A) **Names:** Josh Levin and Hart Neray

Repo: neraylevin

Rent and Room Allocation Algorithm

B) **Description:** We used a combination of a Dutch auction and the Moving Knife algorithm to divide up rent and rooms in an apartment with three people. In the end, our algorithm allocates one room to each player and no player envies another for more than a previously agreed upon amount (resolution/epsilon).

C) **Summary:** The problem we examined is how to properly divide up rent and rooms among three people sharing an apartment. Given that people can (and most likely will) value different rooms differently, we tried to find an approach that would minimize envy between the people. To do this, our algorithm uses elements from Dutch auctions and the Moving Knife algorithm.

Initially, each person in the apartment submits the maximum amount he or she would be willing to pay for each of the three rooms. Those amounts must sum to the total rent for the apartment. Then, using a Dutch auction, each room is bid on one by one; the player willing to pay the most for any given room gets it for the amount he or she was willing to pay. Similar to the version of Dubins Spanier that allows a player to trade in his or her piece if they see another one worth epsilon bigger than his or her current piece, we allow our players to re-enter the bidding (even if they already have a room). To ensure a bound on envy, the player must only say stop a second time if they can get the room for epsilon, an agreed upon value that is less than the total cost of rent, less than what they originally bid. If the same player wants to say stop a third time, he or she must be able to get the room which he or she wants to bid on for 2*epsilon less than his or her initial valuation of the room. A player may only say stop (total cost/epsilon) times. At the end, if the players' payments sum to over the total cost of the apartment, we take

the excess money and divide it evenly between all parties (or give it to the charming lads who provided them with a fair means of dividing up rent).

Proof that the Algorithm Works: Since, as far as we are aware, no papers have been published that combine these two algorithms, we found it necessary to show that such an algorithm will result in rent being paid in full. We will use P1, P2, P3 to denote players one, two, and three. R1, R2, and R3 will represent rooms one, two, and three. B1, B2, and B3 will denote the amount of money player one, two, and three each paid for his or her room. Our general approach in this proof is to consider the last player who gets a room; as we'll show, that'll be (WLOG) P3 in this explanation. Given that the algorithm terminates as soon as the last player gets a room (i.e., once all three have a room, there is no more swapping allowed), we show that the total cost of the apartment will be paid.

WLOG, say P1 bids the most for R1. Then, for R2, there are two cases: either P1 receives R2 or one of the other players does. If P1 receives R2, we can view the bidding process as essentially restating; there will still be two rooms left with two players without a room. In this sense, we can rename R2 ->R1 R1->R3, and R3->R2 and continue; Eventually, one of these remaining two players, (WLOG say p2) will receive R2 since it is the next room to be bid on after p1 receives a room. At this point, the two cases of whether P1 or another player gets the second room to be bid on have essentially become the same case, since there are two players with rooms and one without. Note that regardless of how many times the players swap, we will eventually get to a state where the one player without a room, in this case P3, will receive his or her room. We're interested in the implications of that event. Similar to the scenario above, if P1 or P2 has the highest bid for R3, then WOLG we can swap labels between their old room and R3. As P1 and P2's evaluations diminish, eventually P3 will have the highest bid for the room being auctioned.

Once P1 and P2 have received rooms, if P3 wins the bid for a room, we are done. We also know that this is the first time P3 receives a room; if P3 had had a room beforehand, then the algorithm would've terminated earlier because once a player wins a bid on a room, he always has a room from that point onward—whether he swaps his rooms or not. Since P3 had no room previously , he received R3 (WLOG) for the value he originally put down for it; he never re-entered, so none of his valuations have decreased by epsilon. We also know that no matter how many times P1 and P2 swapped in for new rooms, the amount P1 and P2 paid for those rooms is greater than P3's evaluations for those rooms (i.e., B1(R1)>=V3(R1) and B2(R2)>=V3(R2)) because p3 would have gotten those rooms otherwise. Since, V3(R1) + V3(R2) + V3(R3) = total cost, it follows that B1+B2+B3>=total cost, since B3=V3(R3).

