THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

OPTIMIZATION I

WORKSHOP

GROUP 9

Gokul Kumar Kesavan
Anesh Thangaraj
Vansh Kumar
Deepankar Makwana
Rahul Saha

QUESTIONS

Q1: The Model without the option of additional A tomatoes

- **recommendation about which products to produce in what quantities [**disregarding the availability of the additional A tomatoes]

Product Type	Quantity
A Tomatoes (Whole) (Aw)	525
B Tomatoes (Whole) (Bw)	175
A Tomatoes (Juice) (Aj)	75
B Tomatoes (Juice) (Bj)	225
A Tomatoes (Paste) (Ap)	0
B Tomatoes (Paste) (Bp)	2000

Total Profit: $676,067 as in Appendix-01

Q2: Additional supply of A tomatoes

- **Should Gordon buy the additional A tomatoes?**
  Yes. Gordon should buy additional tomatoes since in the optimization model the total profit increases from $676,067 to $677,347 (profit increase of $1,280) **(appendix-2**)

- **If yes, how should he allocate the additional A tomatoes to the different products?**
  The entire 80,000 lbs. of additional A tomatoes should be allocated to whole tomatoes, as shown in the output:

Product	**Original A Allocation (lbs)**	**New A Allocation (lbs)**	**Additional A Tomatoes Used (lbs)**
Whole Tomatoes	525	535	80
Tomato Juice	75	5	0
Tomato Paste	0	0	0

Thus, no extra A tomatoes are allocated to juice or paste.

- **Is There Only One Optimal Allocation?**
  Yes, since all 80,000 lbs. of additional A tomatoes were allocated to whole tomatoes in the optimal solution. The marginal contribution of using A tomatoes for juice or paste is lower, making this allocation the best option.

- **Maximum Price RBC Should Pay for Additional A Tomatoes**
  The shadow price analysis suggests a maximum price of $0.271 per pound. If the cost exceeds $0.271 per pound, buying additional A tomatoes would not be profitable.

Q3: Advertising

Suppose that the marketing department of Red Brand Canners feels that it could increase the demand for any of the three tomato products by 5,000 cases, by means of advertising.

- **How much should RBC be willing to pay for such a campaign?**

RBC should be willing to pay no more than the increase in profit generated by the advertising campaign. The optimal objective values before and after advertising are:

Profit before advertising (Appendix-2) : $677,347
Profit after advertising (Appendix-4) : $685,733
Increase in profit due to advertising : 685,733−677,347=8,386
Thus, RBC should not spend more than $8,386 on advertising, as this is the maximum additional profit generated.
The decision to add the 5,000 cases only to Tomato Paste (con3) is based on the sensitivity analysis report, which is mentioned down.

- **At which product(s) should the advertising be directed?**

Advertising should be directed only toward Tomato Paste (con3), as it is the only product where increased demand leads to higher profitability. The sensitivity analysis report shows that Tomato Paste has a shadow price of $48.33, meaning that each additional case sold increases profit by this amount. However, this shadow price is valid only up to an upper limit of 2,173.33 cases. Beyond this limit, the shadow price may change, and additional demand might not be profitable as in (Appendix-05).

In contrast, Whole Tomatoes (con1) and Tomato Juice (con2) have shadow prices of $0.00, meaning that increasing their demand does not improve profitability. Additionally, Tomato Paste has zero slack, meaning all available production is fully utilized, while Whole Tomatoes and Tomato Juice have excess capacity but do not yield higher profits. Given these factors, advertising should focus exclusively on Tomato Paste, ensuring the highest return on investment for RBC.

- **Show how you can use the Sensitivity Analysis report for making this decision**:

Understanding Shadow Price and Upper Limits Before Advertising
Before increasing demand for Tomato Paste, the sensitivity analysis report showed the following: Whole Tomatoes and Tomato Juice had a shadow price of $0.00 and upper limit is infinity meaning increasing their demand would not increase profit. Tomato Paste had a shadow price of $48.33 and upper limit of 2,173.33 cases, meaning each additional case increases profit by $48.33 till our upper limit. Tomato Paste was fully utilized (0 slack), while the other products had unused capacity. Tomato Paste had an upper limit of 2,173.33 cases, meaning additional demand up to this limit would maintain the shadow price. (Appendix-02)

Impact of Adding 5,000 Cases to Tomato Paste (con3)
After increasing Tomato Paste demand by 5,000 cases, the new objective value increased from $677,347 to $685,733, showing a profit increase of $8,386**.** The shadow price of Tomato Paste became $0.00, meaning further increasing demand will no longer increase profit. Slack in Tomato Paste increased to 4,800, meaning there is now some unused production capacity. The upper limit for all products changed to infinity, meaning demand constraints are no longer a limiting factor—instead, additional B &/Or A-grade tomatoes are needed to increase production further. (Appendix-05)

Q4: Additional supply of B tomatoes

- What do you think about his reasoning?

