AMERICAN UNIVERSITY OF BEIRUT

MAROUN SEMAAN FACULTY OF ENGINEERING & ARCHITECTURE

Department of Industrial Engineering & Management

INDE 431 – Production Planning and Inventory Control

Final Project

Stochastic Inventory Optimization of a Bakery

By:

Elie Tannoury, Marla Mirza, Ralph Mouawad,
MohamadAli Nachar, Agatha Choucair

To:

Dr. Hicham Abou Ibrahim

Table of Contents

Table of Figures

I – Introduction

This report delves into a production planning analysis conducted for a bakery located in Korea throughout 2020 and 2021. For our analysis, we analyzed factors such as production capacity, order fulfillment, labor costs, cost factors, and past data. Leveraging these considerations, we've developed a comprehensive plan encompassing several aspects:

Part 1 - Sales Forecast: The report includes a detailed sales forecast for both 2020 and 2021. By leveraging past sales data, trends, and seasonality, we have projected the expected sales volume over the forecasted period for every item.

Part 2 - Aggregate Planning (Number of Workers): The report outlines the required number of workers needed on a monthly basis to meet production demands. We have considered the processing capacity of workers. We also studied the feasibility of overlapping worker functions allowing for the efficient allocation of labor resources and cost savings.

Part 3 - Optimal Order Placement: The report presents an analysis that identifies the optimal timing for placing orders, determined the composition of each order through a Lot-Size Reorder Point System, taking into account demand for raw material, shortage penalty, and other detailed computations.

Part 4 - Estimated Total Cost: The report presents an estimation of the total costs associated with production for the years 2020 and 2021. We have considered labor costs, raw material holding costs, order set up cost, and penalty cost to provide a comprehensive overview of the expected expenditures.

By leveraging these insights, the bakery can make informed decisions regarding labor allocation, order placement, and cost management. The proposed plan will optimize production operations, boost profitability, and improve overall efficiency.

II – Sales Forecast

a. Loading Modules

To forecast sales, we first started by importing the necessary libraries and cleaning data by eliminating empty values that would otherwise lead to an inaccurate forecast.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.holtwinters import ExponentialSmoothing

#clean data to remove all empty values
BakerySales_df=BakerySales_df[:2420]
```

This was followed by resampling our data on a day to day basis rather than individual timestamp of sales, as the minimum forecast for planning will be one day.

```
# Sum the data of each day
```

```
daily_sales = BakerySales_df.resample("D").sum()
```

In order to forecast demand of every item using exponential smoothing, function variables relating to trend, seasonality, and seasonality period must be addressed. The most crucial variable is the seasonality period, as it is taken by standard to be 7 days.

```
forecast_model = ExponentialSmoothing(training[item], trend = None,
seasonal='add', seasonal_periods=Pl).fit()
```

$$MAD = \frac{\sum_{i=1}^{n} |e_i|}{n}$$

Figure 1: Mean Absolute Deviation Equation

b. Optimal Seasonality Period

In order to find the optimal seasonality period, we built multiple forecast models and compared them for least error. In specific, our approach entailed the following:

Step 1: Splitting training data & testing data

We split the given historical data into training data (80%) and testing data (20%) which is consistent with industry machine learning standards. The training data will be used to forecast the testing data period (start="2020-03-04", end="2020-05-02"). This data split results in 237 days of data points for the training data. This is important because the exponential smoothing algorithm requires that any seasonality cycle length must go through the training data provided at least twice. Accordingly, the maximum seasonality cycle length to be tested was 118 days.

```
#80% 20% of training to testing data to find optimal seasonality period
training = daily_sales[:237]
testing = daily_sales[237:]
```

Step 2: Iterating over seasonality periods and generating forecasts

Following the split, we iterated over seasonality periods and generated forecasts. “Lforc” constitutes the forecast values, and “Ltest” constates the true values to be tested against.

```
  for i in range (2, 119):
    Pl = i #period length

    forecast_model = ExponentialSmoothing(training[item], trend = None,
seasonal='add', seasonal_periods=Pl).fit()
    forecast_values = forecast_model.predict(start="2020-03-04",
end="2020-05-02") #testing dates

    Lforc = list(forecast_values)
    Ltest = list(testing[item])
```

Step 3: Mean Absolute Deviation for every seasonality period and selecting the optimal period

The Mean Absolute Deviation between the forecasts and true testing data can be calculated according to the below formula (Figure 1).

Accordingly, the MAD value for every forecast is computed and stored in a list “Lmad” previously initialized, such that Lmad[i] corresponds to the MAD value of seasonality period “i”.

```
    #compare accuracy of forecast values to testing data
    Lerror = [abs(Lforc[j] - Ltest[j]) for j in range (len(Lforc))]
    Lmad[i] = np.mean(Lerror)  #MAD
```

Step 4: Finding the minimum error seasonality period

After the list Lmad is completed with MAD values of every seasonality period “i", we can proceed to find the minimum value of the list. The index of the minimum value will be the optimal seasonality period for our forecast.

```
  m = minIndex(Lmad) #minimum MAD period
  (x, y) = (m, Lmad[m]) #x minimum MAD period, y MAD value for minimum MAD
period
  minIndexForecastItem[item] = (x, y) #store in dictionary
```

A function “minIndex(L)” was previously initialized to complete this operation.

```
def minIndex(L):
  m = L[2]
  mi = 2
  for i in range (2,len(L)):
    if L[i] < m:
      m = L[i]
      mi = i
  return mi
```

Step 5: Putting it all together

The steps 1 through 4 described previously help us find the optimal forecast seasonality for any item. Accordingly, to facilitate this process, we looped over all items sold and stored the values of the optimal forecast seasonality period in a dictionary “minIndexForecastItem.”

```
#80% 20% of training to testing data to find optimal seasonality period
training = daily_sales[:237]
testing = daily_sales[237:]


minIndexForecastItem = {} #dictionary

for item in Items:
```

```
  Lmad = [0]*(118+1)

  for i in range (2, 119):
    Pl = i #period length

    forecast_model = ExponentialSmoothing(training[item], trend = None,
seasonal='add', seasonal_periods=Pl).fit()
    forecast_values = forecast_model.predict(start="2020-03-04",
end="2020-05-02") #testing dates

    Lforc = list(forecast_values)
    Ltest = list(testing[item])

    #compare accuracy of forecast values to testing data
    Lerror = [abs(Lforc[j] - Ltest[j]) for j in range (len(Lforc))]
    Lmad[i] = np.mean(Lerror)  #MAD

  m = minIndex(Lmad) #minimum MAD period
  (x, y) = (m, Lmad[m]) #x minimum MAD period, y MAD value for x period
  minIndexForecastItem[item] = (x, y) #store in dictionary

print(minIndexForecastItem)
```

Accordingly, we can find the below values corresponding to a sample of the optimal seasonality resulting in the least MAD error for items sold at the bakery. Full results can be found on the python file.

Item	Optimal Seasonality Period	MAD
Chocolate Chip Cookie	63	4.800660347187381
Jam	28	0.7085646853685293
Iced Coffee	5	1.348700276943558
Croissant	21	1.9590902849504601
Iced Coffee Late	7	0.6915193634594601
Chocolate Croissant	35	1.467257699049322
Brownie	7	1.1381202705761864

c. Forecast and Plotting

Given that the optimal seasonality period for every item is found, we can proceed to forecasting the item sales over the period required (start="2020-05-03", end="2021-12-30"). Each forecast was then plotted on figure “i” in relation to the provided sales data for visualization.