Proof that no player has more than epsilon envy: We define this as having no player see another player get a room for more than epsilon less than what the first player would be willing to pay for it. To start, we'll show that the last player to receive a room cannot envy the other players.

So, WLOG, say P3 is the player that chooses his or her room last. P3 cannot envy P1 or P2 because P3 picked last and thus must think that P1 and P2 paid more for their rooms than P3 would've paid. Whenever P1 or P2 trade in for a new room, P1 and P2's valuations of every room is lowered by epsilon (i.e., they're only willing to trade in the future for prices at epsilon lower than their current valuations). Note that if a player can say stop (total cost/epsilon) times, and his or her values are lowered by epsilon each time, then epsilon*(total cost/epsilon) = total cost. That means we know that a player cannot say stop (total cost/epsilon) times without paying 0 for a room.

By the end, either P1 has swapped a room or has not. In the case in which P1 has not swapped, that means P1 received his or her room for his or her original valuation of that room.

That also mean that no room could have been claimed for less than P1's original valuation of said room minus epsilon, since P1 would have claimed it otherwise. If P1 has swapped, though, then P1's preference for every room is epsilon less than his original value. That means that no player could've claimed a room for less than 2*epsilon less than what P1 originally valued it at (otherwise P1 would've claimed it); regardless of the number of times P1 swaps, his valuations decrease by epsilon, so there's no time another player could receive a room for more than epsilon less than P1's valuation for that room. Thus P1, cannot envy anyone more than epsilon. In this same way, WLOG, we can say the same for P2 (assuming P3 is still the last player to receive a room).

Essentially, for each auction p1 will never allow a player to claim that room for (number of times p1 has swapped)*epsilon less than P1's original valuation. Thus from the point of view of p1, the other players must have gotten a deal no more than epsilon better than theirs: V1(R1)-B1>=V1(R2)-B2-epsilon. WOLOG the same can be said for P2.

D) In order to write code for our algorithm, we first had to utilize the preference files in a manner compatible with our goals. For any given preference file, we wanted to "extract" valuations of the first, second, and third rooms. To do this, we calculated the area under the first third, second third, and last third of the graph and used those values as the player's valuations of the first, second, and third rooms respectively. We used Professor Cytron's code to help accomplish this. We did this for three preference files (one for each player).

We then created player objects based off of those evaluations. In each player instance, we stored the player's name, his or her preference file, and list containing the player's valuations for all three rooms ordered by room number.

For the actual algorithm, we used an outer while loop that runs until all players have a room; once all players have a room, the algorithm should stop. We also maintain a list of three

integers to indicate which player owns a room at any given time. The list is ordered/indexed by room number, so the first item relates the owner of room one. For each room, an integer indicates which player owns it: 1 for P1, 2 for P2, 3 for P3, and 0 if none own it.

Within the outer while loop, we have a for loop that runs through the three rooms. For any given room, the algorithm finds the player with the highest valuation of that room. When a player receives that room, his or her valuations of every room are lowered by epsilon, because we only want that player to get a room in the future if that player can find a room for epsilon less than his or her current valuations. When a player receives a room, the list of room owners is updated to reflect ownership of this room, and if the player already had a room, the owner for the room he or she previously is set to 0 to indicate it's back in the market. At any given iteration of this inner loop, a conditional checks to see whether all players have a room—if they do, both loops terminate. Finally, we add up all the money paid, take the difference between amount paid and total cost of the apartment, and divide it between all the players equally.

E) At first we used basic preference files so as to make sure that the algorithm was doing what we wanted it to. Then, we tested the algorithm by running it on randomly generated preference files (using Professor Cytron's code for the generation of these files) and calculating the amount of envy each player had. First, we looked at the price each person got for his or her room. But we also examined the amount each player would have been willing to pay for each of the other rooms at the end in comparison to how much each room was actually purchased for. We used the difference between those two amounts as the envy for any given player to another player, so there were six such values (2 for each player * 3 players = 6). Often, these values were negative, indicating (as we would expect) that a player was willing to pay less than the player who received a room (i.e., no envy).

