INDEX

Project Part #1 2
Computing environment 2
The distribution 2
Creating different distributions' methods: 2
Graphical histograms of how many attendees attending each session 3
Bounding functions for running times and space: 6
Algorithms for removing duplicates: 7
Output with N, M, T, S & K: 9
Appendix 10
Project Part #2 15
Computing environment 15
Creating three different ordering methods 15
Method 1 --- Smallest Last Vertex Ordering 15
Method 2 --- Largest Last Vertex Ordering 19
Method 3 --- Random Vertex Ordering 22
Different Ordering Methodologies Compare 24
Asymptotic running time 24
Asymptotic space requirement 26
Total number of colors needed 26
Appendix 27

Project Part #1

Computing environment

Operating System: Windows 10, 64bits
Manufacturer: LENOVO
Processor: Intel® Core™ i5-6200U @ 2.30GHz~2.40GHz
Memory: 8076MB RAM
Platform: PyCharm 2018.2.1 (Community Edition)

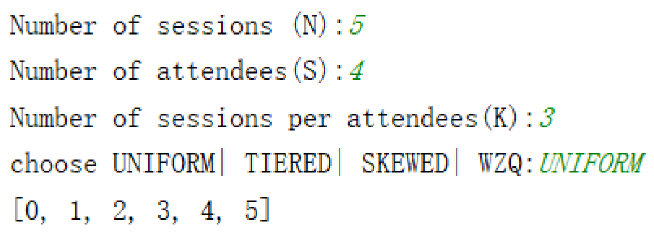
The distribution

I chose to achieve Uniform, Skewed and Two Tiered Distribution. I can simply choose different distribution through input DIST = UNIFORM | TIERED | SKEWED | WZQ

Creating different distributions' methods:

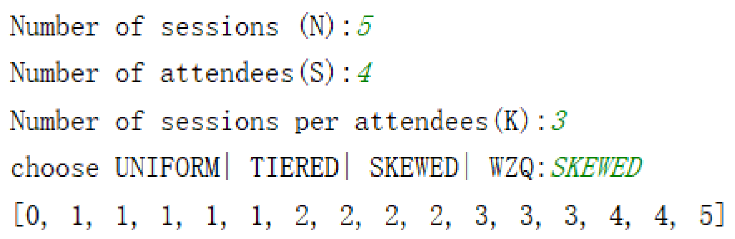
I use the Python internal function: *random.sample(population, K)* to creating different distributes. The function could return a K length list of unique elements chosen from the population sequence, used for random sampling without replacement. If the population contains repeats, then each occurrence is a possible selection in the sample. So with this function, all I need to do is give *population* sequence different number of sessions with different distribution(opportunity).

<u>Uniform Distribution:</u>
If the N=5, S=4, K=3, the *population* sequence will be [0, 1, 2, 3, 4, 5]. So for each attendees, he/she has the same opportunity to choose either 1 or 2 or 3 or 4 or 5(0 represents not attend session).

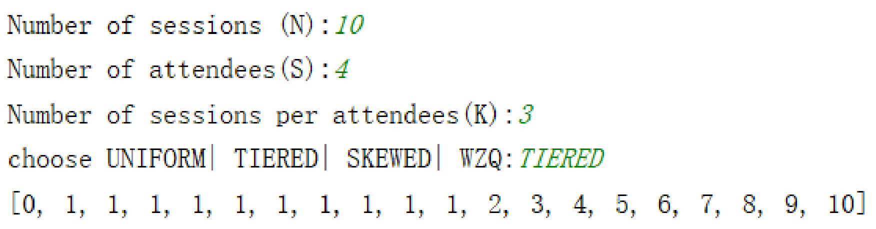


```
Number of sessions (N):5
Number of attendees(S):4
Number of sessions per attendees(K):3
choose UNIFORM| TIERED| SKEWED| WZQ:UNIFORM
[0, 1, 2, 3, 4, 5]
```

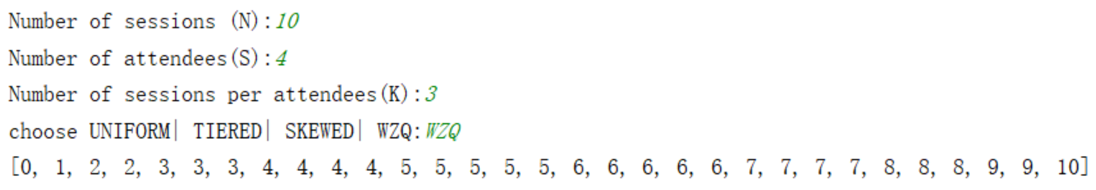
<u>Skewed Distribution:</u>
If the N=5, S=4, K=3, the *population* sequence will be [0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5]. The number of first element is N, the second one is N-1, the third one is N-2, …… , the last one is 1. So when choosing, the smaller numbered sessions are more likely be chosen. In this particular example, session 1 has 5/16 opportunity to be chosen, session 2 has 4/16, session 3 has 3/16, session 4 has 2/16, session 5 has 1/16.


```
Number of sessions (N):5
Number of attendees(S):4
Number of sessions per attendees(K):3
choose UNIFORM| TIERED| SKEWED| WZQ:SKEWED
[0, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5]
```

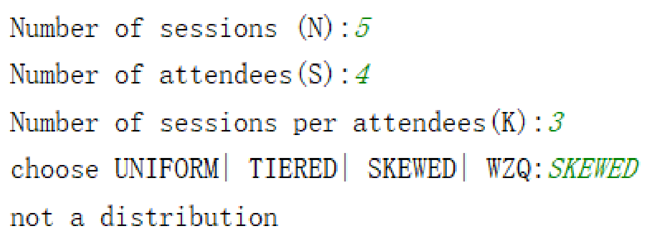
Two Tiered Distribution:
If the N=10, S=4, K=3, the *population* sequence will be [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. So the first 10% element which in this case is 1, has the opportunity 50% to be chosen.


```
Number of sessions (N):10
Number of attendees(S):4
Number of sessions per attendees(K):3
choose UNIFORM| TIERED| SKEWED| WZQ:TIERED
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

My own distribution:
If the N=10, S=4, K=3, the *population* sequence will be [0, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9, 10]. I named the distribution to WZQ(which is my name abbreviation). It is like two tiered skewed distribution. The middle session number was mostly possible to be chosen. And the smallest and biggest session number was least possible to be chosen.

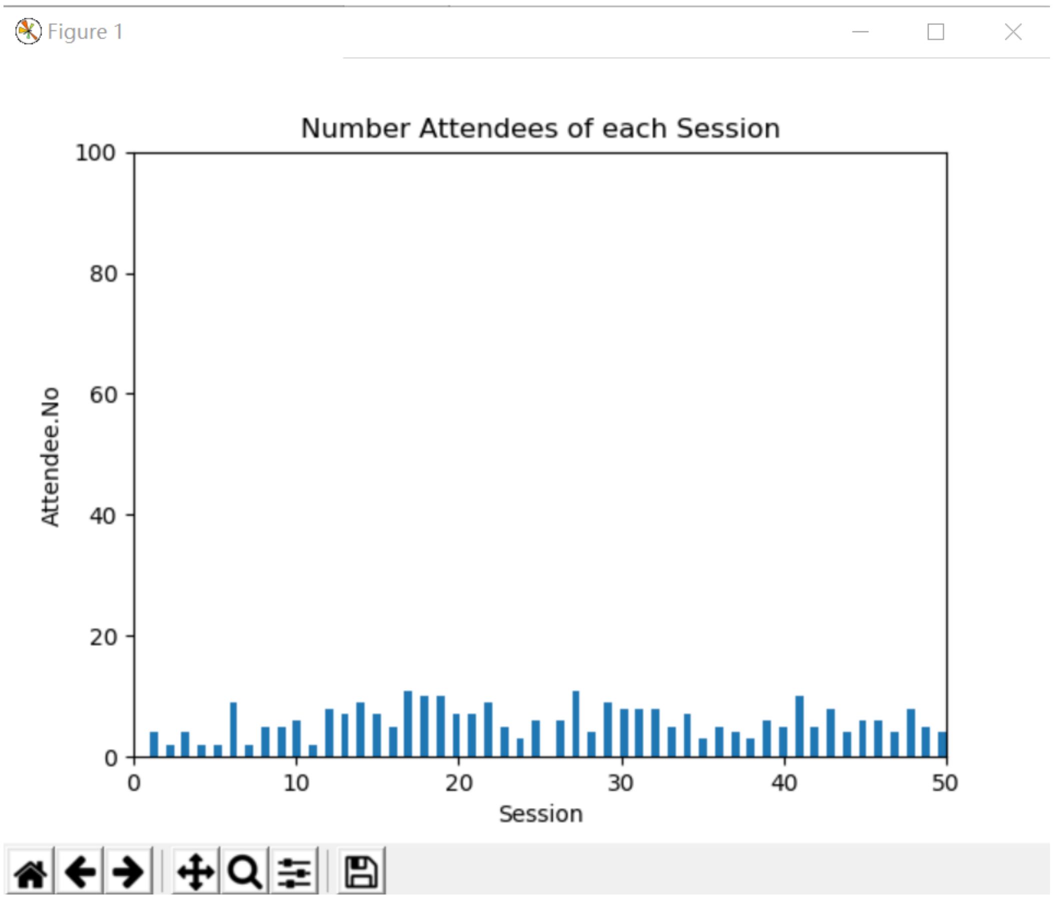

