Contents

Exercises for Week 4: Coding and Debugging More Quickly

Grading Scheme:

Important: This Jupyter notebook needs to be completed and submitted via Blackboard before the due date to receive a non-zero grade.

- 5: Every question is completed and the code returns the correct outputs on all of the sample inputs.
- 4: Almost complete, but certain questions are blank, returns clearly incorrect outputs, or the code there does not run at all.
- 3: This score will not be assigned, as everyone should strive to get 4 or 5.
- 2: Not close to complete, but at least 50% complete.
- 1: At least 10% complete, but less than 50% complete
- 0: Less than 10% complete, or response is identical to someone else's, indicating plagiarism.

A perfect score is 5. Note that your code does not need to be absolutely perfect to receive a 5, but you need to complete every question and ensure that the outputs are correct on all of the sample runs included here. **To ensure that you get 5 out of 5, before you submit, restart the Kernel and run all, and check that all of the outputs are as intended.**

These exercises are intended to be completed in 5-7 hours, including class time. You should budget at least this much time before the due date.

Name: XXX

Exercise 4.1: Practicing Debugging

In the following, you should try to make the code correct using as few changes as possible, after figuring out the errors using code dissection and print debugging. Do not rewrite the code from scratch or copy/paste a correct solution. Refer to previous weeks' handouts for the instructions to these exercises.

a) Attempt for the "Referral Marketing" Example from Week 2

[ ]:
```
# Code to debug
def buyer(nmonths):

    buyers=[1]
    for n in nmonths:
        buyers = buyers[-1]+buyers[-2]
    return buyers[-1]
```

[62]:
```
# Test code
buyer(12)
```

```
233
```

b) Attempt for the "Epidemic Capacity Planning" Example from Week 3

```
[ ]: # Code to debug
     def capacityNeeded(arrivals):
         maxdemand=[arrivals[0]]
         week=0
         for i in range(0,(len(arrivals))):
             week+=1
             if i>=2:
                 demand=arrivals[i]+arrivals[i-1]+arrivals[i-2]
             if i>=1:
                 demand=arrivals[i]+arrivals[i-1]
          #   else:
         #       demand=arrivals[i]
             if maxdemand<demand:
                 maxdemand=demand
         print(maxdemand)
```

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

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

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

```
Week    Arrival Demand  Max Demand
1       5       5       5
2       8       13      13
3       3       16      16
4       10      21      21
5       6       19      21
6       11      27      27
7       9       26      27
8       12      32      32
9       15      36      36
10      9       36      36
11      5       29      36
12      7       21      36

36
```

c) Attempt for Exercise 3.2: Demand Estimation for Substitutable Products

```
[ ]: def demand(priceVector,values):
         for i in values:
             product1=[]
             product2=[]
             diff0=values[0]-priceVector[0]
             diff1=values[1]-priceVector[1]
             if diff0<0 and diff1<0:
```

```
            continue
        elif diff0>=diff1:
            product1.append(1)
        else:
            product2.append(1)
        totalproduct=product1+product2
        return totalproduct
```

```
[52]: # Test code 1
      values=[[25,15],[18,18],[30,20],[30,30]]
      priceVector=[25,20]
      demand(priceVector,values)
```

```
[2, 1]
```

```
[53]: # Test code 2
      values=[[25,15],[18,18],[25,20],[25,35],[18,20],[26,21]]
      priceVector=[25,20]
      demand(priceVector,values)
```

```
[3, 2]
```

Exercise 4.2: Grocery Store Restocking

This question asks you to make a tool that helps a grocery store to analyze their policy for restocking shelves for a certain non-perishable item. Write a function called `analyzeScenario` with three input parameters:

- `demandList`: a non-empty list of non-negative integers representing the forecasted daily demand for the item, corresponding to a period of consecutive days. The number of days is `len(demandList)`.
- `stockingLevel`: a positive integer representing the maximum number of units that the store will stock on its shelves at any time.
- `minimumLevel`: a non-negative integer representing the minimum number of units on the shelves that the store can tolerate without restocking.

