J Book

Jason Corderoy

1 Item Associations

1. d = data. Rows: baskets, columns: products, where 1=product in basket
2. pb = % baskets containing each product
3. ep = Expected % baskets containing each pair of products
4. ap = actual % baskets containing each pair of products
5. Calculate lift: ap/ep

```
   nn=:4
   ]< d=:(,~nn)$ ?2#~ *~nn
┌───────┐
│1 1 0 1│
│0 1 1 0│
│0 1 1 1│
│0 1 1 0│
└───────┘
   ]pb=:(+/ % #)"2 d
0.25 1 0.75 0.5
   ]<ep=:(pb * =/~ i.nn) >. pb *"0 1 pb
┌──────────────────────┐
│  0.25 0.25 0.1875 0.125│
│  0.25    1   0.75   0.5│
│0.1875 0.75   0.75 0.375│
│ 0.125  0.5  0.375   0.5│
└──────────────────────┘
   ]<ap=:>{{(+/ % #) */"1 y {"1 _1 d}} each { ;~ i.nn
┌────────────────┐
│0.25 0.25    0 0.25│
│0.25    1 0.75  0.5│
│   0 0.75 0.75 0.25│
│0.25  0.5 0.25  0.5│
└────────────────┘
   ]<lift=:ap%ep
┌──────────────────┐
│1 1        0        2│
│1 1        1        1│
│0 1        1 0.666667│
│2 1 0.666667        1│
└──────────────────┘
```

2 Demand Fill Optimisation

1. xo = Options
2. xs = Random selection from options (xo)
3. xp = Problem to solve, which is the column sum of xs
4. Solve it. Solve knowing only xp and xo. Being blind to xs

Think of each column as a product and each row as an option for how much of each product. Thinking possible pallet configurations or possible cattle carcasse breakdowns makes these options more understandable.

```
   nn=:4
   ]<xo=:8* (] % +/"1) (,~nn) $ ?2#~*~nn
┌─────────────────────────┐
│2.66667 2.66667 0 2.66667│
│      4       4 0       0│
│      2       2 2       2│
│      0       0 4       4│
└─────────────────────────┘
   ]<xs=:xo {~ ?3#nn
┌───────┐
│2 2 2 2│
│2 2 2 2│
│2 2 2 2│
└───────┘
   ]xp=:+/"2 xs
6 6 6 6
   xt=:(xo,0) {~ ?20#nn NB. rando solve incl all 0 option
   eval=:3 : '+/ | xp - +/"2 y'
   bs=:3 : '(}:xt) ,~ (xo,0){~ (] i. <./) {{eval y, }: xt}}"1 xo, 0'
NB. best solve
   solver=: 3 : 0
xt=:bs 1
eval xt
)
   solver"0 i.25
128 120 112 104 96 88 80 72 64 56 48 40 32 24 16 13.3333 8 4 4 0 0 0 0
0 0
   ]<xt=:xt {~ I. 0< +/"1 xt
┌───────┐
│2 2 2 2│
│0 0 4 4│
│4 4 0 0│
└───────┘
```

3 Profit Optimisation

This is very similar to the demand fill optimisation example. Except with profit we want to maximise plus explicit weights are now needed. For example a product may have a gross profit of $4 per unit when there is demand but may need to sell at a marked down price thereafter and may instead make a loss of $1 per unit from that point on. It is weights like these that we are interested in when considering profit optimisation models.

Below we use a simple example. That us humans can quickly and easily solve. The approach below highlights the importants of this algorithms ability to loop through previous solves to give it a chance to find a better solution to the problem after the first passthrough.

```
   ]<xo=:(3 3) $ 2 0 0 1 1 0 1 0 1
┌─────┐
│2 0 0│
│1 1 0│
│1 0 1│
└─────┘
   ]xp=:4 2 2
4 2 2
   ]<xt=:(4,3) $ 0 NB. start with all zeros
┌─────┐
│0 0 0│
│0 0 0│
│0 0 0│
│0 0 0│
└─────┘
   ]<wdf=:10 1 1 NB. demand fill weights
┌─────┐
│10 1 1│
└─────┘
   ]<wde=:_10 _0.5 _0.5 NB. demand exceeded weights
┌──────────────┐
│_10 _0.5 _0.5│
└──────────────┘
   eval=:3 : 0
csy=.+/"2 y
df=. xp <. csy
de=.0 >. csy-df
+/ (de*wde), (df*wdf)
)
   NB. eval xt
   bs=:3 : '(}:xt) ,~ xo{~ (] i. >./) {{eval y, }: xt}}"1 xo' NB. best
solve finder
   solver=: 3 : 0
xt=:bs 1
eval xt
)
   NB. first try:
   solver"0 i.4
20 40 31 22
```

```
   <xt
┌─────┐
│1 1 0│
│1 1 0│
│2 0 0│
│2 0 0│
└─────┘
   xp - +/"2 xt
_2 0 2
   NB. second try:
   solver"0 i.4
33 44 44 44
   <xt
┌─────┐
│1 1 0│
│1 1 0│
│1 0 1│
│1 0 1│
└─────┘
   xp - +/"2 xt
0 0 0
```

4 The aaaabbbcca problem

I read a post recently by a software developer that ‘most candidates cannot solve this interview problem’:
Input: “aaaabbbcca”
Output: [(“a”,4),(“b”,3),(“c”,2),(“a”,1)]

J solution:

```
   >({. ; #) each {{y <;.1~ 1, ~:/"1] 2 ]\ a. i. y}} 'aaaabbbcca'
┌─┬─┐
│a│4│
├─┼─┤
│b│3│
├─┼─┤
│c│2│
├─┼─┤
│a│1│
└─┴─┘
```

5 Two Sum

From leetcode: https://leetcode.com/problems/two-sum/

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:
Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:
Input: nums = [3,3], target = 6
Output: [0,1]

Such a simple problem to solve using loops but I was interested in how to solve this problem in a loopless manner.

```
   twosum=: 4 : '{. 4 $. $. x = (= i. # y) + +/~ y'
   9 twosum 2 7 11 15
0 1
   6 twosum 3 2 4
1 2
   6 twosum 3 3
0 1
```