PROJECT 3, Com S 228, Spring 2020

Due at 11:59pm, Monday, March 30

1 Prime Factorization

A *prime number* is an integer greater than one and is divisible by one and itself only. The sequence of prime numbers starts with $2, 3, 5, 7, 11, 13, 17, 19, \ldots$. There are infinitely many primes. The current largest known prime is $2^{82589933} - 1$.

An integer greater than one that is not prime is *composite*. For example, 4 is composite because it is also divisible by 2 in addition to 1 and itself. By definition, 1 is neither a prime number nor a composite number.

The *fundamental theorem of arithmetic* states that every integer greater than one can be uniquely *factored* into a product of one or more primes. For example, $25480 = 2^3 \cdot 5 \cdot 7^2 \cdot 13$. More formally, any integer $n \geq 2$ can be written in the form

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k},$$

where $p_1 < p_2 < \cdots < p_k$ are the prime factors, and the exponents $\alpha_1, \alpha_2, \ldots, \alpha_k > 0$ are their respective `multiplicities`. For example, 2 has multiplicity 3 in the factorization of 25480.

1.1 Factorization and Encryption

Prime factorization is widely believed to be a very difficult computational problem. No efficient prime factorization algorithm for large numbers is known. More specifically, there is no known algorithm that can factor all integers in polynomial time, i.e., factor d-digit numbers in time $O(d^k)$ for some constant k.

In 2009, several researchers successfully concluded two years of effort to factor the following 232-digit number (named RSA-768) via the use of hundreds of machines:

12301866845301177551304949583849627207728535695953347921973224521517264005072636575187452021997864693899564749427740638459251925573263034537315482685079170261221429134616704292143116022212404792747377940806653514195974598569021431 3

=

33478071698956898786044169848212690817704794983713768568912431388982883793878002287614711652531743087737814467999489

×

36746043666799590428244633799627952632279158164343087642676032283815739666511279233373417143396810270092798736308917

The presumed difficulty of prime factorization lies at the foundation of cryptosystems such as RSA that are used for secure data transmission. In such a system, a user publishes a large integer, their

public key, while holding onto another large integer, their *private key*. Messages sent to the receiver are encrypted using the public key, but are presumed impossible to decrypt without the knowledge of the private key, because doing so would require prime factorization of very large numbers — even larger than RSA-768.

1.2 Direct Search Factorization

In this project we will use a brute force strategy named the *direct search factorization*. It is based on the observation that a number n, if composite, must have a prime factor less than or equal to $\sqrt{n}$. Clearly, the number cannot have more than one prime factor greater than $\sqrt{n}$.

This method initializes $m \leftarrow n$ and systematically tests divisors d from 2 up to $\lfloor\sqrt{n}\rfloor$.[1] In your implementation, test the condition $d \cdot d \leq n$ instead of $d \leq \sqrt{n}$ for efficiency. If d divides m (i.e., is a factor of m), then set $m \leftarrow m/d$, and repeat this check with the divisor d until it no longer divides m. Then increment d to continue the testing until $d \cdot d > m$.

For example, to factorize 25480, we start by trying to divide the number by 2. Since 2 divides 25480, we record it and divide 25480 by 2 to obtain 12740. We find that the number 12740 can be divided by 2 two more times. The original number thus contains the prime factor 2 with multiplicity 3. Now we work on 25480 / 8 = 3185. The next number to test is 3, which does not divide 3185; neither does 4. The number 5 divides 3185 only one time, yielding the quotient 637. Moving on, 6 does not divide 637. We find that 7 divides 637 twice to yield 13. The algorithm terminates since $8 \cdot 8 > 13$.

You can cut down the number of test candidates by half by ignoring all the even numbers greater than 2 (because they are apparently not prime).

