```
//from State.java //////////////
interface State {
    int size();
    byte[] current();
    boolean swap(int i, int j);
}


//from SwapTest.java ////////////
import java.util.concurrent.ThreadLocalRandom;

class SwapTest implements Runnable {
    private int nTransitions;
    private State state;

    SwapTest(int n, State s) {
    nTransitions = n;
    state = s;
    }

    public void run() {
    int n = state.size();
    if (n != 0)
        for (int i = 0; i < nTransitions; ) {
        int a = ThreadLocalRandom.current().nextInt(0, n);
        int b = ThreadLocalRandom.current().nextInt(0, n - 1);
        if (a == b)
            b = n - 1;
        if (state.swap(a, b))
            i++;
        }
    }
}


//from SynchronizedState.java ////////////////
class SynchronizedState implements State {
    private byte[] value;
    private byte maxval;

    SynchronizedState(byte[] v) { value = v; maxval = 127; }

    SynchronizedState(byte[] v, byte m) { value = v; maxval = m; }

    public int size() { return value.length; }

    public byte[] current() { return value; }

    public synchronized boolean swap(int i, int j) {
    if (value[i] <= 0 || value[j] >= maxval) {
        return false;
    }
    value[i]--;
    value[j]++;
    return true;
    }
}

//from NullState.java /////////////////
// This is a dummy implementation, useful for
// deducing the overhead of the testing framework.
class NullState implements State {
```

```
    private byte[] value;
    NullState(byte[] v, byte maxval) { value = v; }
    public int size() { return value.length; }
    public byte[] current() { return value; }
    public boolean swap(int i, int j) { return true; }
}


//from UnsynchronizedState.java
class UnsynchronizedState implements State {
    private byte[] value;
    private byte maxval;

    UnsynchronizedState(byte[] v) { value = v; maxval = 127; }

    UnsynchronizedState(byte[] v, byte m) { value = v; maxval = m; }

    public int size() { return value.length; }

    public byte[] current() { return value; }

    public boolean swap(int i, int j) {
    if (value[i] <= 0 || value[j] >= maxval) {
        return false;
    }
    value[i]--;
    value[j]++;
    return true;
    }
}


//from GetNSet.java //////////////////////
import java.util.concurrent.atomic.AtomicIntegerArray;

class GetNSet implements State {
    private AtomicIntegerArray value;
    private byte maxval;

    GetNSet(byte[] v) {
        int len = v.length;
        int [] temp = new int[len];
        for(int i = 0; i < len; i++){
            temp[i] = (int) v[i];
        }
        value = new AtomicIntegerArray(temp);
        maxval = 127;
    }

    GetNSet(byte[] v, byte m) {
        int len = v.length;
        int [] temp = new int[len];
        for(int i = 0; i < len; i++){
            temp[i] = (int) v[i];
        }
        value = new AtomicIntegerArray(temp);
        maxval = m;
    }

    public int size() { return value.length(); }

    public byte[] current() {
```

```
        int len = this.size();
        byte [] temp = new byte[len];
        for(int i = 0; i < len; i++){
            temp[i] = (byte) value.get(i);
        }
        return temp;
    }

    public boolean swap(int i, int j) {
        if(value.get(i) <= 0 || value.get(j) >= maxval){
            return false;
        }
        value.getAndDecrement(i);
        value.getAndIncrement(j);
        return true;
    }
}

//from BetterSave.java ///////
import java.util.concurrent.locks.ReentrantLock;

class BetterSafe implements State {
    private byte[] value;
    private byte maxval;
    private int num_locks;
    private ReentrantLock[] bs_lock;

    BetterSafe(byte[] v){
        value = v;
        maxval = 127;
        num_locks = v.length;
        bs_lock = new ReentrantLock[num_locks];
        for(int i = 0; i < num_locks; i++){
           bs_lock[i] = new ReentrantLock();
        }
    }
    BetterSafe(byte[] v, byte m){
        value = v;
        maxval = m;
        num_locks = v.length;
        bs_lock = new ReentrantLock[num_locks];
