Elad Ohayon

My solution to this assignment is developed on 3 observations about the problem. Each ID can only receive money from 1 person (parent) through **one transfer only**, each ID that donates wealth must donate all of it meaning that once designated as a donor, calling setRequiredWealth on an ID is illegal, and that all wealth originates from ID 1. As such, we can build an "imaginary tree" where the leaves are the IDs that are receiving the wealth and the other nodes are the ones donating the wealth. Since each ID can only have wealth transferred it once, we store each "transfer decision" in an array. We must also keep track of data that describes how much wealth a donor currently donated named wealthIntendingToTransfer, how much wealth an id requires named wealthRequired, who are the donors, and parent child relationships named parenChild. With this information we can solve the problem.

The first method that should be called is intendToTransferWealth with parameters(from, to, percentage, isWealthSquared) which indicates that one ID would like to transfer a percentage of its wealth to another and whether that wealth is squared. Each time this method is called, we first run a couple checks to verify that every input is valid such as: it is a member of the population, "to" is not equal to "from", "from" wasn't previously indicated as requiringWealth, to wasn't previously indicated as a donor, and we didn't indicate previously that "to" receives wealth. After these initial checks we add "from" to our set of donors; and add to index["to"] of our array a custom class which stores whether the wealth is squared, and the percent wealth being transferred. We then update parenChild, setting "from" as the parent of "to". Lastly, we update wealthIntendingToTransfer with the most recent call and either add the percent "from" intends to donate if its already donating some percentage, or put it as a new key, value pair.

The way setRequiredWealth works is upon being called it runs checks on the input to verify it is allowed. We check whether the id is valid meaning it is greater than 2, a member of the population, and isn't a donator. We also check to make sure the wealth is positive. After these checks, we update our wealthRequired to store the minimum wealth the ID requires for the problem to be considered solved.

Lastly, after we finish adding data to our problem through calls of intendingToTransferWealth and setRequiredWealth, solveIt is called which returns the minimum amount of wealth ID 1 must have to satisfy everyrequiredWealth. We first check that state we have when entering is valid, meaning that each donor has donated exactly 100% of his wealth and throw an IllegalStateException otherwise. We use a for-each loop to go through each of our transfers and record the amount of wealthRequired to solve the problem. To determine this minimum amount of wealth, we start at a transfer (leaf node) via the array and get the wealthRequired for this ID. We then multiply wealthRequired by 100 and divide by 100* percentWealth where percentWealth is the percent wealth this ID is receiving from its parent. If the transfer indicates that the wealth is squared, we first take the squareroot and then multiply by the percentages. We repeat this process, setting ID to be the parent and calculating the new wealthRequired by multiplying the previous by 100/ 100* percentWealth. We do this in a while loop and keep on going until the parent is ID 1 where we terminate and record wealthRequiredtoSolveProblem. If the parent is null before we reach 1 or an ID has a wealthRequired but no one is intending to transfer to it then the transfer isn't connected to 1 in which case, we throw an IllegalStateException due to an illegal transfer.

To determine the minimum amount of wealth required to solve the problem, we take the Math.max of the previous wealthRequiredtoSolveProblem. Finding the amount required to fulfil the largest transfer will also fulfil the remaining ones. After we go throw all transfers in the array, we return the wealthRequiredtoSolveProblem and return.