Assessed Coursework

MA407

Algorithms and Computation

due: Monday, January 31, 2022 at 14:00

Instructions

Tasks 1–4 on the following pages are assessed coursework, constituting **25%** of the final mark for this course.

It is due on **Monday, January 31, 2022 at 14:00** (week 3 of Michaelmas Term 2022). The tasks ask you, step by step, to complete a Java program consisting of five files in plain text format. One of these files, `Input.java`, is given to you complete. Another file, `Schedule.java`, is given to you as well, but it is your task to complete one method in this file. It is also your task to write the remaining three files, `Course.java`, `Register.java` and `Slot.java`. Make sure that the files you submit compile without loading any external Java packages.

Submit the four files `Schedule.java`, `Course.java`, `Register.java` and `Slot.java` on the Moodle page for MA407 under the submission link in the Assessed Coursework section. The files `Input.java` and `Schedule.java` are also on the Moodle page in this section.

Please submit only a single final version of your files. Otherwise you risk that the wrong submission will be marked. If you submit more than one version, we will mark the last one submitted.

The deadline is sharp. Late work carries an automatic penalty of 5 deducted marks (out of 100) for every 24 hours that the coursework is late.

Submission of answers to this coursework is mandatory. Without any submission your mark for this course is incomplete, and you may not be eligible for the award of a degree.

The work should be yours only. **Plagiarism** is considered an assessment offence at the LSE and is, as an instance of academic misconduct, taken very seriously. In all cases of suspected plagiarism, the Department will act according to the School's Regulations on Assessment Offences—Plagiarism. So only submit work that is completely your own (or uses code fragments from programs discussed in MA407). This also means that for developing your solutions you are not allowed to collaborate with other students or to ask other persons for help.

The contents of your work **must** remain **anonymous**, so do not write your name or student number in any of the program files. Instead, identify your work with your **candidate number**. Please insert your candidate number as a comment in the first line of your files. You can find your candidate number on 'LSE for You'.

If you have any question about this coursework, **do not post your question on Moodle** (it will be deleted), but instead, please email Julia Böttcher (j.boettcher@lse.ac.uk) and Tuğkan Batu (t.batu@lse.ac.uk). We will post responses to your questions on Moodle.

Coursework MA407 – Seminar scheduling

This coursework asks you to write a Java program for solving the following lecture scheduling problem. We are given a set of rooms, and for each room a set of one hour slots when this room is available for teaching (during all other times the room is already booked for other purposes). We are also given a set of lecture courses, each of which requires a certain number of seminars to be scheduled. Each seminar lasts exactly one hour. Moreover, we are given a set of students S and for each student $s \in S$ their course choices $C(s)$. We say that two courses c and c' are *conflicting* if there is a student s, with $c \in C(s)$ and $c' \in C(s)$. We say that two one hour slots t, t' are *overlapping* if t does not start later than t' starts but ends after t' starts, or vice versa. (In particular, t and t' are not overlapping if t ends when t' starts.) Now we want to find an assignment of the seminars of all courses to the one hour slots of the rooms with the following properties:

- Only one seminar takes place in any given room per slot.
- If two courses c and c' are conflicting, no seminar of c is assigned to a slot overlapping any slot assigned to a seminar of c'.

In the following we call these rules the *scheduling rules* and we call each schedule obeying these rules a *valid* schedule. Observe though that two seminars of the same course are allowed to overlap (they are taught by different teachers).

The goal of this project is to implement a greedy algorithm and some corresponding data structures for solving this problem. You should start off from the two files `Schedule.java` and `Input.java`, which are provided on the moodle page for MA407. `Schedule` is the main class of this program. Its `main` method makes use of the static method `read(String file)`, which is the only method of the class `Input`. This method reads in the input, consisting of the information about rooms, slots, courses and students, from a file whose name is given as an argument to the program. (We did not talk about how this is done in Java, so you do not need to understand the code in this file.) The format of the input is explained at the end of the file `Input.java`. On the moodle page for MA407 you also find several example input files: `Data1.txt`, `Data2.txt`, `Data3.txt`, `DataEmpty.txt`. The method `read` then stores the information using three data structures that you should implement in the files `Slot.java`, `Course.java` and `Register.java`. The following three tasks guide you through this process.

