Exam

1 Accounting System

Question 1.1

I have written the following test:

```
for (int i = 0; i < 20000; i++) {
    new Thread(() -> {
        accounts.deposit(n - 1, 1);
        int value = accounts.get(n - 1);
        accounts.transfer(n - 1, n - 2, value);
        assert (accounts.get(n - 1) == 0);
    });
    new Thread(() -> {
        accounts.deposit(n - 1, 1);
        int value = accounts.get(n - 1);
        accounts.transfer(n - 1, n - 2, value);
        assert (accounts.get(n - 1) == 0);
    });
}
```

By transfering an amount to an account, checking that what the current value on the account is and then removing that amount 20000 across two `Thread`s we see that the value on the account is not what we expect due to non-threadsafe interleaved transfers.

Question 1.2

I have written the following test:

```
accounts.deposit(n - 1, 99);
new Thread(() -> {
    while (accounts.get(n - 1) != 100) {
    }
    assert (true); // Never run.
});
new Thread(() -> {
    accounts.deposit(n - 1, 1); // Makes .get(n-1) == 100
    accounts.deposit(n - 1, 1);
    accounts.deposit(n - 1, 1);
});
```

This test never succeeds, which it should. This shows that there is a visiblity problem in `UnsafeAccounts`.

Question 1.3

My full implementation of `LockAccounts` is shown below:

```
import java.util.Arrays;

public class LockAccounts implements Accounts {
    private volatile Integer[] accounts;
```

```
public LockAccounts(int n) {
    accounts = new Integer[n];
    Arrays.fill(accounts, 0, accounts.length, 0);
}

public void init(int n) {
    synchronized (accounts) {
        accounts = new Integer[n];
        Arrays.fill(accounts, 0, accounts.length, 0);
    }
}

public int get(int account) {
    synchronized (accounts[account]) {
        return accounts[account];
    }
}

public int sumBalances() {
    synchronized (accounts) {
        int sum = 0;
        for (int i = 0; i < accounts.length; i++) {
            sum += accounts[i];
        }
        return sum;
    }
}

public void deposit(int to, int amount) {
    synchronized (accounts[to]) {
        accounts[to] += amount;
    }
}

public void transfer(int from, int to, int amount) {
    synchronized (accounts[from]) {
        synchronized (accounts[to]) {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
    }
}

public void transferAccount(Accounts other) {
    synchronized (accounts) {
        synchronized (other) {
            for (int i = 0; i < accounts.length; i++) {
                accounts[i] += other.get(i);
            }
        }
    }
}

public String toString() {
```

```
        String res = "";
        if (accounts.length > 0) {
            synchronized (accounts) {
                res = "" + accounts[0];
                for (int i = 1; i < accounts.length; i++) {
                    res = res + " " + accounts[i];
                }
            }
        }
        return res;
    }
}
```

The accounts field is made `volatile` to ensure visibility.

The `int` array has been turned into an `Integer` array in order to enable "striping" by locking just the accounts being modified.

`init`, `sumBalances`, `transferAccount` and `toString` lock the entire `accounts` array since the operations herein involve the entire array and would give inconsistent results otherwise.

`sumBalances` can show inconsistent results since a specific account can be modified while `sumBalances` iterates over the `accounts` array.

My solution **can** deadlock in both `transfer` and `transferAccount` since one thread can obtain (for example in `transfer`) a lock on one account while another thread has a lock on the other related account (and they are both waiting for each other). The same applies to `transferAccount` with two related `Account` objects.

Running `java -ea Runner` gives:

```
class UnsafeAccounts passed sequential tests
class UnsafeAccounts passed concurrent tests
```

Question 1.4

Relevant snippets of `LockAccountsFast` (the rest of the class is identical to `LockAccounts`) below:

```
public class LockAccountsFast implements Accounts {
    private volatile Integer[] accounts;
    private volatile Integer[] sums;
    private static final int threads = 4;

    public LockAccountsFast(int n) {
        [...]
        sums = new Integer[threads];
    }

    [...]

    public int sumBalances() {
        int sum = 0;
        for (int i = 0; i < sums.length; i++) {
            synchronized (sums[i]) {
                sum += suns[i];
            }
        }
```

```java
        return sum;
    }

    public void deposit(int to, int amount) {
        synchronized (accounts[to]) {
            int index = Thread.currentThread().hashCode()
                % sums.length;
            synchronized (sums[index]) {
                accounts[to] += amount;
                sums[index] += amount;
            }
        }
    }

    [...]
}
```

