Lecture 36: Review of Lectures 19-33 (Scope of Exam 2)

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Announcements & Reminders

- Hw 4 - entire written + programming (Checkpoint #2) is due today at 11:59pm

- The Final exam (in Canvas) is Tuesday, May 11th from 9am - 12pm (CST).
  - You may reschedule the exam time if your current time zone is not CST
  - Exam is open notes, slides, handouts, canvas videos
isolated (() -> <body> );

- Isolated construct identifies a critical section
- Two tasks executing isolated constructs are guaranteed to perform them in mutual exclusion
  ➔ Isolation guarantee applies to (isolated, isolated) pairs of constructs, not to (isolated, non-isolated) pairs of constructs
- Isolated constructs may be nested
  - An inner isolated construct is redundant
- Blocking parallel constructs are forbidden inside isolated constructs
  - Isolated constructs must not contain any parallel construct that performs a blocking operation e.g., finish, future get, next
  - Non-blocking async operations are permitted, but isolation guarantee only applies to creation of async, not to its execution
- Isolated constructs can never cause a deadlock
  - Other techniques used to enforce mutual exclusion (e.g., locks — which we will learn later) can lead to a deadlock, if used incorrectly
Object-based isolation

isolated(obj1, obj2, ..., () -> <body>)

- In this case, programmer specifies list of objects for which isolation is required
- Mutual exclusion is only guaranteed for instances of isolated constructs that have a common object in their object lists
  — Serialization edges are only added between isolated steps with at least one common object (non-empty intersection of object lists)
  — Standard isolated is equivalent to “isolated(*)” by default i.e., isolation across all objects
- Inner isolated constructs are redundant — they are not allowed to “add” new objects
Parallel Spanning Tree Algorithm using object-based isolated construct

class V {
    V [] neighbors; // adjacency list for input graph
    V parent; // output value of parent in spanning tree
    boolean makeParent(final V n) {
        return isolatedWithReturn(this, () -> {
            if (parent == null) { parent = n; return true; }
            else return false; // return true if n became parent
        });}
    } // makeParent
}

void compute() {
    for (int i=0; i<neighbors.length; i++) {
        final V child = neighbors[i];
        if (child.makeParent(this))
            async(() -> { child.compute(); });
    }
} // compute
} // class V

root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
Worksheet #19 Abstract Metrics with Object-based Isolated Constructs

Compute the WORK and CPL metrics for this program with an object-based isolated construct. Indicate if your answer depends on the execution order of isolated constructs. Since there may be multiple possible computation graphs (based on serialization edges), try and pick the worst-case CPL value across all computation graphs.

Answer: WORK = 25, CPL = 7.

```java
1. finish() -> {
2.     // Assume X is an array of distinct objects
3.     for (int i = 0; i < 5; i++) {
4.         async() -> {
5.             doWork(2);
6.             isolated(X[i], X[i+1],
7.                 () -> { doWork(1); });
8.             doWork(2);
9.         }); // async
10.     } // for
11. } // finish
```
### java.util.concurrent.AtomicInteger methods and their equivalent isolated constructs (pseudocode)

<table>
<thead>
<tr>
<th>j.u.c.atomic Class and Constructors</th>
<th>j.u.c.atomic Methods</th>
<th>Equivalent HJ isolated statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicInteger</td>
<td>int j = v.get();</td>
<td>int j; isolated (v) j = v.val;</td>
</tr>
<tr>
<td></td>
<td>v.set(newVal);</td>
<td>isolated (v) v.val = newVal;</td>
</tr>
<tr>
<td>AtomicInteger()</td>
<td>int j = v.getAndSet(newVal);</td>
<td>int j; isolated (v) { j = v.val; v.val = newVal; }</td>
</tr>
<tr>
<td>// init = 0</td>
<td>int j = v.addAndGet(delta);</td>
<td>isolated (v) { v.val += delta; j = v.val; }</td>
</tr>
<tr>
<td>AtomicInteger(init)</td>
<td>int j = v.getAndAdd(delta);</td>
<td>isolated (v) { j = v.val; v.val += delta; }</td>
</tr>
<tr>
<td></td>
<td>boolean b =</td>
<td>boolean b; isolated (v)</td>
</tr>
<tr>
<td></td>
<td>v.compareAndSet</td>
<td>if (v.val==expect) {v.val=update; b=true;}</td>
</tr>
<tr>
<td></td>
<td>(expect,update);</td>
<td>else b = false;</td>
</tr>
</tbody>
</table>