Firstly, we want to store information about any slot in any room available for scheduling. Your first task is to implement a corresponding data type. For simplicity, we are restricting our timetabling problem to a single day; so each one hour slot is determined by a start time consisting of hours and minutes.

Task 1. ***20p***

Write a class `Slot` *in a file* `Slot.java` *implementing the following API.*

(*a*) *Constructor:* `Slot(int room, String time)`
Construct an object storing the given `room` *number and start* `time` *of this one hour slot. Here,* `time` *may be given as either* `h:mm` *or* `hh:mm` *in 24-hour format (for example,* `"13:00"` *for one o'clock in the afternoon and* `"6:00"` *or* `"06:00"` *for six o'clock in the morning).*

(*b*) *Instance method:* `boolean overlaps(Slot s)`
Return `true` *if* `s` *and* `this` *are overlapping slots and* `false` *otherwise.*

(*c*) *Instance method:* `String toString()`
Return a `String` *representation of this object, giving information about the room number and the start time and end time of this slot in* `hh:mm` *format. This string should have the following form:*
`Room 41 09:06-10:06`

Hint: You will need to analyse the second argument of the constructor, the string `time`*, suitably. For this purpose, you might find the instance methods* `indexOf(char)` *and* `substring(int,int)` *of the class* `String` *useful. Moreover, it may be useful not to store the start time as a* `String` *in this class, but use instance variables of different types instead.*

But we do not only want to store one slot, but a whole list of them. For this purpose we want to use a dynamic data structure. Since we also want to store a list of courses later, we will make this data structure generic so that we can reuse it. Moreover, we want to be able to *iterate* through this data structure: that is, to go through all its items visiting each item exactly once.

Task 2. ***30p***

Write a generic class `Register<Type>` *in a file* `Register.java` *as a linked data structure for storing an arbitrary number of values of type* `Type`*. This class should implement the following API:*

(*a*) *Instance method:* `void add(Type t)`
Add a new item with value `t` *to the data structure. In a comment, explain where in the data structure you add and how this influences the result of your greedy algorithm (see Task 4).*

(*b*) *Instance method:* `void startIteration()`
Prepares the data structure for starting a new iteration through the values in the data structure.

(*c*) *Instance method:* `Type next()`
Return the next value stored in the data structure. You may assume that `startIteration()` *was called before. Then, the first time* `next()` *is called it should return the first value stored, the second time the second value stored, and so on. If the last value was reached, return* `null`*.*

(*d*) *Instance method:* `boolean hasNext()`
Check if the current iteration reached the end. You may assume that `startIteration()` *was called before. If* `next()` *was not called since the last call to* `startIteration()` *then return* `true` *if the list is non-empty. Otherwise, return* `true` *if the item whose value was last returned by* `next()` *is not the ultimate in the list. Else return* `false`*.*

(*e*) *Instance method:* `void deleteCurrent()`
Delete the current item in this iteration. That is, delete the item storing the value that was last returned by `next()`*. (You may assume that there was such a call to* `next()` *and that no call to* `startIteration()` *has taken place since then.)*

Hint: You will need some instance variables to store the current position in the iteration.

For each lecture course we want to store the name of the course, the course ID, the number of seminars, a list of conflicting courses, and a list of slots assigned to this course for seminars.

Task 3. ***30p***

Write a class `Course` *in a file* `Course.java` *implementing the following API.*

(*a*) *Constructor:* `Course(int id,String name,int seminars)`
Construct an object storing the given `id`*,* `name` *and the number of seminars* `seminars` *that this course needs.*

(*b*) *Instance method:* `int getID()`
Return the course id.

