1. First compile and run the thread timing code as is, using Mark6, to get a feeling for the variation and robustness of the results. Do not hand in the results but discuss any strangenesses, such as large variation in the time measurements for each case.

There are a couple of large variations of thread related operations (thread's work, thread create, thread create start)
For example in thread create start ,standard deviation suddenly increases almost 3 times, which might have something to do with a garbage collection or other program interfering with our execution.

```
Thread create start            128230.0 ns   66058.64          2
Thread create start            133442.5 ns   46735.46          4
Thread create start            108307.5 ns   21408.85          8
Thread create start             64485.0 ns    7505.79         16
Thread create start             65656.6 ns    9504.78         32
Thread create start             64755.6 ns    7237.45         64
Thread create start             64741.2 ns    3167.40        128
Thread create start             66564.8 ns    2591.43        256
Thread create start             66608.6 ns    2300.70        512
Thread create start             65264.3 ns    1035.75       1024
Thread create start             67951.8 ns    3413.52       2048
Thread create start             66590.2 ns    2219.10       4096
```

2. Now change all the measurements to use Mark7, which reports only the final result. Record the results in a text file along with appropriate system identification.

   Include the results in your hand-in, and reflect and comment on them: Are they plausible? Any surprises? Mention any cases where they deviate significantly from those shown in the lecture.

Comparing the results with ones in a lecture, they seem to be plausible as creation of threads takes approximately 66 000 ns ,whereas in a lecture it was 65 000 ns.

```
# OS:   Windows 10; 10.0; amd64
# JVM:  Oracle Corporation; 22.0.2
# CPU:  Intel64 Family 6 Model 158 Stepping 13, GenuineIntel; 12 "cores"
# Date: 2024-10-23T16:07:00+0200
Mark 7 measurements
Point creation                      31.5 ns       0.16    8388608
Thread's work                     4266.6 ns     241.19      65536
Thread create                     1166.9 ns      11.95     262144
Thread create start              66415.7 ns    2179.12       4096
Thread create start join        135453.0 ns    1621.58       2048
ai value = 1433540000
Uncontended lock                    17.5 ns       0.45   16777216
```

3. Use the same reasoning as in the lecture (slide "Thread create + start") to estimate the cost of creating a thread on your own computer. Include your result in the text files with answers for this exercise.

A lot of time is going to thread creation and start ,especially to thread start where it takes almost half of the "thread create start join " time. Looking at actual work being done by a thread ,it is more expensive to start a thread than to do the work. As it was suggested in the lecture slides, we shouldn't use threads for smaller operations as it is so computational demanding to just start them.

countParallelN	6	4798594.5	ns	79242.31	64
countParallelN	7	4385739.7	ns	132049.48	64
countParallelN	8	4357605.8	ns	133444.72	64
countParallelN	9	4399752.3	ns	96047.11	64
countParallelN	10	4451871.7	ns	80422.61	64
countParallelN	11	4538920.8	ns	95447.59	64
countParallelN	12	4618135.9	ns	115077.33	64
countParallelN	13	4633585.2	ns	127921.43	64
countParallelN	14	4654939.5	ns	185396.53	64
countParallelN	15	4686390.0	ns	142714.28	64
countParallelN	16	4654504.8	ns	80089.83	64
countParallelN	17	4681531.6	ns	77029.48	64
countParallelN	18	4753148.6	ns	181426.13	64
countParallelN	19	4846577.5	ns	149573.76	64
countParallelN	20	4880834.1	ns	130076.22	64
countParallelN	21	4910896.1	ns	111274.13	64
countParallelN	22	5077777.3	ns	168954.72	64
countParallelN	23	5095058.0	ns	114310.63	64
countParallelN	24	4663744.5	ns	677301.48	64
countParallelN	25	4604303.0	ns	539982.97	64
countParallelN	26	4870684.5	ns	546563.99	64
countParallelN	27	4568448.4	ns	665766.69	128
countParallelN	28	4086853.9	ns	624974.64	64
countParallelN	29	4007812.8	ns	605105.73	64
countParallelN	30	5029266.3	ns	1775477.53	64
countParallelN	31	16358437.8	ns	10438009.92	32
countParallelN	32	19114840.0	ns	10982412.06	32

Exercise 8.2 In this exercise you should estimate whether there is a performance gain by declaring a shared variable as volatile. Consider this simple class that has both a volatile `int` and another `int` that is not declared volatile:

```
public class TestVolatile {
  private volatile int vCtr;
  private int ctr;
  public void vInc () {  vCtr++;   }
  public void inc () {   ctr++;   }
}
```