```
Number of sessions (N):10
Number of attendees(S):4
Number of sessions per attendees(K):3
choose UNIFORM| TIERED| SKEWED| WZQ:WZQ
[0, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 9, 9, 10]
```

Also, all the distribution above have different MAX value for the K. In my program, I write a switch to judge whether the K is in the scope. If it is not, the output will show " NOT A DISTRIBUTION" like the screen shot below(the MAX of skewed is 0.1N):

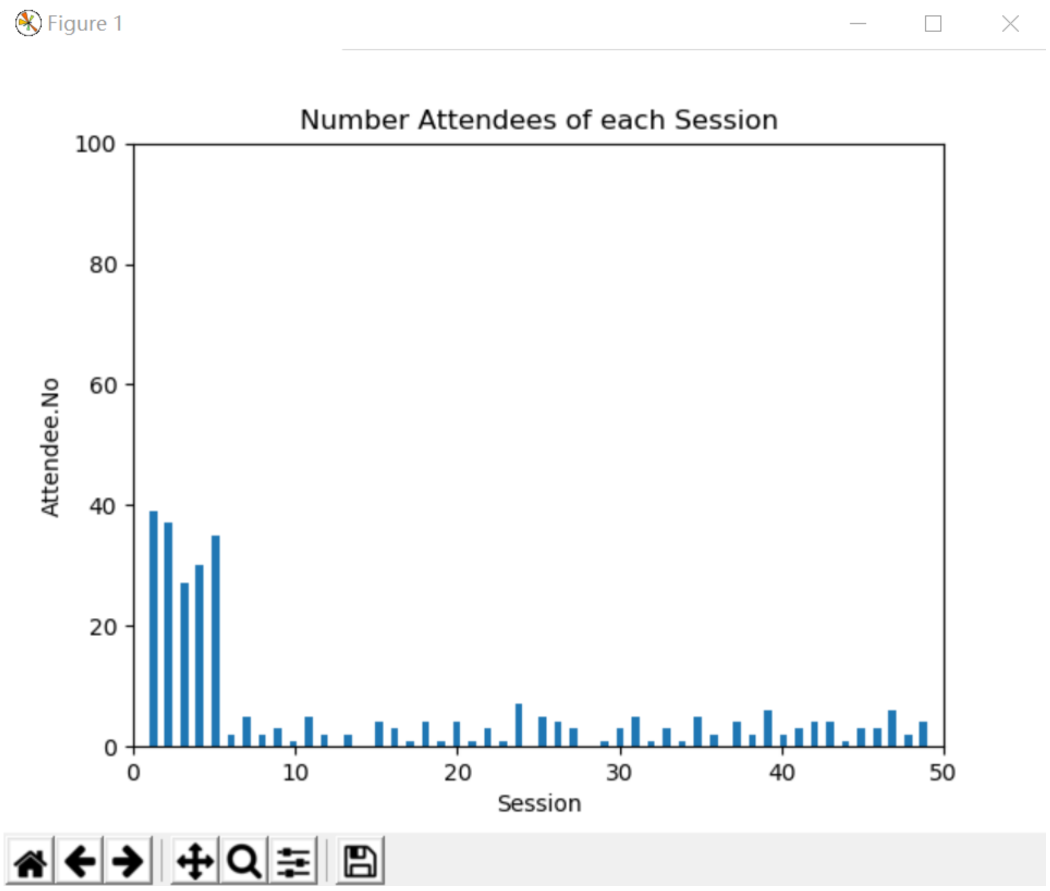

```
Number of sessions (N):5
Number of attendees(S):4
Number of sessions per attendees(K):3
choose UNIFORM| TIERED| SKEWED| WZQ:SKEWED
not a distribution
```

Graphical histograms of how many attendees attending each session

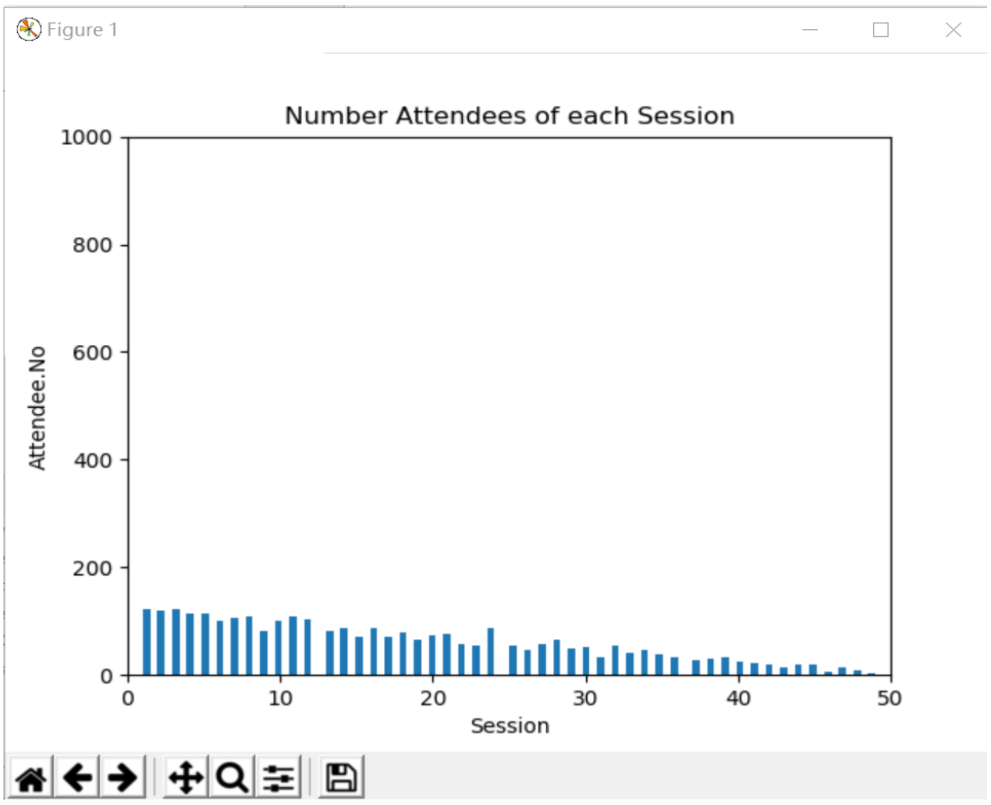
I use *pyplot* to draw the graphical histograms. Here I will give some examples of the graphical histogram of how many attendees actually attending each session. In the graph, X axis represents sessions number, Y axis represents attendees number.
Example 1:
N=50, S=100, K=3, DIST=UNIFORM



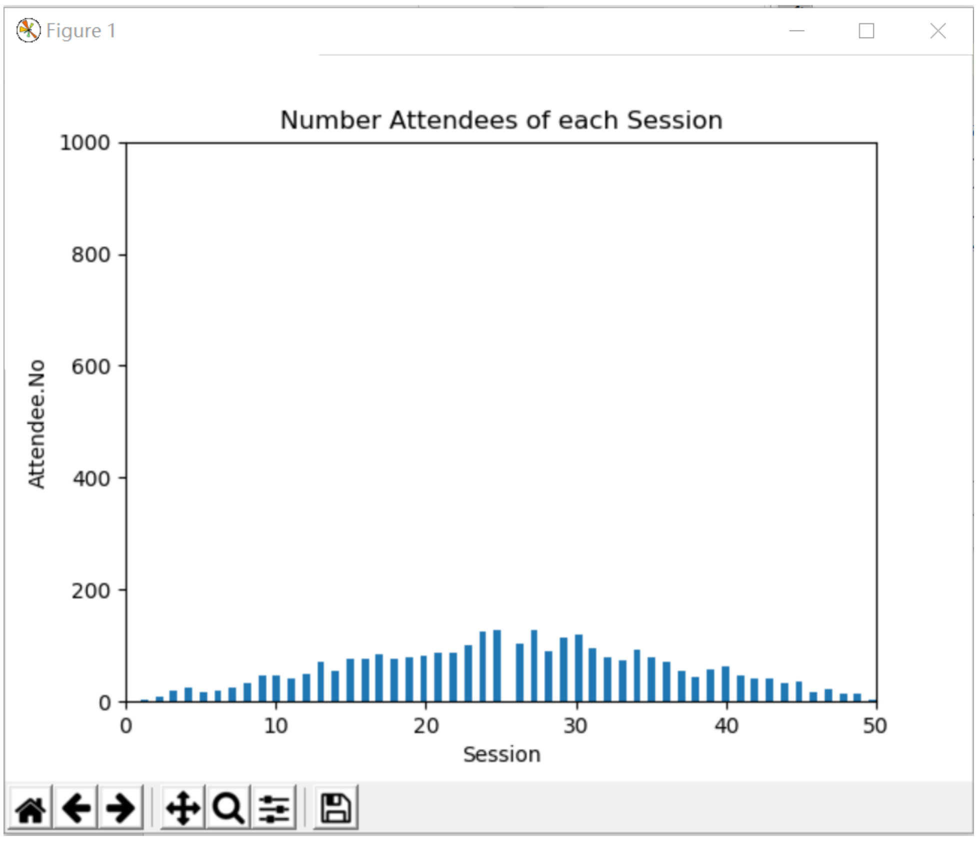
N=50, S=100, K=3, DIST=TIERED



N=50, S=1000, K=3, DIST=SKEWED



My own distribution(WZQ):
N=50, S=1000, K=3, DIST=WZQ



Bounding functions for running times and space:

The asymptotic bounding functions for the running times of calculating distributions:
Uniform distribution: O(N) ---- Because the program has 1 for-loop in this method.
Skewed distribution: O(N^2) ---- Because the program has two for-loop in this method.
Two Tiered distribution: O(N^2) ---- Because the program has two for-loop in this method.
The asymptotic bounding functions for the space of calculating distributions:
Uniform distribution: O(N)
Skewed distribution: O(N(N+1)/2) = O(N^2)
Two Tiered distribution: O(2N)
I also use space to store the session chooses of each attendees which is K*S. Because I build a class called *Attendees(aid, attend)*, which *aid* represents attendee number, *attend* is a list of session number he/she chooses.
The following is the runtime examples to demonstrate my asymptotic bounds:

Uniform distribution:

N	S	K	Time(ms)
100	100	3	0
500	100	3	31244
1000	100	3	93727
2000	100	3	372636
4000	100	3	262775
5000	100	3	985402
1000	200	3	78107
1000	1000	3	93727
1000	10000	3	239301
100	100	30	0
100	100	3000	0
100	100	30000	0

From the table above, we find that K is no relation with the running time. N and S have the relation with running time and it does not seem like square. But I don’t use S while I generate distribution. Maybe the codes after distributing affect this running time.

Skewed distribution:

N	S	K	Time(ms)
100	100	3	0
500	100	3	15663
1000	100	3	62485
2000	100	3	338487
4000	100	3	323219
5000	100	3	882392
1000	200	3	93692
1000	1000	3	93725

1000	10000	3	187451
1000	100	30	93731
1000	100	60	188671
1000	100	90	203070

From the table above, I found that N, S and K all relating to the running time. But through the code, it should be O(N^2). So I think maybe the codes after distributing affect this running time. Maybe the data is too limited and I should do thousands times of experiments.

Two Tiered distribution: O(N^2)

N	S	K	Time(ms)
100	100	3	0
500	100	3	15660
1000	100	3	81159
2000	100	3	328049
4000	100	3	248737
5000	100	3	331285
1000	200	3	109347
1000	1000	3	74178
1000	10000	3	102179
1000	100	30	78104
1000	100	100	159102
1000	100	1000	218697

From the table above, we find that S is kind of no relation with the running time. N and K have the relation with running time and it does not seem like square. But I don't use S and K while I generate distribution. Maybe the codes after distributing affect this running time.

Algorithms for removing duplicates:

O(N^2) space:

I define a function called *removeDuplicatesN2 (pair_list)*. The parameter is the collision pair with duplicates like (1,2) and (2,1). I use matrix to remove the duplicates. Firstly, I convert collision list which the element are pairs of tuples. The element in Row x, Column y = 1 means that (x,y) is a collision.

The algorithm is that traverse all the element in the matrix, is the element in Row x, Column y = 1 and in the meanwhile the element in Row y, Column x = 1, it means duplicate like (1,2) and (2,1). Then make one of their value in matrix to 0.

A simple example: N = 5, S = 4, K = 3

