Assignment overview

Meet Joyce Peterson, the beleaguered administrative assistant at the Greater Vancouver Community Centre. Joyce is responsible for scheduling the groups who reserve the GVCC Great Hall for their meetings.

Sometimes there are more requests than the room can accommodate. Meeting requests may overlap, but *only one group can use the room at a time*.

Joyce wants to please as many groups as she can. On top of that, she gets paid a small bonus for every meeting she can schedule—no matter who or when or how long it is—so she is motivated to select *as many meetings as possible every day*.

We will make Joyce's life better with *GREEDY ALGORITHMS*.

Program overview

Your program will:

1. Read a data file containing a list of meeting requests for one day
2. Use a greedy algorithm to select a set of *scheduled meetings* for the day
3. Display the count, and the names of the groups that were selected
4. Repeat steps 2 & 3 for *two more* greedy algorithms (so we can compare their effectiveness in finding the best answer)

Three greedy algorithms

You must implement *three* greedy algorithms. Do not be alarmed! The algorithms are so similar that you can essentially—and maybe even literally—copy/paste most of the code. All three use the pattern, as follows:

- Rank (i.e. sort) the list of meeting requests in order by "best" (details below)
- Create/initialize an empty list of chosen meetings
- For every meeting request in the sorted list
    - Determine whether this request overlaps with any chosen meetings
    - If there are no overlaps, add this meeting to the chosen meetings
    - Otherwise, continue to the next meeting request

The differences are that the three algorithms will use different definitions of "best":

1. Requests ranked in order by start time (earliest start time is chosen first).
2. Requests ranked in order by length (shortest meeting is chosen first).
3. Requests ranked in order by end time (earliest end time is chosen first).

Program input

Each data file contains all of the meeting requests for one day. The format is two lines per meeting request. The first line contains a string: the Name of the group making the request. The second line contains the Start Time and End Time of the meeting request: two integers separated by white space.

You may assume that all data files conform to the above specification, i.e. no need for error-checking. All meeting End Times will be greater than their corresponding Start Times (i.e. the length will never be zero or negative).

Output specifications

For one data file, your program should display:

- The number of meetings scheduled with Greedy Algorithm #1 (rank by Start)
- The names of the meetings scheduled with Greedy Algorithm #1 (rank by Start)
- The number of meetings scheduled with Greedy Algorithm #2 (rank by Length)
- The names of the meetings scheduled with Greedy Algorithm #2 (rank by Length)
- The number of meetings scheduled with Greedy Algorithm #3 (rank by End)
- The names of the meetings scheduled with Greedy Algorithm #3 (rank by End)

Your program only needs to run for one data file, but it's fine if you set it up to run all of them.

Data files and expected results

There are four data files:

- **data1.txt**: This file has no overlapping requests. All three algorithms should report five meetings scheduled.
- **data2.txt**: This file is a counterexample proving that both the "start time" and "length" algorithms are *not optimal*. Three meetings are possible, but both of these algorithms select fewer than that.
- **data3.txt**: This file is a counterexample for "start time", but the other two algorithms will find the maximum of four scheduled meetings.
- **data4.txt**: This file has more meeting requests than any human—not even Joyce—could ever hope to analyze by hand in every day.

The following are the expected *number of chosen meetings* that will be scheduled by all three algorithms for all four data files. The names of the chosen meetings are not shown here; with data4.txt there could be more than one valid set of 37 meetings.

File	Rank by Start	Rank by Length	Rank by End
data1.txt	5 meetings*	5 meetings*	5 meetings*
data2.txt	1 meeting	2 meetings	3 meetings*
data3.txt	1 meeting	4 meetings*	4 meetings*
data4.txt	29 meetings	31 meetings	37 meetings*

* Best possible answer for this data file.

