Hogwarts Juice Factory Schedule using Linear Programming

Friction problem solved by using Gurobi Optimization

Jay Ashokbhai Jayani
Industrial Engineering
Arizona State University
Tempe AZ USA
jjayani@asu.edu

ABSTRACT

This project aims to create an optimized work schedule for house elves at Severus Co. using linear programming. The objective is to maximize productivity while considering the Society for Promoting Elf Welfare (SPEW) regulations and potion production demands. The model minimizes costs while meeting labor requirements and specific potion brewing timelines. The results aid in efficient workforce allocation and productivity maximization for Professor Snape.

Chapter 1 Problem*

You are given the responsibility of creating a new work schedule for the house elves at Severus Co., a well-known potion making company run by Professor Snape. The professor believes that the elves could be more productive with a different schedule. The company operates daily from 6 AM until 6 PM, beginning its week on Mondays.

Since Hermione Granger's appointment as the minister of magic, she introduced the Society for Promoting the Elf Welfare (SPEW). According to this law, elves earn a minimum wage based on their experience level. Young elves earn 1 galleon per hour, while senior elves earn 2 Galleons per hour. Young elves work 2-hour shifts at any given time, while senior elves work 8-hour shifts starting at any 2-hour time slot. Furthermore, senior elves can work up to 2 hours of overtime daily at 3 Galleons per hour, only after their standard work hours are completed.

Severus Co. has many orders, including monthly contracts for at least 10 pots of Polyjuice potion, which requires 7 hours of labor a day for 30 days; 5 pots of Felix Felicis, requiring a total of 400 labor hours; and 3 pots of Veritaserum, which requires 5 hours of labor a day for 30 days. Each pot needs at least one elf on duty. However, young elves must be supervised by other elves, meaning you'll need the same number of other elves for each young elf during any given time.

The Polyjuice potion and Veritaserum must be brewed following a specific timetable for optimal results:

Time Slot\ Potion	**Polyjuice (Per pot)**	**Veritaserum (Per pot)**
1 (6-8 AM)	2 hours	0 hours
2 (8-10 AM)	0 hours	2 hours
3 (10-12 AM)	2 hours	0 hours
4 (12-2 PM)	0 hours	2 hours
5 (2-4 PM)	2 hours	0 hours
6 (4-6 PM)	0 hours	2 hours
Total	7 hours	5 hours

The Felix Felicis can be brewed at any time during the month.

Your task is to write a linear programming model that provides the optimal number of elves needed per hour each day. Assume all variables are continuous. Give brief description of all the constraints so the professor Snape (who only took an IEE 376 class back in the day in Hogwarts school of witchcraft and wizardry) can follow your “train” of thought (pun intended!).

Chapter 2 Solution

2.1 Decision Variable

- $X_{y,i,j}$: The number of young elves that are scheduled to cover the 2-hour shift starting at time slot i^{th} for day j^{th}
- $X_{s,i,j}$: The number of senior aged elves that are scheduled to start their 8-hour shift at time slot i^{th} for day j^{th}
- $X_{o,i,j}$: The number of senior aged elves working overtime that are scheduled to start their normal shift at time slot i^{th} for *day* j^{th}

2.2 Objective Function

- Minimize this function.

$$\sum_{j=1}^{30} \sum_{i=30}^{6} (\mathbf{1} * \mathbf{2})X_{y,i,j} + (2 * 8)X_{s,i,j} + (3 * 2)\, X_{o,i,j}$$

Total Galleons spent.

2.3 Constraints list

- Scheduling constraints for minimum number of elves needed at each time slot based on the recipe for the Polyjuice and Veritaserum:

$2X_{s,1,j} + 2X_{y,1,j} \geq (2 * 10) +(0 * 3)$

$2X_{s,1,j} + 2X_{s,2,j} + 2X_{y,2,j} \geq (0 * 10) + (2 * 3)$

$2X_{s,1,j} + 2X_{s,2,j} + 2X_{s,3,j} + 2X_{y,3,j} \geq (2 * 10) + (1 * 3)$

$2X_{s,1,j} + 2X_{s,2,j} + 2X_{s,3,j} + 2X_{y,4,j} \geq (0 * 10) + (0 * 3)$

$2X_{0,1,j} + 2X_{s,2,j} + 2X_{s,3,j} + 2X_{y,5,j} \geq (1 * 10) + (1 * 3)$

$2X_{0,2,j}$ + $2X_{s,3,j}$ + $2X_{y,6,j}$ ≥ (2 * 10) + (1 * 3)

∀j = 1, 2, …, 30

- Scheduling constraints for minimum number of elves needed total (so we have enough time also make Felix Felicis):

$\sum_{j=1}^{30} \sum_{j=1}^{6} (2)X_{y,i,j}$ + $(8)X_{s,i,j}$ + $(2)X_{0,i,j} \geq$ (7 * 10) + (400 * 5) + (5 * 3)

(Total potion making time required)

- Constraints ensuring enough senior elves looking out for young elves:

$X_{s,1,j} \geq X_{y,1,j}$ ∀j = 1, 2, …, 30 (1st time period, minimum number of senior elves)

$X_{s,1,j}$ + $X_{s,2,j} \geq X_{y,2,j}$ ∀j = 1, 2, …, 30 (2nd time period, minimum number of senior elves)

$X_{s,1,j}$ + $X_{s,2,j}$ + $X_{s,3,j} \geq X_{y,3,j}$ ∀j = 1, 2, …, 30 (3rd time period, minimum number of senior elves)

$X_{s,1,j}$ + $X_{s,2,j}$ + $X_{s,3,j} \geq X_{y,4,j}$ ∀j = 1, 2, …, 30 (4th time period, minimum number of senior elves)

$X_{0,1,j}$ + $X_{s,2,j}$ + $X_{s,3,j} \geq X_{y,5,j}$ ∀j = 1, 2, …, 30 (5th time period, minimum number of senior elves)

$X_{0,2,j}$ + $X_{s,3,j} \geq X_{y,6,j}$ ∀j = 1, 2, …, 30 (6th time period, minimum number of senior elves)

- Non-Negativity Constraints:

$X_{y,i,j}$, $X_{s,i,j}$, $X_{0,i,j} \geq 0$ ∀i = 1, 2, …, 6 & ∀j = 1, 2, …, 30

Chapter 3 Python LP model using Gurobi Optimization (based on given solution)

This give problem of linear programming is the problem of integer programming, but we here use linear programming for the better understanding of concepts of linear applications and algorithms. Nowadays, there are too many optimizations' libraries, solver and IDE software are available for the linear programming for instances gurobipy, pulp, pyomo, glpk, clpex, AMPL and so on. But here we are going to use the Gurobi Optimization which is a powerful library as compared to other libraries used in python programming and capable to solve mixed-integer linear programming (MILP), mixed-integer quadratic programming (MIQP), quadratic programming (QP), quadratically constrained programming (QCP), and mixed-integer quadratically constrained programming (MIQCP). We are also going to use other libraries, solver, and IDE software for comparisons of sentence formation and time taken to solve this problem. So, now we are going to solve this problem using gurobipy as following steps.

