Introduction, Optimization Problems, and a little more Python

(download slides and .py files from Stellar to follow along)

John Guttag

MIT Department of Electrical Engineering and Computer Science

6.0002 Prerequisites

- Experience writing object-oriented programs in Python 3
- Familiarity with concepts of computational complexity
- Familiarity with some simple algorithms
- 6.0001 sufficient, but not necessary

Question

I'm in 6.00
I took 6.0001 this term
I took 6.0001 a previous term
I took 6.0001 ASE
None of the above

Some Administrative Stuff

- Stellar course site
  - https://stellar.mit.edu/S/course/6/fa18/6.0002
- Post privately on Piazza rather than emailing staff
- Course uses Python 3 (not 2)

Course Policies

- Collaboration
  - Okay: Helping others debug, discussing general attack on problem
  - NOT okay:
    - Copying code (from others in class or previous years)
    - Allowing others to see your code
    - Side-by-side coding
  - Provide names of all “collaborators”
  - We running code similarity program on all psets
- Extensions
  - Will consider requests that come with support from S^3
  - Late days, 3 to use at your discretion

Grading: Problem Sets and Finger Exercises

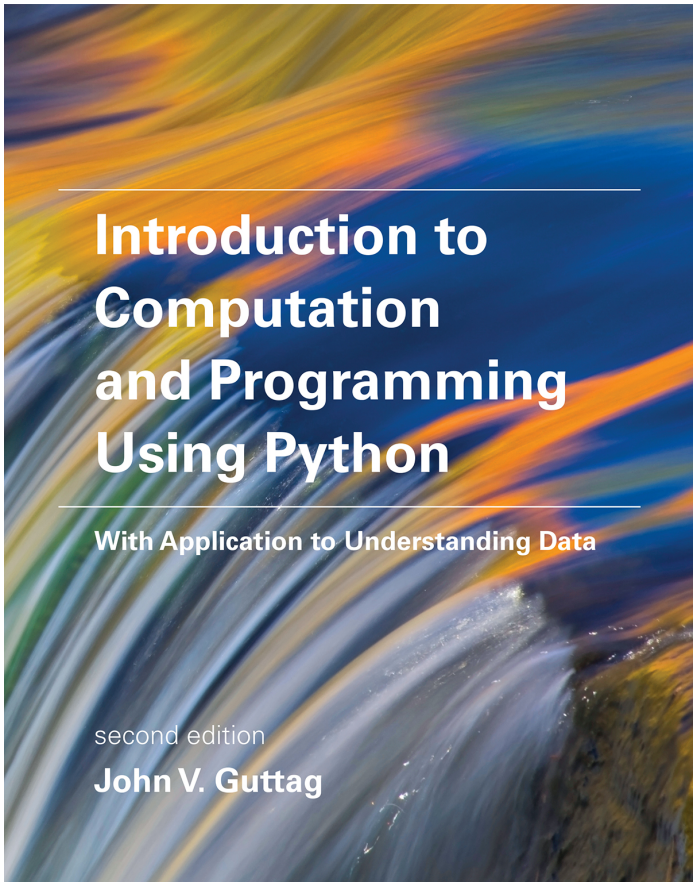
- Problem sets
  - 30% of final grade
  - Due on **Wednesdays at 5PM** (change from 6.0001)
- Finger exercises on MITx
  - Mandatory exercises 10% of final grade
    - One per week
  - No extensions on due dates

Quizzes and Exams

- Microquizzes
  - During class, last 20 minutes of some lectures (see calendar)
  - Must have computer with wireless connections
  - 3 (worth 20%, best 2 out of 3)
  - No makeups
- Final on 12/12 during normal lecture time
  - 40% of grade
- Quizzes and exams will over material from lectures, problems sets, and assigned readings

Relevant Reading

- Section 12.1
- Section 5.4 (lambda functions)



Corrections posted on course web sites

How Does 6.0002 Compare to 6.0001?

- Programming component of assignments a bit easier
  - Focus more on the problem to be solved than on programming
- Lecture content is more abstract
  - Includes things not needed for problem sets, but relevant for exam
- Lectures will be faster paced
- Less about learning to program, more about dipping your toe into data science

Honing Your Programming Skills

- Quite a few additional bits of Python
- Software engineering
- Using packages
- How do you get to Carnegie Hall?