```
figureNumber = 1
for key in minIndexForecastItem:
    item = key
    (m, Em) = minIndexForecastItem[key]
```

```
    plt.figure(figureNumber)
    i += 1

    forecast_model = ExponentialSmoothing(daily_sales[item], trend = None,
seasonal='add', seasonal_periods=m).fit()
    forecast_values = forecast_model.predict(start="2020-05-03",
end="2021-12-30")

    daily_sales[item].plot(legend = True, label = "Historical Data")
    forecast_values.plot(legend=True, label='Forecast', color='red')

    plt.title(item+" Forecast")
    plt.xlabel('Date')
    plt.ylabel('Sales')

plt.legend()
plt.show()
```

We can see below the forecast of “Chocolate Chip Cookie” and “Lemon cake” (Figures 2, 3). Please refer to the python file for the plot figures of every forecasted item.

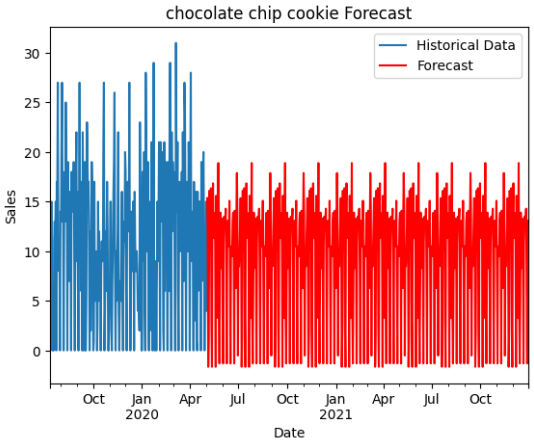

Figure 2: Chocolate Chip Cookie Forecast

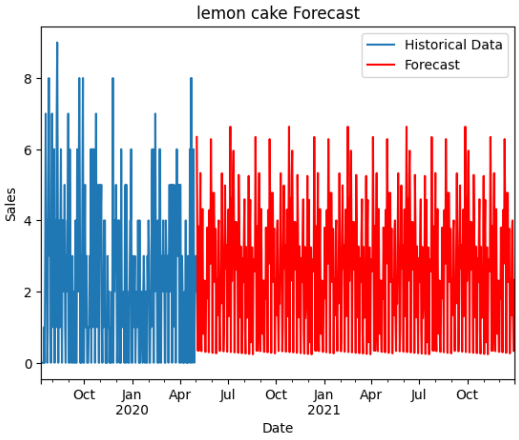

Figure 3: : Lemon Cake Forecast

d. Extracting the Forecast Data & Cleaning

Following these forecasts, we wanted to compile the obtained values into a csv file. This was challenging because the forecast data was segregated for each item. To this end, we created a class “datetime” that allows us to generate and increment the dates automatically for every forecast value to be stored.

```
class datetime (object):

    def __init__(self, y, m, d):
```

```
        self.y = y
        self.m = m
        self.d = d

    def incr (self):
        D = {"1":"31", "2":"28", "3":"31", "4":"30", "5":"31", "6":"30",
"7":"31", "8":"31", "9":"30", "10":"31", "11": "30", "12":"31"}
        if self.y %4 == 0:
            D["02"] = "29"

        if self.d < int(D[str(self.m)]):
            self.d += 1
        elif self.d == int(D[str(self.m)]) and self.m < 12:
            self.m += 1
            self.d = 1
        elif self.d == int(D[str(self.m)]) and self.m == 12:
            self.y += 1
            self.m = 1
            self.d = 1

    def __str__(self):

        sy = str(self.y)
        sm = str(self.m)
        if len(sm) == 1:
            sm = "0" + sm
        sd = str(self.d)
        if len(sd) == 1:
            sd = "0" + sd
        return sy +"-" + sm + "-" + sd
```

Following this, the csv file of the compiled data was generated according to the below code. The main writing procedure for the csv file starts at “NH = open("forecastV.csv", 'w').” All prior lines of code are preparatory.

```
D = {}
Ldate = []
DT = datetime(2020,5,3)
while str(DT) != "2022-01-01":
    Ldate.append(str(DT))
    DT.incr()

D["datetime"] = Ldate

for key in minIndexForecastItem:
```

```
    item = key
    (m, Em) = minIndexForecastItem[key]

    forecast_model = ExponentialSmoothing(daily_sales[item], trend = None,
seasonal='add', seasonal_periods=m).fit()
    forecast_values = forecast_model.predict(start="2020-05-03",
end="2021-12-31")

    D[item] = list(forecast_values)

NH = open("forecastV.csv", 'w')

s = ""
for key in D:
    s += key + ","
NH.write(s[:-1])

NH.write("\n")
for i in range (len(Ldate)):
    s = ""
    for key in D:
        s += str(D[key][i]) + ","
    NH.write(s[:-1])
    NH.write("\n")
NH.close()
```

Once this csv file for the forecast data was created, a final cleaning was done by:

- Replacing all negative values with 0
- Rounding all non-integer values

```
Forecast_df = pd.read_csv('forecastV.csv')
Forecast_df['datetime'] = pd.to_datetime(Forecast_df['datetime'])
Forecast_df.set_index('datetime', inplace=True)

#cleaning forecast to remove negative values and round non integers
Items = list(Forecast_df.columns)
for item in Items:
  for i in range (len(Forecast_df[item])):
    #removing negatives
    if Forecast_df[item][i] < 0:
      Forecast_df[item][i] = 0
    Forecast_df[item][i] = round(Forecast_df[item][i])

