The following average times were computed by running the code for 8 threads, and 1 million transitions 50 times.

Average time for Null:
120.455 ns/transition
Average time for Synchronized:
3029.60356 ns/transition
Average time for Unsynchronized: N/A (stuck in a deadlock due to race condition)
Average time for GetNSet: 2352.123827 ns/transition
Average time for BetterSafe: 1144.98709 ns/transition
Average time for BetterSorry: 1080.49911 ns/transition

The Synchronized class is reliable but it is not fast. Due to an overhead by using synchronous blocks, it is slow.
The Unsynchronized class is highly unreliable and has race conditions, which cause the code to get stuck in a deadlock. The program does not finish executing for 1 million transitions for 8 threads.
GetNSet uses an AtomicIntegerArray. It is not DRF and it is faster than Synchronized. Since AtomicIntegerArray uses volatile memory by default. However we have a race condition as the set() call by one thread might be overwritten by the set() call in another thread. Both threads might read the if-statement at the same time, and then proceed to update the code but one of them might do it it first, leading to an incorrect value being computed.
BetterSafe is reliable and is faster than both Synchronized and GetNSet.
BetterSorry is more reliable than Unsynchronized and is the fastest compared to all the other classes.

For GDI, BetterSafe would be the best choice as it is the fastest most reliable class.

1.
BetterSafe achieves a better performance than Synchronized due to customization of locks. When we use synchronized blocks, the JMM implicitly uses locks to make the application thread-safe. However when we use locks, we are manually making the application by thread safe, knowing what the application does. Hence, we can achieve a thread safe application with manual locks with a less of a overhead. Re-entrant locks also give the thread waiting the longest access to the lock. Hence our BetterSafe achieves better performance than Synchronized and is also 100% reliable.

2.
My BetterSorry implementation is short-circuited by an if condition. When a[i]<=0 or a[j]>= maxval, a thread might check the condition and return since we do not need to do anything. It is because of this short-circuiting that we have a faster performace of BetterSorry as compared to BetterSafe. BetterSorry however is not threadsafe. Since the if-statement is before we establish a lock, multiple threads might be executing that same statement at the same time. However it is still more reliable than Unsynchronized since when we are actually modifying the array, i.e. when the increment to a[i] or decrement to a[j] is performed, we establish a lock. This gives the application some level of synchronicity.
There is no way that we can increase the probability of the race condition to my knowledge. The race condition is only with loading the value for the if-statement. However, since no threads are writing to it, or acquiring the lock, there will be no race condition.

3.
To do the measurements, I wrote a java test code to run the input 50 times and compute the average of the results. These are shown in the beginning of the report.
The Unsynchronized class is not DRF. A simple command like:
java UnsafeMemory Unsynchronized 8 1000000 6 5 6 3 0 3
will not finish executing. Our code is stuck due to race conditions in the load and store of the byte array.
The GetNSet class is not DRF.
A command like:
java UnsafeMemory GetNSet 8 10000 6 5 6 3 0 3
finishes executing but does not compute the correct answer due to a race condition.
The BetterSafe class is DRF. When a thread begins executing, it acquires a lock on the swap method. Therefore, no other thread has access to this memory. The other threads wait

for this thread to finish executing. Hence, there is no race condition in our BetterSafe class.

The BetterSorry class is not DRF. Due to the if statement being outside the part where it is locked, multiple threads might try to access the byte array. If two different threads reach the if-statement at the same time, they will both try to access the byte array and we would have a race condition. A command that would fail is if we have a huge number of transitions to perform with the same i and j.