Methods in java.util.concurrent.AtomicInteger class and their equivalent HJ isolated statements. Variable v refers to an AtomicInteger object in column 2 and to a standard non-atomic Java object in column 3. val refers to a field of type int.
Worksheet #20: Atomic Variables represent a special (and more efficient) case of object-based isolation

```java
class V {
  V [] neighbors; // adjacency list for input graph
  AtomicReference<V> parent; // output value of parent in spanning tree
  boolean makeParent(final V n) {
    // compareAndSet() is a more efficient implementation of
    // object-based isolation
    return parent.compareAndSet(null, n);
  }
  void compute() {
    for (int i=0; i<neighbors.length; i++) {
      final V child = neighbors[i];
      if (child.makeParent(this))
        async(() -> { child.compute(); });
    }
  }
}

root.parent = root; // Use self-cycle to identify root
finish(() -> { root.compute(); });
```
Read-Write Object-based isolation in HJ

isolated(readMode(obj1),writeMode(obj2), ..., () -> <body> );

- Programmer specifies list of objects as well as their read-write modes for which isolation is required
- Not specifying a mode is the same as specifying a write mode (default mode = read + write)
- Mutual exclusion is only guaranteed for instances of isolated statements that have a non-empty intersection in their object lists such that one of the accesses is in writeMode

Sorted List example
1. public boolean contains(Object object) {
2.     return isolatedWithReturn( readMode(this), () -> {
3.         Entry pred, curr;
4.         ...
5.         return (key == curr.key);
6.     });
7. }
8. }
9. public int add(Object object) {
10.    return isolatedWithReturn( writeMode(this), () -> {
11.        Entry pred, curr;
12.        ...
13.        if (...) return 1; else return 0;
14.    });
15. }
Actor Life Cycle (Lecture 21)

Actor states

- **New**: Actor has been created
  — e.g., email account has been created, messages can be received

- **Started**: Actor can process messages
  — e.g., email account has been activated

- **Terminated**: Actor will no longer processes messages
  — e.g., termination of email account after graduation
What output will be printed if the end-finish operation from slide 15 is moved from line 13 to line 11 as shown below?

```java
1. finish() -> {
2.   int threads = 4;
3.   int numberOfHops = 10;
4.   ThreadRingActor[] ring = new ThreadRingActor[threads];
5.   for(int i=threads-1;i>=0; i--) {
6.     ring[i] = new ThreadRingActor(i);
7.     ring[i].start(); // like an async
8.     if (i < threads - 1) {
9.       ring[i].nextActor(ring[i + 1]);
10.    } }
11. }); // finish
12. ring[threads-1].nextActor(ring[0]);
13. ring[0].send(numberOfHops);   
14.
Deadlock (no output): the end-finish operation in line 11 waits for all the actors started in line 7 to terminate, but the actors are waiting for the message sequence initiated in line 13 before they call exit().
```
Worksheet #22: Analyzing Parallelism in an Actor Pipeline

Consider a three-stage pipeline of actors (as in slide 5), set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below.

Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.

```
protected void process(final Object msg) {
    if (msg == null) {
        exit();
    } else {
        doWork(1); // unit work
    }
    if (nextStage != null) {
        nextStage.send(msg);
    }
}
```

