Approximate Comparison in Java

In Java application, we often need to compare and sort different Objects defined by users. Commonly, this step is usually done by implementing the *Comparable* interface, and overriding its *compareTo* method. Sometimes the comparison may cost us lots of time because of its highly complex calculation. However, maybe we could recognize another solution to optimize this method, with lower accuracy requirements. In other words, we may sacrifice some accurate rate to some extent, to obtain a faster calculation. During this project, I'd like to search some actual Java applications and try to prove the above theory.

Application I: *BigInteger* and *BigDecimal* in Java

As we know, Java supports various types of data, such as *byte, int, long, float, double* etc. And each type has its own storage size, the most significant being the double and long, both of which occupy 64 bits in the memory. While in the real engineering projects, we need even larger and more precise quantitative values to assist us in fulfilling our requirements. Therefore Java JDK provides us two more value types: *BigInteger* and *BigDecimal*, both of which are packaged in the *com.java.math* and provide us much larger numbers. These two classes also implement the *Comparable* interface and override the *compareTo* methods. So we could try to find whether these methods could be improved to perform a much faster running speed. To begin with, let's have a short brief with these classes. Because it's necessary to get a throughout understanding about the rationale and mechanism before putting our hand to reconstruct the codes.

BigInteger: this class provides us an immutable arbitrary-precision integers. It follows the semantics of arithmetic operations exactly, as defined in *The Java Language Specification.* For example, division by zero would throw an *ArithmeticException,* and division of a negative by a positive yields a negative (or zero) remainder. All operations behave were represented in 2's-complement notation and in a big-endian style. Besides, the basic operations such as plus, minus, multiply and division are fully supported by the class, even the left shift and right shift (these operations are implemented by methods, but not overloading the '>>' and '<<' operators).

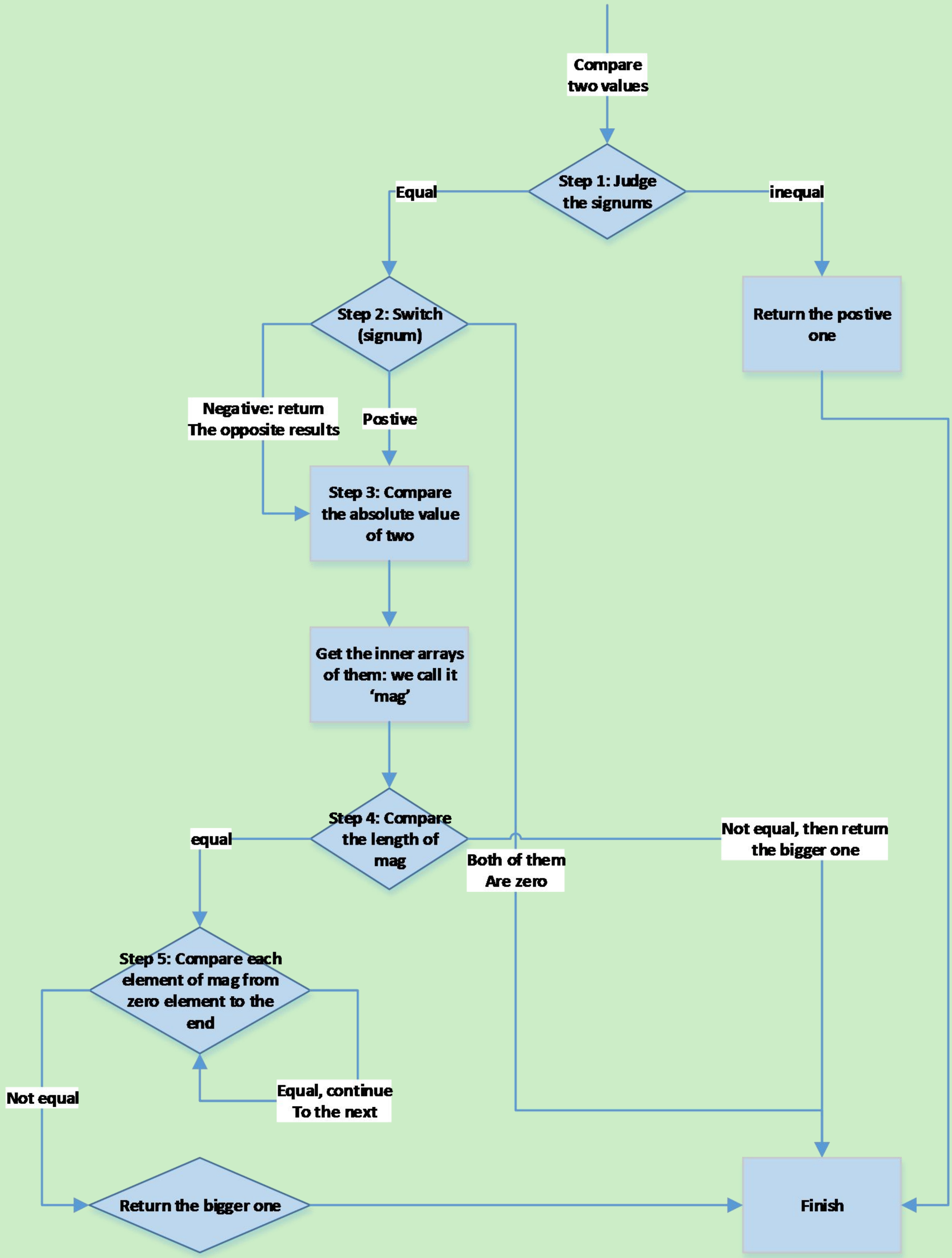
a. *Constructor:* the constructor of *BigInteger* fully supports construction from *String, char[], byte[]* etc.
b. *Range of Value:* by analyzing the source code, we could realize that the class would generate an *int* array to maintain the data and a *signum* flag to indicate its sign. The program will dynamically calculate the size of the array according to the scale of data. It turns out to be that the maximum bits of a data is expanded by this integer array. For example, one *Integer* could only represents a value from -2^{31}