2 List Structure

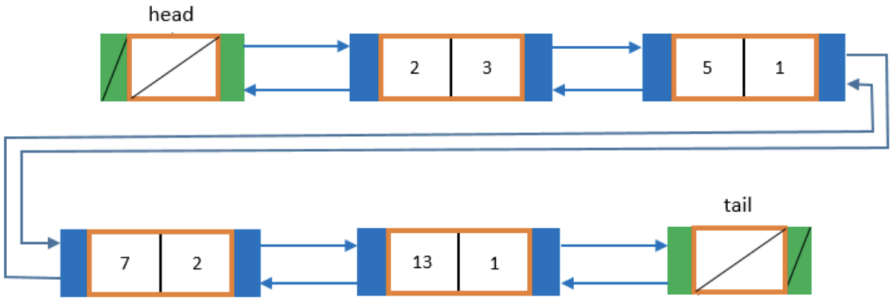
The class `PrimeFactorization` houses a doubly-linked list structure to store the prime factors of a number. Every node on the list stores a **distinct** prime factor and its multiplicity, which are together packaged into an object of the `PrimeFactor` class.

A node is implemented by the private class `Node` inside `PrimeFactorization`. The class `Node` also has two links, `previous` and `next`, to reference the preceding and succeeding nodes.

For example, 25480 has the prime factor 2 with multiplicity 3, so the `PrimeFactorization` list for 25480 will have a `Node` whose data is a `PrimeFactor` object whose fields `prime` and `multiplicity` take values `2` and `3`, respectively.

In the linked list inside the class `PrimeFactorization`, the nodes are connected by the `next` link in the increasing order of prime factor. The list also has two dummy nodes: `head` and `tail`. The factorization of 25480 is represented by the list below with six nodes, four of which represent its prime factors 2, 5, 7, 13 with multiplicities 3, 1, 2, 1, respectively.

[1] The symbols $\lfloor\ \rfloor$ denote the *floor* operator, which gives the largest integer less than or equal to the operand. For example, $\lfloor\pi\rfloor = 3$ and $\lfloor 5\rfloor = 5$.



The class `PrimeFactorization` also stores the factored number in the instance variable `value` of the `long` type. If this number exceeds the maximum $2^{63} - 1$ representable by the type `long`, `value` is set to be `-1` (defined to be a constant `OVERFLOW` in the template code). For this reason, we make a distinction between `value` and the object's *represented integer*. These two integers are equal when `value != -1`.

This class has four constructors:

- `public PrimeFactorization()` is a default constructor that constructs an empty list to represent the number 1.
- `public PrimeFactorization(long n) throws IllegalArgumentException` performs the direct search factorization described in Section 1.2 on the integer parameter `n`. An exception will be thrown if `n < 1`.
- `public PrimeFactorization(PrimeFactorization pf)` is a copy constructor which assumes that the argument object `pf` stores a correct prime factorization.
- `public PrimeFactorization(PrimeFactor[] pfList)` takes an array of `PrimeFactor` objects. This is especially useful for the case of an overflowing number.

3 Arithmetic Operations

The list structure of this `PrimeFactorization` object needs to be updated when its represented integer is multiplied with or divided by another integer, either of type `long` or represented by another `PrimeFactorization` object. After every arithmetic operation, the nodes in the linked list must remain in the increasing order of the prime factors they store.

3.1 Multiplication

There are three methods to carry out multiplication, each of which must update the linked list of the `PrimeFactorization` object that the method belongs to.

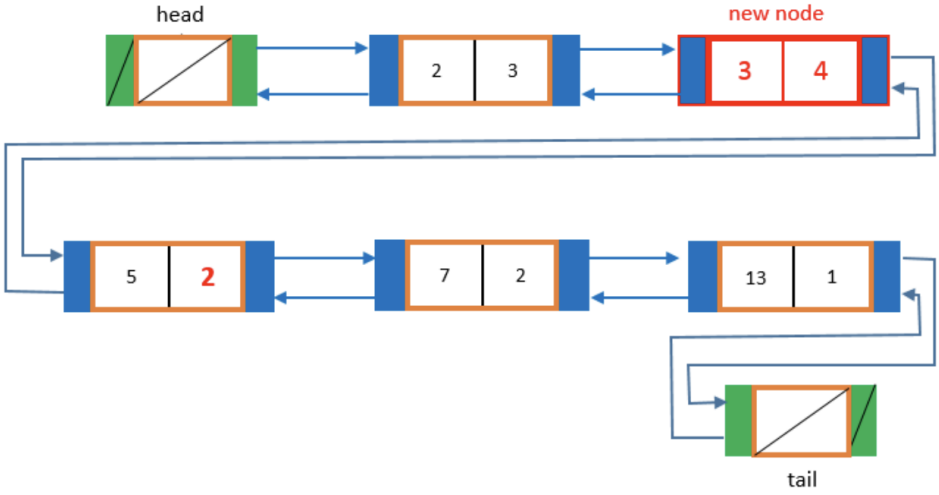
- `public void multiply(long n) throws IllegalArgumentException` can be implemented via a traversal of the linked list referenced by `head` using an iterator (which is an object of the private class `PrimeFactorizationIterator`) as follows.
  Factor `n` using the direct search described in Section 1.2. For each prime factor found (with a multiplicity), move the iterator forward from its current position to the first node storing an equal or larger prime. In the first case, increment the `multiplicity` field of that node. In the second case, create a new node for this factor and add it to the linked list before the

