Math 5593 Linear Programming Problem Set 1

Department of Mathematical and Statistical Sciences

University of Colorado Denver, Fall 2013

Solutions to AMPL Exercises

Initial Remarks About These Solutions and Some General Tips & Tricks As you have completed this problem set, you will have realized that a good organization of your model and data files are of crucial importance to not lose oversight or control of your work. I have found the following approach quite useful.

- Create a new AMPL folder and put all your related files into their own, tidy place. Do not throw your files together with ampl and solver executables onto your desktop or another, similarly wild location!
- Make sure that you can easily open your model and data files using a text editor, and that your computer shows the actual file extensions. There were several cases (especially using newer versions of Windows) where a folder showed a file `model.mod` or `data.dat` but AMPL complained that such a file could not be found, because the actual file name was `model.mod.txt` or `data.dat.txt` and Windows was set to hide extensions for certain (supposedly) known file types. Please, show your file extensions!
- Before modifying a working model or data file, always create a new copy and give it a descriptive name!
- Try hard to minimize the amount you have to type when working with AMPL (and in general). There is no need to repeatedly write `model model.mod; data data.dat; solve;` – on most computers, you can use the arrow keys to repeat previously typed commands; or even better, put all these into their own file (I usually call these “run” files) and use the single command `ampl modeldatasolve.run` or `ampl modeldatasolve.run > output.out` to write the AMPL output into a new file `output.out` (of course, you don’t have to use the `.run` or `.out` extensions and can choose whatever you like). This also makes it very easy to include your solutions in a LaTeX file if you were to choose to do so. And yes, it will take some time to learn at first but eventually save you a huge amount of time – you’ll see!

AMPL Book Exercises 1-2/1-3 For ease of comparison, we include the initial model and data below.

```
set PROD;    # products
set STAGE;   # stages

param rate {PROD,STAGE} > 0; # tons per hour in each stage
param avail {STAGE} >= 0;    # hours available/week in each stage
param profit {PROD};         # profit per ton

param commit {PROD} >= 0;    # lower limit on tons sold in week
param market {PROD} >= 0;    # upper limit on tons sold in week

var Make {p in PROD} >= commit[p], <= market[p]; # tons produced

maximize Total_Profit: sum {p in PROD} profit[p] * Make[p];

                # Objective: total profits from all products

subject to Time {s in STAGE}:
   sum {p in PROD} (1/rate[p,s]) * Make[p] <= avail[s];

                # In each stage: total of hours used by all
                # products may not exceed hours available
```

File 1: steel4.mod

```
set PROD := bands coils plate;
set STAGE := reheat roll;

param rate:   reheat   roll :=
  bands         200     200
  coils         200     140
  plate         200     160  ;

param:      profit   commit   market :=
  bands       25       1000     6000
  coils       30        500     4000
  plate       29        750     3500  ;

param avail :=  reheat 35   roll   40 ;
```

File 2: steel4.dat

To solve this model using the initial data and display its solution, we write a new "run" file as follows:

```
model steel4.mod; data steel4.dat; solve; display Make;
```

File 3: steel4.run

Using the command `ampl steel4.run > steel4.out` from the command prompt (i.e., we do not need to actually start or "go into" ampl), we then create the new file `steel4.out` with the following solution output:

```
MINOS 5.5: optimal solution found.
4 iterations, objective 190071.4286
Make [*] :=
bands  3357.14
coils   500
plate  3142.86
;
```

File 4: steel4.out

1-2(a) We modify the model and run file as shown below and resolve (`ampl steel4a.run > steel4a.out`). Here note that File 5 does not repeat the full model, but only those lines from File 1 that have changed:

```
subject to Time {s in STAGE}:
   sum {p in PROD} (1/rate[p,s]) * Make[p] = avail[s];

                # In each stage: total of hours used by all
                # products MUST EQUAL the hours available
```

File 5: steel4a.mod

```
model steel4a.mod; data steel4.dat; solve; display Make;
```

File 6: steel4a.run

```
MINOS 5.5: optimal solution found.
2 iterations, objective 190071.4286
Make [*] :=
bands  3357.14
coils   500
plate  3142.86
;
```

File 7: steel4a.out

Comparing Files 4 and 7, we see that the solution is the same: because hours that remain available could be used to increase production and profit, the time constraint will generally hold with equality.

1-2(b) We modify model, data, and run file as shown below and resolve (`ampl steel4b.run > steel4b.out`):

```
param max_weight;            # upper limits on tons in total

subject to Total_Weight: sum {p in PROD} Make[p] <= max_weight;
```

File 8: steel4b.mod

```
param max_weight := 6500;
```

File 9: steel4b.dat

```
model steel4b.mod; data steel4b.dat; solve; display Make;
```

File 10: steel4b.run

```
MINOS 5.5: optimal solution found.
3 iterations, objective 183791.6667
Make [*] :=
bands  1541.67
coils  1458.33
plate  3500
;
```

File 11: steel4b.out

Comparing Files 4 and 11, we see that production of bands decreases from 3,357.15 to 1,541.67 tons whereas coils and plates increase from 500 to 1,458.33 tons and from 3,142.86 to 3,500 tons, respectively. In total, this reduces production from 7,000 to 6,500 tons, and profit from \$190,071.43 to \$183,791.67.

1-2(c) We modify model and run file as shown below and resolve (`ampl steel4c.run > steel4c.out`):

```
maximize Total_Make: sum {p in PROD} Make[p];

                # Objective: total make from all products
```

File 12: steel4c.mod

```
model steel4c.mod; data steel4.dat; solve; display Make;
```

File 13: steel4c.run

```
MINOS 5.5: optimal solution found.
1 iterations, objective 7000
Make [*] :=
bands  5750
coils   500
plate   750
;
```

File 14: steel4c.out

Comparing Files 4 and 14, we see that the individual production amounts have changed although the total production is still 7,000 tons. This implies that there are multiple optimal solutions that maximize total production, although their total profit may be different; in fact, the total profit for the solution in File 14 is only $180,500 which is less than the profit for each of the two previous solutions.

