CS1112 Fall 2022 Project 6 due Monday 12/5 at 11pm

You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Objectives

Completing this project will solidify your understanding of object-oriented programming (problem 1) and recursion (problem 2). In problem 1 you will also get practice on developing and testing code *incrementally*—one class (or even one method) at a time.

1 Event Scheduling

Scheduling events under constraints—limited availability, conflicting occurrences—is a necessity: we busy students and professors need to pack many classes and meetings into a day; a convention venue wishes to schedule events in order to maximize profits. In this part of the project you will write code for event scheduling, given the needs of the events and constraints on the scheduling time window. Below is a graphic of an example schedule. Every event has an integer ID while some additionally have names (e.g., course names). Time is represented using non-negative integer values. The colored boxes indicate the actual scheduled time of the events while the white boxes indicate, according to the event data, the time intervals during which the events could be scheduled. The importance of the events is color coded from cyan—most important—to magenta—least important.

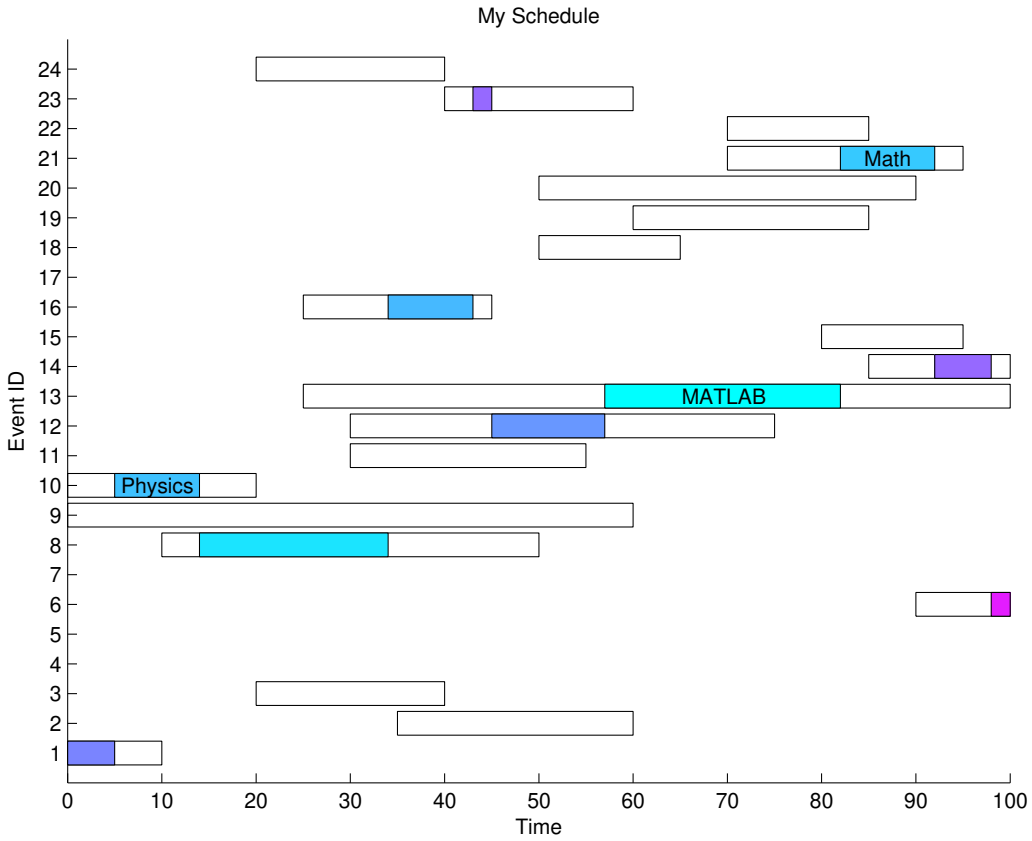

Figure 1: Example schedule

We provide the *design* of the classes that you will *implement* in this project. Read carefully about the reasoning behind the design below. You will see *how objects of four classes interact* in this scheduling project. You will see that one of the classes that we will use is `Interval`, a class that we have used in other settings in lecture and lab examples. As we have discussed before, a class that is designed well can be used in different projects and situations, including being extended for specific purposes.

