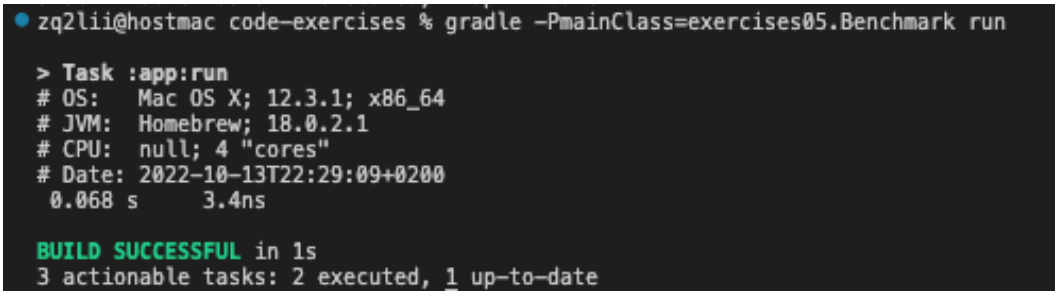
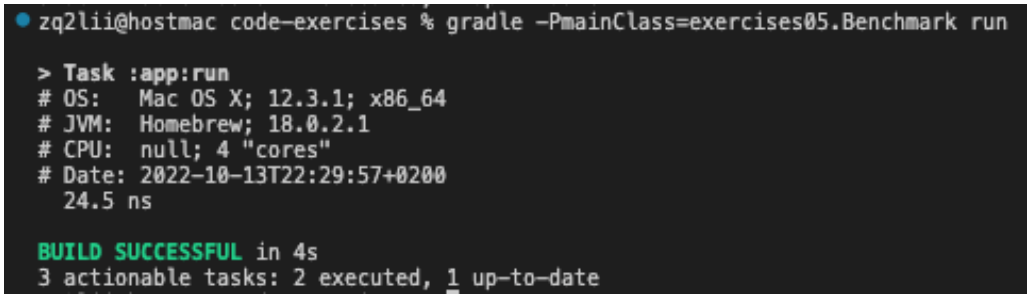
# Exercises week 5 report

## Exercise 5.1

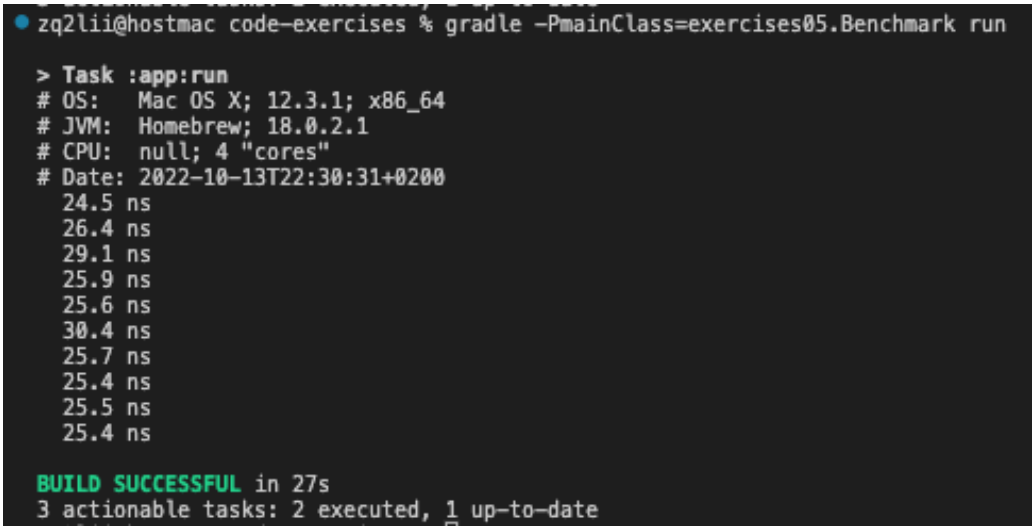
1. Use Benchmark.java to run the Mark1 through Mark6 measurements.
    - Mark1