Assume that the store makes its stocking decision at the end of each day after closing. If the leftover inventory on the shelf at the end of a day is strictly below the "minimumLevel", then the store will restock to a full shelf, and the inventory at the beginning of the next day will be equal to "stockingLevel". If the leftover inventory at the end of a day is greater than or equal to "minimumLevel", then the store will not add anything to the shelf, and the inventory at the beginning of the next day will be the same as the leftover inventory. On the first day, the shelf is full, so the inventory level is equal to "stockingLevel".

Your function should print (not return) the number of times it would decide to restock during the period represented by the input data.

For example, the sample run

```
analyzeScenario([3,4,2,5,15,3,9,3,1,3,9],10,3)
```

should result in exactly the following message printed to screen.

```
The store needs to restock 4 times.
```

The following table illustrates the inventory dynamics.

Beginning Inventory	Demand	Leftover Inventory	Restock?
10	3	7	No
7	4	3	No
3	2	1	Yes
10	5	5	No
5	15	0	Yes
10	3	7	No

Beginning Inventory	Demand	Leftover Inventory	Restock?
7	9	0	Yes
10	3	7	No
7	1	6	No
6	3	3	No
3	9	0	Yes
# of times to restock:			**4**

Note that if demand is greater than the beginning inventory, the leftover inventory is zero. Otherwise, the leftover inventory is equal to the beginning inventory minus the demand. The final answer (the number of times to restock) is equal to the number of Yes's in the last column of the table.

Sample run 2:

```
analyzeScenario([3,4,2,5,15,3,9,3,1,3,9],9,3)
```

The printed message should be exactly as below:

```
The store needs to restock 6 times.
```

Sample run 3:

```
analyzeScenario([8,3,2,6,9,3,5,2,9,10],9,5)
```

The printed message should be exactly as below:

```
The store needs to restock 7 times.
```

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)

```
[1]: # Code with intermediate printing
```

```
[4]: # Final code
```

```
[5]: # Sample run 1
     analyzeScenario([3,4,2,5,15,3,9,3,1,3,9],10,3)
```

```
The store needs to restock 4 times.
```

```
[6]: # Sample run 2
     analyzeScenario([3,4,2,5,15,3,9,3,1,3,9],9,3)
```

```
The store needs to restock 6 times.
```

```
[7]: # Sample run 3
     analyzeScenario([8,3,2,6,9,3,5,2,9,10],9,5)
```

```
The store needs to restock 7 times.
```

Exercise 4.3: Demand Estimation with n Substitutable Products

This exercise generalizes Exercise 3.3 to n products, where n is any positive integer.

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

- `prices`: a list of n prices, one for each product.
- `values`: a list in which each element represents a customer's valuations for the n products. The valuations is a list of length n, which corresponds to the customer's willingness to pay for each of the n products.

The function should return a list of length n representing the number of each product sold. You should assume that each customer:

- Does not purchase anything if his/her valuation for each product is strictly less than its price.
- Otherwise, purchase the product in which the difference between his/her valuation and the price is the largest. When there is a tie, the customer will purchase the product with the smaller index.

For example, if `prices=[10,8,12]`, then

- A customer with valuations `[9,7,11]` purchases nothing.
- A customer with valuations `[10,8,12]` purchases product 1.
- A customer with valuations `[9,8,12]` purchases product 2.
- A customer with valuations `[9,8,13]` purchases product 3.

Sample run 1:

```
prices=[10,8,12]
values=[[9,7,11],[10,8,12],[9,8,12],[9,8,13]]
ans=demand(prices,values)
for i in range(len(prices)):
    print(f'Demand for product {i+1}:',ans[i])
```

Correct output:

```
Demand for product 1: 1
Demand for product 2: 1
Demand for product 3: 1
```

Sample run 2:

```
prices=[20,15,30]
values=[[30,30,20],[40,10,15],[18,13,29],[40,30,50],[10,30,50],[10,10,10],[20,15,30]]
ans=demand(prices,values)
for i in range(len(prices)):
    print(f'Demand for product {i+1}:',ans[i])
```