Another focus of this project is for you to practice *testing* individual methods and classes, one at a time. We suggest tests throughout the project description, but *you need to generate more tests* beyond our suggestions in order to make sure that the methods you implement are correct. Do not assume that just passing the one test case that we have suggested implies that your method is correct. As you have seen in the exercises, *each method/function should be tested with multiple, distinct test cases that are representative of different input scenarios.*

1.0 Object-Oriented Design

In the above description of the project, these *nouns* keep coming up: event, schedule, and time interval. These are the objects that we will use. An *Interval* has a left and right end point. An *Event* has an ID, a duration, and a time interval during which it can take place. There're special events that are named; we'll call such a named event a `Course` in this project. A *Schedule* contains a set of events that are set at specific times within a time window.

Our design involves four classes: `Interval`, `Event`, `Schedule`, and `Course` (subclass of `Event`). We then write a function `createSchedule` to read event data from a file and then instantiate objects of the different classes in order to create a feasible schedule. (To "instantiate" an object is to call the constructor in order to create an object.) Don't be intimidated by the number of classes! Class `Interval` is given, and partial code is given in all the other classes. Below is a summary of the classes, showing what properties and methods are available in each class. Properties and methods with `private` access are shown using an open circle, ∘; those with `private` *set* access[1] are shown using a circle with an X, ⊗; those with `public` access are shown using a filled circle, •. The skeleton code given in the zip file `P6_Files.zip` gives further details on each item. Download the zip *and extract the files from the zip*—do not work from the zip without extracting the actual m-files and data files first. The rest of this document helps you develop the classes incrementally, one at a time.

Interval
Properties:
• left
• right
Methods:
• Interval
• getWidth
• scale
• shift
• isIn
• add
• overlap
• disp

Event
Properties:
∘ id
• duration
• available
• importance
• scheduledTime
Methods:
• Event
• earliestTime
• setScheduledTime
• unschedule
• getId
• draw

Course is an **Event**
Properties:
∘ courseName
Methods:
• Course
• getCourseName
• draw

Schedule
Properties:
⊗ sname
⊗ window
⊗ eventArray
Methods:
• Schedule
• addEvent
• scheduleEvents
• draw

The above summary diagrams will be handy when you later write code that involves interactions among the objects of the different classes.

So which class should you work on first? Answer: the most independent class, the one that doesn't depend on other classes. We will start with class `Interval`. As you complete the classes, do not change the names of the properties, methods and parameters. Do not change the property and method attributes (e.g., `private`). You should not need to add any extra methods in the classes.

[1] MATLAB allows us to assign a different attribute (`private`, `protected`, or `public`) to a property for "set access"—ability to assign a value to a property—and for "get access"—ability to retrieve the value from a property—separately. See the syntax used for the `properties` block of class `Schedule`.

1.1 Class `Interval`

Use the `Interval.m` file extracted from the Project 6 zip. (We used multiple versions of the class definition during lecture; please be sure to use the version released for Project 6.)

Read the class definition and then experiment with it! Assuming that all the skeleton files are in your Current Directory, type the following statements in the Command Window:

```
i1= Interval(3,9);  % Instantiate an Interval with endpoints 3 and 9
a= i1.left          % Should be 3.  The properties have the attribute "public" so
                    %  it is possible to access the property left directly.
o= i1.overlap( Interval(5,15) )
                    % o references an Interval with endpoints 5 and 9.
```

We started a file `testScript.m` to help you test your classes as you develop them. Read and run `testScript` now; you can see that it contains the same kind of code as suggested above for "exercising" the `Interval` class. Since class `Interval` is given, you are not actually testing the methods but instead you are practicing how to access the properties and methods of class `Interval`. For the other classes you will later add code to `testScript.m` in order to test the methods that you implement. You will submit `testScript.m` as a record of your development process.