    - Mark2




    - Mark3




    - Mark4

```
zq2lii@hostmac code-exercises % gradle -PmainClass=exercises05.Benchmark run

> Task :app:run
# OS:   Mac OS X; 12.3.1; x86_64
# JVM:  Homebrew; 18.0.2.1
# CPU:  null; 4 "cores"
# Date: 2022-10-13T22:31:30+0200
  25.6 ns +/-  0.260

BUILD SUCCESSFUL in 27s
3 actionable tasks: 2 executed, 1 up-to-date
```

- Mark5

```
zq2lii@hostmac code-exercises % gradle -PmainClass=exercises05.Benchmark run

> Task :app:run
# OS:   Mac OS X; 12.3.1; x86_64
# JVM:  Homebrew; 18.0.2.1
# CPU:  null; 4 "cores"
# Date: 2022-10-13T22:32:35+0200
 488.5 ns +/-   722.55          2
 212.9 ns +/-    75.21          4
 179.3 ns +/-    49.16          8
 167.2 ns +/-    84.11         16
  48.7 ns +/-     5.83         32
  46.7 ns +/-     8.12         64
  58.8 ns +/-    90.06        128
  32.7 ns +/-     7.86        256
  29.9 ns +/-     5.31        512
 114.8 ns +/-   200.07       1024
  27.3 ns +/-     0.37       2048
  30.7 ns +/-     8.12       4096
  25.8 ns +/-     1.01       8192
  61.6 ns +/-   104.96      16384
  30.6 ns +/-     7.99      32768
  31.2 ns +/-    10.26      65536
  28.3 ns +/-     3.09     131072
  26.1 ns +/-     0.99     262144
  25.5 ns +/-     0.18     524288
  25.6 ns +/-     0.24    1048576
  25.5 ns +/-     0.21    2097152
  25.7 ns +/-     0.35    4194304
  25.7 ns +/-     0.19    8388608
  26.0 ns +/-     0.75   16777216

BUILD SUCCESSFUL in 10s
3 actionable tasks: 2 executed, 1 up-to-date
```

- Mark6

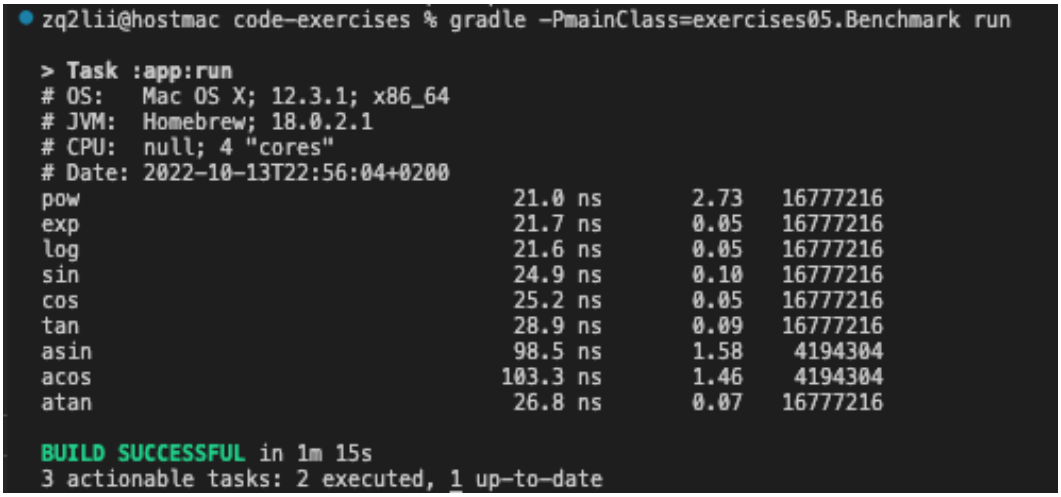
```
zq2lii@hostmac code-exercises % gradle -PmainClass=exercises05.Benchmark run

> Task :app:run
# OS:   Mac OS X; 12.3.1; x86_64
# JVM:  Homebrew; 18.0.2.1
# CPU:  null; 4 "cores"
# Date: 2022-10-13T22:33:23+0200
multiply                          579.2 ns     941.92          2
multiply                          260.7 ns      78.61          4
multiply                          243.7 ns      60.83          8
multiply                          329.2 ns      47.12         16
multiply                          346.1 ns      96.26         32
multiply                          314.7 ns      20.90         64
multiply                          130.7 ns     243.94        128
multiply                           59.4 ns       3.77        256
multiply                           49.1 ns      18.60        512
multiply                           60.5 ns      11.48       1024
multiply                           73.8 ns      35.12       2048
multiply                           50.2 ns      21.21       4096
multiply                           32.5 ns       2.89       8192
multiply                           40.3 ns      21.26      16384
multiply                           36.1 ns      21.45      32768
multiply                           29.4 ns       4.14      65536
multiply                           39.0 ns      13.22     131072
multiply                           35.3 ns      11.68     262144
multiply                           35.1 ns       7.08     524288
multiply                           30.9 ns       9.02    1048576
multiply                           26.6 ns       1.21    2097152
multiply                           26.7 ns       1.79    4194304
multiply                           27.7 ns       2.39    8388608
multiply                           29.9 ns       7.07   16777216

BUILD SUCCESSFUL in 10s
3 actionable tasks: 2 executed, 1 up-to-date
```

