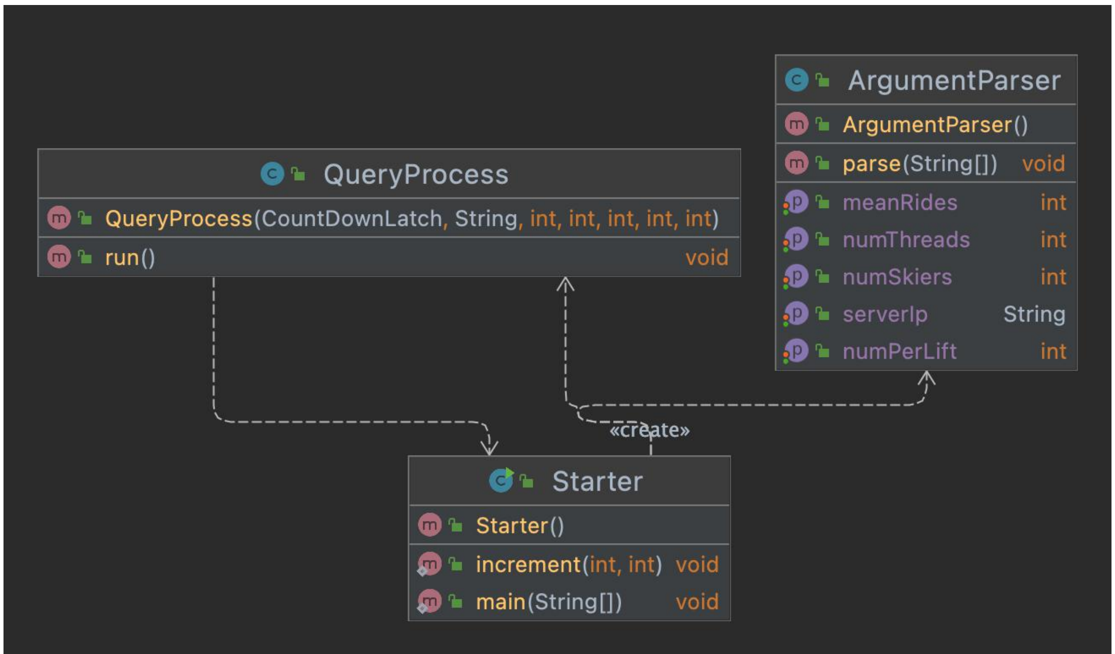
Part 1:

Saiqi He 001096811
URL to github: https://github.com/saiqi1999/BuildingScalableDistributedSystem

Project structure:



COMMAND LINE ARGUMENTS FOR MAIN():
-i IP address (and port), a string like 127.0.0.1:8080, required
-t number of threads, default 200
-n number of skiers, default 10000
-p number per lift, default 40
-m mean ride number of the skiers, default 10

The project have only 3 phase so I have them coded in Starter() - main() method

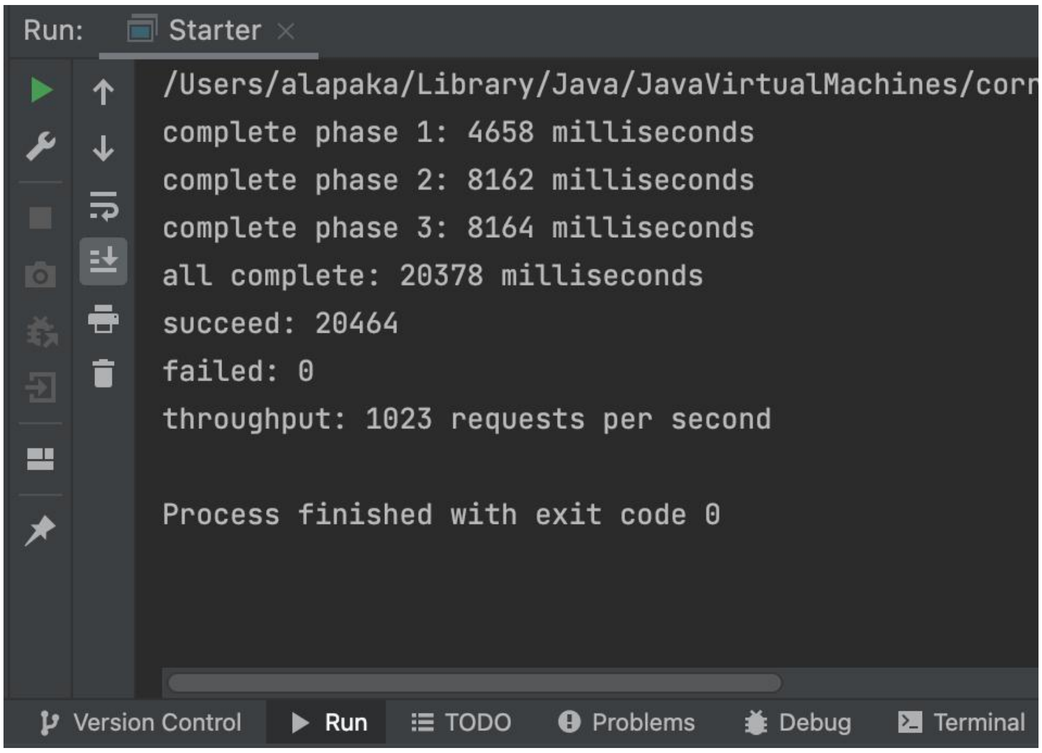
First you'll call main() method and the arguments are send to a customized parser which will extract useful information in the arguments

