Mathematical Sciences Major Course Scheduling Project

Group E

Hannah Fernandes, Hannah Milano, Alex Liu, Bora Odabasi

21-393 F19

Overview

At Carnegie Mellon, deciding which classes to take and in which order can cause students a high amount of unnecessary stress throughout their four years as an undergraduate student. There are a finite number of ways any student could schedule their required courses for their major in a given semester, and yet undergraduates still face the problem of high levels of stress during the selection process and once they are actually enrolled in the courses they have selected. This is especially true of the Mathematical Sciences major who needs to make decisions on which classes to take and in what order fairly blindly besides the necessary prerequisite classes. If there were a tool or algorithm to help students in the Mathematical Sciences major plan their schedule in an optimal way which would reduce their stress during both the selection process and during their time as a student, many math majors would experience far less stress. There are many things to consider when planning a course schedule of a math major at CMU. Mathematics students are frequently faced with required prerequisites which means that some classes must be taken in a certain order. There are also different professors who receive positive or negative feedback from previous students. There are course conflicts and courses not offered during certain semesters. There are FCEs, scores, and unit loads to consider. Students must prioritize which of these characteristics of a schedule matters the most to them and create a schedule that, to them, optimally manages their stress for the upcoming semester based on these inputs. There is also the stress of planning a schedule that has the least amount of free time between classes as possible so that you can take your classes back to back rather than having many small blocks of unproductive free time. With this project we hope to aid mathematics students by creating and implementing an algorithm that plans a schedule for a Mathematical Sciences major, which minimizes the maximum number of FCEs and minimizes the hours of free time during a student's week.

Current Problem

In this project, we would like to address the course scheduling problem of the Mathematical Sciences major with no specified concentration. We will consider the weights which we consider the most important: FCEs, FCE scores, and time between classes during the day. We will plan a schedule that meets the required courses for a Mathematical Sciences major to graduate in four years and meet all of their graduation requirements while minimizing their stress based on the inputs which we consider to have the largest impact on stress. We will also satisfy all prerequisites and ensure that this schedule could be followed by an actual CMU student in this major. We will also follow the necessary unit loads for a math major. The end result will be a schedule which, if followed, would allow a Mathematical Sciences major to graduate with the least amount of stress per semester based on the inputs that we consider most crucial when discussing a student's stress during a given semester. We consider the FCEs and FCE scores to be reliable because they are measures of how long a course takes based on the students' input and not just the professor's. We also want to minimize the breaks throughout the day so that students have classes back to back and not with breaks between classes because these breaks are frequently seen as a waste of time.

Assumptions

- This schedule has been made under the assumption that a student comes to CMU with no AP credit and does not test out of any classes
- All courses are offered in the semesters that we would like them to be offered in (for example, if our algorithm decides that we should take concepts in the first semester then we assume it is offered in that semester)
- We assume that the times that classes occur in future years is the same as in past years when formulating the actual schedule with times
- We assumed a student would want to take classes based on the lowest FCE value and not their interest level in the course

Overview of Course Scheduling and Dynamic Programming

We formulated constraints for the data that we gathered so that when we run our data through the constraints we arrive at a final schedule which satisfies the conditions described above to minimize the stress of the average CMU student. These constraints include minimizing the FCE load per semester, ensuring that all prerequisite classes come before any given scheduled class, ensuring that a class is taken at most one time, ensuring that we take at minimum 36 units and at most 54 units per semester in order for it to be a valid CMU schedule, ensuring we only satisfy a requirement one time. These constraints alongside of an objective function which minimizes the amount of FCEs per semester yielded a general schedule of which semesters we should take every required class in. We formulated a second objective function which minimizes the amount of time between classes during a given day when we schedule the lecture times of the classes our first objective function had chosen. The second objective function has constraints to ensure that no classes overlap and that we only select classes in the semesters that they are offered based on the schedules of previous semesters. Our final result is a schedule which minimizes the maximum number of FCEs in any semester and also minimizes the total amount of time in between classes during a student's schedule.