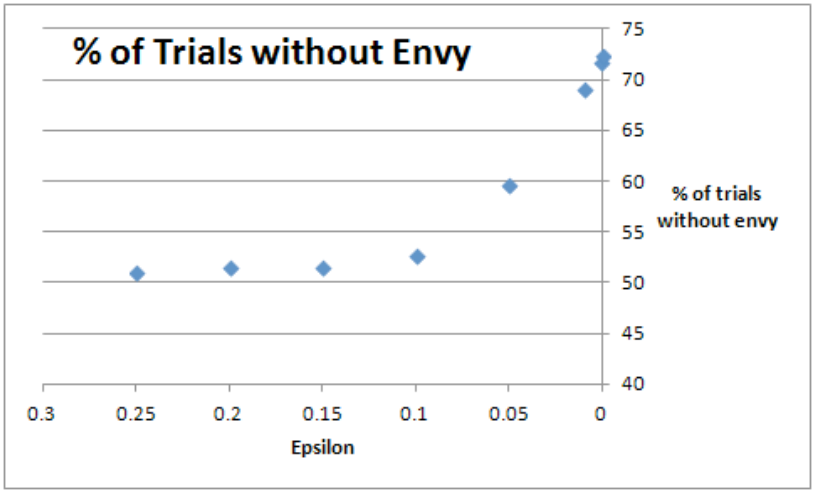
After calculating the amount of envy each player had for each other player, we found the maximum of all of these 6 envy values. We then ran the algorithm 10,000 times, calculating the average of these max envies (if they were > 0). We also kept track of the max envy recorded in any given run of the 10,000, as well as the number of runs that had no envy. We ran this procedure several times, altering the value for epsilon (a parameter provided in the demo method) along the way to see the effect it had on envy.

F) Here is what we found over 10,000 trials for varying values of epsilon:

Epsilon	.250	.200	.150	.100	.050	.010	.00100	.00001
Max (of maxs)	.228	.198	.150	.100	.050	.010	.00100	.00001
Average (of maxs)	.022	.022	.021	.018	.009	.002	.00015	.00000145
%of trials without envy	51.0	51.5	51.5	52.7	59.7	69.1	71.750	72.3

*note that even at epsilon=.001 the run time is trivially short. (~12 seconds)

CHANGE BOTTOM AXIS OF RIGHT GRAPH SAYS EPSILONG



G) As shown by the charts and tables, there is very little envy on average. As expected, sometimes a player envies another player by epsilon but never more than that. Also, as epsilon gets smaller, the chance that none of the parties will envy each other grows, becoming >%70 at epsiolon = 0.01.

One potential issue with our data is that we are used completely random preference files. When looking at an apartment, however, it is often the case that everyone values certain rooms more than others. Although we expect envy to never exceed epsilon, we are not sure how this would play out for some of the other values we looked at.

There are several ways we could proceed with our existing code, data, and algorithm:

1) While we did not implement this, we could adapt our code to allow a user to input the number of players; it may be interesting to see the results with more or fewer players.

2) We could adapt the algorithm so that each player could choose his or her own epsilon.

3) Potentially, we could try to change the algorithm so that it doesn't terminate until total rent has been reduced to the actual price. That is, we repeat the algorithm recursively until the correct rent is reached. If during the auction of a room, exact rent is reached, a referee says stop, and the room being auctioned goes to the current owner for the price the referee stopped at.

4) It'd be interesting to consider profitable applications of this algorithm. It could be used as a standalone application, but it might make more sense as a tool for a site like airbnb.com to allow users who rent multiple rooms together to determine costs and distribution of the rooms. The creators of the algorithm could receive a portion of the total cost paid, or perhaps even the excess cash that would otherwise have been split between the users.

5) This algorithm may be applicable to other problems. It should be usable in any situation with a number of players, equal number of objects to distribute, and a one-room per/player condition.

Overall, we are very happy with the result of our algorithm and its implementation. We may decide to apply it to a business or publish it.