Mitchell Gordon's reasoning for rejecting the offer of additional B tomatoes at 18 cents per pound is incorrect. The Sensitivity Analysis Report (Appendix-06) shows that the shadow price of B tomatoes is $222 per 1,000 lbs, which means each additional pound of B tomatoes contributes $0.222 (or $222 per 1,000 lbs) to the objective value.

Since the offered price of B tomatoes is 18 cents per pound ($180 per 1,000 lbs), the profit per 1,000 lbs would be $42 ($222 - $180). Because this results in a positive contribution to the overall profit, rejecting the offer based on the assumption that 18 cents is a "normal price" is not justified. Thus, Gordon's reasoning is flawed, as he is overlooking the potential profitability of acquiring more B tomatoes at the offered price.

- Would you buy additional B tomatoes at 18 cents per pound?

Yes, but only within the allowable limit. The Sensitivity Analysis Report (Appendix-06) shows that the shadow price for B supply is $222 per 1,000 lbs, meaning each additional pound contributes $0.222 to profit. Since the offered price is $180 per 1,000 lbs, this results in a net gain of $42 per 1,000 lbs, making the purchase profitable.
However, the upper limit for B supply is 7,200 lbs, and 2,400 lbs of demand has already been satisfied. This means that 7,200 – 2,400 = 4,800 lbs is the maximum additional B tomatoes RBC should purchase, as exceeding this limit could change the shadow price, making further purchases unprofitable.
Additionally, 4,800 is also the slack value of Paste demand, indicating that any additional B tomatoes will likely be allocated to Tomato Paste production.
Thus, RBC should buy additional B tomatoes but only up to 4,800 lbs, ensuring that demand for Tomato Paste remains profitable. Purchasing beyond this limit would not generate additional profit**.**

Q5: <u>Closing down production lines?</u>

- **If we decide not to start up the Paste line, will this increase or decrease our total profit?**

Based on the outputs from Appendix-1 and Appendix-7, the total profit when all three production lines (Whole, Juice, and Paste) are running is $676,067. When the Paste production line is removed, as seen in Appendix-7, the profit drops to $313,111 before setup costs and $213,111 after deducting the $100,000 setup cost for the two remaining lines ($50,000 per line). Since both the before and after setup cost values for running only two lines are significantly lower than keeping all three lines active, not starting the Paste line leads to a substantial loss in overall profit. This clearly indicates that the Paste line is a critical contributor to the company's profitability.
Thus, RBC should not shut down the Paste production line, as it results in a major decline in profit.

- **Which production lines would you advise RBC to start up?**

Appendix	Objective Value	Final Profit After Setup Costs
Appendix-1 (All Lines Active)	676,067	526,067
Appendix-07 (Whole & Juice)	313,111	213,111
Appendix-08 (Whole & Paste)	641,333	541,333
Appendix-09 (Juice & Paste)	642,000	542,000
Appendix-10 (Only Whole)	197,333	147,333
Appendix-11 (Only Juice)	198,000	148,000
Appendix-12 (Only Paste)	444,000	394,000

Best Production Line to Set Up:
After adjusting for setup costs, Appendix-09 (Juice & Paste) with $542,000 remains the most profitable option. While Appendix-1 (All Lines Active) had the highest objective value, its final profit after setup costs drops to $526,067, making Juice & Paste (Appendix-09) the best production choice for RBC.

Q6: <u>One year later</u>

- **Assume you know that next year is going to be a sunny year, how many pounds of tomatoes would you advise RBC to buy? Call this S. How about for a normal (N) and poor (P) year? Remember that RBC can order at most 13 million pounds.**

Based on the forecasted weather conditions, the optimal tomato purchase and corresponding profit values for each scenario are derived from Appendix-13 (Sunny Year), Appendix-14 (Normal Year), and Appendix-15 (Poor Year).

Year Type	Appendix Used	Optimal Tomato Pounds	A Tomato %	B Tomato %	Optimal Objective Value (Profit)
Sunny Year	Appendix-13	13,000 lbs	60%	40%	$513,533
Normal Year	Appendix-14	8,000 lbs	50%	50%	$275,333
Poor Year	Appendix-15	2,727.27 lbs	20%	80%	$77,939.40

- **Suppose you order S pounds of tomatoes. What would be the possible outcomes, given that the year could be sunny, normal or poor? Perform a scenario analysis. Do the same for ordering N and P tomatoes.**