Computational Models

- Using computation to help understand the world in which we live
- Experimental devices that help us to understand something that has happened or to predict the future

- Optimization models
- Simulation models
- Statistical models

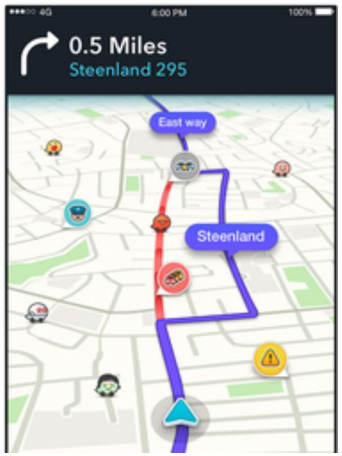
What Is an Optimization Model?

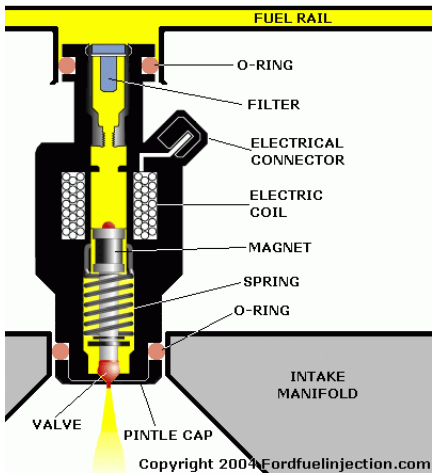
- An objective function that is to be maximized or minimized, e.g.,
  - Minimize money spent traveling from Boston to NYC

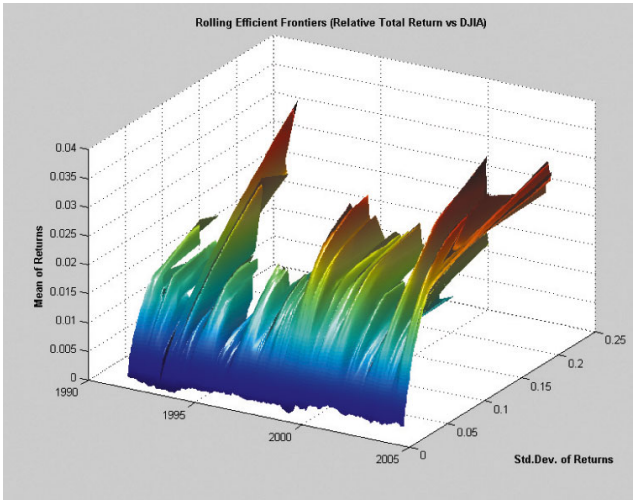
- A set of constraints (possibly empty) that must be honored, e.g.,
  - Expected transit time < 5 hours

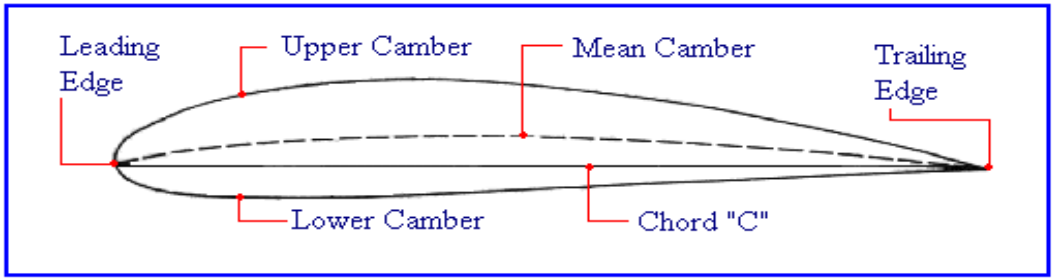
Optimization Problems

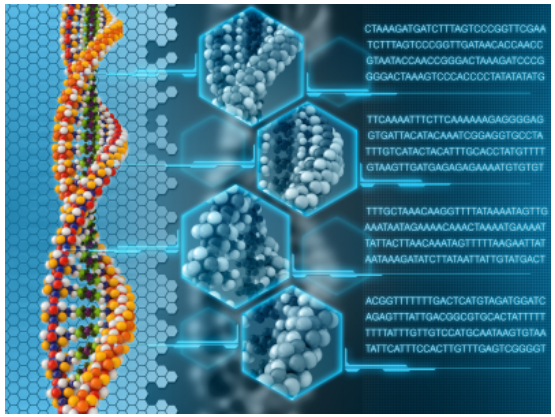
- Anytime you are trying to maximize or minimize something, you are solving an optimization problem