```
[[0, 0, 1, 1, 0], [1, 0, 0, 0, 1], [0, 0, 0, 0, 0], [0, 1, 1, 0, 1], [0, 1, 0, 0, 0]]
[[0, 0, 1, 1, 0], [1, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 1, 1, 0, 1], [0, 1, 0, 0, 0]]
```

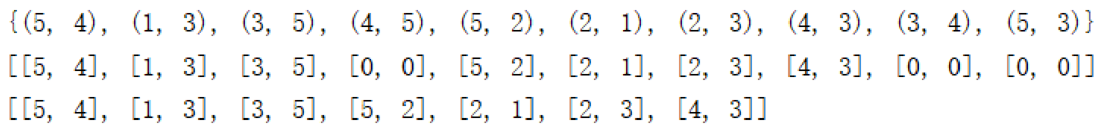
The first line is the matrix(comes from collision list) with duplicates, and the second is

the matrix without duplicates. It shows that (2,5) and (5,2) are duplicated. And (2,5) was eliminated.
Finally, converting matrix to list could benefit other output function.
Space required to remove the duplicates: O(N^2)
The asymptotic bounding functions for the running times: O(N^2) ---- Because the program has two for loop in this method.

N	S	K	M	T	Time
100	50	3	149	146	0
100	100	3	281	276	0
100	1000	3	2562	2250	0
100	1000	10	9782	4949	31245
100	1000	15	9900	4950	31238
100	1000	20	9900	4950	93724
1000	100	3	300	299	421482
5000	100	3	300	300	895634
10000	100	3	300	300	869405

From the table above, the S has no relation with the running time. K has a big effect on the running time, when K become bigger and bigger, the increment of running time is big. So the running time relates with two variables and it is O(N^2) = O(S*K).

O(M) space:
I define a function called *removeDuplicatesM (pair_list)*. The parameter is the collision pair with duplicates like (1,2) and (2,1).
The algorithm works like this: I use list to remove the duplicates. Firstly, I traverse al the tuples in *pair_list.* For each element (x,y), I compare it with other element in *pair_list.* If I find the element (y,x), I change it to (0,0). Finally, just removing all (0,0) from the list, you can get the list without duplicates.
A simple example: N = 5, S = 4, K = 3



```
{(5, 4), (1, 3), (3, 5), (4, 5), (5, 2), (2, 1), (2, 3), (4, 3), (3, 4), (5, 3)}
[[5, 4], [1, 3], [3, 5], [0, 0], [5, 2], [2, 1], [2, 3], [4, 3], [0, 0], [0, 0]]
[[5, 4], [1, 3], [3, 5], [5, 2], [2, 1], [2, 3], [4, 3]]
```

The first line is list with duplicates: (5, 4) and (4, 5), (3,4) and (4, 3), (3, 5) and (5, 3).
The second line is the list after make one of the duplicating conflicts to (0, 0).
The third line remove the (0,0) and that is the final result.
Space required to remove the duplicates: O(M)--- Because it is a M length list.
The asymptotic bounding functions for the running times: O(N^2). One reason I draw this conclusion is that in the program, I use 1 for-loop nested 1 for-loop. I will use the following table to demonstrate this.

N	S	K	M	T	Time
100	50	3	143	143	0
100	100	3	295	289	11575
100	1000	3	2553	2242	780124

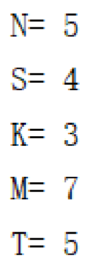
100	1000	10	9771	4950	150199
100	1000	15	9900	4950	735412
100	1000	20	9900	4950	97466
1000	100	3	300	300	11787
5000	100	3	300	300	7445
10000	100	3	300	300	17037

The table above shows that when S and K change, the running time change. So the running time relates with two variables and it is $O(N^2) = O(S*K)$.

Output with N, M, T, S & K:

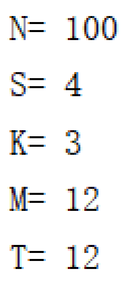
I did not find the test data in canvas. So I test it with my own data.

OUTPUT 1:



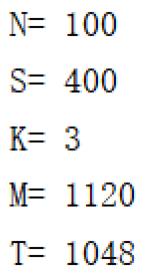
```
N= 5
S= 4
K= 3
M= 7
T= 5
```

OUTPUT 2:



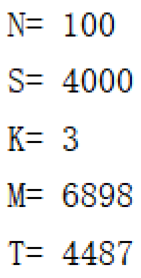
```
N= 100
S= 4
K= 3
M= 12
T= 12
```

OUTPUT 3:



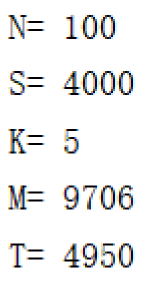
```
N= 100
S= 400
K= 3
M= 1120
T= 1048
```

OUTPUT 4:



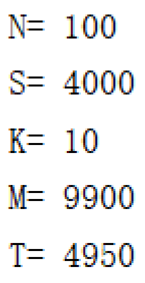
```
N= 100
S= 4000
K= 3
M= 6898
T= 4487
```

OUTPUT 5:



```
N= 100
S= 4000
K= 5
M= 9706
T= 4950
```

OUTPUT 6:



```
N= 100
S= 4000
K= 10
M= 9900
T= 4950
```

Appendix

source code:

```
import random
import itertools
import datetime
from matplotlib import pyplot as plt

def   removeDuplicatesN2 ( pair_list ):
    # remove duplicates --- O(N^2) space
    matrix_lists = [[0 for j in range(int(N))] for i in range(int(N))]    # initial matrix
    result_list = []
    # matrix with duplicates
    for tuple in pair_list:
        matrix_lists[tuple[0] - 1][tuple[1] - 1] = 1
    # remove duplicates in matrix
    for i in range(0, int(N)):
        for j in range(i + 1, int(N)):
            if matrix_lists[i][j] == 1 & matrix_lists[j][i] == 1:
                matrix_lists[i][j] = 0
    # convert matrix to tuple list
    for i in range(0, int(N)):
        for j in range(0, int(N)):
            if matrix_lists[i][j] == 1:
                result_list.append((i+1,j+1))
    return result_list
```

```
def removeDuplicatesM( pair_list ):
    temp = []
    result = []
    # convert tuple to list
    for tuple in pair_list:
        temp.append(list(tuple))
    # remove duplicates -- make dup item [0,0]
    for i in range(0,M):
        for j in range(i,M):
            if temp[i][0] == temp[j][1] and temp[i][1] == temp[j][0]:
                temp[j][0] = 0
                temp[j][1] = 0
    # remove duplicates items
    while [0,0] in temp:
        temp.remove([0,0])
    return temp

def generatePEwithM( new_pair):
    # generate E[] and P[] with O(M) space
    E = []
    P = []
    index = 0
    for num in range(0, int(N)):
        P.append(index)
        for eachtuple in new_pair:
            if eachtuple[0] == num + 1:
                E.insert(index, eachtuple[1])
                index = index + 1
            else:
                if eachtuple[1] == num + 1:
                    E.insert(index, eachtuple[0])
                    index = index + 1
    print("E[]= ",E)
    print("P[]= ",P)
    return 0

def generatePEwithN2( new_pair ):
    matrix_lists = [[0 for j in range(int(N))] for i in range(int(N))]    # initial matrix
    P = []
    E = []
    temp_num = 0 # work for P[]
    # generate matrix
    for tuple in new_pair:
        matrix_lists[tuple[0] - 1][tuple[1] - 1] = 1
    # print(matrix_lists)
    for i in range(0,int(N)):
        for j in range(0,int(N)):
            if matrix_lists[i][j] == 1:
                matrix_lists[j][i] = 1
    print(matrix_lists)
    for i in range(0, int(N)):
        P.insert(i,temp_num)
        for j in range(0,int(N)):
            if matrix_lists[i][j] == 1:
                E.insert(temp_num,j+1)
                temp_num = temp_num + 1
    print("E[]= ", E)
    print("P[]= ", P)
    return 0

# draw graphical histograms
def draw_hist(myList,Title,Xlabel,Ylabel,Xmin,Xmax,Ymin,Ymax):
    plt.hist(myList,100)
    plt.xlabel(Xlabel)
    plt.xlim(Xmin,Xmax)
    plt.ylabel(Ylabel)
    plt.ylim(Ymin,Ymax)
    plt.title(Title)
    plt.show()

class Attendees:
    def __init__(self,aid,attend):
        self.attend = attend
        self.aid = aid
    attend = []

if __name__ == '__main__':
    N = input("Number of sessions (N):")
    S = input("Number of attendees(S):")
    K = input("Number of sessions per attendees(K):")
    DIST = input("choose UNIFORM| TIERED| SKEWED| WZQ:")
    a_list = [] # list of attendees
    pair_list = [] # list of conflict with duplicates
    new_pair = [] # list of conflict with no duplicates
    graphical = [] # for draw the graphical histograms

    # generate distribution
    d_starttime = datetime.datetime.now()
```

```
uniform = [] # uniform distribution
skewed = [] # skewed distribution
two_tired = [] # two tired distribution
wzq = [0] # my own distribution
for k in range(0, int(N)+1):
    if k <= int(int(N)/2) :
        for i in range(1, k+1):
            wzq.append(k)
    else:
        for i in range(1, int(N)-k+2):
            wzq.append(k)
for k in range(0, int(N)+1):
    uniform.append(k)
    if k != 0:
        for i in range(k, int(N)+1):
            skewed.append(k)
        if k in range(1,int(float(N)*0.1)+1):
            for j in range(0,10):
                two_tired.append(k)
        else: two_tired.append(k)
    else:
        skewed.append(k)
        two_tired.append(k)

# print(two_tired)
# print(uniform)
# print(skewed)


for j in range(1, int(S)+1):
    session = random.sample(uniform, int(K))
    if DIST == "UNIFORM" and int(K) <= int(N):
        session = random.sample(uniform, int(K))
    else:
        if DIST == "SKEWED" and int(K) <= 0.1*int(N):
            session = random.sample(skewed, int(K))
        else:
            if DIST == "TIERED" and int(K) <= int(N):
                session = random.sample(two_tired, int(K))
            else:
                print("not a distribution")
                quit()
    a_list.append(Attendees(j, session))
d_endtime = datetime.datetime.now()
```

```
print("time for distributing: ",(d_endtime - d_starttime).microseconds)

# generate collision pairs with duplicates
for each in a_list:
        # for graphical histograms
        for i in range(0,int(K)):
                if each.attend[i] != 0:
                        graphical.append(each.attend[i])
        # generate pairs
        temp_pair_list = list(itertools.combinations(each.attend,2))
        # test: print(temp_pair_list)
        i = 0    # index of pair_list
        for item in temp_pair_list:
                if min(item) != 0:
                        pair_list.insert(i, item)
                        i = i + 1
# draw graphical histograms to show the actually distribution
draw_hist(graphical, 'Number Attendees of each Session', 'Session',
                    'Attendee.No', 0, int(N), 0, int(S))

starttime = datetime.datetime.now() # calculate time of remove dup
# remove 0-> do not attend
pair_list = set(pair_list)
print(pair_list)
M = pair_list.__len__()
print("M=", M)

# remove duplicates like
new_pair = removeDuplicatesM(pair_list)
new_pair = RemoveDuplicatesN2(pair_list)
T = new_pair.__len__()
print(new_pair)
print("T=", T)
endtime = datetime.datetime.now()
print("Time of removing duplicates: ",(endtime - starttime).microseconds)

generatePEwithM( new_pair )
generatePEwithN2(new_pair)
```

Project Part #2

Computing environment

Operating System: Windows 10, 64bits
Manufacturer: LENOVO
Processor: Intel® Core™ i5-6200U @ 2.30GHz~2.40GHz
Memory: 8076MB RAM
Platform: PyCharm 2018.2.1 (Community Edition)

Creating three different ordering methods

All the following methods will use one part 1 OUTPUT as an example. N (number of sessions) = 5, S (number of attendees) = 4, K (number of sessions per attendee) = 3 and DIST (distribution) = 'UNIFORM'. The P and E are:
P= [0, 4, 7, 10, 13, 16]
E= [3, 4, 2, 5, 5, 1, 3, 1, 2, 4, 5, 1, 3, 4, 2, 1]

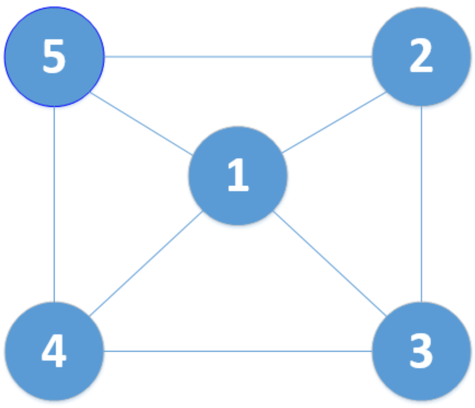


Figure 1. Graph conducted with P and E

I also build a global *DoubleLinkedList(DDL)* class. It contains the class *Node* with pre, data and next. Also has class build in functions to help me do further work:
(1) __len__(): get the doubly linked list length, return an int number.
(2) append(): append node.
(3) get(index): get the node object with index of the doubly linked list.
(4) set(index): change the node data with index.
(5) insert(index, data): insert node with index and data.
(6) delete(index): delete node with index.

Method 1 --- Smallest Last Vertex Ordering

Data Structure:
The first ordering method is smallest last vertex, which means that assign a color value to the vertex who has the smallest degree last. In other words, firstly arranging the most "effectively" vertex (effectively means have most effect to adjacent vertices).

To achieve this, I create the following class for the values of variables in various fields of the data node for each vertex to be used to save storage space.

	name	degree	valid	color
Vertex(i)	Session number	Current degree	Default = 1 -1 when delete	Color value

For each vertex class, each field means different:

a) Name: session number, also the vertex number.

b) Degree: store the degree of this vertex, when later during the recursively deleting process, if this vertex is the deleted vertex adjacent vertex, its degree will minus one and this field will be updated.

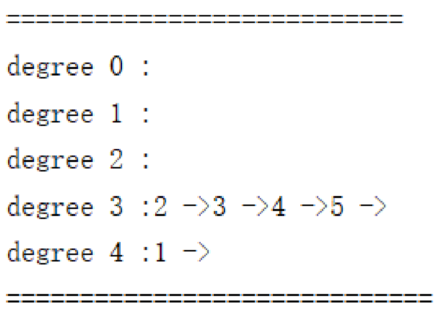
c) Valid: if the vertex still on the degree list, it is 1. Changing this field to –1 when this vertex was deleted.

d) Color value: represent color with number. Default was 0 (no color).

Functions:

1. *genDDL (E, P)*

This function is used to generate a degree list storing doubly linked list. The index of this list means the degree as well. And the doubly linked list in each degree will store the vertices of this degree. The data comes from part #1 OUTPUT, the adjacent list E and pointer to it P. The following is the example's initialized degree list.