        for(int i = 0; i < num_locks; i++){
            bs_lock[i] = new ReentrantLock();
        }
    }

    public int size(){ return value.length; }

    public byte[] current() { return value;}

    public boolean swap(int i, int j) {
        int first;
        int second;
        if (i < j){
            first = i;
            second = j;
        }
        else{
            first = j;
            second = i;
        }
        bs_lock[first].lock();
```

```
        bs_lock[second].lock();
        if (value[i] <= 0 || value[j] >= maxval){
            bs_lock[second].unlock();
            bs_lock[first].unlock();
            return false;
        }
        value[i]--;
        value[j]++;
        bs_lock[second].unlock();
        bs_lock[first].unlock();
        return true;
    }
}


//from UnsafeMemory.java
class UnsafeMemory {
    public static void main(String args[]) {
    if (args.length < 3)
        usage(null);
    try {
        int nThreads = parseInt (args[1], 1, Integer.MAX_VALUE);
        int nTransitions = parseInt (args[2], 0, Integer.MAX_VALUE);
        byte maxval = (byte) parseInt (args[3], 0, 127);
        byte[] value = new byte[args.length - 4];
        for (int i = 4; i < args.length; i++)
        value[i - 4] = (byte) parseInt (args[i], 0, maxval);
        byte[] stateArg = value.clone();
        //System.out.println(stateArg.toString());
        State s;
        if (args[0].equals("Null"))
            s = new NullState(stateArg, maxval);
        else if (args[0].equals("Synchronized"))
            s = new SynchronizedState(stateArg, maxval);
        else if (args[0].equals("Unsynchronized"))
            s = new UnsynchronizedState(stateArg, maxval);
        else if (args[0].equals("GetNSet"))
            s = new GetNSet(stateArg, maxval);
        else if (args[0].equals("BetterSafe"))
            s = new BetterSafe(stateArg, maxval);
        else if (args[0].equals("NullState"))
            s = new NullState(stateArg, maxval);
        else
            throw new Exception(args[0]);
        dowork(nThreads, nTransitions, s);
        test(value, s.current(), maxval);
        System.exit (0);
    } catch (Exception e) {
        usage(e);
    }
    }

    private static void usage(Exception e) {
    if (e != null)
        System.err.println(e);
    System.err.println("Usage: model nthreads ntransitions"
              + " maxval n0 n1 ...\n");
    System.exit (1);
    }

    private static int parseInt(String s, int min, int max) {
    int n = Integer.parseInt(s);
```

```
    if (n < min || n > max)
        throw new NumberFormatException(s);
    return n;
    }

    private static void dowork(int nThreads, int nTransitions, State s)
      throws InterruptedException {
    Thread[] t = new Thread[nThreads];
    for (int i = 0; i < nThreads; i++) {
        int threadTransitions =
        (nTransitions / nThreads
         + (i < nTransitions % nThreads ? 1 : 0));
        t[i] = new Thread (new SwapTest (threadTransitions, s));
    }
    long start = System.nanoTime();
    for (int i = 0; i < nThreads; i++)
        t[i].start ();
    for (int i = 0; i < nThreads; i++)
        t[i].join ();
    long end = System.nanoTime();
    double elapsed_ns = end - start;
    System.out.format("Threads average %g ns/transition\n",
              elapsed_ns * nThreads / nTransitions);
    }

    private static void test(byte[] input, byte[] output, byte maxval) {
    if (input.length != output.length)
        error("length mismatch", input.length, output.length);
    long isum = 0;
    long osum = 0;
    for (int i = 0; i < input.length; i++)
        {
        isum += input[i];
        osum += output[i];
        if (output[i] < 0)
            error("negative output", output[i], 0);
        if (output[i] > maxval)
            error("output too large", output[i], maxval);
        }
    if (isum != osum)
        error("sum mismatch", isum, osum);
    }

    private static void error(String s, long i, long j) {
    System.err.format("%s (%d != %d)\n", s, i, j);
    System.exit(1);
    }
}
```