Constraints

Define variables:
$F = \{\text{classes offered in Fall}\}$
$S = \{\text{classes offered in Spring}\}$
$F \bigcap S = \{\text{classes offered in Fall and Spring}\}$
$P = \{(i, i') : \text{class i' is a prerequisite for class i}\}$
$N_k = \{i : \text{classes that satisfy requirement } k\}$

for $j = 1, 2, ..., 8$ and d $= 1, 2, ..., 7$:

$$x_{ij} = \begin{cases} 1 & \text{if class i is taken in semester j} \\ 0 & \text{otherwise} \end{cases}$$

$$s_{ijd} = \text{start time of class } i \text{ in semester } j \text{ on day } d$$

$$e_{ijd} = \text{end time of class } i \text{ in semester } j \text{ on day } d$$

$$u_i = \text{units of class } i$$

$$h_i = \text{FCE hours of class } i$$

Subject to:

1. $\sum_j x_{ij} \leq 1 \quad \forall i$: ensures each class is taken at most once
2. $x_{ij} < x_{i'j'} \quad \forall (i', i) \in P$: ensures prerequisites are satisfied
3. $36 \leq \sum_i u_i x_{ij} \leq 54 \quad \forall i$: ensures we take between 36 and 54 units per semester
4. $\sum_j x_{ij} = 1 \quad \forall i \in N$: ensures each requirement is satisfied exactly once
5. $\sum_{i \in F \setminus S} x_{ij} = 0 \quad \forall j \in 1, 3, 5, 7$:
   ensures that no class offered only in the Fall is taken in the Spring
6. $\sum_{i \in S \setminus F} x_{ij} = 0 \quad \forall j \in 2, 4, 6, 8$:
   ensures that no class offered only in the Spring is taken in the Fall
7. $s_{i'jd} \geq eijd \quad \forall j, d$ and (i, i') such that $s_{ijd} < s_{i'jd}$:
   ensures that class times don't overlap
   i.e. first class ends before second class begins

Objective Function: minimize maximum FCE's. Let $C', E1', E2', G'$ be set of Core classes, Elective 1 classes, Elective 2 classes and General Education classes respectively. Let $C_t \subseteq C', E1_t \subseteq E1', E2_t \subseteq E2', G_t \subseteq G'$ be set of classes taken.

$$X' := C' \cup E1' \cup E2' \cup G' = \text{set of all classes}$$

$$X_t := C_t \cup E1_t \cup E2_t \cup G_t = \text{set of classes taken}$$

For some set of untaken classes Y,

$$g(Y) := \text{all possible options/semester schedules given constraints above}$$

$$X := g(X' \setminus X_t) = \text{all possible options/semester schedules for next semester}$$

For $x \in X$, define $t(x) := \sum_{i \in x} h_i = \text{FCE hours for classes } x \text{ and } s \text{ be the semester}$

Objective is to find $f(0, \{\}, \{\}, \{\}, \{\})$ where we define

$$f(s, C_t, E1_t, E2_t, G_t) :=$$

$$\min_{x \in X} \Big(\max \Big(t(x), f(s+1, (C_t \cup x) \cap C', (E1_t \cup x) \cap E1', (E2_t \cup x) \cap E2', (G_t \cup x) \cap G' \Big) \Big)$$

Secondary Objective Function: given class schedule per semester from previous function, minimize total time gaps between classes for each day and semester

$$X = \{i \text{ such that } x_{ij} = 1\}$$

$$\min_{i \in X} \sum_{jd} [(\max e_{ijd} - \min s_{ijd}) x_{ij} - \sum_{i} (s_{ijd} - e_{ijd}) x_{ij}]$$

Our Solution / Results

Semester	Mathematics Courses	General Electives	Time Between Courses per Week
1	21-127 and 21-241	33-767 and 1 general elective	5 hours
2	21-120 and 21-373	2 general electives	12 hours
3	21-122 and 21-623	2 general electives	12 hours
4	21-259, 21-355 and 21-738	1 general elective	3 hours
5	21-341 and 21-765	36-225 and 1 general elective	12 hours
6	21-228 and 21-882	2 general electives	8 hours
7	21-260	15-390 and 2 general electives	2 hours
8	21-356	15-810 and 2 general electives	0 hours

