Paul Winzer
CS 520
Hood College Spring 2018
Homework 2

1. [COMPLETE]

We optimize on the price/weight ratio, rather than being greedy by just maximizing price or minimizing weight.
For both solutions, we calculate the price (profit) to weight ratio for all the items...

	1	2	3	4	5	6	7	8
w	20	24	14	20	18	20	10	6
p	15	9	27	12	36	12	9	12
p/w	0.75	0.375	1.9285	0.6	2	0.6	0.9	2

Then we order the items by p/w ratio in nonincreasing order.

	8	5	3	7	1	6	4	2
w	6	18	14	10	20	20	20	24
p	12	36	27	9	15	12	12	9
p/w	2	2	1.92857	0.9	0.75	0.6	0.6	0.375

Fractional solution:
M = 80
We examine the most valuable item based on p/w ratio, either 8 or 5, we use 8. If we take all of item 8, we reduce our knapsack capacity by 6, so remaining capacity is now 74.

	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	0	0	0	0	0	0	0	-
px	12	0	0	0	0	0	0	0	12
wx	6	0	0	0	0	0	0	0	6

We do the same thing for the next most valuable item, 5. If we take all of item 5, we reduce our knapsack capacity by 18, so remaining capacity is now 56.

	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	1	0	0	0	0	0	0	-
px	12	36	0	0	0	0	0	0	48
wx	6	18	0	0	0	0	0	0	24

We do the same thing for the next most valuable item, 3. If we take all of item 3, we reduce our knapsack capacity by 14, so remaining capacity is now 42.

	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	1	1	0	0	0	0	0	-
px	12	36	27	0	0	0	0	0	75
wx	6	18	14	0	0	0	0	0	38

We do the same thing for the next most valuable item, 7. If we take all of item 7, we reduce our knapsack capacity by 10, so remaining capacity is now 32.

	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	1	1	1	0	0	0	0	-
px	12	36	27	9	0	0	0	0	84
wx	6	18	14	10	0	0	0	0	48

We do the same thing for the next most valuable item, 1. If we take all of item 1, we reduce our knapsack capacity by 20, so remaining capacity is now 12.

	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	1	1	1	1	0	0	0	-
px	12	36	27	9	15	0	0	0	99
wx	6	18	14	10	20	0	0	0	68

Now we *try* the same thing for the next most valuable item, 6. When we try to take all of item 6, we would we would need a capacity of 20, but we only have 12, so we must take a fractional amount. If we take 0.6 of item 6 that is available, that will fill exactly the remaining 12 capacity in our knapsack.

	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	1	1	1	1	0.6	0	0	-
px	12	36	27	9	15	7.2	0	0	106.2
wx	6	18	14	10	20	12	0	0	80

Thus, our fractional greedy algorithm optimizing on p/w ratio has filled our knapsack with items of total price 106.2.

Binary solution:
The steps for the binary solution are identical to the steps for fractional solution above for the items 8,5,3,7, and 1. We arrive in the same state after taking all of item 1, so our remaining capacity is again 12.