1-2(d) We modify model, data, and run file as shown below and resolve (`ampl steel4d.run > steel4d.out`). Here note that the new `Min_Share` constraint in Lines 13-15 in File 15 does not replace the original objective in Lines 13-15 in File 1 (which simply moves further to the bottom); the placement of this constraint before the objective was merely a choice of convenience to show as little new code as possible.

```
8  param share {PROD} >= 0;      # minimum share of total tons produced
9  param market {PROD} >= 0;     # upper limit on tons sold in week
10
11 var Make {p in PROD} <= market[p]; # tons produced
12
13 subject to Min_Share {p in PROD}: Make[p] >= share[p] * sum{pp in PROD} Make[pp];
14
15 # Note that you must distinguish between constraint index (p) and sum index (pp).
```

File 15: steel4d.mod

```
param:     profit  share  market :=
  bands      25     0.4    6000
  coils      30     0.1    4000
  plate      29     0.4    3500  ;
```

File 16: steel4d.dat

```
model steel4d.mod; data steel4d.dat; solve; display Make;
```

File 17: steel4d.run

```
MINOS 5.5: optimal solution found.
3 iterations, objective 189700
Make [*] :=
bands  3500
coils   700
plate  2800
;
```

File 18: steel4d.out

Comparing Files 4 and 18, we see that the production amounts of bands, coils, and plates have changed from 3,357.15 (47.96%), 500 (7.14%), and 3,142.86 (44.90%) tons to 3,500 (50%), 700 (10%), and 2,800 (40%) tons, respectively, for a total profit of $189,700 still at the maximum production of 7,000 tons.

Finally, we repeat our solution with new minimum shares of 0.5, 0.1, and 0.5, adding to 110% in total!

```
param:     profit  share  market :=
  bands      25     0.5    6000
  coils      30     0.1    4000
  plate      29     0.5    3500  ;
```

File 19: steel4dd.dat

```
model steel4d.mod; data steel4dd.dat; solve; display Make;
```

File 20: steel4dd.run

```
MINOS 5.5: optimal solution found.
2 iterations, objective -3.873449243e-24
Make [*] :=
bands  -4.45224e-26
coils  -1.11306e-26
plate  -6.67836e-26
;
```

File 21: steel4dd.out

Clearly, a `Min_Share` constraint to produce a sum of at least 110% of total production can only be satisfied if that total production is zero, and values of $\pm 10^{-26}$ – for all practical matters – are zero.

1-2(e) We modify data and run file as shown below and resolve (`ampl steel4e.run > steel4e.out`):

```
set STAGE := reheat roll finish;

param rate:  reheat  roll    finish  :=
  bands        200    200   Infinity
  coils        200    140   Infinity
  plate        200    160      150;

param avail :=  reheat 35  roll 40  finish 20 ;
```

File 22: steel4e.dat

```
model steel4.mod; data steel4e.dat; solve; display Make;
```

File 23: steel4e.run

```
MINOS 5.5: optimal solution found.
3 iterations, objective 189916.6667
Make [*] :=
bands  3416.67
coils   583.333
plate  3000
;
```

File 24: steel4e.out

In File 22, we use the keyword `Infinity` to declare infinite finishing rates for bands and coils which ensures that the available capacity of 20 hours is used exclusively during the finishing stage of plates.

1-3(a) We start with the following model and data file, and its solution (`ampl steel3.run > steel3.out`):

```
set PROD;  # products

param rate {PROD} > 0;      # produced tons per hour
param avail >= 0;           # hours available in week
param profit {PROD};        # profit per ton
param commit {PROD} >= 0;   # lower limit on tons sold in week
param market {PROD} >= 0;   # upper limit on tons sold in week

var Make {p in PROD} >= commit[p], <= market[p]; # tons produced

maximize Total_Profit: sum {p in PROD} profit[p] * Make[p];

                # Objective: total profits from all products

subject to Time: sum {p in PROD} (1/rate[p]) * Make[p] <= avail;

                # Constraint: total of hours used by all
                # products may not exceed hours available
```

File 25: steel3.mod

```
set PROD := bands coils plate;

param:     rate  profit  commit  market :=
  bands     200    25     1000    6000
  coils     140    30      500    4000
  plate     160    29      750    3500 ;

param avail := 40;
```

File 26: steel3.dat

```
model steel3.mod; data steel3.dat; solve; display Time, Make.rc;
```

File 27: steel3.run

```
MINOS 5.5: optimal solution found.
2 iterations, objective 194828.5714
Time = 4640

Make.rc [*] :=
bands    1.8
coils   -3.14286
plate    0
;
```

File 28: steel3.out

As discussed in Section 1.6, AMPL interprets a constraint's name as reference to its associated dual value or shadow price, that predicts by how much the optimal objective value will improve (or worsen) if the constraint were relaxed (or tightened) by a small amount. Similarly, a variable's name appended by `.lb`, `.ub`, or `.rc` corresponds to the dual values or shadow prices of the constraint associated with that variable's lower bound (e.g., ≥ 0 for nonnegative variables, $\geq -\infty$ for free variables), its upper bound (e.g., ≤ 0 for nonpositive variables, $\leq \infty$ for free variables), or its "reduced cost" (i.e., the expected change in the objective – usually a cost reduction – following a small change in the variable's value). Hence, File 28 tells us that up to some (unknown) point, additional rolling time would bring in another \$4,640 of extra profit per hour, higher market demand in bands would yield another \$1.8 per ton, lower (!) commitment of coils would yield an extra \$3.14286 per ton (note that the shadow price is negative!), and (small) changes to either demand or commitment of plates would not affect the current profit; of course, changes in the opposite directions would decrease the profit correspondingly.

1-3(b) Computing the profit rates (in dollars per hour) for both the reheat stage and the rolling stage, we get

profit rate (\$/hour)	bands	coils	plate
reheat	5000	6000	5800
roll	5000	4200	4640

These rates indicate that bands remain the most resource-efficient product during the rolling stage whereas coils and plates are more resource-efficient during the reheating stage. In particular, the efficiency gains of plates (and coils) over bands for reheating exceed their efficiency losses during rolling, resulting in a higher production of plates to be compensated by a lower production of bands.

