IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

Course manager:

Course e-portfolio:

Thesis or project title:

Supervisor:

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. ______	______	______@itu.dk
2. ______	______	______@itu.dk
3. ______	______	______@itu.dk
4. ______	______	______@itu.dk
5. ______	______	______@itu.dk
6. ______	______	______@itu.dk
7. ______	______	______@itu.dk

Practical Concurrent and Parallel Programming – Exam 2016

Written by Rasmus Løbner Christensen (rloc@itu.dk)

Part 1

Question 1 (5%):

Running the class *TestLocking0.java* 10 times produces the following output:

Sum is 1564637,000000 and should be 2000000,000000

Sum is 1530436,000000 and should be 2000000,000000

Sum is 1623439,000000 and should be 2000000,000000

Sum is 1524630,000000 and should be 2000000,000000

Sum is 1581357,000000 and should be 2000000,000000

Sum is 1599967,000000 and should be 2000000,000000

Sum is 1555741,000000 and should be 2000000,000000

Sum is 1604552,000000 and should be 2000000,000000

Sum is 1640719,000000 and should be 2000000,000000

Sum is 1566290,000000 and should be 2000000,000000

The question now is whether the class is thread-safe or not. First of all, using ***static synchronized*** locks the whole class, while ***synchronized*** (non-static) only locks the method. Since we are discussing whether the _class_ is thread safe, the ***synchronized*** (non-static) method seems to be lackluster. Surprisingly enough, adding the *static* keyword to the ***addInstance*** method, produces the expected result:

Sum is 2000000,000000 and should be 2000000,000000

The above is my answer to the entire three questions in Question 1.

Question 2 (5%):

The simplest natural (depends on the interpretation I guess) way would be making the class variables volatile, combined with all the methods being ***synchronized***. While this seems like a feasible solution, the problem is that declaring a volatile array (in this case a double array) does not give volatile access to the fields of the array. Therefore the solution would be to make the double array ***atomic***, but java only support ***AtomicIntegerArray*** which obviously would not solve the problem (we are working with doubles not integers). In order to solve this visibility issue one could re-set the array reference to be itself *every* time we set an element in the array, but this would mean that all writes to the array would involve two writes instead of one.

Moving on the next question about scalability. Beside the above mentioned obnoxious double write solution, using ***synchronized*** means that all the threads are using locks, therefore waiting for each other. In lecture 6 (in this course), we saw a slide about the scalability speedup of different solutions, where it was showed that when reaching 16 threads and using ***synchronized*** as thread safety – the result would actually give almost no speedup. As already mentioned the whole ***synchronized*** approach would only be beneficial while using a limited number of threads.

Question 3 proposes a thread-safe scalable solution. The problem here is that while concurrent ***add*** or ***set*** calls might be safe, the situation with concurrent calls to ***add*** _and_ ***set*** might not be, since these could happen at the same time. In order to fix this, a version like the above mentioned ***synchronized*** one would fix this issue (but obviously would not be so scalable).

Question 3 (5%):

In order to maintain a correct value/count of ***totalSize*** one could simply ***synchronize*** the ***add*** method; since all updates to ***totalSize*** happens through ***add***. To maintain visibility across threads, the ***totalSize*** field also has to be ***volatile***.

Further in order to maintain the correctness of ***allLists***, a solution would be to make the ***allList.add(this)*** call in the constructer synchronized (simply wrapping it in a ***synchronized(this) {…}*** statement). Furthermore making the ***allLists*** field ***volatile***.

Part 2

Question 4 (10%):

See implementation (***SortingPipeline.java***).

Question 5 (10%)

The code for ***WrappedArrayDoubleQueue*** is as follows:

```java
class WrappedArrayDoubleQueue implements BlockingDoubleQueue {

  final ArrayBlockingQueue<Double> myQueue = new ArrayBlockingQueue<Double>(50);

  @Override
  public double take() {
    try {
      return myQueue.take();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return 0;
  }

  @Override
  public void put(double item) {
    try {
      myQueue.put(item);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  @Override
  public int size() {
    return myQueue.size();
  }
}
```

The output for a 4-stage pipeline sorting 40 numbers (and debug turned on) is:

0.1 1.1 2.1 3.1 4.1 5.1 6.1 7.1 8.1 9.1 10.1 11.1 12.1 13.1 14.1 15.1 16.1 17.1 18.1 19.1 20.1 21.1 22.1 23.1 24.1 25.1 26.1 27.1 28.1 29.1 30.1 31.1 32.1 33.1 34.1 35.1 36.1 37.1 38.1 39.1