The maximum FCE units per semester using this schedule is 33.89 units

Total time between classes weekly for all semesters is 54 hours

Conclusion

The planned schedule as stated above is, we believe, a schedule which would minimize the amount of stress that the average Carnegie Mellon University student in the Mathematical Sciences major has to endure. This claim is supported by the FCEs of each of the courses selected and the fact that the algorithm minimizes the amount of time between between courses in a given week. We hope that this algorithm and schedule can assist future students in this major select courses which have low FCE scores and will fit their schedule well. If this schedule is followed it will be the schedule which allows a student to graduate in four years while minimizing their FCE units and the amount of time between their classes. Our schedule is very similar to that which appears on the website for Mathematical Sciences majors which helps to support our conclusion. Ours is slightly different, however, which isn't surprising because we doubt that the faculty in the Math Department were considering the two key components that we considered while planning the suggestion schedule for students at Carnegie Mellon.

Afterthoughts

A major drawback of our algorithm was that it includes graduate student courses. These courses are valid and will allow a student to graduate, but because they are graduate classes the FCEs are likely recorded by graduate students and not undergraduate students so they may not be completely true for an undergraduate's experience. Another drawback is the fact that we weren't able to find the specific general electives a student should take, but because there are so many of them to fulfill the requirements it would have taken much longer for our program to run had we included those courses as well. We improved upon previous students' implementation of this algorithm or a similar one by including the second objective function to minimize gaps between classes as well as including some of the information about when to take general electives during your four years as a student. Please note that our data is all FCEs for both spring and fall semesters and was far too extensive to include in this report, but is available upon request if necessary.

Implementation

classObj.py

```
class Class:

  def __init__(self, id, sem, hours, lecData, recData, units):
  self.id = id
self.sem = sem
self.hours = hours
self.lecData = lecData
self.recData = recData
self.units = units

def __str__(self):
  return "ID: " + str(self.id) + "\nSemester: " + str(self.sem) + "\nFCE Hours:
  " + str(self.hours) + "\nLecture Data: " + str(self.lecData) + "\nRecitation
  Data: " + str(self.recData) + "\nUnits: " + str(self.units)
'''
e.g
id: 21127
sem: 0
hours: 9.43
lecData: ['MWF', '11:30AM', '12:20PM']
recData: [
  ['TR', '08:30AM', '09:20AM'],
  ['TR', '09:30AM', '10:20AM'],
  ['TR', '12:30PM', '01:20PM'],
  ['TR', '02:30PM', '03:20PM']
]
units: 10.0 ''
'''
```

Lookup.py

```
import csv

# only have data for 2019 fall and 2020 spring so a lot won't be accurate
def lookup(ids, spring):
    dict = {}
    id = 0
    units = 0
    lecData = ["","",""]
    recData = []
    layout = "SpringLayout.csv" if spring else "FallLayout.csv"
    with open(layout) as file:
        reader = csv.reader(file, delimiter='"')
        str_type = 0
        count = 0
        for row in reader:
            try:
                row_array = ''.join(row).split()
                if len(row_array) == 0:
                    str_type = 0
                    count = 0
                    continue
                if str_type == 0:
                    if int(row_array[0]) in ids:
                        id = int(row_array[0])
                        str_type = 1
                elif str_type == 1:
                    units = float(row_array[0])
                    ind = 3 if row_array[2] == '1' else 2
                    days = row_array[ind]
                    lecData = row_array[ind : ind+3]
                    str_type = 2
                    if row_array[1] != "Lec":
                        count = 4
                elif str_type == 2:
                    # only considering 1 lecture
                    # at most 3 recitations
                    if not row_array[0].isalpha() or count > 3:
                        dict[id] = (units, lecData, recData)
                        units = 0
                        lecData = ["","",""]
                        recData = []
                        str_type = 0
                        count = 0
                        if int(row_array[0]) in ids:
                            id = int(row_array[0])
                            str_type = 1
                        continue
                    if row_array[0] == "Lec" or len(row_array[1]) > 3:
                        continue
```

```
                    recData.append(row_array[1:4])
                    count += 1
            except (ValueError, IndexError):
                str_type = 0
                count = 0
                continue
    return dict


# ['Year'0, 'Semester'1, 'College'2, 'Dept'3, 'Course ID'4, 'Section'5, 'Name'6, 'Course Name'7, 'Level'8, 'Possible Respondents'9, 'Num Respondents'10, 'Response Rate %'11, 'Hrs Per Week'12, 'Hrs Per Week 5'13, 'Hrs Per Week 8'14, 'Interest in student learning'15, 'Clearly explain course requirements'16, 'Clear learning objectives & goals'17, 'Instructor provides feedback to students to improve'18, 'Demonstrate importance of subject matter'19, 'Explains subject matter of course'20, 'Show respect for all students'21, 'Overall teaching rate'22, 'Overall course rate'23]
```