Tomato Purchase Amount	Year Type	Optimal Tomato Pounds	Optimal Objective Value (Profit $)	Appendix Reference
S = 13,000	Sunny	13,000	513,533	Appendix-16
	Normal	8,000	275,333	Appendix-17
	Poor	2,727.27	77,939.4	Appendix-18
N = 8,000	Sunny	8,000	333,867	Appendix-19
	Normal	8,000	275,333	Appendix-20
	Poor	2,727.27	77,939.4	Appendix-21
P = 2,727.27	Sunny	2,727.27	113,818	Appendix-22
	Normal	2,727.27	104,848	Appendix-23
	Poor	2,727.27	77,939.3	Appendix-24

- **Given the probabilities of sunny, poor and normal years, what is the average total profit of ordering S, P and N tomatoes?**

Given the probabilities of sunny (25%), normal (50%), and poor (25%) years, we can determine the average total profit for ordering S, N, and P tomatoes by considering the profits obtained in the scenario analysis. From Appendix-25, we see that the expected total profit results in $285,534.58.

- **How many pounds of tomatoes would you advise RBC to buy?**

RBC can order up to 13 million pounds of tomatoes, but with only a 25% probability of a sunny year, ordering the maximum amount poses a risk. To maintain a balanced approach, RBC should avoid ordering more than 8 million pounds, as this is the optimal quantity for a normal year, which has the highest probability (50%). If RBC wants to take a risk-averse approach, they should order approximately 2.7 million pounds, ensuring that no tomatoes go to waste regardless of the year's outcome. However, the best strategy is to strike a balance by ordering 8 million pounds, as it aligns with the most likely scenario while minimizing waste.

Appendix-1

```
# Define the probabilities for each scenario
prob_sunny = 0.25  # 25% probability of a Sunny year
prob_normal = 0.50  # 50% probability of a Normal year
prob_poor = 0.25    # 25% probability of a Poor year

# Define the profit values for each scenario
profit_sunny = 513533  # Profit for Sunny year
profit_normal = 275333  # Profit for Normal year
profit_poor = 77939.3    # Profit for Poor year

# Calculate the expected total profit
expected_profit = (prob_sunny * profit_sunny) + (prob_normal * profit_normal) + (prob_poor * profit_poor)

# Print the result
print(f"Expected Total Profit: ${expected_profit:.2f}")
```

```
Expected Total Profit: $285534.58
```

First RBC Model

$$
\begin{aligned}
Z^* = \max \quad & 246.67(Aw + Bw) + 198(Aj + Bj) + 222(Ap + Bp) \\
\text{s.t.} \quad & Aw + Bw \leq 14,400 \\
& Aj + Bj \leq 1000 \\
& Ap + Bp \leq 2000 \\
& Aw + Aj + Ap \leq 600 \\
& Bw + Bj + Bp \leq 2400 \\
& 9Aw + 5Bw \geq 8(Aw + Bw) \\
& 9Aj + 5Bj \geq 6(Aj + Bj) \\
& Aw, Bw, Aj, Bj, Ap, Bp \geq 0
\end{aligned}
$$

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 2400, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "A_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "B_Quality")

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 7 rows, 6 columns and 16 nonzeros
Model fingerprint: 0x03ebcb09
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 1e+04]
Presolve removed 5 rows and 3 columns
Presolve time: 0.01s
Presolved: 2 rows, 3 columns, 5 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    7.4000000e+05   9.624497e+02   0.000000e+00      0s
       2    6.7606667e+05   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.02 seconds (0.00 work units)
Optimal objective  6.760666667e+05
Optimal Solution: Aw = 525
Optimal Solution: Bw = 175
Optimal Solution: Aj = 75
Optimal Solution: Bj = 225
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 676067
```

Appendix-02

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aaw = m.addVar(name="AAw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
aaj = m.addVar(name="AAj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")
aap = m.addVar(name="AAp")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw + aaw) + 198 * (aj + bj + aaj) + 222 * (ap + bp + aap) + (-255) * (aaw + aaj + aap)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + aaw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + aaj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + aap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 2400, "B_supply")
con6 = m.addConstr(aaw + aaj + aap <= 80, "AA_supply")

# Add Quality Constraints:
con7 = m.addConstr(9 * aw + 9 * aaw + 5 * bw >= 8 * (aw + aaw + bw), "A_Quality")
con8 = m.addConstr(9 * aj + 9 * aaj + 5 * bj >= 6 * (aj + aaj + bj), "B_Quality")

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
```

Second RBC Model

$$
\begin{aligned}
Z^* = \max \quad & 246.67(Aw + AAw + Bw) + 198(Aj + AAj + Bj) + 222(Ap + AAp + Bp) - 255(AAw + AAj + AAp) \\
\text{s.t.} \quad & Aw + AAw + Bw \leq 14,400 \\
& Aj + AAj + Bj \leq 1000 \\
& Ap + AAp + Bp \leq 2000 \\
& Aw + Aj + Ap \leq 600 \\
& Bw + Bj + Bp \leq 2400 \\
& AAw + AAj + AAp \leq 80 \\
& 9Aw + 9AAw + 5Bw \geq 8(Aw + AAw + Bw) \\
& 9Aj + 9AAj + 5Bj \geq 6(Aj + AAj + Bj) \\
& Aw, Bw, AAw, Aj, Bj, AAj, Ap, Bp, AAp \geq 0
\end{aligned}
$$