In order to measure the time used to sort an array of 100,000 numbers; the following were outputted from ***SystemInfo()*** as well as calling ***mark7()***:

OS: Windows 8.1; 6.3; amd64

JVM: Oracle Corporation; 1.8.0_60

CPU: Intel64 Family 6 Model 42 Stepping 7, GenuineIntel; 8 "cores"

Date: 2016-01-12T00:10:12+0100

sortPipeline 219.6 ms 42.63 2

Question 6 (10%):

The following code is my implementation of ***BlockingNDoubleQueue***.

```
class BlockingNDoubleQueue implements BlockingDoubleQueue {

  final double[] myQueue;
  volatile int head;
  volatile int tail;
  volatile int size;
  volatile int availableItems;
  volatile int availableSpaces;

  public BlockingNDoubleQueue(int capacity) {
    this.myQueue = new double[capacity];
    this.tail = 0;
    this.head = 0;
    this.availableSpaces = capacity;
    this.availableItems = 0;
  }
  @Override
  public synchronized double take() throws InterruptedException {

    while (availableItems == 0) {
      wait();
    }
    availableItems--;
    double item = myQueue[head];
    head = (head + 1) % myQueue.length;
    availableSpaces++;

    notifyAll();
    return item;
  }
  @Override
  public synchronized void put(double item) throws InterruptedException {
    while (availableSpaces == 0) {
      wait();
    }
    availableSpaces--;
    myQueue[tail] = item;
    tail = (tail + 1) % myQueue.length;
    availableItems++;

    notifyAll();
  }
}
```

The above uses simple (***availableSpace*** and ***availableItems***) counters in order to keep track of the amount of numbers in the different queues. This way it is possible to update the ***head*** and ***tail*** pointers, furthermore, keeping the threads waiting correctly. Inspiration from the ***StmQueues.java*** from this course.

The time measured to sort an array of 100,000 numbers using a pipeline with P=4, including a ***mark7*** measurement:

sortPipeline	1.0 ms	0.02	512

Question 7 (10%):

Not implemented - due to being out of time. If I were to implement this, I would take the same approach as in the other ***DoubleQueues*** I've implemented. Obviously following the given hint, and storing ***head*** and ***tail*** values in dummies (exactly like these values are being stored as 0 in the other approaches!). Since this is unbounded, I think that besides using another data structure, this would look a lot like my ***BlockingNDoubleQueue***, since I believe that an unbounded queue simply is a bounded queue with capacity ***java.lang.Integer.MAX_VALUE***.

Question 8 (10%):

Not implemented.

Question 9 (10%):

Not implemented.

Question 10 (5%):

The code for ***StmBlockingNDoubleQueue***:

```
class StmBlockingNDoubleQueue implements BlockingDoubleQueue {

  private final TxnInteger availableItems, availableSpaces;
  private volatile double[] items;
  private final TxnInteger head, tail;

  public StmBlockingNDoubleQueue(int capacity) {

    this.availableItems = newTxnInteger(0);
    this.availableSpaces = newTxnInteger(capacity);
    this.items = new double[capacity];
    this.head = newTxnInteger(0);
    this.tail = newTxnInteger(0);

  }

  @Override
  public double take() {
    return atomic(() -> {
      if (availableItems.get() == 0) {
        retry();
        return null;     // unreachable
      } else {
        availableItems.decrement();
        double item = items[head.get()];
        head.set((head.get() + 1) % items.length);
        availableSpaces.increment();
        return item;
      }
    });
  }

  @Override
  public void put(double item) throws InterruptedException {

    atomic(() -> {
      if (availableSpaces.get() == 0)
        retry();
      else {
        availableSpaces.decrement();
        items[tail.get()] = item;
        tail.set((tail.get() + 1) % items.length);
        availableItems.increment();
      }
    });
  }

  @Override
  public int size() {
    return 0;
  }

  @Override
  public boolean isEmpty() {
    return atomic(() -> availableItems.get() == 0);
  }

  @Override
  public boolean isFull() {
    return atomic(() -> availableSpaces.get() == 0);
  }
}
```

The above is pretty much inspired 100% from this course ***TestStmQueues.java***, and also reminds of the solution from Question 6. As long as the counters (***availableSpaces*** and ***availableItems***) is maintained correctly, the threads know when to access the queues, further the indexes are being correctly handled.

