Problem Three

Implement the following algorithm. You can use any programming language, any resources.

1 Algorithm

Enclosed is a walkthrough for the Lefty algorithm which computes the number of nxn 0-1 matrices with t ones in each row and column, but none on the main diagonal.

The algorithm used to verify the equations presented counts all the possible matrices, but does not construct them.

It is called "Lefty", it is reasonably simple, and is best described with an example.

Suppose we wanted to compute the number of 6x6 0-1 matrices with 2 ones in each row and column, but no ones on the main diagonal. We first create a state vector of length 6, filled with 2s:

#(2 2 2 2 2 2)

This state vector symbolizes the number of ones we must yet place in each column. We accompany it with an integer which we call the "puck", which is initialized to 1. This puck will increase by one each time we perform a ones placement in a row of the matrix (a "round"), and we will think of the puck as "covering up" the column that we won't be able to place ones in for that round.

Since we are starting with the first row (and hence the first round), we place two ones in any column, but since the puck is 1, we cannot place ones in the first column. This corresponds to the forced zero that we must place in the first column, since the 1,1 entry is part of the matrix's main diagonal.

The algorithm will iterate over all possible choices, but to show each round, we shall make a choice, say the 2nd and 6th columns. We then drop the state vector by subtracting 1 from the 2nd and 6th values, and advance the puck:

#(2 1 2 2 2 1); 2

For the second round, the puck is 2, so we cannot place a one in that column. We choose to place ones in the 4th and 6th columns instead and advance the puck:

#(2 1 2 1 2 0); 3

Now at this point, we can place two ones anywhere but the 3rd and 6th columns. At this stage the algorithm treats the possibilities differently: We can place some ones before the puck (in the column indexes less than the puck value), and/or some ones after the puck (in the column indexes greater than the puck value). Before the puck, we can place a one where there is a 1, or where there is a 2; after the puck, we can place a one in the 4th or 5th columns. Suppose we place ones in the 4th and 5th columns. We drop the state vector and advance the puck once more:

#(2 1 2 0 1 0); 4

For the 4th round, we once again notice we can place some ones before the puck, and/or some ones after.

Before the puck, we can place:

(a) two ones in columns of value 2 (1 choice)

(b) one one in the column of value 2 (2 choices)

(c) one one in the column of value 1 (1 choice)

(d) one one in a column of value 2 and one one in a column of value 1 (2 choices).

After we choose one of the options (a)-(d), we must multiply the listed number of choices by one for each way to place any remaining ones to the right of the puck.

So, for option (a), there is only one way to place the ones.

For option (b), there are two possible ways for each possible placement of the remaining one to the right of the puck. Since there is only one nonzero value remaining to the right of the puck, there are two ways total.

For option (c), there is one possible way for each possible placement of the remaining one to the right of the puck. Again, since there is only one nonzero value remaining, there is one way total.

For option (d), there are two possible ways to place the ones.

We choose option (a). We drop the state vector and advance the puck:

#(1 1 1 0 1 0); 5

Since the puck is "covering" the 1 in the 5th column, we can only place ones before the puck. There are (3 take 2) ways to place two ones in the three columns of value 1, so we multiply 3 by the number of ways to get remaining possibilities. After choosing the 1st and 3rd columns (though it doesn't matter since we're left of the puck; any two of the three will do), we drop the state vector and advance the puck one final time:

#(0 1 0 0 1 0); 6

There is only one way to place the ones in this situation, so we terminate with a count of 1. But we must take into account all the multiplications along the way: 1*1*1*1*3*1 = 3. So, this string of rounds counts the following three matrices:

```
0 1 0 0 0 1    0 1 0 0 0 1    0 1 0 0 0 1
0 0 0 1 0 1    0 0 0 1 0 1    0 0 0 1 0 1
0 0 0 1 1 0    0 0 0 1 1 0    0 0 0 1 1 0
1 0 1 0 0 0    1 0 1 0 0 0    1 0 1 0 0 0
1 1 0 0 0 0    1 0 1 0 0 0    0 1 1 0 0 0 <– only variation
0 0 1 0 1 0    0 1 0 0 1 0    1 0 0 0 1 0
```

Another way of thinking of the varying row is to start with the first matrix, focus on the lower-left 2x3 submatrix, and note how many ways there were to permute the columns of that submatrix. Since there are only 3 such ways, we get 3 matrices.