Correct output:

```
Demand for product 1: 3
Demand for product 2: 1
Demand for product 3: 1
```

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)

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

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

```
[14]: # Sample run 1
      prices=[10,8,12]
```

```
values=[[9,7,11],[10,8,12],[9,8,12],[9,8,13]]
ans=demand(prices,values)
for i in range(len(prices)):
    print('Demand for product',i+1,':',ans[i])
```

```
Demand for product 1 : 1
Demand for product 2 : 1
Demand for product 3 : 1
```

```
[15]: # Sample run 2
prices=[20,15,30]
values=[[30,30,20],[40,10,15],[18,13,29],[40,30,50],[10,30,50],[10,10,10],[20,15,30]]
ans=demand(prices,values)
for i in range(len(prices)):
    print('Demand for product',i+1,':',ans[i])
```

```
Demand for product 1 : 3
Demand for product 2 : 1
Demand for product 3 : 1
```

```
[16]: # Sample run 3
prices=[10,8,12,8]
values=[[9,7,11,12],[10,8,12,8],[9,8,12,5],[9,8,13,12]]
ans=demand(prices,values)
for i in range(len(prices)):
    print('Demand for product',i+1,':',ans[i])
```

```
Demand for product 1 : 1
Demand for product 2 : 1
Demand for product 3 : 0
Demand for product 4 : 2
```

Exercise 4.4: Simulating Availabilities of Hospital Beds

One challenge in health care operations is to forecast the number of hospital beds that are available at a given time, since patients admitted in the past may stay for several days and the number of beds are limited. If no more beds are available, then incoming patients may need to be turned away.

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

- `demandList`: a list of positive integers representing the number of incoming patients desiring a hospital bed in each day. (The first number corresponds to day 0, the second number to day 1, and so on.)
- `beds`: a positive integer representing the total number of hospital beds available.
- `stay`: a positive integer representing the number of days each admitted patient will stay. If `stay=1`, then every admitted patient leaves before any incoming patients arrive the next day. If `stay=2`, then each patient admitted on day t will occupy a bed also for day $t+1$, and leave before incoming patients arrive on day $t+2$.

The function should return a list `admissionRecord`, corresponding to the number of incoming patients admitted on each day.

Sample run 1:

```
demandList=[1,2,1,0,2,3]
beds=2
stay=2
admissionRecord=admissionSimulation(demandList,beds,stay)
print(f'Day\tDemand\tAdmitted')
for i in range(len(demandList)):
    print(f'{i}\t{demandList[i]}\t{admissionRecord[i]}')
```

Correct output:

```
Day    Demand    Admitted
0        1         1
1        2         1
2        1         1
3        0         0
4        2         2
5        3         0
```

Sample run 2:

```
demandList=[5,8,6,8,4,4,8,6,1]
beds=7
stay=3
admissionRecord=admissionSimulation(demandList,beds,stay)
print(f'Day\tDemand\tAdmitted')
for i in range(len(demandList)):
    print(f'{i}\t{demandList[i]}\t{admissionRecord[i]}')
```

Correct output:

```
Day    Demand    Admitted
0        5         5
1        8         2
2        6         0
3        8         5
4        4         2
5        4         0
6        8         5
7        6         2
8        1         0
```

Sample run 3:

```
# Sample run 3
demandList=[5,8,6,4,4,4,8,1,3]
beds=7
stay=3
admissionRecord=admissionSimulation(demandList,beds,stay)
print(f'Day\tDemand\tAdmitted')
for i in range(len(demandList)):
    print(f'{i}\t{demandList[i]}\t{admissionRecord[i]}')
```

Correct output:

```
Day       Demand    Admitted
0         5         5
1         8         2
2         6         0
3         4         4
4         4         3
5         4         0
6         8         4
7         1         1
8         3         2
```

Hint: In Step 2, you want to create a table. The above tables are insufficient to carry through the logic, as you also need to keep track of the number of discharges at the beginning of each day, as well as the number of available beds before new patients arrive.

