# Business Decision Analytics under Uncertainty

### Rutgers Business School, Undergraduate New Brunswick
### Solving a Simple Inventory Problem with Python

We now return the inventory and production planning problem that we solved using a spreadsheet, which had the following characteristics

- We must plan for 4 months
- We start with zero inventory
- We can carry up to 4 units of inventory from month to month, at a cost of $0.50 per unit per month
- We can produce up to 5 units per month
- There is a setup charge of $3 in each month we choose to produce
- The variable cost per unit produced is $1.
- The demand for the coming 4 months is known with certainty to be 1 unit, 3 units, 2 units, and 4 units, respectively.

Our goal is to minimize the cost of meeting the demand. From the class website, we download the following program prologue:

```
import numpy

hugeNumber = float("inf")

stages             = 4
startInventory     = 0
inventoryCapacity  = 4
productionCapacity = 5

setupCost    = 3.0
variableCost = 1.0
holdingCost  = 0.5

demand = numpy.array([-1000,1,3,2,4])
```

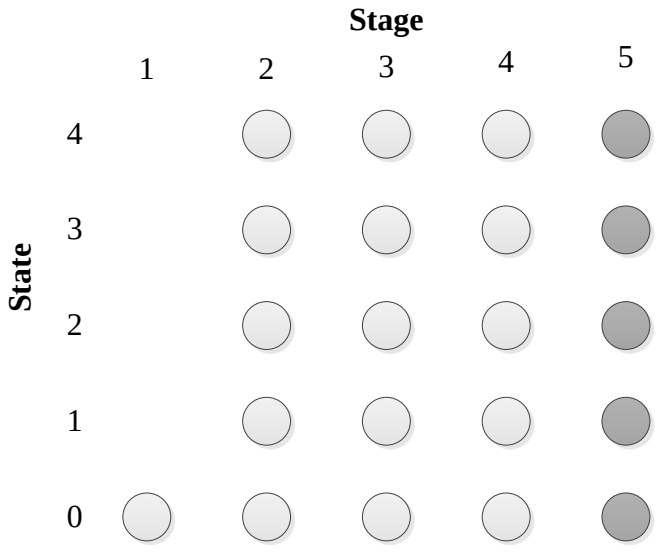
The line `import numpy` makes the NumPy package available, because we will be using NumPy arrays to store our input data and calculations. The line `hugeNumber = float("inf")` sets `hugeNumber` to be $+\infty$. We need this value because we will be using the technique discussed in the previous notes, in which we remember the smallest of a sequence of calculations without having to store them all. The remaining lines of code just set up some convenient variables to hold the problem data. It is much better to put all the data "up front", rather than scattering it throughout the code, because it will then be much easier to modify the program to solve other similar problems. In a real "production" code, we would read the input data from a file or database instead. The variable `stages` is the number of months, because the number of months is the number of stages we have in our dynamic programming solution procedure.

The line that sets up the demand information is worthy of some additional discussion. We have

```
demand = numpy.array([-1000,1,3,2,4])
```

First, we use the technique of creating a Python list, and then turning it into an array, as discussed in the previous set of notes. Second, note that we start the array with one piece of "garbage" data "`-1000`" before the real data "`1,3,2,4`". The reason is that Python arrays are indexed starting at 0, but we would like to number the stages (months) starting with 1, the same way we do by hand and in Excel. The simplest way to do this is to just dimension the `demand` array to have 5 elements (numbered 0 through 4), and then not use the first one. So the useful data are in elements 1 through 4, and the first element contains "garbage" that we will not touch (unless our program has a bug).

Dynamic programming calculations involve computing $f_t(i)$, which is the value of being in state $i$ at stage $t$. Here, the stages are the months and the states are the amount of inventory, so $f_t(i)$ is the value of having $i$ units of inventory at the start of month $t$. In our Python program, we will store $f_t(i)$ in `f[t,i]`, where `f` is a two-dimensional NumPy array. The values of $t$ used in our spreadsheet calculation were 1 through 4, and the values of $i$ were 0 through 4. However, in our Python solution, we will introduce an additional "month", numbered 5, that is, `stages + 1`. This technique will allow us to simplify our code. The number we place in the array element `f[stages + 1, i]` is the value of having `i` units of inventory left after all the production decisions have been made and customer demands filled. If the problem mentioned some kind of salvage value for having leftover inventory, we could place it in `f[stages + 1, i]`. The problem doesn't mention anything like that, so we can just let `f[stages + 1, i]` contain zero for all values of `i`. Graphically, we have the following situation:




The last (darker) column of states are the values of "landing" with a certain amount of inventory once the entire decision process is done. The values of these states are all zero (although we will

change this is some future problems). The states form the "base" of our recursive computational process. Once their values are set, we can compute the values of all the other states by effectively using the recursion formula we covered in class.