from copy import deepcopy
```

```
FC_cleaned = deepcopy(Forecast_df) #deepcopy to avoid accidental changes
in data because of alias
FC_cleaned.to_csv("FC_cleaned.csv")
```

Below is a snapshot of the final forecast data which was used in the next parts of this report (Figure 4). The complete daily forecast of items sold can be found in *"Forecast_Items_Daily_Cleaned.csv"*.

datetime	chocolate	plain breac	jam	iced coffee	croissant	iced coffee	chocolate	brownie	lemon cak	chocolate
5/3/2020	15	6	2	2	3	1	6	3	6	1
5/4/2020	11	3	1	2	3	1	4	2	4	1
5/5/2020	0	0	0	2	0	0	1	0	0	0
5/6/2020	2	3	1	2	3	1	4	2	4	2
5/7/2020	16	4	1	1	3	1	5	1	0	1
5/8/2020	12	3	1	2	5	1	3	1	2	1
5/9/2020	10	3	1	2	4	1	3	2	3	1
5/10/2020	16	6	1	2	5	1	5	3	5	1
5/11/2020	3	2	1	2	5	1	4	2	1	1
5/12/2020	0	0	0	1	0	0	1	0	0	0
5/13/2020	14	4	1	2	4	1	3	2	3	2
5/14/2020	17	5	1	2	5	1	4	1	4	1
5/15/2020	11	4	0	2	4	1	3	1	2	1
5/16/2020	12	5	1	2	5	1	6	2	1	1
5/17/2020	14	6	1	1	7	1	5	3	2	1

Figure 4: Item Forecast Snapshot 5/03/2020 - 5/17/2020

III – Aggregate Planning (Number of Workers)

a. Problem Formulation

The main approach here was to formulate a linear optimization problem to be able to find the number of workers needed every month that would minimize total costs.

After forecasting our products, we segregated them into three categories: Pastry, Drink and Delivery. Then, we would resample the forecasted demand monthly. We formulated the linear program below to describe our problem.

{i, j, k} = {pastry, drink, delivery};

Parameters:

- D_{ti}, D_{tj}, D_{tk} – Forecasted demand of pastry (i), drink (j), and delivery (k) respectively at month 't'
- C_W = 750,000 won – Cost of hiring one worker
- K_i = 30 pastries processed by one worker daily
- K_j = 40 drinks processed by one worker daily
- K_k = 20 deliveries processed by one worker daily
- n = 30 days (per month approximately)
- T = 20 months (horizon)

Decision Variables:

- W_{ti}, W_{tj}, W_{tk} – Number of workers hired at month 't' to process respectively pastries (i), drinks (j) and deliveries (k).
- P_{ti}, P_{tj}, P_{tk} – Production level at month 't' of pastries (i), drinks (j) and deliveries (k).

Objective Function:

Minimize $\Sigma_{t=1}^{20} C_W * (W_{ti} + W_{tj} + W_{tk})$

Subject To:

- Constraint 1: $P_{ti} = K_i * n * W_{ti}$
- Constraint 2: $P_{tj} = K_j * n * W_{tj}$
- Constraint 3: $P_{tk} = K_k * n * W_{tk}$
- Constraint 4: $P_{ti} >= D_{ti}$
- Constraint 5: $P_{tj} >= D_{tj}$
- Constraint 6: $P_{tk} >= D_{tk}$

b. Solving the LP using Python (pulp)

We have implemented the following Linear Program on Python. First, we initialized the constants and decision variables.

```
df = pd.read_csv('DemandDPDMonthly.csv', index_col = 'datetime')

import pulp as pl
D_pastries = list(df['Pastries'])
D_drinks = list(df['Drinks'])
D_delivery = list(df['Delivery'])
hiring = 750000 # Cost of hiring one worker
K_pastries = 30
K_drinks = 40
K_delivery = 20
n = 30
T = 20
W_pastries = pl.LpVariable.dicts('W_pastries', range(0,T), lowBound = 0)#, cat =
"Integer")
W_drinks = pl.LpVariable.dicts('W_drinks', range(0,T), lowBound = 0)#, cat =
"Integer")
W_delivery = pl.LpVariable.dicts('W_delivery', range(0,T), lowBound = 0)#, cat =
"Integer")
P_pastries = pl.LpVariable.dicts('P_pastries', range(0,T), lowBound =0)
P_drinks = pl.LpVariable.dicts('P_drinks', range(0,T), lowBound =0)
P_delivery = pl.LpVariable.dicts('P_delivery', range(0,T), lowBound =0)
```

Then, we defined the model, objective function, and constraints.

```
# Define the Model
model = pl.LpProblem('Aggregate Planning', pl.LpMinimize)

# Objective Function
model += hiring*pl.lpSum(W_pastries[t] for t in range(0,T)) +
hiring*pl.lpSum(W_drinks[t] for t in range(0,T)) + hiring*pl.lpSum(W_delivery[t]
# Constraints
for t in range(0,T):
  model += P_pastries[t] == K_pastries*n*W_pastries[t]
  model += P_drinks[t] == K_drinks*n*W_drinks[t]
  model += P_delivery[t] == K_delivery*n*W_delivery[t]
  model += P_pastries[t] >= D_pastries[t]
  model += P_drinks[t] >= D_drinks[t]
  model += P_delivery[t] >= D_delivery[t]
```

Lastly, we solved the model as follows:

```
# Solve the Model
status = model.solve()
print('Number of Workers', pl.values(model.objective))

for v in model.variables():
  print(v.name, v.varValue)
```

c. Analysis of Solution

While analyzing our solution, we found that solving the problem while setting the number of workers for pastries, drinks, and delivery, each at an integer constraint, would result in a requirement of 4 workers (2 for pastries, 1 for drinks, 1 for delivery). However, upon closer observation, we tested the model by solving for a continuous, non-integer, number of workers. A sample of the solution to the non-integer problem can be shown below (Table 2). The full LP continuous variable solution can be found on the python file.

Datetime	Delivery Workers	Drinks Workers	Pastries Workers
5/31/2020	0.52	0.07	1.08
6/30/2020	0.52	0.07	1.10
7/31/2020	0.55	0.07	1.14
8/31/2020	0.56	0.07	1.18
9/30/2020	0.51	0.07	1.05

We can notice that 1 worker can complete more the one task function. The highest cost-saving strategy would be to hire 2 workers that cover pastries, drinks, and delivery. A more structured job allocation would be to split the workers at the store from the worker on delivery, and that would result in 3 workers in total. For the purposes of cost-saving, we will assume that the 2-worker strategy is implemented. Accordingly, 2 workers are required in the bakery during 2020 and 2021.

IV – Optimal Order Placement

We opted to estimate the optimal order placement quantity and time through the Lot-Size Reorder Point System. This is a a stochastic demand problem with allowed shortages. The goal of this section is three-fold:

a. Compute the daily raw material requirement for 2020 and 2021 based on the forecasted sales
b. Prepare the variables and constants necessary for the Lot-Size Reorder System calculations
c. Iterate over the equations of the Lot-Size Reorder System to estimate the optimal order quantity

The following equations are used to find the optimal order quantities, as consistent with our system of choice (Figures 5, 6).

$$Q = \sqrt{\frac{2\lambda[K + pn(R)]}{h}}.$$

Figure 5: Lot Size Reorder System - Equation 1

$$1 - F(R) = Qh/p\lambda$$

Figure 6: Lot Size Reorder System - Equation 2

Considering the equations above, in order to successfully implement the Lot Size Reorder System, we require knowledge of the mean demand for the planning period (lambda), holding costs (h), and penalty costs (p). Holding costs are straightforward to compute as raw material prices are given. We will proceed to formulate programs to compute mean raw material demand and estimate penalty price (p) in an educated manner.

a. Computing Raw Material Needs Over Forecasted Period

As we have previously forecasted the item sales over 2020 and 2021 (Section II - d), we can forecast the raw material needed by linking every item sold to its raw material components. We are given the raw material requirement to produce every item at the bakery (Figure 7). The complete raw material requirements for all items can be found in *"Data_Raw Material.csv"*.

Name	price	Sugar (in g	Chocolate	Dough (in g	Eggs (in un	Butter (in g	Milk (in L pe	Coffee (in g	Category
chocolate	1500	40	30	0	0.5	30	0	0	pastry
plain breac	3500	10	0	30	0	0	0	0	pastry
jam	1500	30	0	0	0	0	0	0	pastry
iced coffee	4000	0	0	0	0	0	0	10	drink
croissant	3500	0	0	30	0.5	80	0	0	pastry
iced coffee	4500	5	0	0	0	0	0.1	10	drink

Figure 7: Sample of Raw Material Requirement per Item

Accordingly, we can combine both datasets to compute the raw material needed on every given day of the forecasting period to fulfill the forecasted demand. This was implemented over python according to the following steps.

Step 1: Creating a dictionary containing every item at the bakery with raw material requirement

