1. **Overview**

Uncommons Maths is an open source software written in Java by Dan Dyer who is a software developer based in Kent in the UK [1]. This library focus on random number generators, probability distributions, combinatorics and statistics for Java [2]. According to the introduction on the official website and the source code downloaded from GitHUB [3], there are five components in this library: Random Numbers, Combinatorics, Statistics, Rational Arithmetic and Binary tool.

# Random Numbers

Random numbers is the key feature of this library. A Java programmer’s guide to random numbers are provided in the website. These three excellent introduction documents talk about a lot about the background of random numbers generation. [4]

The first step in order to know why the author develop this library is to understand the drawbacks of the implementation about random number with JDK. The JDK provides a standard API to get a random number which is java.util.Random, but there are some issues about this API when the programmer really want to generate a couple of random numbers [4]. Firstly, the result of this API is not random enough, it means that sometime you can predict the next result if you know the algorithms and seeds. Another issue is about performance. In the same time, JDK provides another API java.security.SecureRandom which is designed for cryptography, it can generate a random enough numbers but with poo performance.

So, this library focus on how to generate a random enough numbers but with high performance, it means this API can generate random numbers as fast as java.util.Random API with statistical properties as good as SecureRandom. In order to support this feature, the Uncommons Maths library provides different distribution algorithms to generate a random number. For example, Normal (Gaussian) distribution, Binomial distribution, Poisson distribution and Exponential distribution. This means that the programmer can select different distribution algorithms for different scenarios. [4] [5] [6]

# Combinatorics

This API provides some generics-enabled combination and permutation generators as well. There are two Java classes: CombinationGenerator and PermutationGenerator which are organized in package: org.uncommons.maths.combinatorics.

The combination generator is in charge of creating all combinations of a given size from the specified set of elements, according to the java doc of this API, this generator can produce no more than 2^63 different combinations. [7] The permutation generator can product all permutations for all sets up to 20 elements in size [8].

# Statistics

This API provides a statistical functions in order to calculate a variety of descriptive statistics, such as variance, median, standard deviation, arithmetic and geometric means and etc., according to the introduction in the website of this library. The core function is organized in one Java class: DataSet under package: org.uncommons.maths.statistics. [9]

# Rational Arithmetic

There is not a rational number type support in Java language, so this API provides a Rational number type which allows exact fractional arithmetic without loss of precision. This Rational Java class locate in package: org.uncommons.maths.number. This feature allows the denominator is positive but the numerator may be negative. [10]

# Binary

This is a toot includes two Java classes in order to manipulate binary values. BinaryUtils is for working with binary and hex data. The BitString Java class is an implementation of a fixed-length bit-string. [11]

1. **Data structures**

We downloaded this library, compile and run it using the default test cases. After an in-depth study and analysis of the source code, we found that there are several data structures being used. In this section, we will focus on the Java class: Maths which locate in package: org.uncommons.maths. In this class, there is one static method names bigFactorial which can result the factorial of n. We select this method because there is a data structure: ConcurrentHashMap which is used. We download and running the library with JDK 1.8, it works well. So the follows analysis base on JDK 1.8.

The first step for us is to know what this data structure is because we learned HashMap in class but do now have much concept why this new data structure has been introduced. Generally, the class ConcurrentHashMap was added into JDK from 1.5, it located in the package java.util.concurrent. The purpose of this class is to support a thread-safe tool during multi-threading environment. Before the ConcurrentHashMap, there are HashMap and HashTable, the HashMap do not support thread-safe features, and it means that during more than one thread to operate one HashMap at the same time, there will be some issues or a deadlock will occur. Another class HashTable can avoid this issue but with a very low performance because the get, put methods are synchronized, as a result only one thread can access a hashtable at any time. This means that one thread must wait if another thread I using this hashtable. [12]

In order to avoid lock the whole table, the ConcurrentHashMap uses a different design, there are some segments in one internal HashMap. If one thread access the first segment, only the first segment will be locked, at the same time, another thread can lock the second or other segments. With this implementation, the efficiency has been improved significantly.The difference between HashTable and ConcurrentHashMap is as follows:

ConcurrentHashMap lock one segment

Instead of lock the whole table

Entry

next

null

Entry

next

Entry

next

Entry

next

Entry

next

Entry

next

null

null

null

HashTable lock the whole table

Segment

Entry

null

Entry

Segment

Entry

Entry

Entry

next

Entry

next

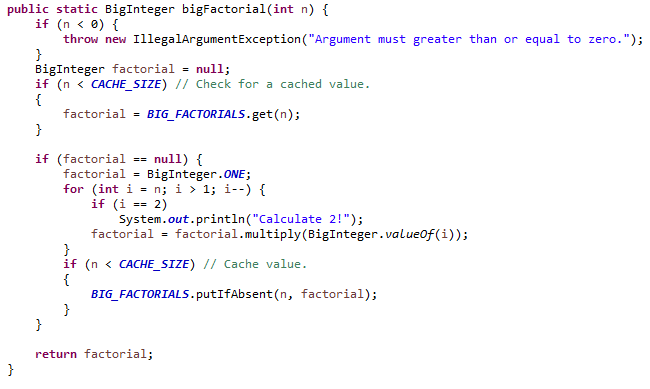
null

null

null

1. **Algorithms**

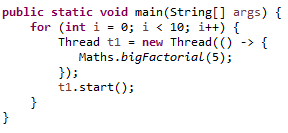
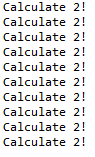
In order to calculate the factorial of n, the first 20 value of factorial will be stored in a ConcurrentHashMap data structure, with this algorithm, the efficiency has been improved. The details implementation is as follows:



Considering there are two threads, both of them needs to calculate factorial(10). After the first one thread finishes, the value of factorial(10) has been put into HashMap. So, when the second thread comes, it just get the value of factorial(10) from HashMap directly.For the first thread, the time complexity is O(n), but for the second thread, the time complexity will decline to O(1).

1. **Critique**

We notice that there is some points which can be improved in the function bigFactorial(n). Considering there are 10 thread to invoke this function in order to get factorial(5), since at the very beginning stage, the ConcurrentHashMap is empty, the 10 threads need to calculate the factorial(5) at the same time. We can verify this with the follows program:

When we notice the output of this test program the above, finding that each thread calculate the value rather getting from ConcurrentHashMap. The reason is that at the very beginning stage, the ConcurrentHashMap is empty.

Time

Thread 1

Thread 2

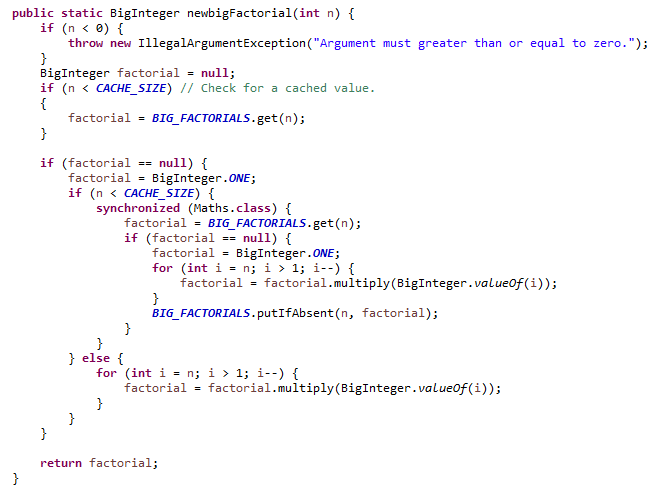
Check if the value in HashMap

Calculate factorial from 1 to n

Put value to HashMap

The 2nd check the value is not in HashMap because 1st thread not finish yet

This implementation need to be improved because if there are a couple of threads at the beginning stage, there will be a couple of duplicate calculations. An improved implementation about this function is as follows:



With our solution, only one thread can access the calculation section when n is less than 20.

1. **Conclusions**

In conclusion, we analyse the source code and make a depth-study about this Java library, the data structures are used appropriately and the algorithms are efficient. We give an example how to improve the time complexity.

In the future, some task need to be performed:

1. Learning deeply about the JDK ConcurrentHashMap class;
2. Learning deeply about the details of algorithms about how to generate distributions.
3. **Reference**

[1] <https://dandyer.co.uk/>

[2] <https://maths.uncommons.org/>

[3] <https://github.com/dwdyer/uncommons-maths>

[4] <https://blog.dandyer.co.uk/2008/04/03/a-java-programmers-guide-to-random-numbers-part-1-beyond-javautilrandom/>

[5] <https://blog.dandyer.co.uk/2008/04/06/a-java-programmers-guide-to-random-numbers-part-2-not-just-coins-and-dice/>

[6] <https://blog.dandyer.co.uk/2008/04/10/a-java-programmers-guide-to-random-numbers-part-3-seeding/>

[7] <https://maths.uncommons.org/api/org/uncommons/maths/combinatorics/CombinationGenerator.html>

[8] <https://maths.uncommons.org/api/org/uncommons/maths/combinatorics/PermutationGenerator.html>

[9] <https://maths.uncommons.org/api/org/uncommons/maths/statistics/DataSet.html>

[10] <https://maths.uncommons.org/api/org/uncommons/maths/number/Rational.html>

[11] <https://maths.uncommons.org/api/org/uncommons/maths/binary/BinaryUtils.html>

[12] <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html>