Learning objective: explore executor implementations, how to submit tasks, how to divide work into multiple tasks and how fork-join pools help implementing divide-and-conquer algorithms.

**Exercise 1: direct executor**
Implement an executor (void `execute(Runnable r)` that launches directly the execution of each runnable synchronously.

**Exercise 2: one thread per task**
Implement an executor that launches the execution of each runnable asynchronously in a new thread.

**Exercise 3: serialized executions**
Implement an executor that launches the execution of each runnable in sequence in a thread. Task submissions should be asynchronous.

**Exercise 4: seizing a thread pool**
We consider a large number of tasks that each consists of a compute time $C$ and a wait time $W$. We assume the overlap of the waiting times is as low as possible. If the number of threads in the pool is too big, the threads end up competing for scarce cores and memory resources, wasting their time performing context switching. What is the minimum size for a thread pool to fully use all $n$ cores?

**Exercise 5: Fibonacci on an executor service**
Write an implementation that computes a given number of the Fibonacci sequence using an executor service. Explain which executor service must be chosen.

**Exercise 6: sum of square values on an executor service**
Implement an algorithm that sums the square of all values in an array using an executor service.