```
RawMaterial_df = pd.read_csv('Raw Material.csv')
RawMaterial_df.set_index('Name', inplace=True)

RM_df = RawMaterial_df.transpose()

RM_dict = dict(RM_df)
for key in RM_dict:
  RM_dict[key] = list(RM_dict[key])
print(RM_dict)
```

Step 2: Multiplying the item sales forecast with the RM requirements and setting up CSV file

```
Items = list(Forecast_df.columns)[:21]
RMList = list(RM_df.index)[1:-1]
Dates = list(Forecast_df.index)

SugarReqL = [0]*len(Dates)
ChocolateReqL = [0]*len(Dates)
DoughReqL = [0]*len(Dates)
EggsReqL = [0]*len(Dates)
ButterReqL = [0]*len(Dates)
MilkReqL = [0]*len(Dates)
CoffeeReqL = [0]*len(Dates)

for i in range (len(Dates)):

  dateSugarReq = 0
  dateChocolateReq = 0
  dateDoughReq = 0
  dateEggsReq = 0
```

```
  dateButterReq = 0
  dateMilkReq = 0
  dateCoffeeReq = 0

  for item in Items:

    itemDemand = FC_cleaned[item][i]

    itemSugarReq = itemDemand * RM_dict[item][1]
    itemChocolateReq = itemDemand * RM_dict[item][2]
    itemDoughReq = itemDemand * RM_dict[item][3]
    itemEggsReq = itemDemand * RM_dict[item][4]
    itemButterReq = itemDemand * RM_dict[item][5]
    itemMilkReq = itemDemand * RM_dict[item][6]
    itemCoffeeReq = itemDemand * RM_dict[item][7]

    dateSugarReq += itemSugarReq
    dateChocolateReq += itemChocolateReq
    dateDoughReq += itemDoughReq
    dateEggsReq += itemEggsReq
    dateButterReq += itemButterReq
    dateMilkReq += itemMilkReq
    dateCoffeeReq += itemCoffeeReq

  SugarReqL[i] = dateSugarReq
  ChocolateReqL[i] = dateChocolateReq
  DoughReqL[i] = dateDoughReq
  EggsReqL[i] = dateEggsReq
  ButterReqL[i] = dateButterReq
  MilkReqL[i] = dateMilkReq
  CoffeeReqL[i] = dateCoffeeReq

NH = open("RMperDate.csv", 'w')

s = "datetime"
for RMi in RMList:
  s += "," + RMi

NH.write(s + "\n")

for i in range (len(Dates)):
  s = str(Dates[i])
  s += "," + str(SugarReqL[i])
  s += "," + str(ChocolateReqL[i])
  s += "," + str(DoughReqL[i])
```

```
  s += "," + str(EggsReqL[i])
  s += "," + str(ButterReqL[i])
  s += "," + str(MilkReqL[i])
  s += "," + str(CoffeeReqL[i]) +"\n"
  NH.write(s)
NH.close()
```

At this stage, we have succeeded at creating a CSV file containing the forecasts of the raw material needs over 2020 and 2021 (Figure 8). The full forecast of raw material needs on a daily basis can be found in *"Forecast_Raw Material_Daily.csv"*.

datetime	Sugar (in g per unit)	Chocolate (in g per unit)	Dough (in g per unit)	Eggs (in units per unit)	Butter (in g per unit)	Milk (in L per unit)	Coffee (in g per unit)
5/3/2020 0:00	1140	630	680	18.45	1566	0.4	30
5/4/2020 0:00	785	460	450	13.45	1131	0.3	30
5/5/2020 0:00	90	50	60	1.95	101	0	30
5/6/2020 0:00	460	225	490	9.45	921	0.3	30
5/7/2020 0:00	890	605	450	14.45	1216	0.2	20
5/8/2020 0:00	800	465	480	14.2	1196	0.2	40
5/9/2020 0:00	805	435	515	13.95	1166	0.3	30
5/10/2020 0:00	1100	650	660	18.7	1616	0.4	40
5/11/2020 0:00	405	205	450	10.2	961	0.3	50
5/12/2020 0:00	65	35	40	1.2	96	0	20

Figure 8: Sample Forecast of Daily Raw Material Needs

This CSV file of the raw material needs per specified is used in subsequent stages and concludes the first step towards computing a Lot-Size Reorder Point System.

```
#uploading output into a dataframe to work with
RMperDate_df = pd.read_csv('RMperDate.csv')
RMperDate_df['datetime'] = pd.to_datetime(RMperDate_df['datetime'])
RMperDate_df.set_index('datetime', inplace=True)

#we can now compute the mean demand per day of every item
print("Mean demand of sugar per day in g:", np.mean(RMperDate_df["Sugar
(in g per unit)"]))
```

b. Estimating the Shortage Penalty Cost (p)

In order to find optimal order quantity in a stochastic demand problem with allowed shortages, we must first find the value of penalty cost p. This can be estimated by the loss of profit per gram or unit of raw material. Accordingly, we must estimate the revenue per gram of raw material type and cost per gram of raw material type. The costs per gram or unit of raw material is straight forward and can be computed from the data provided (Figure 9). The complete prices list of raw material can be found in *"Data_Bakery Prices.csv"*.

	Price (in $)	Price (in wons)
Sugar (per kg)	3	3948
Chocolate (per kg)	5	6580
Dough (per kg)	2	2632
Eggs (per unit)	0.5	658
Butter (per kg)	5	6580
Milk (per L)	2	2632
Coffee (per Kg)	15	19740

Figure 9: Cost of Raw Material per Unit

The challenge in this problem is to estimate the revenue per raw material unit. This can be done by "allocating" the revenues of items sold at the bakery to its raw material components.

As point of example, we can take the case of the chocolate chip cookie (Figure 10). The complete raw material requirements for all items can be found in *"Data_Raw Material.csv"*.

Name	price	Sugar (in g	Chocolate	Dough (in g	Eggs (in un	Butter (in g
chocolate chip cookie	1500	40	30	0	0.5	30

Figure 10: Chocolate Chip Cookie Raw Material Revenue Allocation

Considering the revenue generated by this item to be 1,500 Wons, we want to allocate every part of this revenue to its raw material components. For example, we can see that a chocolate chip cookie requires 40g of Sugar, 30g of Chocolate, 0.5 Eggs, and 30g of Butter.

A simple way of doing this is to sum the total units (40g + 30g + 0.5 units + 30g = 100.5), and get the proportion of contribution of sugar (40/100.5), chocolate (30/100.5), eggs(0.5/100.5), and butter (30/100.5).

Following this example, the revenue allocation of sugar would be: 1,500 Wons * 40 / 100.5 = 597 Wons.

However, it can be clear that this would result in a false estimate because of the inconsistency of units used (1 egg unit is not equivalent to 1 gram of sugar). To this end, we intend to establish a common monetary unit for every raw material type to estimate the contribution weight towards creating the item at the bakery. Once this contribution weight is found, we can find the revenue allocation of the item sold to its raw material components in the same manner above.

Step 1: Importing prices of raw material

