# Scheduling CSP

Aravind Sundaresan – sundrsn2 (3 Units)
Krish Masand – masand2 (3 Units)

## 1.0 Setting up the Environment

To make our search method more intuitive, we created `Class`, `Semester`, and `Simulation` objects.

`Class` objects store all characteristics of a class, such as fall price, spring price, credit hours, and an `ArrayList` of prerequisites.

`Semester` objects store an `ArrayList` of `Classes`, an integer value which determines if it is a fall or spring semester (fall if not divisible by 2, spring if divisible by 2), the total number of hours that each `Class` in the semester adds up to, and the total cost that the cost of each `Class` in the semester adds up to.

`Simulation` objects contain most of the methods used for searching for classes, as `Simulation` objects set up the entire environment and instantiate the `Class` and `Semester` objects when necessary. A `Simulation` object is what takes in one of the actual text files provided, parses the data and assigns it to the right areas. It assigns the first value to an integer that holds the number of courses, the second value to the minimum credit requirement for each semester, and the third value to the maximum credit requirement for each semester. It similarly goes through every other line and assigns proper values. When creating `Class` objects, the methods creates the `Class` objects based on each line in the input file that describes the `Class`, and then adds that `Class` to an `ArrayList` of all `Classes`. It then revisits each `Class` in the `ArrayList` when it reaches the lines that assign prerequisites to each `Class`. It reaches the second to last line, sets the first value to the number of courses that the student is interested in, and then adds each value in the line after that to an `ArrayList` that contains the number of each course that the student is interested in. Lastly, it takes in the last line, which is assigned to an integer called "`budget`," which is actually never used because we did not have to keep track of the budget.

CSP `Class`ification:

Variables: `Classes`, `Semesters`

Constraints: Credit Hours, Prerequisites

Domain: `Simulations`

# 1.1 Creating the Schedule

Our backtracking search primarily uses two methods: The `Simulation` object's "*add*" and the `Semester` object's "*add*." These two methods are utilized by the `Simulation` object's "*schedule"* method to create a working schedule.

`Semester`'s *add* operates by taking in a `Class` object and returning the `Class` object that it added to the `Semester`'s *classes* `ArrayList`(they may not be the same `Class`). It then checks to see if adding the `Class` would overflow the `Semester` object that's adding the class (making sure that adding the class would not lead to the `Semester` object exceeding the maximum allowed credit hours). After it has checked for this, it then checks if any of the `Class`'s prerequisites have already been added to that `Semester`. If they have, the method returns null, and what happens next in this situation is handled by `Simulation`'s *add* (to quickly summarize, it adds that particular `Class` to the next possible `Semester` that does not already have any of its prerequisites). The method then checks if any of the `Class`'s prerequisites have not been taken yet by checking the `Class`'s remaining prerequisites. If there are prerequisites, the method is called recursively on the next prerequisite (this is why the original `Class` that the method takes in may not be the `Class` that the method returns). If there are no prerequisites left to be taken, the class is added to the `Semester`.

`Simulation`'s *add* uses `Semester`'s *add* along with other tools to add a `Class` to the next available `Semester`. It takes in a `Class` object and returns nothing. *add* starts off by checking to see if the `ArrayList` of `Semester` objects is empty, and if it is, it adds an empty `Semester` object to the `ArrayList`. It then adds a `Class` to the next available `Semester` using `Semester`'s *add*. If the `Class` that is returned is null, we know that one of the `Class`'s prerequisites is already in that available `Semester`, so we attempt to add it to the next available `Semester` that does not have one of its prerequisites. If one does not exist, a new `Semester` object is created, and the `Class` is put in that `Semester`. After a `Class` was successfully added, we add that `Class` to an `ArrayList` of `Classes` that have been taken.

Using these two methods, the *schedule* method first adds all of the courses that the student is interested in. These are added in order as provided in the input text file. During this process, each prerequisite for all of the courses is also added, as part of `Semester`'s recursive *add* function. After this, it is possible that some of the `Semesters` that have been created may not fulfill the minimum number of credit hours. To fulfill the minimum number of credit hours, we then add any class that has not been added using another implementation of `Simulation`'s *add* that never creates new `Semesters` (it takes in the `Class` we want to add and the number of the `Semester` we want to add it to, instead of just a `Class` object). After that class or its prerequisite has been added (depending on what `Semester`'s recursive *add* function returns), we check to see if the minimum number of credit hours has been met. If it hasn't, we continue to add `Classes` (skipping a `Class` that might potentially go past the maximum limit instead of just hitting the minimum number of credit hours). If it has, we move on the next `Semester` until every `Semester` is fulfilled.

# 1.2 Results

First Scenario:
2347 3
2 9 8
2 3 5
3 1 2 4
638 907 802

Second Scenario:
6432 3
4 8 19 12 7
6 11 14 1 4 15 3
3 13 1 6
1549 3391 1492

Third Scenario:
11010 5
4 27 8 13 28
4 20 25 8 13
5 14 28 12 3 13
4 30 15 7 3
4 16 21 17 8
2152 718 2809 2690 2641

Fourth Scenario:
15959 7
4 2 38 14 26
4 34 14 2 12
5 28 31 17 3 19
4 9 19 38 12
6 17 13 2 38 12 14
4 18 13 38 12
5 3 13 17 38 12
2218 1734 2855 1408 3193 1766 2785