The time measured to sort an array of 100,000 numbers using a pipeline with P=4, including a ***mark7*** measurement:

OS: Windows 8.1; 6.3; amd64

JVM: Oracle Corporation; 1.8.0_60

CPU: Intel64 Family 6 Model 42 Stepping 7, GenuineIntel; 8 "cores"

Date: 2016-01-12T11:36:45+0100

sortPipeline   1003.0 ms   159.83   2

Question 11 (10%):

The below is my attempt to implement the two-stage actor-based sorting pipeline using the Java Akka library. This does not work though (I have included the code in the submission as well – ***SortingPipelineAKKA.java***).

The whole ***transmit*** method in the Erlang code, was really hard to figure out. Furthermore some of the list operations done in the ***Sorter*** actor might also be wrong. Given a bit more time, I'm pretty sure I would be able to solve this problem. My approach regarding the setup of the two input arrays, is also wrong at the moment I believe.

```
public class SortingPipelineAKKA {

    public static void main(String args[]) throws InterruptedException {

        final ActorSystem system = ActorSystem.create("SortingPipeSystem");

        // TODO: Actors
        final ActorRef First = system.actorOf(Props.create(Sorter.class),
"First");
        final ActorRef Second = system.actorOf(Props.create(Sorter.class),
"Second");
        final ActorRef Echo = system.actorOf(Props.create(Echo.class), "Echo");

        First.tell(new InitMessage(Second), ActorRef.noSender());
        Second.tell(new InitMessage(Echo), ActorRef.noSender());

        int input[] = {4,7,2,8,6,1,5,3};
        int washout[] = {9,9,9,9,9,9,9,9};

        for (int x=0; x<input.length; x++){
            First.tell(new NumMessage(input[x]), ActorRef.noSender());
        }

        for (int x=0; x<washout.length; x++){
            First.tell(new NumMessage(washout[x]), ActorRef.noSender());
        }

}

}

// - MESSAGES ------------------------

class InitMessage implements Serializable {

    public final ActorRef pid;

    public InitMessage(ActorRef pid) {this.pid=pid;}

}

class NumMessage implements Serializable {

    public final int n;

    public NumMessage(int n) {this.n=n;}
```

```java
}

// - ACTORS ---------------------------

class Sorter extends UntypedActor {

    List<Integer> L;
    ActorRef Out;

    @Override
    public void onReceive(Object o) throws Exception {

        // the first stage
        if (o instanceof InitMessage) {
            InitMessage initMsg = (InitMessage) o;
            Out = ((InitMessage) o).pid;
        }

        // later stages
        if (o instanceof NumMessage) {
            NumMessage numMsg = (NumMessage) o;

            while (L.size() < 4) {
                L.add(numMsg.n);
                Collections.sort(L);
                // Out missing??
            }

            L.add(numMsg.n);
            Out.tell(new NumMessage(numMsg.n), getSelf());
        }
    }
}

class Echo extends UntypedActor {

    @Override
    public void onReceive(Object o) throws Exception {
        if (o instanceof NumMessage) {

            NumMessage numMsg = (NumMessage) o;
            System.out.println(numMsg.n);
        }
    }
}
```

Question 12 (10%):
Not implemented.