CountDownLatch is used for change between phases, and Thread.join() is used to determine the end.

I have a synchronized method in Starter class to calculate total success/failure, it's called by processing thread, which will send request to server and process.

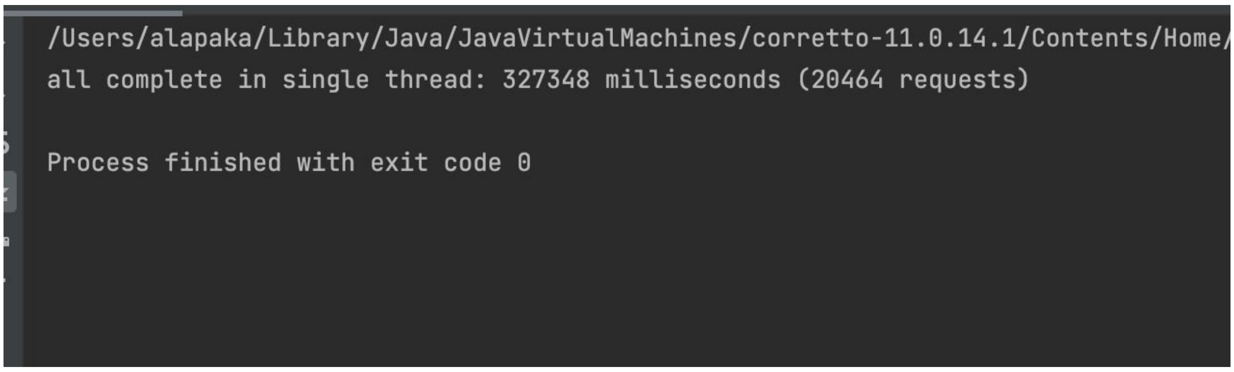
Below is the prediction of Little's Law:

First I tested with 64 threads & 1024 skiers. Result:



```
/Users/alapaka/Library/Java/JavaVirtualMachines/corr
complete phase 1: 4658 milliseconds
complete phase 2: 8162 milliseconds
complete phase 3: 8164 milliseconds
all complete: 20378 milliseconds
succeed: 20464
failed: 0
throughput: 1023 requests per second

Process finished with exit code 0
```

The number of succeed request never changes, so the statistic is correct. Then I tested with a single thread and same number of request:

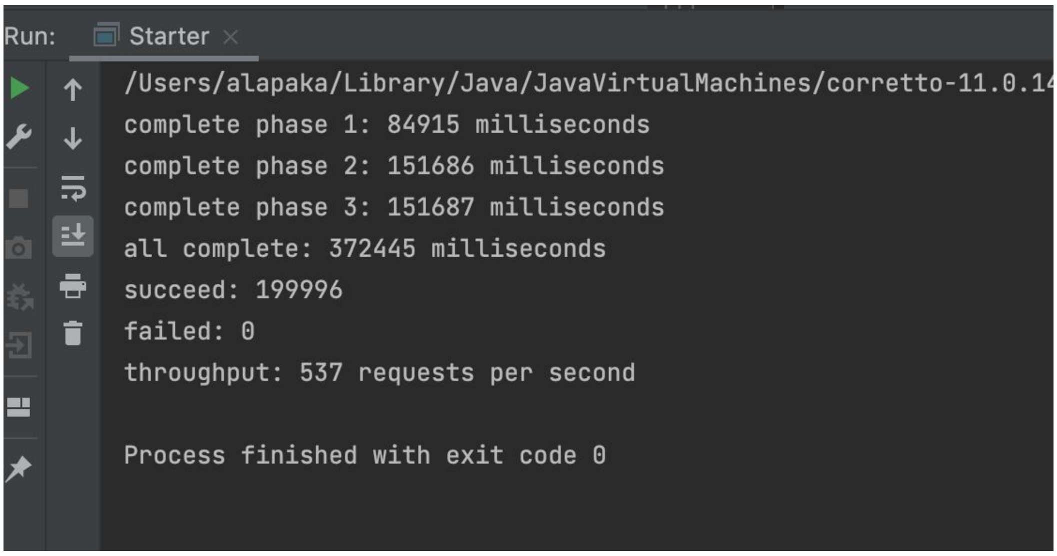


```
/Users/alapaka/Library/Java/JavaVirtualMachines/corretto-11.0.14.1/Contents/Home/
all complete in single thread: 327348 milliseconds (20464 requests)

Process finished with exit code 0
```