Mandatory

Use Mark7 (from `Bendchmark.java`) to compare the performance of incrementing a `volatile int` and a normal `int`. Include the results in your hand-in and comment on them: Are they plausible? Any surprises?

```
> Task :app:CompareTestVolatile.main()
# OS:   Windows 10; 10.0; amd64
# JVM:  Eclipse Adoptium; 21.0.4
# CPU:  Intel64 Family 6 Model 158 Stepping 13, GenuineIntel; 12 "cores"
# Date: 2024-10-23T19:39:31+0200
Mark 7 measurements
Testing volatile int increment             4.7 ns       1.09   134217728
Testing regular int increment              2.3 ns       0.06   134217728
```

Volatile `int` increment:

- **4.7 ns** per increment.
- The slightly higher time reflects the **overhead** of ensuring **memory visibility** and consistency across threads. The JVM must place memory barriers to ensure that the value is correctly visible to other threads, preventing caching optimizations.
- **Regular `int` increment**:
- **2.3 ns** per increment.
- This is faster because no synchronization or memory barrier is required. The JVM can optimize the increment operation freely (e.g., through caching or reordering instructions), as there are no concerns about thread visibility.

Exercise 8.3 In this exercise you must use the benchmarking infrastructure to measure the performance of the prime counting example given in file `TestCountPrimesThreads.java`.

Mandatory

1. Measure the performance of the prime counting example on your own hardware, as a function of the number of threads used to determine whether a given number is a prime. Record system information as well as the measurement results for 1...32 threads in a text file. If the measurements take excessively long time on your computer, you may measure just for 1...16 threads instead.
2. Reflect and comment on the results; are they plausible? Is there any reasonable relation between the number of threads that gave best performance, and the number of cores in the computer you ran the benchmarks on? Any surprises?

The time for the **sequential execution** (using 1 thread) is **14,919,292.2 ns** (~14.9 ms), which serves as a baseline.The multiple threads ensure parallel execution that can be seen in case of having 2 threads. With 2 threads, the time is **8,294,694.7 ns** (~8.3 ms), showing a significant improvement (almost halved from the sequential execution).

The **best performance** appears to be around 28 threads, with **4,086,853.9 ns** (~4.1 ms), after which the execution time begins to fluctuate or degrade.
After hitting the 28 number of threads, the performance begins to **degrade**, with 31 and 32 threads showing execution times as high as **16,358,437.8 ns** (~16.3 ms) and **19,114,840.0 ns** (~19.1 ms), respectively, which are even **worse than the sequential time**.

Mandatory

1. `TestTimeSearch` uses a slightly extended version of the `LongCounter` where two methods have been added `void add(long c)` that increments the counter by `c` and `void reset()` that sets the counter to 0.

   Extend `LongCounter` with these two methods in such a way that the counter can still be shared safely by several threads.

2. How many occurencies of "ipsum" is there in `long-text-file.txt`. Record the number in your solu tion.

There is # Occurences of ipsum :1430

3. Use Mark7 to benchmark the search function. Record the result in your solution.

```
Array Size: 5697
search                          8962528.8 ns   62766.28        32
# Occurences of ipsum :888030
```

4. Extend the code in `TestTimeSearch` with a new method

```
private static long countParallelN(String target,
                 String[] lineArray, int N, LongCounter lc) {
// uses N threads to search lineArray
  ...
}
```

Fill in the body of `countParallelN` in such a way that the method uses N threads to search the lineArray. Provide a few test results that make i plausible that your code works correctly.

```
private static long countParallelN(String target,String[] lineArray, int
N,LongCounter lc) {
  int range = lineArray.length;
  final int perThread= range / N;
  Thread[] threads= new Thread[N];
```

```
  for (int t=0; t<N; t++) {
    final int from= perThread * t,
            to= (t+1==N) ? range : perThread * (t+1);
    threads[t]= new Thread(()->
    {for (int i=from; i<to; i++)
      if (lineArray[i].equals(target)) lc.increment();
    });
  }
  for (int t=0; t<N; t++) threads[t].start();
  try { for (int t=0; t<N; t++) threads[t].join();
  } catch (InterruptedException exn) { }
  return lc.get();
}
```

5. Use Mark7 to benchmark `countParallelN`. Record the result in your solution and provide a small discussion of the timing results.

The parallel search (`countParallelN`) is significantly faster than the sequential search, as expected. With the array size of 5697 elements, the parallel version took around ~3.2 ms, while the sequential version took around ~10.7 ms.
The performance boost from using multiple threads is quite clear. By dividing the workload across `N` threads, each thread handles a smaller portion of the array concurrently, allowing for faster completion compared to the sequential approach where one thread processes the entire array.

```
Array Size: 5697
Sequential Search              10677269.4 ns  248249.30         32
Occurrences of ipsum (Sequential): 1430
```

```
> Task :app:TestCountParallelN.main()
Array Size: 5697
Parallel Search                 3238491.3 ns   62304.01        128
Occurrences of ipsum (Parallel): 1430
```