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aaw = m.addVar(name="AAw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
aaj = m.addVar(name="AAj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")
aap = m.addVar(name="AAp")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw + aaw) + 198 * (aj + bj + aaj) + 222 * (ap + bp + aap) + (-255) * (aaw + aaj + aap)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + aaw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + aaj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + aap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 2400, "B_supply")
con6 = m.addConstr(aaw + aaj + aap <= 80, "AA_supply")

# Add Quality Constraints:
con7 = m.addConstr(9 * aw + 9 * aaw + 5 * bw >= 8 * (aw + aaw + bw), "A_Quality")
con8 = m.addConstr(9 * aj + 9 * aaj + 5 * bj >= 6 * (aj + aaj + bj), "B_Quality")

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 9 columns and 24 nonzeros
Model fingerprint: 0xa4f7625f
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [8e+00, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [8e+01, 1e+04]
Presolve removed 3 rows and 3 columns
Presolve time: 0.00s
Presolved: 5 rows, 6 columns, 14 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    8.4529491e+05   1.660224e+02   0.000000e+00      0s
       3    6.7734667e+05   0.000000e+00   0.000000e+00      0s

Solved in 3 iterations and 0.00 seconds (0.00 work units)
Optimal objective  6.773466667e+05
Optimal Solution: Aw = 535
Optimal Solution: Bw = 205
Optimal Solution: AAw = 80
Optimal Solution: Aj = 65
Optimal Solution: Bj = 195
Optimal Solution: AAj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Solution: AAp = 0
Optimal Objective Value: 677347
```

Appendix-03

```python
# Create table for constraints' sensitivity analysis
constraint = OrderedDict([
    ('Name', ['con1', 'con2', 'con3']),
    ('Shadow Price', [con1.Pi, con2.Pi, con3.Pi]),
    ('RHS Coeff', [14400,1000,2000]),
    ('Slack', [con1.Slack, con2.Slack, con3.Slack]),
    ('Upper Range', [con1.SARHSUp, con2.SARHSUp, con3.SARHSUp]),
    ('Lower Range',[con1.SARHSLow, con2.SARHSLow, con3.SARHSLow])
])


# Print sensitivity analysis tables for decision variables and constraints
print('\n Sensivitity Report:')
print('\n')
print(pd.DataFrame.from_dict(constraint))
```

Sensivitity Report:

	Name	Shadow Price	RHS Coeff	Slack	Upper Range	Lower Range
0	con1	0.000000	14400	13580.0	inf	820.0
1	con2	0.000000	1000	740.0	inf	260.0
2	con3	48.333333	2000	0.0	2173.333333	1960.0

Appendix-04

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aaw = m.addVar(name="AAw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
aaj = m.addVar(name="AAj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")
aap = m.addVar(name="AAp")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw + aaw) + 198 * (aj + bj + aaj) + 222 * (ap + bp + aap) + -255 * (aaw + aaj + aap)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + aaw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + aaj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + aap + bp <= 2000 + 5000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 2400, "B_supply")
con6 = m.addConstr(aaw + aaj + aap <= 80, "AA_supply")

# Add Quality Constraints:
con7 = m.addConstr(9 * aw + 9 * aaw + 5 * bw >= 8 * (aw + aaw + bw), "A_Quality")
con8 = m.addConstr(9 * aj + 9 * aaj + 5 * bj >= 6 * (aj + aaj + bj), "B_Quality")

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 9 columns and 24 nonzeros
Model fingerprint: 0x6a8d4c9b
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [8e+00, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [8e+01, 1e+04]
Presolve removed 8 rows and 9 columns
Presolve time: 0.01s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    6.8573333e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.01 seconds (0.00 work units)
Optimal objective  6.857333333e+05
Optimal Solution: Aw = 600
Optimal Solution: Bw = 200
Optimal Solution: AAw = 0
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: AAj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2200
Optimal Solution: AAp = 0
Optimal Objective Value: 685733
```

Appendix-05

```python
# Create table for constraints' sensitivity analysis
constraint = OrderedDict([
    ('Name', ['con1', 'con2', 'con3']),
    ('Shadow Price', [con1.Pi, con2.Pi, con3.Pi]),
    ('RHS Coeff', [14400,1000,7000]),
    ('Slack', [con1.Slack, con2.Slack, con3.Slack]),
    ('Upper Range', [con1.SARHSUp, con2.SARHSUp, con3.SARHSUp]),
    ('Lower Range',[con1.SARHSLow, con2.SARHSLow, con3.SARHSLow])
])


# Print sensitivity analysis tables for decision variables and constraints
print('\n Sensivitity Report:')
print('\n')
print(pd.DataFrame.from_dict(constraint))
```