3.1 Python Programming

1. First, we install the gurobipy library from the server.

```
!pip install gurobipy
```

2. Importing gurobipy library (in this model I used panda library)

```
import gurobipy as gp
```

3. Define model, here I use the Hogwarts Juice Factory.

- You use any name based on your problem, for instance we work on ASU class schedule then you may use ASU_class_schedule. But remember one thing, whatever you type must be continues no space.
- Because any library we you define something it must in one line so you may use underscored "_" for making long name.

```
Hogwarts_Juice_Factory =
gp.Model()
```

4. Now we must define the parameters, values, and variables for building the objective function and constraints.

- In this section we must validate all variables and their values using the empty set or by takin desired value.
- Otherwise, it won't verify that variable while we run the code.
- Here we don't know the value of Elves, so we keep it blank. It's fine.
- And in time slot we can use te number also, but we use the Slot_1 and so on for better understanding of model.
- Same as in days. Here we can also use the range(1,31) function if we don't want to type all numbers.

```
Young_Elves = {}
Senior_Elves = {}
Overtime_Elves = {}
Time_Slots = ["Slot_1", "Slot_2",
"Slot_3", "Slot_4", "Slot_5",
"Slot_6"]
Days = ["1", "2", "3", "4", "5",
"6", "7", "8", "9", "10", "11",
"12", "13", "14", "15", "16",
"17", "18", "19", "20", "21",
"22", "23", "24", "25", "26",
"27", "28", "29", "30"]
```

5. Next turn to define the non-negativity constraints of the variable.

- Few examples of types of variables

  Continuous: GRB.CONTINUOUS
  General integer: GRB.INTEGER
  Binary: GRB.BINARY
  Semi-continuous: GRB.SEMICONT
  Semi-integer: GRB.SEMIINT

```
Young_Elves =
Hogwarts_Juice_Factory.addVars(Tim
e_Slots, Days, name="Young_Elves")
Senior_Elves =
Hogwarts_Juice_Factory.addVars(Tim
e_Slots, Days,
name="Senior_Elves")
Overtime_Elves =
Hogwarts_Juice_Factory.addVars(Tim
e_Slots, Days,
name="Overtime_Elves")
```

6. After defining all parameters and variables, we define objective function.

- Here "gp" recalls as gurobipy short form name. We define "gp" as importing library. You can use whatever name you want to use like gbp, grurobi etc.
- But when you call the function from the library you must use that defined name, otherwise the model shows error.
- "Senior_Elves[Time_Slot, Day]", This function make one type of loop and generates the all 180 constraint at same time. So, this is time saving we use the lager number of constraints within sort time.
- Here I used full names Young Elves, Senior Elves, and Overtime Elves.
- Also, I pre-defined time slots and days.
- And of course, this is the minimization problem, so we use MINIMIZE.

```
Objective_Function = gp.quicksum(2
* Young_Elves[Time_Slot, Day] + 16
* Senior_Elves[Time_Slot, Day] + 6
* Overtime_Elves[Time_Slot, Day]
for Time_Slot in Time_Slots for
Day in Days)
Hogwarts_Juice_Factory.setObjectiv
e(Objective_Function,
gp.GRB.MINIMIZE)
```

7. Next constraints,

- The first six constraints are determined time slots as per the factory rules.
- Following one for all three juices with total time required for production.
- Then the last six denote about supervised the young elves buy senior elves.
- Don't confuse with the time slot start with "0", because in the list function starts with 0, 1, 2, 3, 4, 5.
  - 0 = Time slot 1
  - 1 = Time slot 2
  - 2 = Time slot 3
  - 3 = Time slot 4
  - 4 = Time slot 5
  - 5 = Time slot 6

Slot Constraints:

```
Slot_6to8 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[0],
Day] + 2 *
Senior_Elves[Time_Slots[0], Day]
>= 2 * 10 + 0 * 3
for Day in Days)
Slot_6to10 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[1],
Day] + 2 *
Senior_Elves[Time_Slots[0], Day] +
2 * Senior_Elves[Time_Slots[1],
Day] >= 0 * 10 + 2 * 3
for Day in Days)
Slot_10to12 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[2],
Day] + 2 *
Senior_Elves[Time_Slots[0], Day] +
2 * Senior_Elves[Time_Slots[1],
Day] + 2 *
Senior_Elves[Time_Slots[2], Day]
>= 2 * 10 + 1 * 3
for Day in Days)
Slot_12to2 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[3],
Day] + 2 *
Senior_Elves[Time_Slots[0], Day] +
2 * Senior_Elves[Time_Slots[1],
Day]+ 2 *
Senior_Elves[Time_Slots[2], Day]
>= 0 * 10 + 0 * 3
for Day in Days)
Slot_2to4 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[4],
Day] + 2 *
Senior_Elves[Time_Slots[1], Day] +
2 * Senior_Elves[Time_Slots[2],
Day] + 2 *
Overtime_Elves[Time_Slots[4], Day]
>= 1 * 10 + 1 * 3
for Day in Days)
```

```
Slot_4to6 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[5],
Day] + 2 *
Senior_Elves[Time_Slots[2], Day] +
2 * Overtime_Elves[Time_Slots[5],
Day] >= 2 * 10 + 1 * 3
for Day in Days)
```

Total Hour Constraints:

```
Felix_Felici =
Hogwarts_Juice_Factory.addConstr(g
p.quicksum(2 *
Young_Elves[Time_Slot, Day] + 8 *
Senior_Elves[Time_Slot, Day] + 2 *
Overtime_Elves[Time_Slot, Day]
for Time_Slot in Time_Slots for
Day in Days) >= (7 * 10 * 30 + 400
* 5 + 5 * 3 * 30))
```

Watching Younger Elves:

```
Watch_1 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[0], Day] <=
Senior_Elves[Time_Slots[0], Day]
for Day in Days)
Watch_2 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[1], Day] <=
Senior_Elves[Time_Slots[0], Day] +
Senior_Elves[Time_Slots[1], Day]
for Day in Days)
Watch_3 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[2], Day] <=
Senior_Elves[Time_Slots[0], Day] +
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day]
for Day in Days)
Watch_4 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[3], Day] <=
Senior_Elves[Time_Slots[0], Day] +
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day]
for Day in Days)
Watch_5 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[4], Day] <=
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day] +
Overtime_Elves[Time_Slots[4], Day]
for Day in Days)
Watch_6 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[5], Day] <=
Senior_Elves[Time_Slots[2], Day] +
Overtime_Elves[Time_Slots[5], Day]
for Day in Days)
```

8. Now implement optimization function and run the full model, let's see what happens?

```
Hogwarts_Juice_Factory.optimize()
objective_value =
Hogwarts_Juice_Factory.objVal
print("Objective Function Value:",
objective_value)
```

