Yadel Abraham
Aldo Mendez

Final Project

Initial Ideas and Pitfalls:

Initially we tried to reduce this problem into a Integer Linear Programming instance: First off, we tried to figure out exactly how to formulate this as an LP, and we had the following:

For an item *i*, we are given the following: class, weight, cost, and resale value, let's call these $class_i,\ w_i,\ c_i,$ and r_i, respectively. Then, we denote the following variables as well: $p_i = r_i - c_i$, which is the profit gained from buying and selling item *i*, and $b_i \in \{0,1\}$, which is defined so that we either take or don't take item *i*. So, our objective function becomes

$$\max\ \Sigma_{j \in items} b_j p_j + (M - b_j c_j)$$

So we tried to maximize the sum of our total profits and whatever is left over from not buying certain items.

As for the constraints, setting up the typical knapsack constraints such as sum of the weight of the items must be less than the total weight allowance, cost of the items must be less than the budget, etc. are all pretty straight forward.

Where it got tricky is the incompatible class constraint. What we have done so far is the following: we created a *C*-by-*C* matrix (initially instantiated as a matrix of ones), where *C* is the total number of classes, let's call this matrix *A*. This "constraint matrix" follows roughly the same idea as an adjacency matrix. For every incompatibility between any 2 arbitrary items *i* and *j*, then the $(i,j)^{th}$ entry (and $(j,i)^{th}$ entry due to symmetry) would be changed to 0. Interestingly, this should give us a symmetric matrix. The part that we now needed to focus on is finding out a way to create linear constraints from this matrix.

Lastly, once we had this entire LP setup, we were planning on using the PuLP library for Python (2.7) so actually solve the LP, which looked pretty straightforward to use, so long as we have the entire LP set up properly.

After our linear programming approach failed. We decided to use a graph representation of this problem to run the knapsack algorithm followed by independent set to handle the constraints. the idea would have been to pick the items according to the knapsack algorithm then make vertices out of these items and then finding approximation of independent sets on that too handle the conflict. With this approach we wanted to utilize the knapsack algorithm with some approximation to help us pick the items smartly. However problems arise as we tried to handle the conflict and constraints of items that can be in the GargSack. The main issue for us was we couldn't figure out a good to heuristic to solve the independence sets problem. Upon asking Professor Prasad he clarified that the independent sets problem has no good approximation. Therefore, we throw away the code we wrote for that approach and then decided to stick to our new greedy approach.

Final Approach: Greedy Solver

This was the the direction we ended up going to solve the problem. In this approach we pick from random items based on the weight-over-profit ratio for an item *i*:

$$\frac{w_i}{p_i} = \frac{w_i}{r_i - c_i}$$

Normally, it would make sense to formulate it as the highest profit-to-weight (dollar to pound) ratio. However, the priority queue implementation we used is from CS188, which prioritizes items from lowest to highest values. Hence, having a higher dollar-per-pound ratio is the same as having a lower pound-per-dollar ratio, which is what we ultimately used. We calculated this ratio for all the items and then pushed them onto a priority queue. At each point we pop the biggest item from the priority queue. Then, as long as that item's weight and cost is less than our max weight and Max budget, we put it into our list of items also known as our GargSack. Also, we handled our constraints by keeping track of what classes of items we have added to the GargSack, and ensuring that the item we have just popped off the list is compatible with the items added so far (again, this is done greedily).