All the results above are plausible. The actual running time may vary from the *Microbenchmarks note*, however they're reasonable.

The running time in Mark1 is significantly small, this may because in JIT, the loop isn't processed actually. Once we return *dummy*, the result in Mark2 is more reasonale. And in Mark3 we can see all the iterations take nearly the same amount of time.

There're some exception in Mark5 when there are sudden increase for iteration count 128/1024/16384, but we also see that the standard deviations are very large in those cases, which tells us we can have no confidence in those results.

2. Use Mark7 to measure the execution time for the mathematical functions pow, exp, and so on, as in *Microbenchmarks note* Section 4.2. Record the results in a text file along with appropriate system identification. Preferably do this on at least two different platforms, eg. your own computer and a fellow student/friends computer.
    - MacOS




- Windows

some analysis here...

## Exercise 5.2

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 cases.

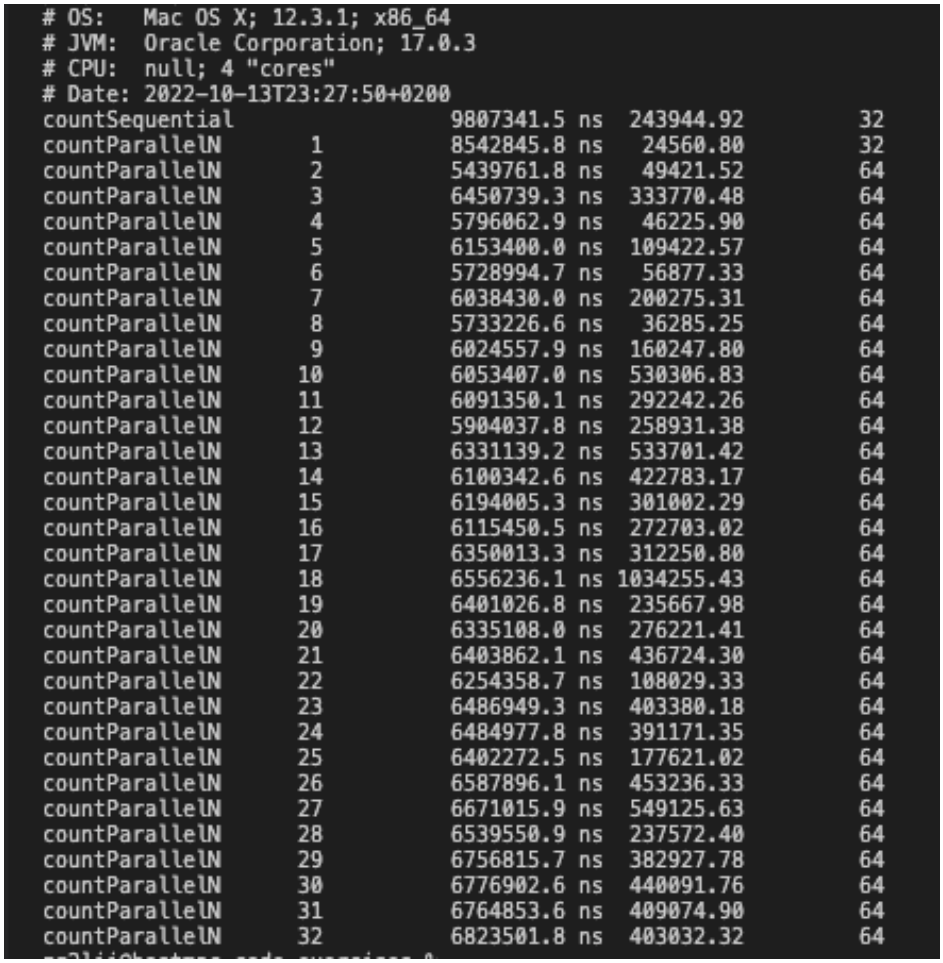
   There're always some exceptions in different cases. Such as for *Thread create start join*, when iteration count is  512, there's significant increase of the mean time. This outlier measurement may be caused by the garbage collector accidentally performing some work at that time, or the just-in-time compiler, or some other external disturbance.