Main3.py

```
1. import read
2. import math
3. import copy
4. import itertools
5. from collections import OrderedDict
6.
7. Prereqs = read.setPrereqs()
8. (CoreAvg, Elc1Avg, Elc2Avg) = read.avgAll()
9. Elc1, Elc2 = read.setElectives(Elc1Avg, Elc2Avg)
10.memo = {}
11.
12.def id_to_index(id):
13.    if id in CoreAvg:
14.        return list(CoreAvg).index(id)
15.    if id in Elc1:
16.        return 12 + list(Elc1).index(id)
17.    if id in Elc2:
18.        return 16 + list(Elc2).index(id)
19.    return ValueError
20.
21.def ind_to_obj(ind):
22.    if ind < len(CoreAvg):
23.        return CoreAvg[list(CoreAvg)[ind]]
24.    ind -= len(CoreAvg)
25.    if ind < len(Elc1):
26.        return Elc1[list(Elc1)[ind]]
27.    ind -= len(Elc1)
28.    if ind < len(Elc2):
29.        return Elc2[list(Elc2)[ind]]
30.    return IndexError
31.
32.def compute_options(class_bin, num_genEd):
33.    option_set = []
34.    # TODO: Change 3,5 to 3,7?
35.    for nc in range(3,5):
36.        # TODO: Change 2 to 6?
37.        for ge in range(3):
38.            for comb in itertools.combinations(range(class_bin.count(0)),
   nc-ge):
39.
40.                #set option: 1's for index of class in option
41.                option = [0] * len(class_bin)
42.                count = 0
43.                for i in range(len(class_bin)):
44.                    if class_bin[i] == 1:
45.                        continue
46.                    if count in comb:
47.                        option[i] = 1
48.                    count += 1
```

```
49.
50.                 # genEds count
51.                 genEds = ge
52.
53.                 # calculate total time for this semester
54.                 time = 9 * genEds
55.                 for i in range(len(option)):
56.                     if option[i] == 0:
57.                         continue
58.                     time += ind_to_obj(i).hours
59.
60.                 # 36 <= units <= 54 constraint
61.                 units = 9 * genEds
62.                 for i in range(len(option)):
63.                     if option[i] == 0:
64.                         continue
65.                     units += ind_to_obj(i).units
66.                 if units < 36 or units > 54:
67.                     continue
68.
69.                 # prerequisite constraint
70.                 prereq_satisfied = True
71.                 for i in range(len(option)):
72.                     if option[i] == 0:
73.                         continue
74.                     P_ID = Prereqs.get(ind_to_obj(i).id)
75.                     if P_ID is None:
76.                         continue
77.                     for id in P_ID:
78.                         ind = id_to_index(id)
79.                         if class_bin[ind] == 0:
80.                             prereq_satisfied = False
81.                 if not prereq_satisfied:
82.                     continue
83.
84.                 # don't exceed more units of gen ed than necessary (114 + 9)
85.                 #if num_genEd + genEds > math.ceil(114 / 9):
86.                 #    continue
87.
88.                 repeat = set()
89.                 for i in range(len(option)):
90.                     if option[i] == 0:
91.                         continue
92.                     data = ind_to_obj(i)
93.                     lec_data = data.lecData
94.                     rec_data = data.recData
95.                     # TODO
96.
97.                 option_set.append((option, time, genEds))
98.
```

```
    return option_set

def bin_format(arr, n):
    new = [0] * (n - len(arr))
    new.extend(arr)
    return new

def success_condition(class_bin, num_genEd):
    if all(x == 1 for x in class_bin) and num_genEd * 9 >= 0:
        return (0, [])
    else:
        return (1000000, [])

def optimal(state):
    if state in memo:
        return memo[state]
    dec_to_bin = bin_format([int(x) for x in bin(state)[2:]], 26)
    sem = int("".join(str(x) for x in dec_to_bin[:3]), 2)
    class_bin = dec_to_bin[3:22]
    num_genEd = int("".join(str(x) for x in dec_to_bin[22:]), 2)
    options = compute_options(class_bin, num_genEd)
    min = 1000000
    best_option = None
    fn_plus1 = None
    count = 0
    for (option, time, genEds) in options:
        # runtime: tracking progress
        if sem == 0:
            count += 1
            print(count, len(options), round(count/len(options),2))
        new_class_bin = [a ^ b for a, b in zip(option, class_bin)]
        new_genEd = num_genEd+genEds

        if sem != 7:
            # convert binary rep back to state int
            new_temp = bin_format([int(x) for x in bin(sem+1)[2:]], 3)
            new_temp.extend(new_class_bin)
            genEd_count = bin_format([int(x) for x in bin(new_genEd)[2:]],
4)
            new_temp.extend(genEd_count)
            new_dec = int("".join(str(x) for x in new_temp), 2)
            # f_n+1
            (fn_plus1_time, fn_plus1_sch) = optimal(new_dec)
            # add to memory
            memo[new_dec] = (fn_plus1_time, fn_plus1_sch)
            max_t = max(time, fn_plus1_time)
        else:
            max_t = time
            fn_plus1_sch = []

