CS2030 Programming Methodology
Semester 2 2018/2019

12 April – 18 April 2019
Tutorial 9 Suggested Answers
Parallel Streams
Fork/Join Framework

1. What is the outcome of the following stream pipeline?

```
Stream.of(1, 2, 3, 4)
    .reduce(0, (result, x) -> result * 2 + x);
```

What happens if we parallelize the stream? Explain.

Running the stream sequentially gives 26 since the pipeline evaluates

```
((((0 * 2 + 1) * 2 + 2) * 2 + 3) * 2 + 4)
```

A possible parallel run (with output from each `reduce` operation) gives 18.

```
0 * 2 + 4 = 4 : ForkJoinPool.commonPool-worker-370
0 * 2 + 3 = 3 : main
0 * 2 + 2 = 2 : ForkJoinPool.commonPool-worker-441
0 * 2 + 1 = 1 : ForkJoinPool.commonPool-worker-299
3 * 2 + 4 = 10 : main
1 * 2 + 2 = 4 : ForkJoinPool.commonPool-worker-299
4 * 2 + 10 = 18 : ForkJoinPool.commonPool-worker-299
18
```

Notice that reduction with the identity value `0` is happens for all four stream elements. This is followed by reducing `(1, 2)` to give `4`, and reducing `(3, 4)` to give `10`. Finally, reducing `(4, 10)` gives `18`.

The above stream cannot be parallelized because `2 * result + x` is not associative, i.e. the order of reduction matters.

2. Consider the following `RecursiveTask` called `BinSearch` that finds an item within a sorted array using binary search.

```
class BinSearch extends RecursiveTask<Boolean> {
    int low;
    int high;
    int toFind;
    int[] array;

    BinSearch(int low, int high, int toFind, int[] array) {
```

```
        this.low = low;
        this.high = high;
        this.toFind = toFind;
        this.array = array;
    }

    protected Boolean compute() {
        if (high - low <= 1) {
            return array[low] == toFind;
        } else {
            int middle = (low + high)/2;
            if (array[middle] > toFind) {
                BinSearch left = new BinSearch(low, middle, toFind, array);
                left.fork();
                return left.join();
            } else {
                BinSearch right = new BinSearch(middle, high, toFind, array);
                return right.compute();
            }
        }
    }
}
```

As an example,

```
int[] array = {1, 2, 3, 4, 6};
new BinSearch(0, array.length, 3, array).compute(); // return true
new BinSearch(0, array.length, 5, array).compute(); // return false
```

Assuming that we have a large number of parallel processors in the system and we never run into stack overflow, comment on how `BinSearch` behaves in the following situations:

i. Replace the statements

   ```
   left.fork();
   return left.join();
   ```

   with

   ```
   return left.compute();
   ```

ii. Swap the order of `fork()` and `join()`, i.e. replace

   ```
   left.fork();
   return left.join()
   ```

   with

```
left.join();
return left.fork();
```

iii. Searching for the largest element versus searching for the smallest element in the input array.

`BinSearch` should not be parallelized since we always either search the left half or search the right half, but never both at the same time.

In the given code, we could just call `left.compute()` instead of `left.fork()` then `return left.join()`. This reduces the overhead of interacting with the `ForkJoinPool` and therefore (i) will likely achieve a faster performance.

(ii) is obviously wrong since calling `left.join()` before `left.fork()` would cause the task to block.

(iii) requires an algorithmic understanding of the code in `compute()` method. If the item that we are looking for is smaller than the middle element, we search on the left. This means that the array is sorted in increasing order. So, searching for the smallest element would lead to the code to keep going left (i.e. keep forking and joining). On the other hand, searching for the largest element would lead to the code to keep going to right (i.e. keep invoking `compute()` which is faster). So, searching for the largest element is faster.

3. Given below is the classic recursive method to obtain the n^{th} term of the Fibonacci sequence *without memoization*

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

```
static int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

(a) Parallelize the above implementation by transforming the above to a recursive task and inherit from `java.util.concurrent.RecursiveTask`

```
import java.util.concurrent.RecursiveTask;

class Fib extends RecursiveTask<Integer> {

    final int n;

    Fib(int n) {
        this.n = n;
    }

    @Override
    protected Integer compute() {
        if (n <= 1) {
            return n;
        }

        Fib f1 = new Fib(n - 1);
        Fib f2 = new Fib(n - 2);

        // try different variants here...
    }
}
```

(b) Explore different variants and combinations of `fork`, `join` and `compute` invocations.

(a) `f1.fork();`
`return f2.compute() + f1.join();`
This is what students would probably come up with following the lecture example.

(b) `f1.fork();`
`return f1.join() + f2.compute();`
This works, but slow, since in Java subexpressions are evaluated left to right, i.e. for A + B, A is evaluated first before B (by the away this has nothing to do with associativity). So `f1.join()` needs to wait for `f1.fork()` to complete before `f2.compute()` can be evaluated. Compare this with (a) where `f2.compute()` proceeds while `f1.fork()` is running.

(c) `return f1.compute() + f2.compute();`
This is sequentially recursive. Not much different from (b), but still slightly faster as there is no overhead involved in forking and joining. Everything is done by the main thread.

(d) `f1.fork();`
`f2.fork();`
`return f2.join() + f1.join();`
Apart from the first recursion, main thread delegates all other work to worker threads in the common pool.

(e) `f1.fork();`
`f2.fork();`
`return f1.join() + f2.join();`
Looks the same as (d), but (d) still preferred as it follows the convention of joins to be returned innermost first. Since a thread forks tasks to the front of its own double-ended queue, the last task forked should be the one that is joined when the thread becomes idle; tasks at the back of the dequeue are stolen by other idle worker threads. This becomes apparent when we set the parallelism level to 0 using
`java.util.concurrent.ForkJoinPool.common.parallelism=0`
The main thread actually blocks waiting for `f2.join()` is behind `f1.join()`.

(f) Other non-functional combinations

- `return f1.join() + f2.join();`
- `return f1.fork() + f2.fork();`
- `return f1.compute() + f2.fork();`
- `return f1.fork() + f2.join();`

A `fork()` must be followed by a `join()` to get the result back. None of the options that uses fork also join back the result. The only option that gives us the correct result is A. Note that it computes the Fibonacci number sequentially.

You can use this version to test the preformance

```
import java.util.concurrent.RecursiveTask;
import java.time.Instant;
import java.time.Duration;

class Fib extends RecursiveTask<Integer> {

    final int n;

    Fib(int n) {
        this.n = n;
    }

    private void waitOneSec() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { }
    }

    @Override
    protected Integer compute() {
        System.out.println(Thread.currentThread().getName() + " : " + n);
        waitOneSec();

        if (n <= 1) {
            return n;
        }

        Fib f1 = new Fib(n - 1);
        Fib f2 = new Fib(n - 2);

        // try different variants here...
    }

    public static void main(String[] args) {
        Instant start = Instant.now();
        System.out.println(new Fib(5).compute());
        Instant end = Instant.now();

        System.out.println(Duration.between(start, end).toMillis());
    }
}
```