1-3(c) Using Files 1 (steel4.mod) and 2 (steel4.mod), we solve `ampl run steel3c.run > steel3c.out`:

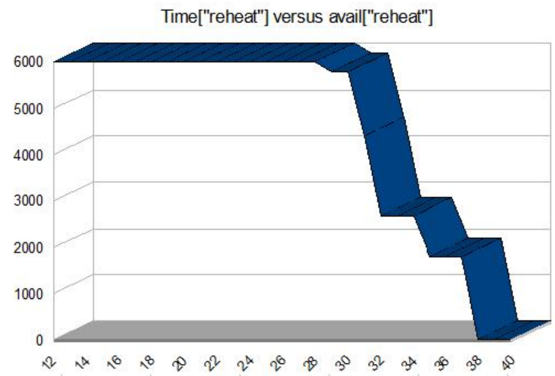
```
model steel4.mod; data steel4.dat; solve;

let avail["reheat"] := 36; solve;
let avail["reheat"] := 37; solve;
let avail["reheat"] := 38; solve;
let avail["reheat"] := 38.1; solve;
```

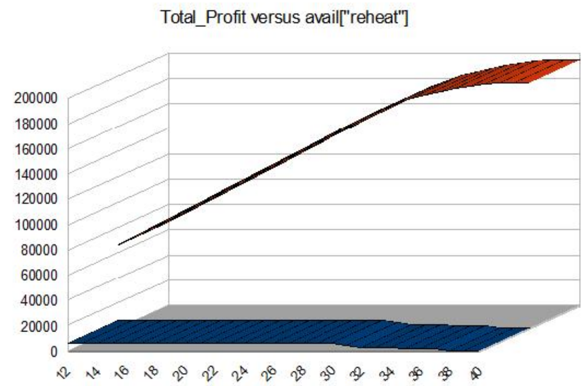
File 29: steel3c.run

```
MINOS 5.5: optimal solution found.
4 iterations, objective 190071.4286
MINOS 5.5: optimal solution found.
1 iterations, objective 191871.4286
MINOS 5.5: optimal solution found.
1 iterations, objective 193671.4286
MINOS 5.5: optimal solution found.
1 iterations, objective 194828.5714
MINOS 5.5: optimal solution found.
0 iterations, objective 194828.5714
```

File 30: steel3c.out

File 30 confirms that the initial profit of \$190,071.43 at 35 hours increases by \$1,800 to \$191,871.43 at 36 hours, by another \$1,800 to \$193,671.43 at 37 hours, by only \$1,157.14 to \$194,828.57 at 38 hours, and then stays constant. Corresponding plots of time and profit versus reheat hours are given below.





1-3(d) Similar to what is shown in the plots above, the following table lists the reduced costs or (dual) shadow prices of `Time["reheat"]` together with the total profit for different integer values of `avail["reheat"]`.

`avail["reheat"]`	12	13	14	15	16	17	18	19	20
`Time["reheat"]`	6000	6000	6000	6000	6000	6000	6000	6000	6000
`Total_Profit`	66250	72250	78250	84250	90250	96250	102250	108250	114250
21	22	23	24	25	26	27	28	29	30
6000	6000	6000	6000	6000	6000	6000	5800	5800	5800
120250	126250	132250	138250	144250	150250	156250	162250	168200	174000
31	32	33	34	35	36	37	38	39	40
4400	2667	2667	2667	1800	1800	1800	0	0	0
178600	182458	185125	187792	190071	191871	193671	194829	194829	194829

To drop reheat time to 11 hours, we change the data and resolve (`ampl steel3d.run > steel3d.out`):

```
model steel4.mod; data steel4.dat; let avail["reheat"] := 11; solve;
```

File 31: steel3d.run

```
presolve: constraint Time['reheat'] cannot hold:
  body <= 11 cannot be >= 11.25; difference = -0.25
```

File 32: steel3d.out

This error message tells us that our problem has become infeasible, and it even hints at the reason: the production of our committed minimum quantities (1,000 tons bands, 500 tons coils, and 750 tons plates) with reheat rates of 200 tons/hour each already require 2250/200 = 11.25 hours of reheat time.

AMPL Book Exercise 2-5 The following model was already given in class: Let S be the set of schedules with $a_{ij} = 1$ if schedule j includes work on day i and $a_{ij} = 0$ otherwise. Let b_i be the minimum number of kitchen employees needed on day i, and let x_j be the number of employees to be hired onto schedule j.

$$\begin{aligned} &\text{minimize} \sum\nolimits_{j \in S} x_j \\ &\text{subject to} \sum\nolimits_{j \in S} a_{ij} x_j \geq b_i && \text{for all days } i \\ &x_j \geq 0 && \text{for all } j \in S \end{aligned}$$

2-5(a) The following model and data file implement the model using more descriptive set and variable names:

```
set DAYS;    # set of work days
set WORK;    # set of work schedules

param work_on_day {DAYS,WORK} binary;
param work_demand {DAYS} >= 0;

var Employees {WORK} >= 0;

minimize Total_Employees: sum {w in WORK} Employees[w];

subject to Work_Demand {d in DAYS}:
    sum {w in WORK} work_on_day[d,w] * Employees[w] >= work_demand[d];
```

File 33: worksched.mod

```
set DAYS := Mon Tue Wed Thu Fri Sat Sun;

set WORK := MonTueWedThuSat MonTueWedThuSun
            MonTueWedFriSat MonTueWedFriSun
            MonTueThuFriSat MonTueThuFriSun
            MonWedThuFriSat MonWedThuFriSun
            TueWedThuFriSat TueWedThuFriSun;

param work_on_day (tr): Mon Tue Wed Thu Fri Sat Sun :=

       MonTueWedThuSat   1   1   1   1   0   1   0
       MonTueWedThuSun   1   1   1   1   0   0   1
       MonTueWedFriSat   1   1   1   0   1   1   0
       MonTueWedFriSun   1   1   1   0   1   0   1
       MonTueThuFriSat   1   1   0   1   1   1   0
       MonTueThuFriSun   1   1   0   1   1   0   1
       MonWedThuFriSat   1   0   1   1   1   1   0
       MonWedThuFriSun   1   0   1   1   1   0   1
       TueWedThuFriSat   0   1   1   1   1   1   0
       TueWedThuFriSun   0   1   1   1   1   0   1;

param work_demand := Mon 45 Tue 45 Wed 40 Thu 50 Fri 65 Sat 35 Sun 35;
```

File 34: worksched.dat

```
model worksched.mod; data worksched.dat; solve; option omit_zero_rows 1;
display Employees; display Work_Demand.slack;
```

File 35: worksched.run

Note that we have set the option `omit_zero_rows` to show only those schedules that are actually used.

```
MINOS 5.5: optimal solution found.
3 iterations, objective 70
Employees [*] :=
MonTueThuFriSat  20
MonTueWedFriSat  15
MonTueWedFriSun   5
MonTueWedThuSun   5
TueWedThuFriSun  25
;

Work_Demand.slack [*] :=
Tue  25
Wed  10
;
```

File 36: worksched.out

Note that we have also displayed the slack residuals associated with each of the `Work_Demand` constraints, indicating that the suggested schedule accommodates up to 25 slackers on Tuesday and 10 slackers on Wednesday. Of course, a better schedule would still use 70 employees in total but try to distribute these slacks more equally among all days - maybe you can think about how this can be done?

