Group members: Bowei Dong, Ruiyan Song

1. **How did you implement your best backtracking search?**
   We first listed out all the possible values of the cells based only on the arithmetic constraints. For example: if the arithmetic operation is 6x, then the possible values are 1,2,3,6.
   In a simple backtracking search, each cell will be assigned a value from its possible value lists, one cell at a time from left to right and top to bottom, when the row and column constraint allows. For our best backtracking search, in each iteration, we find the cell that has the minimum possible values remained and assign a value from its possible value lists. If one value gets assigned, we will remove that value from the possible value lists of all cells in the same row/column. Similarly, if we backtrack and turn the value of a cell back to zero, we would give back that value to cells in the same row/column that has that value in its original possible value list. If all the possible values have been taken in that row/column (the possible value list is empty), then it will backtrack to change the previous value.
2. **What did you use for a utility function to guide your local search?**
   The total number of conflicts is the utility function in our local search.
   Total number of conflicts = sum(the number of duplicate cell values in every row and every column) + sum(the number of cells that do not satisfy its arithmetic constraints)
   For example, a row of [1,2,3,4,4,4] counts as 3 conflicts. An arithmetic constraint of 6*:[1,2,3,4] counts as 4 conflicts.
3. **How did you choose which nodes to change for each iteration?**
   We would switch the values of the two cells that would result in the smallest total number of conflicts.
   A puzzle of size n*n has n number of 1's, n number of 2's, … , and n number of n's. For example, a puzzle of 6*6 has 6 1's, 6 2's, 6 3's, 6 4's, 6 5's, and 6 6's. In our local search, we first initialize the puzzle by assigning these values randomly to each cell. The reason we do this is that the number of each value never changes since in each iteration we only switch values between two cells. Then, in each iteration, we iteration through all possible pairs of cells to find one pair of cells, which, after switching values, leads to the lowest number of conflicts. We also use a tabu list of size 10 to reduce the possibility of looping among nodes having the same number of conflicts in an iteration. A new pair of cells can only be selected if the resulting puzzle after switching is not in the tabu list. After a pair of cells is found, we switch them, add the resulting puzzle to the tabu list, and continue to the next iteration. The first element in the tabu list is popped if tabu list reaches its size limit. The search ends if the number of conflicts becomes 0, which means the puzzle is solved, or the number of iterations reaches 500.
4. **All group reports need to include a brief statement of individual contribution, i.e., which group member(s) was/were responsible for which parts of the solution and submitted material.**
   Ruiyan: provided ideas, and wrote helper functions for implementing backtrack and local search.
   Bowei: contributed ideas, pair programming with Ruiyan on coding.