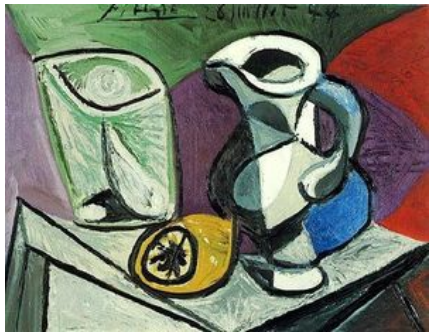








Imagine that You Are a Burglar



Knapsack Problems

- You have limited strength, so there is a maximum weight knapsack that you can carry
- You would like to take more stuff than you can carry
- How do you choose which stuff to take and which to leave behind?
- Two variants
  - Continuous or fractional knapsack problem
  - 0/1 knapsack problem

versus

A Fun 0/1 Knapsack Problem

name	fg	ft	X3pm	reb	ast	stl	to	blk	pts	score
LeBron James	1.769	0.456	0.813	1.146	2.072	2.012	-1.959	0.169	3.134	9.612
Stephen Curry	-0.013	1.587	3.739	-0.241	2.848	1.754	-2.696	-0.722	2.497	8.753
Anthony Davis	1.090	0.329	-0.990	1.915	-0.395	0.980	-0.109	3.693	2.106	8.619
Chris Paul	-0.055	1.402	0.537	-0.429	3.592	3.475	-0.850	-0.942	1.151	7.882
DeAndre Jordan	4.186	-3.505	-1.118	3.858	-0.832	0.403	-0.190	4.195	-0.029	6.969
James Harden	-0.267	1.148	2.340	-0.077	1.800	1.609	-2.511	-0.419	2.979	6.602
Kevin Durant	0.644	1.345	1.273	0.647	0.989	0.603	-1.957	0.410	2.616	6.571
Kevin Love	0.072	0.756	1.825	2.511	0.584	-0.445	-0.850	-0.093	2.046	6.406
Serge Ibaka	1.684	0.075	-0.640	1.567	-0.765	-0.756	0.086	3.959	0.615	5.823
Blake Griffin	1.450	-0.791	-0.861	2.266	0.887	0.979	-1.243	0.152	2.695	5.534
Dirk Nowitzki	0.496	1.541	0.997	0.568	0.009	0.193	0.170	-0.087	1.601	5.488
Carmelo Anthony	-0.395	0.941	1.733	0.758	0.451	0.511	-1.222	-0.191	2.887	5.473
Kyle Lowry	-0.776	0.629	2.101	-0.051	2.579	1.590	-1.403	-0.589	1.368	5.448
Chris Bosh	0.687	0.502	0.133	1.405	-0.360	0.182	-0.665	1.685	1.838	5.406
John Wall	-0.586	0.525	0.813	-0.185	3.219	2.368	-2.893	0.042	1.951	5.255
DeMarcus Cousins	0.560	-0.260	-1.118	2.780	0.379	1.449	-2.416	1.426	2.250	5.049
Damian Lillard	-0.713	1.148	2.690	-0.377	1.871	0.153	-1.495	-0.596	2.246	4.926
Kyrie Irving	-0.034	1.183	1.457	-0.510	1.801	1.131	-1.218	-0.519	1.586	4.877
Kawhi Leonard	0.602	0.571	0.390	0.541	-0.295	1.745	0.539	0.296	0.261	4.650
Andre Drummond	3.126	-3.805	-1.118	3.318	-1.039	0.980	0.267	2.310	0.568	4.606
Al Jefferson	0.857	-0.399	-1.082	2.292	-0.262	0.033	-0.006	1.039	2.106	4.577
Paul Millsap	-0.013	-0.168	0.335	1.429	0.314	1.637	-1.124	0.870	1.244	4.524
Marc Gasol	0.666	0.306	-1.100	1.162	0.549	0.355	-0.196	1.931	0.502	4.174
Kyle Korver	-0.013	1.529	2.469	-0.404	0.079	0.307	0.467	-0.326	-0.007	4.101
Mike Conley	-0.352	0.744	0.813	-0.871	1.767	2.370	-0.657	-0.701	0.961	4.074
Thaddeus Young	-0.077	-0.214	0.169	0.731	0.042	3.174	-0.665	-0.096	0.880	3.945
Klay Thompson	-0.501	0.883	2.745	-0.649	-0.124	0.653	-0.383	-0.252	1.490	3.863
Trevor Ariza	-0.543	0.375	1.972	0.513	0.315	1.449	-0.196	-0.548	0.508	3.845
Nicolas Batum	-0.161	0.814	1.439	1.025	1.160	0.182	-1.305	0.282	0.342	3.778
Derrick Favors	1.472	-0.849	-1.118	1.834	-0.563	0.182	-0.012	2.310	0.501	3.757
Dwight Howard	2.532	-2.731	-1.100	2.992	-0.395	0.352	-1.861	2.440	1.480	3.710

