Name: _____ Section: 12:30pm or 2pm

Sample Midterm B

Learning Objective:

- Create Python code to automate a given task.

Instructions:

The midterm tests your mastery of skills taught in Weeks 1-5, which culminates in creating simulation models using Python and algorithmic thinking. There are three questions, worth a total of 24 points. The exam is 80 minutes, and is open notes but closed computer. You can bring paper notes or books of any kind, but no computers, tablets, or cell phones are allowed. Before the exam is graded, do not share the questions or your solutions with anyone else, including students of other sections. When you turn in the exam, you must also hand in all scrap paper that you wrote on. Do not use a cell phone, tablet or computer in the classroom before all of the exams are handed in. **Any violation of academic integrity will result in a zero grade for the midterm for everyone involved.**

As long as you fulfill all the specifications described in the problem description, it doesn't matter how you solve the problem or how efficient is your code. However, if you cannot solve a problem, you may get partial credits for submitting whatever you have, including any parts of the four steps of algorithmic thinking.

Q1. Election Simulator (7 points)

This question asks you to write code to simulate the outcome of an election based on a simple probabilistic model estimated from polls data. Assume that there are two political parties, which we refer to as the Democrats and the Republicans. There is a number of regions participating in the election, and each region has a certain number of delegates that can be awarded to one of the two parties. Within each region, the party receiving the most votes will be awarded all of the delegates from that region. Assume also that the number of votes each party receives within each region is drawn from a normal distribution with a given mean and standard deviation.

For example, consider the following sample data with four regions. The first two columns denote the mean and standard deviation of the number of votes for Democrats in each region. The next two columns denote the same information for the Republicans. The final column states the number of delegates in each region. A likely outcome of the election is that the Democrats win the first two regions, with a total delegate count of 25, while the republicans win the other two regions, with a total delegate count of 23. Of course, there are other possible outcomes, and the likelihood of each outcome depends on the given means and standard deviations.

	Dem-Mean	Dem-Std	Rep-Mean	Rep-Std	Delegates
Region A	10	2	8	1.5	15
Region B	8	1.5	6	1	10
Region C	6	1	8	2	13
Region D	5	1	8	1	10

Although random samples from a normal distribution will be fractional, you should NOT round the random samples in any way in your simualtion, but you should leave them as fractional numbers with many decimal places. (To interpret this, think of the means and standard deviations in the above table in terms of millions of people.) In the extremely unlikely scenario in which there is an exact tie in the number of votes to all decimal places, you can decide the winner of the region arbitrarily, and it does not matter what you decide. Moreover, ignore the fact that the normal samples can be negative. You should not do anything to alter the samples in this case.

Write a function named `election` with two input arguments:

- `df`: a DataFrame encoding a table of the above format. See the test code below for examples. You may assume that the column names are exactly as in the test code.
- `T`: a positive integer denoting the number of simulations to run.

The function should return two lists, each containing T numbers. In the first list, each number corresponds to the number of delegates won by the Democrats in each of the T simulations. The second list is analogous and corresponds to the Republicans. Note that the first number in each list corresponds to the first simulation, so the sum of those numbers should be equal to the total delegate count in all regions, which is 48 in the above example. The same statement holds for the second number of each list, and so on. See the sample inputs and outputs for examples.

```
[2]: # Example of an input DataFrame
     import pandas as pd
     df=pd.DataFrame([[10,2,8,1.5,15],\
                      [8, 1.5,6,1,10],\
                      [6, 1,  8,2,13],\
                      [5, 1,  8,1,10]],\
                     index=['Region A','Region B','Region C','Region D'],\
                     columns=['Dem-Mean','Dem-Std','Rep-Mean','Rep-Std','Delegates'])
     df
```

```
          Dem-Mean  Dem-Std  Rep-Mean  Rep-Std  Delegates
Region A        10      2.0         8      1.5         15
Region B         8      1.5         6      1.0         10
Region C         6      1.0         8      2.0         13
Region D         5      1.0         8      1.0         10
```