We set up the `f` array as follows:

```
f = numpy.zeros([stages + 2, inventoryCapacity + 1])
```

This statement sets up a two-dimensional array currently containing all zeroes. The first dimension specifier says that the first index of the array `f` takes `stages + 2` different values, running from 0 to `stages + 2 - 1`, that is, `stages + 1`, which is 5. Thus, we can use any value from 0 to 5 for the first index. However, we won't bother using element 0 because we won't have a stage 0. The second dimension specifier says that the second index can take `inventoryCapacity + 1` different values, running from 0 to `inventoryCapacity`. In our case, `inventoryCapacity` is 4, so the second index will run from 0 to 4. We will use all these values, because it is possible to have zero inventory.

We also need an array to remember the optimal decisions, which we marked by "*" characters when solving problems by hand. By convention, we will call this array `x` in all our programs. In each case, `x[t,i]` will store the best decision to make if you are in state *i* at stage *t* of the decision process. We set up this array as follows:

```
x = numpy.zeros([stages + 1, inventoryCapacity + 1], dtype=int)
```

This is almost the same as what we did for `f`, with two differences. First, we don't need to store any decisions for the extra "fake" stage we added at the end, because that stages occurs after all decisions have been made. Therefore, the first index only needs to run up to 4, so we can dimension it to have just `stages + 1` (that is, 5) elements, recalling that we don't use stage 0. Second, we know that we will be storing only integers in this array, because the decision is an amount of production, which can only be integer. That is why we add the optional keyword argument `dtype=int`.

At this point, in general, we would now initialize the values of `f[stages + 1, i]` to contain the "terminal" or "salvage" value of having `i` units in inventory after the entire decision process is over. For this problem, that is just zero for all `i`, and we already initialized the entire `f` array to contain zeroes, the terminal values are already set up correctly.

Now we can start our main calculation. As always, we must loop backwards over stages. Here we are starting at stage 4 (that is, `stages`) and counting back to 1. This is accomplished by the loop control statement

```
for t in range(stages,0,-1) :
```

This instructs Python to count `t` backwards from `stages` to 1, in the manner we discussed in the previous notes.

Within each stage, we must compute the value of each state. This we do by a nested loop, as follows

```
for i in range(inventoryCapacity + 1) :
```

This sets `i` to run from 0 to `inventoryCapacity`. Generally, it does not matter which order you process states within a stage, so we simply process states in increasing order since that order is the to code.

For each particular state and stage, the task, is to determine the lowest-cost decision, taking into account its repercussions in all subsequent stages. The first part of this is to determine which decisions are possible. In this problem, the decision is the amount to produce. This can be between zero and `productionCapacity` (set to 5), but there are also limits imposed by the inventory capacity and having to meet demand. Specifically, if we have `i` units of inventory and demand for this period is `demand[t]`, we must produce at least `demand[t] - i` units in order to meet demand. Thus the minimum amount we must produce is

```
minProduction = max(0, demand[t] - i)
```

This says that the smallest amount of production me may consider is the larger of 0 and `demand[t] - i`, because we cannot produce less than 0.

Next, we set

```
maxProduction = min(productionCapacity,
                    inventoryCapacity - i + demand[t])
```

The reason for this is that if we produce more than `inventoryCapacity - i + demand[t]` units, we will exceed the inventory capacity at the next stage – basically, if we have `inventoryCapacity - i` room to fill up the inventory, but `demand[t]` units will "go away" because of demand. But on the other hand, we cannot produce more than `productionCapacity` units, so our upper limit is the lower of these two.

We are now ready to find the lowest-cost amount of production. The production amounts we need to loop over are `minProduction` through `maxProduction`, inclusive, so we use the range specifier `range(minProduction, maxProduction+1)`, which is equivalent to `range(minProduction, maxProduction+1, 1)`. Remember that a Python range never includes its ending point, so to get `maxProduction` included in the calculation, we must specify an endpoint of `maxProduction+1`.