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.


Result is plausible, we can see that the creation of simple object cost just 2.7ns, but the creation of thread takes more than 800ns. And the start of a thread is even more, it takes almost 70000ns, even after creating those threads.

## Exercise 5.3

1. Measure the performance of the primecounting 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



```
# OS:   Mac OS X; 12.3.1; x86_64
# JVM:  Oracle Corporation; 17.0.3
# CPU:  null; 4 "cores"
# Date: 2022-10-13T23:27:50+0200
countSequential                 9807341.5 ns  243944.92         32
countParallelN        1         8542845.8 ns   24560.80         32
countParallelN        2         5439761.8 ns   49421.52         64
countParallelN        3         6450739.3 ns  333770.48         64
countParallelN        4         5796062.9 ns   46225.90         64
countParallelN        5         6153400.0 ns  109422.57         64
countParallelN        6         5728994.7 ns   56877.33         64
countParallelN        7         6038430.0 ns  200275.31         64
countParallelN        8         5733226.6 ns   36285.25         64
countParallelN        9         6024557.9 ns  160247.80         64
countParallelN       10         6053407.0 ns  530306.83         64
countParallelN       11         6091350.1 ns  292242.26         64
countParallelN       12         5904037.8 ns  258931.38         64
countParallelN       13         6331139.2 ns  533701.42         64
countParallelN       14         6100342.6 ns  422783.17         64
countParallelN       15         6194005.3 ns  301002.29         64
countParallelN       16         6115450.5 ns  272703.02         64
countParallelN       17         6350013.3 ns  312250.80         64
countParallelN       18         6556236.1 ns 1034255.43         64
countParallelN       19         6401026.8 ns  235667.98         64
countParallelN       20         6335108.0 ns  276221.41         64
countParallelN       21         6403862.1 ns  436724.30         64
countParallelN       22         6254358.7 ns  108029.33         64
countParallelN       23         6486949.3 ns  403380.18         64
countParallelN       24         6484977.8 ns  391171.35         64
countParallelN       25         6402272.5 ns  177621.02         64
countParallelN       26         6587896.1 ns  453236.33         64
countParallelN       27         6671015.9 ns  549125.63         64
countParallelN       28         6539550.9 ns  237572.40         64
countParallelN       29         6756815.7 ns  382927.78         64
countParallelN       30         6776902.6 ns  440091.76         64
countParallelN       31         6764853.6 ns  409074.90         64
countParallelN       32         6823501.8 ns  403032.32         64
```

