Okay so I'll walk through some ideas behind how the algorithm should work first, then give the algorithm, then a very wordy explanation of how the proof works.

So the problem here is that you're given a description of an $n \times n$ matrix, namely the number of 1's in every row and column, and you're asked to reconstruct the matrix. So at face value, this might seem really hard. What if you made the wrong choice? (Ah well you always ask that when a problem is to be solved greedily) How many possibilities do you need to go through before you find the right matrix? What if you mistook the matrix to be solvable when it wasn't?

Well turns out that if we were a little haphazard, we still get a solution. So the high level idea is that we're going to keep track of how much each column still requires, as we move down, row by row. For each row, we take note of how many 1's we need to put on that row. Then how are we going to put down the 1's? How are we going to choose the columns? We are going to do this by prioritising the columns that need a lot more 1's, currently.

Okay so with a little more details, here's the algorithm:

1. Make an array that keeps track of the number of remaining 1's needed per column. So at the beginning this is literally just C, from the input.
2. For every row r, we refer to R and check how many 1's we need to hand out in this row. Let's say for some row we need to hand out t 1's
   - 2.1. Take the t columns that have the most 1's that are yet to be placed.
   - 2.2. Put a 1 for those columns on this row, and subtract the number of 1's the corresponding column needs, by 1.
   - 2.3. If we run out of columns that still need a 1, we output `impossible`.
3. Move onto the next row and repeat until we run out of rows.

Okay so intuitively the algorithm seems to be fine. We need to check two things: (1a) If it were truly possible, the algorithm does indeed give a matrix that satisfies it and (2a) if it were not possible, then the algorithm does indeed say that it isn't. Remember, it would be bad if our algorithm didn't satisfy both constraints! Here it will be easier to rephrase (1a) and (2a) using contrapositivity to say: (1b) if our algorithm says it's impossible, then it was truly impossible, and that (2b) if our algorithm returns a matrix, then it should be a matrix that satisfies the constraints.

I think it should be quite clear our algorithm here works. At every row, we faithfully put down exactly the right number of 1's and every time we did so, we made sure every column still needed a 1. So we definitely satisfied both our constraints. Okay so (2) was easy.. but how about (1)?

Well we haven't used an exchange argument right? I suppose we would probably use it to prove (1a). So let's say that the input provided specified a matrix that was possible to construct, but our algorithm told us otherwise. We now need to show that this gives us a contradiction. (i.e. If it were possible, our algorithm should definitely give a matrix.) Before moving on, let's say that G was the "correct" matrix that satisfied our constraints, and M was the matrix our algorithm was working on. (By our assumption, such a matrix should exist.)

So what does this mean? Well, our algorithm should have made a bad decision somewhere. Let's say up to row $i-1$, everything it did matched up with G. So starting row i, our algorithm slipped up a little. Okay so what kind of mistake could the algorithm have made? Well at the very least, there must have been some cell that it filled wrongly. But you should take note, that trouble here comes in at least doubles! Why?

Well for example let's say the j^{th} column was supposed to be q (based on G), but our algorithm gave it 0. Well since the number of rows assigned in our algorithm still need to be the same as G, (i.e. $R[i]$) and right now we're short by at least 1, some other column that should have been 0 in G is now 1 instead.

So let's say that our algorithm made a doodoo[1] at row i, such that there must be two columns, say.. j and j' such that $M[i][j] = 0$ and $M[i][j'] = 1$ whereas $G[i][j] = 1$ and $G[i][j'] = 0$. Okay, so what else can we infer? Why did our algorithm make that decision? Well by how our algorithm ran, at row i, we know it chose j' because by row i, the number of 1's needed by column j' were at least the number of 1's needed by column j (edit: this fact is useful immediately later on).

Okay so let's keep chugging along, to some further row down below, let's say i' . (Sorry I get that it's hard that I need to keep introducing many variables). There must be such a row i', so that in G, we have that $G[i'][j] = 0$ and $G[i'][j'] = 1$, (edit: we cannot just take any random i', we are guaranteed by the previous fact that such an i' **must** exist). Well what happens then? Well what's going to happen is that the we know the j^{th} column earlier on lost a 1 when it wasn't supposed to, but now it got one back, and likewise for the j'^{th} column it took up a 1 when it wasn't supposed to but now it lost one.

Okay, so what if we had swapped the values? What if we took the feasible solution and inverted these 4 values? Is that still a feasible solution? Why yes! Think about it, the row counts remain unchanged, and the column counts remain unchanged! But that's not the important part. The important part was that, this was what our algorithm should have given us! So would our algorithm have ever complained that it was impossible? No.

In some way what we are trying to say is, "so long as some solution exists, our algorithm will output one of them". Which completes (2), which finishes the proof.

[1] I got bored of calling it a mistake all the time.