Announcements
● Midterm: Wed Nov 12 7:30-9pm, room 10-250
● Conflict midterm: Fri Nov 14 8-9:30am, room 4-270
● PS3 due Mon Nov 17 11:59pm

Part 1: Optimization (20 min.)
Any problem where you want to find the biggest, smallest, fastest, least expensive, etc. solution is an optimization problem.

Intro (5 min.)
Every optimization problem has 2 parts
1. an objective function: the thing that needs to be optimized (i.e. maximized or minimized). For example, the cost of a trip through Europe.
2. a set of constraints that must be honored. Might be empty. For example, you want to go to Amsterdam, London, Paris and Madrid and Rome, but you don’t want to take more than 2 train rides.

Approaches
● Brute force algorithm
  ○ Enumerate and look at all possible solutions
  ○ Can make a decision tree to describe all possible solutions
  ○ Usually takes a long time (we can figure out exactly how long later)
● Greedy algorithm
  ○ Continually pick “best” valid choice in each step of the algorithm until requirements/constraints are met
  ○ Faster than brute force
  ○ But, this only finds the local optimum, which may or may not be the global optimum

Example 1: Groceries
● (Note: This is a variation of the knapsack problem. In general, a knapsack problem means maximizing value of stuff in your knapsack, subject to the constraint that your knapsack can only fit so much weight/volume)
● You are a starving grad (or undergrad student). You are stocking up on groceries for the week, so you want to maximize the number of calories in your knapsack, but you only have $50.
● The only grocery store near you is a very overpriced place called Whole Grubs, and their selection is very limited. Here are your choices:
  ○ Ramen cup: 200 cal, $3
  ○ Steak: 600 cal, $23
  ○ Cheese block: 400 cal, $5
Apple: 75 cal, $1

Brute force solution (5 min.)
- Draw decision tree to enumerate all possibilities
- Each path down the decision tree is one possible answer (e.g. [ramen, ramen, ramen … ramen] = 16 ramen cups)
- Pick the best valid possibility

```
Ramen
  `- Steak
      `- Cheese
          `- Apple
```

Q: What is the algorithmic efficiency?
A: Exponential $O((\text{total # of items})^{\text{(max # of items we can buy)}})$.

Brute force can always get the best answer (global optimum), but is often prohibitively slow.

Greedy solution (10 min.)
- In this case, let’s try to maximize for calories (you’ll see in the edX exercises that you COULD also maximize for calories/cost)
- Pseudocode
  1. Add an item (in summary: pick the biggest valid item from all possible items)
     1.1. Look at every possible item
     1.2. Can you afford it?
     1.3. Is it more better (more calories) than your previous best?
     1.4. If yes to both, remember it
  2. Add best item to cart
  3. If you can’t afford any more items, return
  4. Go back to #1
- Q: What is the running time of this algorithm?
- A: $O(\text{total # of items}) \cdot \text{(# of items we can afford)}$

Run the code in `grubsack.py`. The solution given by the greedy algorithm is:
  1 ramen cups
  2 steaks
0 cheese blocks
1 apples
Total calories: 1475

Is this the best we can do? Obviously, even buying 50 apples would be better.

**Greedy algorithms are faster, and always find a local optimum, but are not guaranteed to find the global optimum.**

**Part 2: Curve fitting (35 min.)**

When we collect data, it's usually because we want to create some theory or model of how some process works, and then validate that model. We can come up with models by doing curve fitting. Most of you have been making best fit curves since high school, but did you know that curve fitting is actually an optimization problem? How does that graphing program know how to draw a best-fit curve through a bunch of data points?

**Objective function (5 min.)**

Like any optimization problem, curve fitting requires an **objective function**; that is, something to tell us how good the fit is. The most commonly used objective function is **least squares**.

The objective function in this case is:

\[ \sum_{i=0}^{N-1} (\text{observed}[i] \text{ - } \text{predicted}[i])^2 \]

Where your data set is N points long. If we wanted to come up with a fit curve for a data set, we would make a list of N predicted values using some equation that minimizes the value above.

How would we go about doing this? (haha) Luckily, pylab has a function that does it for us using linear regression. This function is called **polyfit**.

**If anybody asks, it's called linear regression because it models the relationship between Y and x^n, x^(n-1), etc. in terms of a linear function**

**The “RankWarning: Polyfit may be poorly conditioned…” warning that pops up in IDLE occurs for high-order polynomials, where numerical errors due to super large values of x^n may cause the best fit to not actually be that great**

**Part A: Training and test sets (5 min.)**

We're going to be looking at space cow populations. Before we get started on plotting curves, we need to figure out what data to use. For a lot of real-world data, we can't actually set up a controlled experiment, but we can simulate an experiment by dividing the data we already have into two sets: a **training set**, and a **holdout** or **test set**. We use the training set to create our models (i.e. best fit curves), and the **test set** to validate the model. Usually, we pick our training set to be representative of our data (Q: how? A: random sampling). In this
case, we have data from 2 space cow breeding locations and we’ve randomly sampled to get our training set.

**Part B: Plotting polynomial fit curves (10 min.)**

spacecows.py has some examples of polynomial fits of different degrees. We use `pylab.polyval` to evaluate the values of a polynomial function at the given values of $x$.

(show part B of spacecows.py)

What can we see in the plots?

- linear fit is too simple (often the case for many real-life data sets)
- decatic fit looks good

**Part C: Goodness of fit (10 min.)**

How good are these fits? How do we numerically describe the goodness of the fit?

- (OPTIONAL: We could use mean square error (as defined above). The minimum of this expression is 0 (perfect fit). But the maximum is unbounded. At what point do we say that the fit is bad? e.g. is a value of 50 okay? What about 100?)
- Instead, we often use the $R^2$ value, or the coefficient of determination, to describe the goodness of a fit.

$$R^2 = 1 - \frac{\sum (\text{observed}_i - \text{predicted}_i)^2}{\sum (\text{observed}_i - \text{mean of observed})^2}$$

- Using this value, 0 indicates no relationship, while 1 indicates a perfect fit.
- Our $R$ squared values indicate that the decatic curve has a really good fit on the training data (almost perfect)
- In fact, we can look at $R$ squared with respect to polynomial degree on the training data data
- Q: Why don’t we just use a high degree polynomial for everything? A: because a good fit on the training data doesn’t necessarily mean a good fit on test data

**Part D: Under-/overfitting (5 min.)**

Our linear model is underfitting because it isn’t really representing the data that well. But what about our other models? Let’s see how they do in another couple of years.

(run part D)

Our quadratic and decatic models give ridiculous results in the year 5. They are **overfitting**, which means that the fit is capturing more noise than meaningful relationships. This is common of higher-order polynomials, or in general, more complex models.