2-5(b) Here, we only need to read, understand, and then reword the book's discussion: If we consider the general blending (input-output) model and let the inputs and outputs be the work schedules and corresponding days of work, respectively, then the decision variable is the number of workers hired onto each schedule and the objective to minimize cost becomes an objective to minimize workers. The constraints simply say that on each day, we must have hired enough workers to cover our work demand.

AMPL Book Exercise 4-5 This problem deals with the multiperiod production model shown below:

```
set PROD;      # products
param T > 0;   # number of weeks

param rate {PROD} > 0;           # tons per hour produced
param inv0 {PROD} >= 0;          # initial inventory
param avail {1..T} >= 0;         # hours available in week
param market {PROD,1..T} >= 0;   # limit on tons sold in week

param prodcost {PROD} >= 0;      # cost per ton produced
param invcost {PROD} >= 0;       # carrying cost/ton of inventory
param revenue {PROD,1..T} >= 0;  # revenue per ton sold

var Make {PROD,1..T} >= 0;       # tons produced
var Inv {PROD,0..T} >= 0;        # tons inventoried
var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t]; # tons sold
```

```

maximize Total_Profit:
   sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
       prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t]);

                # Total revenue less costs in all weeks

subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];

                # Total of hours used by all products
                # may not exceed hours available, in each week

subject to Init_Inv {p in PROD}:  Inv[p,0] = inv0[p];

                # Initial inventory must equal given value

subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];

                # Tons produced and taken from inventory
                # must equal tons sold and put into inventory
```

File 37: steelT.mod

4-5(a) We first create three separate data files (we only show the different revenue data for Files 39 and 40):

```
param T := 4;
set PROD := bands coils;

param avail :=  1 40   2 40   3 32   4 40 ;

param rate :=   bands 200    coils 140 ;
param inv0 :=   bands  10    coils   0 ;

param prodcost :=  bands 10     coils  11 ;
param invcost  :=  bands  2.5   coils   3 ;

param revenue:    1      2      3      4 :=
        bands    25     26     27     27
        coils    30     35     37     39 ;

param market:     1      2      3      4 :=
        bands  6000   6000   4000   6500
        coils  4000   2500   3500   4200 ;
```

File 38: steelT(scenario1).dat

```
param revenue:    1      2      3      4 :=
        bands    23     24     25     25
        coils    30     33     35     36 ;
```

File 39: steelT(scenario2).dat

```
param revenue:    1      2      3      4 :=
        bands    21     27     33     35
        coils    30     32     33     33 ;
```

File 40: steelT(scenario3).dat

```
model steelT.mod; data steelT(scenario1).dat; solve; display Inv, Make, Sell; reset;
model steelT.mod; data steelT(scenario2).dat; solve; display Inv, Make, Sell; reset;
model steelT.mod; data steelT(scenario3).dat; solve; display Inv, Make, Sell;
```

File 41: steelT.run

```
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
:          Inv    Make    Sell    :=
bands 0     10      .       .
bands 1      0    5990    6000
bands 2      0    6000    6000
bands 3      0    1400    1400
bands 4      0    2000    2000
coils 0      0      .       .
coils 1   1100    1407     307
coils 2      0    1400    2500
coils 3      0    3500    3500
coils 4      0    4200    4200
;
```

```

MINOS 5.5: optimal solution found.
15 iterations, objective 462944.2857
:          Inv    Make      Sell     :=
bands 0    10       .         .
bands 1     0    2285.71   2295.71
bands 2     0    4428.57   4428.57
bands 3     0    1400      1400
bands 4     0    2000      2000
coils 0     0       .         .
coils 1     0    4000      4000
coils 2     0    2500      2500
coils 3     0    3500      3500
coils 4     0    4200      4200
;

MINOS 5.5: optimal solution found.
19 iterations, objective 549970
:            Inv    Make   Sell    :=
bands 0      10      .      .
bands 1       0      0     10
bands 2       0    6000   6000
bands 3       0    4000   4000
bands 4       0    6500   6500
coils 0       0      .      .
coils 1    1600    5600   4000
coils 2     500    1400   2500
coils 3       0    1680   2180
coils 4       0    1050   1050
;
```

File 42: steelT.out

We see that the optimal production of bands, coils, and plates for each of the three scenarios is different.

4-5(b) We follow the instructions and combine all three scenarios into a single multiscenario formulation, in which we also duplicate all variables and constraints. In addition, we introduce an auxiliary `Profit` variable to facilitate the tracking of scenario profits. Of course, we obtain the same solutions as before.

```
param S > 0;  # number of scenarios

param prob {1..S} >= 0, <= 1;
    check: 0.99999 < sum {s in 1..S} prob[s] < 1.00001;

param revenue {PROD,1..T,1..S} >= 0;  # revenue per ton sold

var Make {PROD,1..T,1..S} >= 0;        # tons produced
var Inv {PROD,0..T,1..S} >= 0;         # tons inventoried
var Sell {p in PROD, t in 1..T, s in 1..S} >= 0, <= market[p,t]; # tons sold

var Profit {s in 1..S }                # scenario profit (for easier tracking)
    = sum {p in PROD, t in 1..T} (revenue[p,t,s] * Sell[p,t,s]
      - prodcost[p] * Make[p,t,s] - invcost[p] * Inv[p,t,s]);

                # Expected revenue less costs in all weeks

maximize Expected_Profit: sum {s in 1..S} prob[s] * Profit[s];

subject to Time {t in 1..T, s in 1..S}:
   sum {p in PROD} (1/rate[p]) * Make[p,t,s] <= avail[t];

                # Total of hours used by all products
                # may not exceed hours available, in each week and scenario

subject to Init_Inv {p in PROD, s in 1..S}:  Inv[p,0,s] = inv0[p];

                # Initial inventory must equal given value

subject to Balance {p in PROD, t in 1..T, s in 1..S}:
   Make[p,t,s] + Inv[p,t-1,s] = Sell[p,t,s] + Inv[p,t,s];

                # Tons produced and taken from inventory
                # must equal tons sold and put into inventory
```