WORK = 300, CPL = 102
Worksheet #23: Synchronized Reply using Pause/Resume

Actors don’t normally require synchronization with other actors. However, sometimes we might want actors to be in sync with one another. Using a DDF and pause/resume, ensure that the SynchSenderActor doesn’t process the next message until notified by the SyncReplyActor that the message was received and processed.

```java
1. class SynchSenderActor
2. extends Actor<Message> {
3.  private Actor otherActor = ...
4.  void process(Msg msg) {
5.    ...
6.    DDF<T> ddf = newDDF();
7.    otherActor.send(ddf);
8.    pause(); // non-blocking
9.    asyncAwait(ddf, () -> {
10.       T synchronousReply = ddf.get();
11.       println("Response received");
12.       resume(); // non-blocking
13.    });
14.    ...
15. } }
```

```java
1. class SynchReplyActor
2. extends Actor<DDF> {
3.  void process(DDF msg) {
4.    ...
5.    println("Message received");
6.    // process message
7.    T responseResult = ...;
8.    msg.put(responseResult);
9.    ...
10. } }
```
Synchronized statements and methods in Java (Lecture 24)

- Every Java object has an associated lock acquired via:
  - synchronized statements
    - synchronized( foo ) { // acquire foo’s lock
      // execute code while holding foo’s lock
    } // release foo’s lock
  - synchronized methods
    - public synchronized void op1() { // acquire ‘this’ lock
      // execute method while holding ‘this’ lock
    } // release ‘this’ lock
- Java language does not enforce any relationship between the object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like a monitor
- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, return, break
    - When an exception is thrown and not caught
Dynamic Order Deadlocks

- There are even more subtle ways for threads to deadlock due to inconsistent lock ordering
  - Consider a method to transfer a balance from one account to another:

    ```java
    public class SubtleDeadlock {
      public void transferFunds(Account from, Account to, int amount) {
        synchronized (from) {
          synchronized (to) {
            from.subtractFromBalance(amount);
            to.addToBalance(amount);
          }
        }
      }
    }
    ```
  - What if one thread tries to transfer from A to B while another tries to transfer from B to A?
    Inconsistent lock order again – Deadlock!
Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that all locks are acquired in the same order
- => no deadlock

```java
public class ObviousDeadlock {
    ...
    public void leftHand() {
        isolated(lock1, lock2) {
            for (int i=0; i<10000; i++)
                sum += random.nextInt(100);
        }
    }
    ...
    public void rightHand() {
        isolated(lock2, lock1) {
            for (int i=0; i<10000; i++)
                sum += random.nextInt(100);
        }
    }
}
```
One possible solution to Worksheet #24

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

1. // Start of thread t0 (main program)
2. sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields
3. // Compute sum1 (lower half) and sum2 (upper half) in parallel
4. final int len = X.length;
5. Thread t1 = new Thread(() -> {
6.     for(int i=0 ; i < len/2 ; i++) sum1+=X[i];});
7. t1.start();
8. Thread t2 = new Thread(() -> {
9.     for(int i=len/2 ; i < len ; i++) sum2+=X[i];});
10. t2.start();
11. int sum = sum1 + sum2; //data race between t0 & t1, and t0 & t2
12. t1.join(); t2.join();
2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements.

1. // Start of thread t0 (main program)
2. sum = 0; // static int field
3. Object a = new ... ;
4. Object b = new ... ;
5. Thread t1 = new Thread(() ->
   { synchronized(a) { sum++; } });
6. Thread t2 = new Thread(() ->
   { synchronized(b) { sum++; } });
7. t1.start();
8. t2.start(); // data race between t1 & t2
9. t1.join(); t2.join();
java.util.concurrent.locks.Lock interface (Lecture 26)

1. interface Lock {
2.     // key methods
3.     void lock(); // acquire lock
4.     void unlock(); // release lock
5.     boolean tryLock(); // Either acquire lock (returns true), or return false if lock is not obtained.
6.         // A call to tryLock() never blocks!
7.     
8.     Condition newCondition(); // associate a new condition
9. }

java.util.concurrent.locks.Lock interface is implemented by java.util.concurrent.locks.ReentrantLock class
java.util.concurrent.locks.ReadWriteLock interface

```java
interface ReadWriteLock {
    Lock readLock();
    Lock writeLock();
}
```

- Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows
  - **Case 1:** a thread has successfully acquired `writeLock().lock()`
    - No other thread can acquire `readLock()` or `writeLock()`
  - **Case 2:** no thread has acquired `writeLock().lock()`
    - Multiple threads can acquire `readLock()`
    - No other thread can acquire `writeLock()`