Sensivitity Report:

	Name	Shadow Price	RHS Coeff	Slack	Upper Range	Lower Range
0	con1	0.0	14400	13600.0	inf	800.0
1	con2	0.0	1000	1000.0	inf	0.0
2	con3	0.0	7000	4800.0	inf	2200.0

Appendix-06

```
# Sensitivity Analysis for Decision Variables
decision_var = OrderedDict([
    ('Name', [v.VarName for v in m.getVars()]),
    ('Final Value', [v.X for v in m.getVars()]),
    ('Reduced Cost', [v.RC for v in m.getVars()]),
    ('Obj Coeff', [v.Obj for v in m.getVars()]),
    ('Upper Range', [v.SAObjUp for v in m.getVars()]),
    ('Lower Range', [v.SAObjLow for v in m.getVars()])
])

# Sensitivity Analysis for Constraints
constraint = OrderedDict([
    ('Name', [c.ConstrName for c in m.getConstrs()]),
    ('Shadow Price', [c.Pi for c in m.getConstrs()]),
    ('RHS Coeff', [c.RHS for c in m.getConstrs()]),
    ('Slack', [c.Slack for c in m.getConstrs()]),
    ('Upper Range', [c.SARHSUp for c in m.getConstrs()]),
    ('Lower Range', [c.SARHSLow for c in m.getConstrs()])
])

# Print Sensitivity Analysis Reports
print('\nSensitivity Report:')
print(pd.DataFrame.from_dict(decision_var))
print('\n')
print(pd.DataFrame.from_dict(constraint))
```

Sensitivity Report:

	Name	Final Value	Reduced Cost	Obj Coeff	Upper Range	Lower Range
0	Aw	600.0	0.000000	246.666667	inf	213.777778
1	Bw	200.0	0.000000	246.666667	247.000000	222.000000
2	AAw	0.0	-0.111111	-8.333333	-8.222222	-inf
3	Aj	0.0	-56.888889	198.000000	254.888889	-inf
4	Bj	0.0	-24.000000	198.000000	222.000000	-inf
5	AAj	0.0	-57.000000	-57.000000	-0.000000	-inf
6	Ap	0.0	-32.888889	222.000000	254.888889	-inf
7	Bp	2200.0	0.000000	222.000000	246.666667	221.666667
8	AAp	0.0	-33.000000	-33.000000	-0.000000	-inf

	Name	Shadow Price	RHS Coeff	Slack	Upper Range	Lower Range
0	W_demand	0.000000	14400.0	13600.0	inf	800.0
1	J_demand	0.000000	1000.0	1000.0	inf	0.0
2	P_demand	0.000000	7000.0	4800.0	inf	2200.0
3	A_supply	254.888889	600.0	0.0	7200.0	0.0
4	B_supply	222.000000	2400.0	0.0	7200.0	200.0
5	AA_supply	0.000000	80.0	80.0	inf	0.0
6	A_Quality	-8.222222	0.0	0.0	600.0	-6600.0
7	B_Quality	0.000000	0.0	-0.0	0.0	-inf

Appendix-07

```python
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")

# Add Suppy Constraints:
con3 = m.addConstr(aw + aj <= 600, "A_supply")
con4 = m.addConstr(bw + bj <= 2400, "B_supply")

# Add Quality Constraints:
con5 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "A_Quality")
con6 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "B_Quality")

m.optimize()

after_cost_of_line = m.objVal - 100000

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
print('Optimal Objective Value after Setting up Cost of Production 2 Lines (Whole and Juice): %g' % after_cost_of_line)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 6 rows, 4 columns and 12 nonzeros
Model fingerprint: 0xae78dc67
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 1e+04]
Presolve removed 6 rows and 4 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.1311111e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.01 seconds (0.00 work units)
Optimal objective  3.131111111e+05
Optimal Solution: Aw = 350
Optimal Solution: Bw = 116.667
Optimal Solution: Aj = 250
Optimal Solution: Bj = 750
Optimal Objective Value: 313111
Optimal Objective Value after Setting up Cost of Production 2 Lines (Whole and Juice): 213111
```

Appendix-08

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 222 * (ap + bp)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con3 = m.addConstr(aw + ap <= 600, "A_supply")
con4 = m.addConstr(bw + bp <= 2400, "B_supply")

