1. Study the given class `A` below, which uses the methods `incr` and `decr` to imitate slow computations.

```
1  class A {
2    private final int x;
3
4    A() {
5      this(0);
6    }
7
8    private A(int x) {
9      this.x = x;
10   }
11
12   void sleep() {
13     System.out.println(Thread.currentThread().getName() + " " + x);
14     try {
15       Thread.sleep(1000);
16     } catch (InterruptedException e) {
17       System.out.println("interrupted");
18     }
19   }
20
21   A incr() {
22     sleep();
23     return new A(this.x + 1);
24   }
25
26   A decr() {
27     sleep();
28     if (x < 0) {
29       throw new IllegalStateException();
30     }
31     return new A(this.x - 1);
32   }
33
34   @Override
35   public String toString() {
36     return "" + x;
37   }
38 }
```

   (a) Suppose we have a method

```
1  static A foo(A a) {
2    return a.incr().decr();
3  }
```

   Convert the method `foo` above to a method that returns `CompletableFuture` so that the body of the method is executed asynchronously. Try different variations by using:

   i. `supplyAsync` only

   ii. `supplyAsync` and `thenApply`

   iii. `supplyAsync` and `thenApplyAsync`

Demonstrate how you would retrieve the result of the computation.

See also: `thenRun`, `thenAccept`, `runAsync`

(b) Suppose now we have another method

```
1  static A bar(A a) {
2    return a.incr();
3  }
```

which we would like to invoke using `bar(foo(new A()))`. Convert the computation within `bar` to run asynchronously as well. `bar` should now return a `CompletableFuture`. In addition, show the equivalent of calling `bar(foo(new A()))` in an asynchronous fashion, using the method `thenCompose`.

See also: `thenCombine`

(c) Suppose now we have yet another method

```
1  static A baz(A a, int x) {
2    if (x == 0) {
3      return new A();
4    } else {
5      return a.incr().decr();
6    }
7  }
```

Convert the computation within `baz` in the `else` clause to run asynchronously. `baz` should now return a `CompletableFuture`. You may find the method `completedFuture` useful.

(d) Let's now call `foo`, `bar`, and `baz` asynchronously. We would like to output the string `"done!"` when ***all*** three method calls are complete. Show how you can use the `allOf()` method to achieve this behaviour.

See also: `anyOf`, `runAfterBoth`, `runAfterEither`

(e) Calling `new A().decr()` would cause an exception to be thrown, even when it is done asynchronously. Show how you would use the `handle()` method to gracefully handle exceptions thrown (*e.g.*, *such as printing them out*) within a chain of `CompletableFuture` calls.

See also: `whenComplete`, `exceptionally`

2. Modify the following sequences of code such that `f`, `g`, `h`, and `i` are now invoked asynchronously, via `CompletableFuture`. Assume that `a` has been initialized as

```
A a = new A();
```

(a) Problem #A

```
B b = f(a);
C c = g(b);
D d = h(c);
```

(b) Problem #B

```
B b = f(a);
C c = g(b);
h(c); // no return value
```

(c) Problem #C

```
B b = f(a);
C c = g(b);
D d = h(b);
E e = i(c, d);
```

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

```
class B {
  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?

4. Given below is the classic recursive method to obtain the nth term of the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . *without memoization*

```
1  static int fib(int n) {
2    if (n <= 1) {
3      return n;
4    } else {
5     return fib(n - 1) + fib(n - 2);
6    }
7  }
```

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

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