Contents

Exercises for Week 3: Algorithmic Thinking

Name: XXX

Grading Scheme:

- **3 (Essentially perfect)**: All required questions have a complete solution and all outputs are essentially correct based on the sample outputs.
- **2 (Solid effort)**: Several required questions may be incomplete or the outputs incorrect, but at least two thirds has a correct solution.
- **1 (Some effort)**: Between one third and two thirds of the required questions have a correct solution.
- **0 (No submission or essentially blank)**: No submission on Brightspace by the deadline, or less than one third of the required questions have a correct solution.

Every question is required unless it is marked with "(optional)". **To ensure that you get 3 out of 3, before you submit, restart the Kernel and run all, and check that all of the outputs match the sample outputs from the PDF version of the exercises. You need to develop the habit of meticulously checking your outputs in order to ensure the best grade.** I will not count minor formating, rounding, or importing issues, but otherwise your output should match the sample outputs exactly when rerun. The weekly exercises are intended to be completed in 4-5 hours, excluding class time. You should budget at least this much time per week for these exercises.

After you submit, download the .ipynb file you uploaded to Brightspace and double check that you uploaded the correct file, and that every question has been properly saved! Do not assume that Jupyter will save automatically for you.

Exercise 3.1: Optimal Pricing

Write a function "optPrice" with two input arguments:

- `priceList`: a list of proposed prices.
- `valueList`: a list of numbers. Each number represents the willingness to pay for the product from a particular customer.

For a given price, the demand is equal to the number of customers with willingness to pay greater than or equal to the price. The function should iterate through the list of prices, and compute the estimated revenue for each price, which is equal to the price times the demand.

The function should return two objects: the first is the best price found. The second object is a dictionary mapping each price to the estimated revenue for that price.

Sample run:

```
priceList=range(0,36,5)
valueList=[32,10,15,18,25,40,50,43]
bestPrice,revenueDict=optPrice(priceList,valueList)
print('Best price:',bestPrice)
print('Revenue dictionary:',revenueDict)
```

Correct output:

```
Best price: 25
Revenue dictionary: {0: 0, 5: 40, 10: 80, 15: 105, 20: 100, 25: 125, 30: 120, 35: 105}
```

Solve this problem by applying the four steps of algorithmic thinking.

Step 1. Understand (Write your summary in this Markdown cell)

Step 2. Decompose (Write your instructions in this Markdown cell)

Step 3. Analyze (Write code fragments in separate code cells to implement the trickiest steps, as in the in-class demonstration)

[ ]:

[ ]:

[ ]:

Step 4. Synthesis

[ ]: *# Version for debugging: with intermediate printing and no function encapsulation*

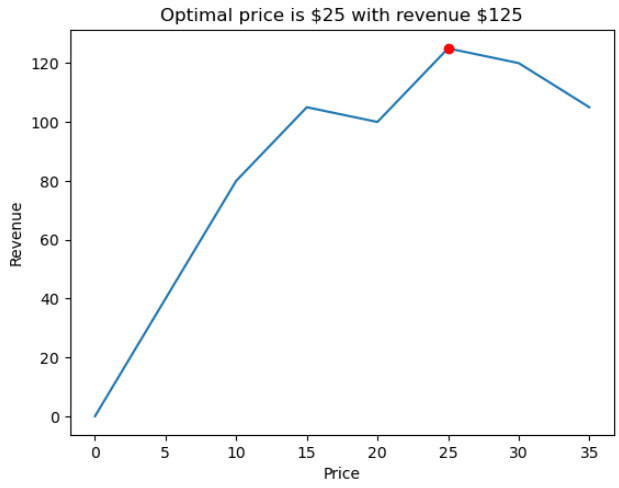
[ ]: *# Final code*

[14]:
```
# Test code 1
priceList=range(0,40,5)
valueList=[32,10,15,18,25,40,50,43]
bestPrice,revenueDict=optPrice(priceList,valueList)
print('Best price:',bestPrice)
print('Revenue dictionary:',revenueDict)
```

```
Best price: 25
Revenue dictionary: {0: 0, 5: 40, 10: 80, 15: 105, 20: 100, 25: 125, 30: 120, 35: 105}
```