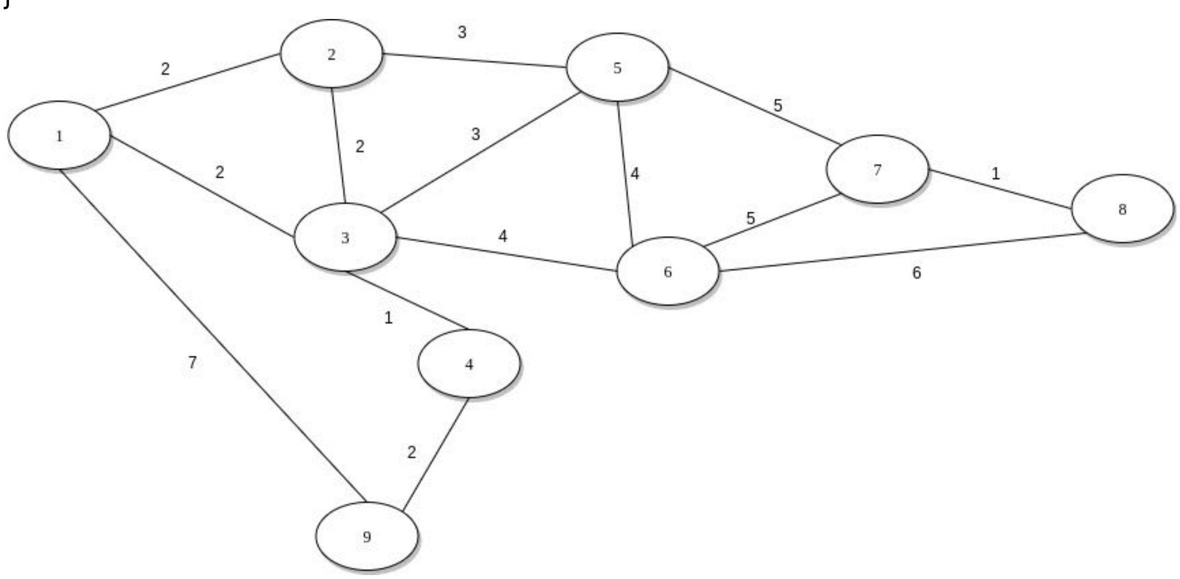
	8	5	3	7	1	6	4	2	Sum
w	6	18	14	10	20	20	20	24	-
p	12	36	27	9	15	12	12	9	-
p/w	2	2	1.9285	0.9	0.75	0.6	0.6	0.375	-
x	1	1	1	1	1	0	0	0	-
px	12	36	27	9	15	0	0	0	99
wx	6	18	14	10	20	0	0	0	68

At this point, we cannot take any of the remaining items: 6, 4, or 2. They have weights of 20, 20, and 24, respectively, so adding any of them would exceed the M = 80 total capacity of our knapsack. Thus, the binary greedy algorithm optimizing on p/w ratio has filled our knapsack with items of total price 99.

2. [COMPLETE]
    a. Prim's Algorithm for MST

Using the pseudocode on page 240 of the Horowitz text, with the variation on page 239 that allows starting at any vertex rather than the one with the edge of least weight.

Prim(*E, cost*, *n*, *t*)
// *E* is the set of edges in the graph
// *n* is the number of vertices in the graph
// *cost* is the cost adjacency matrix
// *t* is a multidimensional array that holds the edges in the MST
{
    mincost = 0;
    for *i* = 2 **to** *n* **do** near[*i*] = 1;
    near[1] = 0;
    for *i* = 1 **to** *n* - 1 **do**:
    { // find *n* - 1 edges in MST *t*
        Let *j* be an index such that *near*[*j*] $\neq$ 0 and
        cost[*j*, *near*[*j*]] is minimum;
        t[*i*,1] = *j*;
        t[*i*,2] = *near*[*j*];
        mincost = *mincost* + *cost*[*j*,*near*[*j*]];
        near[*j*] = 0;
        for *k* = 1 **to** *n* **do** // update *near*[]:
            if ((*near*[*k*] $\neq$ 0) **and** (*cost*[*k*,*near*[k]] > *cost*[*k*,*j*])):
                then *near*[*k*] = *j*;
    }
    return *mincost*;
}



Here is the *cost* adjacency matrix represented in a table

	1	2	3	4	5	6	7	8	9
1	0	2	2	inf	inf	inf	inf	inf	7
2	2	0	2	inf	3	inf	inf	inf	inf
3	2	2	0	1	3	4	inf	inf	inf
4	inf	inf	1	0	inf	inf	inf	inf	2
5	inf	3	3	inf	0	4	5	inf	inf
6	inf	inf	4	inf	4	0	5	6	inf
7	inf	inf	inf	inf	5	5	0	1	inf
8	inf	inf	inf	inf	inf	6	1	0	inf
9	7	inf	inf	2	inf	inf	inf	inf	0