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)

[3]:
```
# Version for debugging: with intermediate printing and no function encapsulation
```

[20]:
```
# Final code: removing intermediate printing and encapuslating in a function
```

[22]:
```
# Sample run 1
demandList=[1,2,1,0,2,3]
beds=2
stay=2
admissionRecord=admissionSimulation(demandList,beds,stay)
print(f'Day\tDemand\tAdmitted')
for i in range(len(demandList)):
    print(f'{i}\t{demandList[i]}\t{admissionRecord[i]}')
```

```
Day     Demand  Admitted
0       1       1
1       2       1
2       1       1
3       0       0
4       2       2
5       3       0
```

[23]:
```
# Sample run 2
demandList=[5,8,6,8,4,4,8,6,1]
beds=7
stay=3
admissionRecord=admissionSimulation(demandList,beds,stay)
print(f'Day\tDemand\tAdmitted')
for i in range(len(demandList)):
    print(f'{i}\t{demandList[i]}\t{admissionRecord[i]}')
```

```
Day     Demand  Admitted
0       5       5
1       8       2
2       6       0
3       8       5
4       4       2
5       4       0
6       8       5
7       6       2
8       1       0
```

[24]:
```
# Sample run 3
demandList=[5,8,6,4,4,4,8,1,3]
beds=7
stay=3
admissionRecord=admissionSimulation(demandList,beds,stay)
print(f'Day\tDemand\tAdmitted')
for i in range(len(demandList)):
    print(f'{i}\t{demandList[i]}\t{admissionRecord[i]}')
```

```
Day     Demand  Admitted
0       5       5
1       8       2
2       6       0
3       4       4
4       4       3
5       4       0
6       8       4
```

```
7       1       1
8       3       2
```

Exercise 4.5: Health Insurance Selection

This question asks you to create a function to help individuals select the most cost effective health insurance given the terms of each plan and the individual's estimated health expenditures.

Write a function called "bestPlan" with two input arguments:

- `planInfo`: a dictionary in which the index is the name of a plan and the value is a list of 3 non-negative numbers. The first number is the annual premium of this plan, the second number is the annual deductible, and the third number is the proportion of expenditures (beyond the deductible) reimbursed by the plan.
- `expenditure`: a non-negative number representing the individual's total health care spending without any health insurance.

The function should return (not print) the name of the best plan for the individual, where best is defined as having the lowest total cost, as given by

$$\text{Total cost} = \text{Premium} + \text{Expenditure} - \text{Reimbursements},$$

where Reimbursements is equal to the proportion reimbursed multiplied by the reimbursable amount, and the reimbursable amount is equal to the amount by which the expenditures exceeds the deductible.

For example, if Premium = 2000, Proportion reimbursed = 0.9, Deductible = 200, and Expenditure = 100, then the reimbursable amount is 0, and Total cost = $2000 + 100 - 0 = 2100$.

However, if Expenditure = 10000, then the reimbursable amount is $10000 - 200 = 9800$, and Total cost = $2000 + 10000 - (9800) \times 0.9 = 3180$. If there are multiple plans tied for the lowest total cost, then the function can return any of them.

For example, if

```
planInfo={'BlueCross PPO':[3000,0,0.95],'Anthem HMO':[1200,100,0.8],'Cigna EPO':[2400,1000,0.98],'No Insurance':[0,0,0]}
```

Then,

- `bestPlan(planInfo,1500)` should return `'No Insurance'`.
- `bestPlan(planInfo,1700)` should return `'Anthem HMO'`.
- `bestPlan(planInfo,11000)` should return `'Anthem HMO'`.
- `bestPlan(planInfo,12000)` should return `'BlueCross PPO'`.
- `bestPlan(planInfo,13000)` should return `'Cigna EPO'`.
- `bestPlan(planInfo,100000)` should return `'Cigna EPO'`.

Apply the four steps of algorithmic thinking to solve this problem.