Appendixes

The code

```
// Pipelined sorting using P>=1 stages, each maintaining an internal
// collection of size S>=1.  Stage 1 contains the largest items, stage
// 2 the second largest, ..., stage P the smallest ones.  In each
// stage, the internal collection of items is organized as a minheap.
// When a stage receives an item x and its collection is not full, it
// inserts it in the heap.  If the collection is full and x is less
// than or equal to the collections's least item, it forwards the item
// to the next stage; otherwise forwards the collection's least item
// and inserts x into the collection instead.

// When there are itemCount items and stageCount stages, each stage
// must be able to hold at least ceil(itemCount/stageCount) items,
// which equals (itemCount-1)/stageCount+1.

// sestoft@itu.dk * 2016-01-10

import org.multiverse.api.StmUtils;
import org.multiverse.api.Txn;
import org.multiverse.api.references.TxnDouble;
import org.multiverse.api.references.TxnInteger;
import org.multiverse.api.references.TxnRef;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.ReentrantLock;
import java.util.function.IntToDoubleFunction;

import static org.multiverse.api.StmUtils.*;

/** -------------------------------------------- **/
/** CLASS SortingPipeline       (le main...)     **/
/** -------------------------------------------- **/
public class SortingPipeline {
  public static void main(String[] args) throws InterruptedException {
    SystemInfo();
    final int count = 100_000;
    final int P = 4;
    final double[] arr = DoubleArray.randomPermutation(count);
    final BlockingDoubleQueue[] queues = new BlockingDoubleQueue[P+1];

    for (int i=0; i<queues.length; i++) {
      queues[i] = new WrappedArrayDoubleQueue();
      //queues[i] = new BlockingNDoubleQueue(50);
      //queues[i] = new StmBlockingNDoubleQueue(50);
    }

    sortPipeline(arr,P,queues);

    // Benchmarking
/*    System.out.println(Mark7("sortPipeline", i -> {
      try {
        return sortPipeline(arr, P, queues);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
```

```
    return 0;
    }));*/
  }

  private static int sortPipeline(double[] arr, int P, BlockingDoubleQueue[]
queues) throws InterruptedException {

    int threadCount = P + 2;
    int stageCount = 1;
    Thread[] threads = new Thread[threadCount];
    int colSize = arr.length / P;

    // creating of P+2 threads
    for (int i=0; i < threadCount; i++) {

      // DoubleGenerator instance
      if (i==0) {
        threads[i] = new Thread(new DoubleGenerator(arr, arr.length,
queues[i]));
        threads[i].start();
        //System.out.println(threads[i].getName() + " has started!");
      }

      // SortedChecker instance
      else if (i==threadCount-1){
        threads[i] = new Thread(new SortedChecker(arr.length, queues[i-1]));
        threads[i].start();
        //System.out.println(threads[i].getName() + " has started!");
      }

      // Sorting instance
      else {
        threads[i] = new Thread(new SortingStage(queues[i-1], queues[i],
colSize, P, stageCount));
        threads[i].start();
        //System.out.println(threads[i].getName() + " has started!");
        stageCount++;
      }
    }

    // waiting for threads to complete ...
    try {
      for (int i=0; i<threadCount; i++) {
        threads[i].join();
        //System.out.println(threads[i].getName() + " joined!");
      }
    } catch (InterruptedException exn) {
      System.out.println("Something went wrong ...");
    }
    return 0;
  }

  /** ------------------------------------------- **/
  /** CLASS SortingStage                          **/
  /** ------------------------------------------- **/
  static class SortingStage implements Runnable {
    private final BlockingDoubleQueue input;
    private final BlockingDoubleQueue output;
    private final int colSize;
```

```java
    private volatile int itemCount;
    private final double[] heap;
    private final int P;
    private final int I;

    public SortingStage(BlockingDoubleQueue input, BlockingDoubleQueue output,
int colSize, int P, int I) {
      this.input = input;
      this.output = output;
      this.colSize = colSize;
      this.heap = new double[colSize];
      this.P = P;
      this.I = I;
      // the below is not a beauty ...
      this.itemCount =(heap.length*P) + (P - I) * colSize;

    }

    public void run() {

      int heapSize = 0;
      while (itemCount > 0) {

        double x = 0;
        try {
          x = input.take();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }

        // heap not full, put "x" into it
        if (heapSize < heap.length) {
          heap[heapSize++] = x;
          DoubleArray.minheapSiftup(heap, heapSize-1, heapSize-1);
        }
        // "x" is small, forward
        else if (x <= heap[0]) {
          try {
            output.put(x);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          itemCount--;
        }
        // forward least, replace with "x"
        else {
          double least = heap[0];
          heap[0] = x;