```
===========================
degree 0 :
degree 1 :
degree 2 :
degree 3 :2 ->3 ->4 ->5 ->
degree 4 :1 ->
=============================
```

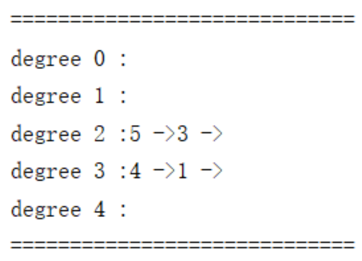
Figure 2.1. Initialized degree list

2. *getFirstVertex() & removeVertex()*

I create the function *getFirstVertex* to get the vertex number of smallest degree, which is the first element on the degree list. Then I use the function *removeVertex()* to remove the vertex with the smallest degree and updating the adjacent vertices' degree to current degree minus one. Also adding the vertex deleted to the ordered list *deletedorder*[] of vertices deleted and updating the vertex attribute valid to -1.

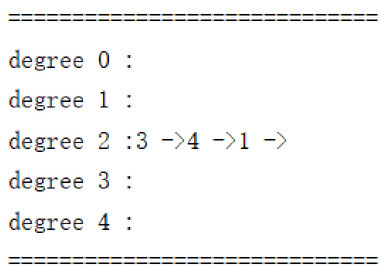
The following is the changes of degree list after continuously deleting the smallest degree vertex.

In figure1, the smallest degree is 3 and the first vertex is 2. After deleting vertex 2, its adjacent vertices 5, 1 and 3's degree minus 1. After moving the vertex, the degree list shown in figure 2.2. And figure 2.3- figure 2.6 shows the continuing deleting process in a recursive manner.



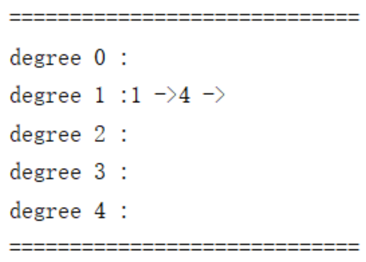
```
=============================
degree 0 :
degree 1 :
degree 2 :5 ->3 ->
degree 3 :4 ->1 ->
degree 4 :
=============================
```

Figure 2.2. Degree list after deleting vertex 2



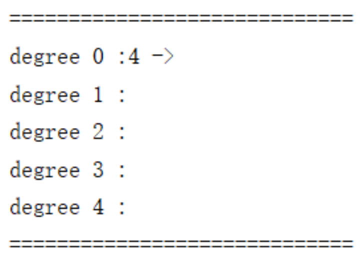
```
=============================
degree 0 :
degree 1 :
degree 2 :3 ->4 ->1 ->
degree 3 :
degree 4 :
=============================
```

Figure 2.3. Degree list after deleting vertex 5



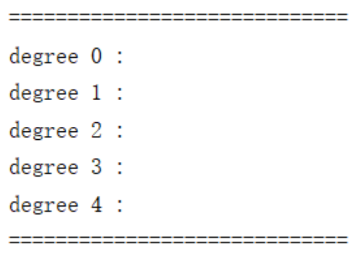
```
=============================
degree 0 :
degree 1 :1 ->4 ->
degree 2 :
degree 3 :
degree 4 :
=============================
```

Figure 2.4. Degree list after deleting vertex 3



```
=============================
degree 0 :4 ->
degree 1 :
degree 2 :
degree 3 :
degree 4 :
=============================
```

Figure 2.5. Degree list after deleting vertex 1

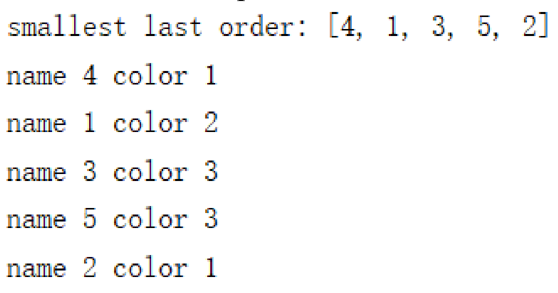


```
=============================
degree 0 :
degree 1 :
degree 2 :
degree 3 :
degree 4 :
=============================
```

Figure 2.6. Degree list after deleting vertex 4

3. *smallestcolor()*
The function *smallestcolor()* is the coloring procedure. After all vertices have been deleted, this function scan and assign colors values (session periods) to each vertex in

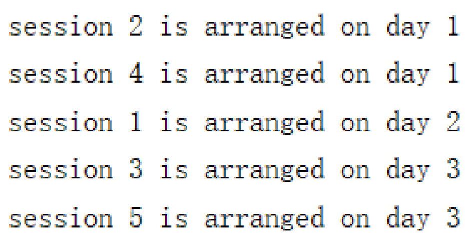
an order opposite to the order deleted, assigning for a “color” the smallest non-negative integer not previously assigned to any adjacent vertex. To achieve this goal, for each vertex in *deletedorder*, I traverse adjacent list to check the adjacent vertices’ color. If the vertex color value is 0, it means this vertex has no colored. So assign color 1 to it. If the vertex color value is not 0, traverse all its adjacent vertices and find the minimum non-appeared and different color value. The color values begin with 1 and increase 1 at each time.
The coloring process result of the example shows as the following:



```
smallest last order: [4, 1, 3, 5, 2]
name 4 color 1
name 1 color 2
name 3 color 3
name 5 color 3
name 2 color 1
```

Figure 2.7. The smallest last order and each session name with its color

4.*Print further output*
Because the vertex is the session number and the color is the date number. So the session arrangement result shows the following with the summary information for the schedule.



```
session 2 is arranged on day 1
session 4 is arranged on day 1
session 1 is arranged on day 2
session 3 is arranged on day 3
session 5 is arranged on day 3
```

Figure 2.8. The session arrangement result

For additional output information with METHOD 1 is that (using the example above):
i A graph indicating the degree when deleted on the vertical axes and the order colored on the x-axis.

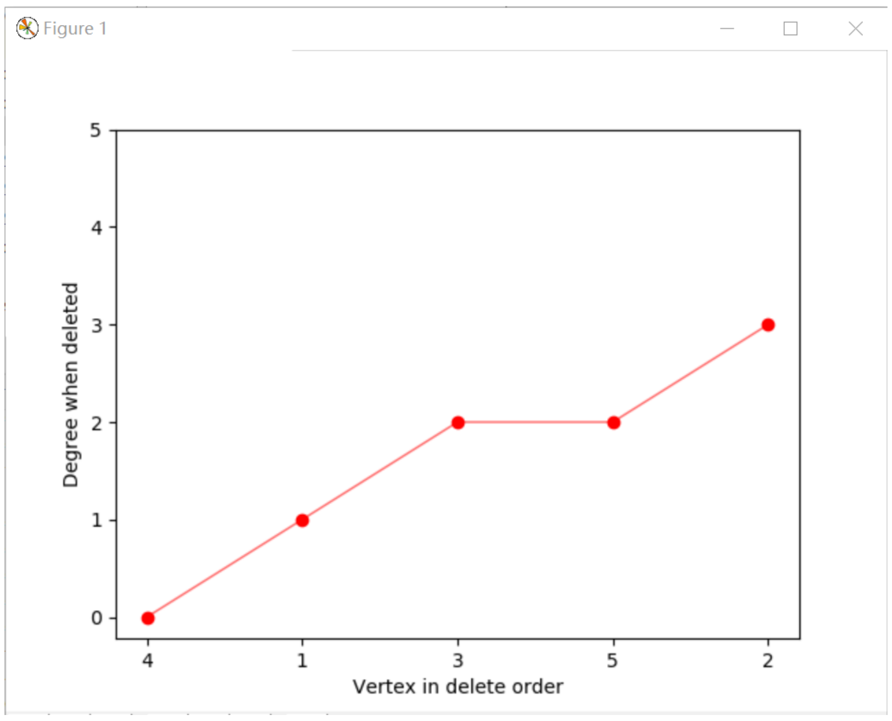


Figure 2.9. The graph of vertex and degree when deleted

ii The maximum degree when deleted. As the example shown above, the maximum degree is on the following screen shot:

```
maximum degree when deleted: 3
```

Method 2 --- Largest Last Vertex Ordering

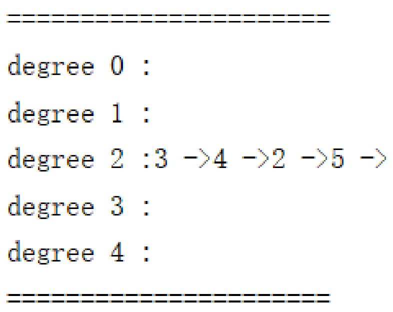
For method 2, I choose the largest last vertex ordering, which means coloring the vertex who has the largest degree lastly. This method is different with the method 1 on the sequence of deleting the vertex.

Data Structure
it is the same with the method 1 – the *Vertex* class with four attributes.

Functions:
1. *genDDL (E, P)*
This function is used to generate a degree list storing different doubly linked list. The index of this list means the degree as well. And the doubly linked list in each degree will store the vertices which have this degree. The data comes from part #1 OUTPUT, the adjacent list E and pointer to it P. Fugure 2.1 shows the example's initialized degree list.

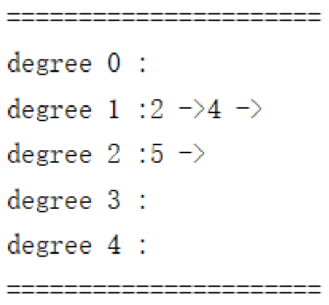
2. *getLastVertex() & removeVertex()*
I implement the function *getLastVertex()* to get the vertex number of largest degree, which is the first element on the largest degree doubly linked list. Then I use the function *removeVertex()* to remove the vertex with the smallest degree and updating the

adjacent vertices' degree to current degree minus one. Also adding the vertex deleted to the ordered list *deletedorder*[] of vertices deleted and updating the vertex attribute valid to -1. The *removeVertex()* function is the same with method1.
The following is the changes of degree list after continuously deleting the largest degree vertex for the example listed at the first of report.
In figure 2.1, the initial degree list, the largest degree is 4 and the first vertex is 1. After deleting vertex 1, its adjacent vertices 2, 3, 4 and 5's degree minus 1. After moving the vertex, the degree list shown in figure 3.1. And figure 3.2- figure 3.5 shows the continuing deleting process in a recursive manner.



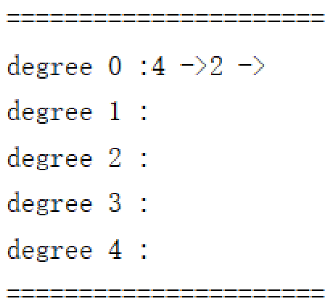
```
======================
degree 0 :
degree 1 :
degree 2 :3 ->4 ->2 ->5 ->
degree 3 :
degree 4 :
======================
```

Figure 3.1. Degree list after deleting vertex 1



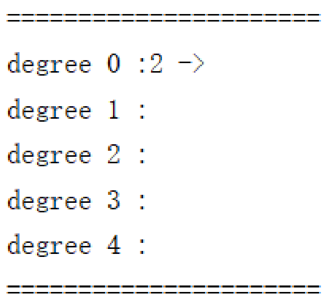
```
======================
degree 0 :
degree 1 :2 ->4 ->
degree 2 :5 ->
degree 3 :
degree 4 :
======================
```

Figure 3.2. Degree list after deleting vertex 3



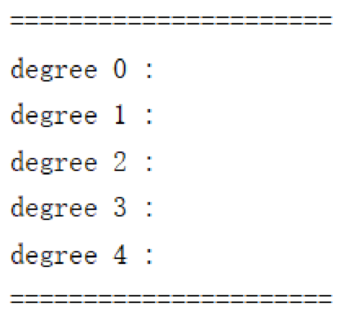
```
======================
degree 0 :4 ->2 ->
degree 1 :
degree 2 :
degree 3 :
degree 4 :
======================
```

Figure 3.3. Degree list after deleting vertex 5



```
======================
degree 0 :2 ->
degree 1 :
degree 2 :
degree 3 :
degree 4 :
======================
```

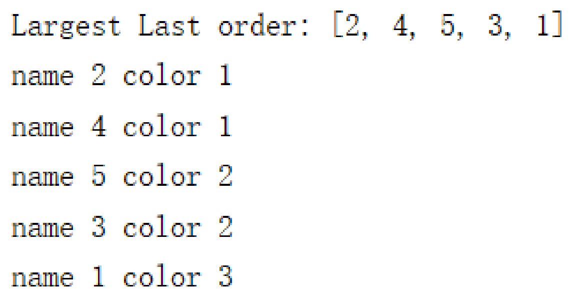
Figure 3.4. Degree list after deleting vertex 4



```
======================
degree 0 :
degree 1 :
degree 2 :
degree 3 :
degree 4 :
======================
```

Figure 3.5. Degree list after deleting vertex 2

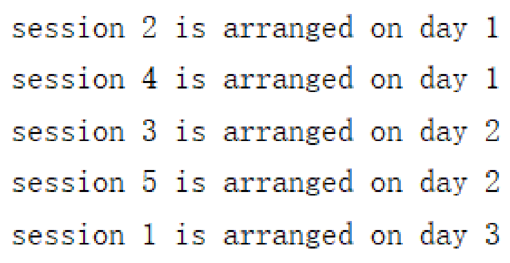
3. *largestcolor()*
The function *largestcolor()* is the coloring procedure. After all vertices have been deleted, this function scan and assign colors values (session periods) to each vertex in an order opposite to the order deleted, assigning for a "color" the smallest non-negative integer not previously assigned to any adjacent vertex. To achieve this goal, for each vertex in *deletedorder*, I traverse adjacent list to check the adjacent vertices' color. If the vertex color value is 0, it means this vertex has no colored. So assign color 1 to it. If the vertex color value is not 0, traverse all its adjacent vertices and find the minimum non-appeared and different color value. The color values begin with 1 and increase 1 at each time.
The coloring process result of the example shows as the following:



```
Largest Last order: [2, 4, 5, 3, 1]
name 2 color 1
name 4 color 1
name 5 color 2
name 3 color 2
name 1 color 3
```

Figure 3.6. The largest last order and each session name with its color

4.*Print further output*
Because the vertex is the session number and the color is the date number. So the session arrangement result shows the following with the summary information for the schedule.



```
session 2 is arranged on day 1
session 4 is arranged on day 1
session 3 is arranged on day 2
session 5 is arranged on day 2
session 1 is arranged on day 3
```

Figure 3.7. The session arrangement result

For additional output information with METHOD 2 is that (using the example above):
i A graph indicating the degree when deleted on the vertical axes and the order colored on the x-axis.

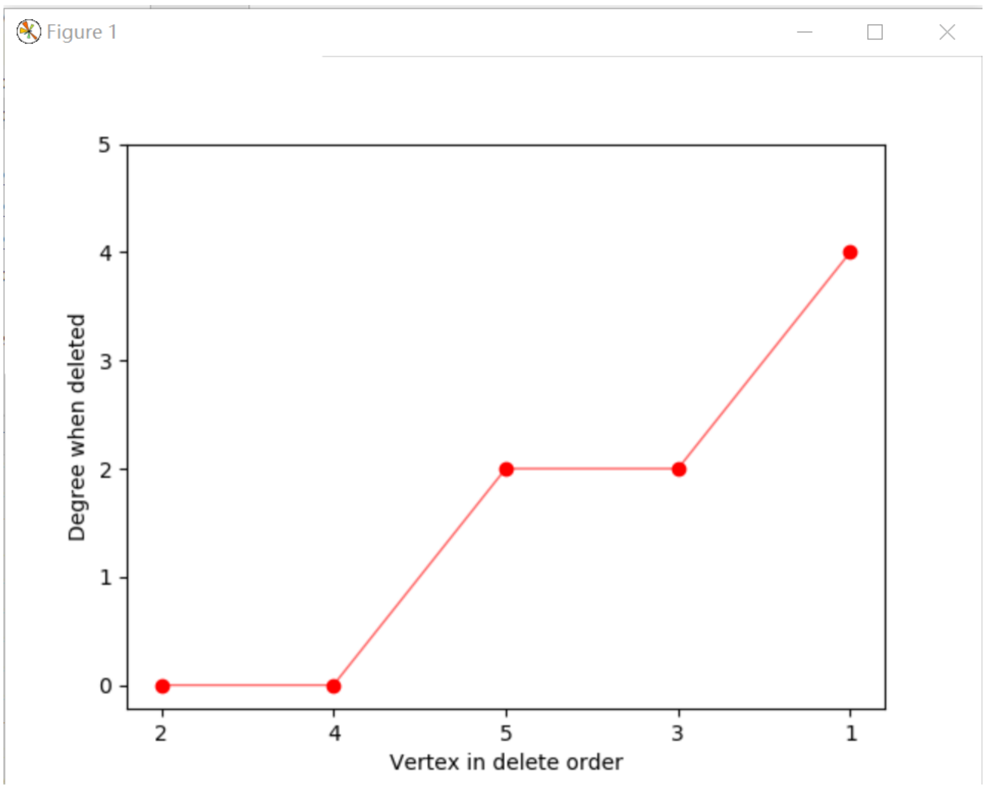


Figure 3.8. The graph of vertex and degree when deleted

ii The maximum degree when deleted. As the example shown above, the maximum degree is on the following screen shot. This number is also the length of degree list minus one. I collect this data here to compare it with method 1 on the latter section.

