Exercise 2.1

1. The sequential version took 5.6 seconds.
2. The time taken by 10-thread version is 1.4 seconds which is really fast than the sequential one.
3. No, it produces different result each time I run the program.
4. No, the synchronized is not required here because the get() method is not accessed by any threads and the count variable state is not changed.

Exercise 2.2

1. The program takes 5.9 seconds to execute.
2. MyAtomicInteger is implemented in TestCountFactors.java.
3. Yes, it is displaying the same answer and the time to execute is 1.5 seconds.
4. No, we cannot implement MyAtomicInteger without synchronization because volatile field can only guarantee visibility but not atomicity. In absence of synchronization mutual exclusion cannot be obtained and thread will see stale data.
5. Yes, it is important to declare AtomicInteger as final so that its state is immutable.

Exercise 2.3

1. Cache field is declared volatile because when a thread sets the volatile cache field to reference a new OneValueCache, the new cached data becomes immediately visible to other threads. [answer from page 33 from the book]
2. Both need to be declared final because when a thread acquires a reference to immutable object it need never worry about another thread modifying its state.

Exercise 2.4

1. Implemented histogram2 in SimpleHistogram.java
    a. increament() and getCount() should be made synchronized in order to achieve atomicity.
    b. The getSpan() method does not need to be synchronized because it is not the part of atomic operation.
2. Implemented in SimpleHistogram.java
3. Yes, we can remove synchronized now because each integer is atomic now and no thread can interfere in between the operation making it atomic without synchronized keyword. The results are correctly printed.
4. The results are correctly printed.
5. Implemented in SimpleHistogram.java
6. Implemented in SimpleHistogram.java

Exercise 2.5

1. /

2.
Memoizer1 gives us no poor concurrency. No parallelism at all actually.
Output:
class Memoizer1
115000

real 0m26.428s
user 0m24.228s
sys 0m1.800s

3.
Memoizer2 gives us a decent degree of parallelism but there are many cache misses (because there is still a window of vulnerability). 132486 - 115000 = 17486
cache misses. The problem here is that two or more threads can start the computation at the same time for the same number.

class Memoizer2
132486

real 0m19.790s
user 0m37.720s
sys 0m0.088s

4.
Memoizer3 is faster than all the previous executions but there are still some cache misses. This happens because the if statement in the compute method is
non-atomic. The Figure 5.4 page68 (Goetz) depicts this problem very well.

class Memoizer3
115092

real 0m16.857s
user 0m25.260s
sys 0m0.744s

5.
As mentioned in the code, a hybrid of Memoizer3 and Memoizer. We can see that from the degree of parallelism and speed of execution, they give roughly the
same results. But, in this case we also get no cache misses which is because now we use an atomic putIfAbsent call.

class Memoizer4
115000

real 0m16.754s
user 0m25.508s
sys 0m0.744s

6.
No cache misses but slower than Memoizer4.

class Memoizer5
115000

real 0m23.842s
user 0m38.272s
sys 0m1.032s

7.
This simple implementation actually gives good results in the degree of parallelism, cache hits and speed of execution.

class Memoizer0
115000

real 0m18.424s
user 0m30.040s
sys 0m1.012s

Exercise 2.6

1. The thread that finishes last does not produce the same number as the main thread.
Example output:
main
main finished 40000000
fresh 1 stops: 23909808

fresh 0 stops: 24015599

2. The problem here is that we have synchronized on a Boxed primitive (a Long in this case). Boxed primitives are immutable which means that when we do
"count++" this actually ends up being executed as smth like this:
count = new Long(count + 1);

So basically, we end up synchronizing on a different reference (Object) each time we call count++ which is not how synchronization properly works.

3.
I solved this by creating an explicit lock:
final static Object lock = new Object();

main
main finished 40000000
fresh 1 stops: 39670696
fresh 0 stops: 40000000