Question 1.5

The full `STMAccounts` can seen below.

```java
import java.util.Arrays;

public class STMAccounts implements Accounts {
    private volatile Integer[] accounts;

    public STMAccounts(int n) {
        accounts = new Integer[n];
        Arrays.fill(accounts, 0, accounts.length, 0);
    }

    public void init(int n) {
        atomic(() -> {
            this.accounts = new Integer[n];
            Arrays.fill(accounts, 0, accounts.length, 0);
        });
    }

    public int get(int account) {
        return atomic(() -> this.accounts[account]);
    }

    public int sumBalances() {
        return atomic(() -> {
            int sum = 0;
            for (int i = 0; i < this.accounts.length; i++) {
                sum += this.accounts[i];
            }
            return sum;
        });
    }

    public void deposit(int to, int amount) {
        atomic(() -> this.accounts[to] += amount);
```

```
    }

    public void transfer(int from, int to, int amount) {
        atomic(() -> {
            this.accounts[from] -= amount;
            this.accounts[to] += amount;
        });
    }

    public void transferAccount(Accounts other) {
        atomic(() -> {
            for (int i = 0; i < accounts.length; i++) {
                this.accounts[i] += other.get(i);
            }
        });
    }

    public String toString() {
        return atomic(() -> {
            String res = "";
            if (this.accounts.length > 0) {
                res = "" + this.accounts[0];
                for (int i = 1; i < this.accounts.length; i++) {
                    res = res + " " + this.accounts[i];
                }
            }
            return res;
        });
    }
}
```

I have been unable to run `Runner` with the multiverse .jar file. The code looks almost identical to the code I wrote in hand-in 9, though.

I have chosen to implement the `sumBalances` that risks competing with other method calls. The other implementation option would use more granular transactions, but would then risk that the array could change while `toString`'ing.

Question 1.6

My full implementation of `CASAccounts` can be seen below:

```
public class CASAccounts implements Accounts {
    private AtomicInteger[] accounts;
    private AtomicInteger sum = new AtomicInteger();

    public CASAccounts(int n) {
        accounts = new AtomicInteger[n];
        for (int i = 0; i < accounts.length; i++) {
            accounts[i] = new AtomicInteger(0);
        }
    }

    public void init(int n) {
        accounts = new AtomicInteger[n];
```

```java
        for (int i = 0; i < accounts.length; i++) {
            accounts[i] = new AtomicInteger(0);
        }
    }

    public int get(int account) {
        return accounts[account].get();
    }

    public int sumBalances() {
        return sum.get();
    }

    public void deposit(int to, int amount) {
        int previous, previousSum;
        do {
            previous = accounts[to].get();
            previousSum = sum.get();
        } while (!(accounts[to].compareAndSet(previous, previous + amount)
                && sum.compareAndSet(previousSum, previousSum + amount)));
    }

    public void transfer(int from, int to, int amount) {
        int previousTo, previousFrom;
        do {
            previousFrom = accounts[from].get();
            previousTo = accounts[to].get();
        } while (!(accounts[from].compareAndSet(previousFrom, previousFrom - amount)
            && accounts[to].compareAndSet(previousTo, previousTo + amount)));
    }

    public void transferAccount(Accounts other) {
        for (int i = 0; i < accounts.length; i++) {
            int previous, otherValue, sumPrevious;
            do {
                previous = accounts[i].get();
                otherValue = other.get(i);
                sumPrevious = sum.get();
            } while (otherValue == other.get(i)
                && !(accounts[i].compareAndSet(previous, otherValue + previous)
                && sum.compareAndSet(sumPrevious, sumPrevious + otherValue)));
        }
    }

    public String toString() {
        String res = "";
        if (accounts.length > 0) {
            res = "" + accounts[0].get();
            for (int i = 1; i < accounts.length; i++) {
                res = res + " " + accounts[i].get();
            }
        }
        return res;
    }
```

```
}
```

I have added the `sum` field which mainains the current total sum of the accounts. `sum` is set using CAS in every method that has to update the sum. The update of `sum` is done in the same `while` loop as the other CAS updates in order to ensure correctness of the values.

