Induction should match pseudocode.

Collaboration and Reflection (0.5 points each)

Grading: give points for reasonable answers.