[15]:
```
# Graphical display
priceList=range(0,40,5)
valueList=[32,10,15,18,25,40,50,43]
bestPrice,revenueDict=optPrice(priceList,valueList)

import matplotlib.pyplot as plt
revenueList=[revenueDict[price] for price in priceList]
plt.plot(priceList,revenueList)
plt.plot([bestPrice],[revenueDict[bestPrice]],'ro')
plt.xlabel('Price')
plt.ylabel('Revenue')
plt.title(f'Optimal price is \${bestPrice} with revenue \${revenueDict[bestPrice]}')
plt.show()
```



```
[16]: # Test code 2
      bestPrice,revenueDict=optPrice(range(0,50),[10,15,12,30,42,50,18,13,15,5,3,10,35,33,10])
      import matplotlib.pyplot as plt
      import pandas as pd
      pd.Series(revenueDict).plot()
      plt.plot([bestPrice],[revenueDict[bestPrice]],'ro')
      plt.xlabel('Price')
      plt.ylabel('Revenue')
      plt.title(f'Optimal price is \${bestPrice} with revenue \${revenueDict[bestPrice]}')
      plt.show()
```

Exercise 3.2: Optimal Stocking Level

Write a function named `optBaseStock` with four input arguments:

- `levelList`: a list of possible stocking levels to optimize over.
- `demandList`: a list of demand scenarios.
- `underage`: the unit cost of having too little inventory to meet demand.
- `overage`: the unit cost of having too much inventory.

For each possible stocking level, the function should compute the average inventory cost, which is defined as the average over all demand scenarios of the total underage cost plus the total overage cost. For example, if the stocking level is 10, the demand scenarios are `[6,12,14]`, the underage cost is 9 and the overage is 5, then

- The inventory cost for the scenario `demand=6` is $(10-6)\times 5 = 20$, because the stocking level is 4 units too high. (The overage cost of 5/unit is applied when the inventory is too high.)
- The inventory cost for the scenario `demand=12` is $(12-10)\times 9 = 18$, because the stocking level is 2 units too low. (The underage cost of 9/unit is applied when the inventory is too low.)
- The inventory cost for the scenario `demand=14` is $(14-10)\times 9 = 36$, because the stocking level is 4 units too low.

The average inventory cost for stocking level 10 is $(20+18+36)/3 = 74/3 \approx 24.67$.

The function should return two objects:

- `bestLevel`: the stocking level in `levelList` that achieves the minimum average inventory cost (if there is a tie, return the smallest stocking level that yields the minimum cost).
- `avCost`: a dictionary that maps each stocking level to the corresponding average inventory cost.

Sample run:

```
demandList=[10,18,5,20,16,30,15,3,5,10]
levelList=range(0,30,5)
```

```
underage=10
overage=3
bestLevel,avCost=optBaseStock(levelList,demandList,underage,overage)
print('bestLevel:',bestLevel)
print('avCost:',avCost)
```

Correct output:

```
bestLevel 20
avCost {0: 132.0, 5: 84.6, 10: 54.1, 15: 36.6, 20: 33.4, 25: 41.9}
```

Solve this problem by applying the four steps of algorithmic thinking.

Step 1. Understand (Write your summary of the task in this cell:)

Step 2. Decompose (Write your instructions in this Markdown cell)

Step 3. Analyze (Write code fragments in separate code cells to implement the trickiest steps)

[ ]:

[ ]:

[ ]:

[ ]:

Step 4. Synthesize (Combine your code fragments from Step 3, but do so in an incremental fashion and print intermediate results)

[ ]: *# Version for debugging: with intermediate printing and no function encapsulation*

[ ]: *# Final code: removing intermediate printing and encapuslating in a function*