No writes are lost in the tests I have performed, and `CASAccounts` passes both sequential and concurrent tests.

The operations cannot be guaranteed to happen in constant time since any given update/set of a value might have to be retried a number of times.

I cannot guarantee that the implementation does not livelock, since i.e. a `transfer` operation between threads could go "back and forth" between values indefinitely.

Question 1.7.1

My implementation of `applyTransactionsLoop` with a helper (`printAccounts`) for printing balances can be seen below:

```java
private static void printAccounts(Accounts accounts, int numberOfAccounts) {
    System.out.println("sumBalances is: " + accounts.sumBalances());
    if (numberOfAccounts <= 100) {
        System.out.println("accounts contain: ");
        for (int i = 0; i < numberOfAccounts; i++) {
            System.out.println("Account " + i + " is: " + accounts.get(i));
        }
    }
}

// Question 1.7.1
private static void applyTransactionsLoop(int numberOfAccounts, int numberOfTransactions,
        Supplier<Accounts> generator) {
    final Accounts accounts = generator.get();
    Stream<Transaction> transaction = IntStream.range(0, numberOfTransactions).parallel()
            .mapToObj((i) -> new Transaction(numberOfAccounts, i));

    transaction.parallel().forEach(t -> {
        if (t.from == -1) {
            accounts.deposit(t.to, t.amount);
        } else {
            accounts.transfer(t.from, t.to, t.amount);
        }
    });
    printAccounts(accounts, numberOfAccounts);
}
```

The output of running the above with n = 10 can be seen below:

```
sumBalances is: 9811
accounts contain:
Account 0 is: 1632
Account 1 is: 40
Account 2 is: 1272
Account 3 is: 992
Account 4 is: 940
Account 5 is: 577
```

```
Account 6 is: 1841
Account 7 is: 1071
Account 8 is: 339
Account 9 is: 1107
```

Question 1.7.2

I have not been able to get my code for this question to compile, but pseudocode and a description of my intended solution can be seen below:

```java
// Question 1.7.2
private static void applyTransactionsCollect(int numberOfAccounts, int numberOfTransactions,
        Supplier<Accounts> generator) {
    Stream<Transaction> transactions = IntStream.range(0, numberOfTransactions).parallel()
            .mapToObj((i) -> new Transaction(numberOfAccounts, i));

    // (Failed) attempt using collect:
    //var collect = transactions
    //  .collect(Collectors.mapping(t -> generator.get(), Accounts::transferAccount));

    // Attempt using map:
    var mapping = transactions.parallel().map(t -> {
            var a = generator.get();
            if (t.from == -1) {
                a.deposit(t.to, t.amount);
            } else {
                a.transfer(t.from, t.to, t.amount);
            }
            return a;
        }).collect(Accounts::transferAccount);
}
```

My understanding is that we want to build an `Accounts` object that is the result of applying all transactions. Using `collect` I wanted to get an initial `Accounts` object using the generator, and then run through all `Transactions` generating `Accounts` that are the representation of applying a `Transaction`, on one `Accounts`object, then finally collecting/folding the `Accounts` into one `Accounts` object using `transferAccount`. Unfortunately, I did not succeed.

I then wanted to do it in a more simple way (in my opinion) using a `map`. I attempted to `map` over all transactions, then applying them to the aggregated `Accounts` object (again initially created using the generator). This results in an `Accounts` object `Stream` representing all applied `Transactions`. I would then be able to flatten all of these `Accounts` into one `Accounts` object using `transferAccount`.

I currently get cryptic type errors on `map`, but I hope that my attempt and explanation of my thought process is worth something.

Question 1.7.3

I've used the `Timer` class from the course material to time the performance of the serial `UnsafeAccounts` and `applyTransactionsLoop` tests using `n = 1000`

Both methods use the `UnsafeAccounts` implementation.

- Running serially through all accounts takes 0.048224722 seconds.
- Running through all transactions takes only 0.018207435 seconds, which is about a 2.5x speed-up.