Which follows Little’s law. The multithread method has roughly 16 times of throughput, and we assume they have same processing time in server for each thread. Which means the average number of thread running in tomcat is 16. That’s a reasonable outcome because in both phase 1 and phase 3 there should be less than 16 threads working in server (we start 1/4 & 1/10 threads in these phases) but phase 2 will compensate for that.

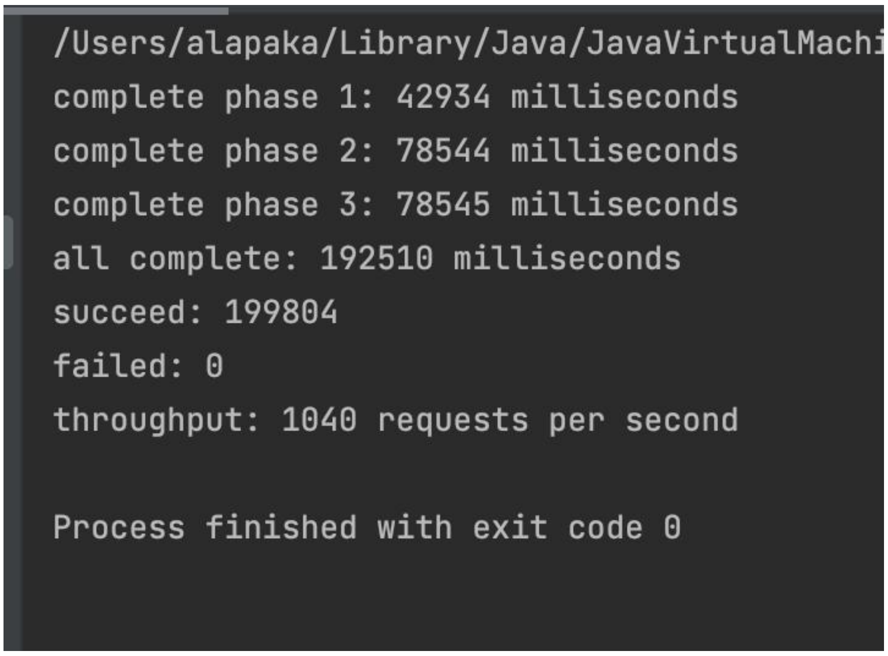
Now run the client with 32, 64, 128 and 256 threads, with numSkiers=20000, and numLifts=40

32 threads:



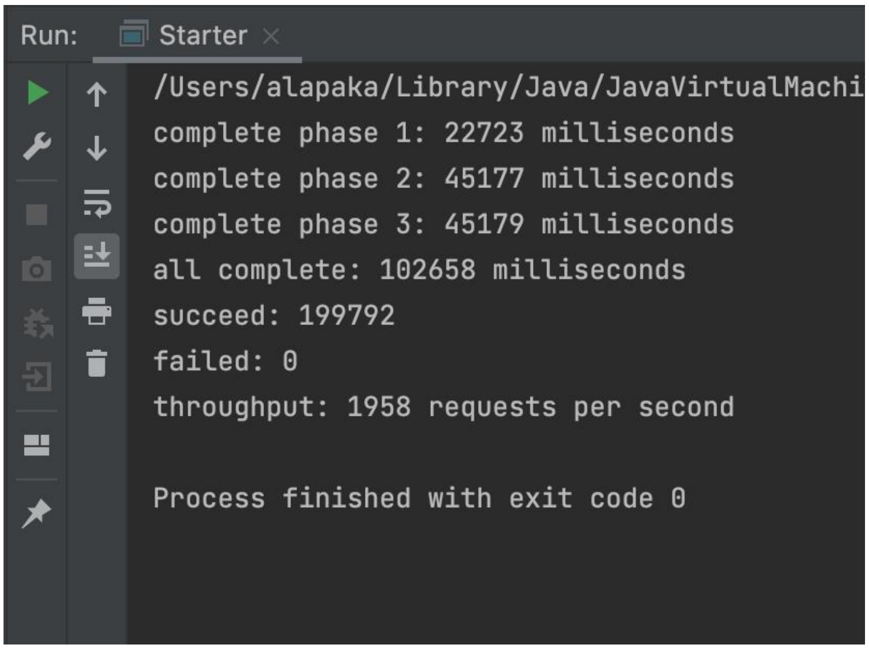
```
/Users/alapaka/Library/Java/JavaVirtualMachines/corretto-11.0.14
complete phase 1: 84915 milliseconds
complete phase 2: 151686 milliseconds
complete phase 3: 151687 milliseconds
all complete: 372445 milliseconds
succeed: 199996
failed: 0
throughput: 537 requests per second

Process finished with exit code 0
```