```

```
            if max_t < min:
                min = max_t
                best_option = option
                best_option.extend(bin_format([int(x) for x in bin(genEds)[2:]], 4))
                fn_plus1 = copy.copy(fn_plus1_sch)
        if fn_plus1 is None:
            return (2000000, [])
        fn_plus1.insert(0, best_option)
        if sem == 7 and not (all(x == 1 for x in new_class_bin) and new_genEd * 9 >= 114):
            return (3000000, [])
        return (min, fn_plus1)


    if __name__ == "__main__":

        unit, sch = optimal(0)
        print("Min max FCE hours per semester: " + str(unit))
        print("Using Schedule:")
        for i in range(len(sch)):
            print("Semester " + str(i) + ": ", end=" ")
            for j in range(19):
                if sch[i][j] == 1:
                    print(str(ind_to_obj(j).id), end=" ")
            print("| Gen Eds: " + str(int("".join(str(x) for x in sch[i][19:]), 2)))

    '''''
    Over 6 semesters, max 1 gen ed per semester
    Min max FCE hours per semester: 33.01
    Using Schedule:
    Semester 0:  21127 21241 21623 33767 | Gen Eds: 0
    Semester 1:  21120 21373 21738 | Gen Eds: 1
    Semester 2:  21122 21341 21765 | Gen Eds: 1
    Semester 3:  21259 21355 15812 | Gen Eds: 1
    Semester 4:  36225 21228 21882 | Gen Eds: 1
    Semester 5:  21260 21356 15390 | Gen Eds: 1

    Over 6 semesters, max 2 gen eds per semester
    Min max FCE hours per semester: 32.92
    Using Schedule:
    Semester 0:  21120 21127 21623 33767 | Gen Eds: 0
    Semester 1:  21122 21228 15812 | Gen Eds: 1
    Semester 2:  21259 21241 21738 | Gen Eds: 1
    Semester 3:  36225 21355 21373 21765 | Gen Eds: 0
    Semester 4:  21356 21341 21882 | Gen Eds: 1
    Semester 5:  21260 15390 | Gen Eds: 2