First we initialize *mincost* = 0
Then we initialize *near*[] = [0, 1, 1, 1, 1, 1, 1, 1, 1]
Entering for loop:
i = 1
We choose *j* = 2, though we could also choose 3. They are the only indices where *near*[j] is not equal to 0, and they are the same so they are both the minimum.
t[1,1] = 2; *t*[1,2] = 1;
t = [[2,1]]
mincost = 0 + 2 = 2
near[2] = 0;
near[] = [0, 0, 1, 1, 1, 1, 1, 1, 1]
Update *near*[] for all indices where *near*[*k*] isn't 0 (3-9), we update for 5 only.
near[] = [0, 0, 1, 1, 2, 1, 1, 1, 1]

i = 2
We choose *j* = 3 because *near*[3] is not equal to 0 and it is the smallest *cost*[*j*, *near*[*j*]].
t[2,1] = 3; *t*[2,2] = 1;
t = [[2,1], [3,1]]
mincost = 2 + 2 = 4
near[3] = 0;
near[] = [0, 0, 0, 1, 2, 1, 1, 1, 1]
Update *near*[] for all indices where *near*[*k*] isn't 0 (4-9), we update 4, 6.
near[] = [0, 0, 0, 3, 2, 3, 1, 1, 1]

i = 3
We choose *j* = 4 because *near*[4] is not equal to 0 and it is the smallest *cost*[*j*, *near*[*j*]].
t[3,1] = 4; *t*[3,2] = 3;
t = [[2,1], [3,1], [4,3]]
mincost = 4 + 1 = 5
near[4] = 0;
near[] = [0, 0, 0, 0, 2, 3, 1, 1, 1]
Update *near*[] for all indices where *near*[*k*] isn't 0 (5-9), we update 9.
near[] = [0, 0, 0, 0, 2, 3, 1, 1, 4]

i = 4
We choose *j* = 9 because *near*[9] is not equal to 0 and it is the smallest *cost*[*j*, *near*[*j*]].
t[4,1] = 9; *t*[4,2] = 4;
t = [[2,1], [3,1], [4,3], [9,4]]
mincost = 5 + 2 = 7
near[9] = 0;
near[] = [0, 0, 0, 0, 2, 3, 1, 1, 0]
Update *near*[] for all indices where *near*[*k*] isn't 0 (5-8), we update nothing.
near[] = [0, 0, 0, 0, 2, 3, 1, 1, 0]