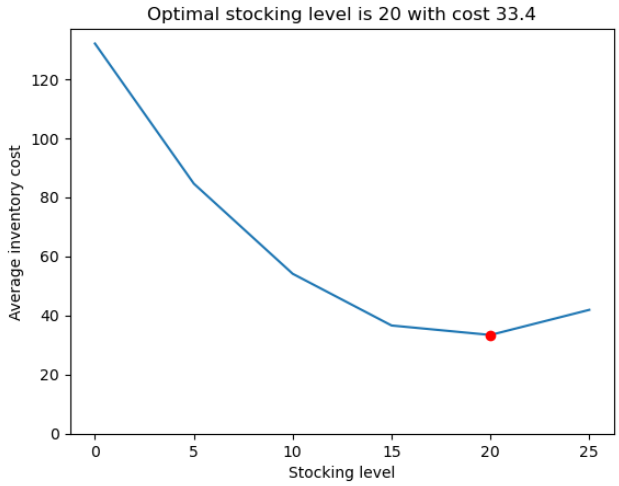
[23]:
```
# Sample run
demandList=[10,18,5,20,16,30,15,3,5,10]
levelList=range(0,30,5)
underage=10
overage=3
bestLevel,avCost=optBaseStock(levelList,demandList,underage,overage)
print('bestLevel:',bestLevel)
print('avCost:',avCost)
```

```
bestLevel: 20
avCost: {0: 132.0, 5: 84.6, 10: 54.1, 15: 36.6, 20: 33.4, 25: 41.9}
```

The following code illustrates how the results might be graphed as in Exercise 3.1

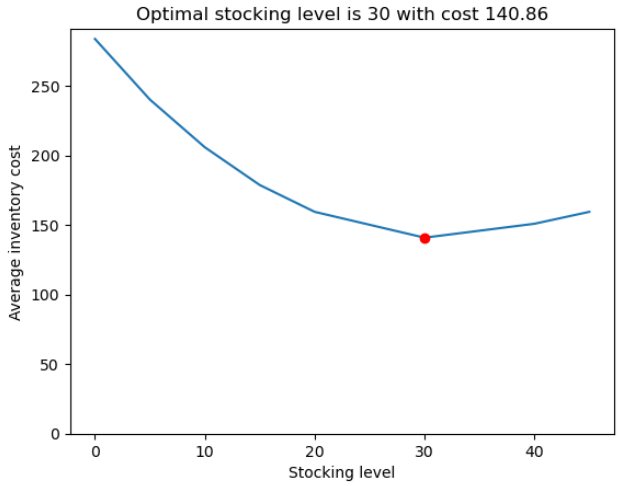
[24]:
```
demandList=[10,18,5,20,16,30,15,3,5,10]
levelList=range(0,30,5)
underage=10
overage=3
bestLevel,cost=optBaseStock(levelList,demandList,underage,overage)

import matplotlib.pyplot as plt
levelList=sorted(levelList)
costList=[cost[level] for level in levelList]
plt.plot(levelList,costList)
plt.plot([bestLevel],[cost[bestLevel]],'ro')
plt.title(f'Optimal stocking level is {bestLevel} with cost {round(cost[bestLevel],2)}')
plt.xlabel('Stocking level')
plt.ylabel('Average inventory cost')
plt.ylim(bottom=0)
plt.show()
```



[25]:
```
#Sample run 2
demandList=[10,18,5,16,30,15,3,5,10,60,50,30,40,20,30,20,50,80,30,60,80]
levelList=range(0,50,5)
underage=9
overage=6
bestLevel,cost=optBaseStock(levelList,demandList,underage,overage)

import matplotlib.pyplot as plt
levelList=sorted(levelList)
costList=[cost[level] for level in levelList]
plt.plot(levelList,costList)
plt.plot([bestLevel],[cost[bestLevel]],'ro')
plt.title(f'Optimal stocking level is {bestLevel} with cost {round(cost[bestLevel],2)}')
plt.xlabel('Stocking level')
plt.ylabel('Average inventory cost')
plt.ylim(bottom=0)
plt.show()
```



Exercise 3.3: Demand Estimation for Substitutable Products

Write a function named "demand" with two input arguments:

- `priceVector`: a list of length 2 containing two positive numbers, corresponding to the proposed prices for the two products.
- `values`: a list in which each element is a list of length 2, corresponding to the valuation of a customer

for the two products.

The function should return a list of two numbers, representing the number of customers purchasing each product. Assume the same customer behavior as in the paper coding exercise.

Sample run:

```
values=[[25,15],[18,18],[30,20],[30,30]]
priceVector=[25,20]
demand(priceVector,values)
```

Correct result:

```
[2, 1]
```

Solve this problem by applying the four steps of algorithmic thinking.