```
RMCost_df = pd.read_csv("Bakery Price.csv")

LpricesofRM = list(RMCost_df["Price (in wons)"])
for i in {0,1,2,4,6}:
  LpricesofRM[i] = LpricesofRM[i]/1000 #converting prices from kg to g
print(LpricesofRM)

#Adding the values to a dictionary
```

```
RMCostD = {"Sugar (in g per unit)": 3.948, "Chocolate (in g per unit)":
6.58, "Dough (in g per unit)": 2.632, "Eggs (in units per unit)": 658,
           "Butter (in g per unit)": 6.58, "Milk (in L per unit)": 2632,
"Coffee (in g per unit)": 19.74}
```

Step 2: Multiplying number of units of each raw material component of an item by the price per unit of the raw material component to establish a common monetary value

```
RMinWons_df = deepcopy(RawMaterial_df)
RMList = list(RMinWons_df.columns)[1:-1]
for RMi in RMList:
  for i in range (21+1):
    RMinWons_df[RMi][i] = RMinWons_df[RMi][i] * RMCostD[RMi]
```

Accordingly, we obtain the following costs of raw material components (in Wons) used to produce each item.

	Sugar (in Wons per unit)	Chocolate (in Wons per unit)	Dough (in Wons per unit)	Eggs (in Wons per unit)	Butter (in Wons per unit)	Milk (in Wons per unit)	Coffee (Wons p Unit)
chocolate chip cookie	157.92	197.4	0.00	329.0	197.4	0.0	0.0
plain bread	39.48	0.0	78.96	0.0	0.0	0.0	0.0
jam	118.44	0.0	0.00	0.0	0.0	0.0	0.0
iced coffee	0.00	0.0	0.00	0.0	0.0	0.0	197.4
croissant	0.00	0.0	78.96	329.0	526.4	0.0	0.0

Step 3: Calculate the revenue allocation of sold item to raw material type

```
RM_RevenueAllocation_df = deepcopy(RMinWons_df).transpose()
Items = list(RM_RevenueAllocation_df.columns)[:-1]

for item in Items:
  Litem = list(RM_RevenueAllocation_df[item])
  totalWons = sum(Litem[1:-1])


  for i in range (1, 7+1):
    RM_RevenueAllocation_df[item][i]=Litem[0]*RM_RevenueAllocation_df[item
][i]/totalWons
```

The formula in the last line of the above code multiplies the price of sale of the item under consideration by the cost of raw material used over the total cost of raw material for that item.

For example, in the case of the chocolate chip cookie, the price of sale is 1,500 Wons, the cost of sugar used to produce one cookie is 157.92 Wons, and the total cost of raw material is (157.92 + 197.4 + 329.0 + 197.4 = 881.72 Wons). In this case, the revenue allocation for the sugar would be:

Price of cookie * (cost of sugar / cost of all raw material) = 1,500 * 157.92 / 881.72 = 268.7 Wons

Step 4: Calculate the revenue allocation of sold item to raw material gram or unit

Now that we have the revenue allocation per raw material type, we can find the revenue allocation of each item per raw material gram. This is done by dividing the previous revenue allocation of item sold per raw material type to the number of grams or units of raw material used.

```
RM_RevAll_T_df = RM_RevenueAllocation_df.transpose()

Items = list(RM_RevAll_T_df.index[:21])
RML = list(RM_RevAll_T_df.columns[1:8])

for RMi in RML:
  for i in range (len(Items)):
    if RawMaterial_df[RMi][i] != 0:
      RM_RevAll_T_df[RMi][i] =
(float(RM_RevAll_T_df[RMi][i])/float(RawMaterial_df[RMi][i]))
       #Revenue Allocation per RM type/ units of RM used
```

Taking the previous example, the revenue allocation of the chocolate chip cookie to the sugar used was 268.7 Wons. We are now dividing by the number of grams of sugar used (40 grams) to get the revenue allocation of the cookie per sugar gram.

Accordingly, we obtain the following revenue allocations (in Wons) of each item to the raw material used per gram.

	Sugar (in Wons per unit)	Chocolate (in Wons per unit)	Dough (in Wons per unit)	Eggs (in Wons per unit)	Butter (in Wons per unit)	Milk (in Wons per unit)	Coffee (Wons p Unit)
chocolate chip cookie	6.716418	11.19403	0.0	1119.402985	11.19403	0.0	0.0
plain bread	116.666667	0.0	77.777778	0.0	0.0	0.0	0.0
jam	50.0	0.0	0.0	0.0	0.0	0.0	0.0
iced coffee	0.0	0.0	0.0	0.0	0.0	0.0	400.0
croissant	0.0	0.0	9.859155	2464.788732	24.647887	0.0	0.0

This gives us an estimate of the value of every gram or unit of raw material used. However, this estimate is not constant and varies from item to item. For this reason, we can proceed to find the expected value of

the revenue of the raw material per gram by computing a weighted overage using the probability of demand. This requires us to compute the probability that a customer coming into the store orders any item from the data forecasted.

That is, if chocolate chip cookie orders constitute $1/5^{th}$ of total orders, the corresponding weight of the sugar revenue would be 1/5 * 6.716418.

Step 5: Finding the demand probability of each item sold

This is done by dividing the sum of demand of an item over the forecasted period over the total demand of all items of the forecasted period.

```
#Finding the demand probability of each item
Items = list(FC_cleaned.columns)[:21]

TotalDemandperItemD = {}
TotalDemandVal = 0

for item in Items:

  S = sum(list(FC_cleaned[item]))
  TotalDemandperItemD[item] = S
  TotalDemandVal += S

WeightedDemandperItemD = {}

for item in TotalDemandperItemD:
  WeightedDemandperItemD[item] = TotalDemandperItemD[item]/TotalDemandVal


DemandWeightL = []
for key in WeightedDemandperItemD:
  DemandWeightL.append(WeightedDemandperItemD[key])

print("DemandWeightL:", DemandWeightL)
```

The corresponding demand weight is the following: [0.2661745151084415, 0.09817048007703241, …, 0.0032555367050300333]. We can see that the chocolate chip cookie is ordered 26.6% of the time.

Step 6: Finding the expected value of the revenue allocation of each raw material

Now that we have the demand probability of each item and the revenue allocation of each raw material per item, we can multiply both by a weighted average to obtain the expected value of revenue allocation per gram of raw material.