Objective function: Maximize total "score" for roster

Constraints:

- Total cost < $50k
- Maximum roster size

A Not So Much Fun Knapsack Problem

0/1 Knapsack Problem, Formalized

- Each item is represented by a pair, *<value, weight>*
- The knapsack can accommodate items with a total weight of no more than *w*
- A vector, *L*, of length *n*, represents the set of available items. Each element of the vector is an item
- A vector, *V*, of length *n*, is used to indicate whether or not items are taken. If *V[i] = 1*, item *I[i]* is taken. If *V[i] = 0*, item *I[i]* is not taken

0/1 Knapsack Problem, Formalized

Find a V that maximizes

$$\sum_{i=0}^{n-1} V[i] * I[i].value$$

subject to the constraint that

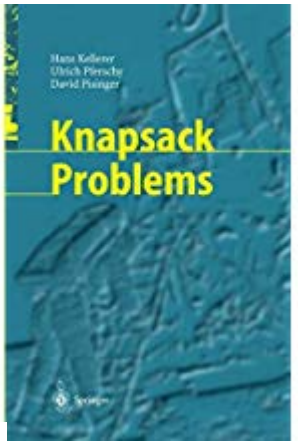
$$\sum_{i=0}^{n-1} V[i] * I[i].weight \leq w$$

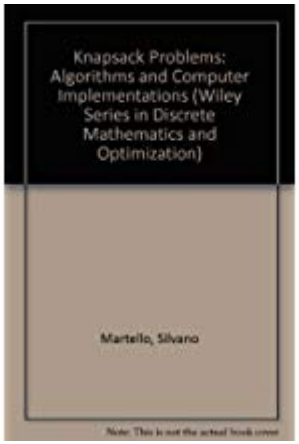
Going from an informal understanding of a problem to a rigorous problem statement is an important skill to develop.

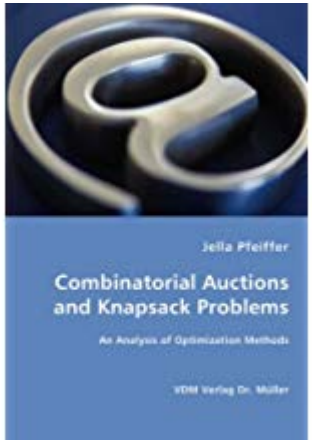
Vague PS-> Rigorous PS -> Algorithm -> Code

Many Closely Related Problems

- Multiple knapsack problem
- Integer knapsack problem
- Multiple constrain knapsack problem
- Bin-packing problem
- …