64 threads:



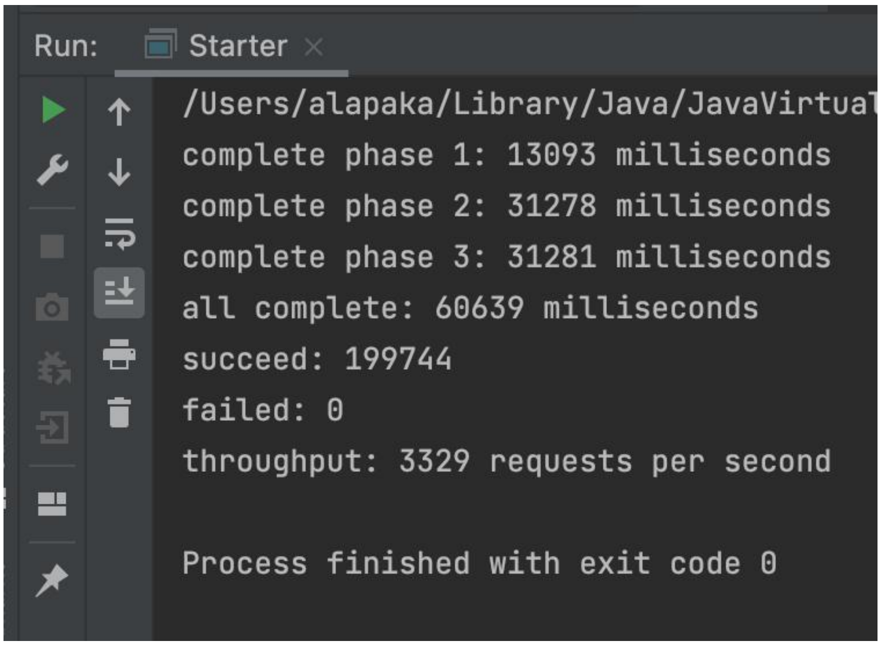
```
/Users/alapaka/Library/Java/JavaVirtualMachi
complete phase 1: 42934 milliseconds
complete phase 2: 78544 milliseconds
complete phase 3: 78545 milliseconds
all complete: 192510 milliseconds
succeed: 199804
failed: 0
throughput: 1040 requests per second

Process finished with exit code 0
```

128 threads:



```
/Users/alapaka/Library/Java/JavaVirtualMachi
complete phase 1: 22723 milliseconds
complete phase 2: 45177 milliseconds
complete phase 3: 45179 milliseconds
all complete: 102658 milliseconds
succeed: 199792
failed: 0
throughput: 1958 requests per second

Process finished with exit code 0
```

256 threads:



```
/Users/alapaka/Library/Java/JavaVirtual
complete phase 1: 13093 milliseconds
complete phase 2: 31278 milliseconds
complete phase 3: 31281 milliseconds
all complete: 60639 milliseconds
succeed: 199744
failed: 0
throughput: 3329 requests per second

Process finished with exit code 0
```