```
maximum degree when deleted: 4
```

Method 3 --- Random Vertex Ordering

For this ordering method, I choose the random vertex ordering. That's to say that generate a random sessions number list and coloring the vertex as the list number sequence.

Data Structure

it is the same with the method 1 – the *Vertex* class with four attributes.

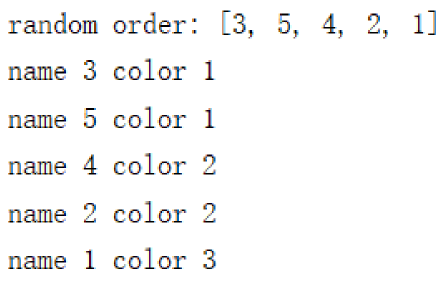
Functions:

1. *genDDL (E, P)*

This function is used to generate a degree list storing different doubly linked list. The index of this list means the degree as well. And the doubly linked list in each degree will store the vertices which have this degree. The data comes from part #1 OUTPUT, the adjacent list E and pointer to it P. Fugure 2.1 shows the example's initialized degree list.

2. *dllprint()*
This function is to print the doubly linked degree list in a readable manner. All the figures above about degree list are using this function to generate. To implement this, it’s just traversing all doubly linked list items and printing all nodes in lists.

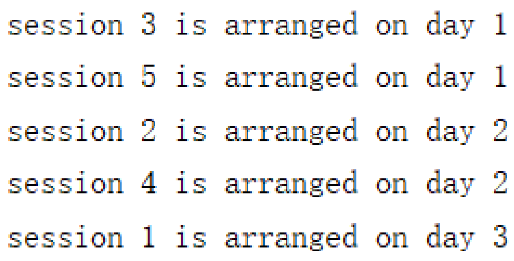
3. *color()*
The function *color()* is the coloring procedure. The parament of this function is a list – *deletedorder* - generating with the build in *random.sample()* function. To assign color values for all the vertices with no conflicts, for each vertex in *deletedorder*, I traverse adjacent list to check the adjacent vertices’ color. If the vertex color value is 0, it means this vertex has no colored. So assign color 1 to it. If the vertex color value is not 0, traverse all its adjacent vertices and find the minimum non-appeared and different color value. The color values begin with 1 and increase 1 at each time.
The coloring process result of the example shows as the following:



```
random order: [3, 5, 4, 2, 1]
name 3 color 1
name 5 color 1
name 4 color 2
name 2 color 2
name 1 color 3
```

Figure 4.1. The largest last order and each session name with its color

4.*Print further output*
Because the vertex is the session number and the color is the date number. So the session arrangement result shows the following with the summary information for the schedule.



```
session 3 is arranged on day 1
session 5 is arranged on day 1
session 2 is arranged on day 2
session 4 is arranged on day 2
session 1 is arranged on day 3
```

Figure 4.2. The session arrangement result

For additional output information with METHOD 3 is that (using the example above): A graph indicating the degree when deleted on the vertical axes and the order colored on the x-axis.

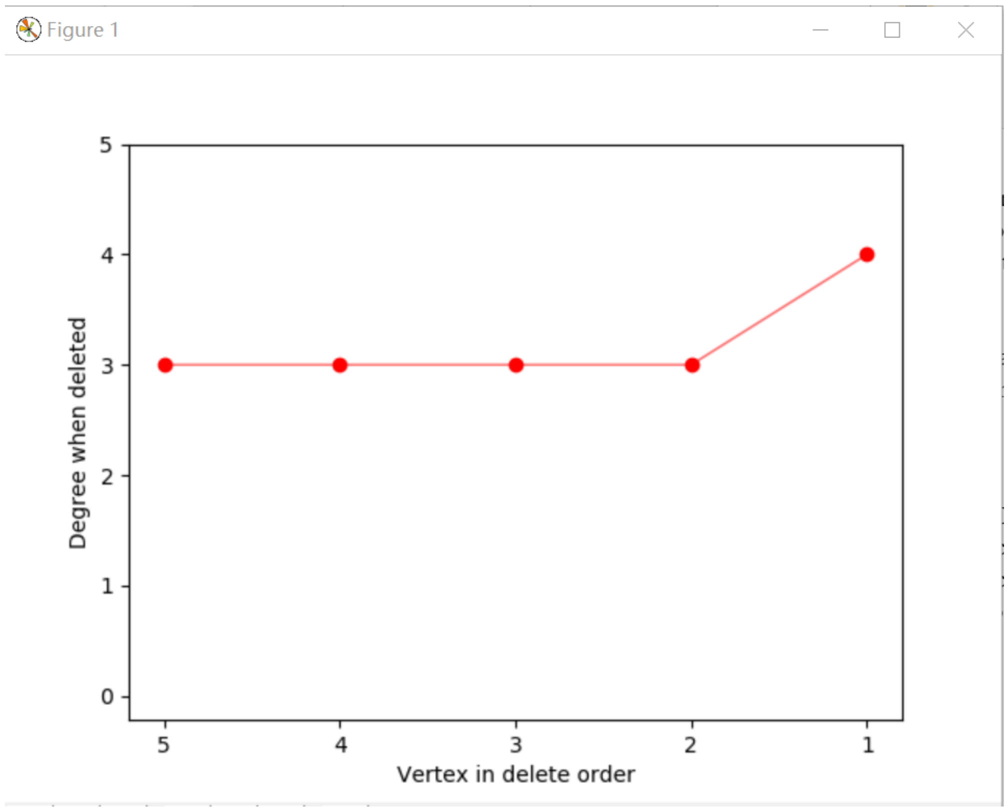


Figure 4.3. The graph of vertex and degree when deleted

Different Ordering Methodologies Compare

In this section, I will compare three methods above in different aspects – asymptotic running time, asymptotic space requirement and total number of colors needed.

Asymptotic running time

For the asymptotic running time, I will analysis from code respect. And I will give some simple data about the execution time of different method's generating *deletedorder* and coloring process.

Method 1 – O(n)

The detail asymptotic running time analysis:

a) *genDDL(E, P)*: O(n). Because doubly linked list don't need to traverse to insert and delete node, it is only linear time to build a degree list of doubly linked list.

b) *getFirstVertex()*: O(n). Only one traversal of degree list.

c) *removeVertex()*: O(n). After deleting the first vertex in O(1), the first step of *removeVertex* is to find adjacent vertices in adjacent list. With P[], the pointer of adjacent list, you can find adjacent vertices in E[] in O(1). All you need to do is to search doubly linked list on specific degree once to find adjacent vertex to change the position of them. This will take O(n).

d) *smallestcolor()*: O(n). One loop with color values (0-N, because the worst situation is different session have different color, and total color number is N) is O(N). And one loop with adjacent list to check the adjacent vertices color value with O(1).

So the final asymptotic time is O(n)

Method 2 - O(n)

The detail asymptotic running time analysis:

a) *genDDL(E, P)*: O(n). Because doubly linked list don't need to traverse to insert and delete node, it is only linear time to build a degree list of doubly linked list.
b) *getLastVertex()*: O(n). Only one traversal of degree list.
e) *removeVertex()*: O(n). After deleting the last vertex in O(1), the first step of *removeVertex* is to find adjacent vertices in adjacent list. With P[], the pointer of adjacent list, you can find adjacent vertices in E[] in O(1). All you need to do is to search doubly linked list on specific degree once to find adjacent vertex to change the position of them. This will take O(n).
c) *largestcolor()*: O(n) One loop with color values (0-N, because the worst situation is different session have different color, and total color number is N) and find adjacent vertices in adjacent list with O(N-1) (N is the number of session).

So the final asymptotic time is O(n)

Method 3 - O(n)

The details of analysis is showing below:

a) *generate order list*: O(n). generating a list from 1 to N, and using the build in random function to get the random deleting order.
b) *color()*: O(n) One nested loop with color values (0-N, because the worst situation is different session have different color, and total color number is N) and adjacent list.

The final asymptotic time is O(n)

The following is the table of different method generating delete vertex order list and coloring process execution time compare.

N	S	K	DIST	METHOD1	METHOD2	METHOD3
10	5	5	UNIFORM	3990	2990	997
20	5	5		11968	2991	996
20	10	5		1995	1995	999
20	20	5		1993	2991	996
20	40	5		2990	2032	997
100	40	5		17952	16956	3991
100	80	5		15957	28923	4987
100	80	10		52859	43883	5983

Through the table, within the 2 lines in blue background, the N become 2 times bigger, the execution time with all methods became about 2 times bigger. So the result confirm my analysis of asymptotic time bound.

Although I have to admit that there are some violations exist. The data set is too small and the asymptotic analysis may not correct. This part needs further analysis with bigger data set.

Asymptotic space requirement

Method 1-O(N^2)
I use a list of Vertex class, which the space is N*4 (N is the session/vertex number, 4 is that every Vertex class has 4 attributes). About the degree doubly linked list, the maximum space is N*N (first N for the degree, the degree scope is 0-N; the second N for the length of doubly linked list, the maximum of one doubly linked list on specific degree is N - the situation that all vertex has the same degree)
The final asymptotic space requirement is O($N*4+N^2$) = O(N^2)

Method 2-O(N^2)
The same with method 1, using a list of Vertex class and a degree doubly linked list.
The final asymptotic space requirement is O($N*4+N^2$) = O(N^2)

Method 3-O(N)
Only using a Vertex class list and also a *deleteorder* list.
The final asymptotic space requirement is O(N*4+N) = O(N)

Total number of colors needed

I count the total color values, which is also the day to hold the sessions. And the result shows on the following table.

N	S	K	DIST	METHOD1	METHOD2	METHOD3
10	5	5	UNIFORM	6	6	6
20	5	5		5	6	6
20	10	5		6	7	7
20	20	5		10	17	17
20	40	5		12	11	12
100	40	5		6	9	8
100	80	5		10	11	11
100	80	10		23	26	26

Under different situation with different N, S and K, method 1 is the method that has the minimum total number of colors needed. Method 2 and method 3 almost the same, has more colors than method 1. In this aspect, I can say that method 1 is better than others. For the customers who need to hold all sessions in limited date, using method 1 to assign all the sessions is a better choice.

Max degree when deleted
The max degree when deleted is the supplement of the method 1.

N	S	K	DIST	METHOD1	METHOD2
10	5	5	UNIFORM	5	9
20	5	5		4	8
20	10	5		6	15
20	20	5		10	11
20	40	5		14	19
100	40	5		6	19
100	80	5		10	29
100	80	10		36	75

Comparing with method 2, the max degree when deleted of method 1 is almost half of method 2. With the deleting process continuing, all the vertex gather in the first half of degree doubly linked list. This will make the algorithm more faster.

Appendix

Method 1:

Dependencies:

```
import Part1
import DoubleLinkedList
import random
import itertools
import matplotlib.pyplot as plt
import datetime
```

Vertex data structure:

```
class Vertex():
    def __init__(self, name,degree,index):
        self.name = name    #session number
        self.degree = degree #current degree
    valid = 1    # -1 when deleted
    color = 0    #color value
```

Functions:

```
def genDDL(E, P): #initialize dll and Vertex list
    ddlist = [] # list of degree double linked lists
    p = P.__len__()
    # initialize double linked list
    for i in range(0,p-1):
        ddlist.append(DoubleLinkedList.DDL())
    # generate double linked list
    for i in range(0,p-1):
        indexDDL = P[i+1] - P[i] # vertex i+1 's degree
        ddlist[indexDDL].append(i + 1)
        # to create Vertex and its field(degree and name)
        vertIndex = ddlist[indexDDL].__len__()
        vertexlist.append(Vertex(i+1,indexDDL,vertIndex-1))
    return ddlist

def dllprint(ddlist):
    p = ddlist.__len__()
    # print double linked list
    for i in range(0, p):
        length = ddlist[i].__len__()
        print("degree", i, ":",end='')
        for j in range(0, length):
            print(ddlist[i].get(j).__dict__['data'], '->', end="")
        print("\r")
    return 0

def getFirstVertex(ddlist):
    p = ddlist.__len__()
    for i in range(0,p-1):
        if(ddlist[i].__len__()!=0):
            vertex = ddlist[i].get(0)
            break
    return vertex.__dict__['data']

def removeVertex(vlist,vertex):
    vlist[vertex-1].valid = -1 #indicate it was delete
    degree = vlist[vertex-1].degree
    for i in range(0,ddlist[degree].__len__()):
        if ddlist[degree].get(i).__dict__['data'] == vertex:
            SLFdeleteOrder.append(vertex)
            ddlist[degree].delete(i)
            break
    for i in range(P[vertex-1],P[vertex]):    # E[i] is the vertex adjacent to deleted
vertex
        #find the vertex E[i] than degree-1 and remove it
        dlly = vlist[E[i]-1].degree # y axis of dll(horizon axis)
        for dllx in range(0,ddlist[dlly].__len__()):
            if ddlist[dlly].get(dllx).__dict__['data'] == E[i]:
                ddlist[dlly].delete(dllx)
                dlly = dlly - 1
                vlist[E[i]-1].degree = dlly
                ddlist[dlly].append(E[i])
                break
    dllprint(ddlist)

def smallestcolor(sequence):
    vertexlist[sequence[0]-1].color = 1 # give the first vertex color-1
    print("name", vertexlist[sequence[0] - 1].name, "color", vertexlist[sequence[0]
- 1].color)
    colorlist = [] # store the color options
    for color in range(1,P.__len__()):
        colorlist.append(color)
    for v in range(1, sequence.__len__()):   # sequence[v] is the name of color vetex
        adjcVcolor = []
        for i in range(P[sequence[v] - 1], P[sequence[v]]):
            adjcVcolor.append(vertexlist[E[i]-1].color)
        list3 = list(set(colorlist) - set(adjcVcolor))
        vertexlist[sequence[v]-1].color = min(list3)
        print("name",vertexlist[sequence[v]-1].name,
            "color",vertexlist[sequence[v]-1].color)
```

Main function:

```
if __name__ == '__main__':
    # the first part is get P[] E[] from Part 1. It is repeatable, so I omit this part.
    # I use Part1.removeDuplicatesM(pair_list,M) to remove duplicates
    # I use Part1.generatePEwithM( new_pair,N ) to generate P[] E[]

    # initialize vertex data structure
    vertexlist=[]
    SLFdeleteOrder=[] #small least first order
    p = P.__len__()
    #generate doubled linked list
    ddlist1 = ddlist = genDDL(E, P)
    dllprint(ddlist)

    starttime = datetime.datetime.now()
    #delete vertex in smallest first order
    for k in range(0,P.__len__()-1):
        deletedvertex = getFirstVertex(ddlist)
        removeVertex(vertexlist,deletedvertex)
    for g in range(0,P.__len__()-1):
        print("name:",vertexlist[g].name,"    ","degree when
delete:",vertexlist[g].degree)
    print("+++++++++++++++++")
    SLFdeleteOrder.reverse() # change it to color sequence
    print("smallest last order:",SLFdeleteOrder)
    smallestcolor(SLFdeleteOrder)
    endtime = datetime.datetime.now()
    print("Time of method1: ", (endtime - starttime).microseconds)

    # draw the graph that degree when deleted on the vertical axes and the order
colored on the x-axis
    yaxis = [] # xaxis is SLFdeleteOrder
    for xaxis in SLFdeleteOrder:
        yaxis.append(vertexlist[xaxis-1].degree)
    print('x',SLFdeleteOrder)
    print('y',yaxis)
    ax = plt.gca()
    ax.plot(yaxis, color='r', linewidth=1, alpha=0.6)
    plt.plot(yaxis, 'ro')
    plt.ylim(top=SLFdeleteOrder.__len__(), bottom=-0.22) # set y-axis height
    plt.xticks(range(0,SLFdeleteOrder.__len__()), SLFdeleteOrder, rotation=0)
#change x-axis number
    plt.xlabel("Vertex in delete order")
    plt.ylabel("Degree when deleted")
    plt.show()

    print("maximum degree when deleted:", max(yaxis))
    print("-----------------------")
    maxcolor = 0
    # print summary information for the schedule
    for day in range(1, vertexlist.__len__() + 1):
        for session in vertexlist:
            if session.color == day:
                print("session",session.name,"is arranged on day",day)
                maxcolor = day
    print("maximum color number is:",maxcolor)
```

Method 2:

The only different about the functions and data structure is that during the deleting process, method1 get the vertex with smallest degree first, method 2 get the vertex with largest degree first. So I only post the different part down here.

```
def getLastVertex(ddlist):
    p = ddlist.__len__()
    global vertex
    for i in range(p-1,-1,-1):
        if (ddlist[i].__len__() != 0):
            vertex = ddlist[i].get(0)
            break
    return vertex.__dict__['data']
```

Method 3:

This method is the simplest among three method. I only post the different part below.

```
# class Vertex()
# def genDDL(E, P)
# dllprint(ddlist)
# def color(sequence) is the same with smallestcolor(sequence)
if __name__ == '__main__':
    vertexlist = []
    RMdeleteOrder = []    # random order
    p = P.__len__()
    # generate doubled linked list
    ddlist = genDDL(E, P)
    dllprint(ddlist)
    temp = []
    for i in range(1, P.__len__()):
        temp.append(i)
    RMdeleteOrder = random.sample(temp, P.__len__() - 1)
    print(RMdeleteOrder)
    for g in range(0, P.__len__() - 1):
        print("name:", vertexlist[g].name, "    ", "degree when delete:",
vertexlist[g].degree)
    print("random order:", RMdeleteOrder)
    color(RMdeleteOrder)

    # draw the graph that degree when deleted on the vertical axes and the order
colored on the x-axis
    yaxis = []    # xaxis is SLFdeleteOrder
    for xaxis in RMdeleteOrder:
        yaxis.append(vertexlist[xaxis - 1].degree)
    print('x', RMdeleteOrder)
    print('y', yaxis)
    ax = plt.gca()
    ax.plot(yaxis, color='r', linewidth=1, alpha=0.6)
    plt.plot(yaxis, 'ro')
    plt.ylim(top=RMdeleteOrder.__len__(), bottom=-0.22)    # set y-axis height
    plt.xticks(range(0, RMdeleteOrder.__len__()), RMdeleteOrder, rotation=0)    #
change x-axis number
    plt.xlabel("Vertex in delete order")
    plt.ylabel("Degree when deleted")
    plt.show()
```

```
print("maximum degree when deleted:", max(yaxis))
print("-----------------------")

# print summary information for the schedule
for day in range(1, vertexlist.__len__() + 1):
    for session in vertexlist:
        if session.color == day:
            print("session", session.name, "is arranged on day", day)
            maxcolor = day
print("maximum color number is:", maxcolor)
```