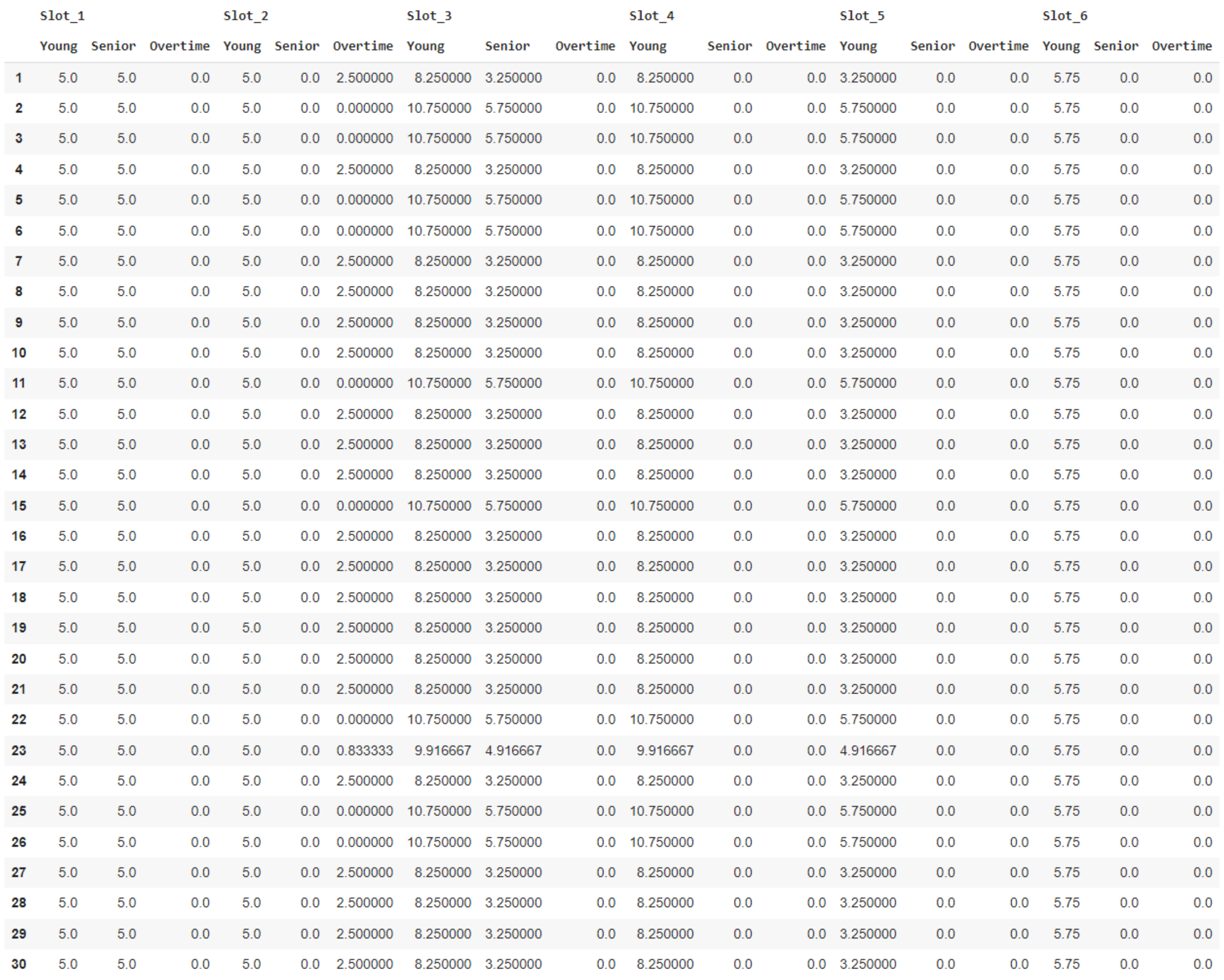
Output:

```
Gurobi Optimizer version 10.0.2
build v10.0.2rc0 (linux64)

CPU model: Intel(R) Xeon(R) CPU @
2.20GHz, instruction set
[SSE2|AVX|AVX2]
Thread count: 1 physical cores, 2
logical processors, using up to 2
threads

Optimize a model with 361 rows,
540 columns and 1740 nonzeros
Model fingerprint: 0xc7fa9ea0
Coefficient statistics:
  Matrix range     [1e+00, 8e+00]
  Objective range  [2e+00, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 5e+03]
Presolve removed 30 rows and 209
columns
Presolve time: 0.01s
Presolved: 331 rows, 331 columns,
1411 nonzeros

Iteration    Objective
Primal Inf.    Dual Inf.      Time
       0    0.0000000e+00
2.412500e+03   0.000000e+00
0s
     350    6.9266667e+03
0.000000e+00   0.000000e+00
0s

Solved in 350 iterations and 0.03
seconds (0.00 work units)
Optimal objective  6.926666667e+03
Objective Function Value:
6926.666666666667
```

```
 for v in
Hogwarts_Juice_Factory.getVars():
        print(v.varName, v.x)
```

Output:

```
Young_Elves[Slot_1,1] 5.0
Young_Elves[Slot_1,2] 5.0
Young_Elves[Slot_1,3] 5.0
Young_Elves[Slot_1,4] 5.0
Young_Elves[Slot_1,5] 5.0
Young_Elves[Slot_1,6] 5.0

.

.

.

.

Overtime_Elves[Slot_6,27] 0.0
Overtime_Elves[Slot_6,28] 0.0
Overtime_Elves[Slot_6,29] 0.0
Overtime_Elves[Slot_6,30] 0.0
```



	Slot_1			Slot_2			Slot_3			Slot_4			Slot_5			Slot_6		
	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime
1	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
2	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
3	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
4	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
5	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
6	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
7	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
8	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
9	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
10	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
11	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
12	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
13	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
14	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
15	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
16	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
17	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
18	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
19	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
20	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
21	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
22	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
23	5.0	5.0	0.0	5.0	0.0	0.833333	9.916667	4.916667	0.0	9.916667	0.0	0.0	4.916667	0.0	0.0	5.75	0.0	0.0
24	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
25	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
26	5.0	5.0	0.0	5.0	0.0	0.000000	10.750000	5.750000	0.0	10.750000	0.0	0.0	5.750000	0.0	0.0	5.75	0.0	0.0
27	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
28	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
29	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0
30	5.0	5.0	0.0	5.0	0.0	2.500000	8.250000	3.250000	0.0	8.250000	0.0	0.0	3.250000	0.0	0.0	5.75	0.0	0.0

Image 3.1 Hogwarts Factory Schedule based on given solution

9. Now we can see that objective function value base on the given solution gives the $Z = 6926.67$ Galleons, which basically also obtained by the AMPL programming (see next chapter 4).

- Here the output value given by the default function is quite messy and hard to read the value. So, we must rearrange all values for better understanding.
- That's why we must use the Panda library, which is best for table formation and arrangement.
- You can show that amazing output show with proper rows and columns in image 3.1.

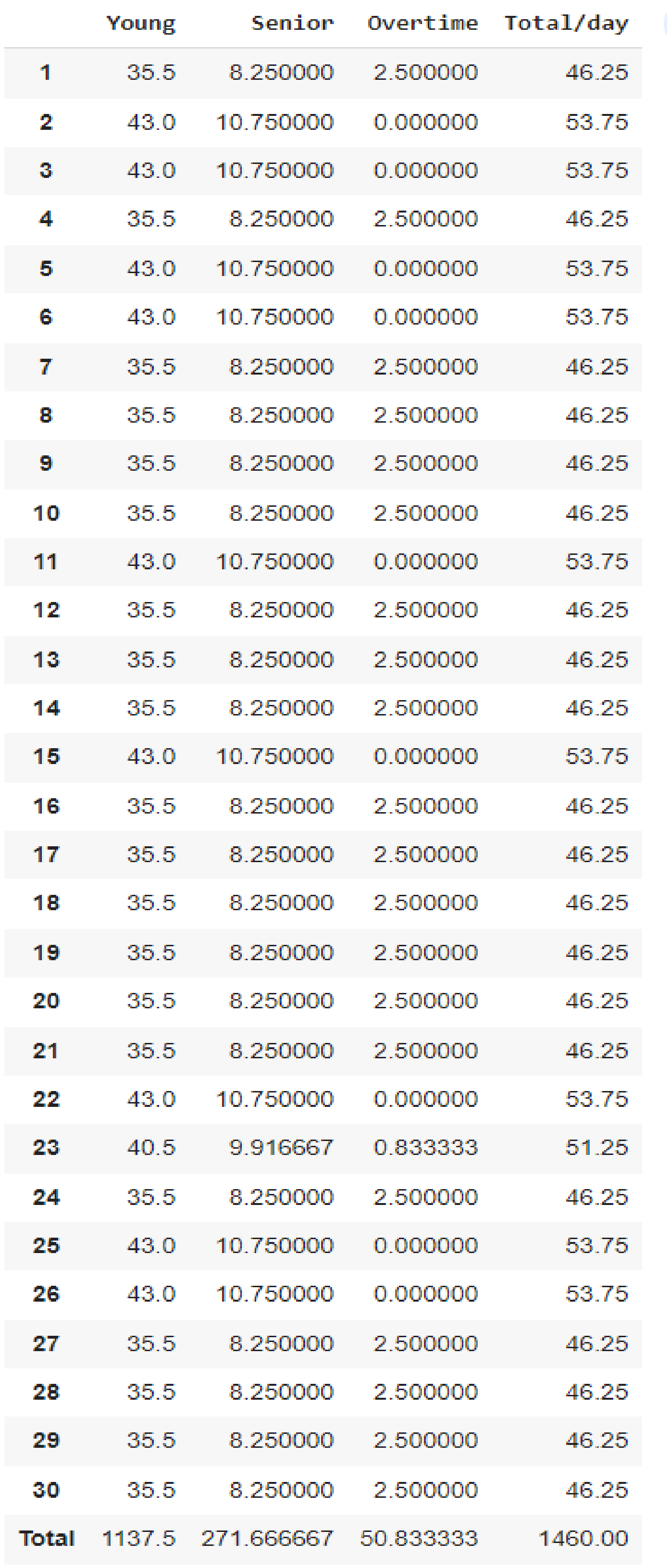
```
# Create data dictionaries for
Young Elves, Senior Elves, and
Overtime Elves
Wokers_Requirements = {}
# Populate data dictionaries with
the respective values
for Time_Slot in Time_Slots:
    young = {}
    senior = {}
    overtime = {}
    for Day in Days:
        young[Day] =
Young_Elves[Time_Slot, Day].x
        senior[Day] =
Senior_Elves[Time_Slot, Day].x
```

```
        overtime[Day]=
Overtime_Elves[Time_Slot, Day].x
    Wokers_Requirements[Time_Slot]
= {'Young': young, 'Senior':
senior, 'Overtime': overtime}
# Create dataframes from the data
dictionaries
Factory_Schedule_df =
pd.DataFrame(Wokers_Requirements,
index=['Young Elves'])
# Transpose the dataframes to have
time slots as rows and days as
columns
Factory_Schedule_df =
Factory_Schedule_df.T

# Create an empty dataframe for
Young Elves
Factory_Schedule_df =
pd.DataFrame.from_dict({(i,j):
Wokers_Requirements[i][j]
for i in
Wokers_Requirements.keys()
for j in
Wokers_Requirements[i].keys()},
orient='index')
Factory_Schedule_df.T
```