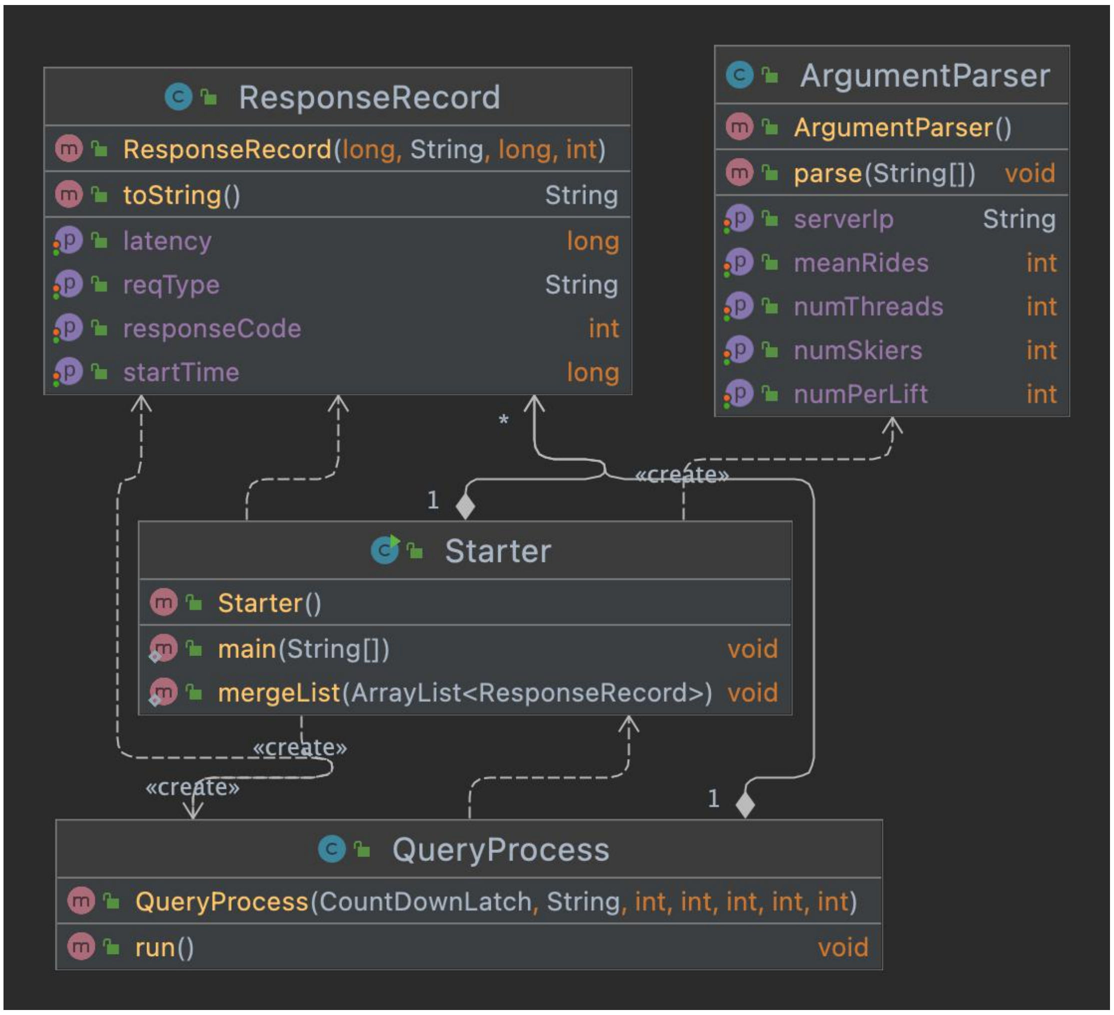
I made a chart of #Thread <--> Throughput:

#THREAD	32	64	128	256
Throughput(request/s)	537	1040	1958	3329

The default settings of tomcat is allowing 200 threads working inside it. So when #threads < 200 they will expand proportional but not for 256 threads, there are threads waiting starting from peak phase. I also noticed the total number of request is varying, which should due to rounding the number of scheduled request between threads.

Part 2:

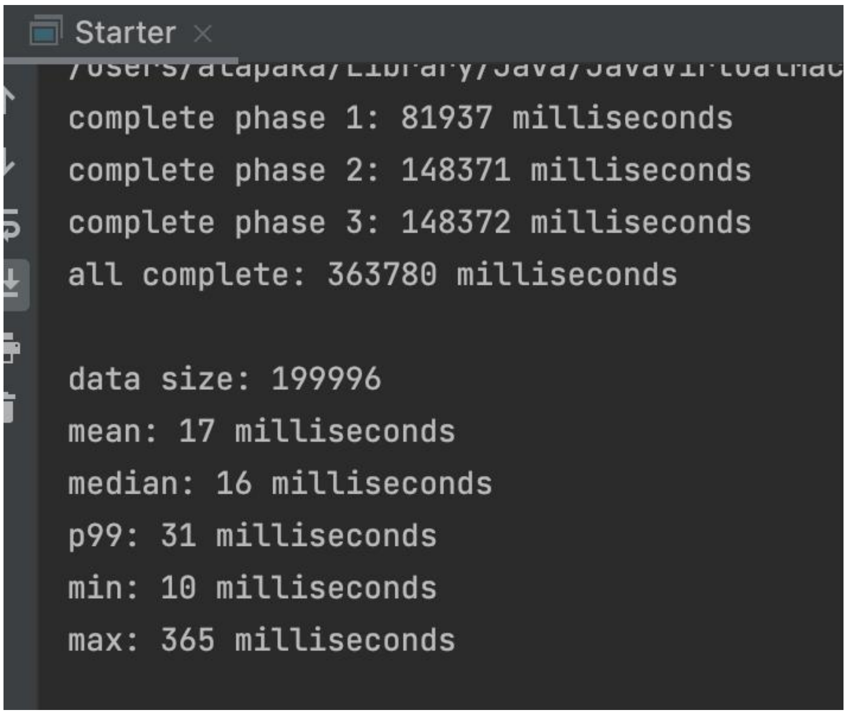
Project structure:



I created another ResponseRecord class to hold the result of one POST

In this part we run for threads number = 32, 64, 128, 256 with skier number = 20000 and mean ride = 10, lift number = 40.

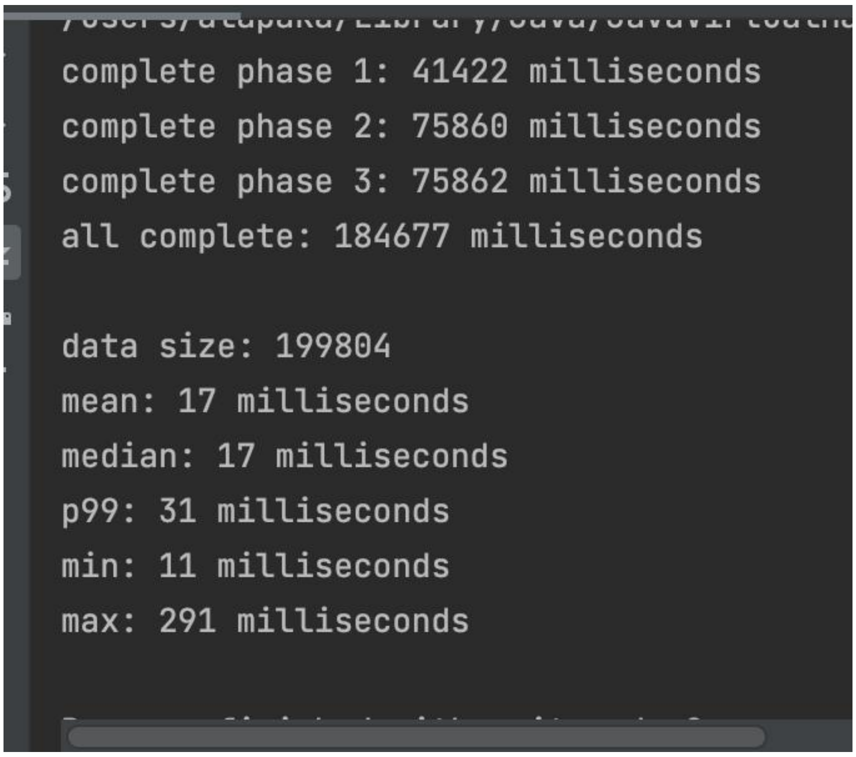
32 Threads:



Starter

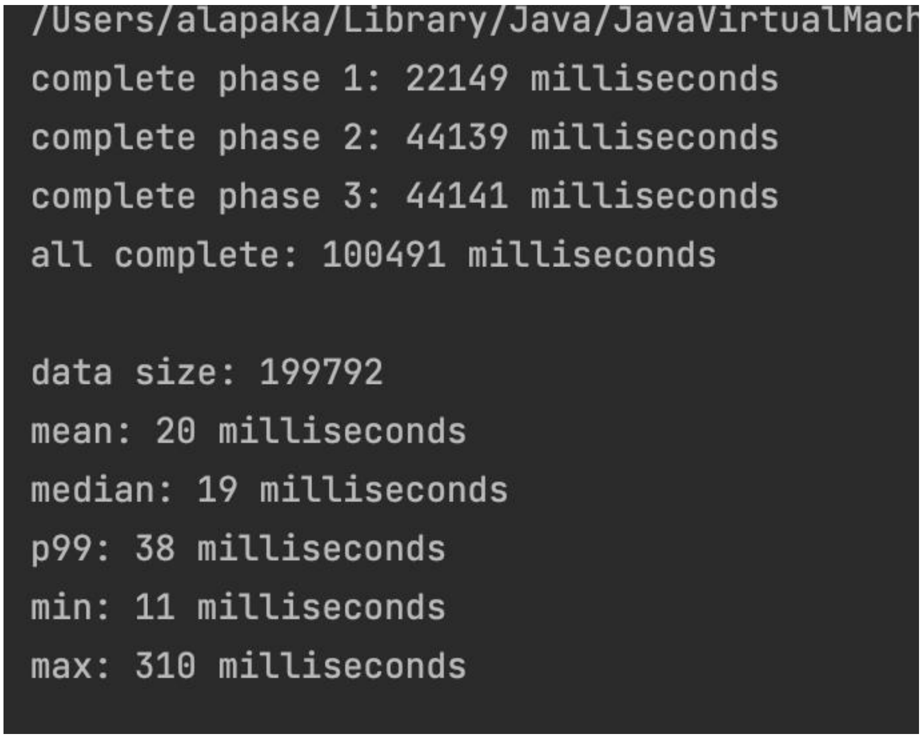
```
/Users/atapaka/Library/Java/JavaVirtualMac
complete phase 1: 81937 milliseconds
complete phase 2: 148371 milliseconds
complete phase 3: 148372 milliseconds
all complete: 363780 milliseconds

data size: 199996
mean: 17 milliseconds
median: 16 milliseconds
p99: 31 milliseconds
min: 10 milliseconds
max: 365 milliseconds
```