File 43: steelT(multiscenario).mod

```
param S := 3;

param prob := 1 0.45   2 0.35   3 0.20 ;

param revenue :=

```

```
[*,*,1]:    1     2     3     4 :=
 bands     25    26    27    27
 coils     30    35    37    39

[*,*,2]:    1     2     3     4 :=
 bands     23    24    25    25
 coils     30    33    35    36

[*,*,3]:    1     2     3     4 :=
 bands     21    27    33    35
 coils     30    32    33    33 ;
```

File 44: steelT(multiscenario).dat

```
model steelT(multiscenario).mod; data steelT(multiscenario).dat; solve;
display Inv, Make, Sell;
```

File 45: steelT(multiscenario).run

```
MINOS 5.5: optimal solution found.
49 iterations, objective 503789.35
:            Inv     Make      Sell      :=
bands 0 1     10       .          .
bands 0 2     10       .          .
bands 0 3     10       .          .
bands 1 1      0    5990       6000
bands 1 2      0    2285.71    2295.71
bands 1 3      0       0         10
bands 2 1      0    6000       6000
bands 2 2      0    4428.57    4428.57
bands 2 3      0    6000       6000
bands 3 1      0    1400       1400
bands 3 2      0    1400       1400
bands 3 3      0    4000       4000
bands 4 1      0    2000       2000
bands 4 2      0    2000       2000
bands 4 3      0    6500       6500
coils 0 1      0       .          .
coils 0 2      0       .          .
coils 0 3      0       .          .
coils 1 1   1100    1407        307
coils 1 2      0    4000       4000
coils 1 3   1600    5600       4000
coils 2 1      0    1400       2500
coils 2 2      0    2500       2500
coils 2 3    500    1400       2500
coils 3 1      0    3500       3500
coils 3 2      0    3500       3500
coils 3 3      0    1680       2180
coils 4 1      0    4200       4200
coils 4 2      0    4200       4200
coils 4 3      0    1050       1050
;
```

File 46: steelT(multiscenario).out

4-5(c) We continue to follow the instructions and add "nonanticipativity" constraints for the first time period.

```
subject to Make_na {p in PROD, s in 1..S-1}:
   Make[p,1,s] = Make[p,1,s+1];

subject to Inv_na {p in PROD, s in 1..S-1}:
   Inv[p,1,s] = Inv[p,1,s+1];

subject to Sell_na {p in PROD, s in 1..S-1}:
   Sell[p,1,s] = Sell[p,1,s+1];
```

File 47: steelT(nonanticipativity).mod

```
model steelT(nonanticipativity).mod; data steelT(multiscenario).dat; solve;
display Inv, Make, Sell;
```

File 48: steelT(nonanticipativity).run

```
MINOS 5.5: optimal solution found.
34 iterations, objective 500740.7143
:            Inv    Make     Sell      :=
bands 0 1    10       .        .
bands 0 2    10       .        .
bands 0 3    10       .        .
bands 1 1     0    5990     6000
```

```
bands 1 2    0   5990      6000
bands 1 3    0   5990      6000
bands 2 1    0   4428.57   4428.57
bands 2 2    0   4428.57   4428.57
bands 2 3    0   6000      6000
bands 3 1    0   1400      1400
bands 3 2    0   1400      1400
bands 3 3    0   4000      4000
bands 4 1    0   2000      2000
bands 4 2    0   2000      2000
bands 4 3    0   6500      6500
coils 0 1    0      .         .
coils 0 2    0      .         .
coils 0 3    0      .         .
coils 1 1    0   1407      1407
coils 1 2    0   1407      1407
coils 1 3    0   1407      1407
coils 2 1    0   2500      2500
coils 2 2    0   2500      2500
coils 2 3    0   1400      1400
coils 3 1    0   3500      3500
coils 3 2    0   3500      3500
coils 3 3    0   1680      1680
coils 4 1    0   4200      4200
coils 4 2    0   4200      4200
coils 4 3    0   1050      1050
;
```

File 49: steelT(nonanticipativity).out

As expected, we find that the production of bands and coils is the same for all three scenarios in the first period (5,990 tons and 1,407 tons, respectively) but will again be different for all later period (precisely, scenarios 1 and 2 yield the same amounts whereas scenario 3 yields a different solution).

4-5(d) The following run file will show the actual scenario profits for the two given scenario probability vectors:

```
model steelT(nonanticipativity).mod; data steelT(multiscenario).dat; solve;
display {s in 1..S} Profit[s];
let prob[1] := 0.0001; let prob[2] := 0.0001; let prob[3] := 0.9998; solve;
# reset data prob; data; param prob := 1 0.0001  2 0.0001  3 0.9998; solve;
display {s in 1..S} Profit[s];
```

File 50: steelT(showProfits).run

```
MINOS 5.5: optimal solution found.
34 iterations, objective 500740.7143
Profit[s] [*] :=
1  514090
2  461833
3  538793
;

MINOS 5.5: optimal solution found.
6 iterations, objective 549956.4197
Profit[s] [*] :=
1  504493
2  459644
3  549970
;
```

File 51: steelT(showProfits).out

In both cases, we see that the most profitable scenario is Scenario 3 whereas the least profitable scenario is Scenario 2. This also makes intuitive sense if we look at the actual revenue data in File 44 and remember that the production of bands is generally more profitable than the production of coils, so that a higher band revenue is more likely to increase total profit. Now consider the following table.

Profit	certainty	$p = (0.45, 0.45, 0.20)$	$p = (0.0001, 0.0001, 0.9998)$
Scenario 1	515,033	514,090	504,493
Scenario 2	462,944.29	461,833	459,644
Scenario 3	549,970	538,793	549,970
Expected Value	–	500,740.71	549,956.42

The second column lists the optimal scenario profits for the single and multi-scenario formulations in Files 42 and 46. Assuming full knowledge about future revenue prices, mathematically we will never be able to do better. The scenario profits in the third and fourth column represent those profits that we would achieve if we chose the common first-period decision suggested by the solution, and the scenario-specific decision for all remaining periods (this modeling technique is called "stochastic programming with recourse" – we make a first decision under uncertainty but can "adjust" later decisions once that uncertainty has become known). In particular, because the common first-period decision must be "acceptable" or "compromise" between all scenarios, the resulting profits are typically slightly smaller than if we were to know with certainty which scenario would occur. However, if we are very confident that one scenario is significantly more likely to occur than others (Scenario 3 in the above case), then this scenario will dominate the solution (e.g., note that the third scenario profit \$549,970 is optimal, and that the expected profit is only slightly less) with only a small risk of making less profit than possible should one of the two scenarios occur (only \$504,493 compared to \$514,090 or \$515,033 for the first scenario, and only \$459,644 compared to \$461,833 or \$462,944.29 for the second scenario).