   Based on the results, when the number of threads is 2, it gives the best performance, it has the lowest mean running time. This may because that my computer has 2 cores(not as the system info, but the infomation on my computer). And we can also see that when the number of threads is 4, 6 and 8, they have a better performance. Maybe because they can be devided by 2- the number of cores. But when the number of threads are bigger than 8, we can't see a significant difference.
3. Now instead of the LongCounterclass, use the java.util.concurrent.atomic. AtomicLong class for the counts. Perform the measurements again as indicated above. Discuss the results: is the performance of AtomicLong better or worse than that of LongCounter? Should one in general use adequate built-in classes and methods when they exist?

```
# OS:   Mac OS X; 12.3.1; x86_64
# JVM:  Oracle Corporation; 17.0.3
# CPU:  null; 4 "cores"
# Date: 2022-10-14T00:02:51+0200
countSequential                 9808714.3 ns  205362.00         32
countParallelN         1        9731490.7 ns   42877.05         32
countParallelN         2        6153951.4 ns  432055.12         64
countParallelN         3        6509242.1 ns   46128.94         64
countParallelN         4        5740885.5 ns   74381.42         64
countParallelN         5        6000743.4 ns   42408.12         64
countParallelN         6        5745877.0 ns  168440.64         64
countParallelN         7        5881720.6 ns   77541.73         64
countParallelN         8        5928870.8 ns  481795.17         64
countParallelN         9        5868264.4 ns  126325.55         64
countParallelN        10        5900890.9 ns  257207.72         64
countParallelN        11        5915625.4 ns  110263.80         64
countParallelN        12        5903269.2 ns  271632.53         64
countParallelN        13        6009187.0 ns  173450.31         64
countParallelN        14        6011901.6 ns  255129.99         64
countParallelN        15        6016384.9 ns  224439.55         64
countParallelN        16        6063277.4 ns  294851.70         64
countParallelN        17        6073173.1 ns  170994.97         64
countParallelN        18        6503371.9 ns  686517.34         32
countParallelN        19        6551316.0 ns  482633.60         64
countParallelN        20        6276914.9 ns  419257.23         64
countParallelN        21        6520606.5 ns  455838.68         64
countParallelN        22        6341613.7 ns  382063.24         64
countParallelN        23        6442255.3 ns  469971.44         64
countParallelN        24        6464152.7 ns  569967.67         64
countParallelN        25        6470200.0 ns  394618.30         64
countParallelN        26        6866614.5 ns  734571.52         64
countParallelN        27        6542770.6 ns  437615.06         64
countParallelN        28        6643148.4 ns  427067.34         64
countParallelN        29        6457241.5 ns   55336.02         64
countParallelN        30        6732200.1 ns  521407.05         64
countParallelN        31        6710040.1 ns  427852.87         64
countParallelN        32        7106116.3 ns  546215.39         64
```

When use AtomicLong class, the mean run time is longer.

## Exercise 5.4

1. 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?

```
# OS:   Mac OS X; 12.3.1; x86_64
# JVM:  Oracle Corporation; 17.0.3
# CPU:  null; 4 "cores"
# Date: 2022-10-14T00:48:06+0200
test normal int                    1.2 ns       0.01  268435456
test volatile int                  7.2 ns       0.09   67108864
```

The code is within *TestVolatile.java*, the result is shown as above. From the result we can see that the performance of incrementing a volatile int is worse than a normal int.

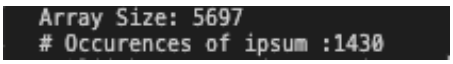
Volatile int always read the value from the main memory, this may result in some error, when some threads changed the value in the register but hasn't write the value into the main memory. The performance of the normal int is better may because that normal int read the value from the local register for each thread, which will cost less time than reading from the main memory.

# Exercise 5.5

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

   The implementation of this part can be seen in *LongCounter.java*, use *ReentrantLock*.

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

   The size of the array is 5697, and occurences of ipsum is 1430.

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

   Test search function 12340448.0 ns 585142.47 32

4. Extend the code in TestTimeSearch with a new method