          DoubleArray.minheapSiftdown(heap, 0, heapSize-1);
          try {
            output.put(least);
          } catch (InterruptedException e) {
            e.printStackTrace();
          }
          itemCount--;
        }
      }
    }
  }
```

```java
  /** ------------------------------------------- **/
  /** CLASS DoubleGenerator                        **/
  /** ------------------------------------------- **/
  static class DoubleGenerator implements Runnable {
    private final BlockingDoubleQueue output;
    private final double[] arr;  // The numbers to feed to output
    private final int infinites;

    public DoubleGenerator(double[] arr, int infinites, BlockingDoubleQueue
output) {
      this.arr = arr;
      this.output = output;
      this.infinites = infinites;
    }

    public void run() {
      for (int i=0; i<arr.length; i++)  // The numbers to sort
        try {
          output.put(arr[i]);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
      for (int i=0; i<infinites; i++)   // Infinite numbers for wash-out
        try {
          output.put(Double.POSITIVE_INFINITY);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
    }
  }

  /** ------------------------------------------- **/
  /** CLASS SortedChecker                          **/
  /** ------------------------------------------- **/
  static class SortedChecker implements Runnable {
    // If DEBUG is true, print the first 100 numbers received
    private final static boolean DEBUG = false;
    private final BlockingDoubleQueue input;
    private final int itemCount; // the number of items to check

    public SortedChecker(int itemCount, BlockingDoubleQueue input) {
      this.itemCount = itemCount;
      this.input = input;
    }

    public void run() {
      int consumed = 0;
      double last = Double.NEGATIVE_INFINITY;
      while (consumed++ < itemCount) {
        double p = 0;
        try {
          p = input.take();
        } catch (InterruptedException e) {
          e.printStackTrace();
        }

        if (DEBUG && consumed <= 100)
          System.out.print(p + " ");
```

```java
      if (p <= last)
        System.out.printf("Elements out of order: %g before %g%n", last, p);
      last = p;
    }
    if (DEBUG)
      System.out.println();
  }
}

// --- Benchmarking infrastructure ---

// NB: Modified to show milliseconds instead of nanoseconds

public static double Mark7(String msg, IntToDoubleFunction f) {
  int n = 10, count = 1, totalCount = 0;
  double dummy = 0.0, runningTime = 0.0, st = 0.0, sst = 0.0;
  do {
    count *= 2;
    st = sst = 0.0;
    for (int j=0; j<n; j++) {
      Timer t = new Timer();
      for (int i=0; i<count; i++)
        dummy += f.applyAsDouble(i);
      runningTime = t.check();
      double time = runningTime * 1e3 / count;
      st += time;
      sst += time * time;
      totalCount += count;
    }
  } while (runningTime < 0.25 && count < Integer.MAX_VALUE/2);
  double mean = st/n, sdev = Math.sqrt((sst - mean*mean*n)/(n-1));
  System.out.printf("%-25s %15.1f ms %10.2f %10d%n", msg, mean, sdev, count);
  return dummy / totalCount;
}

public static void SystemInfo() {
  System.out.printf("# OS:   %s; %s; %s%n",
                    System.getProperty("os.name"),
                    System.getProperty("os.version"),
                    System.getProperty("os.arch"));
  System.out.printf("# JVM:  %s; %s%n",
                    System.getProperty("java.vendor"),
                    System.getProperty("java.version"));
  // The processor identifier works only on MS Windows:
  System.out.printf("# CPU:  %s; %d \"cores\"%n",
                    System.getenv("PROCESSOR_IDENTIFIER"),
                    Runtime.getRuntime().availableProcessors());
  java.util.Date now = new java.util.Date();
  System.out.printf("# Date: %s%n",
    new java.text.SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ssZ").format(now));
}

// Crude wall clock timing utility, measuring time in seconds

static class Timer {
  private long start, spent = 0;
  public Timer() { play(); }
  public double check() { return (System.nanoTime()-start+spent)/1e9; }
  public void pause() { spent += System.nanoTime()-start; }
```

```java
    public void play() { start = System.nanoTime(); }
  }
}

// -----------------------------------------------------------------------

// Queue interface

/** -------------------------------------------- **/
/** Interface BlockingDoubleQueue                **/
/** -------------------------------------------- **/
interface BlockingDoubleQueue {
  double take() throws InterruptedException;
  void put(double item) throws InterruptedException;
  int size();
  boolean isEmpty();
  boolean isFull();
}

class WrappedArrayDoubleQueue implements BlockingDoubleQueue {

  final ArrayBlockingQueue<Double> myQueue = new ArrayBlockingQueue<Double>(50);

  @Override
  public double take() {
    try {
      return myQueue.take();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return 0;
  }

  @Override
  public void put(double item) {
    try {
      myQueue.put(item);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  @Override