Step 1. Understand: Summarize the task in your own words and verify your understanding by manually computing the results for a few inputs.

Step 2. Decompose: Write clear and precise instructions for another human being to manually compute the appropriate results for any possible input, imagining that the person does not have access to the problem description but only has your instructions to go on.

Step 3. Analyze: For each part of the instructions above, plan how you would implement it using computer code. For the trickest parts, write code fragments and create intermediate inputs to test each fragment by itself.

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

Step 4. Synthesize: Using the results of Steps 1 and 2, put the code fragments from Step 3 together to create a complete solution. You should do this in an incremental fashion and print intermediate outputs as you go to make sure that each part of the code matches your expectations.

```
[ ]: # Version for debugging: with intermediate printing and no function encapsulation
```

```
[ ]: # Final code (modify the above after it already works)
```

```
[33]: # Test code
      values=[[25,15],[18,18],[30,20],[30,30]]
      priceVector=[25,20]
      demand(priceVector,values)
```

```
[2, 1]
```

```
[34]: # Test code 2
      values=[[25,15],[18,18],[30,20],[30,30],[20,10],[60,30],[0,20],[30,21]]
      priceVector=[30,20]
      demand(priceVector,values)
```

```
[2, 3]
```

For Exercise 3.4 and 3.5, you do not have to write out the four steps, but you are encouraged to code on paper first before typing.

Exercise 3.4: Airline Revenue Management

The question asks you to simulate the performance of a common revenue management strategy practiced by airlines for selling seats on a given flight.

Write a function called "simulateRevenue" with five input arguments:

- `valueList`: a list of numbers, with each number representing a customer's maximum willingness to pay for a seat in the given flight. The list is ordered according to the sequence in which customers make purchase decisions, with the earlier entries corresponding to customers who try to make purchases earlier than others.
- `inventory`: a positive number representing the total number of seats that can be sold on the given flight.
- `threshold`: a non-negative number such that as soon as the number of remaining seats is less than or equal to this threshold, then the price changes.
- `price1`: the initial price for each seat before the price change described above.
- `price2`: the updated price for each seat after the price change.

Assume that each customer goes online to purchase the seat in the order as they appear in the list, and each customer buys whenever his/her willingness to pay is greater than or equal to the current price, which is equal to `price1` if the number of unsold tickets is strictly greater than `threshold` and is equal to `price2` if the number of unsold tickets is less than or equal to `threshold`. (The number of unsold tickets is initially equal to `inventory`, before any customer makes a purchase.) **The function should return (not print) the total revenue corresponding to the given parameters.**

For example, if `valueList=[50,60,100,20,80,90,80,200,250,100]`, then

- `simulateRevenue(valueList, 3, 0, 60, 1)` should return `180` because all 3 tickets are sold at the initial price of 60, since the price change never happens as threshold=0.
- `simulateRevenue(valueList, 3, 0, 200, 1)` should return `400` because only 2 people are willing to pay the initial price of 200.
- `simulateRevenue(valueList, 3, 1, 60, 200)` should return `320` because seats are sold at 60 each to the customer in position 1 (with willingness to pay 60) and to the customer in position 2 (with willingness to pay 100). At this point, only 1 seat is left, and the price is increased to 200. The next person who buys the last seat is the one in position 7 (with willingness to pay 200).
- `simulateRevenue(valueList, 3, 1, 60, 251)` should return `120` because after two seats are sold at price 60, the price increases to 251, and none of the later customers are willing to pay this much.
- `simulateRevenue(valueList, 3, 1, 200, 110)` should return `400` because only two customers are willing to pay the initial price of 200.
- `simulateRevenue(valueList, 3, 2, 95, 200)` should return `495` because after selling an initial ticket at 95 to the customer in position 2 (with willingness to pay 100), the price becomes 200 and the customers in positions 7 and 8 buy (with willness to pay 200 and 250).

```
[ ]: # Write your function here
```

```
[36]: # Sample runs
      valueList=[50,60,100,20,80,90,80,200,250,100]
      simulateRevenue(valueList, 3, 0, 60, 1)
```

```
180
```

```
[37]: simulateRevenue(valueList, 3, 0, 200, 1)
```

```
400
```

```
[38]: simulateRevenue(valueList, 3, 1, 60, 200)
```

```
320
```

```
[39]: simulateRevenue(valueList, 3, 1, 60, 251)
```

```
120
```

```
[40]: simulateRevenue(valueList, 3, 1, 200, 110)
```

```
400
```

```
[41]: print(simulateRevenue(valueList, 3, 2, 95, 200))
```

