For the purpose of this problem, plot the different advisor curves on the same plot to compare their performance by calling measureTimes with subjectSizes being the list [10, 20, 30, 40, 50], maxWork set to 40, maxLottery set to 10 and numRuns set to 5 (it might take some time to finish all the runs, so make sure you get it to work on smaller samples first before generating the final plot).

Make sure you choose a new random sample of subjects for each run.

Use cmpRatio as the comparator for the greedy advisor.

(1) What trend do you observe among the three advisors?

(2) How does the time taken to pick subjects grow as the number of subject used increases?
(3) Why do you think that is the case? Include your answer in the write-up along with the plot.

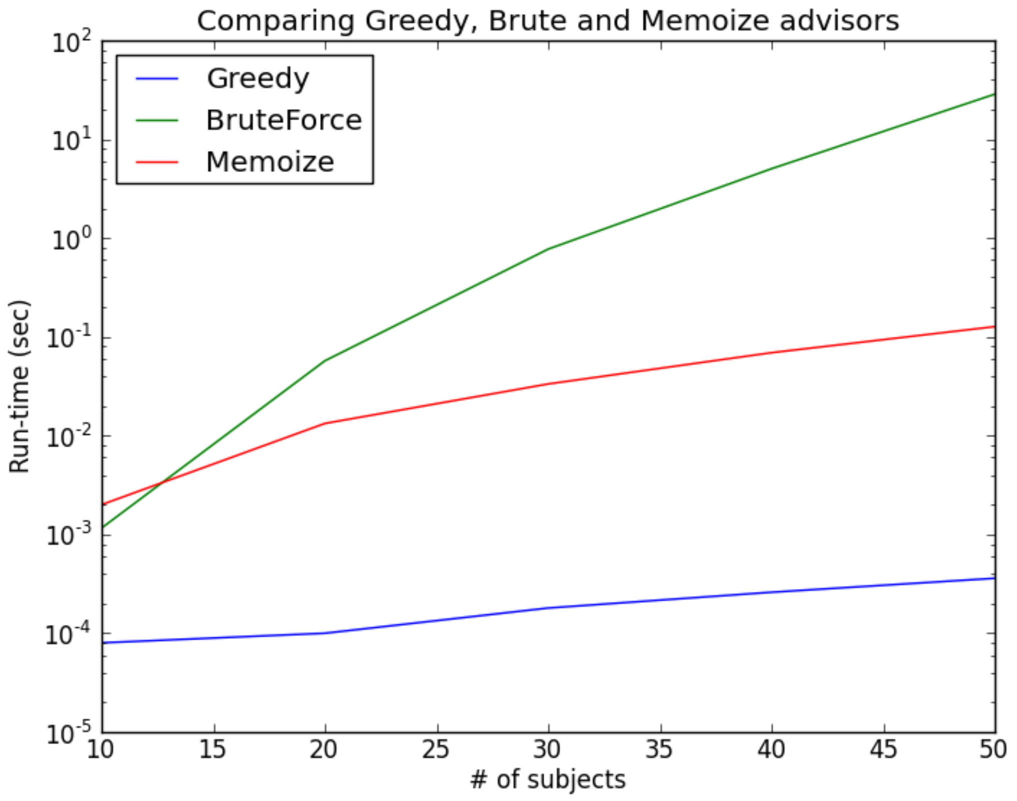
Do not generate the plot every time your code is run since it takes a while to run.
That is, running your ps9.py file should not output anything. We should be able to generate your plot by calling the measureTimes function you wrote.
Also, make your plot look nice with titles, labels, and a legend.

Considering that we're using algorithms that sometimes have logarithmic or exponential growth, you may find it useful to visualize your plot with a log scale on the y-axis.

As the number of subjects increases, run-time for Greedy increases relatively slowly; this is because the sorting is in O (n log n) (note: the for loop is O(n)). Once sorted, Greedy just loops through the list adding items that fit within the constraints and skipping over those that do not.

BruteForce and Memoize both take longer than Greedy across inputs. Like greedy, Memoize appears to increase sublinearly when graphed on log scale, suggesting it increases polynomially. In contrast, BruteForce increases exponentially – in the worst case, it is 2^n as it considers all possible subsets (even though it is better than the bruteForce algorithm we implemented for the last pset using partitions—as that created all the partitions first, whereas this recursive BruteForce discards the subject whose attributes would exceed constraints of maxWork or maxLottery).

It is interesting to note that for very small inputs, Memoize is slower than BruteForce (due to the overhead of storing & retrieving information from the dictionary); however, the rate of re-use (of the sub-routines) appears to grow rapidly as BruteForce rapidly increases far more quickly than Memoize.



```
greedy: OrderedDict([(10, 8e-05), (20, 0.0001), (30, 0.00017999999999999998), (40,
0.00026), (50, 0.00035999999999999997)])

brute: OrderedDict([(10, 0.00116), (20, 0.05717999999999995), (30, 0.77256), (40,
5.028059999999999), (50, 28.565139999999996)])

memoize: OrderedDict([(10, 0.002), (20, 0.01328), (30, 0.03334), (40,
0.06910000000000001), (50, 0.12684)])
Total run-time is: 186.799893141
```