[3]:
```
# Sample run 1
dem,rep=election(df,10)
print('Democrat   # of Delegates:',dem)
print('Republican # of Delegates:',rep)
```

```
Democrat   # of Delegates: [25, 38, 25, 15, 25, 25, 10, 15, 25, 25]
Republican # of Delegates: [23, 10, 23, 33, 23, 23, 38, 33, 23, 23]
```

Write your final code below:

Q2. Waiting Time Simulator (8 points)

This question asks you to create a tool to estimate the average time customers wait outside a store before being allowed to enter. The store has set a limit on the maximum number of customers that can be inside the store at any given time. Once this limit is reached, customers must queue outside the store until some customers exit, at which point the store will admit as many customers as possible while observing the maximum occupancy limit. You are to simulate the queueing dynamics and use a certain mathematical formula called Little's Law to estimate the average waiting time.

Write a function called `waiting_time` with three input arguments:

- `arrivalsList`: a list of non-negative integers corresponding to the number of new customers lining up to enter the store in each minute. For simplicity, assume that all of the arriving customers show up exactly at the beginning of the minute.
- `n`: the maximum occupancy limit. This is the maximum number of people allowed inside the store at any given time.
- `k`: the number of minutes each customer stays inside the store after being admitted. For simplicity, assume that every customer stays the same amount of time. If k=2, then every customer who is admitted is assumed to be inside the store for exactly 2 minutes before exiting.

The function should return the average waiting time customers spend outside the store before being admitted, rounded to two decimal places. For example, suppose that `arrivalsList=[5,9,14,5,3,0,9,20,30,0,0]`, $n = 15$ and $k = 2$, the queuing dynamics can be described by the following table. Note that the number of people who exit is exactly the same as the number of people admitted k minutes ago. Moreover, the occupancy changes from minute to minute based on the difference between the number of admits and the number of exits. On the other hand, the queue changes from minute to minute based on the difference between the number of arrivals and the number of admits.

Minute	Arrivals	Exit	Admitted	Occupancy	Queue
0	5	0	5	5	0
1	9	0	9	14	0
2	14	5	6	15	8
3	5	9	9	15	4
4	3	6	6	15	1
5	0	9	1	7	0
6	9	6	9	10	0
7	20	1	6	15	14
8	30	9	9	15	35
9	0	6	6	15	29
10	0	9	9	15	20
Total	**95**				**111**

$$\text{Av. Wait Time} = \frac{\text{Total Queue}}{\text{Total Arrivals}} \times 1 \text{ min} = \frac{111}{95} \times 1 \text{ min} \approx 1.17 \text{ minutes}$$

See the sample inputs and outputs for more examples.

```
[9]: # Sample run 1
     arrivalsList=[5,9,14,5,3,0,9,20,30,0,0]
     waiting_time(arrivalsList,15,2)
```

```
1.17
```

```
[10]: # Sample run 2
      arrivalsList=[5,9,14,5,3,0,9,20,30,0,0]
      print(f'Occupancy limit = 10\tWait time = {waiting_time(arrivalsList,10,3)} min.')
```

```
Occupancy limit = 10    Wait time = 3.23 min.
```

```
[11]: # Sample run 3
      arrivalsList=[5,9,14,5,3,0,9,20,30,0,0]
      print(f'Occupancy limit = 50\tWait time = {waiting_time(arrivalsList,50,3)} min.')
```

```
Occupancy limit = 50    Wait time = 0.09 min.
```

Write your final code below:

Q3. Job Decision Simulator (9 points)

This question asks you to write Python code to simulate the responses of a given student to a series of job offers based on the timing of the offers and the student's personal preferences. Some job offers are acceptable to her and others are unacceptable, and among the acceptable offers, she may like some better than others. Every job offer has a deadline, and if she does not accept an offer by the deadline, then the offer expires. The student is very risk averse, meaning that she never lets an acceptable job offer expire without having another offer in hand that she likes at least as much. More precisely, assume that the student always responds to job offers based on the following rules:

- She turns down unacceptable offers immediately.
- She holds on to the first acceptable job offer that she receives. (Here, "holding on" to an offer does not necessarily mean that she will eventually accept the offer, but it means that she does not turn it down unless she gets another offer at least as good.)
- At any time, she would hold on to at most one offer, which is her favorite acceptable offer received so far. If she receives a new acceptable offer while already holding on to an offer, she would compare the two and hold on to the offer she likes better, while turning down the other. If she likes the new offer equally as much as her favorite offer received so far, then she holds on to the one with the later deadline to respond, while turning down the one with the earlier deadline. If both of these favorite offers have the same deadline to respond, then she holds on to the one received earlier, while turning down the one received later.
- For the favorite offer that she is holding on to, if by the end of the last day to respond to this offer, the offer is still her favorite (i.e. she has not turned it down for something else), then she will accept this offer, at which point she is required to turn down all future offers since her acceptance of an offer cannot be reneged.

