Testing Platform (SEASNET):

- java version "1.8.0_112"
- Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
- Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
- 32 cores of Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
- Approximately 65.8 GB of memory total

Note: All states tested with 20 threads, 1 million swaps, 100 random numbers ranging from 0-127

NullState:
No changes were made to this file. This is the dummy implementation used for deducing overhead of the testing framework.
performance:

- Average: 7525.29 ns/transition
- Stdev: 619.79 ns/transition

SynchronizedState:
No changes were made to this file. This uses the synchronized keyword in Java to ensure 100% reliability and is DRF. However, as a result of ensuring reliability there is a performance hit.
performance:

- Average: 11308.40 ns/transition
- Stdev: 960.00 ns/transition

UnsynchronizedState:
This is the same as SynchronizedState with the synchronized keyword removed. One might expect this to work faster but the resulting lack of reliability due to this having data races causes horrible performance. Essentially we can have a deadlock situation or the case where the swap function always returns false resulting in an infinite loop.
performance: (100% unreliable - Mismatch every time)

- Average: 6799.09 ns/transition
- Stdev: 809.50 ns/transition

GetNSetState:
This implements State using an AtomicIntegerArray and the get() and set() functions. However, the manner in which I implemented this file results in it being Non-DRF.
performance: (100% unreliable - Mismatch every time)

- Average: 7693.23 ns/transition
- Stdev: 863.63 ns/transition

BetterSafeState:
I implemented this file using locks. I lock before checking any values and unlock right before returning true or false. This ensures 100% reliability and that this is DRF. However, due to the overhead of locking this results in a performance hit that can lead to performance being slower than synchronized state which is not desirable and also unexpected.
performance:

- Average: 13694.81 ns/transition
- Stdev: 964.29 ns/transition

BetterSorryState:
I implemented this using AtomicInteger. I used an array of AtmoicIntegers and then used the getAndIncrement() and getAndDecrement() functions. This way the accessing and modification is atomic. Furthermore threads can access different parts of the arrays at the same time but this also risks the chance that multiple arrays are modifying the same data thus this is Non-DRF.
performance:

- Average: 6763.70 ns/transition
- Stdev: 997.80 ns/transition

Best choice for GDI:
The BetterSorryState implementation is probably the best for GDI. Some incorrect results is acceptable as GDI is using this for heuristics rather than working on some sensitive data like bank account info where 100% reliability is needed. We want speed. In this regard, BetterSorryState is the best as it is fastest implementation..

BetterSafeState faster and more reliable than Synchronized? :
BetterSafeState is more reliable as I used locks before checking or modifying any values then unlocking after. Therefore the threads can't step on each others feet. However, the overhead from locking and unlocking in each thread causes this to unfortunately run slower than Synchronized.

Comparing BetterSorryState with BetterSafeState and Unsynchronized:
BetterSorryState is faster than BetterSafeState because it is able to have mutiple threads modifying data in the array at the same time as it doesn't have to lock to run like BetterSafeState does. BetterSorryState is more reliable than UnsynchronizedState because it utilizes atomic instructions while UnsynchronizedState does not. So UnsynchronizedState can be preempted in the middle of incrementing or decrementing which means the value of that array element could change before this thread gets to run again while BetterSorry doesn't suffer from this problem as it uses atomic functions. BetterSorryState still suffers from the case where two

different threads try to modify the same location in the array or read while another thread writes.
The following test causes BetterSorryState to fail:
The case where this fails theoretically exists as explained above but relies purely on the chance that multiple threads are accessing the same element at the same exact time. However, I was unable to produce a test that causes this to break although it should.