10. Now, we separate basic and non-basic variables. (This is not need but only educational purpose only)

- This is a more technical and theory term we don't have to care about this term (we don't know LP), otherwise is more useful in LP for tableau methods to find objective function values.
- See output on Python file.

11. Now, we want to check how many young, senior and overtime elves work during each day and total required in 30 days.



	Young	Senior	Overtime	Total/day
1	35.5	8.250000	2.500000	46.25
2	43.0	10.750000	0.000000	53.75
3	43.0	10.750000	0.000000	53.75
4	35.5	8.250000	2.500000	46.25
5	43.0	10.750000	0.000000	53.75
6	43.0	10.750000	0.000000	53.75
7	35.5	8.250000	2.500000	46.25
8	35.5	8.250000	2.500000	46.25
9	35.5	8.250000	2.500000	46.25
10	35.5	8.250000	2.500000	46.25
11	43.0	10.750000	0.000000	53.75
12	35.5	8.250000	2.500000	46.25
13	35.5	8.250000	2.500000	46.25
14	35.5	8.250000	2.500000	46.25
15	43.0	10.750000	0.000000	53.75
16	35.5	8.250000	2.500000	46.25
17	35.5	8.250000	2.500000	46.25
18	35.5	8.250000	2.500000	46.25
19	35.5	8.250000	2.500000	46.25
20	35.5	8.250000	2.500000	46.25
21	35.5	8.250000	2.500000	46.25
22	43.0	10.750000	0.000000	53.75
23	40.5	9.916667	0.833333	51.25
24	35.5	8.250000	2.500000	46.25
25	43.0	10.750000	0.000000	53.75
26	43.0	10.750000	0.000000	53.75
27	35.5	8.250000	2.500000	46.25
28	35.5	8.250000	2.500000	46.25
29	35.5	8.250000	2.500000	46.25
30	35.5	8.250000	2.500000	46.25
Total	1137.5	271.666667	50.833333	1460.00

Image 3.2 Total number of Elves work during month

- Output values shown in image 3.2.

```
Young_Elves_Total = {}
Senior_Elves_Total = {}
Overtime_Elves_Total = {}
```

```
for day in Days:
  young_sum = 0
  senior_sum = 0
  overtime_sum = 0
  for slot in Time_Slots:
    young_sum +=
Factory_Schedule_df[day][slot]['Yo
ung']
    senior_sum +=
Factory_Schedule_df[day][slot]['Se
nior']
    overtime_sum +=
Factory_Schedule_df[day][slot]['Ov
ertime']
  Young_Elves_Total[day] =
young_sum
  Senior_Elves_Total[day] =
senior_sum
  Overtime_Elves_Total[day] =
overtime_sum
```

```
totals = {}
totals['Young'] =
Young_Elves_Total
totals['Senior'] =
Senior_Elves_Total
totals['Overtime'] =
Overtime_Elves_Total
```

```
totals_df =
pd.DataFrame.from_dict(totals)
totals_df['Total/day'] =
totals_df['Young'] +
totals_df['Senior'] +
totals_df['Overtime']
```

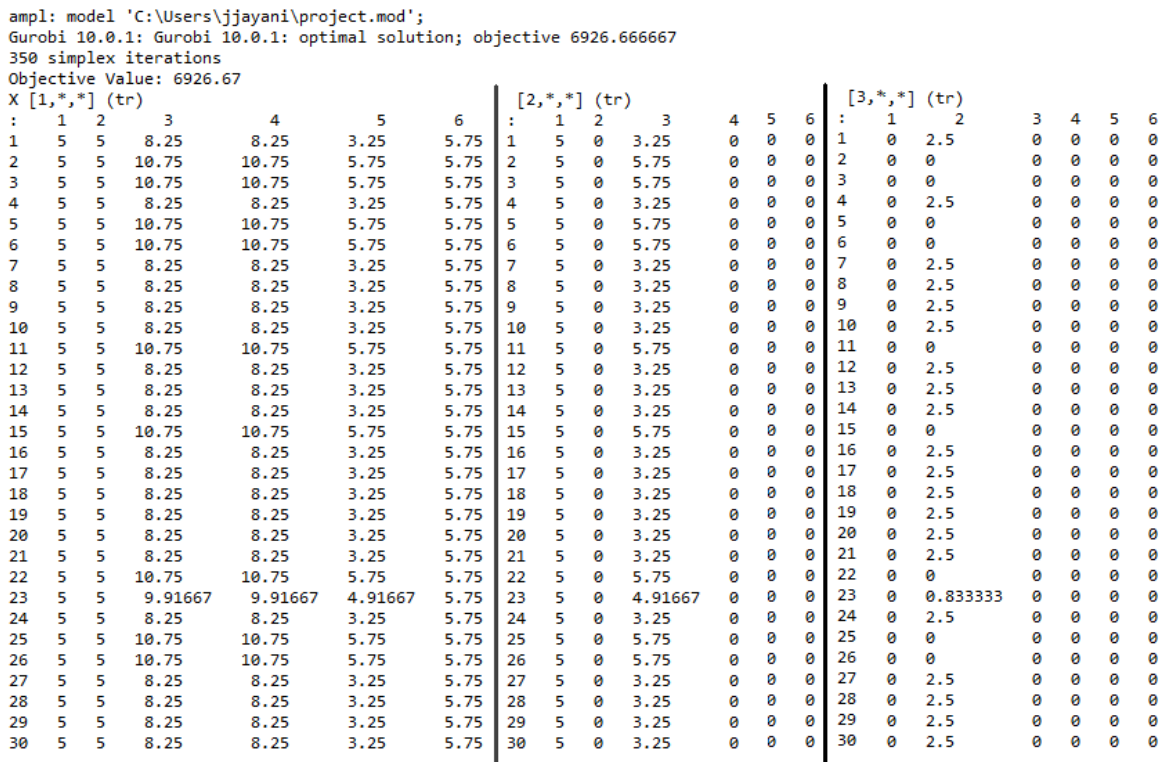
```
totals_df.loc['Total'] =
totals_df.sum(numeric_only=True)
totals_df
```

Chapter 4 AMPL Program (For reference only to compare objective function value)

AMPL is also one of the best optimizations solves and software which is mostly used in industries and academic purpose also the AMPL software comparable with another solver. So, we don't have to change code for another solver. Here I show you this code because of to compare the objective function and variable value. Also, this is the first time I use the gurobipy so I want to check if my objective function value is true or not.