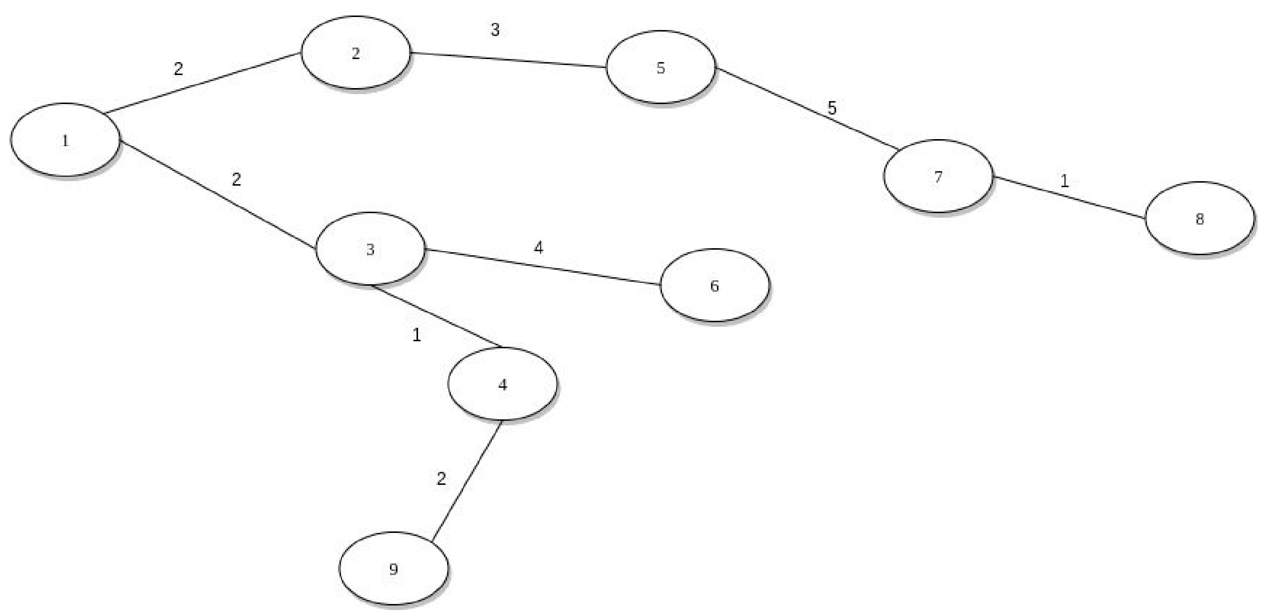
i = 5
We choose *j* = 5 because *near*[5] is not equal to 0 and it is the smallest *cost*[*j*, *near*[*j*]].
t[5,1] = 5; *t*[5,2] = 2;
t = [[2,1], [3,1], [4,3], [9,4], [5,2]]
mincost = 7 + 3 = 10
near[5] = 0;
near[] = [0, 0, 0, 0, 0, 3, 1, 1, 0]
Update *near*[] for all indices where *near*[*k*] isn't 0 (6-8), we update 7.
near[] = [0, 0, 0, 0, 0, 3, 5, 1, 0]

i = 6
We choose *j* = 6 because *near*[6] is not equal to 0 and it is the smallest *cost*[*j*, *near*[*j*]].
t[6,1] = 6; *t*[6,2] = 3;
t = [[2,1], [3,1], [4,3], [9,4], [5,2], [6,3]]
mincost = 10 + 4 = 14
near[6] = 0;
near[] = [0, 0, 0, 0, 0, 0, 5, 1, 0]
Update *near*[] for all indices where *near*[*k*] isn't 0 (7 and 8), we update 8.
near[] = [0, 0, 0, 0, 0, 0, 5, 6, 0]

i = 7
We choose *j* = 7 because *near*[7] is not equal to 0 and it is the smallest *cost*[*j*, *near*[*j*]].
t[7,1] = 7; *t*[7,2] = 5;
t = [[2,1], [3,1], [4,3], [9,4], [5,2], [6,3], [7,5]]
mincost = 14 + 5 = 19
near[7] = 0;
near[] = [0, 0, 0, 0, 0, 0, 0, 6, 0]
Update *near*[] for all indices where *near*[*k*] isn't 0 (8), we update 8.
near[] = [0, 0, 0, 0, 0, 0, 0, 7, 0]

i = 8
We choose *j* = 8 because *near*[8] is the only one not equal to 0.
t[8,1] = 8; *t*[8,2] = 7;
t = [[2,1], [3,1], [4,3], [9,4], [5,2], [6,3], [7,5], [8,7]]
mincost = 19 + 1 = 20
near[8] = 0;
near[] = [0, 0, 0, 0, 0, 0, 0, 0, 0]
We don't update *near*[] because there aren't any *k* where *near*[*k*] isn't 0.