We cannot optimize by permuting submatrices that contain an entry of the main diagonal, since that is a 'fixed' position that must contain a zero.

We note that, in the actual implementation, after each round, the state vector values to the left of the puck are sorted (but the values to the right of the puck maintain their exact positions) to make counting possibilities easier. Hence, we would have in the third and fourth rounds, respectively,

#(1 2 2 1 2 0); 3

#(1 2 2 0 1 0); 4

In a larger example (13x13 matrix with 3 ones in each row/column), we might come across the following state:

#(0 1 1 1 2 2 3 3 0 1 0 0 1); 9

To place three ones in this case, the algorithm would branch depending on how many ones it wishes to place to the right of the puck, make that choice, and then multiply by the possibilities for placing the remaining ones to the left of the puck. Hence,

Case 1: Right of the puck gets 3 ones.

Not possible since there are only two nonzero columns there.

Case 2: Right of the puck gets 2 ones.

Only one way to do this, but there are three different ways to place the third one to the left of the puck:

(a) under a column with a 1 value (3 ways), with resultant state #(0 0 1 1 2 2 3 3 0 0 0 0 0); 10

(b) under a column with a 2 value (2 ways), with resultant state #(0 1 1 1 1 2 3 3 0 0 0 0 0); 10

(c) under a column with a 3 value (2 ways), with resultant state #(0 1 1 1 2 2 2 3 0 0 0 0 0); 10.

Case 3: Right of the puck gets 1 one.

There are two ways to do this, so we have to branch depending on if it's going in the 10th column or 13th column.

Subcase 1: 10th column.

To place the other two ones to the left of the puck, we have choices:

(d) both ones under a 1-value ((3 take 2) ways),
with resultant state #(0 0 0 1 2 2 3 3 0 0 0 0 1); 10

(e) one one under 1-value, one under 2-value ((3 take 1)*(2 take 1) ways),
with resultant state #(0 0 1 1 1 2 3 3 0 0 0 0 1); 10

(f) one one under 1-value, one under 3-value ((3 take 1)*(2 take 1) ways),
with resultant state #(0 0 1 1 2 2 2 3 0 0 0 0 1); 10

(g) both ones under 2-value ((2 take 2) ways),
with resultant state #(0 1 1 1 1 1 3 3 0 0 0 0 1); 10

(h) one one under 2-value, one under 3-value ((2 take 1)*(2 take 1) ways),
with resultant state #(0 1 1 1 1 2 2 3 0 0 0 0 1); 10

(i) both ones under 3-value ((2 take 2) ways),
with resultant state #(0 1 1 1 2 2 2 2 0 0 0 0 1); 10.

Subcase 2: 13th column.

The options (j)-(o) are the same as (d)-(i) in the above subcase, but the resultant states have #(... 0 1 0 0 0) at the end instead.

Case 4: Right of the puck gets 0 ones.

So all three ones go to the left of the puck. We have choices:

(p) all ones under 1-value ((3 take 3) ways),
with resultant state #(0 0 0 0 2 2 3 3 0 1 0 0 1); 10

(q) two ones under 1-value, one under 2-value ((3 take 2)*(2 take 1) ways), with resultant state #(0 0 0 1 1 2 3 3 0 1 0 0 1); 10
(r) two ones under 1-value, one under 3-value ((3 take 2)*(2 take 1) ways), with resultant state #(0 0 0 1 2 2 2 3 0 1 0 0 1); 10
(s) two ones under 2-value, one under 3-value ((2 take 2)*(2 take 1) ways), with resultant state #(0 1 1 1 1 1 2 3 0 1 0 0 1); 10
(t) one one under 2-value, two under 3-value ((2 take 1)*(2 take 2) ways), with resultant state #(0 1 1 1 1 2 2 2 0 1 0 0 1); 10

In all options (a)-(t), the state would be resorted: since the puck moved from the 9th column to the 10th column, it will reveal a 0 in the 9th column, which will then get moved to the front of the state vector.

In general, Lefty will iterate over all possible choices (optimizing for permutations below the main diagonal by multiplying by the indicated cofactors), add up the values, and produce the result. To provide a further speedup, a storage object is used to store each state vector for which a count has been acquired, so that if that state vector is seen again, the count can be produced from memory instead of recalculated. This speedup is necessary, and without it the algorithm will take too long.