```

```
Over 7 semesters, max 2 gen eds per semester
Min max FCE hours per semester: 33.370000000000005
Using Schedule:
Semester 0:  21120 21127 21882 | Gen Eds: 1
Semester 1:  21122 21241 | Gen Eds: 2
Semester 2:  21259 21373 21765 | Gen Eds: 1
Semester 3:  21355 15812 | Gen Eds: 2
Semester 4:  36225 21356 21738 21623 | Gen Eds: 0
Semester 5:  21228 21260 15390 | Gen Eds: 1
Semester 6:  21341 33767 | Gen Eds: 2

Over 8 semesters, max 2 gen eds per semester, 3 or 4 courses per semester
Min max FCE hours per semester: 33.89
Using Schedule:
Semester 0:  21127 21241 33767 | Gen Eds: 1
Semester 1:  21120 21373 | Gen Eds: 2
Semester 2:  21122 21623 | Gen Eds: 2
Semester 3:  21259 21355 21738 | Gen Eds: 1
Semester 4:  36225 21341 21765 | Gen Eds: 1
Semester 5:  21228 21882 | Gen Eds: 2
Semester 6:  21260 15390 | Gen Eds: 2
Semester 7:  21356 15812 | Gen Eds: 2
'''
```

Read.py

```
import csv
import copy
from collections import OrderedDict

import classObj
import lookup as lk

'''''
Class Requirements from MCS website: Take all core classes, 45 units from Electives1, 27 units from Electives2. Some of thse courses overlap and may not be double counted.
Gen Eds are not considered
'''

CoreClasses = [21120,21122,21127,21228,21241,36225,21259,21260,21341,21355,21356,21373]

def isElective1(id):
    return (id >= 21300 and id < 22000) or id in [21270,21272,21292]

def isElective2(id):
    return isElective1(id) or (id >= 15200 and id < 16000) or (id >= 33300 and id < 34000) or (id >= 36300 and id < 37000)

'''''
Adds class to Type. Throws ValueError if class is discarded because not useful. If found repeat class in same semester, keep class with less FCE hours.
Input:  Type - type of class (Core, Electives1, Electives2)
        row - input row (see bottom for index info) of class containing all strings
        rightType - boolean if class id fits class Type
'''
def addClass(Type, row, rightType, id, f_lookup, s_lookup):
    if rightType:
        year = int(row[0])
        season = row[1] == "Spring"
        sem = 2 * (year - 2015) - season
        lookup = f_lookup if sem%2 == 0 else s_lookup
        if sem not in range(8):
            raise ValueError
        hours = float(row[12])
        units = lookup[id][0]
        lecData = lookup[id][1]
        recData = lookup[id][2]
        # take class with min hours
        if Type[sem].get(id) is not None:
            if Type[sem][id].hours < hours:
                raise ValueError
        Type[sem][id] = classObj.Class(id, sem, hours, lecData, recData, units)
```

```
44.
45. '''''
46. Reads FCE and gets classes relevant to Math major degree
47. Output: (Core, Electives1, Electives2)
48. Core[sem][id] contains class data of core class with id (21393) and semester
    (0-7)
49. See classObj.py for info on class object
50. '''
51.
52. def getClasses():
53.     Core = [{}, {}, {}, {}, {}, {}, {}, {}]
54.     Electives1 = [{}, {}, {}, {}, {}, {}, {}, {}]
55.     Electives2 = [{}, {}, {}, {}, {}, {}, {}, {}]
56.     ids = set()
57.     with open("Export.csv") as file:
58.         reader = csv.reader(file, delimiter=',')
59.         skip = True # skip first row
60.         for row in reader:
61.             if skip:
62.                 skip = False
63.                 continue
64.             try:
65.                 id = int(row[4])
66.                 if id in CoreClasses or isElective1(id) or isElective2(id):
67.                     ids.add(id)
68.             except ValueError:
69.                 continue
70.
71.     with open("Export.csv") as file:
72.         reader = csv.reader(file, delimiter=',')
73.         skip = True
74.         f_lookup = lk.lookup(ids, 0)
75.         s_lookup = lk.lookup(ids, 1)
76.         for row in reader:
77.             if skip:
78.                 skip = False
79.                 continue
80.             try:
81.                 id = int(row[4])
82.                 addClass(Core, row, id in CoreClasses, id, f_lookup, s_lookup)
83.                 addClass(Electives1, row, isElective1(id), id, f_lookup,
    s_lookup)
84.                 addClass(Electives2, row, isElective2(id), id, f_lookup,
    s_lookup)
85.             except (ValueError, KeyError):
86.                 continue
87.
