6.
java version "1.8.0_45"
Java(TM) SE Runtime Environment (build 1.8.0_45-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.45-b02, mixed mode)

Model Name: MacBook Air
Model Identifier: MacBookAir, 62
Processor Name: Intel Core i7
Processor Speed: 1.7 GHz
Number of Processors: 1
Total Number of Cores: 2
L2 Cache (per Core): 256 KB
L3 Cache: 4 MB
Memory: 8 GB

11. In the synchronized model, the swap function is a synchronized function, meaning that a thread has to first acquire the "lock" of the synchronized function before it can use it, and a function unlocks the lock after it is done with the function call. Therefore, data races can not occur because there is only one lock, and thus threads have to wait for each other to finish before they can acquire the lock and use the swap function. And as we can see, there is a lot of overhead involved where the threads are waiting for other threads to be done with swap.

In the BetterSafe model, I avoid data races using the AtomicIntegerArray model along with its atomic decrementAndGet and incrementAndGet functions to carry out the swap function. Because the increment and decrement operations are both atomic, no data races will occur between the threads. The performance is also much better than the synchronized model because threads do not have to wait for the lock, and thus we skip out on all of that overhead.

13. My BetterSorry implementation uses the volatile keyword to make the byte array a volatile array. Other than declaring the byte array as volatile, the rest of the code is the same as in the Unsynchronized model. As I've documented in this report, BetterSorry has higher reliability than Unsynchronized, however still not DRF, and better performance than BetterSafe.
BetterSorry is still vulnerable to data races due to the incrementing and decrementing operations still being non-atomic, however the volatile keyword helps by making sure read operations read from the "main memory" and not from the cached memories of threads. However data races can still occur if threads interleave during the read and write operations.
To make BetterSorry crash frequently, increase the number of threads and number of swap operations and decrease the length of the array. This way, it is

more likely that threads will operate on the same element of the array, and thus data races will occur more frequently and the program will crash more often.

ex: java UnsafeMemory BetterSorry 20 10000000 100 30 41 55 79 41 13 2

****

For each of the models, I characterize its performance and reliability as follows:

java UnsafeMemory Unsynchronized 20 10000000 100 (500 random nums)
Performance: avg ns/transition over 10 runs
Reliability: 100 - (summation (abs(expected - real)/expected))* 100

Performance is simply the average of the (avg ns/transition) over 10 runs.

Reliability was a bit trickier to characterize. I decided to use the difference between the expected sum and the actual sum as the way to determine reliability However, because methods with data races would hang too often, I decided to increase the number of threads to better model multithreading, I also increased the maxval to 100, increased the number of iterations to 10000000, and finally increased the length of the array to length 500 and populated with random nums. The purpose of these changes is so that data races will still occur, however because the length of the array is much longer, it is less likely data races will happen over the same element over and over again, and thus the program will hang less. Therefore it was easier for me to collect data on reliability using my definition of reliability.

A sample call is given below :
java UnsafeMemory GetNSet 20 10000000 100 30 41 55 79 41 13 2 12 75 71 89 95
22 21 22 3 99 34 53 59 6 27 18 91 56 78 12 48 29 92 22 71 36 15 52 85 16 34
100 97 54 73 31 78 23 81 97 5 71 58 15 14 27 49 55 56 64 49 94 28 22 22 88 27
8 87 4 8 31 79 23 14 28 52 40 35 80 93 33 45 19 53 1 90 23 27 26 79 59 25 83
63 21 3 74 64 52 43 12 88 85 93 21 39 64 2 56 53 11 40 6 24 76 26 97 34 13 80
81 13 69 46 41 20 2 68 81 82 9 46 42 93 97 26 94 46 87 18 46 50 13 10 84 52 12
69 39 56 76 35 12 15 21 91 10 18 42 82 94 34 16 74 74 48 79 65 20 92 91 89 16
66 95 90 80 61 36 82 54 20 1 6 8 59 60 70 38 70 56 53 20 17 63 72 5 33 32 31 62
49 98 55 62 86 99 51 47 28 86 86 94 8 63 16 67 11 68 43 2 94 51 6 81 3 54 30
23 70 70 78 89 55 39 34 55 87 40 81 94 34 37 11 26 8 10 76 24 99 78 25 70 17
86 68 55 42 25 57 66 74 57 100 90 83 33 16 59 95 71 77 65 75 89 83 5 69 82 9
2 11 57 24 73 33 72 85 9 18 7 96 18 11 53 55 51 85 41 66 73 48 70 16 68 16 75
48 70 60 84 81 13 31 58 53 83 67 42 56 86 60 62 13 64 42 99 64 8 87 32 74 53