Important stuff:
t = [[2,1], [3,1], [4,3], [9,4], [5,2], [6,3], [7,5], [8,7]]
mincost = 20
And it looks like this:



b. Shortest path from node 1 of same graph

```
ShortestPaths(v, cost, dist, n):
    for i = 1 to n do:
    {
        S[i] = 0;
        dist[i] = cost[v,i];
    }
    S[v] = 1;
    dist[v] = 0;
    for num = 2 to n do:
    {
        Choose u from the vertices not in S
        Such that dist[u] is minimum;
        S[u] = 1;
        for (each w adjacent to u with S[w] = 0) do:
        {
            if (dist[w] > dist[u] + cost[u,w]) then:
                dist[w] = dist[u] + cost[u,w];
        }
    }
```

v = 1
cost is the same as in part a.
n = 9
S[] = [0,0,0,0,0,0,0,0,0]
dist[] = [0,2,2,inf,inf,inf,inf,inf,7]
S[v] = 1
S[] = [1,0,0,0,0,0,0,0,0]
dist[] is unchanged

Entering **for** loop:
num = 2
For *u*, we can choose either 2 or 3. We choose 2.
S[*u*] = 1
S[] = [1,1,0,0,0,0,0,0,0]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. We adjust 5.
dist[5] = *dist*[2] + *cost*[2,5]
dist[] = [0,2,2,inf,5,inf,inf,inf,7]

num = 3
For *u*, we choose 3.
S[*u*] = 1
S[] = [1,1,1,0,0,0,0,0,0]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. We adjust 4 and 6.
dist[4] = *dist*[3] + *cost*[3,4]
dist[6] = *dist*[3] + *cost*[3,6]
dist[] = [0,2,2,3,5,6,inf,inf,7]

num = 4
For *u*, we choose 4.
S[*u*] = 1
S[] = [1,1,1,1,0,0,0,0,0]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. We adjust 9.
dist[9] = *dist*[4] + *cost*4,9]
dist[] = [0,2,2,3,5,6,inf,inf,5]

num = 5
For *u*, we choose 9, but we could also choose 5.
S[*u*] = 1
S[] = [1,1,1,1,0,0,0,0,1]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. There aren't any *w* adjacent to *u* where *S*[*w*] = 0 so we don't adjust anything in *dist*.

num = 6
For *u*, we chose 5.
S[*u*] = 1
S[] = [1,1,1,1,1,0,0,0,1]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. We adjust 7.
dist[7] = *dist*[5] + *cost*[5,7]
dist[] = [0,2,2,3,5,6,10,inf,5]

num = 7
For *u*, we chose 6.
S[*u*] = 1
S[] = [1,1,1,1,1,1,0,0,1]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. We adjust 8.
dist[8] = *dist*[6] + *cost*[6,8]
dist[] = [0,2,2,3,5,6,10,12,5]

num = 8
For *u*, we chose 7.
S[*u*] = 1
S[] = [1,1,1,1,1,1,1,0,1]
For all *w* adjacent to *u* where *S*[*w*] = 0, if there's a shorter path adjust *dist*[*w*]. We adjust 8.
dist[8] = *dist*[7] + *cost*[7,8]
dist[] = [0,2,2,3,5,6,10,11,5]

num = 9
For *u*, we chose 8, it's the only *S*[*u*] = 0.
S[*u*] = 1
S[] = [1,1,1,1,1,1,1,1,1]
Now there are no longer any *S*[*u*] = 0 so we don't adjust *dist*[].

Table showing steps:

num	*S*	Vertex Selected	1	2	3	4	5	6	7	8	9
Init.	{1}	--	0	2	2	inf	inf	inf	inf	inf	7
2	{1}	2	0	2	2	inf	5	inf	inf	inf	7
3	{1,2}	3	0	2	2	3	5	6	inf	inf	7
4	{1,2,3}	4	0	2	2	3	5	6	inf	inf	5
5	{1,2,3,4}	9	0	2	2	3	5	6	inf	inf	5
6	{1,2,3,4,9}	5	0	2	2	3	5	6	10	inf	5
7	{1,2,3,4,9,5}	6	0	2	2	3	5	6	10	12	5
8	{1,2,3,4,9,5,6}	7	0	2	2	3	5	6	10	11	5
9	{1,2,3,4,9,5,6,7}	8	0	2	2	3	5	6	10	11	5

