Learning objective: parallelize algorithm using the fork-join mechanism (with a divide-and-conquer approach).

All exercises are essential.

Exercise 1: fork-join prediction

Let consider the following fork-join task.

```java
class RecAction extends RecursiveAction {
    int work = 0;
    RecAction(int work) {
        this.work = work;
    }
    protected void compute() {
        if (work <= 1000) {
            System.out.println("Working on " + 
                Thread.currentThread() + " with work " + work);
            Thread.sleep(1000);
        } else {
            System.out.println("Splitting work on " + 
                Thread.currentThread() + " with work " + work);
            RecAction otherAction = new RecAction(work / 2);
            otherAction.fork();
            RecAction currAction = new RecAction(work / 2);
            currAction.compute();
        }
    }
}
```

How to invoke it with a ForkJoinPool with 2 threads? What would be the expected output with input 4000? When is any thread stealing tasks?

We can invoke it with two non-blocking solutions:

```java
new ForkJoinPool(2).execute(new RecAction(4000))
new ForkJoinPool(2).submit(new RecAction(4000))
```

The second method returns a future, which can be used to check the completion of the task.

We can also invoke it with the blocking `invoke` method. However, in this case it makes sense to add `otherAction.join()` at the end of `compute` if we want to make sure we proceed once all tasks has been completed.

The expected output is:
Splitting work on Thread1 with work 4000
Splitting work on Thread1 with work 2000
Working on Thread1 with work 1000
Splitting work on Thread2 with work 2000
Working on Thread2 with work 1000

And after one second:

Working on Thread1 with work 1000
Working on Thread2 with work 1000

The second thread steals the second piece of work of size 2000 from the first thread. This occurs just after the first thread performs its first split. At this moment, the first thread has two tasks (among which it continues processing the first task) while the second thread has no task.

Exercise 2: fork-join Fibonacci
Write a Fibonacci implementation relying on a fork-join pool.

class Fibonacci extends RecursiveTask<Integer> {
    int n;
    Fibonacci(int n) { this.n = n; }
    protected Integer compute() {
        if (n <= 1)
            return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}

Exercise 3: fork-join array increment
Implement an algorithm that increments each value of an array by one and that relies on a fork-join pool.

class IncrementTask extends RecursiveAction {
    long[] array; int lo, hi;
    IncrementTask(long[] array, int lo, int hi) {
        this.array = array; this.lo = lo; this.hi = hi;
    }
    protected void compute() {
        if (hi - lo < THRESHOLD) {
            for (int i = lo; i < hi; ++i)
                array[i]++;
        } else {
            int mid = (lo + hi) >>> 1;
            invokeAll(new IncrementTask(array, lo, mid),
                       new IncrementTask(array, mid, hi));
        }
    }
}
Exercise 4: fork-join sum of squares
Implement an algorithm that sums the squares of all values in an array using a fork-join pool.

```java
class SumOfSquaresTask extends RecursiveTask<Integer> {
    long[] array; int lo, hi;
    SumOfSquaresTask(long[] array, int lo, int hi) {
        this.array = array; this.lo = lo; this.hi = hi;
    }
    protected Integer compute() {
        if (hi != lo) {
            int mid = (lo + hi) >>> 1;
            SumOfSquaresTask left =
                new SumOfSquaresTask(array, lo, mid);
            left.fork();
            SumOfSquaresTask right =
                new SumOfSquaresTask(array, mid, hi);
            return left.join() + right.compute();
        } else {
            return array[lo] * array[lo];
        }
    }
}
```

Exercise 5: fork-join array sort
Implement the merge sort using a fork-join pool.

```java
static class SortTask extends RecursiveAction {
    long[] array; int lo, hi;
    SortTask(long[] array, int lo, int hi) {
        this.array = array; this.lo = lo; this.hi = hi;
    }
    SortTask(long[] array) { this(array, 0, array.length); } 
    protected void compute() {
        if (hi - lo < THRESHOLD)
            sortSequentially(lo, hi);
        else {
            int mid = (lo + hi) >>> 1;
            invokeAll(new SortTask(array, lo, mid),
                        new SortTask(array, mid, hi));
            merge(lo, mid, hi);
        }
    }
}
```
// implementation details follow:
static int THRESHOLD = 1000;
void sortSequentially(int lo, int hi) {
    Arrays.sort(array, lo, hi);
}
void merge(int lo, int mid, int hi) {
    long[] buf = Arrays.copyOfRange(array, lo, mid);
    for (int i = 0, j = lo, k = mid; i < buf.length; j++)
        array[j] = (k == hi || buf[i] < array[k]) ?
            buf[i++] : array[k++];
}