# Contents


Week 3: Algorithmic Thinking . . . 1
Session 5: The Four Steps of Algorithmic Thinking . . . 1
Example: Epidemic Capacity Planning . . . 1
Exercise 3.1: Optimal Pricing . . . 3
Exercise 3.2: Optimal Stocking Level . . . 6
Session 6: Translating English Steps into Working Code . . . 9
In-Class Exercise: Paper Coding . . . 9
Recap of the four steps: . . . 10
Exercise 3.3: Demand Estimation for Substitutable Products . . . 10
Exercise 3.4: Airline Revenue Management . . . 10
Exercise 3.5: Simulating Hospital Diversions . . . 11


# Week 3: Algorithmic Thinking

Algorithmic thinking is a process of systematically breaking down a complex task into a sequence of simpler, well-defined steps. It is the main skill required when creating larger computer programs. Mastering this thinking skill would enable you to make steady progress when solving seemingly intractable problems and arrive at creative solutions in a methodical way.

# Session 5: 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.

## Example: Epidemic Capacity Planning

A city is trying to predict how many ventilators it will need during a certain epidemic. To help the policy makers analyze various scenarios, write a function called **capacityNeeded** with one input parameter:

- **arrivals**: a list of numbers, representing a forecast of the weekly arrivals of new patients requiring a ventilator.

Assume that each patient requires a ventilator for exactly 3 weeks. The function should return an integer corresponding to the minimum number of ventilators needed to satisfy all the demand.

For example, if `arrivals=[5,8,3,10,7,4,9,5,8]`, then the function should return 22, because in the last three weeks, 9+5+8=22 ventilators are needed, and having 22 is sufficient to satisfy demand in any week. (In the first week, only 5 ventilators are needed; in the second week, 5+8=13 are needed; in the third week 5+8+3=16 are needed; in the fourth week 8+3+10=21 are needed; in the fifth week, 3+10+7=20 are needed, etc.)

### In-class exercise: Practicing Steps 1 and 2

Manually compute the desired answer corresponding to the sample input `arrivals=[5,8,3,10,7,4,9,5,8]`. Ideally, you should create a table to track the computations, and describe precisely how each entry in the table is computed. This logic will be the basis of your coding later.

### Demonstration of Steps 3 and 4

#### Step 3. Analyze

The trickiest step is in computing the "Demand" column. A conceptually simple approach is to use the `sum` function along with list slicing. Recall that slicing does not include the final index, which is why we need `week+1`.

```
[1]: week=4
     arrivals=[5,8,3,10,7,4,9,5,8]
     demand=sum(arrivals[week-2:week+1])
     demand
```

20

The above will yield an error if week is 0 or 1, and one way to correct is as follows:

```
[2]: week=1
     arrivals=[5,8,3,10,7,4,9,5,8]
     demand=sum(arrivals[max(0,week-2):week+1])
     demand
```

13

Finding the "Max Demand" can be done by keeping track of the maximum so far, as below.

```
[3]: maxDemand=0
     demand=20
     if demand>maxDemand:
         maxDemand=demand
     maxDemand
```

20

#### Step 4. Synthesis

We combine the above code fragments and place everything in a for loop. For ease of debugging, we printing intermediate outputs to recreate the table.

```
[4]: arrivals=[5,8,3,10,7,4,9,5,8]
     print('Week\tArrival\tDemand\tMax Demand')
     maxDemand=0
     week=0
     for arrival in arrivals:
         demand=sum(arrivals[max(0,week-2):week+1])
         if demand>maxDemand:
             maxDemand=demand
         print(f'{week}\t{arrival}\t\t{demand}\t{maxDemand}')
         week+=1
     print('Final answer:', maxDemand)
```

```
Week    Arrival Demand  Max Demand
0       5               5       5
1       8               13      13
2       3               16      16
3       10              21      21
4       7               20      21
5       4               21      21
6       9               20      21
7       5               18      21
8       8               22      22
Final answer: 22
```

After verifying the logic of the code by reproducing the desired table, we remove the intermediate print statements and place everything in a function, as asked for by the question prompt.

```
[5]: def capacityNeeded(arrivals):
         maxDemand=0
         week=0
         for arrival in arrivals:
             demand=sum(arrivals[max(0,week-2):week+1])
             if demand>maxDemand:
                 maxDemand=demand
             week+=1
         return maxDemand
```

```
[6]: # Test code 1
     arrivals=[5,8,3,10,7,4,9,5,8]
     print('Capacity Needed:',capacityNeeded(arrivals))
```

```
Capacity Needed: 22
```

```
[7]: # Test code 2
     capacityNeeded([5,8,3,10,6,11,9,12,15,9,5,7])
```

```
36
```

### Alternative Solution

An alternative way to compute the "Demand" column is by using the previous row's Demand, as below.

```
[8]: def capacityNeeded(arrivals):
         maxDemand=0
         demand=0
         week=0
         for arrival in arrivals:
             demand+=arrival
             if week>=3:
                 demand-=arrivals[week-3]
             if demand>maxDemand:
                 maxDemand=demand
             week+=1
         return maxDemand
```