495

(Optional) Exercise 3.5: Simulating Hospital Diversions

Suppose that the hospital has a limited number of ICU beds for Critical patients and regular beds for Non-Critical patients. When not enough ICU beds are available to accommodate all Critical patients, then Critical patients are diverted to regular beds. (Here, diversion means that the patient who was supposed to be assigned to an ICU bed is now assigned to a regular bed.) When not enough regular beds are available, some patients may need to be diverted to facilities outside the hospital. A Non-Critical patient is NOT allowed to occupy an ICU bed, even if there are spare ICU beds but no regular beds available.

For simplicity, assume that the diversions in a given week do not affect the number of Critical and Non-Critical patients in later weeks. Moreover, when not enough regular beds are available, Critical patients have priority over Non-Critical patients.

Write a function called "simulateDiversions" with four input parameters:

- **critical**: a list corresponding to the number of patients each week exhibiting Critical symptoms.
- **nonCritical**: a list corresponding to the number of patients each week exhibiting Non-Critical symptoms.
- **ICU**: a positive integer representing the number of ICU beds at the hospital.
- **regular**: a positive integer representing the number of regular beds at the hospital.

The function should return two numbers:

- the average number of patients diverted from ICU beds each week. (These are Critical patients needing an ICU bed but could not get one. This includes Critical patients diverted to regular beds as well as to facilities outside the hospital.)
- the average number of patients diverted from regular beds each week. (These are patients who are forced to be transferred to facilities outside the hospital, and may include both Critical and non-Critical patients.)

See the following example with 3 ICU beds and 20 regular beds and five weeks of data:

Critical	Non-Critical	ICU Diversions	Regular Diversions
2	23	0	3
4	19	1	0
5	20	2	2
1	24	0	4
6	18	3	1

In the example, the two numbers returned should be $(0+1+2+0+3)/5 = 1.2$ and $(3+0+2+4+1)/5 = 2$. After writing your function, an user should be able to run the following test code without any error.

```
[ ]: # Write your code below
```

```
[43]: critical=[2,4,5,1,6]
      nonCritical=[23,19,20,24,18]
      avICUDiv,avRegularDiv=simulateDiversions(critical,nonCritical,3,20)
      print(f'The average number of ICU vDiversions is {avICUDiv}.')
      print(f'The average number of Regular Diversions is {avRegularDiv}.')
```

```
The average number of ICU vDiversions is 1.2.
The average number of Regular Diversions is 2.0.
```

```
[44]: avICUDiv,avRegularDiv=simulateDiversions([3,0,0,1,2,3,1,0],[0,5,3,6,0,0,10,2],2,5)
      print(f'The average number of ICU vDiversions is {avICUDiv}.')
      print(f'The average number of Regular Diversions is {avRegularDiv}.')
```

```
The average number of ICU vDiversions is 0.25.
The average number of Regular Diversions is 0.75.
```

```
[45]: avICUDiv,avRegularDiv=simulateDiversions([3,0,0,1,2,3,1,0],[0,5,3,6,0,0,10,2],2,6)
      print(f'The average number of ICU vDiversions is {avICUDiv}.')
      print(f'The average number of Regular Diversions is {avRegularDiv}.')
```

```
The average number of ICU vDiversions is 0.25.
The average number of Regular Diversions is 0.5.
```

```
[46]: avICUDiv,avRegularDiv=simulateDiversions([3,0,0,1,2,3,1,0],[0,5,3,6,0,0,10,2],1,4)
      print(f'The average number of ICU vDiversions is {avICUDiv}.')
      print(f'The average number of Regular Diversions is {avRegularDiv}.')
```

```
The average number of ICU vDiversions is 0.625.
The average number of Regular Diversions is 1.125.
```