The learning outcome of this practical session is to confirm skills related to streams by implementing common scheduling strategies for the problem $P||C_{\text{max}}$.

The first section is essential.

1 Scheduling Algorithms Implementation

Implement a function that uses a stream to generate an array of $n$ task execution times. The generation is assumed to be random uniformly between 0 and 1.

```java
double[] genCost(int n) {
    return DoubleStream.generate(Math::random)
        .limit(n)
        .toArray();
}
```

The maximum completion time of any schedule is bounded by 2 values: the sum of the execution times divided by the number of processors, and the maximum execution time of any task. Implement a function that computes these two lower bounds from the execution times and the number of processors (using a stream).

```java
double LB(double[] cost, int m) {
    return Math.max(Arrays.stream(cost).sum() / m,
                    Arrays.stream(cost).max().orElse(0));
}
```

Implement LST using streams: one stream can be used to initialize an array of availability times, one for each processor, and another can be used to iterate over all tasks. The function takes as input execution times and a number of processors and returns the maximum completion time of the schedule.

Adapt the code for LPT and SPT.

The solution illustrates 3 different ways to use streams to generate the array of availability times and 3 others to iterate over all tasks.
Arrays.stream(cost)
    .forEach(t -> {
        Arrays.sort(C);
        C[0] += t;
    });
    return Arrays.stream(C).max().orElse(0);
}
static double LPT(double[] cost, int m) {
    var C = Arrays.stream(cost)
        .boxed()
        .sorted(Comparator.reverseOrder())
        .reduce(DoubleStream.iterate(0, x -> 0)
            .limit(m).toArray(),
            (acc, t) -> {
                Arrays.sort(acc); acc[0] += t; return acc;
            }, (acc1, acc2) -> null);
    return Arrays.stream(C).max().orElse(0);
}
static double SPT(double[] cost, int m) {
    var C = Arrays.stream(cost)
        .sorted()
        .collect(() -> IntStream.range(0, m)
            .mapToDouble(i -> 0)
            .toArray(),
            (acc, t) -> { Arrays.sort(acc); acc[0] += t; },
            (acc1, acc2) -> {});
    return Arrays.stream(C).max().orElse(0);
}

2 Performance Evaluation

In the following, we will establish for which instance sizes (i.e., number of tasks) the previously imple-
mented strategies reach a given level of performance. It is required to produce a structured code with
simple and generic functions that rely on streams as much as possible. We can observe that strategies
are first optimal for small instances, then their performance differ from the lower bound for medium
sizes. For large sizes, the difference decreases. The objective consists in identifying for which minimum
number of tasks (excluding small instances) the maximum completion time is at most at 1% of the lower
bound for each strategy. For each instance size, it is necessary to compute the average ratio between
the maximum completion time and its lower bound on 30 distinct instances. We assume the number of
processors to be m = 10.

double ratio(double[] cost, int m, BiFunction<double[], Integer, Double> sched) {
    return sched.apply(cost, m) / LB(cost, m);
}
double ratioAvg(int n, int m, BiFunction<double[], Integer, Double> sched) {
    return IntStream.range(0, 30)
        .mapToDouble(i -> ratio(genCost(n), m, sched))
        .average()
3 Parallelization

How to best parallelize the previous performance evaluation?

It is best to start considering the parallelization of the most external processing structures. Processing LST, LPT and SPT each on a separate thread is not sufficient for 2 reasons: there are only 3 strategies, which will fully used all the available cores as the grain is too coarse; the evaluation of each strategy takes a different duration (the processing is heterogeneous). We could parallelize the most external stream, but it requires synchronization due to `findFirst`, which will be inefficient. Finally, we can focus on the function `ratioAvg`, which does not require synchronization (except for the call to `Math.random` that can be replaced with a thread-local equivalent). However, at this stage, the grain may be too fine and the parallelization may not provide any gain due to costly synchronizations.