**CS2030 Programming Methodology**
Semester 1 2020/2021

4 November 2020
Problem Set #10 Suggested Guidance
**Streams**

1. Write a method `omega` with signature `LongStream omega(int n)` that takes in an `int n` and returns a `LongStream` containing the first $n$ omega numbers. We use `LongStream` in order to work with large integer values.

   The $i^{\text{th}}$ omega number is the number of distinct prime factors for the number $i$. The first 10 omega numbers are $0, 1, 1, 1, 1, 2, 1, 1, 1, 2$. The `isPrime` method is given below:

```
boolean isPrime(int n) {
    return IntStream
        .range(2, n)
        .noneMatch(x -> n%x == 0);
}
```

   **Answer:**

```
jshell> boolean isPrime(int n) {
   ...>     return IntStream
   ...>         .range(2, n)
   ...>         .noneMatch(x -> n%x == 0);
   ...> }
jshell> IntStream factors(int x) {
   ...>     return IntStream
   ...>         .rangeClosed(2, x)
   ...>         .filter(d -> x % d == 0);
   ...> }
jshell> IntStream primeFactorsOf(int x) {
   ...>     return factors(x)
   ...>         .filter(d -> isPrime(d));
   ...> }
jshell> LongStream omega(int n) {
   ...>     return IntStream
   ...>         .range(1, n + 1)
   ...>         .mapToLong(x -> primeFactorsOf(x).count());
   ...> }
jshell> omega(10).toArray()
$5 ==> long[10] { 0, 1, 1, 1, 1, 2, 1, 1, 1, 2 }
jshell> /exit
```

2. Write a method that returns the first $n$ Fibonacci numbers as a `Stream<Integer>`.

   For instance, the first 10 Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

   *Hint*: Write an additional `Pair` class that keeps two items around in the stream

   **Answer:**

   *We use the* `BigInteger` *class to avoid overflow.*

```
jshell> class Pair<T,U> {
   ...>     T first;
   ...>     U second;
   ...>     Pair(T first, U second) {
   ...>         this.first = first;
   ...>         this.second = second;
   ...>     }
   ...> }
jshell> Stream<BigInteger> fibonacci(int n) {
   ...>     return Stream.iterate(
   ...>                 new Pair<>(BigInteger.ZERO, BigInteger.ONE),
   ...>                 pr -> new Pair<>(pr.second, pr.first.add(pr.second)))
   ...>             .map(pr -> pr.second).limit(n);
   ...> }
jshell> fibonacci(5).toArray(BigInteger[]::new)
$3 ==> BigInteger[5] { 1, 1, 2, 3, 5 }
jshell> fibonacci(48).toArray(BigInteger[]::new)
$4 ==> BigInteger[48] { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610
, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418
, 317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14930352,
 24157817, 39088169, 63245986, 102334155, 165580141, 267914296, 433494437, 70140
8733, 1134903170, 1836311903, 2971215073, 4807526976 }
jshell> /exit
```

3. In this question, we shall attempt to parallelize the generation of the Fibonacci sequence. As an example, supose we are given the first $k = 4$ values of the sequence $f_1$ to $f_4$, i.e.

$$1, 1, 2, 3$$

   To generate the next $k - 1$ values, we observe the following:

$$\begin{array}{rclcrcl} f_5 & = & f_3 + f_4 & = & 1 \cdot f_3 + 1 \cdot f_4 & = & f_1 \cdot f_3 + f_2 \cdot f_4 \\ f_6 & = & f_4 + f_5 & = & 1 \cdot f_3 + 2 \cdot f_4 & = & f_2 \cdot f_3 + f_3 \cdot f_4 \\ f_7 & = & f_5 + f_6 & = & 2 \cdot f_3 + 3 \cdot f_4 & = & f_3 \cdot f_3 + f_4 \cdot f_4 \end{array}$$

   Notice that generating each of the terms $f_5$ to $f_7$ only depends on the terms of the given sequence. This actually means that generating the terms can now be done in parallel! In addition, repeated application of the above results in an exponential growth of the Fibonacci sequence.

   You are now given the following program fragment:

```
BigInteger findFibTerm(int n) {
    List<BigInteger> fibList = new ArrayList<>();
    fibList.add(BigInteger.ONE);
    fibList.add(BigInteger.ONE);

    while (fibList.size() < n) {
        fibList = generateFib(fibList);
    }
    return fibList.get(n-1);
}
```

(a) Using Java parallel streams, complete the `generateFib` method such that each method call takes in an initial sequence of $k$ terms and fills it with an additional $k-1$ terms. The `findFibTerm` method calls `generateFib` repeatedly until the $n^{th}$ term is generated and returned.

(b) Using the `Instant` and `Duration` classes, determine the time it takes the Fibonacci sequence of $n = 50000$. Compare the times for sequential and parallel generations.

**Answer:**

```
jshell> import java.time.Instant;
jshell> import java.time.Duration;
jshell> List<BigInteger> generateFib(List<BigInteger> fibs) {
   ...>     List<BigInteger> list = new ArrayList<>(fibs);
   ...>     int k = list.size();
   ...>     list.addAll(Stream
   ...>             .iterate(0, i -> i < k - 1, i -> i + 1)
   ...>             .parallel()
   ...>             .map(i ->
   ...>                 fibs.get(k-2).multiply(fibs.get(i)).add(
   ...>                     fibs.get(k-1).multiply(fibs.get(i+1))))
   ...>             .collect(Collectors.toList()));
   ...>     return list;
   ...> }
jshell> BigInteger findFibTerm(int n) {
   ...>     List<BigInteger> fibList = new ArrayList<>();
   ...>     fibList.add(BigInteger.ONE);
   ...>     fibList.add(BigInteger.ONE);
   ...>     Instant start = Instant.now();
   ...>     while (fibList.size() < n) {
   ...>         fibList = generateFib(fibList);
   ...>     }
   ...>     Instant stop = Instant.now();
   ...>     System.out.println(Duration.between(start,stop).toMillis() + "ms");
   ...>     return fibList.get(n-1);
   ...> }
```

```
jshell> findFibTerm(50000)
7355ms
$5 ==> 10777734893072974780279038855119480829625106769411579784902309210032744735364652304984884440204760298493194332832740549533075398173304830674148353871755545405198446200873464249380723258213016701908119882516186149595860854099373751065304487446378299685138932566366816331317320459189318988631355996126556155463897643055715140539792260124322730482900071690886378620675517700832269328087849866274058836537593758274508704744192976808834961311297128859285176754848415103252385146533492152825245908446669841110158718288730189450631134151562379824589360041756899572012659615762899095935573140213802968576539055708953584711268396232119536018973338811423814051526459374097 ... 67763105426068549719178378668819786805212576177264040949511215576188269822366838153968218686762926290755720567510373245164756842944423699212491240487464281580686750806724451064512444192234336251813764582803376461209571993619736455646214921063358870308182304266593049366953768037220397037490781969011126652402029761830536425237355312 5
jshell> /exit
```