We will use the standard pattern of remembering the smallest of a sequence of calculations, as covered in the previous notes. We thus have the general pattern

```
value = hugeNumber

for p in range(minProduction, maxProduction+1) :

    < Calculate moveValue to be the value of producing p >

    if moveValue < value :
        value = moveValue
        bestMove = p
```

Here, `p` loops over the possible production levels. When this loop is done, `value` will contain the smallest value of `moveValue` encountered, and `bestMove` will contain the value of `p` that led to it. As mentioned in the previous notes, initializing `value` to infinity (already stored in `hugeNumber`) ensures that the first time through the loop will always stored as the best value `value` and decision encountered so far; subsequent loop executions will only update value and `bestMove` if they encounter a lower cost decision. Note that in the event of "ties", only the first of the optimal moves will be stored. It is possible to store all the optimal moves at the cost of adding some complexity to the program, but we will not bother with such variations at the moment.

It remains to calculate `moveValue` to be the value of the producing `p` units if one is in state `i` at stage `t`, which is the as-yet-unstated code contained between angle brackets above. This we do as follows:

```
j = i + p - demand[t]
productionCost = variableCost*p
if p > 0 :
    productionCost += setupCost

moveValue = holdingCost*j + productionCost + f[t+1,j]
```

We will generally use the convention that `j` will be next state resulting from our current decision. In this case, the next state is `i + p - demand[t]`, because we add `p` units of inventory to the current inventory `i`, but `demand[t]` units removed for delivery to customers. This ending inventory for the current month becomes the starting inventory for the next month. The three lines following the calculation of `j` compute the production cost for the current period, adding in the setup cost if we produce more than zero units. Finally, the following line of code "puts everything together":

```
moveValue = holdingCost*j + productionCost + f[t+1,j]
```

The total cost of producing `p` units if one has `i` units of inventory at the start of month `t` is the holding cost of the inventory left over (`holdingCost*j`) plus the current production cost (already calculated in `productionCost`), plus the "cost to go", the lowest cost one can obtain from the next period to the end of the time horizon if we start with `j` units of inventory. Since we are looping backwards over `t`, this cost to go is already stored in `f[t+1,j]`. The first time through the loop, when `t` is 4, this cost to go is `f[5,j]`, which was initialized to 0 and thus

effectively disappears from the calculation. It is much easier to set things up this way than to have special code for the first pass through the loop.

One we are out of the p loop, we simply record the best value and decision encountered in the `f` and `x` arrays:

```
f[t,i] = value
x[t,i] = bestMove
```

Once all the loops are done, the value of the optimal solution is stored in `f[1,0]`, which is the value of being in stage 1 with 0 inventory – if we had a different amount of starting inventory, we would simply change the "0" here. Note that we have actually computed the optimal solution for all amounts of starting inventory, which we would not bother to do by hand. But the computer does the computations so quickly there is no good reason to bother avoiding them. We print the optimal cost by writing:

```
print("Optimal cost is " + str(f[1,startInventory]))
```

To print the optimal decision, we try have to do a "forward pass" in which we effectively do the same thing as when we "followed the stars" in our manual solution. This we do as follows:

```
solutionString = "Production amounts:"
i = startInventory
for t in range(1,stages+1) :
    solutionString += " " + str(x[t,i])
    i = i + x[t,i] - demand[t]

print(solutionString)
```

Here, we set `i` to the starting inventory and then enter a loop in which `t` runs from 1 up through `stages`. In each stage, we take the string representation of the decision made in the current state `i`, which is stored in `x[t,i]`, and concatenate it to the string we will eventually print. We then update `i` to be the state in the following stage by calculating `i = i + x[t,i] - demand[t]`, that is, current inventory plus production minus delivered demand. The following statement would do the same thing:

```
i += x[t,i] - demand[t]
```

We then pass to the next state. After this loop is complete, `solutionString` contains a description of the optimal solution, which we simply print.

We obtain the following output when we run the program (the first time you import the NumPy package, which is pretty large, you may notice a short delay):

```
Optimal cost is 20.0
Production amounts: 1 5 0 4
```

The nice thing about how we've organized this program is that to solve larger instances of the same class of problems, we only need to change some of the data setup at the beginning.

For example, suppose we change the data setup section near the start of the program to

```
stages              = 10
startInventory      = 0
inventoryCapacity   = 20
productionCapacity = 15

setupCost    = 3.0
variableCost = 1.0
holdingCost  = 0.5

demand = numpy.array([-1000,1,3,2,4,6,10,4,2,8,9])
```

Now we are solving the same kind of problem, but with 10 months, and inventory capacity of 20, and a production capacity of 15. This problem instance would be extremely painful on a spreadsheet and absolute torture by hand. But we nearly instantaneously obtain

```
Optimal cost is 74.5
Production amounts: 4 0 6 0 6 10 6 0 8 9
```

You could also solve this problem using an Excel solver model with integer variables. But in this case dynamic programming is probably faster.