3. [COMPLETE]
    a. Pseudocode

Data structures overview:

1. Input param *activities* is a list of objects with two properties, *start* and *finish*
2. *LectureHall* is an object class with properties *activities* (list), and *id* (int)
3. *firstAvailableTimes* is a min heap of tuple pairs (*finishTime*, *lectureHallId*)
    a. At all times, there are exactly *n* number of entries in the heap, where *n* is the number of lecture halls that have been assigned at least one activity
    b. The heap is ordered by the *finishTime* properties of each tuple, so that the next available lecture hall is the root entry's *lectureHallId*
    c. *firstAvailableTimes.min()* returns the tuple at the root without altering the heap
    d. *firstAvailableTimes.minPop()* deletes the root node from the heap, adjusts the heap, and returns the root node tuple
4. *lectureHalls* is a dictionary holding all the *LectureHall* objects created so far, keyed by the *LectureHall*'s id. You could easily use an array/list here and operate with the index but I chose dictionary/hash table so I could follow the assignments more directly

AssignActivities(activities)

```
// int, next hall id to be assigned, equal to number of existing halls
hallCount = 0

// a dictionary to hold the lecture halls
lectureHalls  = {}

// a min heap of tuples (finishTime, lectureHallId), ordered on finishTime
firstAvailableTimes = []

// sort list of activities by finish time, nondecreasing order
activities.sortByFinishTime()

for a in activities:
    if firstAvailableTimes.isEmpty() or firstAvailableTimes.min().finishTime > a.start:
        lh = new LectureHall()
        lh.activities = [a]
        // assign hallCount's value as the new LectureHall's id, increment hallCount
        lh.id = hallCount++
        // insert the lecture hall into the dict using its key
        lectureHalls[lh.id] = lh
        // insert the entry for this lecture hall into the heap
```

firstAvailableTimes.insert((a.finish, lh.id))
else:
key = firstAvailableTimes.minPop().lectureHallId
lectureHalls[key].activities.append(a)
firstAvailableTimes.insert((a.finish, lectureHalls[key].id))