If there is anything that you don't understand in this class, ask and figure it out before moving on! You want to make sure that you understand everything here before working on another class that depends on class `Interval`.

1.2 Class `Event`

Read the partially completed file `Event.m`. Notice that the property that has a non-numeric type is given an initial value so that its type is made clear. Read the given constructor and the associated comments carefully; note how `nargin` is used to check for the number of arguments passed.

Chaining up references Consider the following fragment:

```
e1= Event(3, 20, 10, .5, 4);
                  % An Event with id 4, importance .5, and duration 10.
                  % It's available for scheduling in the interval [3,20].
disp(e1.available.right)       % Should be 20
disp(e1.available.getWidth())  % Should be 17
disp(e1.id)       % Error: id is private
e1.setScheduledTime(5)
figure;  hold on
e1.draw()         % Should see colored box with left edge at x=5
hold off
```

Notice how two references are "chained together" using the dot notation:
`e1` references an `Event`.
An `Event` object has the field `available`, which references an `Interval`. Therefore
`e1.available` references an `Interval`.
An `Interval` has the property `right` and an instance method `getWidth`, therefore
`e1.available.right` is a scalar numeric value (the right end point) and
`e1.available.getWidth()` returns a scalar numeric value (the width).

Add the above statements to `testScript.m`, which exercises the constructor and several instance methods and confirms that property `id` is private. *As you work on the individual instance methods, after implementing each one be sure to add code to* `testScript.m` *to call that method as a test.* Test each method multiple times with different input values representative of various *valid* scenarios. It is tempting to rush through coding without stopping to test thoroughly, but that will burn more time in the end because you will have to deal with many confounding errors in your buggy atop buggy code when you finally get around to running your program. Make sure you understand everything in this class and solve any problems before moving on.

Comment out the test statements that cause errors, e.g., `disp(e1.id)` above, but do not remove such statements from `testScript`. `testScript` is a documentation of all the tests that you do.

1.3 Class `Schedule`

Read the partially completed file `Schedule.m`. The properties block is given; read it carefully. You need to implement the constructor and all the instance methods.

> Class `Schedule` involves a cell array of objects. Be sure to build a cell array by *directly assigning to a new indexed cell instead of using concatenation.* Any concatenation of a cell array of objects will result in a new *regular* array of objects, within which will be objects of the same type. Here is an example: suppose `x` references an `Event` object and `y` references an `Interval` object, you can create a cell array `c` of the two object handles with the statements `c{1}=x; c{2}=y`. However, if you use concatenation syntax `c=[x y]` then `c` will be a *regular* array of type `Event` where `c(1)` is the handle to `x` but `c(2)` will be the handle of a *new* object of type `Event` containing default property values—the `Interval` object referenced by `y` will not get stored in this regular `Event` array `c`. **Do not use concatenation on cell arrays of objects.**

Constructor Be sure to check the number of arguments passed using `nargin`. Assign to the field `window` only if the number of arguments is at least two. Assign to the field `name` only if the number of arguments is three.

addEvent This is a short method; don't be alarmed.

scheduleEvents We use a heuristic[2] to schedule events. Implement this instance method *according to the heuristic given* in the comments. Depending on the actual event data and the scheduling window, not all events in `eventArray` may get scheduled. (In the figure on page 1 you see both the scheduled and unscheduled events.)

draw This method draws the schedule, an example of which is shown on page 1. Start a figure window using the following command to make it full-screen[3]:

```
figure('units','normalized','outerposition',[0 .05 1 .95], 'name', 'Schedule')
hold on
```

At the end of the method, set the y-tick marks at integer (ID) values only and set the axis limits using the following commands:

```
set(gca, 'ytick', minId:maxId)
axis([xmin xmax minId-1 maxId+1])
```

where `minId` and `maxId` stores the lowest and highest ID values of the events in the event array, respectively, and `xmin` and `xmax` store the limits of the scheduling window. The schedule should be drawn only if the event array is not empty. Also use the command `hold off` at the end of the method.