- `java.util.concurrent.locks.ReadWriteLock` interface is implemented by `java.util.concurrent.locks.ReadWriteReentrantLock` class
Hashtable Example

class Hashtable<K,V> {
    ...
    // coarse-grained, one lock for table
    ReadWriteLock lk = new ReentrantReadWriteLock();
    V lookup(K key) {
        int bucket = hasher(key);
        lk.readLock().lock(); // only blocks writers
        ... read array[bucket] ...
        lk.readLock().unlock();
    }

    void insert(K key, V val) {
        int bucket = hasher(key);
        lk.writeLock().lock(); // blocks readers and writers
        ... write array[bucket] ...
        lk.writeLock().unlock();
    }
}
Worksheet #26 Solution: Use of trylock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 4) instead of synchronized.

Your goal is to write a correct implementation that never deadlocks, unlike the buggy version below (which can deadlock).

Assume that each Account object already contains a reference to a ReentrantLock object dedicated to that object e.g., from.lock() returns the lock for the from object. Sketch your answer using pseudocode.

```java
1. public void transferFunds(Account from, Account to, int amount) {
2.   while (true) {
3.     // assume that trylock() does not throw an exception
4.     boolean fromFlag = from.lock.trylock();
5.     if (!fromFlag) continue;
6.     boolean toFlag = to.lock.trylock();
7.     if (!toFlag) {
45.       from.lock.unlock(); continue; }
8.     try { from.subtractFromBalance(amount);
9.       to.addToBalance(amount); break; }
10.    finally { from.lock.unlock(); to.lock.unlock(); }  
11.  } // while  
12. }
```

Linearizability of Concurrent Objects (Lecture 27)

Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  - Examples: Concurrent Queue, AtomicInteger

Linearizability

- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points.
- An object is linearizable if all its possible executions are linearizable.
Example 2: is this execution linearizable?

$q\text{.enq}(x)$, $q\text{.enq}(y)$, $q\text{.deq}():y$, $q\text{.enq}(x)$, $q\text{.enq}(y)$

Task T1

Task T2

Source: [http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt](http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt)
Worksheet #27: Execution of concurrent implementation of FIFO queue q

Is this a linearizable execution?

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Work on q.enq(x)</td>
<td>Invoke q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td>Work on q.enq(x)</td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>3</td>
<td>Return from q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Invoke q.deq()</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Return x from q.deq()</td>
</tr>
</tbody>
</table>

Yes! Can be linearized as “q.enq(x) ; q.enq(y) ; q.deq():x”
Figure (a)

- Host node \((P_c)\) connected to a cluster of processor nodes \((P_0 \ldots P_m)\)
- Processors \(P_0 \ldots P_m\) communicate via an interconnection network which could be standard TCP/IP (e.g., for Map-Reduce) or specialized for high performance communication (e.g., for scientific computing)

Figure (b)

- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node \((R)\) in the interconnect

Processors communicate by sending messages via an interconnect
Our First MPI Program (mpiJava)

```java
1. import mpi.*;
2. class Hello {
3.    static public void main(String[] args) {
4.        // Init() be called before other MPI calls
5.        MPI.Init(args);
6.        int npes = MPI.COMM_WORLD.Size();
7.        int myrank = MPI.COMM_WORLD.Rank();
8.        System.out.println("My process number is " + myrank);
9.        MPI_Finalize(); // Shutdown and clean-up
10.    }
11.}
```

main() is enclosed in an implicit “for all” --- each process runs a separate instance of main() with “index variable” = myrank
Example of Send and Recv