The serial run will be slower due to the fact that we cannot execute work concurrently. Given more threads, more `Accounts` can be processed in a shorter amount of time. There does not seem to be an overhead in having many `Transaction` objects in memory, nor does there seem to be a big overhead in using the Stream API, which is to be expected.

The amount of `numberOfTransactions` can be raised to 200000 before the execution time of the sequential and stream-based approaches begin to look alike. If `n` is raised, then the two approaches diverge again and the stream-based approach is much faster again.

2 Buffered Merging Priority Queue

Question 2.1

Results of benchmarking the class can be seen below:

```
Default parameters:
n 10000000  s 45678, extract 2500000  bufLen 20  maxDepth 4  cutOff 4
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:43:18+0100
-1073351325 One Serial                     Real time:     1.040 (    1.022)
-1073351325 One Parallel                   Real time:     0.455 (    0.445)
-1073351325 BufferedPQ Ser/Serial          Real time:     0.911 (    0.695)

One Parallel:           ~50% speed-up
BufferedPQ Ser/Serial:  ~10% speed-up


Big bufLen:
n 10000000  s 45678, extract 2500000  bufLen 10000  maxDepth 4  cutOff 4
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:49:17+0100
-1073351325 One Serial                     Real time:     1.023 (    1.003)
-1073351325 One Parallel                   Real time:     0.456 (    0.446)
-1073351325 BufferedPQ Ser/Serial          Real time:     0.878 (    0.712)

One Parallel:           ~45% speed-up
BufferedPQ Ser/Serial:  ~15% speed-up


Small bufLen:
n 10000000  s 45678, extract 2500000  bufLen 10  maxDepth 4  cutOff 4
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:50:18+0100
-1073351325 One Serial                     Real time:     1.067 (    1.051)
-1073351325 One Parallel                   Real time:     0.457 (    0.447)
-1073351325 BufferedPQ Ser/Serial          Real time:     0.935 (    0.703)

One Parallel:           ~45% speed-up
```

```
BufferedPQ Ser/Serial:  ~10% speed-up


Big cutOff:
n 10000000  s 45678, extract 2500000  bufLen 20  maxDepth 4  cutOff 12
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:51:03+0100
-1073351325 One Serial                    Real time:     1.019 (    1.002)
-1073351325 One Parallel                  Real time:     0.459 (    0.448)
-1073351325 BufferedPQ Ser/Serial         Real time:     0.922 (    0.704)

One Parallel:           ~45% speed-up
BufferedPQ Ser/Serial:  ~10% speed-up


Small cutOff:
n 10000000  s 45678, extract 2500000  bufLen 20  maxDepth 4  cutOff 1
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:51:33+0100
-1073351325 One Serial                    Real time:     1.023 (    1.006)
-1073351325 One Parallel                  Real time:     0.465 (    0.454)
-1073351325 BufferedPQ Ser/Serial         Real time:     0.927 (    0.706)

One Parallel:           ~50% speed-up
BufferedPQ Ser/Serial:  ~10% speed-up


Huge n:
n 100000000  s 45678, extract 25000000  bufLen 20  maxDepth 4  cutOff 4
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:52:18+0100
-1073571373 One Serial                    Real time:    11.857 (   11.779)
-1073571373 One Parallel                  Real time:     5.573 (    5.541)
-1073571373 BufferedPQ Ser/Serial         Real time:    10.794 (    9.342)

One Parallel:           ~50% speed-up
BufferedPQ Ser/Serial:  ~10% speed-up



Tiny n:
n 1000000  s 45678, extract 250000  bufLen 20  maxDepth 4  cutOff 4
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T18:53:17+0100
-1073155617 One Serial                    Real time:     0.125 (    0.121)
-1073155617 One Parallel                  Real time:     0.212 (    0.212)
```

```
-1073155617 BufferedPQ Ser/Serial          Real time:     0.111 (    0.065)

One Parallel:             NO  speed-up
BufferedPQ Ser/Serial:  ~10% speed-up
```

It is interesting to see that there is a slowdown on small n. The overhead of multithreading is simply too big to outweigh the single-core speed. Otherwise the results are quite consistent with the biggest speed-up seen using *One Parallel*.