(*c*) *Instance method:* `void addConflict(Course c)`
Add `c` *to the list of conflicting courses for this course.*

(*d*) *Instance method:* `void addSlot(Slot s)`
Add `s` *to the list of slots assigned to this course.*

(*e*) *Instance method:* `boolean slotNeeded()`
Return `true` *if more slots need to be assigned to this course (because there are fewer slots assigned so far than the number of seminars),* `false` *otherwise.*

(*f*) *Instance method:* `boolean slotPossible(Slot s)`
Return false if assigning `s` *to this course violates the timetabling rules when we consider all slots that have so far been assigned to any course in the list of conflicting courses,* `true` *otherwise.*

(*g*) *Instance method:* `String toString()`
Return a `String` *representation of this object, providing the id, name and assigned slots for this course. This string should have the following form:*

```
Course 2 "Algorithms"
Seminars:  Room 1 11:30-12:30 Room 5 11:30-12:30 null null
```

Here, each `null` *indicates that there is a seminar to which no slot has been assigned yet.*

You need to decide, which data structures to use for storing the list of conflicts and the list of assigned slots. In each of these two cases, document and justify your choice of data structure together with the corresponding instance variables.

With these three data structures at hand the method `Input.read()` can store all the information it gets from the input, using a variable `slots` of type `Register<Slot>` for storing the list of slots and a variable `courses` of type `Register<Courses>` for storing the list of courses (together with their conflicts). It then returns an object of type `Schedule`, which references this course list and slot list in two instance variables, and is assigned to the variable `schedule` in the `main` method of `Schedule`.

It remains to use this data for creating a schedule using a greedy algorithm. For this purpose you should complete the method `greedySchedule()` of `Schedule`.

Task 4. ***20p***

In the class `Schedule`*, complete the instance method* `greedySchedule()` *to implement a greedy algorithm for assigning slots to courses, such that each course receives as many slots as it needs seminars (if the algorithm succeeds).*
More precisely, create a schedule in rounds, assigning one slot to one course greedily per round without violating the timetabling rules. Use the method `addSlot` *of the class* `Course` *for this purpose. Your algorithm does not need to succeed (even if a valid schedule exists): If no further slot can be assigned to some course* `c` *then print the string*
`"GREEDY FAILED: could not assign further slot to course " + c.getID()`
and stop the greedy algorithm.
In your algorithm the order in which slots are assigned to lectures plays an important role. Choose an order that makes it more likely that your algorithm succeeds. Explain which order you chose and justify why in a comment.
Hint: Observe that slots were added to the list of slots of type `Register<Slot>` *in the following order. First, all slots for the first room were added (in the order given in the input file), then all slots for the second room, and so on.*

After calling `schedule.greedySchedule()` the `main` method in `Schedule` uses `schedule.toString()` for printing the timetabling information. The method `toString()` of `Schedule` in turn iterates through the list of courses and uses the method `toString()` of `Course` for each course.

Once you solved these tasks the output of your program should look similar to the following.

```
java Schedule Data.txt
Course 1 "A brief history of times"
Seminars: Room 1 09:00-10:00 Room 2 09:00-10:00 Room 2 15:00-16:00

Course 2 "Algorithms"
Seminars: Room 1 11:30-12:30 Room 2 11:30-12:30

Course 3 "Principles of scheduling"
Seminars: Room 1 12:30-13:30 Room 2 12:30-13:30 Room 3 09:30-10:30

Course 4 "Conflict of Ideas"
Seminars: Room 1 14:00-15:00

Course 5 "Learning Theory"
Seminars: Room 1 15:00-16:00

Course 6 "Abstract Algebra"
Seminars: Room 1 16:30-17:30
```

Remember to explain all your code well with *comments* and to *indent* properly. Do not use any Java libraries in your files (even though `Input.java` uses some).

END of assessed coursework, 2021/22 Good luck!