```
ExpRMRevD = {}
ExpRMRev_L = []

for RMi in RML:
  LRevAllocation = list(RM_RevAll_T_df[RMi])[:-1]

  WeightedSum = 0
  for i in range (len(LRevAllocation)):

    WeightedSum += DemandWeightL[i] * LRevAllocation[i]

  ExpRMRevD[RMi] = WeightedSum
  ExpRMRev_L.append(WeightedSum)

print("Revenue Allocation:", ExpRMRev_L)
```

Output:

Revenue Allocation: [30.865601481963523, 20.252688244671393, 17.324050099042502, 3134.3043060883506, 29.953427652154097, 1168.0808159942899, 28.415908776427752]

In addition to this revenue allocation, we can add the 3,000 Wons in revenue from delivery 40% of the time. This translates to an expected delivery revenue of 1,2000 Wons. However, adding this would over estimate the penalty on the items because we are not accounting for the delivery cost. Further information on the delivery cost should be provided in order to proceed with that step.

The next step is to convert from revenue allocation per type of raw material to revenue allocation per gram of raw material.

Step 7: Compute the shortage penalty by taking revenue per gram - cost per gram

At this point, this is only the revenue allocation, so we need to deduct the cost of the item in Won.

```
P_L = list(RMCost_df['Price (in wons)']) #price of raw material
for i in {0,1,2,4,6}:
  P_L[i] = P_L[i]/1000 #kg price to g price

print("prices of RM:", P_L)

Penalty_L = [0]*7
for i in range (len(Penalty_L)):
  Penalty_L[i] = ExpRMRev_L[i] - P_L[i]

  if Penalty_L[i] < 0:
  #setting negative penalty value to 0
    Penalty_L[i] = 0
```

```
print("Penalty: ", Penalty_L)
```

Output:

Penalty: [26.917601481963523, 13.672688244671393, 14.692050099042502, 2476.3043060883506, 23.3734276521541, 0, 8.675908776427754]

Notice that the penalty on milk shortage is set to zero because the revenue allocation (1168) is smaller than the price (2632) per Liter. This will be subsequently solved by establishing optimal order estimates for the milk using standard EOQ with no shortages.

c. Lot-Size Reorder System Formulation

Now that we have estimated shortage penalty (p) for every raw material, we can proceed to formula the Lot-Size Reorder System problem with the formulas above (Figure 5, 6).

Step 1: Initialization

To initialize the problem, we must start with the EOQ equivalent, Q0 = (2DK/h)**0.5. Accordingly, we need the values of demand (D), ordering set up cost (K), and holding costs (h).

Demand is computed in D_L by taking the mean of the previously forecasted raw material need per day, and multiplying that by 30 days per month. K is given by our client as 131,000 Wons per order. Holding cost is calculated by taking the cost of the raw material given, multiplying by the holding interest rate (10%), and dividing over 12 months per year.

```
#Initializing the problem formulation
#Q0 = EOQ = (2DK/h)**0.5
K = 131000
hpercentage = 0.1

D_L = [0]*7 #month demand value of raw material
for i in range(len(RML)):
  dailyDemandVal = np.mean(list(RMperDate_df[RML[i]]))
  D_L[i] = dailyDemandVal * 30

P_L = list(RMCost_df['Price (in wons)']) #price of raw material
for i in {0,1,2,4,6}:
  P_L[i] = P_L[i]/1000 #kg price to g price

H_L = [0]*7 #holding cost of raw material
for i in range (len(H_L)):
  H_L[i] = P_L[i]*hpercentage/12 #per month

Q0_L = [0]*7
for i in range (len(Q0_L)):
  Q0_L[i] = (2*D_L[i]*K/H_L[i])**0.5
```

```
print("Q0_L:", Q0_L)
```

Output:

Q0_L: [406784.56732099154, 240258.26968954748, 401753.4101555188, 4168.230732146455, 393468.60560027545, 294.9513190258772, 40284.36852110824]

Step 2: Setting up the iterations with F(R), Z Value, Standard of Deviation, R, Lz, and nR

The equations corresponding to the Lot-Size Reorder Point System have been implemented below.

```
#Calculating F(R), z, std, R, Lz, and nR

#Computing F_L and Z_L
import scipy.stats as st

Q_L = Q0_L.copy()
F_L = [0]*7

for i in range (7):
  F_L[i] = 1 - (Q_L[i]*H_L[i])/(Penalty_L[i]*D_L[i])

Z_L = st.norm.ppf(F_L)

print("F_L:", F_L)
print("Z_L:", Z_L)

#solve for R, need s and u, s calculated below, u is D_L
import statistics
Std_L = [0]*7

for i in range(len(RML)):
  L = list(RMperDate_df[RML[i]])
  Std_L[i] = statistics.stdev(L)

print("Std_L:",Std_L)

R_L = [0]*7
for i in range (7):
  R_L[i] = Z_L[i]*Std_L[i]+D_L[i]

print("R_L:", R_L)

#Solve for Lz (standardized loss function)
```

```
import math
def Lz(x): #defining the standardized loss function
  return math.exp(-(x**2)/2)/(math.sqrt(2*math.pi)) - x*(1-st.norm.cdf(x))

Lz_L = [0]*7

for i in range(7):
  Lz_L[i] = Lz(Z_L[i])

print("Lz_L:", Lz_L)

#solve for nR
nR_L = [0]*7
for i in range (7):
  nR_L[i] = Std_L[i]*Lz_L[i]

print("nR_L:", nR_L)
```

Output:

F_L: [0.9760723270475464, 0.9202429586918099, 0.9556126396546515, 0.9746168493599947, 0.9715115513713414, -inf, 0.2503651746020018]

Z_L: [1.97865071 , 1.4067077, 1.70189677, 1.95344996, 1.90348801, nan, -0.67334104]

Std_L: [312.13879150560086, 192.32911418814837, 185.760276132747, 5.184365904812698, 454.2880026029078, 0.11797409800555984, 10.26399506242397]

R_L: [21396.561010245696, 12351.471897797324, 13828.233630431097, 373.7466427872678, 33266.11334317473, nan, 1012.0033045757449]

Lz_L: [0.008988882613239546, 0.036129809384467554, 0.018203429088170034, 0.009610066095749423, 0.01095458585755621, nan, 0.8227825622930712]

nR_L: [2.805778955882299, 6.9488142347012944, 3.3814740139813444, 0.04982209900979979, 4.976536928571273, nan, 8.445036156824624]

In consistency with the previous null value of the penalty of milk shortage, we find the above computations for the milk raw material not applicable, “nan.”

Step 3: Repeat computation iterations until the difference between two subsequent Q values are close

The formulas corresponding to the Lot-Size Reorder Point System are implemented below. Every iteration is shown in the output to monitor the algorithm. In this code, even though the Q values are stored in a list, every Q value is iterated over independently from the other.

