Introduction

When we represent a number on a computer we must allocate a fixed number of bits to it. The number of bits dictates the range of numbers that can be stored. This data limitation applies to all data types, and prevents us from storing large numbers.

One way to represent a number is in terms of its digits, similar to how `int` is stored in terms of bits, or how a `String` represents a sequence of characters. In this assignment you will represent large integral numbers in a linked list representation.

Instead of storing bits, we store individual digits (0-9) in a single node of such a list. We can store them in any order (corresponding to big-endian and little-endian in bit representation). For example, the number `32411` can be stored as `3 -> 2 -> 4 -> 1 -> 1` or `1 -> 1-> 4 -> 2 -> 3`.

Given an integral number, we can shift its digits to the left or right. For example `32411` can be "left-shifted" to get the number `324110`. Thus left-shifting by one position is equivalent to multiplying the number by 10 (similar to how left-shifting by one bit in a binary representation multiplies the number by 2). Similarly, `32411` can be "right-shifted" to get the number `3241`, which is the integer-division by 10.

We can support basic addition of a single digit to a number. For example `324115 + 7 = 324122`. Shifting and adding single digits can allow us to create arbitrarily large numbers, one digit at a time. For example `32411` can be created by:

1. Start with `0`.
2. Left-shift by 1 position, and add 3.
3. Left-shift by 1 position (to get 30) and add 2.
4. Left-shift by 1 position (to get 320) and add 4.
5. Left-shift by 1 position (to get 3240) and add 1.
6. Left-shift by 1 position (to get 32410) and add 1.

Numbers can also be added by using simple arithmetic: start from the right-most digits and add them. Record the sum and carry, and add the carry to the next pair of digits, and so on. Note that the numbers may be of different lengths.

2 What to do

All your interfaces and classes should be in the bignumber package. The tests should be in the default package.

Design an interface `BigNumber` that defines the above operations, specifically with the following method signatures (methods do not return anything unless explicitly stated):

1. A method `length` that returns the number of digits in this number.
2. A method `shiftLeft` that takes the number of shifts as an argument and shifts this number to the left by that number. A negative number of left-shifts will correspond to those many right shifts.
3. A method `shiftRight` that takes the number of shifts as an argument and shifts this number to the right by that number. The number `0` can be right-shifted any positive number of times, yielding the same number `0`. A negative number of right-shifts will correspond to those many left shifts.
4. A method `addDigit` that takes a single digit as an argument and adds it to this number. This method throws an `IllegalArgumentException` if the digit passed to it is not a single non-negative digit.
5. A method `getDigitAt` that takes a position as an argument and returns the digit at that position. Positions start at 0 (rightmost digit). This method throws an `IllegalArgumentException` if an invalid position is passed.
6. A method `copy` that returns an identical and independent copy of this number.
7. A method `add` that takes another `BigNumber` and returns the sum of these two numbers. This operation does not change either number.
8. A method to compare two `BigNumber` objects for ordering, using the `Comparable` interface.
9. A method to determine if two numbers are the same.

Now implement this interface in a class `BigNumberImpl`. This implementation represents non-negative numbers of arbitrary lengths. Beyond implementing the `BigNumber` interface, this implementation should have the following features/obey these constraints:

1. You are not allowed to use any of Java's existing list implementations, interfaces, **arrays** or otherwise any collection classes or maps to implement big numbers. You must create your own list implementation. The implementation may be customized for this application. You may use existing list implementations in your tests.
2. You are not allowed to use the `BigInteger` or any similar existing classes from JDK in your implementation. You may use `BigInteger` in your tests.
3. This class should have a constructor with no parameters that initializes this number to `0`.

4. This class should have another constructor that takes a number as a string and represents it. This constructor should throw an `IllegalArgumentException` if the string passed to it does not represent a valid number.
5. This representation should not contain any unnecessary digits. That is, if representing a 5-digit number, there should be exactly 5 digits represented in this object.
6. This class should include a `toString` method that returns a string representation of this number, as simply the number itself.
7. Your implementation should be reasonably efficient. There are several ways you could lose points for efficiency: if tests time out because of an inefficient solution, if your implementation is an order of magnitude less efficient than what is possible for another implementation that obeys all the above constraints, if your implementation regularly encounters a stack overflow error for numbers that have a few hundred digits, etc. Testing your implementation with large inputs is a good way to verify whether it is efficient.

2.1 Hints

Start by creating blank interfaces and classes. Now select an operation and implement it end-to-end (add it to the interface, add empty implementation, tests). Now start with another operation.

Think about how the double-dispatch technique may help you to implement `add`.

2.2 Higher order functions

It is highly recommended that you consider higher-order functions when implementing operations. We will give a small amount of extra credit if you implement them correctly. In order to accomplish this, review the higher order functions, think about how they are applicable to what you are doing in this assignment, and implement them accordingly. Please review the higher-order functions from the lists lectures here and here. You will find various interfaces in the `function` package Links to an external site. useful to implement higher-order functions (similar to how we used `Predicate<T>` in the list lecture).

Write tests that thoroughly test this implementation. As always, it is recommended to write the test *before* completing the `BigNumberImpl` implementation. Be sure to consider special/edge cases!

3 Criteria for grading

You will be graded on:

1. The design of your interface.

2. Whether your implementation obeys all the above constraints.
3. The design of your implementation.
4. The correctness of your methods.
5. The quality of your code (documented and well-arranged methods that use helpers appropriately and are not long and unwieldy, usage of access modifiers, interface names, etc.)
6. The quality and coverage of tests.
7. The quality of documentation.
8. The style of your code.