2.1) Algorithm

```
def remake_graph(R,C,zeroes, ones):
        Initialize:
        T← 100X128 matrix initialized to all zeroes #will later be output
        #create flow graph to mimic bipartisan matching problem between rows and
        columns
        G←new directed graph with sourse s and sink t

        for i=0 to len(R) do:
                G.append(s,R[i]) #add edge from source every "row" node
                G[s][i]←R[i] #set capacities of edge (source→row node) as row sum to limit
max capacity of that row's max flow path
        for i=0 to len(R) do:
                for j=0 to len(C) do:
                        if (i,j) not in zeroes do:
                                G.append(R[i],C[j]) # add edge between all "row" and
                                "column"  nodes that aren't known to be zero
                                G[i][j]←∞ #initialize all non-zero row-column edge capacities
                                as infinity
        for j=0 to len(C) do:
                G.append(C[j],t) #add edge from every "column" node to the sink
                G[j][t]←C[i] #set capacities of edge (column→sink) as column sum to limit
max capacity of that row's max flow path

        #call Ford-Fulkerson as described in lectures but store all non-negative flow paths
as separate graph G'
        G' ←Ford-Fulkerson (G,s,t)

        for i=0 to len(R) do:
                for j=0 to len(C) do:
                        if (I,j) in ones do:
                                T[i][j]= G'[i][j] #where each G'[i][j] is the max flow from derived
from F-F
        return T
```

2.2) Proof of Correctness

Assertion 1: The algorithm correctly sets up directed graph G that represents the given problem.

We create a directed flow graph by adding a single source node and connecting it to all rows, where the edge weights are the sums of each row. By properties of conservation, this means the total flow into row i is at most R[i]. Similarly, we connected all columns with a sink node t and assigned each edge capacity the sum of each column. Therefore, the total flow out of every column j is at most C[j]. Finally, edges that represent a cell with value 0 in the final table are not added to the graph, and therefore are not considered.

Assertion 2: The algorithm produces another directed graph G' that correctly computes the solution to the given problem.

By properties of Ford-Fulkerson proved in lecture, we see properties of flow and conservation are preserved in the final graph G'. Therefore, the max flow calculated for each cell upholds said conservation, which upholds the constraints outlined by the problem. Finally, the line if "(I,j) in ones do: T[i][j]= G'[i][j]" ensures non-zeroes are respected.

2.3) Complexity

SPACE: To analyze space complexity, we look at all inputs, outputs, and structures.

Inputs: the rows array takes O(k) space, the columns array takes O(t), while both zeroes and ones will take O(p) space in the worst case. Overall, the inputs in total take O(k+t+p)
Structures: since there are k+t nodes and k+t+kt edges in both graph, each directed graph G and G' take O(kt) space. In total both graphs take O(2kt)=O(kt) space disregarding constants.
Outputs: the reconstructed table T is a k x t matrix, which takes O(kt) space as well
All-in-all, the total space complexity can be given by O(3kt + k + t + p) = O(kt) space.

TIME: To analyze time complexity, we look at every line of the algorithm and analyze running time.

1) Initializations of graph G takes O(kt) time since it is a directed graph
2) Running Ford-Fulkerson to construct the max-flow graph G' takes O(ktp) as observed in lectures
3) Filling the final reconstructed table takes O(kt) time since it's a 2-D matrix

Since all other operations are constant or negligible. Therefore, the total running time can be seen as O(ktp)