88.     return (Core, Electives1, Electives2)
89.
90. '''''
```

```
Calculated via lowest FCE hour while still 9.0 units or higher
Input: ClassType is dict of id as key, Class Obj as value, N is int
Output: Ordered Dict of length N of Class Objs, ith value is ith best class
'''
def topN(ClassType,N):
    top = {}
    for i in range(N):
        max = 0
        bestk = 0
            for k in ClassType.keys():
                measure = 1 / ClassType[k].hours
                if measure > max and k not in top and ClassType[k].units > 8.9:
                    max = measure
                    bestk = k
            top[bestk] = ClassType[bestk]
        return top

    '''''
    Calculated via best unit to FCE hour ratio
    Input: ClassType is dict of id as key, Class Obj as value, N is int
    Output: Ordered Dict of length N of Class Objs, ith value is ith best class
    '''
    def topNRatio(ClassType,N):
        top = {}
        for i in range(N):
            max = 0
            bestk = 0
            for k in ClassType.keys():
                measure = ClassType[k].units / ClassType[k].hours
                if measure > max and k not in top:
                    max = measure
                    bestk = k
            top[bestk] = ClassType[bestk]
        return top

    '''''
    Input: Array of length 8, ith entry is dict of id key, Class Obj as value
    Output: Dict of id key, Class Obj as value
    '''
    def averageData(ClassType):
        data = {}
        IDs = set()
        for sem in range(8):
            IDs = IDs.union(ClassType[sem].keys())
        for k in IDs:
            fcehour = 0
            count = 0
            needTemplate = True;
            t = None
            for i in range(8):
```

```
                classData = ClassType[i].get(k)
                if classData is not None:
                    if k != classData.id:
                        print(k, classData.id)
                    if needTemplate:
                        needTemplate = False
                        t = copy.copy(classData)
                    fcehour += classData.hours
                    count += 1
            if t is None:
                continue
            t.hours = round(fcehour / count, 2)
            t.sem = None
            data[k] = t
        return data

    def avgAll():
        (Core, Electives1, Electives2) = getClasses()
        return (averageData(Core), averageData(Electives1),
    averageData(Electives2))

    '''''
    Since we hard code prerequisites and are limiting the classes, we don't need
    "or". So ouput will be id pointing to set of prerequisite ids
    '''
    def setPrereqs():
        dict = {}
        dict[21122] = {21120}
        dict[21228] = {21127}
        dict[36225] = {21259}
        dict[21259] = {21122}
        dict[21260] = {21122}
        dict[21341] = {21241, 21373}
        dict[21355] = {21127, 21122}
        dict[21356] = {21259, 21241, 21355}
        dict[21373] = {21127, 21241}
        dict[15390] = {36225}
        return dict

    def setElectives(E1, E2):
        Elc1 = {}
        Elc1[21882] = E1[21882]
        Elc1[21738] = E1[21738]
        Elc1[21623] = E1[21623]
        Elc1[21765] = E1[21765]
        Elc2 = {}
        Elc2[33767] = E2[33767]
        Elc2[15812] = E2[15812]
        Elc2[15390] = E2[15390]
        return (Elc1, Elc2)
```

```
189.
190.   '''''
191.   Too many gen eds to hard code. Just assigning random 76100 id to class with
   arbitrary data, 9 units and 9 FCE hours
192.   '''
193.   def getGenEd():
194.       dict = {}
195.       dict[76100] = classObj.Class(76100, None, 9.0, [], [], 9.0)
196.       return dict
197.
198.
199.   # ['Year'0, 'Semester'1, 'College'2, 'Dept'3, 'Course ID'4, 'Section'5,
   'Name'6, 'Course Name'7, 'Level'8, 'Possible Respondents'9, 'Num
   Respondents'10, 'Response Rate %'11, 'Hrs Per Week'12, 'Hrs Per Week 5'13, 'Hrs
   Per Week 8'14, 'Interest in student learning'15, 'Clearly explain course
   requirements'16, 'Clear learning objectives & goals'17, 'Instructor provides
   feedback to students to improve'18, 'Demonstrate importance of subject
   matter'19, 'Explains subject matter of course'20, 'Show respect for all
   students'21, 'Overall teaching rate'22, 'Overall course rate'23]
```