63 67 18 74 4 64 58 40 21 10 78 7 79 70 9 22 96 67 26 2 58 11 55 35 88 43 2
77 89 79 85 42 84 44 41 63 29 40 92 79 53 82 24 90 15 16 67 2 71 97 62 97 9
88 20 86 77 92 53 73 63 22 89 72 56 94 23 57 66 89 94 69 87 81 42 92 56 64
16 44 73 79 6 91 45 34 31 47 100 41 64 33 55 16 17 95 15 49 84 62 32 100
67 17 24 34 61 93 47 25 88 96 56 51 41 66 83 10 76 22 56 58 39 84 23 2 95
32 9 29 74 52 13 65 82 18 55 27 74 90 42 61 13 3 39 25 54 43 24 96 60 22
12 38 38 59 74 47 89 1 45 5 25 10 83 15 5 10 58

****

Synchronized:

Reliability = 100
Performance = 1337.44 ns/transition

Basic synchronized function. Has the slowest performance out of all of the models, and has 100% reliability.

****

Unsynchronized:

Reliability = 99.9256
Performance = 561.5502 ns/transition

The unsynchronized model has the worst reliability, but the fastest perf
This makes sense as the unsynchronized model takes no precaution agaisnt data races, therefore the worst reliability, but also has zero overhead and thus is the fastest.

****

GetNSet:

Reliability = 99.9607
Performance = 878.936 ns/transition

I implemented my GetNSet method with the AtomicIntegerArray and its get and set methods as detailed by the specs. GetNSet has better performance than synchronized, better reliability than unsynchronized, but is not DRF. GetNSet is not DRF because its get and set methods that we use are still not atomic and thus threads can interleave with each other between the get and set.
GetNSet has better performance than Synchronized for the same reasons why BetterSafe has better performance than synchronized as explained above.

****

BetterSafe:

Reliability = 100%
Performance = 841.9897 ns/transition

In the BetterSafe model, I avoid data races using the AtomicIntegerArray model along with its atomic decrementAndGet and incrementAndGet functions to carry out the swap function. Because the increment and decrement operations are both atomic, no data races will occur between the threads. The performance is also much better than the synchronized model because threads do not have to wait for the lock, and thus we skip out on all of that overhead.

I also tried out another implementation for BetterSafe, where I use reentrant locks inside of the swap function. The idea here is that if I only synchronize a portion of the swap function, there should be less overhead than when you synchronize the entire swap function. With this implementation, I found that BetterSafe was only slightly faster.

****

BetterSorry:

Reliability = 99.9904
Performance = 604.5431 ns/transition

My BetterSorry implementation uses the volatile keyword to make the byte array a volatile array. Other than declaring the byte array as volatile, the rest of the code is the same as in the Unsynchronized model. As I've documented in this report, BetterSorry has higher reliability than Unsynchronized, however still not DRF, and better performance than BetterSafe.
BetterSorry is still vulnerable to data races due to the incrementing and decrementing operations still being non-atomic, however the volatile keyword helps by making sure read operations read from the "main memory" and not from the cached memories of threads. However data races can still occur if threads interleave during the read and write operations.

If I were to suggest to GDI a model, I'd suggest to them either BetterSafe or BetterSorry. If performance is by far their main concern, then I'd push for BetterSorry. However if they're willing to compromise on a bit of speed in return for DRF, then definitely BetterSafe.