**CS2030 Programming Methodology**
Semester 1 2019/2020

1 November 2019
Problem Set #9 Suggested Answers
**Fork/Join Framework**

1. Run the following program and observe which worker is running which task.

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

class A {
    static class Task extends RecursiveTask<Integer> {
        int count;
        Task(int count) {
            this.count = count;
        }

        public Integer compute() {
            System.out.println(Thread.currentThread().getName()
                    + " " + this.count);
            if (this.count == 4) {
                return this.count;
            }
            Task t = new Task(this.count + 1);
            t.fork();
            return t.join();
        }
    }

    public static void main(String[] args) {
        ForkJoinPool.commonPool().invoke(new Task(0));
    }
}
```

Suppose the program is invoked with a maximum of three additional workers. What can you observe about the behaviour of a worker when the task that it is running blocks at the call to `join`?

*Students should observe that there exists a worker running Task i that also picks up Task j ($j > i$). Since Task i blocks at* `join()`*, this means that the worker does not sit idling waiting at* `join()` *but put the blocking task aside and picks up (steals) another task to execute.*

2. Given below is the classic recursive method to obtain the $n^{th}$ term of the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ *without memoization*

```
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.*

*(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());
    }
}
```

3. 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.

`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.*

4. Many ways have been devised to multiply two large integers. One of these ways is attributed to Anatoly Karatsuba in 1960 and is described below using the example $1234 \times 567 = 699678$.

   Step 1. If necessary, pad the smaller number with leading zeros to make two numbers of the same length $L$, i.e. 1234 and 0567.

   Step 2. Divide the two numbers into equal left and right portions and label them $a, b, c, d$

   $$\begin{array}{c|c} a = 12 & b = 34 \\ \hline c = 05 & d = 67 \end{array}$$

   Step 3. Calculate $ac = 12 \times 5 = 60$

   Step 4. Calculate $bd = 34 \times 67 = 2278$

   Step 5. Calculate $(a + b)(c + d) = 46 \times 72 = 3312$

   Step 6. Calculate the result of step (5) - step (4) - step (3) $= 3312 - 2278 - 60 = 974$

   Step 7. Add the partial results with zero padding

   | | |
   |---|---|
   | 600000 | from step (3) by padding $L$ trailing zeroes |
   | 2278 | from step (4) with no additional padding of trailing zeroes |
   | 97400 | from step (6) by padding $L/2$ trailing zeroes |
   | 699678 | |

   Notice that multiplying two large numbers require three smaller multiplications which can be done independently in steps (3), (4) and (5).

   Your task is to define a `Task` class that extends `RecursiveTask<BigInteger>` and computes the multiplication in parallel. The following methods from the `BigInteger` class may be useful to you.

   - `public BigInteger(String val)`
     Translates the decimal `String` representation of a `BigInteger` into a `BigInteger`.
   - `public BigInteger add(BigInteger val)`
     Returns a BigInteger whose value is `this + val`.
   - `public BigInteger subtract(BigInteger val)`
     Returns a BigInteger whose value is `this - val`.
   - `public BigInteger multiply(BigInteger val)`
     Returns a BigInteger whose value is `this * val`. Use this when the numbers to be multiplied are of length less than 2.
   - `public BigInteger pow(int exponent)`
     Returns a BigInteger whose value is `this` raised to the power of `exponent`.
   - `public String toString()` Returns the String representation of this `BigInteger`.

   You may also use other methods from the Java API.

```
import java.math.BigInteger;
import java.lang.Math;
import java.util.Scanner;
import java.util.concurrent.RecursiveTask;

class Multiply extends RecursiveTask<BigInteger> {
    static int threshold = 2;
    private final BigInteger x;
    private final BigInteger y;

    Multiply(BigInteger x, BigInteger y) {
        this.x = x;
        this.y = y;
    }

    BigInteger shiftNDigit(BigInteger in, int len) {
        String str = String.format("%0"+len+"d", in);
        return new BigInteger(str.substring(0, len/2));
    }

    BigInteger lastNDigit(BigInteger in, int len) {
        String str = String.format("%0"+len+"d", in);
        return new BigInteger(str.substring(len/2));
    }

    BigInteger padNZeros(BigInteger in, int len) {
        String str = in.toString() + String.format("%0"+len+"d", 0);
        return new BigInteger(str);
    }

    protected BigInteger compute() {
        int len = Math.max(x.toString().length(), y.toString().length());

        if (len < threshold) {
            return x.multiply(y);
        } else {
            BigInteger a = shiftNDigit(x, len);
            BigInteger b = lastNDigit(x, len);
            BigInteger c = shiftNDigit(y, len);
            BigInteger d = lastNDigit(y, len);

            Multiply step3 = new Multiply(a, c);
            Multiply step4 = new Multiply(b, d);
            Multiply step5 = new Multiply(a.add(b), c.add(d));
            step3.fork();
            step4.fork();
            step5.fork();
            BigInteger abcd = step5.join();
            BigInteger bd = step4.join();
            BigInteger ac = step3.join();
            BigInteger result = abcd.subtract(ac).subtract(bd);

            return padNZeros(ac,(len+1)/2*2).add(bd).add(padNZeros(result,(len+1)/2));
        }
    }
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        BigInteger x = new BigInteger(sc.next());
        BigInteger y = new BigInteger(sc.next());
        Multiply.threshold = 2;
        System.out.println(new Multiply(x, y).compute());
    }
}
```