larger prime. If the cursor reaches the end of the list, simply add the new node as the `last` node.

- `public void multiply(PrimeFactorization pf)` updates the linked list by traversing it and the linked list of the argument object `pf` simultaneously. Essentially, it merges a copy of the second list into the first list by doing the following on the first list:
  - Copy over the nodes from the second list corresponding to prime factors that do not appear in the first list.
  - For each prime factor shared with the second list, increase the `multiplicity` field of the `PrimeFactor` object stored at the corresponding node in the first list.

  The implementation requires the use of two iterators, one for each list, and works in a way similar to merging two sorted arrays into one.

  For example, suppose a `PrimeFactorization` object for 25480 (with the linked list shown in Section 2) calls `multiply()` on another object storing the factorization of 405, which is $3^4 \cdot 5$. After the call, the first linked list will become:



  A new node (drawn in red) has been added, and the `PrimeFactor` object stored at an existing node has its `multiplicity` field updated from `1` to `2` (shown in red).
- `public static PrimeFactorization multiply (PrimeFactorization pf1, PrimeFactorization pf2)` is a static method that takes two `PrimeFactorization` objects and returns a new object storing the product of the two numbers represented by the first two objects.

The first two `multiply()` methods also call `Math.multiplyExact()` to carry out the multiplication of `this.value` (if not `-1`) with `n` or `pf.value`. An `ArithmeticException` will be thrown if an overflow happens. In this case, set the value field to `-1`. Otherwise, store the product in `value`. Note that `this.value == -1` only means that the number represented by the linked list cannot be represented in Java as a `long` integer. We can still perform arithmetic operations on the large number by this `PrimeFactorization` object.

3.2 Division

The `PrimeFactorization` class provides three methods to perform division — by an integer, by the integer encapsulated in another `PrimeFactorization` object, and of the first such object by a second object. The linked list is updated to store the result.

First, the `public boolean dividedBy(long n)` immediately returns `false` if `this.value != -1` and `this.value < n`. Otherwise, it constructs a `PrimeFactorization` object for `n`, and then calls the second `dividedBy()` method with the constructed object as the argument.

Second, `public boolean dividedBy(PrimeFactorization pf)` immediately returns `false` if

`this.value != -1 && pf.value != -1 && this.value < pf.value`, or if

`this.value != -1 && pf.value == -1`.

If the dividend and the divisor have equal values, it removes all the nodes storing prime factors by calling the private method `clearList()`. In this case, it sets value to `1` and returns `true`.

Otherwise, it makes a copy of the linked list of this object and does the following.

Traverse the list `copy` and the list of the object `pf` using two separate iterators, say, `iterCopy` and `iterPf`, respectively. Advance `iterCopy.cursor` until

`iterCopy.cursor.pFactor.prime >= iterPf.cursor.pFactor.prime`

or

`(!iterCopy.cursor.hasNext() \&\& iterPf.hasNext())`.