```
#Repeating until difference between Qprev and Qnew is less than 1 unit

Qprev_L = Q_L.copy()
```

```
print("Q0:", Qprev_L)

Qnew_L = [0]*7

c = 1
for i in {0,1,2,3,4,6}:

  while True:
    Qnew_L[i] = ((2*D_L[i]*(K+Penalty_L[i]*nR_L[i]))/(H_L[i]))**0.5

    print("Iteration "+str(c)+":",Qnew_L)
    c+=1

    if abs(Qprev_L[i] - Qnew_L[i]) < 1:
      break

    F_L[i] = 1 - (Qnew_L[i]*H_L[i])/(Penalty_L[i]*D_L[i])
    Z_L = st.norm.ppf(F_L)
    Lz_L[i] = Lz(Z_L[i])
    nR_L[i] = Std_L[i]*Lz_L[i]

    Qprev_L[i] = Qnew_L[i]
```

Output:

Q0: [406784.56732099154, 240258.26968954748, 401753.4101555188, 4168.230732146455, 393468.60560027545, 294.9513190258772, 40284.36852110824]

Iteration 1: [406901.8494768749, 0, 0, 0, 0, 0, 0]

Iteration 2: [406901.84947687894, 0, 0, 0, 0, 0, 0]

Iteration 3: [406901.84947687894, 240345.41617293484, 0, 0, 0, 0, 0]

Iteration 4: [406901.84947687894, 240345.41617294177, 0, 0, 0, 0, 0]

…

Iteration 11: [406901.84947687894, 240345.41617294177, 401829.60062302137, 4170.194118937307, 393643.3411120839, 0, 40295.63924927513]

Iteration 12: [406901.84947687894, 240345.41617294177, 401829.60062302137, 4170.194118937307, 393643.3411120839, 0, 40295.63924927758]

We observe that the total iterations on the Q values are 12, with 2 iterations per item (6 items excluding milk). Since the final Q value for milk cannot be computed using this algorithm due to the absence of penalty value, we can take it as Q0.

d. Analysis of Results

Accordingly, we can compute the order quantities, reorder point values, and the cycle time through the formulas of the system.

```
D_L = np.array([20779, 12081, 13512, 364, 32401, 7.3, 1019])
Q_L = np.array([406902, 240345, 401830, 4170, 393643, 295, 40296])

Flast_L = F_L.copy()

Zlast_L = st.norm.ppf(Flast_L)

Rlast_L = [0]*7
for i in range (7):
  Rlast_L[i] = Zlast_L[i]*Std_L[i]+D_L[i]

Rlast_L[5] = D_L[5] #milk

print("Reorder Quantity:", Rlast_L)