64 Threads:



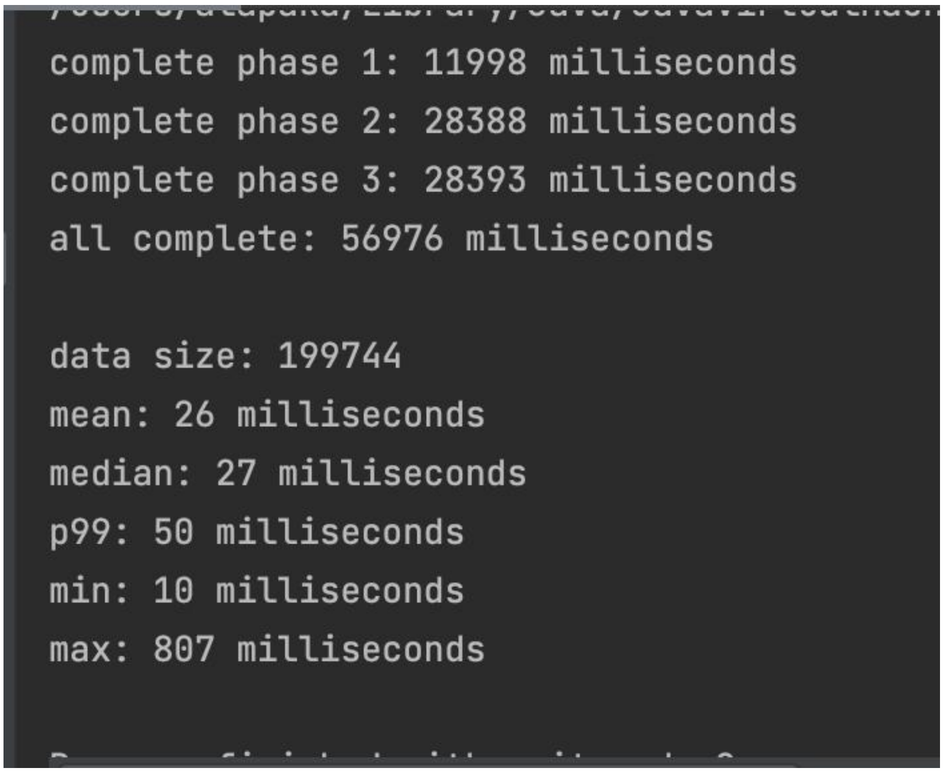
```
/Users/atapaka/Library/Java/JavaVirtualMa
complete phase 1: 41422 milliseconds
complete phase 2: 75860 milliseconds
complete phase 3: 75862 milliseconds
all complete: 184677 milliseconds

data size: 199804
mean: 17 milliseconds
median: 17 milliseconds
p99: 31 milliseconds
min: 11 milliseconds
max: 291 milliseconds
```

128 Threads:



```
/Users/alapaka/Library/Java/JavaVirtualMach
complete phase 1: 22149 milliseconds
complete phase 2: 44139 milliseconds
complete phase 3: 44141 milliseconds
all complete: 100491 milliseconds

data size: 199792
mean: 20 milliseconds
median: 19 milliseconds
p99: 38 milliseconds
min: 11 milliseconds
max: 310 milliseconds
```