Write a function called `job_decision` with two input arguments:

- `offers`: a list in which each item corresponds to a job offer, and the order of the items corresponds to the order by which she receives the offers. Each job offer is represented by a list of three objects, `[offer_date,job_name,days_to_respond]`. The first object, `offer_date`, is an integer representing the day she receives the offer. The second object, `job_name`, is a string which serves as an unique identifier for the job. The third object, `days_to_respond`, is a positive integer indicating how many days she has to respond to the offer. For simplicity, assume that every offer is received on the morning of a day, and the deadline is always in the evening. example, if `offer_date=5` and `days_to_respond=3`, then she receives the offer on the morning of Day 5, and she must respond to the offer by the evening of Day 8 (because 5+3=8). You may assume that the offers are sorted so that those with earlier offer dates appear earlier in the list.
- `utility`: a dictionary in which the key is the unique identifier of a job (i.e. the same as `job_name` above), and the value is how much she likes the offer, with higher values corresponding to jobs she likes better. If two jobs have the same value, then she likes the two jobs equally. All of the entries in this dictionary are jobs she find acceptable, and if a job is not in the dictionary, then it means she does not find that job acceptable.

The function should return a string denoting the unique identifier (i.e. `job_name`) of the offer she accepts. If none of the offers are acceptable to her, then the function should return the empty string `''`. See the sample inputs and outputs below for examples.

[16]:
```
# Sample input 1
# She first holds on to the Intel offer, then Amazon (better than Intel), then Google (later␣
↪deadline than Amazon).
offers=[[5,'Intel',3],[8,'Amazon',7],[12,'Disney',3],[15,'Google',2],[15,'Facebook',2]]
utility={'Intel':5,'Amazon':8,'Google':8,'Facebook':8}
print(f'The student chose {job_decision(offers,utility)}.')
```

```
The student chose Google.
```

```
[17]: # Sample input 2
      # She first holds on to the Intel offer, but nothing as good appears by Day 8, so she␣
↪accepts Intel.
      offers=[[5,'Intel',3],[9,'Amazon',7],[12,'Disney',3],[15,'Google',2],[15,'Facebook',2]]
      utility={'Intel':5,'Amazon':9.5,'Google':10,'Facebook':10}
      print(f'The student chose {job_decision(offers,utility)}.')
```

```
The student chose Intel.
```

```
[18]: # Sample input 3
      # She first holds on to Amazon, then to Google, then to Facebook (each has later deadline␣
↪than the previous.)
      offers=[[5,'Intel',3],[8,'Amazon',7],[12,'Disney',3],[15,'Google',1],[15,'Facebook',2]]
      utility2={'Amazon':8,'Google':8,'Facebook':8}
      print(f'The student chose {job_decision(offers,utility2)}.')
```

```
The student chose Facebook.
```

```
[19]: # Sample input 4
      # She first holds on to Amazon, and by Day 15, there's no offer that is as good with later␣
↪deadline.
      offers=[[5,'Intel',3],[8,'Amazon',7],[12,'Disney',3],[13,'Google',2],[16,'Facebook',2]]
      utility2={'Amazon':8,'Google':8,'Facebook':8}
      print(f'The student chose {job_decision(offers,utility2)}.')
```

```
The student chose Amazon.
```

```
[20]: # Sample input 5
      # None of the offers are acceptable to her
      offers3=[[8,'Amazon',7],[12,'Disney',3],[13,'Google',2],[17,'Facebook',2]]
      utility3={'Apple':100, 'Intel': 80}
      job_decision(offers3,utility3)
```

```
''
```

Write your final code below (use the back sided if needed):