WealthTransfer Writeup

For my algorithm, I used a tree implemented similar to Sedgewick's in UnionFind, where each node is represented as an array index, and different arrays store different properties for each node at its index. My `children` array stores the set of all the children the child at each index has. Each of my other arrays stores an attribute a node might have.

My implementation of `intendToTransferWealth()` makes `to` a child of `from` and sets the attribute arrays at their indexes using the other parameters. My implementation of `setRequiredWealth()` only sets a single attribute array element, `wealthRequired`.

My initial implementation of `solveIt()` used a recursion, but suffered from stack overflow if there was a long, linear inheritance. Although you told us that you would not give a population of greater than 1,000, and it wouldn't make sense for a person to donate to their descendant a million generations later, I decided to refactor my code to avoid recursion anyway because I felt it would be a good test of my programming skills.

My implementation (after error checking) uses a combination of a loop and stack to simulate a depth-first postorder tree traversal. Each iteration of the loop can be one of four stages. If the node is a leaf node, it calculates wealth using my Fractional Method, which I describe below, and puts it in the outside variable `wealth` to be accessed by its parent below it in the stack. If this is the first time visiting a parent node, an iterator for its children is added to its stack frame. It is then placed back on the stack, followed by the first of its children. If this is a parent node already visited, then the money the child placed in `wealth` might be stored on the parent frame, based on the Fractional Method. If the parent still has at least one more child on its iterator, we put the parent and the next child on the stack. If the parent has no more children, we place the parent's wealth in `wealth` using the Fractional Method, allowing it to be accessed by the grandparent below it on the stack, or the method's return value if the parent is the root.

My Fractional Method allows me to calculate the minimum amount of money a parent must receive in order to give this child what it demands. First, I square-root the child's money if it receives money through a square transfer. I multiply the money by 100 and divide it by the percentage of the parent's wealth given to the child, which determines how much money the parent must have to give this amount to this child. If this is more than the parent's current required wealth, it becomes the parent's new required wealth; otherwise, it is ignored. This accounts for how a parent giving different percentages of its money to different children could allow for a child to receive more money than they asked for.

In my order of growth analysis, c is the number of method calls, p is the population parameter in the constructor, and s is the size of the inheritance tree. Since `intendToTransferWealth()` and `setRequiredWealth()` run in constant time, the order of growth to run c calls of those methods is O(c). In `solveIt()`, the loop iterates each time it pops a node off the stack. Each node is placed on the stack once when it is first found and an additional time after examining each of its children. Since each child must have a single parent on the inheritance tree, the loop must run $2s - 1$ times (ID #1 is not a child), giving the main algorithm O(s) efficiency. However, to check for certain errors, `solveIt()` must traverse the entire population, even members not on the tree. Since each node is a member of the population, $s \subseteq p$, so `solveIt()`'s order of growth is O(p). Since `intendToTransferWealth()` can only be called once per `from` $\in p$, and `setRequiredWealth()` can only be called once per `id` $\in p$, $c \leq 2p$. This means that the order of growth of calling setup methods followed by `solveIt()` is O(p). I'm not sure why you said this was so hard to calculate.