Drew Goldman
Social Distancing - Description

The Social Distancing problem requires an elegant yet simple solution in order to solve large inputs in reasonable time. The objective of the problem was to find the minimum distance between any two people for the optimal orientation of people at the picnic tables, i.e., the arrangement that maximizes the smallest distance between any two people. I approached this problem by defining the range within which the solution could be via two variables, *l* and *r*. The smallest possible distance will always begin as 1 and the largest possible distance, *maxDistance*, will always begin as the *value of the largest table position - value of the smallest table position*. To recursively search for the largest minimum distance, I created a method, *binaryDivide()*. Essentially, *if* some distance, *mid = (l + r) / 2*, within the range of possible values allows for an orientation of *p* number of people such that the smallest distance between any two people is at least as large as that *mid* value, then the algorithm can proceed to search for distances within the range [*mid, r*]. The value *mid* is included in this range because it is still part of the possible solution. *Otherwise*, i.e., the aforementioned conditional is false, the algorithm will proceed to search for distances within the range [*l, mid*-1]. The algorithm decrements mid because it is no longer part of the solution. This process repeatedly divides the range of possible values in half, approximately, until it reaches the base case, at which point *r - l* is no more than 1, i.e., *r = l* + 1 or *r = l*. *If* the base case is reached, then the algorithm will check whether a distance equal to *r* allows for a valid orientation of *p* people. *If* that is the case, then the algorithm will return *r*. *Otherwise*, the algorithm will return *l*.

In order to recursively converge upon the largest minimum distance, *binaryDivide()* utilizes a helper method, *fit()*, which returns whether or not *p* people can be orientated among the tables, given the inputted distance. To determine this, *fit()* iterates through a vector, *differences*, which contains the difference between the positions of every adjacent table from the original input to the problem. I discovered that summing up every element between some range of indices of the differences is the same as taking the distance between the two tables corresponding to those indices. Therefore, we can iterate through the vector of *differences* and add the values of the differences to a variable, *sum*. Within each iteration, *if sum* is at least as large as the inputted distance, then we can hypothetically place a person at the current index by incrementing a counter variable and reset *sum* to 0. *If*, after iterating through the entire vector of *differences*, the counter is at least as large as *p - 1* (one less than *p* because we always place someone at the first table), then we know that that distance works.

This algorithm runs in $\Theta(n*\log(n))$ time because the algorithm recursively divides the range of possible values—which is at most a constant times the size of the array, *n*—to converge upon the correct distance. Therefore, the recursive dividing takes $\Theta(\log(n))$ time, and each division of the range of possible values calls *fit()* which linearly iterates through a vector of size *n-1*, which

takes $\Theta(n)$ each. Performing n operations $\log(n)$ times amounts to $\Theta(n*\log(n))$. I used a vector because it is simple to find its size; however, I could have also used an array.

No pseudocode required, as the code is available to the grader via Gradescope.