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)

[ ]:

[ ]:

[ ]:

Step 4. Synthesize

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

```
[29]: # Final code
```

```
[30]: # Test code
      planInfo={'BlueCross PPO':[3000,0,0.95],'Anthem HMO':[1200,100,0.8],'Cigna EPO':[2400,1000,0.98],'No␣
↪Insurance':[0,0,0]}
      bestPlan(planInfo,1500)
```

```
'No Insurance'
```

```
[31]: bestPlan(planInfo,1700)
```

```
'Anthem HMO'
```

```
[32]: bestPlan(planInfo,11000)
```

```
'Anthem HMO'
```

```
[33]: bestPlan(planInfo,12000)
```

```
'BlueCross PPO'
```

```
[34]: bestPlan(planInfo,13000)
```

```
'Cigna EPO'
```

```
[35]: bestPlan(planInfo,100000)
```

```
'Cigna EPO'
```

For Exercise 4.6 and 4.7, you do not have to write out the four steps, but you should still apply algorithmic thinking when solving these problems.

Exercise 4.6: Meeting Scheduling

This question asks you to create a tool to obtain the best times to schedule a meeting given the availabilities of all people interested. The tool outputs the time slot(s) that the most number of people can make.

Write a function called `getBestTimes` with one input argument:

- `availabilities`: a dictionary in which the key is the name of a person and the value is the list of all time slots that the person can make. Each time slot is represented by a string containing the date and the starting time. You can assume that each time slot has the same length so that only the start time matters. Moreover, the strings encoding the start times are formatted in a consistent way so that different strings represent different time slots, while equal strings represent the same time slot.

The function should return (not print) a dictionary where the keys are the best time slots (i.e. the slots the most number of people can make) and the value is the list of people who can make that time slot. In other words, if there are two time slots, say A and B, that 4 people can make, and for each of the remaining time slots, at most 3 people can make it, then the dictionary should contain the time slots A and B as keys and the values should be the people who can make each time slot.

Your returned dictionary must contain all of the optimal time slots. However, the order in which time slots appear in the dictionary does not matter. Moreover, the order names appear in the list of people who can make each time slot does not matter. See the sample outputs for illustrations.

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

```
[37]: # Sample run 1
      best=getBestTimes({'Alice':['Tue 9am','Tue 9:30am', 'Tue 2pm','Wed 11am'],\
                         'Bob':['Tue 11am','Tue 11:30am','Tue 1:30pm','Tue 2pm']})
      best
```

```
{'Tue 2pm': ['Alice', 'Bob']}
```

```
[38]: # Sample run 2
      getBestTimes({'Alice':['Tue 9am','Tue 9:30am', 'Tue 2pm','Wed 11am'],\
                    'Bob':['Tue 11am','Tue 11:30am','Tue 1:30pm','Tue 2pm', 'Wed 11am']})
```

```
{'Tue 2pm': ['Alice', 'Bob'], 'Wed 11am': ['Alice', 'Bob']}
```

```
[39]: # Sample run 3
      availabilities={}
      availabilities['Alice']=['Tue 9am','Tue 9:30am', 'Tue 2pm','Wed 11am']
      availabilities['Bob']=['Tue 11am','Tue 11:30am','Tue 1:30pm','Tue 2pm', 'Wed 11am']
      availabilities['Charlie']=['Wed 11am','Thu 11am','Fri 11am']
      availabilities['Dylan']=['Mon 9am','Mon 2pm','Tue 9am','Tue 2pm','Wed 9am']
      getBestTimes(availabilities)
```

```
{'Tue 2pm': ['Alice', 'Bob', 'Dylan'], 'Wed 11am': ['Alice', 'Bob', 'Charlie']}
```

```
[40]: # Sample run 4
      d={}
      d['Jack']=['5/12 9am','5/12 11am','5/12 12pm','5/13 2pm','5/13 3pm']
      d['Jill']=['5/12 8am','5/12 9am','5/12 10am','5/13 8am','5/13 9am', '5/13 10am']
      d['Jason']=['5/12 2pm','5/12 3pm','5/12 4pm','5/13 10am','5/13 2pm']
      d['Jenna']=['5/11 9pm','5/12 9am','5/12 3pm','5/12 9pm','5/13 12pm']
      d['Juan']=['5/12 2pm','5/12 3pm','5/13 2pm','5/13 3pm']
      best=getBestTimes(d)
      print('Best time slots:')
      for time in best:
          print(f'\t{time}\tAvailable:',best[time])
```

```
Best time slots:
        5/12 9am        Available: ['Jack', 'Jill', 'Jenna']
        5/13 2pm        Available: ['Jack', 'Jason', 'Juan']
        5/12 3pm        Available: ['Jason', 'Jenna', 'Juan']
```