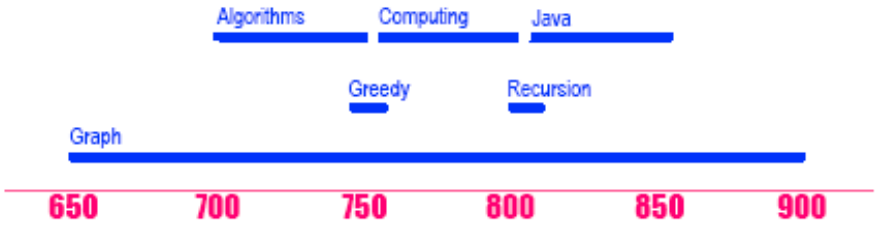
Understanding the data

It is helpful to visualize meeting requests as line segments with the corresponding start/end values. *This section is just for your own understanding, especially while you are debugging. Your program does not need to output anything like this.*

For example, consider the file `data2.txt`:

```
Algorithms Club
700 750
Computing Club
760 810
Java Club
820 870
Greedy Club
745 765
Recursion Club
805 825
Graph Club
650 900
```

Visualized as line segments, this list of requests would look like this:



Overlapping meetings:

Two meetings overlap if they share any non-zero-length segment of the timeline. Despite the crudeness of this sketch, you can see that several of these meetings overlap and therefore cannot be simultaneously chosen.

> Example: If an algorithm chooses Greedy Club first ("shortest length"), then it cannot subsequently choose Algorithms Club OR Computing Club OR Graph Club.
>
> Example: If an algorithm chooses Graph Club first, then it will be prevented from choosing *any* other meeting.

If a meeting starts at the exact same time as another one finishes, they *do not* overlap. For example (not shown on the above diagram) Meeting1 (500-575) and Meeting2 (575-650) *do not* overlap.

Meeting class

I recommend that you define a class to represent a meeting and/or request. This is not a requirement, just a suggestion. The encapsulation makes the entire program much easier than managing multiple data structures (such as using parallel arrays for the start and end times). The fact that Java passes objects by reference also contributes to this advantage.

I've provided a bare-bones class for this called "Meeting" (see `Meeting.java`) that you may use (but you do not have to). This class has everything that I needed for my code, but you are free to modify it in any way that you need.

Even if you don't use my Meeting class, you might find the logic in this method useful:

```
/*
 * Check whether this meeting overlaps with another one.
```

```
 */
public boolean overlapsWith(Meeting other) {
    return !((start >= other.getEnd()) || (other.getStart() >= end));
}
```

Sorting

You will need to sort a list of meetings three times, different ways (one for each greedy algorithm). I don't care how you do this. *Your highest design priority for this should be "easy for yourself to code".* Our data sets are so small that $O(N^2)$ sorting algorithms are fine.

If you are adept with using Comparator/Comparable in Java, I think you can probably accomplish this elegantly using those interfaces. Full disclosure: I am *not* adept at this in Java, so I did *not* do this. If you are not adept with these interfaces, then you don't have to do it, either. (This part of) my code is *not* elegant. I just wrote three different (bubble) sort algorithms in my main program.

How, what, and when to submit:

Please submit the following to the dropbox on Learning Hub:

- Java file containing your Main class
- Any other Java file(s) containing other class definitions

Please do not zip or compress your files (tempting though it may be).

This lab is worth 20 points. As with other labs, 4 points are reserved for coding style as per the guidelines set out in Lab 2.

It is due at midnight next Sunday, Nov 21.

Bonus entertainment

This is not a required part of the assignment.

Here is another greedy strategy for this problem:

- Rank the requests in ascending order by the *number of conflicts (overlaps)* that it has with all other requests.
- Proceed as with the other greedy strategies.

The rationale behind this idea is: If a meeting has fewer overlaps, it is less likely to cause problems in scheduling additional meetings. So choose it sooner.

Do you think this will yield an optimal algorithm for the scheduling problem? In practice, it does tend to perform better on many data sets. But it is not optimal.

Results on our data files:

data1.txt: No overlapping requests. Any greedy algorithm will schedule all 5 meetings.

data2.txt: Best answer is 3 meetings. This algorithm will schedule either 2 or 3 meetings, depending on what order the "ties" end up after you sort the list.

data3.txt: The algorithm will find the optimal 4-meeting answer, because the one big, problematic meeting has many overlaps and will get sorted/considered last.

data4.txt: I have no idea. I have not written any code for this part (yet?), and I have internally conflicting ideas about how many overlapping requests might appear in randomly-generated data with such a large quantity of items.