# Add Quality Constraints:
con5 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "A_Quality")

m.optimize()

after_cost_of_line = m.objVal - 100000

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
print('Optimal Objective Value after Setting up Cost of Production 2 Lines (Whole and Paste): %g' % after_cost_of_line)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 5 rows, 4 columns and 10 nonzeros
Model fingerprint: 0xeb13bafd
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 1e+04]
Presolve removed 5 rows and 4 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    6.4133333e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Optimal objective  6.413333333e+05
Optimal Solution: Aw = 600
Optimal Solution: Bw = 200
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 641333
Optimal Objective Value after Setting up Cost of Production 2 Lines (Whole and Paste): 541333
```

Appendix-09

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

# Set Max objective
Obj = 198 * (aj + bj) + 222 * (ap + bp)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aj + bj <= 1000, "J_demand")
con2 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con3 = m.addConstr(aj + ap <= 600, "A_supply")
con4 = m.addConstr(bj + bp <= 2400, "B_supply")

# Add Quality Constraints:
con5 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

m.optimize()

after_cost_of_line = m.objVal - 100000

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
print('Optimal Objective Value after Setting up Cost of Production 2 Lines (Juice and Paste): %g' % after_cost_of_line)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 5 rows, 4 columns and 10 nonzeros
Model fingerprint: 0x9f394f44
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 2e+03]
Presolve removed 1 rows and 0 columns
Presolve time: 0.00s
Presolved: 4 rows, 4 columns, 9 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    6.4200000e+05   5.248497e+02   0.000000e+00      0s
       4    6.4200000e+05   0.000000e+00   0.000000e+00      0s

Solved in 4 iterations and 0.00 seconds (0.00 work units)
Optimal objective  6.420000000e+05
Optimal Solution: Aj = 600
Optimal Solution: Bj = 400
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 642000
Optimal Objective Value after Setting up Cost of Production 2 Lines (Juice and Paste): 542000
```

Appendix-10

```python
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aw + bw <= 14400, "W_demand")

# Add Suppy Constraints:
con2 = m.addConstr(aw <= 600, "A_supply")
con3 = m.addConstr(bw <= 2400, "B_supply")

# Add Quality Constraints:
con4 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "A_Quality")

m.optimize()

after_cost_of_line = m.objVal - 50000

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
print('Optimal Objective Value after Setting up Cost of Production 1 Line (Whole): %g' % after_cost_of_line)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 4 rows, 2 columns and 6 nonzeros
Model fingerprint: 0x62db94e5
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 1e+04]
Presolve removed 4 rows and 2 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.9733333e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Optimal objective  1.973333333e+05
Optimal Solution: Aw = 600
Optimal Solution: Bw = 200
Optimal Objective Value: 197333
Optimal Objective Value after Setting up Cost of Production 1 Line (Whole): 147333
```

Appendix-11

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")

# Set Max objective
Obj = 198 * (aj + bj)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(aj + bj <= 1000, "J_demand")

# Add Suppy Constraints:
con2 = m.addConstr(aj <= 600, "A_supply")
con3 = m.addConstr(bj <= 2400, "B_supply")

# Add Quality Constraints:
con5 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

m.optimize()

after_cost_of_line = m.objVal - 50000

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
print('Optimal Objective Value after Setting up Cost of Production 1 Line (Juice): %g' % after_cost_of_line)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 4 rows, 2 columns and 6 nonzeros
Model fingerprint: 0x4436403e
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 2e+03]
Presolve removed 2 rows and 0 columns
Presolve time: 0.00s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.9800000e+05   1.000000e+02   0.000000e+00      0s
       1    1.9800000e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.00 seconds (0.00 work units)
Optimal objective  1.980000000e+05
Optimal Solution: Aj = 600
Optimal Solution: Bj = 400
Optimal Objective Value: 198000
Optimal Objective Value after Setting up Cost of Production 1 Line (Juice): 148000
```

Appendix-12

```python
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

# Set Max objective
Obj = 222 * (ap + bp)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
con1 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con2 = m.addConstr(ap <= 600, "A_supply")
con3 = m.addConstr(bp <= 2400, "B_supply")

m.optimize()

after_cost_of_line = m.objVal - 50000

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)
print('Optimal Objective Value after Setting up Cost of Production 1 Line (Paste): %g' % after_cost_of_line)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 3 rows, 2 columns and 4 nonzeros
Model fingerprint: 0x9ee7dd31
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [2e+02, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+02, 2e+03]
Presolve removed 3 rows and 2 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    4.4400000e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.01 seconds (0.00 work units)
Optimal objective  4.440000000e+05
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 444000
Optimal Objective Value after Setting up Cost of Production 1 Lines (Paste): 394000
```

Appendix-13

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 7800, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 5200, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.6 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.4 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Sunny Year: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xf965ae6e
Coefficient statistics:
  Matrix range     [4e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 8e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    6.0666667e+05   1.628521e+03   0.000000e+00      0s
       1    5.1353333e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.00 seconds (0.00 work units)
Optimal objective  5.135333333e+05
Optimal Solution: Aw = 7575
Optimal Solution: Bw = 2525
Optimal Solution: Aj = 225
Optimal Solution: Bj = 675
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 513533
Optimal Tomato Pounds Sunny Year: 13000
```