Question 2.2

Implementation can be seen below:

```java
public void createPairParam(Parameters param, Function<Parameters, PQ> instanceCreator) {
    var executor = Executors.newWorkStealingPool();
    var tasks = new ArrayList<Callable<PQ>>();
    tasks.add(() -> left = instanceCreator.apply(param.left()));
    tasks.add(() -> right = instanceCreator.apply(param.right()));
    try {
        executor.invokeAll(tasks);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
};
```

Question 2.3

My initial implementation can be seen below. Parts of the code have been omitted since it is identical to `BufferedPQ`.

```java
public class BufferedPQP implements PQ {
    // buffer containing the elements in order
    private int[] buffer;
    private int[] nextBuffer;
    private ExecutorService executorService = Executors.newWorkStealingPool();

    [...]

    BufferedPQP(Parameters pp) {
        nextBuffer = getNewBuffer(bufLen);
        [...]
    }

    [...]

    public int getMin() {
        int res = peek();
        current++;
        if (current >= buffer.length) { // the buffer is empty
            buffer = nextBuffer;
            executorService.submit(() -> nextBuffer = getNewBuffer(bufLen));
            current = 0;
        }
        return res;
```

```
    }

    [...]
}
```

Question 2.5

```
n 10000000  s 45678, extract 2500000  bufLen 20  maxDepth 4  cutOff 4
# OS:   Mac OS X; 10.14.2; x86_64
# JVM:  Oracle Corporation; 11.0.1
# CPU:  null; 4 "cores"
# Date: 2018-12-18T21:27:45+0100
-1073351325 One Serial                    Real time:     1.002 (    0.986)
-1073351325 One Parallel                  Real time:     0.456 (    0.445)
-1073351325 BufferedPQ Ser/Serial         Real time:     0.909 (    0.693)
-1073351325 BufferedPQ Par/Parallel       Real time:     0.532 (    0.335)
-2094454890 BufferedPQP                   Real time:     0.785 (    0.716)
-1814981005 BufferedPQP ParallelPair      Real time:     0.420 (    0.341)
```

We see that the `ParallelPQPair` implementation halves the execution time for the `BufferedPQ` implementation and the `BufferedPQP`, which is to be expected considering that we divide the work into two concurrent tasks in the `Pair`. As we have seen previously in the course, using a `workStealingPool` is efficient since we use all available processors as the parallelism level for the `ExecutorService`. We therefore do not need to implement checking the available number of threads on the CPU of the machine and adjust accordingly by creating manual `Threads`. Furthermore creating a task and submitting it to the `ExecutorService` is fast and takes little memory, whereas creating a `Thread` is slow and takes much memory.

3 Message Passing

I have implemented the Erlang reference implementation in *Java+Akka* according to spec using Java 10 (which lets me use `var` type declarations).