- AMPL Code and its output shown on image 4.1.

```
set I; #I := 1 2 3; # 1= Young_Elves,
2= Senior_Elves, 3= Overtime_Elves

set J; #J := 1 2 3 4 5 6; # 1= Slot_1,
2= Slot_2, 3= Slot_3, 4= Slot_4, 5=
Slot_5, 6= Slot_6
set K; #K := 1..30; # 1= Day_1,
......., 30= Day_30

param param1 {I};
param param2 {J};
param param3 {K};

var X{I, J, K} >=0 ;

minimize objective_function:
sum {j in J, k in K} (2*X[1,j,k] +
16*X[2,j,k] + 6*X[3,j,k]);

subject to slot_1{k in K}:
2*X[2,1,k] + 2*X[1,1,k] >= 2*10 + 0*3;
subject to slot_2{k in K}:
2*X[2,1,k] + 2*X[2,2,k] + 2*X[1,2,k] >=
0*10 + 2*3;
subject to slot_3{k in K}:
2*X[2,1,k] + 2*X[2,2,k] + 2*X[2,3,k] +
2*X[1,3,k] >= 2*10 + 1*3;
subject to slot_4{k in K}:
2*X[2,1,k] + 2*X[2,2,k] + 2*X[2,3,k] +
2*X[1,4,k] >= 0*10 + 0*3;
subject to slot_5{k in K}:
2*X[3,1,k] + 2*X[2,2,k] + 2*X[2,3,k] +
2*X[1,5,k] >= 1*10 + 1*3;
subject to slot_6{k in K}:
2*X[3,2,k] + 2*X[2,3,k] + 2*X[1,6,k] >=
2*10 + 1*3;

subject to felix_felicis {k in K}:
sum {m in K} (2 * X[1,1,m] + 8 *
X[2,1,m] + 2 * X[3,1,m]
+ 2 * X[1,2,m] + 8 * X[2,2,m] + 2 *
X[3,2,m]
+ 2 * X[1,3,m] + 8 * X[2,3,m] + 2 *
X[3,3,m]
+ 2 * X[1,4,m] + 8 * X[2,4,m] + 2 *
X[3,4,m]
+ 2 * X[1,5,m] + 8 * X[2,5,m] + 2 *
X[3,5,m]
+ 2 * X[1,6,m] + 8 * X[2,6,m] + 2 *
X[3,6,m]) >= 7 * 10 * 30 + 400 * 5 + 5
* 3 * 30;

subject to watch_1{k in K}:
X[1,1,k] <= X[2,1,k];
subject to watch_2{k in K}:
X[1,2,k] <= X[2,1,k] + X[2,2,k];
subject to watch_3{k in K}:
X[1,3,k] <= X[2,1,k] + X[2,2,k] +
X[2,3,k];
subject to watch_4{k in K}:
X[1,4,k] <= X[2,1,k] + X[2,2,k] +
X[2,3,k];
subject to watch_5{k in K}:
X[1,5,k] <= X[3,1,k] + X[2,2,k] +
X[2,3,k];
subject to watch_6{k in K}:
X[1,6,k] <= X[3,2,k] + X[2,3,k];

data;
set I := 1 2 3;
set J := 1 2 3 4 5 6;
set K := 1 2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30;
solve;

printf "Objective Value: %g\n",
objective_function;

display X;
```



```
ampl: model 'C:\Users\jjayani\project.mod';
Gurobi 10.0.1: Gurobi 10.0.1: optimal solution; objective 6926.666667
350 simplex iterations
Objective Value: 6926.67
X [1,*,*] (tr)                                       | [2,*,*] (tr)                    | [3,*,*] (tr)
:    1   2     3          4          5         6     | :    1   2    3         4  5  6 | :    1    2         3  4  5  6
1    5   5    8.25       8.25       3.25      5.75   | 1    5   0   3.25       0  0  0 | 1    0   2.5        0  0  0  0
2    5   5   10.75      10.75       5.75      5.75   | 2    5   0   5.75       0  0  0 | 2    0   0          0  0  0  0
3    5   5   10.75      10.75       5.75      5.75   | 3    5   0   5.75       0  0  0 | 3    0   0          0  0  0  0
4    5   5    8.25       8.25       3.25      5.75   | 4    5   0   3.25       0  0  0 | 4    0   2.5        0  0  0  0
5    5   5   10.75      10.75       5.75      5.75   | 5    5   0   5.75       0  0  0 | 5    0   0          0  0  0  0
6    5   5   10.75      10.75       5.75      5.75   | 6    5   0   5.75       0  0  0 | 6    0   0          0  0  0  0
7    5   5    8.25       8.25       3.25      5.75   | 7    5   0   3.25       0  0  0 | 7    0   2.5        0  0  0  0
8    5   5    8.25       8.25       3.25      5.75   | 8    5   0   3.25       0  0  0 | 8    0   2.5        0  0  0  0
9    5   5    8.25       8.25       3.25      5.75   | 9    5   0   3.25       0  0  0 | 9    0   2.5        0  0  0  0
10   5   5    8.25       8.25       3.25      5.75   | 10   5   0   3.25       0  0  0 | 10   0   2.5        0  0  0  0
11   5   5   10.75      10.75       5.75      5.75   | 11   5   0   5.75       0  0  0 | 11   0   0          0  0  0  0
12   5   5    8.25       8.25       3.25      5.75   | 12   5   0   3.25       0  0  0 | 12   0   2.5        0  0  0  0
13   5   5    8.25       8.25       3.25      5.75   | 13   5   0   3.25       0  0  0 | 13   0   2.5        0  0  0  0
14   5   5    8.25       8.25       3.25      5.75   | 14   5   0   3.25       0  0  0 | 14   0   2.5        0  0  0  0
15   5   5   10.75      10.75       5.75      5.75   | 15   5   0   5.75       0  0  0 | 15   0   0          0  0  0  0
16   5   5    8.25       8.25       3.25      5.75   | 16   5   0   3.25       0  0  0 | 16   0   2.5        0  0  0  0
17   5   5    8.25       8.25       3.25      5.75   | 17   5   0   3.25       0  0  0 | 17   0   2.5        0  0  0  0
18   5   5    8.25       8.25       3.25      5.75   | 18   5   0   3.25       0  0  0 | 18   0   2.5        0  0  0  0
19   5   5    8.25       8.25       3.25      5.75   | 19   5   0   3.25       0  0  0 | 19   0   2.5        0  0  0  0
20   5   5    8.25       8.25       3.25      5.75   | 20   5   0   3.25       0  0  0 | 20   0   2.5        0  0  0  0
21   5   5    8.25       8.25       3.25      5.75   | 21   5   0   3.25       0  0  0 | 21   0   2.5        0  0  0  0
22   5   5   10.75      10.75       5.75      5.75   | 22   5   0   5.75       0  0  0 | 22   0   0          0  0  0  0
23   5   5    9.91667    9.91667    4.91667   5.75   | 23   5   0   4.91667    0  0  0 | 23   0   0.833333   0  0  0  0
24   5   5    8.25       8.25       3.25      5.75   | 24   5   0   3.25       0  0  0 | 24   0   2.5        0  0  0  0
25   5   5   10.75      10.75       5.75      5.75   | 25   5   0   5.75       0  0  0 | 25   0   0          0  0  0  0
26   5   5   10.75      10.75       5.75      5.75   | 26   5   0   5.75       0  0  0 | 26   0   0          0  0  0  0
27   5   5    8.25       8.25       3.25      5.75   | 27   5   0   3.25       0  0  0 | 27   0   2.5        0  0  0  0
28   5   5    8.25       8.25       3.25      5.75   | 28   5   0   3.25       0  0  0 | 28   0   2.5        0  0  0  0
29   5   5    8.25       8.25       3.25      5.75   | 29   5   0   3.25       0  0  0 | 29   0   2.5        0  0  0  0
30   5   5    8.25       8.25       3.25      5.75   | 30   5   0   3.25       0  0  0 | 30   0   2.5        0  0  0  0
```

