The process of getting to my current algorithm was largely due to the help of the TAs. My original idea was to place visitors at the far ends of the array of tables. Then, place a visitor in the middle. If there are more than 3 visitor, I would place them between the overall middle and the far ends, determined by whichever had a greater distance. But this idea had a flaw in that I would be stuck on one side, unable to return to the other side.

The second algorithm I came up with was using an iterative approach, where I would place a person at each jump, which were determined by [last table position - first table position]/[number of visitors - 1]. This would work only for well spaced out tables, but for bunched tables with high position values, it would not work.

The final algorithm was based on binary search. Instead of finding position for visitors, I reduced the problem to finding the checking what is the best distance. The algorithm starts with a value we will call "mid" equal to [(the greatest - smallest table position)]/2] because it splits possible distances into two lists. This sets up the logarithmic part of the run-time. A subroutine, called doesItWork, would check if it worked or not and return a Boolean value. It checks it by jumping by a value of mid from the first position, which decrements the number of people by one. Before it starts jumping, the number of visitors is subtracted by one to count for the first visitor at the first table. This jumps continues until out of visitors or pass the last table with visitors left, with the former returning true and the latter false. For each of these checks, we are doing a linear amount of work, since it is based on the size of the table array. If true, it would see if there is a better one by running the function with the lower bound being the position that worked +1. If not, the upper bound would be one less than the position we just tried. Essentially, dividing the amount to check after each check. Once a number is found and the bounds are the same, the algorithm checks if the new bound works or not. If so, that is the maximum minimum distance. If not, the last distance that worked stored as a parameter is the best.

Overall, the run-time is $\Theta(n * \log n)$ because the best distance was found logarithmically and each time, it was checked linearly. This algorithm used a divide and conquer strategy with the possible distances.

This problem is reduced to finding the best possible distance, not placing visitors at tables. I approached this pro