My full implementation of the Erlang reference implementation can be seen below:

```java
import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

import akka.actor.*;

class MergeSort {
    public static void main(String[] args) {
        final var system = ActorSystem.create("MergeSortPipelineSystem");
        final var tester = system.actorOf(Props.create(TesterActor.class));
        final var sorter = system.actorOf(Props.create(SorterActor.class));
        tester.tell(new InitMessage(sorter), ActorRef.noSender());
    }
}

// -- Actors

class SorterActor extends UntypedActor {
```

```
    public void onReceive(Object o) throws Exception {
        if (o instanceof SortMessage) {
            var list = ((SortMessage) o).list;
            var x = ((SortMessage) o).receiver;

            if (list.size() > 1) {
                var m = getContext().actorOf(Props.create(MergerActor.class));
                m.tell(new ResultMessage(x), ActorRef.noSender());

                var l1 = list.subList(0, list.size() / 2);
                var l2 = list.subList(list.size() / 2, list.size());

                var s1 = getContext().actorOf(Props.create(SorterActor.class));
                s1.tell(new SortMessage(l1, m), ActorRef.noSender());

                var s2 = getContext().actorOf(Props.create(SorterActor.class));
                s2.tell(new SortMessage(l2, m), ActorRef.noSender());
            }

            else {
                x.tell(new SortedMessage(list), ActorRef.noSender());
            }
        }
    }
}

class MergerActor extends UntypedActor {
    private ActorRef receiver;
    private List<Integer> l1;
    private List<Integer> l2;

    private List<Integer> merge(List<Integer> l1, List<Integer> l2) {
        var left = new ArrayDeque<Integer>(l1); var right = new ArrayDeque<Integer>(l2);
        var result = new ArrayList<Integer>();
        while (!left.isEmpty() && !right.isEmpty()) {
            if (left.peek().compareTo(right.peek()) > 0) {
                result.add(right.poll());
            } else {
                result.add(left.poll());
                var temp = left; left = right; right = temp; // Swap
            }
        }
        result.addAll(left); result.addAll(right); // Add remainder (if any)
        return result;
    }

    public void onReceive(Object o) throws Exception {
        if (o instanceof ResultMessage) {
            this.receiver = ((ResultMessage) o).receiver;
        }

        else if (o instanceof SortedMessage) {
            // Since we can't do nested onReceive, we do stateful receiver and
            // list building on this actor.
```

```java
            if (receiver == null)
                return;
            if (l1 == null) {
                l1 = ((SortedMessage) o).sorted;
            } else {
                l2 = ((SortedMessage) o).sorted;
                var sorted = merge(l1, l2);
                System.out.println("Merged: " + sorted);
                receiver.tell(new SortedMessage(sorted), ActorRef.noSender());
            }
        }
    }
}

class TesterActor extends UntypedActor {
    public void onReceive(Object o) throws Exception {
        if (o instanceof InitMessage) {
            var sorter = ((InitMessage) o).sorter;
            //Hardcoded list as in .erl example, changed in test runs.
            var list = Arrays.asList(new Integer[] { 8, 7, 6, 5, 4, 3, 2, 1 });
            sorter.tell(new SortMessage(list, getSelf()), ActorRef.noSender());
        } else if (o instanceof SortedMessage) {
            System.out.println("RESULT: " + ((SortedMessage) o).sorted);
        }
    }
}

// -- Messages

class InitMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    public final ActorRef sorter;

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

class SortMessage implements Serializable {
    private static final long serialVersionUID = 2L;
    public final List<Integer> list;
    public final ActorRef receiver;

    public SortMessage(List<Integer> list, ActorRef receiver) {
        this.list = list;
        this.receiver = receiver;
    }
}

class ResultMessage implements Serializable {
    private static final long serialVersionUID = 3L;
    public final ActorRef receiver;

    public ResultMessage(ActorRef receiver) {
```

```java
        this.receiver = receiver;
    }
}

class SortedMessage implements Serializable {
    private static final long serialVersionUID = 4L;
    public final List<Integer> sorted;

    public SortedMessage(List<Integer> sorted) {
        this.sorted = sorted;
    }
}
```

Test input and results can be seen below:

```
# Input:
[8, 7, 6, 5, 4, 3, 2, 1]

# Result:
Merged: [7, 8]
Merged: [3, 4]
Merged: [5, 6]
Merged: [1, 2]
Merged: [5, 6, 7, 8]
Merged: [1, 2, 3, 4]
Merged: [1, 2, 3, 4, 5, 6, 7, 8]
RESULT: [1, 2, 3, 4, 5, 6, 7, 8]

# Input:
[8, 8, 8, 8, 8, 8, 2, 1]

# Result:
Merged: [8, 8]
Merged: [8, 8]
Merged: [8, 8]
Merged: [8, 8, 8, 8]
Merged: [1, 2]
Merged: [1, 2, 8, 8]
Merged: [1, 2, 8, 8, 8, 8, 8, 8]
RESULT: [1, 2, 8, 8, 8, 8, 8, 8]

# Input:
[8, 8, 8, 8, 1, 1, 1, 1]

#Result:
Merged: [1, 1]
Merged: [8, 8]
Merged: [1, 1]
Merged: [8, 8]
Merged: [8, 8, 8, 8]
Merged: [1, 1, 1, 1]
Merged: [1, 1, 1, 1, 8, 8, 8, 8]
RESULT: [1, 1, 1, 1, 8, 8, 8, 8]
```