T_L = Q_L/D_L
print("T_L:",list(T_L))
```

Output:

Reorder Quantity: [21396.575421252324, 12351.51333813556, 13828.128135769357, 374.1263524155697, 33265.64360592397, 7.3, 1012.0820604030943]

T_L: [19.582366812647383, 19.89446237894214, 29.73875074008289, 11.456043956043956, 12.14910033640937, 40.41095890410959, 39.54465161923454]

Note that the values of order time are very close to 20, 20, 30, 10, 10, 40, 40 months. This is attractive because it would allow overlap in order time, reducing order cost. Because these values are large, we are obliged to assume that the items are not perishable.

We obtain the final values of Order Quantity, Reorder Point, and Cycle Time below (Figure 11). These values can be seen in depth in *"Analysis_Optimal Order Quantity.xlsx"*.

	Sugar Demand	Chocolate Demand	Dough Demand	Eggs Demand	Butter Demand	Milk Demand	Coffee Demand
Order Quantity	406902	240345	401830	4170	393643	295	40296
Reorder Time	20	20	30	10	10	40	40
Reorder Quantity	21396.57542	12351.51334	13828.12814	374.1263524	33265.64361	nan	1012.08206

Figure 11: Order Quantities, Reorder Point Values, And Cycle Time of Raw Material

e. When to Place the Orders and the Content of Every Order

We now have all the necessary information to specify when the orders should be placed along the contents of every order.

We are aware that the lead time from when an order is placed to when the raw material is received is 1 month. Accordingly, the first order will be placed in April of 2020, to be received during May of 2020. This order will constitute all raw materials at their optimal order quantity.

datetime	Sugar Demand	Chocolate Demand	Dough Demand	Eggs Demand	Butter Demand	Milk Demand	Coffee Demand	Sugar Ordered	Chocolate Ordered	Dough Ordered	Eggs Ordered	Butter Ordered	Milk Ordered	Coffee Ordered	Sugar Inventory	Chocolate Inventory	Dough Inventory	Eggs Inventory	Butter Inventory	Milk Inventory	Coffee Inventory
4/30/2020								406902	240345	401830	4170	393643	295	40296							
5/31/2020	20090	11760	13195	354.0499999999995	30879	7.2	970								406902	240345	401830	4170	393643	295	40296
6/30/2020	20610	12010	13250	361	32540	7.1	1100								386812	228585	388635	3815.95	362764	287.8	39326
7/31/2020	21490	12255	13995	373.45	33386	7.5	1030								366202	216575	375385	3454.95	330234	280.7	38226
8/31/2020	22370	13220	14440	391.2	34576	7.8	990								344712	204320	361390	3081.5	296838	273.2	37196
9/30/2020	19695	11270	13000	345.25	30865	7.1	1010								322342	191100	346950	2690.3	262262	265.4	36206
10/31/2020	21850	12680	14165	383.45	34401	7.5	1160								302647	179830	333950	2345.05	231397	258.3	35196
11/30/2020	21275	12490	13855	373.25	32905	7.5	990								280797	167150	319785	1961.6	196996	250.8	34036
12/31/2020	20590	12030	13435	360.45	32421	7.3	990								259522	154660	305930	1588.35	164091	243.3	33046
1/31/2021	22365	13005	14255	387.7	34446	7.7	1100								238932	142630	292495	1227.9	131670	236	32056
2/28/2021	19310	11075	12775	340.8499999999997	30413	6.8	1000								216567	129625	278240	840.2	97234	228.3	30956
3/31/2021	21430	12560	13585	370.95	33121	7.4	1010								197257	118550	265465	499.35	66821	221.5	29956
4/30/2021	19915	11600	13305	350.5	31260	7.2	950				4170				175827	105990	251880	148.4	33700	214.1	28946
5/31/2021	22770	13290	14585	397.45	35461	7.8	1120					393643			155912	94390	238575	3947.9	3680	206.9	27996
6/30/2021	19915	11305	13185	350	31220	7.1	1040								133142	81100	223990	3550.45	360622	199.1	26876
7/31/2021	22030	13005	13945	381.7	34171	7.5	1000								113227	69795	210805	3200.45	329402	192	25836
8/31/2021	20740	11965	13935	366.7	32556	7.5	1050								91197	56790	196860	2818.75	295231	184.5	24836
9/30/2021	21295	12455	13425	371	33040	7.3	1100								70457	44825	182925	2452.05	262675	177	23786
10/31/2021	21720	12585	14395	380.95	34146	7.7	1030								49162	32370	168500	2081.05	229635	169.7	22686
11/30/2021	19840	11615	12955	348.5	30920	7.1	950								27442	19785	155105	1700.1	195489	162	21656
12/31/2021	21820	12665	14165	380.95	33951	7.5	1060	406902	240345						7692	8130	142150	1351.6	164569	154.9	20706
										401830							17580				
	Sugar Demand	Chocolate Demand	Dough Demand	Eggs Demand	Butter Demand	Milk Demand	Coffee Demand														
Order Quantity	406902	240345	401830	4170	393643	295	40296														
Reorder Time	20	20	30	10	10	40	40														
Reorder Quantity	21396.57542	12351.51334	13828.12814	374.1263524	33265.64361	nan	1012.08206														

Figure 12: Excel Worksheet Snapshot of Computations of When to Order and Contents of Orders

Subsequent orders will be placed once an inventory level lower than the R Reorder Point value is reached. To this end, we hold our computations on an Excel Worksheet for ease of use (Figure 12).

Red highlights indicated that the R Reorder Point is reached, accompanied by green highlights indicating that an order of size Q was placed. The in depth computations can be found in *"Analysis_Optimal Order Quantity.xlsx"*.

After the first order placed during April 2020, subsequent orders for every raw material during the forecasting period are as follows:

- Sugar: December 2021 (406.902 Kg)
- Chocolate: December 2021 (240.345 Kg)
- Dough: January 2022 (401.830 Kg)
- Eggs: April 2021 (4170 Units)
- Butter: May 2021 (393.643 Kg)

We can notice that the Lot-Size Reorder Point System falls at a limitation of combining orders that are close together(such as Eggs & Butter, or Sugar, Chocolate, & Dough) to minimize total cost.

Upon closer calculations, we find that ordering the Butter one month in advance (April 2021) and the Dough one month in advance (December 2021) reduces total cost. This is because the holding cost of Butter and Dough for 1 month (21,584 Wons & 8,813 Wons) is much smaller than the setup cost that would be reduced by combining orders (131,000 Wons)

Accordingly, the final order placement & content schedule for 2020 & 2021 is as follows:

Order 1 (April 2020):

- 406.902 Kg of Sugar
- 240.345 Kg of Chocolate
- 401.830 Kg of Dough
- 4170 Units of Eggs
- 393.643 Kg of Butter
- 295 Liters of Milk
- 40.296 Kg of Coffee

Order 2 (April 2021):

- 4170 Units of Eggs
- 393.643 Kg of Butter

Order 3 (December 2021):

- 406.902 Kg of Sugar
- 240.345 Kg of Chocolate
- 401.830 Kg of Dough

It is significant to finally reemphasize that this ordering schedule assumes that items do not perish.

V – Total Cost Estimates for 2020 & 2021

The cost estimates for the production plan proposed for 2020 and 2021 are constituted by worker salaries, order set-up costs, holding costs, and shortage penalty costs. Cost estimates would also include production cost, however, this is not provided by the client.

1. Worker Salaries

From the production plan described in Section III, we advise the bakery to hire 2 workers who will cover pastries, drinks, and delivery. Given that the worker Salaries are listed ad 750,000 Wons per month, the total salary cost over the 20 month planning period will be *30 Million Wons.*

2. Order Set-Up Costs

Over the 20-month planning period, we forecasted raw material demand, and calculated the optimal order size and set up time in Section IV. After further studying, we found that we can combine orders that are close in time to further reduce costs.

The final order schedule constituted of 3 order batches over the planning period. Given that the set-up cost per order is 131,000 Wons, the total set-up cost over the planning period is *393,000 Wons.*

3. Holding & Shortage Penalty Costs for Raw Materials

From the Lot-Size Reorder Point System equations, we can compute the holding cost using the below formula (Figure 14). Similarly, the penalty shortage costs can be computed (Figure 15).

$$h(Q/2 + R - \lambda\tau)$$

Figure 14: Holding Cost in Lot-Size Reorder Point Systems

$$p\lambda n(R)/Q.$$

Figure 13: Penalty Shortage Cost in Lot-Size Reorder Point Systems

Accordingly, we can calculate the holding cost and the penalty shortage cost for the 20-month forecast period by implementing the formulas as follows:

```
#computing total cast of holding and shortage penalty over forecast period
G_L = [0]*7
HoldingCosts = [0]*7
PenaltyCosts = [0]*7
for i in range (7):
  HoldingCosts[i] = H_L[i]*((Q_L[i]/2)+Rlast_L[i]-D_L[i]) * 20 #months
  PenaltyCosts[i] = Penalty_L[i]*D_L[i]*nRlast_L[i]/Q_L[i] * 20 #months

  G_L[i] = HoldingCosts[i]+PenaltyCosts[i]

PenaltyCosts[5] = 0 # milk penalty cost is 0
G_L[5] = HoldingCosts[5]

print("Holding Costs:", HoldingCosts)
print("Penalty Costs:", PenaltyCosts)
print("Total cost:", G_L)
```

Output:

Holding Costs: [134277.12263537763, 132085.83797851953, 88273.38821049221, 229765.52337610495, 216795.80420325435, 64703.33333333333, 66264.15999210814]

Penalty Costs: [77.16069560592135, 95.55408923738192, 33.41879535449784, 215.50278045277082, 191.58194147478434, 0, 37.0782766629947]

Total cost: [134354.28333098354, 132181.3920677569, 88306.80700584671, 229981.02615655772, 216987.38614472913, 64703.33333333333, 66301.23826877114]

Accordingly, the total shortage and penalty costs for all raw materials over the 20-month period is *932,816 Wons*.

4. Total Costs

To sum up our total costs, we have salary cost of 30 M Wons, set up cost of 393 K Wons, and holding & shortage cost of 933 K Wons. Accordingly, the total cost estimate is 31.326 Million Wons. It is important to note that the salaries of the workers is the largest cost element. This reinforces our decision to proceed with 2 workers with function overlap rather than focusing on 1 function per worker with 3 or 4 workers.

VI – Conclusion

Following our analysis of the bakery in Korea that seeks our advice, we studied the demand forecast of their sales over 2020 and 2021, the required number of workers at the bakery to meet the demand, the

schedule of raw material ordering time and content, as well as a final estimate of their costs over this planning period.

In our study of the forecast, we found that the different items sold in the bakery have different seasonality periods which the bakery can capitalize on to better plan their production. Following this demand forecast, we found that the bakery required 2 workers to be hired to take care of demand for pastries, drinks, and delivery. We also found the optimal order quantities of each raw material, and planned an ordering schedule with 3 order batches on April 2020, April 2021, and December 2021. This production plan constitutes various costs, including worker salaries (largest cost element), order set up, raw material holding cost, and penalty cost, which all account to 31.326 Million Wons.