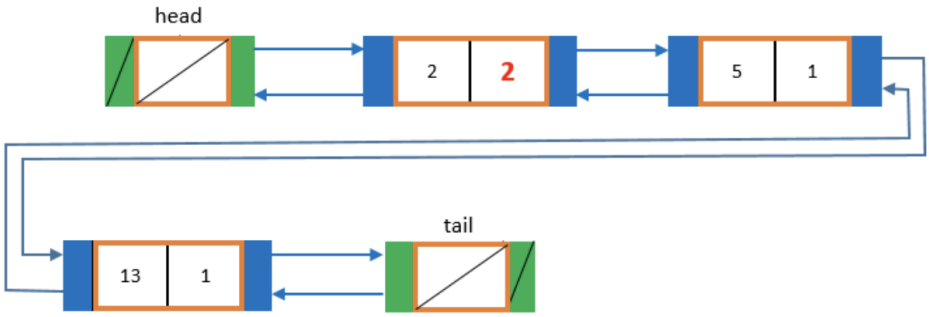
In the latter case, return `false` immediately. In the former case, there are three possible situations:

- `iterCopy.cursor.pFactor.prime > iterPf.cursor.pFactor.prime`. Then the number is not divisible by the number represented by `pf`, so return `false` immediately.
- `iterCopy.cursor.pFactor.prime == iterPf.cursor.pFactor.prime` but `iterCopy.cursor.pFactor.multiplicity < iterPf.cursor.pFactor.multiplicity`. Again, the number is not divisible by the number represented by `pf`, so return `false` immediately.
- Otherwise, the following two conditions must hold:
  `iterCopy.cursor.pFactor.prime = iterPf.cursor.pFactor.prime`
  `iterCopy.cursor.pFactor.multiplicity >= iterPf.cursor.pFactor.multiplicity`
  In this case, reset `iterCopy.cursor.pFacor.multiplicity` to the difference between itself and `iterPf.cursor.pFactor.multiplicity`. If the difference is zero, the node needs to be removed. Advance both iterators.

Repeat until either `false` is returned or `iterPf.cursor == iterPf.tail` eventually holds. If the latter happens, the represented integer of this object is divisible by that of `pf`. Set the `copy` list to be the linked list of this object by updating its head and tail. Also, update the object's `size` and `value` fields accordingly. Return `true` after the updates.

Example: Suppose the earlier `PrimeFactorization` object representing 25480 calls `divideBy(98)`. A `PrimeFactorization` object for $98 = 2 \cdot 7^2$ is first constructed with a linked list containing four nodes: `head`, a node whose data is a `PrimeFactor` object with `prime` value 2 and `multiplicity` value 1, a node whose data is a `PrimeFactor` object with `prime` value 7 and `multiplicity` value 2, and `tail`.

Then, this temporary object is provided as the argument for a call to the second `divideBy()` method. The new linked list for the original object is shown below with the `multiplicity` of 2 decreased to 2 (colored red) and the node for the factor 7 deleted.



In the case that this object represents a number that overflowed before the division, which is indicated by `this.value == -1`, the quotient to replace this number in the object might not overflow. When that happens, the `value` field should store the quotient instead of `-1`. Checking can be done at the end of `divideBy()` using a method called `updateValue()`. Initialize a variable to have value `1`. Keep multiplying it with the power of every prime factor (determined by its multiplicity) stored in the new linked list representing the quotient. Use `Math.multiplyExact()` to carry out the multiplications, which should stop as soon as the method throws an `ArithmeticException`.

Finally, `public static PrimeFactorization dividedBy(PrimeFactorization pf1, PrimeFactorization pf2)` is a static method that takes two `PrimeFactorization` objects and returns a new object storing the quotient of the first corresponding number divided by the second number, or `null` if the first number is not divisible by the second one.

4 Greatest Common Divisor and Least Common Multiple

The class `PrimeFactorization` provides methods to compute the greatest common divisor of two positive integers and return the result as a `PrimeFactorization` object. The linked list in a returned `PrimeFactorization` object must have its nodes store prime factors in the increasing order.

4.1 The Euclidean Algorithm

A `common divisor` d of two positive integers a and b is a positive integer that divides both a and b. The `greatest common divisor (GCD)` of a and b is the biggest of all their common divisors. For example, 12 and 42 have four common divisors 1, 2, 3, 6. Their GCD is 6.

The GCD of two numbers can be efficiently computed using the *Euclidean algorithm*, named for the Greek mathematician Euclid, who recorded it in his treatise *Elements* circa 300 BCE.