An Alternative Stochastic Program Without Recourse An "easier" stochastic programming formulation (without recourse) assumes that all decisions must be made at the beginning of the planning period. In this case, we can avoid the duplication of decision variables and the additional nonanticipativity constraints, but we also lose the flexibility to adjust our decisions based on partially revealed uncertainty in later periods.

```
set PROD;      # products
param T > 0;   # number of weeks
param S > 0;   # number of scenarios

param prob {1..S} >= 0, <= 1;
    check: 0.99999 < sum {s in 1..S} prob[s] < 1.00001;

param rate {PROD} > 0;            # tons per hour produced
param inv0 {PROD} >= 0;           # initial inventory
param avail {1..T} >= 0;          # hours available in week
param market {PROD,1..T} >= 0;    # limit on tons sold in week

param prodcost {PROD} >= 0;       # cost per ton produced
param invcost {PROD} >= 0;        # carrying cost/ton of inventory

param revenue {PROD,1..T,1..S} >= 0;  # revenue per ton sold

var Make {PROD,1..T} >= 0;         # tons produced
var Inv {PROD,0..T} >= 0;          # tons inventoried
var Sell {p in PROD, t in 1..T} >= 0, <= market[p,t]; # tons sold

var Profit {s in 1..S }            # scenario profit (for easier tracking)
    = sum {p in PROD, t in 1..T} (revenue[p,t,s] * Sell[p,t]
      - prodcost[p] * Make[p,t] - invcost[p] * Inv[p,t]);

                # Expected revenue less costs in all weeks

maximize Expected_Profit: sum {s in 1..S} prob[s] * Profit[s];

subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p]) * Make[p,t] <= avail[t];

                # Total of hours used by all products
                # may not exceed hours available, in each week and scenario

subject to Init_Inv {p in PROD}:  Inv[p,0] = inv0[p];

                # Initial inventory must equal given value

subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1] = Sell[p,t] + Inv[p,t];

                # Tons produced and taken from inventory
                # must equal tons sold and put into inventory
```

File 52: steelT(noRecourse).mod

```
model steelT(noRecourse).mod; data steelT(multiscenario).dat; solve;
display Inv, Make, Sell;

reset data prob; data; param prob := 1 0.0001  2 0.0001  3 0.9998; solve;
# let param[1] = 0.0001; param[2] = 0.0001; let param[3] = 0.9998; solve;

display Inv, Make, Sell;
```

File 53: steelT(noRecourse).run

```
MINOS 5.5: optimal solution found.
17 iterations, objective 488493
:          Inv    Make    Sell    :=
bands 0     10      .       .
bands 1      0    5990    6000
bands 2      0    6000    6000
bands 3      0    4000    4000
bands 4      0    2000    2000
coils 0      0      .       .
coils 1   1100    1407     307
coils 2      0    1400    2500
coils 3      0    1680    1680
coils 4      0    4200    4200
;

MINOS 5.5: optimal solution found.
3 iterations, objective 549953.559
:          Inv    Make    Sell    :=
bands 0     10      .       .
bands 1      0       0      10
bands 2      0    6000    6000
bands 3      0    4000    4000
bands 4      0    6500    6500
coils 0      0      .       .
coils 1   1600    5600    4000
coils 2    500    1400    2500
coils 3      0    1680    2180
coils 4      0    1050    1050
;
```

File 54: steelT(noRecourse).out

The following table collects the results from Files 51 and 54 and shows that the expected profits with recourse are larger than the expected profit with recourse (which makes sense, because we are losing the flexibility to react), and that this difference gets smaller the better we can predict the future (which also makes sense).

Expected Profit	$p = (0.45, 0.45, 0.20)$	$p = (0.0001, 0.0001, 0.9998)$
With Recourse	500,740.71	549,956.42
No Recourse	488,493	549,953.56

AMPL Book Exercise 15-8 The original article "The Caterer Problem" by Walter Jacobs appeared in Naval Research Logistics Quarterly, Volume 1, Issue 2, Pages 154-165, June 1954 (it is posted on Canvas).

15-8(a) The main difficulty in formulating a linear programming model for this situation is to realize (and subsequently implement) that the two and four day laundering variables may reach outside the actual planning period. In the model below, this is achieved by extending the domain of the `Wash2` and `Wash4` variables and using a pair of `Start_Fresh` constraints to initialize any auxiliary variables to zero. Similarly, we extend the `Carry` variable to an additional time period 0 in order to specify the initial stock (we have seen this trick for the `Inv` variable in the multiperiod production model in File 37).

```
param T > 0;  # number of days in planning period
param demand {1..T} >= 0; # napkin requirement

param stock;  # initial stock of napkins
param price;  # purchase price for new napkins
param cost2;  # cost of fast (2day) laundering
param cost4;  # cost of slow (4day) laundering

var Buy {1..T} >= 0;       # clean napkins bought
var Carry {0..T} >= 0;     # clean napkins still on hand
var Wash2 {-2..T} >= 0;    # used napkins sent to the fast laundry
var Wash4 {-4..T} >= 0;    # used napkins sent to the slow laundry
var Throw {1..T} >= 0;     # used napkins discarded

minimize Total_Cost: sum {t in 1..T} (price*Buy[t] + cost2*Wash2[t] + cost4*Wash4[t]);

    # total cost of purchase and laundering over the full time period

subject to Initial_Stock: Carry[0] = stock;

    # The initial stock is carried over into time period 1.

subject to Start_Fresh2 {t in -2..0}: Wash2[t] = 0;
subject to Start_Fresh4 {t in -4..0}: Wash4[t] = 0;

    # The initial stock is all fresh (no current laundering).
```

```

subject to No_Lost_Napkins {t in 1..T}:

    Buy[t] + Carry[t-1] + Wash2[t-3] + Wash4[t-5]
    = Wash2[t] + Wash4[t] + Throw[t] + Carry[t];

    # The number of clean napkins acquired through purchase,
    # carryover and laundering on day t must equal the number
    # sent to laundering, discarded or carried over after day t.

subject to No_Dirty_Napkins {t in 1..T}: Wash2[t] + Wash4[t] + Throw[t] = demand[t];

    # The number of used napkins laundered or discarded after day t
    # must equal the number that were required for that day's catering.
```

