Exercise 4.1

Question 1

The results of the measurements of the mark* methods are placed in the 'benchmarks' folder together with the system info.

Our benchmarks are almost identical to the ones made by Peter, so we conclude that they are very plausible.

In the Ex4_1_6 benchmarks we experienced a large standard deviation which might be because of concurrent programs running on the computer and maybe because of the garbage collector as it is similar to Peter's results where it does the same.

Question 2

Our results are stored in the benchmarks folder. Our benchmarks of mark7 shows that the stronger computer (Ex4_1_2_1) is faster than (Ex4_1_2_2) as expected. It is a little strange that the deviation of Ex4_1_2_2 is consistently larger than the other. This could be because the computer running Ex4_1_2_2 having more programs running and thus each of the benchmark runs doesn't have the same basis.

Exercise 4.2

Question 1

We think that there are quite a few outbursts in standard deviation during the different benchmarking runs. It is definitely not a steady move towards a robust result.

Question 2

The means of our benchmarking results are almost the same as in the lecture notes with just a very small constant factor larger (~10% slower). However, the standard deviations on the last results are very large compared to those of the lectures notes.

Exercise 4.3

Question 1

Question 2

The visual graphs of the result is saved as Ex4_3_2.png.

Question 3

Our benchmarking results shows that the 8-thread benchmark is the fastest. This makes sense as the computer used have 4 cores with hyperthreading, thus, 8 processors.

Question 4

There is no significant difference in the performance between AtomicLong and LongCounter - if anything, AtmoicLong is a little bit faster. When adequate built-in classes are available one should use them. They are most likely optimized and thoroughly tested.

The graph of the implementation using Atomic long is saved in Ex4_3_4.png

Question 5

We experienced that our modified version was slower than the original version.

We cannot argument for this strange behaviour.

Exercise 4.4

Question 1-6

```
Memoizer1                    1871977212,6 ns     29854323,13          2
Memoizer2                    1405574646,1 ns     52687573,52          2
Memoizer3                     999874912,2 ns     22564365,87          2
Memoizer4                     993271221,6 ns     26108494,08          2
Memoizer5                     977745474,8 ns     32062326,46          2
Memoizer                      986815814,1 ns     24626683,76          2
```

Question 7

We expected the first memoizer to be the slowest, as it blocks on the factorizer call. Though the second one doesn't block during the computation, it has a large risk of doing the same computation more than once, thereby spending time on unneccessary computations. The third version avoids the duplicate computations, but might create unnecessary `FutureTask` objects, which, as the second version, spends time on an unneccessary computation (the object construction), though the computation here is faster. The fourth and fifth version avoids these extra object constructions completely and blocks minimally.

As expected we see a larger improvement from the first cache version to the second, and again from the second to the third. The last three versions are almost identically, as the only thing removed, is the extra construction of small unneccessary objects.

The benchmarks correspond well with our expectations and with the literature.