Problems getting measurements:
The Non-DRF functions usually go into an infinite loop with this command:

```
java UnsafeMemory Synchronized 8 1000000 6 5 6 3
0 3
```

Otherwise they would print a message saying that there is a sum mistmatch. I used the following test script (The numbers were taken from a random number generator) called with 20 threads and 1 million swaps:

```
#!/bin/bash

echo "Null:"
for i in {1..10};
do
	java UnsafeMemory Null $1 $2 127 79 105 32
41 20 43 86 81 125 47 14 69 108 17 122 91 9 73 101
63 74 96 98 2 93 10 18 58 28 60 38 118 110 100 3 12
56 117 120 102 103 109 24 45 35 82 83 124 62 106
11 13 6 5 114 94 1 107 115 61 50 31 75 64 70 48 27
21 59 87 78 88 85 36 0 97 127 54 7 67 126 89 57 121
51 68 116 66 113 80 95 90 33 30 37 34 53 16 49 65 |
grep -Eow "[0-9.]+" | tr '\n' ' '
done
echo ''

echo "Synchronized:"
for i in {1..10};
do
	java UnsafeMemory Synchronized $1 $2 127 79
105 32 41 20 43 86 81 125 47 14 69 108 17 122 91 9
73 101 63 74 96 98 2 93 10 18 58 28 60 38 118 110
100 3 12 56 117 120 102 103 109 24 45 35 82 83 124
62 106 11 13 6 5 114 94 1 107 115 61 50 31 75 64 70
48 27 21 59 87 78 88 85 36 0 97 127 54 7 67 126 89
57 121 51 68 116 66 113 80 95 90 33 30 37 34 53 16
49 65 | grep -Eow "[0-9.]+" | tr '\n' ' '
done
echo ''

echo "BetterSafe: "
for i in {1..10};
do
	java UnsafeMemory BetterSafe $1 $2 127 79
105 32 41 20 43 86 81 125 47 14 69 108 17 122 91 9
73 101 63 74 96 98 2 93 10 18 58 28 60 38 118 110
100 3 12 56 117 120 102 103 109 24 45 35 82 83 124
62 106 11 13 6 5 114 94 1 107 115 61 50 31 75 64 70
48 27 21 59 87 78 88 85 36 0 97 127 54 7 67 126 89
57 121 51 68 116 66 113 80 95 90 33 30 37 34 53 16
49 65 | grep -Eow  "[0-9.]+" | tr '\n' ' '
done
echo ''

echo "BetterSorry: "
for i in {1..10};
do
	java UnsafeMemory BetterSorry $1 $2 127 79
105 32 41 20 43 86 81 125 47 14 69 108 17 122 91 9
73 101 63 74 96 98 2 93 10 18 58 28 60 38 118 110
100 3 12 56 117 120 102 103 109 24 45 35 82 83 124
62 106 11 13 6 5 114 94 1 107 115 61 50 31 75 64 70
48 27 21 59 87 78 88 85 36 0 97 127 54 7 67 126 89
57 121 51 68 116 66 113 80 95 90 33 30 37 34 53 16
49 65 | grep -Eow  "[0-9.]+" | tr '\n' ' '
done
echo ''

echo "Unsychronized: "
for i in {1..10};
do
	java UnsafeMemory Unsynchronized $1 $2 127
79 105 32 41 20 43 86 81 125 47 14 69 108 17 122
91 9 73 101 63\
 74 96 98 2 93 10 18 58 28 60 38 118 110 100 3 12
56 117 120 102 103 109 24 45 35 82 83 124 62 106
11 13 6 5 114 94 1 107 115 61 50 31 75 64 70 48 27
21 59 87 78 88 85 36 0 97 127 54 7 67 126 89 57 121
51 68 116 66 113 80 95 90 33 30 37 34 53 16 49 65 |
grep -Eow "[0-9.]+" | tr '\n' ' '
done
echo ''

echo "GetNSet: "
for i in {1..10};
do
	java UnsafeMemory GetNSet $1 $2 127 79 105
32 41 20 43 86 81 125 47 14 69 108 17 122 91 9 73
101 63 74 96 98 2 93 10 18 58 28 60 38 118 110 100
3 12 56 117 120 102 103 109 24 45 35 82 83 124 62
106 11 13 6 5 114 94 1 107 115 61 50 31 75 64 70 48
27 21 59 87 78 88 85 36 0 97 127 54 7 67 126 89 57
121 51 68 116 66 113 80 95 90 33 30 37 34 53 16 49
65 | grep -Eow "[0-9.]+" | tr '\n' ' '
done
echo ''
```