Appendix-14

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 6500, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 6500, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.5 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.5 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Normal Year: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xf215fe21
Coefficient statistics:
  Matrix range     [5e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 7e+03]
Presolve removed 8 rows and 6 columns
Presolve time: 0.01s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    2.7533333e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.01 seconds (0.00 work units)
Optimal objective  2.753333333e+05
Optimal Solution: Aw = 3750
Optimal Solution: Bw = 1250
Optimal Solution: Aj = 250
Optimal Solution: Bj = 750
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 275333
Optimal Tomato Pounds Normal Year: 8000
```

Appendix-15

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 2600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 10400, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.2 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.8 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Poor Year: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xd9365f0e
Coefficient statistics:
  Matrix range     [2e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 1e+04]
Presolve removed 8 rows and 6 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    7.7939394e+04   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.01 seconds (0.00 work units)
Optimal objective  7.793939394e+04
Optimal Solution: Aw = 545.455
Optimal Solution: Bw = 181.818
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 77939.4
Optimal Tomato Pounds Poor Year: 2727.27
```

Appendix-16

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 7800, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 5200, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.6 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.4 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Sunny Year S=13000: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xf965ae6e
Coefficient statistics:
  Matrix range     [4e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 8e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    6.0666667e+05   1.628521e+03   0.000000e+00      0s
       1    5.1353333e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.01 seconds (0.00 work units)
Optimal objective  5.135333333e+05
Optimal Solution: Aw = 7575
Optimal Solution: Bw = 2525
Optimal Solution: Aj = 225
Optimal Solution: Bj = 675
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 513533
Optimal Tomato Pounds Sunny Year S=13000: 13000
```

Appendix-17

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 6500, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 6500, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.5 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.5 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Normal Year S=13000: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xf215fe21
Coefficient statistics:
  Matrix range     [5e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 7e+03]
Presolve removed 8 rows and 6 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    2.7533333e+05   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Optimal objective  2.753333333e+05
Optimal Solution: Aw = 3750
Optimal Solution: Bw = 1250
Optimal Solution: Aj = 250
Optimal Solution: Bj = 750
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 275333
Optimal Tomato Pounds Normal Year S=13000: 8000
```

Appendix-18

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 2600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 10400, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.2 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.8 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Poor Year S=13000: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xd9365f0e
Coefficient statistics:
  Matrix range     [2e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 1e+04]
Presolve removed 8 rows and 6 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    7.7939394e+04   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Optimal objective  7.793939394e+04
Optimal Solution: Aw = 545.455
Optimal Solution: Bw = 181.818
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 77939.4
Optimal Tomato Pounds Poor Year S=13000: 2727.27
```

Appendix-19

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 4800, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 3200, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.6 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.4 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Sunny Year N=8000: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0x2e284caa
Coefficient statistics:
  Matrix range     [4e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 5e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.7333333e+05   7.535206e+02   0.000000e+00      0s
       1    3.3386667e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.00 seconds (0.00 work units)
Optimal objective  3.338666667e+05
Optimal Solution: Aw = 4800
Optimal Solution: Bw = 1600
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 1600
Optimal Objective Value: 333867
Optimal Tomato Pounds Sunny Year N=8000: 8000
```

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 4000, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 4000, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.5 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.5 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Normal Year N=8000: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0x620baae6
Coefficient statistics:
  Matrix range     [5e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 4e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.01s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.7333333e+05   1.687187e+03   0.000000e+00      0s
       1    2.7533333e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.01 seconds (0.00 work units)
Optimal objective  2.753333333e+05
Optimal Solution: Aw = 3750
Optimal Solution: Bw = 1250
Optimal Solution: Aj = 250
Optimal Solution: Bj = 750
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 275333
Optimal Tomato Pounds Normal Year N=8000: 8000
```

Appendix-21

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 1600, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 6400, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.2 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.8 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Poor Year N=8000: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0x36022a90
Coefficient statistics:
  Matrix range     [2e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 6e+03]
Presolve removed 8 rows and 6 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    7.7939394e+04   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds (0.00 work units)
Optimal objective  7.793939394e+04
Optimal Solution: Aw = 545.455
Optimal Solution: Bw = 181.818
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 77939.4
Optimal Tomato Pounds Poor Year N=8000: 2727.27
```

