Zachary Shaffer and Adam Spence
Professor John Wenskovitch
Analysis of Algorithms
May 02, 2016
Allegheny College

Allegheny Room Draw Number Generator - Report

At this point in time, our project is complete and runs as intended. The main program, RoomDrawGenerator.java, read in three input files, “soIn.txt,” “jrIn.txt,” and “srIn.txt.” These text files represented data that each student would have in the Allegheny School system, including their grade level, name, what kind of disability the student has (if any), credits taken this semester, previous residence hall, and previous room draw number. The data in the files we use are randomly generated with a small external program we wrote to automate the process of making pretentious data. The names of the students are simple generice “HumanX,' where X is a number between 1 and 100, and each class has 100 students.

We stored all of this data in Student objects, with our Student.java. Student.java was a simple object with a lot of getter/setter methods. One unique thing that we hadn't done before was we did an override of the compareTo method, and made the object implement Comparable. That way, we could use Collections.sort() to sort ArrayLists of Student Objects based on their room draw number very quickly. Collections uses a modified mergesort algorithm, and the overall complexity of the sort is O(N*logN). Prior to sorting the ArrayLists, however, the ArrayLists are shuffled as soon as they are created. This is to prevent bias by student name. The simulated names we chose, Human1 through Human100, helped us to make sure this type of bias did not happen, as the names simulated a list in alphabetical order, as the school system probably would have them ordered. Collections.shuffle() runs in an overall time complexity of O(n). Our code for reading lines from the text files also reads the lines one by one, for each file, and therefore also runs in O(n).

Once all of the Student objects are made and stored in ArrayLists, we then send the ArrayLists to DrawCalc objects. These objects hold data that were required for the actual algorithm for computing draw numbers - things like the number of students in that particular class (since each ArrayList was a grade level of students), the median room draw number, the average number of credits for that grade level, and a primitive integer array for the actual generation of room draw numbers. The way the array worked was that we designed an iterative linear probing style of number insertion method, that received two parameters. The first was the number where the generator intended the student to go on the first try. If the spot were already occupied, the method would iterate across until it found an empty spot. The second number was the position in the ArrayList of the Student object for which the number was being generated. So if the Student in ArrayList position 64 generated the room draw number 32, the method would check the 32nd slot in the roomDrawNums array. If the spot had a -1, the default we chose to initialize the entire array with, it was empty and therefore would have the student's position (64) stored in that index (32) of the array. Otherwise, it would check the next index (33) and try to store it there. If it ever got to the end of the array, it would double-back to the very beginning and continue that way, but a student would have to get very lucky for that to happen to them. This makes O(n) checks in the worst case scenario.

Just about every step of our algorithm was, at worst case, O(n). We determined this to be absolutely fine for the nature of our project, because n would be the size of the student class, which, unless Allegheny had a massive influx of thousands of students in the next few years, would never really reach above 500 or 600, relatively small for a computer to run. We decided to test how far our system of linear order of growth operations ran with different student sizes. To do this, we made extra pretentious data files, with 200, 300, 500, and 1000 students.