Image 4.1 AMPL Programming Output

Chapter 5 Change in overtime constraints

After careful observation of the given model, it gives overtime shift during time slots 1 and 2 in between 6 to 10 AM. Which is impossible to do based on problem definition. It's said that senior elves can do overtime of 2 hours after completing the 8-hour shift. But here they are starting the overtime shift in the morning. You also can refer the image 3.1 (image 4.1) So, we must change constraint formations in Gurobi model, so we get true values of scheduling.

```
Slot_2to4 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[4],
Day] + 2 *
Senior_Elves[Time_Slots[1], Day] +
2 * Senior_Elves[Time_Slots[2],
Day] + 2 *
Overtime_Elves[Time_Slots[4], Day]
>= 1 * 10 + 1 * 3
for Day in Days)
Slot_4to6 =
Hogwarts_Juice_Factory.addConstrs(
2 * Young_Elves[Time_Slots[5],
Day] + 2 *
Senior_Elves[Time_Slots[2], Day] +
2 * Overtime_Elves[Time_Slots[5],
Day] >= 2 * 10 + 1 * 3
for Day in Days)
Watch_5 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[4], Day] <=
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day] +
Overtime_Elves[Time_Slots[1], Day]
for Day in Days)
Watch_6 =
Hogwarts_Juice_Factory.addConstrs(
Young_Elves[Time_Slots[5], Day] <=
Senior_Elves[Time_Slots[2], Day] +
Overtime_Elves[Time_Slots[5], Day]
for Day in Days)
```

- Here I am only going to show constraints and the output image 5.1 so you can see what the change as compared to previous model.
- Interestingly there is no change on the objective function value. So, can keep the new constraints.

Chapter 6 Pyomo Model with Gurobi solver

Let's do something interesting. So here I will be going to show the pyomo model with gurobi solver. I want to check how much time takes to solve the problem and compare other solver using same model. So, here I used gurobi, glpk and cplex. You may take other solvers to compare but I want to give you rough idea which is fast among these three solvers.

Pyomo Code:

```
# Define pyomo model as "model"
model = ConcreteModel()

# Define parameters and values
using range function
I = range(1, 4) # 1= Young_Elves,
2= Senior_Elves, 3= Overtime_Elves
J = range(1, 7) # 1= Slot_1, 2=
Slot_2, 3= Slot_3, 4= Slot_4, 5=
Slot_5, 6= Solt_6
K = range(1, 31) # 1= Day_1,
......., 30= Day_30

# Define the variables
model.X = Var(I, J, K,
within=NonNegativeReals)

# Define the objective function
def objective_function(model,
sense= minimize):
  return sum(2 * model.X[1, j, k]
+ 16 * model.X[2, j, k] + 6 *
model.X[3, j, k]  for j in J for k
in K)
model.objective_function =
Objective(rule=
objective_function,
sense=minimize)

# Define the constraint
def slot_1(model, k):
    return (2 * model.X[2, 1, k] +
2 * model.X[1, 1, k]) >= 2 * 10 +
0 * 3
model.slot_1 = Constraint(K,
rule=slot_1)

def slot_2(model, k):
    return (2* model.X[2, 1, k] +
2 * model.X[2, 2, k] + 2 *
model.X[1, 2, k]) >= 0 * 10 + 2 *
3
model.slot_2 = Constraint(K,
rule=slot_2)

def slot_3(model, k):
    return (2* model.X[2, 1, k] +
2 * model.X[2, 2, k] + 2 *
model.X[2, 3, k] + 2 * model.X[1,
3, k]) >= 2 * 10 + 1 * 3
model.slot_3 = Constraint(K,
rule=slot_3)

def slot_4(model, k):
    return (2* model.X[2, 1, k] +
2 * model.X[2, 2, k] + 2 *
model.X[2, 3, k] +  2 * model.X[1,
4, k]) >= 0 * 10 + 0 * 3
model.slot_4 = Constraint(K,
rule=slot_4)

def slot_5(model, k):
    return (2* model.X[3, 1, k] +
2 * model.X[2, 2, k] + 2 *
model.X[2, 3, k] +  2 * model.X[1,
5, k]) >= 1 * 10 + 1 * 3
model.slot_5 = Constraint(K,
rule=slot_5)

def slot_6(model, k):
    return (2* model.X[3, 2, k] +
2 * model.X[2, 3, k] + 2 *
model.X[1, 6, k]) >= 2 * 10 + 1 *
3
model.slot_6 = Constraint(K,
rule=slot_6)

def felix_felicis(model, j, k):
    return sum(2* model.X[1, j, k]
+ 8 * model.X[2, j, k] + 2 *
model.X[3, j, k] for j in J for k
in K) >= 7 * 10 * 30 + 400 * 5 + 5
* 3 * 30
model.felix_felicis =
Constraint(J, K, rule=
felix_felicis)

def watch_1(model, k):
    return (model.X[1, 1, k]) <=
model.X[2, 1, k]
model.watch_1 = Constraint(K,
rule= watch_1)

def watch_2(model, k):
    return (model.X[1, 2, k]) <=
model.X[2, 1, k] + model.X[2, 2,
k]
model.watch_2 = Constraint(K,
rule= watch_2)

def watch_3(model, k):
    return (model.X[1, 3, k]) <=
model.X[2, 1, k] + model.X[2, 2,
k] + model.X[2, 3, k]
model.watch_3 = Constraint(K,
rule= watch_3)

def watch_4(model, k):
    return (model.X[1, 4, k]) <=
model.X[2, 1, k] + model.X[2, 2,
k] + model.X[2, 3, k]
model.watch_4 = Constraint(K,
rule= watch_4)

def watch_5(model, k):
    return (model.X[1, 5, k]) <=
model.X[3, 1, k] + model.X[2, 2,
k] + model.X[2, 3, k]
model.watch_5 = Constraint(K,
rule= watch_5)

def watch_6(model, k):
    return (model.X[1, 6, k]) <=
model.X[3, 2, k] + model.X[2, 3,
k]
model.watch_6 = Constraint(K,
rule= watch_6)

# Solve the model
Gurobi_solver_start_time =
time.time()
solver = SolverFactory('gurobi')
results = solver.solve(model, tee=
True)
Gurobi_solver_end_time =
time.time()

# # Print the results
# model.display()
# print("Objective Value:",
model.objective_function())
```

Time Comparison:

```
# Calculate the elapsed time
Gurobi_solver_elapsed_time =
Gurobi_solver_end_time -
Gurobi_solver_start_time
# Print the elapsed time
print("Time taken for Gurobi
solver:",
Gurobi_solver_elapsed_time,
"seconds")

# Check time for Cplex solver

Cplex_solver_start_time =
time.time()
```

```
solver =
SolverFactory('cplex_direct')
results = solver.solve(model)
Cplex_solver_end_time =
time.time()
# Calculate the elapsed time
Cplex_solver_elapsed_time =
Cplex_solver_end_time -
Cplex_solver_start_time
# Print the elapsed time
print("Time taken for cplex
solver:",
Cplex_solver_elapsed_time,
"seconds")

# Check time for glpk solver GNU
Linear Programming Kit

glpk_solver_start_time =
time.time()
solver = SolverFactory('glpk')
results = solver.solve(model)
glpk_solver_end_time = time.time()
# Calculate the elapsed time
glpk_solver_elapsed_time =
glpk_solver_end_time -
glpk_solver_start_time
# Print the elapsed time
print("Time taken for glpk
solver:",
glpk_solver_elapsed_time,
"seconds")
```