to 2^{31} in Java, but if we have two integers and try to combine them together, then we could get a value from 2^{-63} to 2^{63}, which is a *Long* type value. By this analogy, we could add numerous *Integers* to accumulate a big enough data. Thus if we have an *int* array and its size is *L*, then theoretically we could store a value from -2^{32L-1} to 2^{32L-1}. However, the implementation of *BigInteger* restricts the maximum size of array, which could be found in the source code as the following:

```
/**
  * This constant limits {@code mag.length} of BigIntegers to the supported
  * range.
  */
 private static final int MAX_MAG_LENGTH = Integer.MAX_VALUE / Integer.SIZE + 1;
```

So the *BigInteger* could only support the value in the range ($-2^{Integer.MAX\_VALUE}$, $2^{Integer.MAX\_VALUE}$).

c. *CompareTo* method: by analyzing the source code, we could obtain the following procedure of comparison.



d. *Improvements:* actually we can see that this procedure is concise enough, but still we could find out someplace to optimize. Pay attention in *Step 5: Compare each element of mag from zero element to the end.* Considering the max size of the array referred before, we could omit the lower order of data, when the data is extremely vast so that the size of array is large as well. On the contrary, if the value is not that large, we could not omit the lower order because it would cause high possibility of error, which indicates that we should carefully choose the threshold of truncating the length of array. I prefer to use the following code to determine a new length we could count:

```
// threshold is a float number
int len_round = Math.round(len1 * threshold);
```

After choosing a suitable value of *threshold*, the difference between the *len1* and the *len_round* gradually increases as the length increasing. For example, if threshold equals 0.95. Then we could get the following exemplificative sequences:

Length	*Len1*	*Len_round*
1	1	1
20	19.0	19
85	80.75	81
101	95.95	96
1568	1489.6	1490
12425	11803.75	11804
…	…	…

As we can see, if the data is enough large, the lower order could be omitted by truncating the length.

e. *Probability:* so what degree would improve the comparison by this approximate calculation? Hard to tell, because in the real world, the sequences of input big integers are difficult to predict. Only after the following requirements are met that can trigger our optimization mechanism, otherwise immediately we will know which one is bigger or smaller.

a) We should be given two numbers which have the same signs, if completely random, the probability should be 50%;

b) They need to have the same number of digits, or their magnitude can be immediately determined. If the numbers are fully random, the probability that they have same numbers of digits should be: $\frac{1}{\mathrm{MAX\_SIZE}^2}$, the *MAX_SIZE* is the maximum number of digits. So it would approach zero when the *MAX_SIZE* is large;