Exercise 4.7: Longest Period without Rain

This question asks you to make a tool to help a roofing company plan its work given a forecast of whether it will rain in a period of consecutive days. Write a function called `analyzeForecast` with one input argument:

- `forecast`: a list of 1's and 0's indicating whether it will rain on a given day. Each 1 indicates that it will rain on that day and 0 indicates that it will not. The first entry of the list corresponds to Day 0, the second entry to Day 1, etc.

The function should return two numbers:

- `maxDryPeriod`: the longest number of consecutive days without rain in this given period.
- `startDay`: the first position in the list that begins a consecutive sequence of 0's of length `maxDryPeriod`. If `maxDryPeriod` is 0, then it doesn't matter what you return for `startDay`.

Sample run 1:

```
maxDryPeriod,startDay=analyzeForecast([1,1,0,0,0,0,1,1,0,0,1,0,0,0])
print(f'maxDryPeriod={maxDryPeriod} startDay={startDay}')
```

Correct output:

```
maxDryPeriod=4 startDay=2
```

Sample run 2:

```
maxDryPeriod,startDay=analyzeForecast([0,0,0,0,1,1,1,0,0,0,0,0,0,0])
print(f'maxDryPeriod={maxDryPeriod} startDay={startDay}')
```

Correct output:

```
maxDryPeriod=7 startDay=7
```

Sample run 3:

```
maxDryPeriod,startDay=analyzeForecast([0,1,0,0,1,1,1,0,0,0,1,0,0,0])
print(f'maxDryPeriod={maxDryPeriod} startDay={startDay}')
```

Correct output

```
maxDryPeriod=3 startDay=7
```

Sample Solutions:

One way of solving the problem is to keep track of the length of the current dry period as in the table below:

Day	Rain?	Length of Current Dry Period
0	1	0
1	1	0
2	0	1
3	0	2
4	0	3
5	0	4
6	1	0
7	1	0
8	0	1
9	0	2
10	1	0
11	0	1
12	0	2
13	0	3

In the above table, the first column correspond to the positions of the list. The second column is the corresponding value in the list `forecast`. The last column is 0 whenever the current day is wet, and increments by 1 whenever the current day is dry. The desired "maxDryPeriod" is the maximum entry in the last column, and the desired "startDay" is equal to `t-maxDryPeriod+1` where `t` is the first "Day" in which the last column attains the value "maxDryPeriod". In the above example, "maxDryPeriod" is 4 and "t" is 5, so "startPosition" is 2.

```
[5]: # Version for debugging, with with intermediate print statements
```

```
[42]: # Final code
```

```
[43]: maxDryPeriod,startDay=analyzeForecast([1,1,0,0,0,0,1,1,0,0,1,0,0,0])
      print(f'maxDryPeriod={maxDryPeriod} startDay={startDay}')
```

```
maxDryPeriod=4 startDay=2
```

```
[44]: maxDryPeriod,startDay=analyzeForecast([0,0,0,0,1,1,1,0,0,0,0,0,0,0])
      print(f'maxDryPeriod={maxDryPeriod} startDay={startDay}')
```

```
maxDryPeriod=7 startDay=7
```

```
[45]: maxDryPeriod,startDay=analyzeForecast([0,1,0,0,1,1,1,0,0,0,1,0,0,0])
      print(f'maxDryPeriod={maxDryPeriod} startDay={startDay}')
```

```
maxDryPeriod=3 startDay=7
```