A Solution to the Nrich Factors and Multiples Problem

By Alexandre Otto and Isaac Yip

In its simplest form, the method of solution to the problem as provided is simple. Find an arrangement of the heading-cards where no two heading-cards that contradict each other affect the same square, and then find the numbers that fit within.

Before beginning to explore our method of solving this problem, however, a few quick definitions of terminology we came to use are necessary:

- “Grid”: the playing-board. In the code attached it is referred erroneously to as a “solution”,
- “Tag”: a heading-card,
- “Vertical tags”, or similar: the five tags which occupy the left-hand-side spaces,
- “Horizontal tags” or similar: the five tags which occupy the lower five spaces
- “Group”: two tags which are incompatible with(i.e. contradict) each other

The first step, thus, after realising the most basic point of the problem, was to figure out which tags were incompatible with each other.

The first Group is obvious: a number cannot be at the same time odd or even.
The second is similarly obvious: a number cannot be at the same time greater than twenty or lesser than twenty.
The third is, again, obvious - though it is less so: Square numbers, being a product of multiplication, cannot be prime(one is not prime).

Thus, we have our groups.
`Groups = [[ODD, EVEN], [x>20, x<20], [SQUARE, PRIME]]`
Given that we know that each group must stay on the same side of the playing-board to avoid affecting each other, we can quickly conclude that there will be two groups on one side(and one tag that is not in any group), and one group on the other(and three additional non-grouped tags).

In our initial attempt by hand, we chose the groups as follows:

SQUARE					
PRIME					
ODD					
EVEN					
TRI-ANGULAR					
	X < 20	**X > 20**	**Factor of 60**	**Multiple of 3**	**Multiple of 5**

Which, at its surface, looked like a perfectly reasonable way of doing things: as far as I am aware, even within the constraints of the number-list provided, there is a number that suits any two tags chosen from different sides.

Thus, how to solve it?

We quickly landed on an approach that - while slow, perhaps - was very steady(and, as we all know, slow but steady wins the race): by drawing an enlarged version of the grid on one of the massive whiteboards in the maths room which we were attempting to solve this in, we were able to write down all the numbers which suited a grid-square's conditions. The digitized recreation is below:

SQUARE	16, 1, 4, 9	25, 36	1, 4	9, 36	25
PRIME	2, 3, 5, 7, 11	23	2, 3, 5	3	5
ODD	1, 3, 5, 7, 9, 11, 15	35, 45, 21, 55, 23, 25	1, 3, 5, 15	3, 9, 45, 15, 21	35, 5, 45, 15, 55, 25
EVEN	2, 4, 6, 10, 12, 16, 18	24, 36, 60, 30	2, 4, 6, 10, 12, 20, 60, 30	36, 6, 12, 18, 24, 60, 30	10, 20, 60, 30
TRI-ANGULAR	1, 3, 6, 10, 15	21, 36, 45, 55	1, 3, 6, 10, 15	3, 36, 6, 45, 15, 21	10, 55, 45, 15
	X < 20	**X > 20**	**Factor of 60**	**Multiple of 3**	**Multiple of 5**

So, great: we have ourselves the beginning of a solution. The way we progressed now split: in our hand-written attempt, we iteratively identified where numbers must be (for instance, the

number three must be used in the square labelled both with “Multiple of 3” and “Prime Number”(this is not limited to cases where there is only a number in a cell, but can also include other cases, such as when a group of numbers is common to the same number of cells as the size of the group)), and removed that number from the rest of the squares. This then left us with this grid(the heading-cards have been trivially re-arranged to suit what we actually did, as the table above was computer-generated):

FACTOR OF 60	6	4	2	1	60
MULTIPLE OF 3	15	9	3	45	12
MULTIPLE OF 5		25	5	35	20
X > 20	21	36	23	55	30
X < 20	10	16	7	11	18
	Triangular Numbers	**Square**	**Ptime**	**Odd**	**Even**

We found that our tag-placement was, in fact, not optimal: there was a square we could not fill.

However, we did not lose hope. After a quick calculation determined that there were only twelve possible tag-combinations(after factoring in the Groups), we decided to write a python program to calculate all of the tag-combinations.

In lieu of copy-pasting the program here, and hoping readers understand python, I will explain the general *process*, and direct any readers curious as to the exact implementation towards the repository here.
https://github.com/padfoot9445/NRich-factors-and-multiples/blob/master/main.py

Firstly, we began by defining *sets* - we had realized that the “writing down of all members which suited a cell’s conditions” was nothing more than writing down the intersection of the set of the two tags(i.e. 1,4 is the intersection between the set of square numbers and the set of factors of sixty). To take this intersection later, however, we had to define these sets at the start of the program.

Then, for every possible combination of tags(which we generated by first choosing one Group to be on-its-own, and then rotating the remaining four free tags, and then repeating this for every Group), we attempted to “solve” the grid.

This was not done at-all beautifully; this did not use some form of greater understanding of the puzzle or concepts thereof to solve grids. Instead, we used an iterative approach, selectively trying vaguely possible solutions(I had attempted a pure brute-force before this approach, but was unable to get any results returned even after five minutes of waiting).

In essence, the algorithm took the first number that it was allowed to, and moved on to the next cell. If it could take no number, it doubled back, and used a different number upstream. If we use an example, simpler grid here:

1,2,3	1,3
3	1,4

The algorithm would, upon encountering the first cell, decide based simply on how the intersection was ordered(here, it is sorted; in reality, the intersections were ordered arbitrarily) what number to take - here, it took the first number, one.

1	?
?	?

Upon encountering the second cell, it would take the number three: one was already used, and three is the number positioned after one.

1	3
?	?

But, upon encountering the third cell, there is a problem: there is *only* the number three to take, and it has already been used! So, the algorithm backtracks back to the second square - and sees that there is nothing else to use there as well.

It thus backtracks again, this time going back to the first square, where it takes the next number(2), freeing up the number one.

2	?
?	?

As the number one is free, the algorithm picks it(as it is in-front of the number three).

2	1
?	?

(If the number one was not in-front, the algorithm would pick “3”, promptly, upon the third cell, have to backtrack back to the second cell, where it would *then* pick “1”)

And then, 3 is the only choice,

2	1
3	?

And then finally, as one has been used, it takes four, and solves the grid.

2	1
3	4

Using this approach, we iterated over all twelve possible arrangements of tags, and found three solutions:

X < 20	16	7	1	6	10
X > 20	25	23	60	24	30
ODD	9	11	5	45	35
EVEN	4	2	12	18	20
TRI-ANGULAR	36	3	15	21	55
	Square	**Prime**	**Factor of 60**	**Multiple of 3**	**Multiple of 5**

X < 20	16	7	1	3	10
X > 20	25	23	21	24	60
ODD	9	11	55	45	35
EVEN	36	2	6	18	20
FACTOR OF 60	4	5	15	12	30
	Square	**Prime**	**Triangular Number**	**Multiple of 3**	**Multiple of 5**

SQUARE	16	36	1	4	25
PRIME	7	23	3	2	5
ODD	11	35	45	15	55
EVEN	18	24	10	6	20
MULTIPLE OF 3	9	60	21	12	30
	X < 20	**X > 20**	**Triangular Number**	**Factor of 60**	**Multiple of 5**

We will seek a mathematical proof of these results, and re-submit if necessary; however, to the best of our knowledge these three tag-arrangements are the only ones which will emit solutions(the number-arrangements themselves were generated using a greedy algorithm; that is to say, it outputs the first valid solution and then exits; hence, there may be different arrangements available for the same tag-arrangement).