In `testScript`, create several `Events` to add to a `Schedule` for testing:

```
e2= Event(0, 30, 8, .3, 1)
e3= Event(8, 25, 6, 0, 5)
s = Schedule(0, 40, 'Test Schedule')  % Instantiate a Schedule object.  s.eventArray is empty.
s.sname= 'New name'                   % ERROR: property sname has private set access
disp(s.sname)                         % Should see 'Test Schedule' since get access is public
s.addEvent(e2)                        % Add Event e2 to s.eventArray
s.addEvent(e3)
s.addEvent( Event( 10, 38, 5, 1, 2) )
disp(s)                               % s.eventArray should be a length 3 cell array of Events
s.eventArray{1}.getId()               % Should see 1
s.eventArray{1}.setScheduledTime(21)
figure;  hold on
s.eventArray{1}.draw()                % Should see colored box with left edge at x=21
hold off
```

Add more code to `testScript` to exercise every instance method defined in class `Schedule`! Do you understand every method call above? If not, ask for help! Make sure that all the methods work before moving on.

[2] A heuristic is a rule-of-thumb, or an experience-based method to solve a problem. A heuristic is often used instead of exhaustive search to avoid the computational cost associated with exhaustive search. Heuristics do not guarantee that the optimal solution will be found but generally the result is "good enough."

[3] Almost full-screen. The given vector [0 .05 1 .95] in the `figure` command specifies that the figure has a lower-left corner at 0% from the left and 5% from the bottom, a width of 100%, and a height of 95%.

1.4 Function `createSchedule`

We take a break from class definitions now and write a function to read event data from a file and then schedule the events. Read the partially implemented function `createSchedule`. (Note: this is an independent function in the file `createSchedule.m`, not an instance method inside some class definition.)

The given code opens and closes the data file and instantiates a `Schedule` object. You need to insert code to read the data file, instantiate `Event` objects and add them to the event array in `Schedule s`, schedule the events (by calling an appropriate method), and draw the schedule graphic (by calling an appropriate method). For now, *assume that there are* `Event` *objects only*, not `Course` objects.

Syntax note: The comment block showed the expression `L(30:end)` where `L` is a vector. The `end` keyword, when used as an index, is the last index of the vector. So `L(30:end)` is the same as `L(30:length(L))`.

Test your function using the data file `eventdata1.txt`, which contains data for `Events` only (no `Course`). For example, try to schedule the events in `eventdata1.txt` in a time window of 0 to 100, like this:
`[a, ex] = createSchedule('eventdata1.txt', 0, 100)`

Is everything working? If not, debug and if necessary get help from the course staff before moving on.

1.5 Class `Course`

Class `Course` is a child class of `Event`. Read the given constructor and then implement the instance methods as specified.

Graphics note: To place a course name centered in the color box of that course in the schedule, use the `text` built-in function and its property `HorizontalAlignment`, e.g.,

```
text(13, 4, 'Hola', 'HorizontalAlignment', 'center')
```

will place the text `Hola` horizontally centered over the x-coordinate 13 and at the y-coordinate 4. (In the future, you can always search the MATLAB documentation to identify the functions and their properties that allow one to produce informative, professional looking graphics!)

Consider these examples and add more to `testScript.m` for testing your code:

```
c1= Course(8, 25, 6, 0.5, 6, 'CS1000')
figure;  hold on
c1.draw()                % Should see white box with x range of 8 to 25
c1.setScheduledTime(9)
hold off
figure;  hold on
c1.draw()                % Should see colored box with left edge at x=9 and
                         %  the course name in the middle
hold off
s.addEvent(c1)
disp(s.eventArray)  % Should see that the last cell references a Course,
                    %  not an Event
```

Again, add code to `testScript` to test every instance method.

1.6 Function `createSchedule` (again)

You're at the final step! Now modify your code so that the function can handle `Course`s as well as `Event`s. Then use the data file `eventdata2.txt` to create a schedule! (You can modify the data file or create another file for more testing.)

