For our input generation, we implemented an algorithm that randomized the happiness and stress values for each pair of students. These students are then also put into random rooms. The total stress value was determined by adding up the stresses of each room. We found our optimal solution for our inputs using a simple algorithm that compared how much taking the student out of their room affects the stress or happiness of that room and different rooms. The metric that we used for optimization (happiness or stress) to determine this solution was chosen at random for every student. After the rooms were finalized, we ensured that the inputs and outputs were valid by normalizing stress values for each student pair. We took the room with the maximum stress value and divided the stresses of all pairs with the value so that no room had stress over the stress budget. As an alternative, we first tried to calculate stress and happiness values for each pair manually. However, we quickly found out that such a method was inefficient, especially when trying to determine a high-quality solution.

For our solver algorithm, we implemented two sub algorithms for different sized inputs. For small inputs, we implemented a branch and bound search algorithm. This algorithm created subproblems for each student added to all possible rooms given valid stress conditions. Then we found the best room that the student could be added to and updated our rooms. For medium and large inputs, we implemented a greedy algorithm. The greedy algorithm had each student start off in their own rooms and combined all possible rooms based on the most optimal happiness and stress values. If a room combined with another room contributed less stress than the stress budget and had an optimal happiness and stress gain greater than their current rooms (which can be a room by themselves or a room with several students), these rooms were added to a list of possible merges. We compared room values using the formula: (happiness gain \* happiness gain) / stress gain. All the possible combinations of rooms were compared with each other and the combined room with the greatest value according to our formula was actually merged. If a lone student cannot be optimally combined into any room, that student stays in a breakout room by themselves. In the end, we had breakout rooms that had the most optimum values according to our formula. We thought this was a good solution because while the search algorithm is accurate, it can only be run on the small inputs due to its runtime. While our greedy algorithm does not perform as well as our search algorithm, it is a pragmatic algorithm for our medium and large inputs as we were trying to find the global maximum for happiness given a well-connected graph of students efficiently. For our greedy algorithm, we tried two alternative implementations with similar approaches. Our alternative greedy algorithm combined all possible rooms by iterating through every room and merging with it all possible successive rooms to find the best room based on the formula: (happiness of room \* happiness of room) / stress of room. The algorithm would continue until it can no longer merge any rooms. While the first algorithm used arrays as the main data structure, this algorithm used dictionaries as its main data structure. The algorithm employs the same logic as that of a reverse merge sort algorithm. However, this algorithm would break for certain large inputs so we ran with Mark’s greedy algorithm. We did not use any computational resources other than our local machines. If given more time, we would have further optimized our greedy algorithm. We could have done this by find the best performing value formula, decrease the runtime by implementing more efficient data structures into our algorithm, or decrease the performance costs by shortcutting the number of rooms that are merged and compared.