1. import mpi.*;
2. class myProg {
3.   public static void main( String[] args ) {
4.     int tag0 = 0; int tag1 = 1;
5.     MPI.Init( args ); // Start MPI computation
6.     if ( MPI.COMM_WORLD.rank() == 0 ) { // rank 0 = sender
7.       int loop[] = new int[1]; loop[0] = 3;
8.       MPI.COMM_WORLD.Send( "Hello World!", 0, 12, MPI.CHAR, 1, tag0 );
9.       MPI.COMM_WORLD.Send( loop, 0, 1, MPI.INT, 1, tag1 );
10.   } else { // rank 1 = receiver
11.     int loop[] = new int[1]; char msg[] = new char[12];
12.     MPI.COMM_WORLD.Recv( msg, 0, 12, MPI.CHAR, 0, tag0 );
13.     MPI.COMM_WORLD.Recv( loop, 0, 1, MPI.INT, 0, tag1 );
14.     for ( int i = 0; i < loop[0]; i++ )
15.       System.out.println( msg );
16.   }
17.   MPI.Finalize( ); // Finish MPI computation
18. }
19.}

Send() and Recv() calls are blocking operations
Worksheet #29: MPI send and receive

In the space below, indicate what values you expect the print statement in line 10 to output, assuming that the program is executed with two MPI processes.

1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Send(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI_INT, 0, 1);
10.    System.out.println("a = " + a + " ; b = " + b);
11. }
12. ...

Answer: Nothing! The program will deadlock due to mismatched tags, with process 0 blocked at line 4, and process 1 blocked at line 8.
Non-Blocking Send and Receive Operations (Lecture 30)

• In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations (“I” stands for “Immediate”)

  Request Isend(Object buf, int offset, int count, Datatype type, int dst, int tag) ;  
  Request Irecv(Object buf, int offset, int count, Datatype type, int src, int tag) ;

• Use Wait() to wait for operation to complete (like future get).

  Status Wait(Request request)

• The Wait() operation is declared to return a Status object. In the case of a non-blocking receive operation, this object has the same interpretation as the Status object returned by a blocking Recv() operation.
Worksheet #30: MPI send and receive

In the space below, use the minimum amount of non-blocking communication to reach the print statement in line 10 (assume that the program is executed with two MPI processes).

1. int a[], b[];
2. ...
3. if (MPI.COMM_WORLD.rank() == 0) {
4.     MPI.COMM_WORLD.Isend(a, 0, 10, MPI.INT, 1, 1);
5.     MPI.COMM_WORLD.Send(b, 0, 10, MPI.INT, 1, 2);
6. }
7. else {
8.     Status s2 = MPI.COMM_WORLD.Recv(b, 0, 10, MPI.INT, 0, 2);
9.     Status s1 = MPI.COMM_WORLD.Recv(a, 0, 10, MPI.INT, 0, 1);
10.    System.out.println("a = " + a + " ; b = " + b);
11. }
12. ...
A popular feature of MPI is its family of collective communication operations.

Each collective operation is defined over a communicator (most often, MPI.COMM_WORLD)

- Each collective operation contains an *implicit barrier*. The operation completes and execution continues when all processes in the communicator perform the *same* collective operation.

- A mismatch in operations results in *deadlock* e.g.,

  Process 0: .... MPI.Bcast(...) ....
  Process 1: .... MPI.Bcast(...) ....
  Process 2: .... MPI.Gather(...) ....

A simple example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

```c
void Bcast(Object buf, int offset, int count, Datatype type, int root)
```

```c
```
MPI Reduce

```c
void MPI.COMM_WORLD.Reduce(
    Object sendbuf /* in */,
    int sendoffset /* in */,
    Object recvbuf /* out */,
    int recvoffset /* in */,
    int count /* in */,
    MPI.Datatype datatype /* in */,
    MPI.Op operator /* in */,
    int root /* in */ )
```

```
MPI.COMM_WORLD.Reduce(msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
```
In the space below, indicate what value should be provided instead of ??? in line 6, and how it should depend on myrank.