File 55: caterer(linear).mod

15-8(b) For the network model, our decision variables become different sets of arcs that connect a node `Store` with a nonnegative supply (from where we buy new napkins), a node `Trash` with a nonnegative demand (to where we throw used napkins), and a set of nodes `Party[t]` to where we deliver fresh napkins (that are newly bought, cleanly carried, or completedly washed) and from where we collect both fresh napkins (that are either carried to the next `Party[t+1]` or thrown into the `Trash`) and used napkins (that are either thrown or washed). Similarly to the LP model, again we need to be a little creative to get those `Wash` arcs into their proper positions and to find a way to supply that initial stock to our first party!

```
param T > 0;  # number of days in planning period

param demand {1..T} >= 0; # napkin requirement

param stock;  # initial stock of napkins
param price;  # purchase price for new napkins
param cost2;  # cost of fast (2day) laundering
param cost4;  # cost of slow (4day) laundering

minimize Total_Cost;  # total cost of purchase and laundering

node Store: net_out >= 0;  # napkin "source" node
node Trash: net_in >= 0;   # napkin "sink" node
node Party {t in 1..T+5}: net_out = (if t = 1 then stock);  # awesome :)

arc Buy {t in 1..T} >= 0, from Store, to Party[t], obj Total_Cost price;
arc Throw {t in 1..T} >= 0, from Party[t], to Trash;
arc Carry {t in 1..T} >= 0, from Party[t], to Party[t+1];
arc Wash2 {t in 1..T} >= 0, from Party[t], to Party[t+3], obj Total_Cost cost2;
arc Wash4 {t in 1..T} >= 0, from Party[t], to Party[t+5], obj Total_Cost cost4;

subject to No_Dirty_Napkins {t in 1..T}: Wash2[t] + Wash4[t] + Throw[t] = demand[t];
```

File 56: caterer(network).mod

15-8(c) From the original paper: "The Caterer Problem is a paraphrased version of a practical military problem which arose in connection with the estimation of aircraft spare engine requirements."[...] "In the actual military application to aircraft spare engine requirements, 'laundering' corresponds to overhaul of engines removed from the aircraft because of failure, and the unit of time is a month rather than a day. As long as the time periods covered in the computation extend sufficiently beyond the period of peak operation of the engines, the use of a formulation which ignores the effect of requirements after n periods is acceptable. A more general formulation, which would apply to other economic situations involving the choice between more rapid and cheaper methods of repairing spare parts, would have to consider starting and ending inventories and other complications as well. Some consideration has been given to the more general problem, but it is not the concern of the present paper."

15-8(d) Most of my data tests produced somewhat trivial results, but this one is quite interesting. In particular, and maybe surprisingly at first, this solution frequently uses the 2-day laundering service. Can you explain why this happens? Also try to change some of the data, and see how the solution will change.

```
param T = 14;

param demand default 100;

param stock := 200;
param price := 1.2;
param cost2 := 0.3;
param cost4 := 0.1;
```

File 57: caterer.dat

```
model caterer(linear).mod; data caterer.dat; solve;
# option display_width 200; display {t in 1..T}
# (Carry[t-1], Buy[t], demand[t], Carry[t], Wash2[t], Wash4[t], Throw[t]);
display demand, Buy, Carry, Wash2, Wash4, Throw; reset;
model caterer(network).mod; data caterer.dat; solve;
display demand, Buy, Carry, Wash2, Wash4, Throw;
```

File 58: caterer.run

```
MINOS 5.5: optimal solution found.
35 iterations, objective 440
:   demand   Buy    Carry           Wash2          Wash4         Throw    :=
-4    .       .       .               .            0               .
-3    .       .       .               .            0               .
-2    .       .       .             0              0               .
-1    .       .       .             0              0               .
0     .       .     200             0              0               .
1    100      0     100             0            100               0
2    100      0       0           100              1.64931e-14     0
3    100    100       0             1.64931e-14  100               0
4    100    100       0           100              0               0
5    100      0       0             0            100               0
6    100      0       0           100              0               0
7    100      0       0             8.69126e-15  100               0
8    100      0       0           100              7.8018e-15      0
9    100      0      -1.64931e-14   1.64931e-14  100               0
10   100      0       0           100              0               0
11   100      0      -1.56036e-14   0              0             100
12   100      0       0             0              0             100
13   100      0       0             0              0             100
14   100      0       0             0              0             100
;

MINOS 5.5: optimal solution found.
31 iterations, objective 440
:   demand   Buy    Carry           Wash2          Wash4         Throw    :=
1    100      0     100            -1.55399e-14  100               0
2    100      0       0           100              0               0
3    100    100       0            -8.23962e-15  100               0
4    100    100       0           100             -8.11117e-24     0
5    100      0       0            -7.30025e-15  100               0
6    100      0       0           100              8.23962e-15     0
7    100      0       0            -8.23962e-15  100               0
8    100      0       0           100              2.47189e-14     0
9    100      0       0             0            100               0
10   100      0       0           100              0               0
11   100      0       0            -7.30025e-15    0             100
12   100      0       0             0              0             100
13   100      0       1.27217e-14   0              0             100
14   100      0       0             0              0             100
;
```

File 59: caterer.out

AMPL Book Exercise 20-4 Knapsack problems are always fun, and this (easy) exercise was no exception!

20-4(a) Model and data are written quickly, but remember to use an integer solver (`lpsolve`, `gurobi`, `cplex`)!

```
set OBJECTS;

param value {OBJECTS} > 0;
param weight {OBJECTS} > 0;

param weight_limit >= 0;

var X {OBJECTS} binary;

maximize Total_Value: sum {x in OBJECTS} value[x] * X[x];

subject to Weight_Limit: sum {x in OBJECTS} weight[x] * X[x] <= weight_limit;
```

File 60: knapsack(a).mod

```
param: OBJECTS: value weight :=
          a      1000    55
          b       800    50
          c       750    40
          d       700    35
          e       600    30
          f       550    30
```

```
            g       250     15
            h       200     15
            i       200     10
            j       150      5;

param weight_limit := 100;
```

File 61: knapsack(a).dat

```
model knapsack(a).mod; data knapsack(a).dat;
option omit_zero_rows 1; solve; display X;
option solver lpsolve; solve; display X;
option solver gurobi; solve; display X;
option solver cplex; solve; display X;
```