b. Code, also submitted separately

```
import heapq
import sys

class Activity(object):
	'''
	represents an activity to be assigned to a LectureHall

	instance attributes:
	s : int, start time
	f : int, finish time
	'''

	def __init__(self, start, finish):
		self.s = int(start)
		self.f = int(finish)



class LectureHall(object):
	'''
	class that represents a lecture hall
	for the purposes of assigning Activity objs

	class attributes:
	count : int, incrementer used to give instances unique ids
	available_times : min heap, latest Activity finish time for each
LectureHall

	class methods:
	first_available_time()
	reset()

	instance attributes:
	activities : list of Activity objs assigned to the instance

	instance methods:
	__init__(Activity)
	assign(Activity)

	'''
```

```
    # number of LectureHalls, used to give them int IDs
    count = 0

    # min heap
    available_times = []


    @classmethod
    def first_available_time(c):
        '''
        returns top of LectureHall.end_times min heap, without popping it
off
        '''
        return LectureHall.available_times[0]

    @classmethod
    def reset(c):
        '''
        resets all class attributes
        '''
        LectureHall.count = 0
        LectureHall.available_times = []

    def __init__(self, activity):
        '''
        initialize by adding Activity to instance's list
        and pushing the activity's finish time onto available_times
        '''
        self.activities = [activity]

        self.id = str(LectureHall.count)
        LectureHall.count = LectureHall.count + 1

        time_tup = (activity.f,self.id)
        heapq.heappush(LectureHall.available_times,time_tup)


    def assign(self, activity):
        '''
        append Activity to the LectureHall instance's activites list
        and replace the top of the heap with the new available time
        '''
        self.activities.append(activity)

        time_tup = (activity.f, self.id)
        heapq.heapreplace(LectureHall.available_times,time_tup)
```

```
def AssignActivities(file_name):
      '''
      Greedy algorithm to assign Activitys to LectureHalls
      based on the machine scheduling problem

      params
      file_name : str, containing the path to a file of the following format

      {n}
      {s1,f1}
      {s2,f2}
      ...
      {sn,fn}

      where {n} is the number of activities to be assigned
      and all n lines afterward contain the start and finish times
      of those activities {s,f}
      '''

      # reset LectureHall class variables so that you can run
      # different inputs in succession on shell
      LectureHall.reset()

      activities = []
      lecture_halls = {}


      with open(file_name) as f:
            doc = f.readlines()

      n = int(doc[0])

      activity_data = [a.strip().split(',') for a in doc[1:]]

      # put all Activity objects in a list
      for i in range(0,n):

activities.append(Activity(activity_data[i][0],activity_data[i][1]))

      # sort activities by their start times
      activities.sort(key=lambda a : a.s)

      for a in activities:
            if not LectureHall.available_times or
LectureHall.first_available_time()[0] > a.s:
```

```
                # if there aren't any schedule activities yet
                # or the one with the earliest end time is later than this
activity's start
                # assign the activity to a new lecture hall
                # and key the lecture hall into the dict
                lh = LectureHall(a)
                lecture_halls[lh.id] = lh

            else:
                # get the lecture hall key from the top of the min heap of
times
                # and assign this activity to that key's entry in the
dictionary
                key = LectureHall.first_available_time()[1]
                lecture_halls[key].assign(a)


        # print the final assignments
        for key in lecture_halls:
            print key
            for a in lecture_halls[key].activities:
                print a.s,a.f



#
# run it from cmd line if filename arg is given
#
if len(sys.argv) > 1:
    AssignActivities(sys.argv[1])
```

b. Explain time complexity

Ignoring the overhead of initializing data structures, which takes constant time, we first examine the sorting of the *activities* array input parameter. The time complexity of the sorting algorithm in my implementation is O(n * log(n)) as described here: http://svn.python.org/projects/python/trunk/Objects/listsort.txt.
On to the for-loop, which is executed *n* times, once for each activity in the *activities* input, we first check to see if the min element in *firstAvailableTimes* is smaller larger than the start time of the activity. This takes O(1) time and does not change our current complexity of O(n * log(n)). Everything in the if-block takes constant time until the insert into the min heap, which takes O(n * log(n)), see here: https://docs.python.org/2.7/library/heapq.html. Similarly in the else-block, the heap pop and the insert into the heap each take O(n * log(n)) time. Looking up the LectureHall by id in lectureHalls should take O(1), here: https://wiki.python.org/moin/TimeComplexity
The worst case time complexity for each of the subroutines is O(n * log(n)) and so is the time complexity for the algorithm overall.

4. [INCOMPLETE - LACK OF MASTERY]
   Negative weights for MST algos?
   Yes, Prim's and Kruskal's algorithms for MSTs work correctly when the graphs they examine have edges of negative weights.

   At every step of Prim's algorithm, we look at the nodes that aren't yet included in the MST and examine the node that has the edge of lowest weight. We add that edge to the edges in the MST, we remove that second node on the edge from the potential nodes to add to the MST, and then we see if that second node is closer to any of the remaining nodes that the nodes we have already examined.

   If one or more of the edges in the graph have negative weight, everything goes exactly the same way. Since we are only calculating the paths from each node to its adjacent node, and not any further away, there is no chance that we can end up with a non-minimum spanning tree because of negatively weighted edges. Not like a Shortest Path Algorithm that will alter the distance from A to B is a shorter route is found.

   I feel like a fairly simple proof by contradiction or induction is necessary here but I can't quite grasp the angle.