2. MPI.Init(args);  
3. int myrank = MPI.COMM_WORLD.Rank();  
4. int numProcs = MPI.COMM_WORLD.Size();  
5. int size = ...;  
6. int[] sendbuf = new int[size];  
7. int[] recvbuf = new int[???];  
8. // Each process initializes sendbuf  
9. MPI.COMM_WORLD.Gather(sendbuf, 0, size, MPI.INT,  
10. recvbuf, 0, size, MPI.INT,  
11. 0/*root*/);  
12. ...
13. MPI.Finalize();

Solution: myrank == 0 ? (size * numProcs) : 0
// Main program starts at place 0
asyncAt(place(0), () -> S1);
asyncAt(place(0), () -> S2);
asyncAt(place(1), () -> S3);
asyncAt(place(1), () -> S4);
asyncAt(place(1), () -> S5);
asyncAt(place(2), () -> S6);
asyncAt(place(2), () -> S7);
asyncAt(place(2), () -> S8);
asyncAt(place(3), () -> S9);
asyncAt(place(3), () -> S10);
Worksheet #32: impact of distribution on parallel completion time

1. public void sampleKernel(
2.     int iterations, int numChunks, Distribution dist) {
3.     for (int iter = 0; iter < iterations; iter++) {
4.         finish(() -> {
5.             forseq (0, numChunks - 1, (jj) -> {
6.                 asyncAt(dist.get(jj), () -> {
7.                     doWork(jj);
8.                     // Assume that time to process chunk jj = jj units
9.                 });
10.             });
11.         });
12.     } // for iter
13. } // sample kernel

• Assume an execution with n places, each place with one worker thread
• Will a block or cyclic distribution for dist have a smaller abstract completion time, assuming that all tasks on the same place are serialized with one worker per place?
• Answer: Cyclic distribution because it leads to better load balance (locality was not a consideration in this problem)
What is “Eureka Style” Computation?

• Many optimization and search problems attempts to find a result with a certain property or cost
• Announce when a result has been found
  • An "aha!" moment – Eureka event
  • Can make rest of the computation unnecessary

==> Opportunities for “speculative parallelism”, e.g., Parallel Search, Branch and Bound Optimization, Soft Real-Time Deadlines, Convergence Iterations, . . .
Tree Min Index Search Example

```java
HjExtremaEureka<Integer> eureka = new ExtremaEureka(
    Integer.MAX_VALUE, (Integer i, Integer j) -> j.compareTo(i));
finish(eureka, () -> {
    async(() -> {
        minIndexSearchBody(eureka, rootNode, elemToSearch);
    });
});

private static void minIndexSearchBody(
    HjExtremaEureka<Integer> eureka, Node rootNode, int elemToSearch) throws SuspendableException {
    eureka.check(rootNode.id);
    if (rootNode.value == elemToSearch) {
        eureka.offer(rootNode.id);
    }
    if (rootNode.left != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.left, elemToSearch);
        });
    }
    if (rootNode.right != null) {
        async(() -> {
            minIndexSearchBody(eureka, rootNode.right, elemToSearch);
        });
    }
}
```

Inputs:
- binary tree, T
- id for each node in T, in breadth-first order e.g., root.id = 0, root.left.id = 1, root.right.id = 2, ...
- value for each node in T that is the search target

Outputs:
- calls to offer() update eureka with minimum id found so far (among those that match)
- calls to check() can lead to early termination if the argument is >= than current minimum in eureka
- final value of eureka contains minimum id of node with value == elemToSearch
Worksheet #33: Finding maximal index of goal in matrix

Below is a code fragment intended to find the maximal (largest) index of a goal value that occurs multiple times in the input matrix. What logical error(s) are there in the code?

```java
class AsyncFinishEurekaSearchMaxIndexOfGoal {
    HjEureka eurekaFactory() {
        comparator = (cur, newVal) -> {
            // cur is initially [-1, -1]
            (cur.x==newVal.x) ? (newVal.y - cur.y) : (newVal.x - cur.x)
        }
        return new MaximaEureka([-1, -1], comparator)
    }
    int[] doWork(matrix, goal) {
        val eu = eurekaFactory()
        finish (eu, () -> {
            // eureka registration
            forasync (0, matrix.length - 1, (r) ->
                procRow(matrix(r), r, goal));
        });
        return eu.get()
    }
    void procRow(array, r, goal) {
        for (int c = 0; c < array.length(); c++)
            check([r, c]) // terminate if comparator returns negative
            if goal.match(array(c)) offer([r, c]) // updates cur in eureka
    }
}
```

This code has no logical error, but is inefficient due to starting c at 0 instead of array.length() -1.