Graph:

```
import plotly.graph_objects as go

solver_list = ['Gurobi', 'CPLEX',
'GLPK']
time_list =
[Gurobi_solver_elapsed_time,
Cplex_solver_elapsed_time,
glpk_solver_elapsed_time]
```

```
bar_colors = ['rgb(31, 119, 180)',
'rgb(255, 127, 14)', 'rgb(44, 160,
44)', 'rgb(214, 39, 40)']
fig =
go.Figure(data=go.Bar(x=solver_lis
t, y=time_list,
marker_color=bar_colors))
fig.update_layout(
    title="Solver Execution Time",
    xaxis_title="Solver",
    yaxis_title="Execution Time
(seconds)",
    width=800,
    height=500
)

# Show the graph
fig.show()
```

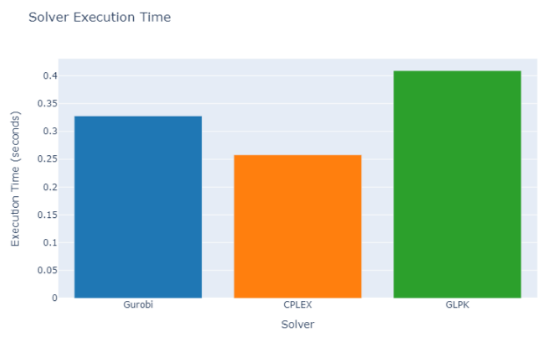


Image 6.1 Time Comparison

Chapter 7 Sensitivity Analysis

7.1 Shadow Price

In this section I want to check how the affect the price when I will add the 1 unit into the right-hand side of the ith constraint. Which basically shows what amount will change in the objective function value when we add 1 unit in one of the model constraints.

- Check output on program file.

```
# Optimize the model
Hogwarts_Juice_Factory.optimize()

# Check the optimization result
if Hogwarts_Juice_Factory.status
== GRB.OPTIMAL:
    # Print the objective function
value
    objective_value =
Hogwarts_Juice_Factory.objVal
    print("Objective Function
Value:", objective_value)

    # Print the shadow prices
    print("\nShadow Prices:")
    for constraint in
Hogwarts_Juice_Factory.getConstrs(
):
        shadow_price =
constraint.Pi
        print(constraint.ConstrNam
e, ":", "%.3f" %
round(shadow_price, 3))

else:
    print("The problem is
infeasible or unbounded.")
```

7.2 Add new constraints

Here in the Elves community the Senior Elves did the strict, they put their demand. And says that they want to spend more time with family and do not want to do overtime after their standard time. So now Professor Snape decided to remove overtime after the shift. So based on this decision I update the LP model. Check model on program file.

New add constraint:

```
No_Overtime =
Hogwarts_Juice_Factory_No_Overtime
.addConstrs(Overtime_Elves[Time_Sl
ots[4], Day] +
Overtime_Elves[Time_Slots[5],
Day]  == 0

                                for Day
in Days)
```

Output:

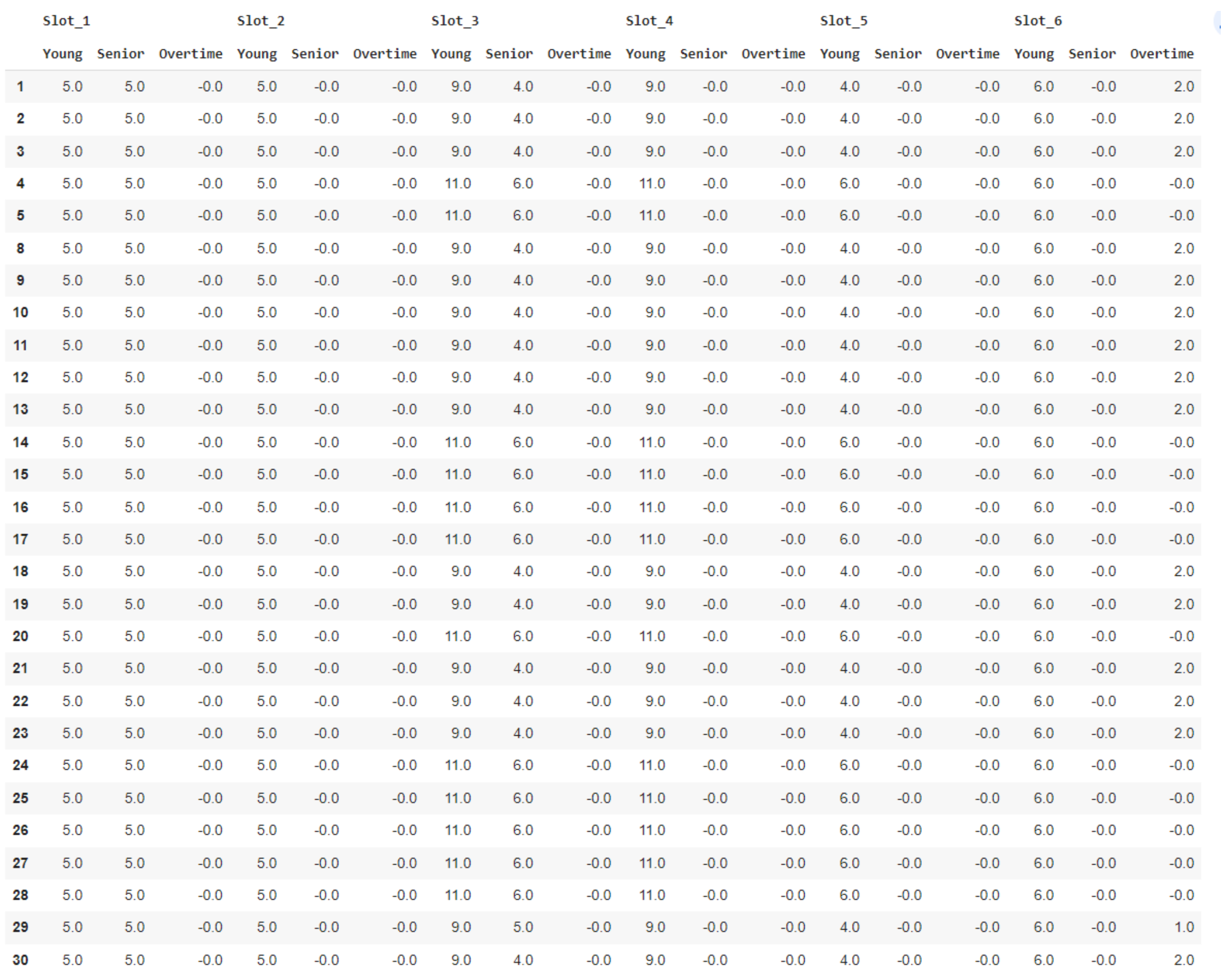
```
Optimal objective  6.958000000e+03
      Objective Function Value:
      6958.0
```