File 62: knapsack(a).run

```
MINOS 5.5: ignoring integrality of 10 variables
MINOS 5.5: optimal solution found.
5 iterations, objective 2025
X [*] :=
c  0.5
d  1
e  1
i  1
j  1
;

LP_SOLVE 4.0.1.0: optimal, objective 2000
90 simplex iterations
33 branch & bound nodes: depth 9
X [*] :=
d  1
e  1
f  1
j  1
;

Gurobi 5.5.0: optimal solution; objective 2000
X [*] :=
d  1
e  1
f  1
j  1
;

CPLEX 12.5.1.0: optimal integer solution; objective 2000
1 MIP simplex iterations
0 branch-and-bound nodes
X [*] :=
d  1
e  1
f  1
j  1
;
```

File 63: knapsack(a).out

Note that `MINOS` does not respect the integrality constraints and returns a fractional solution that packs items $\{d, e, i, j\}$ and half of c at a total value of 2025, whereas `LP_Solve`, `Gurobi`, and `CPLEX` return the same (optimal) integer solution that packs items $\{d, e, f, j\}$ at the smaller value of 2000.

20-4(b) By now, you should be an AMPL expert and the following modifications will cause you no difficulties!

```
param knapsacks > 0 integer; # number of knapsacks

param weight_limit {1..knapsacks} >= 0;

var X {OBJECTS,1..knapsacks} binary;

maximize Total_Value: sum {x in OBJECTS, k in 1..knapsacks} value[x] * X[x,k];

subject to Weight_Limit {k in 1..knapsacks}:

    sum {x in OBJECTS} weight[x] * X[x,k] <= weight_limit[k];

subject to No_Double_Packing {x in OBJECTS}: sum {k in 1..knapsacks} X[x,k] <= 1;
```

File 64: knapsack(b).mod

```
param knapsacks = 2;

param weight_limit := 1 50  2 50;
```

File 65: knapsack(b).dat

```
model knapsack(b).mod; data knapsack(b).dat;
option omit_zero_rows 1; solve; display X;
option solver gurobi; solve; display X;
```

File 66: knapsack(b).run

```
MINOS 5.5: ignoring integrality of 18 variables
MINOS 5.5: optimal solution found.
9 iterations, objective 2025
X :=
c 1   0.125
c 2   0.375
d 2   1
e 1   1
e 2   4.64989e-17
i 1   1
j 1   1
;

Gurobi 5.5.0: optimal solution; objective 1950
13 simplex iterations
X :=
c 1   1
e 2   1
g 2   1
i 1   1
j 2   1
;
```

File 67: knapsack(b).out

Note that the fractional solution simply splits up the former solution and packs items $\{e, i, j\}$ and one eight of item c into the first knapsack and item d and another three eights of item c into the second knapsack (with the same value as before), whereas the (optimal) integer solution packs items $\{c, i\}$ into the first knapsack and items $\{e, g, j\}$ into the second knapsack at a (further reduced) value of 1,950.

20-4(c) In comparison to the standard $n \times n$ assignment problem, a first difference is the number of possible combinations: whereas there are $n!$ possible assignments (the same as the permutations in the traveling salesman problem), there are (in principle) 2^n possible packings. This, however, seems to favor the knapsack problem because $n! \gg 2^n$ for large n, and would also not explain why the TSP is so much harder than the assignment problem. To truly understand the additional level of difficulty for the TSP and the knapsack problem, it is important to realize that all $n!$ assignments are actually feasible (unless there are additional side constraints) which allows for a simple algebraic characterization of the underlying feasible polyhedral set (more precisely, of the convex hull of its integer points). It is the additional constraints for the TSP, and the single constraint for the knapsack problem that destroy this simplicity and make an algebraic characterization significantly harder. We will revisit these ideas once we learn more about the geometry of LP, and its implications for the solution of integer programs.

20-4(d) The problem did not specify a volume limit, but it must be less than 7 for both solutions to change.

```
set OBJECTS;

param value {OBJECTS} > 0;
param weight {OBJECTS} > 0;
param volume {OBJECTS} > 0;

param weight_limit >= 0;
param volume_limit >= 0;

var X {OBJECTS} binary;

maximize Total_Value: sum {x in OBJECTS} value[x] * X[x];

subject to Weight_Limit: sum {x in OBJECTS} weight[x] * X[x] <= weight_limit;
subject to Volume_Limit: sum {x in OBJECTS} volume[x] * X[x] <= volume_limit;
```

File 68: knapsack(d).mod

```
param: OBJECTS:   value   weight  volume :=
          a        1000      55      3
          b         800      50      3
          c         750      40      3
          d         700      35      2
          e         600      30      2
          f         550      30      2
          g         250      15      2
          h         200      15      1
          i         200      10      1
          j         150       5      1;

param weight_limit := 100;
param volume_limit := 6.9;
```

File 69: knapsack(d).dat

```
model knapsack(d).mod; data knapsack(d).dat;
option omit_zero_rows 1; solve; display X;
option solver gurobi; solve; display X;
```

File 70: knapsack(d).run

```
MINOS 5.5: ignoring integrality of 10 variables
MINOS 5.5: optimal solution found.
6 iterations, objective 2006
X [*] :=
a  0.44
d  1
e  1
i  0.58
j  1
;

Gurobi 5.5.0: optimal solution; objective 1900
3 simplex iterations
X [*] :=
a  1
d  1
i  1
;
```

File 71: knapsack(d).out

20-4(e) If we consider the audiences to represent the values of those items that we choose and whose sum we like to maximize, then the advertising problem in Exercise 20-1 corresponds to a knapsack problem with four knapsack constraints, one each for cost and the creative time of writers, artists, and others.

20-4(f) This new restriction can be implemented by the following modification to our initial model in File 60:

```
var X {OBJECTS} in {0,1,2,3};
```

File 72: knapsack(f).mod

```
model knapsack(f).mod; data knapsack(a).dat;
option omit_zero_rows 1; solve; display X;
option solver gurobi; solve; display X;
```

File 73: knapsack(f).run

```
MINOS 5.5: ignoring integrality of 10 variables
MINOS 5.5: optimal solution found.
2 iterations, objective 2150
X [*] :=
e  2.83333
j  3
;

Gurobi 5.5.0: optimal solution; objective 2150
1 simplex iterations
X [*] :=
d  1
e  1
i  2
j  3
;
```

File 74: knapsack(f).out

Interestingly, we find both a fractional and an alternative integer solution with the same total value.