Appendix-22

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 1636.362, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 1090.908, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.6 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.4 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Sunny Year P=2727.27: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0x458fc225
Coefficient statistics:
  Matrix range     [4e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 2e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.2727260e+05   2.045453e+02   0.000000e+00      0s
       1    1.1381807e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.01 seconds (0.00 work units)
Optimal objective  1.138180680e+05
Optimal Solution: Aw = 1636.36
Optimal Solution: Bw = 545.454
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 545.454
Optimal Objective Value: 113818
Optimal Tomato Pounds Sunny Year P=2727.27: 2727.27
```

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 1363.635, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 1363.635, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.5 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.5 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Normal Year P=2727.27: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xdfec64f4
Coefficient statistics:
  Matrix range     [5e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+03, 2e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.00s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.2727260e+05   4.486299e+02   0.000000e+00      0s
       1    1.0484838e+05   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.00 seconds (0.00 work units)
Optimal objective  1.048483800e+05
Optimal Solution: Aw = 1363.63
Optimal Solution: Bw = 454.545
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 909.09
Optimal Objective Value: 104848
Optimal Tomato Pounds Normal Year P=2727.27: 2727.27
```

```
# Create a new model
m = Model("RBC")

# Create variables
# default is nonnegative: lb=0
aw = m.addVar(name="Aw")
bw = m.addVar(name="Bw")
aj = m.addVar(name="Aj")
bj = m.addVar(name="Bj")
ap = m.addVar(name="Ap")
bp = m.addVar(name="Bp")

S = aw+bw+aj+bj+ap+bp

# Set Max objective
Obj = (4.44/18 * 1000) * (aw + bw) + 198 * (aj + bj) + 222 * (ap + bp) - 200 * (S)
m.setObjective(Obj, GRB.MAXIMIZE)

# Add Demand constraints:
#con1 = m.addConstr(aw + bw <= 14400, "W_demand")
con2 = m.addConstr(aj + bj <= 1000, "J_demand")
con3 = m.addConstr(ap + bp <= 2000, "P_demand")

# Add Suppy Constraints:
con4 = m.addConstr(aw + aj + ap <= 545.454, "A_supply")
con5 = m.addConstr(bw + bj + bp <= 2181.816, "B_supply")

# Add Quality Constraints:
con6 = m.addConstr(9 * aw + 5 * bw >= 8 * (aw + bw), "W_Quality")
con7 = m.addConstr(9 * aj + 5 * bj >= 6 * (aj + bj), "J_Quality")

con8 = m.addConstr(aw + aj + ap == 0.2 * (aw + aj + ap + bw + bj + bp), name='A_ratio')
con9 = m.addConstr(bw + bj + bp == 0.8 * (aw + aj + ap + bw + bj + bp), name='B_ratio')

m.optimize()

# print optimal solutions
for v in m.getVars():
    print('Optimal Solution: %s = %g' % (v.varName, v.x))

# print optimal value
print('Optimal Objective Value: %g' % m.objVal)

# print optimal value of S
S_value = aw.x + bw.x + aj.x + bj.x + ap.x + bp.x
print('Optimal Tomato Pounds Poor Year P=2727.27: %g' % S_value)
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (win64 - Windows 11.0 (26100.2))

CPU model: AMD Ryzen 9 8945HS w/ Radeon 780M Graphics, instruction set [SSE2|AVX|AVX2|AVX512]
Thread count: 8 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 8 rows, 6 columns and 26 nonzeros
Model fingerprint: 0xade0478f
Coefficient statistics:
  Matrix range     [2e-01, 3e+00]
  Objective range  [2e+00, 5e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [5e+02, 2e+03]
Presolve removed 6 rows and 3 columns
Presolve time: 0.01s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    7.7943830e+04   3.431250e-02   0.000000e+00      0s
       1    7.7939316e+04   0.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.01 seconds (0.00 work units)
Optimal objective  7.793931600e+04
Optimal Solution: Aw = 545.454
Optimal Solution: Bw = 181.818
Optimal Solution: Aj = 0
Optimal Solution: Bj = 0
Optimal Solution: Ap = 0
Optimal Solution: Bp = 2000
Optimal Objective Value: 77939.3
Optimal Tomato Pounds Poor Year P=2727.27: 2727.27
```

Appendix-25

```python
# Define the probabilities for each scenario
prob_sunny = 0.25  # 25% probability of a Sunny year
prob_normal = 0.50  # 50% probability of a Normal year
prob_poor = 0.25    # 25% probability of a Poor year

# Define the profit values for each scenario
profit_sunny = 513533  # Profit for Sunny year
profit_normal = 275333  # Profit for Normal year
profit_poor = 77939.3    # Profit for Poor year

# Calculate the expected total profit
expected_profit = (prob_sunny * profit_sunny) + (prob_normal * profit_normal) + (prob_poor * profit_poor)

# Print the result
print(f"Expected Total Profit: ${expected_profit:.2f}")
```

```
Expected Total Profit: $285534.58
```