Submit your files `testScript.m`, `Event.m`, `Schedule.m`, `Course.m`, and `createSchedule.m` on CMS.

2 H-Tree

In this problem you will write a recursive function `drawHTree` to draw an "H-tree." You will use recursion—no loops! The idea is that you start with one 'H', then add four smaller 'H's, one at each end of the previous 'H', then add four more 'H's at each end point, and again, and ...

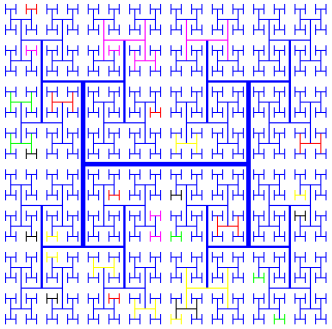

Figure 2: Level 5 H-tree

The H-tree is isn't just an interesting fractal that one can draw; it is the pattern used in laying out an interconnection network on microprocessor chips! Such a network has the feature that at any "level" the ends of the 'H' are the same distance from the center of the network. Consider a 2-level H-tree. The ends of the smaller 'H's

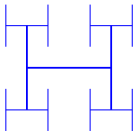

Figure 3: Level 2 H-tree

are the same distance from the center of the network. In microprocessor chip design this is important because the delay in routing time signal from the center clock to each part (end point) of the network is then the same. The H-tree is also a space-efficient layout for cramming components onto a chip!

Function `drawHTree` has the following specifications:

```
function drawHTree(x,y,len,w,level)
% Draw recursively an H-tree centered at (x,y).
% Each of the three lines of the 'H' has length len and width w.
% level is a non-negative integer indicating the level of the recursion.
```

At each level of recursion, the length and width of each line of the 'H' are halved. Therefore the level of recursion corresponds to the number of different sizes of 'H's in the tree. The level 5 H-tree in Figure 2 shows five different sizes of 'H's; the level 2 H-tree in Figure 3 shows two different sizes of 'H's. From Figure 2, you see that most of the 'H's are drawn in blue. Write your function so that there is a *one in ten* probability that an 'H' is drawn in a color that is not blue. In our implementation, one of the following colors are chosen randomly when a "non-blue" color is needed: black, red, yellow, magenta, green. (*Hint*: we used the char array `'krymg'` somehow.) You can choose your own colors, but your code should allow for at least four different colors to be used.

If you are having trouble with starting, read §14.1 of *Insight* (you can use similar ideas and organization from the `MeshTriangle` function to complete this problem). In fact, `drawHTree` is simpler because there's nothing to do when the minimum recursion level (0) is reached. Use the `plot` function to draw each line. For example,

```
plot([x1 x2], [y1 y2], 'Color', c, 'Linewidth', w)
```

draws a line of width `w` between (`x1`,`y1`) and (`x2`,`y2`) in color `c`, where `c` is an rgb vector or a predefined color name such as `'r'`, `'b'`, ..., etc. (Note: Don't worry about `plot`'s minimum line width. As long as the calculated line width is positive, `plot` will produce a line even though at some point the lines will stop getting thinner visually.)

As usual, decompose the problem! Start by drawing the H-tree in just one color and with one line width. Test your function to make sure that it draws the structure correctly! Be sure to check the base case, i.e., `level`=0

should produce *no* 'H'. `level`=1 should produce one 'H', `level`=2 should produce a tree with two different sizes of 'H', and so forth. After you have tested your function to see that it draws the structure correctly, then deal with the line width and implement the random color functionality.

Submit your function `drawHTree`. You can assume that `hold` is on and axes are appropriately set. For example, we can call `drawHTree` with the following script:

```
% Show drawHTree
close all
figure
x= 0;
y= 0;
len= 10;
width= 5;  % corresponds to the "Linewidth" property in function plot
level= 4;
axis equal off
hold on
drawHTree(x,y,len,width,level)
hold off
```

Submit your function file `drawHTree.m` in CMS.