Stefan Knott
CSCI 4830
HW 3

Problem 1:
1) See code.
2) See code. There is an infinite loop somewhere within my laziest_remove() method but I am unable to figure out how to fix it. Next HW assignment I'm definitely going to use an IDE so I can debug easier.
3) For a small number of threads, no matter what operations you are doing it would always be wise to use a coarse-grained locking algorithm. From my tests I found coarse-grained was always the fastest and had the least amount of involuntary context switches as well. The one test where another algorithm was as good as coarse was when I performed a low amount of add operations and a high amount of contain operations.
For a medium number of threads, the coarse-grained algorithm was still great, however now the lazy algorithm has caught up to coarse's performance. For tests involving high add operations and a mix of any other operation, the lazy algorithm was as fast as the coarse-grained algorithm. One drawback of the lazy algorithm though is that it seems to always have more involuntary context switches than coarse-grained. This makes me believe that coarse-grained is still the way to go. Although for my last test, low add high contains, the fine-grained algorithm was as efficient as the lazy algorithm, but the fine made less involuntary context switches. So this may be an instance where we would want to use a fine-grained lock.
For a high number of threads the data follows a similar trend. Where coarse will be close with lazy but consistently beating it, and in the last test fine-grained will win again as it did in the medium thread test.

When to use what
Coarse-grained: Low number of threads, high add operations relative to remove/contain/etc
Fine-grained: Low add, high contains, and a medium number of threads or more
Lazy: rarely use, could use in place of coarse-grained as they are pretty similar in performance

4) To test the correctness of my concurrent implementation I could make a sequentially functioning program which adds/removes/does the same operations as the concurrent program would. I would just compare the final output of these two programs (sequential and concurrent) and use that to determine correctness. This may be difficult because I could have to wait for the sequential implementation to finish, which may take awhile as I have thousands of add/rem/contains operations in my testing code.

Problem 2:
Read function of a register must return an old value or a new value only if read/write overlap.

```
Public int read(){
        BoolRegister[] c = new BoolMRSWRegister(3*N);
        for(int i = 3 * N-1; I >=0; I--){ //get what is in global reg
                c[i] = b[i];
        }
        if(c[0..N-1] is equal to c[N..2N-1]) {return booleanArrayToInt(c[0..N-1]); //return older value
        }else{return booleanArrayToInt(c[2N..3N-1]);}//return newer value
}
```

Problem 3:SKIPPING. Prof said if someone pointed out an error in the code when we were going over monitors in class that they could skip a hw problem. I found an error relating to fairness/starvation-freedom and am skipping this problem.