Chapter 8 Integer Programming

This problem can only be solved by integer programming. Because we see that there are workers who do their job. If we consider that solution as reference only but we cannot implement that. Because there is no fraction worker available in real life. So, we must do that kind of problem into integer programming but in the previous chapter we did all things by considering linear programming, there is the only reason that we can understand what actual linear programming is do.

```
Hogwarts_Juice_Factory_Integer=
gp.Model()

Young_Elves =
Hogwarts_Juice_Factory_Integer.add
Vars(Time_Slots, Days,
vtype=gp.GRB.INTEGER,
name="Young_Elves")
Senior_Elves =
Hogwarts_Juice_Factory_Integer.add
Vars(Time_Slots, Days,
vtype=gp.GRB.INTEGER,
name="Senior_Elves")
Overtime_Elves =
Hogwarts_Juice_Factory_Integer.add
Vars(Time_Slots, Days,
vtype=gp.GRB.INTEGER,
name="Overtime_Elves")

Objective_Function = gp.quicksum(2
* Young_Elves[Time_Slot, Day] + 16
* Senior_Elves[Time_Slot, Day] + 6
* Overtime_Elves[Time_Slot, Day]

for Time_Slot in Time_Slots for
Day in Days)
Hogwarts_Juice_Factory_Integer.set
Objective(Objective_Function,
gp.GRB.MINIMIZE)
```

```
Slot_6to8 =
Hogwarts_Juice_Factory_Integer.add
Constrs(2 *
Young_Elves[Time_Slots[0], Day] +
2 * Senior_Elves[Time_Slots[0],
Day] >= 2 * 10 + 0 * 3

            for Day in Days)

Slot_6to10 =
Hogwarts_Juice_Factory_Integer.add
Constrs(2 *
Young_Elves[Time_Slots[1], Day] +
2 * Senior_Elves[Time_Slots[0],
Day] + 2 *
Senior_Elves[Time_Slots[1], Day]
>= 0 * 10 + 2 * 3

            for Day in Days)

Slot_10to12 =
Hogwarts_Juice_Factory_Integer.add
Constrs(2 *
Young_Elves[Time_Slots[2], Day] +
2 * Senior_Elves[Time_Slots[0],
Day] + 2 *
Senior_Elves[Time_Slots[1], Day] +
2 * Senior_Elves[Time_Slots[2],
Day] >= 2 * 10 + 1 * 3

            for Day in Days)

Slot_12to2 =
Hogwarts_Juice_Factory_Integer.add
Constrs(2 *
Young_Elves[Time_Slots[3], Day] +
2 * Senior_Elves[Time_Slots[0],
Day] + 2 *
Senior_Elves[Time_Slots[1], Day]+
2 * Senior_Elves[Time_Slots[2],
Day] >= 0 * 10 + 0 * 3

            for Day in Days)
```



	Slot_1			Slot_2			Slot_3			Slot_4			Slot_5			Slot_6		
	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime	Young	Senior	Overtime
1	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
2	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
3	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
4	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
5	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
8	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
9	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
10	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
11	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
12	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
13	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
14	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
15	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
16	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
17	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
18	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
19	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
20	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
21	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
22	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
23	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0
24	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
25	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
26	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
27	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
28	5.0	5.0	-0.0	5.0	-0.0	-0.0	11.0	6.0	-0.0	11.0	-0.0	-0.0	6.0	-0.0	-0.0	6.0	-0.0	-0.0
29	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	5.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	1.0
30	5.0	5.0	-0.0	5.0	-0.0	-0.0	9.0	4.0	-0.0	9.0	-0.0	-0.0	4.0	-0.0	-0.0	6.0	-0.0	2.0

Image 8.1 Integer Programming output

```
Slot_2to4 =
Hogwarts_Juice_Factory_Integer.add
Constrs(2 *
Young_Elves[Time_Slots[4], Day] +
2 * Senior_Elves[Time_Slots[1],
Day] + 2 *
Senior_Elves[Time_Slots[2], Day] +
2 * Overtime_Elves[Time_Slots[4],
Day] >= 1 * 10 + 1 * 3

            for Day in Days)

Slot_4to6 =
Hogwarts_Juice_Factory_Integer.add
Constrs(2 *
Young_Elves[Time_Slots[5], Day] +
2 * Senior_Elves[Time_Slots[2],
Day] + 2 *
```

```
Overtime_Elves[Time_Slots[5], Day]
>= 2 * 10 + 1 * 3

            for Day in Days)

Felix_Felici =
Hogwarts_Juice_Factory_Integer.add
Constr(gp.quicksum(2 *
Young_Elves[Time_Slot, Day] + 8 *
Senior_Elves[Time_Slot, Day] + 2 *
Overtime_Elves[Time_Slot, Day]

                           for
Time_Slot in Time_Slots for Day in
Days) >= (7 * 10 * 30 + 400 * 5 +
5 * 3 * 30))
```

```
Watch_1 =
Hogwarts_Juice_Factory_Integer.add
Constrs(Young_Elves[Time_Slots[0],
Day] <=
Senior_Elves[Time_Slots[0], Day]

          for Day in Days)

Watch_2 =
Hogwarts_Juice_Factory_Integer.add
Constrs(Young_Elves[Time_Slots[1],
Day] <=
Senior_Elves[Time_Slots[0], Day] +
Senior_Elves[Time_Slots[1], Day]

          for Day in Days)

Watch_3 =
Hogwarts_Juice_Factory_Integer.add
Constrs(Young_Elves[Time_Slots[2],
Day] <=
Senior_Elves[Time_Slots[0], Day] +
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day]

          for Day in Days)

Watch_4 =
Hogwarts_Juice_Factory_Integer.add
Constrs(Young_Elves[Time_Slots[3],
Day] <=
Senior_Elves[Time_Slots[0], Day] +
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day]

          for Day in Days)

Watch_5 =
Hogwarts_Juice_Factory_Integer.add
Constrs(Young_Elves[Time_Slots[4],
Day] <=
Senior_Elves[Time_Slots[1], Day] +
Senior_Elves[Time_Slots[2], Day] +
Overtime_Elves[Time_Slots[4], Day]

          for Day in Days)

Watch_6 =
Hogwarts_Juice_Factory_Integer.add
Constrs(Young_Elves[Time_Slots[5],
Day] <=
Senior_Elves[Time_Slots[2], Day] +
Overtime_Elves[Time_Slots[5], Day]

          for Day in Days)

Hogwarts_Juice_Factory_Integer.opt
imize()
objective_value =
Hogwarts_Juice_Factory_Integer.obj
Val
print("Objective Function Value:",
objective_value)

# for v in
Hogwarts_Juice_Factory_Integer.get
Vars():
        #print(v.varName, v.x)

# Create data dictionaries for
Young Elves, Senior Elves, and
Overtime Elves
Wokers_Requirements_Integer = {}

# Populate data dictionaries with
the respective values
for Time_Slot in Time_Slots:
    young = {}
    senior = {}
    overtime = {}
    for Day in Days:
        young[Day] =
Young_Elves[Time_Slot, Day].x
        senior[Day] =
Senior_Elves[Time_Slot, Day].x
        overtime[Day]=
Overtime_Elves[Time_Slot, Day].x
    Wokers_Requirements_Integer[Ti
me_Slot] = {'Young': young,
'Senior': senior, 'Overtime':
overtime}

# Create dataframes from the data
dictionaries
Factory_Schedule_Integer_df =
pd.DataFrame(Wokers_Requirements_I
nteger, index=['Young Elves'])

# Transpose the dataframes to have
time slots as rows and days as
columns
Factory_Schedule_Integer_df =
Factory_Schedule_Integer_df.T

# Create an empty dataframe for
Young Elves
Factory_Schedule_Integer_df =
pd.DataFrame.from_dict({(i,j):
Wokers_Requirements_Integer[i][j]

      for i in
Wokers_Requirements_Integer.keys()

      for j in
Wokers_Requirements[i].keys()},

      orient='index')

Factory_Schedule_Integer_df.T
```

Output:

```
Optimal solution found (tolerance
1.00e-04)
Best objective 6.890000000000e+03,
best bound 6.890000000000e+03, gap
0.0000%
Objective Function Value: 6890.0
```

Chapter 9 Conclusion

Finally, by analyzing the whole report and each chapter I can say that by using each parameter of linear and integer programming we can solve any kind of problem.