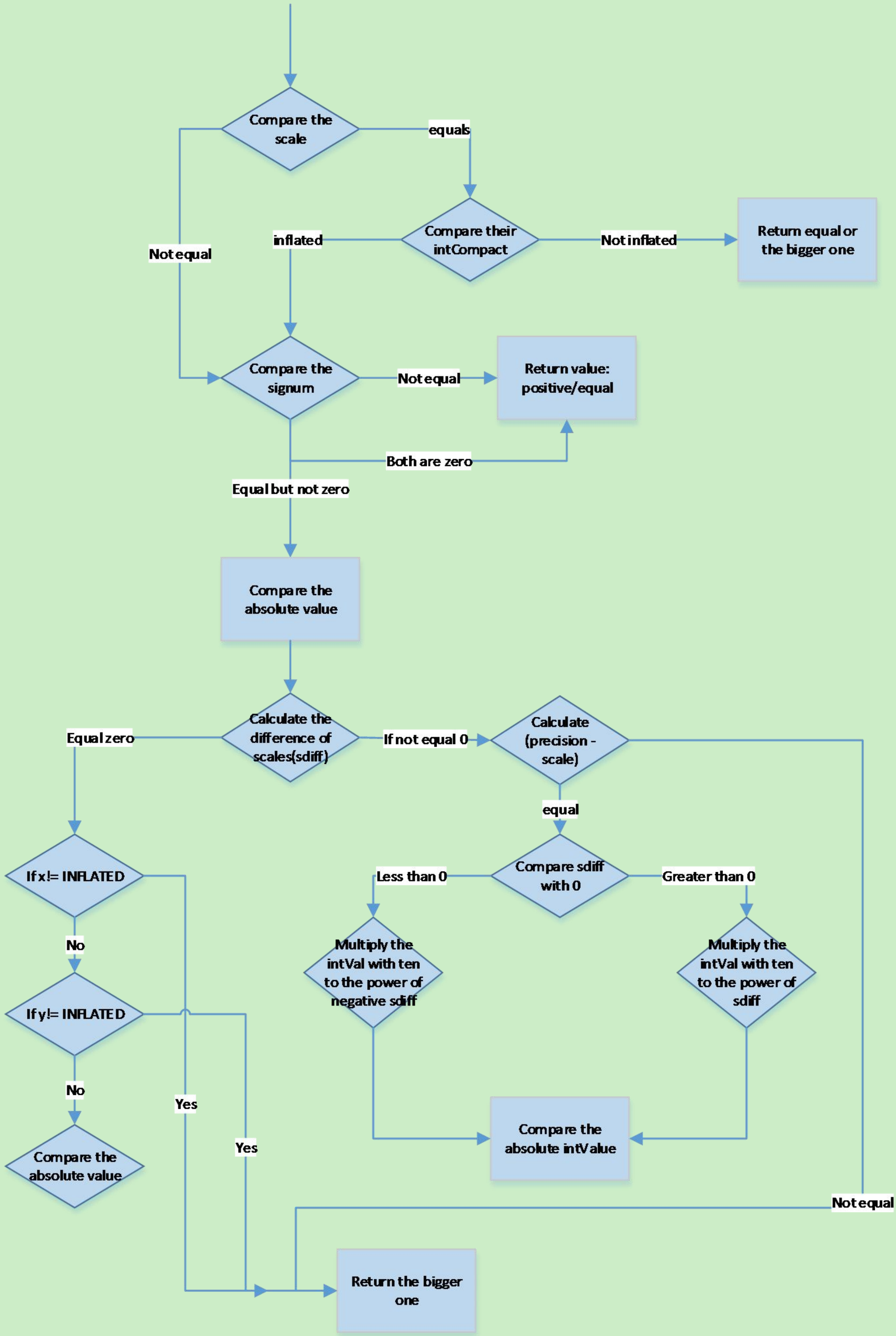
c) Then we could compare the element in the array one by one. Any difference in the high order would end the comparison, and each element has 2^{32} possibilities.

BigDecimal: this class provides us an immutable arbitrary-precision signed decimal numbers. A *BigDecimal* consists of a *BigInteger* (unscaled but signed value) and a 32-bit integer *scale*. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value of the number is multiplied by ten to the power of the negation of the scale. The value of the number represented by the *BigDecimal* is therefore $scaledValue \times 10^{-scale}$. The *BigDecimal* also follows the semantics of arithmetic operations as the same as *BigInteger*.

a. *Constructor:* the constructor of *BigDecimal* could be found in the API document.
b. *CompareTo method:* this method is far more complex than that in *BigInteger*. To start with, we should emphasize several variables defined in the class:

```
// The unscaled but signed value
private final BigInteger intVal;
// The scale, if the intVal is zero, it equals zero
private final int scale;
// The numbers of digit in BigDecimal, we usually
// obtain the number of integerpart by counting
// (precision - scale)
private transient int precision
/**
  * Sentinel value for {intCompact} indicating the significand
  * information is only available from {intVal}.
  */
static final long INFLATED = Long.MIN_VALUE;
/**
 * If the absolute value of the significand of this BigDecimal is less than or
 * equal to {Long.MAX_VALUE}, the value can be compactly stored in this
 * field and used in computations.
*/
private final transient long intCompact;
```

We could draw the following flow chart after analyzing its source code:


Compare the scale
equals
Not equal
Compare their intCompact
inflated
Not inflated
Return equal or the bigger one
Compare the signum
Not equal
Return value: positive/equal
Both are zero
Equal but not zero
Compare the absolute value
Calculate the difference of scales(sdiff)
Equal zero
If not equal 0
Calculate (precision - scale)
equal
Compare sdiff with 0
Less than 0
Greater than 0
If x!= INFLATED
No
If y!= INFLATED
No
Yes
Yes
Multiply the intVal with ten to the power of negative sdiff
Multiply the intVal with ten to the power of sdiff
Compare the absolute value
Compare the absolute intValue
Not equal
Return the bigger one

Through the flow chart, we could find that the calculation mainly happens after ensuring the two big decimals have same signs, same number of integer bits and each of them has exceeded the *LONG.MAX_VALUE* (which is represented by *INFLATED* flag). Then we transform them to *BigInteger* by multiply ten to *N(sdiff)* times, and compare the two *BigInteger* using the *compareMagnitude* method in the *BigInteger* class. So obviously, we have transformed the optimization of *BigDecimal* to the optimization of *BigInteger*, which is mentioned before.