Lab 1
Arta Seyedian

Findings

Elevator use in most office buildings is important because buildings are usually built vertically for vertical movement (i.e. they go up and down instead of sideways, to save ground space), and because there needs to be an efficient and energy-conserving way of changing one's position in the building with respect to altitude/floor. In ABC Corp., the status of the building elevators is dismal. One elevator is "permanently broken," which, if we were to interpret the word "permanently" literally, as we should, that means that that elevator has absolutely no chance of being repaired and should be removed and demolished to make way for a new, functional elevator. The other elevator's inner dimensions are extremely hazardous, as people are so tightly packed in that they are arranged single file and all the passengers in front of the one unloading at a floor must temporarily get off for that individual to leave the elevator, and then file back in. One could describe this elevator of having a bandwidth of one person. I urge that you discontinue the use of that elevator and contract the expansion of the elevator shaft to make room for a more spacious elevator.

The obvious aside, let us consider the usage statistics gleaned from the simulation. Of the 49 people who attempted to use the elevator, only 32 of them were able to do so. That is a 65% actual usage rate. This is the result of the elevator being full 10 times. The range of the number of times an individual had to get off the elevator temporarily ran from 2 to 32. This means that one individual by the name of Sal had to get off the elevator 32 times to let someone else out before they reached their destination. The results of this simulation reinforce my recommendations, which are, again, to remove and demolish both elevators and reconstruct wider elevator shafts for more spacious elevators. If that is not possible, then use of these elevators are strongly discouraged. These elevators are an endangerment to the well-being of your workers.

Simulation

The simulation was organized into three separate classes: Elevator, Passenger, and Main. The Elevator class contained within it the variables that were most instrumental to the functioning of the elevator. Two arrays held passengers while the elevator was still working; one main elevator array implemented as a stack (elevator Pass) and another temporary holder of passengers (temp) to represent when the passengers would file out to allow a fellow passenger off the elevator.

When Main is executed, the file that has been inputted has its first, data-containing line split by tabs and stored as information about the passenger (name, floor entered, floor exited). Main then calls a method that is designed to determine which floor the elevator should travel to: the floor that the current incoming passenger would like to go to or the closest destination floor of one of the passengers. If the absolute difference between the current floor and the floor of the new passenger is smaller than the absolute difference between the floor that the passenger at the top of the temp stack would like to travel to and the current floor, then the next floor remains the

floor which the incoming passenger would board; otherwise, the next passenger in the elevator is evaluated. That is an example of the temp elevator serving its purpose as a holder of passengers for evaluation.

Regardless of the floor that the elevator eventually goes to, all passengers are again popped to the temporary stack and evaluated once more to determine if the chosen floor is their exit floor. If so, they are removed and ascend to elevator heaven (where people go when they unload from the elevator, represented by array elevatorHeaven in the Elevator class). The Elevator object's isFull method is then called with argument newPass, which is the next passenger, and if the elevator is full, then the passenger waiting to board is condemned to stair jail (where passengers go when they fail to board the elevator; an eternity of taking the stairs, removed from mankind and permanently and totally disfigured in their ability to relate to another human being, represented by array stairJail in the Elevator class). This goes on for each line in the input file, until the end is reached, at which point, the method printStats is called to write all the final counts to a new text file generated in the package directory.

The Passenger class is simple in that it only has four variables, a constructor and a getter method for each of the variables. Passenger also has numTimesTemp, which is a method that, when called, increments the integer value for the number of times they had to temporarily step off the main elevator.

Addendum

In the Main class, the big while-loop starting on line 40 is what handles and executes the entire functioning of the elevator class. Within it, I have a pair of if-else-if statements. One checks to see if the elevator is not full, and the other if the elevator *is* full. If the elevator is *not* full then the code that if statement brackets, which adds passengers, is executed. However, if the *is* full, no passengers are added. There is a method in both of those if statements that are called by the elevator object, named setIsFullNotAgain (for lack of a better name). This method resets a counter in the elevator class known as IsFullNotAgain to 0. IsFullNotAgain is used in the isFull method to prevent the code from printing text in the isFull method multiple times in one iteration if the isFull method already returned true. Why this happens to begin with is because each time the isFull method is called, the accompanying text about the elevator being full is also called, and this was an easy way to circumvent that problem. I could have removed the text from the method and used it in Main, but then I would have to create a setter method for stairJail, and that would overly complicate a program that was already sufficiently complicated for me. There are many ways to render any program more efficient, but given the relative abundance of computing power, especially regarding a program like this, the amount of power conserved could be considered negligible. Because of my use of arrays as stacks, and how the array must be sifted through to assess and evaluate properties of each array element, the big O notation of the program could be said to be close to O(n). Large input files (>100 mb, for example) could potentially slow down the performance of the computer, but most modern computers could execute this program almost immediately.

Using stacks for a project like this is the most obvious choice, due to the unique circumstances of this problem which I outlined earlier and include the small bandwidth of the elevator and the stack-like nature of the elevator itself. There is an in-built stack class in java which I could have also used to similar effect but implementing an array to do the same thing gave me an intimate understanding of how stacks work, and how pointers can be used as the stack head.

In addition to gaining perspective about stacks, I also learned about how to conduct data analysis using java. In my field of bioinformatics, java is not usually considered for development and scripting. However, it is similar enough to many other languages in concept that I have gained much from my practice of java coding, and I can add processing tab-delimited text files to my area of experience with java. If I were to spend more time with this simulation, I would try to make it more efficient than it is now; remove or consolidate redundant methods and statements and minimize the amount of code in main without over burdening other class methods with extraneous code. I would also compare the output of this program with that of others' programs to try to determine if this program was accurate.