  public int size() {
    return myQueue.size();
  }

  @Override
  public boolean isEmpty() {
    return false;
  }

  @Override
  public boolean isFull() {
    return false;
  }

}
```

```java
class BlockingNDoubleQueue implements BlockingDoubleQueue {

  final double[] myQueue;
  volatile int head;
  volatile int tail;
  volatile int size;
  volatile int availableItems;
  volatile int availableSpaces;

  public BlockingNDoubleQueue(int capacity) {

    this.myQueue = new double[capacity];
    this.size = 0;
    this.tail = 0;
    this.head = 0;
    this.availableSpaces = capacity;
    this.availableItems = 0;
  }

  @Override
  public synchronized double take() throws InterruptedException {

    while (availableItems == 0) {
      wait();
    }

    availableItems--;
    double item = myQueue[head];
    head = (head + 1) % myQueue.length;
    availableSpaces++;

    notifyAll();

    return item;

  }

  @Override
  public synchronized void put(double item) throws InterruptedException {

    while (availableSpaces == 0) {
      wait();
    }

    availableSpaces--;
    myQueue[tail] = item;
    tail = (tail + 1) % myQueue.length;
    availableItems++;

    notifyAll();

  }

  @Override
  public synchronized int size() {
    return size;
  }

  @Override
```

```java
  public boolean isEmpty() {
    return false;
  }

  @Override
  public boolean isFull() {
    return false;
  }
}

class StmBlockingNDoubleQueue implements BlockingDoubleQueue {

  private final TxnInteger availableItems, availableSpaces;
  private volatile double[] items;
  private final TxnInteger head, tail;

  public StmBlockingNDoubleQueue(int capacity) {

    this.availableItems = newTxnInteger(0);
    this.availableSpaces = newTxnInteger(capacity);
    this.items = new double[capacity];
    this.head = newTxnInteger(0);
    this.tail = newTxnInteger(0);

  }

  @Override
  public double take() {
    return atomic(() -> {
      if (availableItems.get() == 0) {
        retry();
        return null;    // unreachable
      } else {
        availableItems.decrement();
        double item = items[head.get()];
        head.set((head.get() + 1) % items.length);
        availableSpaces.increment();
        return item;
      }
    });
  }

  @Override
  public void put(double item) throws InterruptedException {

    atomic(() -> {
      if (availableSpaces.get() == 0)
        retry();
      else {
        availableSpaces.decrement();
        items[tail.get()] = item;
        tail.set((tail.get() + 1) % items.length);
        availableItems.increment();
      }
    });
  }

  @Override
  public int size() {
```

```java
    return 0;
  }

  @Override
  public boolean isEmpty() {
    return atomic(() -> availableItems.get() == 0);
  }

  @Override
  public boolean isFull() {
    return atomic(() -> availableSpaces.get() == 0);
  }
}

// ----------------------------------------------------------------------

/** ------------------------------------------- **/
/** CLASS DoubleArray                            **/
/** ------------------------------------------- **/
class DoubleArray {
  public static double[] randomPermutation(int n) {
    double[] arr = fillDoubleArray(n);
    shuffle(arr);
    return arr;
  }

  private static double[] fillDoubleArray(int n) {
    double[] arr = new double[n];
    for (int i = 0; i < n; i++)
      arr[i] = i + 0.1;
    return arr;
  }

  private static final java.util.Random rnd = new java.util.Random();

  private static void shuffle(double[] arr) {
    for (int i = arr.length-1; i > 0; i--)
      swap(arr, i, rnd.nextInt(i+1));
  }

  // Swap arr[s] and arr[t]
  private static void swap(double[] arr, int s, int t) {
    double tmp = arr[s]; arr[s] = arr[t]; arr[t] = tmp;
  }

  // Minheap operations for parallel sort pipelines.
  // Minheap invariant:
  // If heap[0..k-1] is a minheap, then heap[(i-1)/2] <= heap[i] for
  // all indexes i=1..k-1.  Thus heap[0] is the smallest element.

  // Although stored in an array, the heap can be considered a tree
  // where each element heap[i] is a node and heap[(i-1)/2] is its
  // parent. Then heap[0] is the tree's root and a node heap[i] has
  // children heap[2*i+1] and heap[2*i+2] if these are in the heap.

  // In heap[0..k], move node heap[i] downwards by swapping it with
  // its smallest child until the heap invariant is reestablished.

  public static void minheapSiftdown(double[] heap, int i, int k) {
```

```java
    int child = 2 * i + 1;
    if (child <= k) {
      if (child+1 <= k && heap[child] > heap[child+1])
        child++;
      if (heap[i] > heap[child]) {
        swap(heap, i, child);
        minheapSiftdown(heap, child, k);
      }
    }
  }

  // In heap[0..k], move node heap[i] upwards by swapping with its
  // parent until the heap invariant is reestablished.
  public static void minheapSiftup(double[] heap, int i, int k) {
    if (0 < i) {
      int parent = (i - 1) / 2;
      if (heap[i] < heap[parent]) {
        swap(heap, i, parent);
        minheapSiftup(heap, parent, k);
      }
    }
  }
}
```