256 Threads:



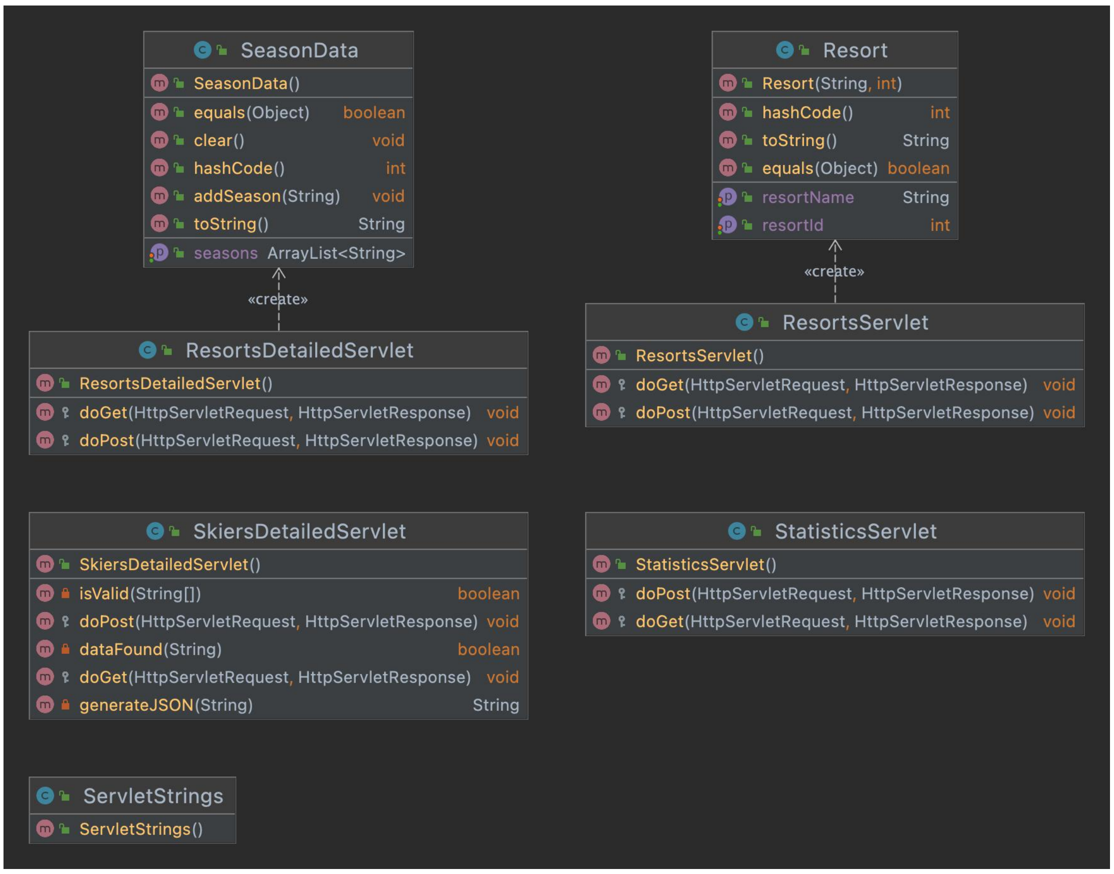
```
[illegible]
complete phase 1: 11998 milliseconds
complete phase 2: 28388 milliseconds
complete phase 3: 28393 milliseconds
all complete: 56976 milliseconds

data size: 199744
mean: 26 milliseconds
median: 27 milliseconds
p99: 50 milliseconds
min: 10 milliseconds
max: 807 milliseconds

[illegible]
```

Now I’ll plot a chart of #thread <--> mean latency

As naive thinking, more threads means slower response to each one. For example, max latency raise dramatically for 256 thread since there are only 200 threads allowed concurrently in tomcat and others has to wait in queue.

#THREAD	32	64	128	256
Mean Latency(ms)	17	17	20	26

Not mentioned in grading but if you need a structure of server side:


SeasonData
SeasonData()
equals(Object) boolean
clear() void
hashCode() int
addSeason(String) void
toString() String
seasons ArrayList<String>
«create»
Resort
Resort(String, int)
hashCode() int
toString() String
equals(Object) boolean
resortName String
resortId int
«create»
ResortsDetailedServlet
ResortsDetailedServlet()
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
ResortsServlet
ResortsServlet()
doGet(HttpServletRequest, HttpServletResponse) void
doPost(HttpServletRequest, HttpServletResponse) void
SkiersDetailedServlet
SkiersDetailedServlet()
isValid(String[]) boolean
doPost(HttpServletRequest, HttpServletResponse) void
dataFound(String) boolean
doGet(HttpServletRequest, HttpServletResponse) void
generateJSON(String) String
StatisticsServlet
StatisticsServlet()
doPost(HttpServletRequest, HttpServletResponse) void
doGet(HttpServletRequest, HttpServletResponse) void
ServletStrings
ServletStrings()

I’m having servlet, bean, string package in this structure.