## 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**

**Step 2. Decompose**

**Step 3. Analyze**

**Step 4. Synthesis**

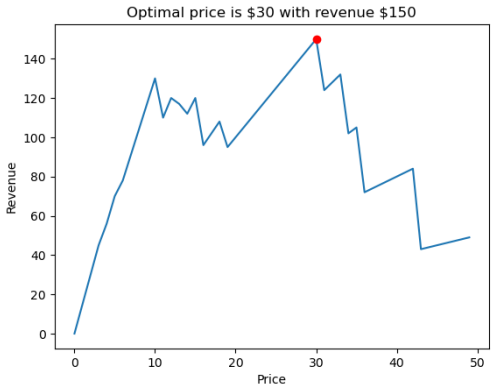
[14]:
```
# Test code 1
priceList=range(0,40,5)
valueList=[32,10,15,18,25,40,50,43]
bestPrice,revenueDict=optPrice(priceList,valueList)
print('Best price:',bestPrice)
print('Revenue dictionary:',revenueDict)
```

```
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**

**Step 2. Decompose**

**Step 3. Analyze**

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

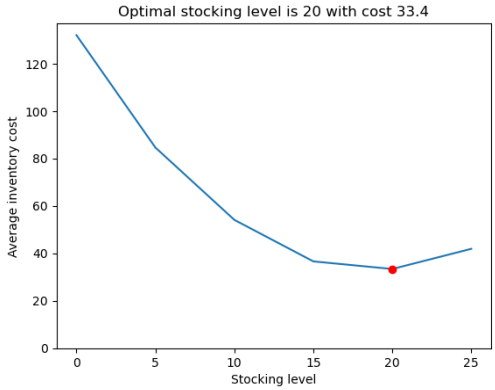
[23]:
```
# Sample run
demandList=[10,18,5,20,16,30,15,3,5,10]
levelList=range(0,30,5)
underage=10
overage=3
bestLevel,avCost=optBaseStock(levelList,demandList,underage,overage)
print('bestLevel:',bestLevel)
print('avCost:',avCost)
```

```
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

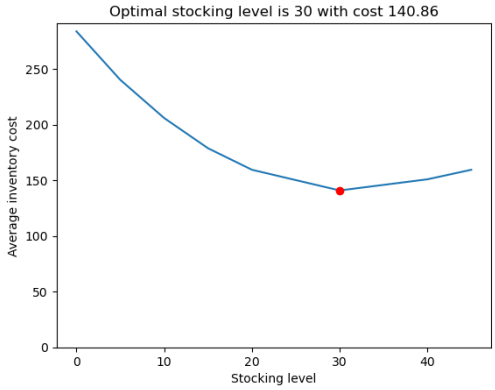
[24]:
```
demandList=[10,18,5,20,16,30,15,3,5,10]
levelList=range(0,30,5)
underage=10
overage=3
bestLevel,cost=optBaseStock(levelList,demandList,underage,overage)

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()
```



# Session 6: Translating English Steps into Working Code

## In-Class Exercise: Paper Coding

One tip for efficiently solving a complex programming task is to first draft the code on paper. Perhaps counterintuitively, it is often faster if you first code on paper and then type it in, rather than if you worked directly on the computer to begin with. This is because handwriting helps you to focus on the overall logic rather than the syntax.

**Without using a computer, write on a piece of paper Python code that implements the following logic:** Given a list named `curVal`, representing the valuation (willingness to pay) of a customer for two products, as well as list named `priceVector`, representing the price of the two products, print `"Purchase Product 0"` if the customer purchases the first product; print `"Purchase Product 1"` if the customer purchases the second product; print `"Purchase nothing"` if the customer purchases neither.

**Assumptions:** The customer will purchase at most one product, and the product that is purchased is one in which the customer's willingness to pay is at least equal to the price. If the customer's valuations for both products are greater than or equal to the corresponding prices, then the customer will purchase the product in which the difference between his/her valuation and the price is the largest. If there is a tie in this difference, then the customer will purchase Product 0. If the valuation is less than the price for both products, then the customer would not purchase anything.

For example, if the valuation of a customer is `[9,8]` then

- If `priceVector=[6,4]`, then the customer will purchase Product 1 because $8-4>9-6$.
- If `priceVector=[5,4]`, then the customer will purchase Product 0 because $9-5\geq 8-4$.
- If `priceVector=[10,8]`, then the constumer will purchase Product 1.
- If `priceVector=[10,10]`, then the customer will purchase nothing.

For concreteness, you can start your code with the following two lines.

```
curVal=[25,15]
priceVector=[25,10]
```

## Recap of the four steps:

**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.

## 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.

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

```
[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).

```
[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
```

## 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.

```
[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.
```