Suppose we are to find the GCD of 184 and 69. We divide the bigger number by the smaller one, obtaining

$$184 = 2 \cdot 69 + 46\,.$$

The remainder 46 is less than the divisor 69, and any common divisor of 184 and 69 must also divide

$$46 = 184 - 2 \times 69\,.$$

Therefore, the GCD of 184 and 69 is also the GCD of 69 and 46. Next, we divide 69 by 46, yielding

$$69 = 1 \cdot 46 + 23\,.$$

Thus, the problem further boils down to finding the GCD of 46 and 23. One more round of division results in

$$46 = 2 \cdot 23 + 0\,.$$

Since the remainder becomes 0, 23 divides 46. We have thus obtained the GCD:

$$GCD(184, 69) = GCD(69, 46) = GCD(46, 23) = 23\,.$$

In general, suppose we want to find the GCD of two numbers a and b, with $a > b$. Let $r_0 = a$ and $r_1 = b$.

The Euclidean algorithm carries out the following sequence of divisions:

$$\begin{aligned} r_0 &= q_1 \cdot r_1 + r_2 && (0 < r_2 < r_1) \\ r_1 &= q_2 \cdot r_2 + r_3 && (0 < r_3 < r_2) \\ r_2 &= q_3 \cdot r_3 + r_4 && (0 < r_4 < r_3) \\ &\vdots \\ r_{k-1} &= q_k \cdot r_k + 0\,. \end{aligned}$$

The dividend and divisor in a division were respectively the divisor and remainder in the previous division. Since the remainder is always less than the divisor, it follows that

$$r_1 > r_2 > \cdots > r_k > r_{k+1} = 0\,.$$

The algorithm will terminate because the remainder decreases by at least one after each division step. It holds that

$$GCD(r_0, r_1) = GCD(r_1, r_2) = \cdots = GCD(r_{k-1}, r_k) = r_k\,.$$

The following static method implements the Euclidean algorithm:

```
public static long Euclidean(long m, long n) throws IllegalArgumentException
```

The `Euclidean()` method must be used by the `gcd()` method below to compute the GCD of `this.value` and another number `n` when `this.value != -1`.

```
public PrimeFactorization gcd(long n) throws IllegalArgumentException
```

5 Computing GCD from Prime Factorizations

If the second number is encapsulated in a `PrimeFactorization` object, the GCD computation is carried out as follows.

- Traverse the linked lists of the two objects to find the common prime factors of the two numbers.
- For every common prime factor p, create a new node to store p and $\min(\alpha_1, \alpha_2)$, the minimum of the multiplicities α_1 and α_2 of p stored in the two lists.

- Link all the new nodes together to construct a `PrimeFactorization` object to represent the GCD.

The work can be done during one round of simultaneous traversals of the linked lists of both `PrimeFactorization` objects. The above procedure is implemented by the following method:

```
public PrimeFactorization gcd(PrimeFactorization pf)
```

Even when one or both of the numbers cause overflows, their GCD may not. Thus, if `this.value == -1` or `pf.value == -1`, call `updateValue()` to check if the GCD overflows, and set the value field of the generated `PrimeFactorization` object properly.

When `this.value != -1` and the second number for a GCD computation is provided as an integer, an alternative to calling the first `gcd()` method (described in Section 4.1) would be to construct a `PrimeFactorization` object `pf` over the second number, and call the second `gcd()` method above. However, it is more efficient to directly use the first `gcd()` method, even though it returns a `PrimeFactorization` object constructed through factoring the GCD. This factorization is less expensive than factoring `n`, which has to be done to construct the object `pf` to call the second `gcd()` method with. The reason is that `n` could be significantly larger than the GCD.

The third GCD method is static and takes two `PrimeFactorization` objects.

```
public static PrimeFactorization gcd(PrimeFactorization pf1,
  PrimeFactorization pf2)
```

6 Iterators

The iterator class `PrimeFactorizationIterator` implements `ListIterator<PrimeFactor>`. Refer to the Javadocs of these methods, especially `add()` and `remove()`.

7 Submission

Write your classes in the `edu.iastate.cs228.hw3` package. Turn in the zip file, not your class files. Please follow the submission guidelines posted on Canvas. You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students. Include the Javadoc tag @author in every class source file you have made changes to. Your zip file should be named `Firstname_Lastname_HW3.zip`