Complementary Knapsack Problem

minimizes

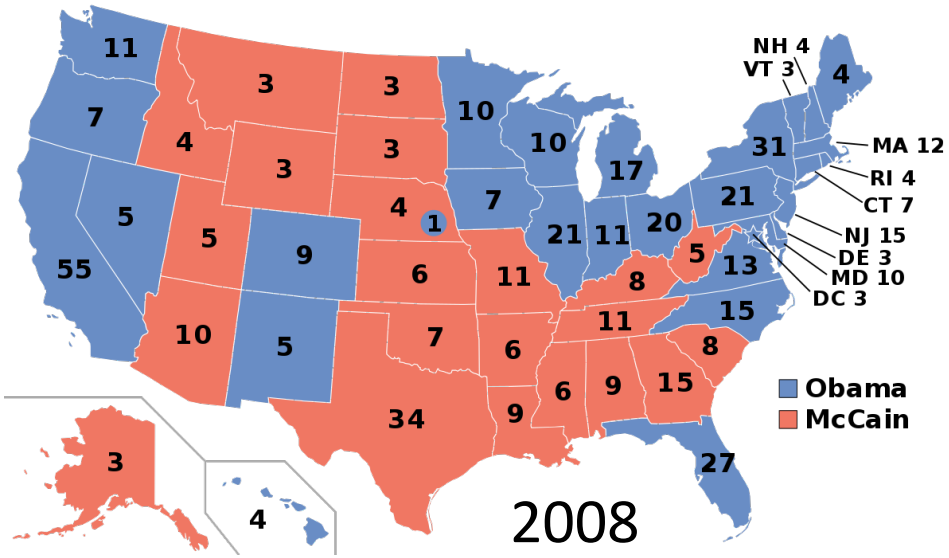
Find a V that ~~maximizes~~

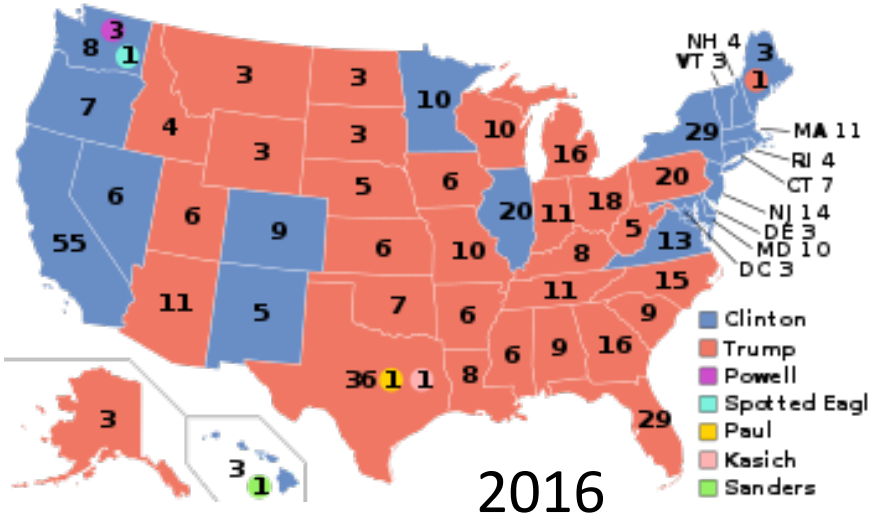
$$\sum_{i=0}^{n-1} V[i] * I[i].value$$

subject to the constraint that

$$\sum_{i=0}^{n-1} V[i] * I[i].weight \geq w$$

How Close Was the Election?





Assume two candidates, and winner-take-all.
Assume that candidate Winner won the election.
What is the smallest number of voters for candidate Loser that could have changed the outcome by moving to a different state?

Complementary Knapsack Problem

minimizes

Find a V that ~~maximizes~~

$$\sum_{i=0}^{n-1} V[i] * I[i].value$$

i is a state won by Winner

I[i].value is the total number of additional votes needed to flip state

subject to the constraint that

$$\sum_{i=0}^{n-1} V[i] * I[i].weight \geq w$$

i is a state won by Winner

I[i].weight is the number of electoral college votes of i

w the number of electoral college votes needed to win

Reduction to Knapsack Problem

- Solve 0/1 knapsack problem with
  - Same set of items
  - W = total # of electoral votes – # of votes needed to win
- The states **not** selected are the ones to which voters should move

Reduction of a new problem to a problem with a well-known solution is an important problem-solving technique

Solving 0/1 Knapsack Problem: Brute Force

- 1. Enumerate all possible combinations of items. That is to say, generate all subsets of the set of items. This is called the power set (see 6.0001 lecture 10).
- 2. Remove all of the combinations whose total units exceeds the allowed weight.
- 3. From the remaining combinations choose any one whose value is the largest.

Often Not Practical

- How big is power set when the set is of size 100?
  - 1,267,650,600,228 (~1.25 trillion)?
  - 1,267,650,600,228,229,401 (~1.25 quintillion)?
  - 1,267,650,600,228,229,401,496,703 (~1.25 septillion)?
  - 1,267,650,600,228,229,401,496,703,205,376
  - Bonus question: What is the word for a number of this magnitude?
    - Nonillion
    - Decillion
    - Monillion
    - Quintillion
    - Trumpillion
    - U.S. Budget deficit

Why is the Powerset So Big?

- Recall
  - A vector, *V*, of length *n*, is used to indicate whether or not items are taken. If *V[i] = 1*, item *I[i]* is taken. If *V[i] = 0*, item *I[i]* is not taken
- How many possible different values can *V* have?
  - As many different binary numbers as can be represented in *n* bits
- For example, if there are 100 items to choose from, the power set is of size 2^{100}
  - 1,267,650,600,228,229,401,496,703,205,376

Are We Just Being Stupid?

- Alas, no
- 0/1 knapsack problem is inherently exponential
- But don't despair

Greedy Algorithm a Practical Alternative

```
while knapsack not full
      put "best" available item in knapsack
```

- But what does best mean?
  - Most valuable
  - Least expensive
  - Highest value/units

An Example

- You are about to sit down to a meal
- You know how much you value different foods, e.g., you like donuts more than apples
- But you have a calorie budget, e.g., you don't want to consume more than 750 calories
- Choosing what to eat is a knapsack problem

A Menu

Food	wine	beer	pizza	burger	fries	coke	apple	donut
Value	89	90	30	50	90	79	90	10
calories	123	154	258	354	365	150	95	195

- Let's look at a program that we can use to decide what to order

Class Food

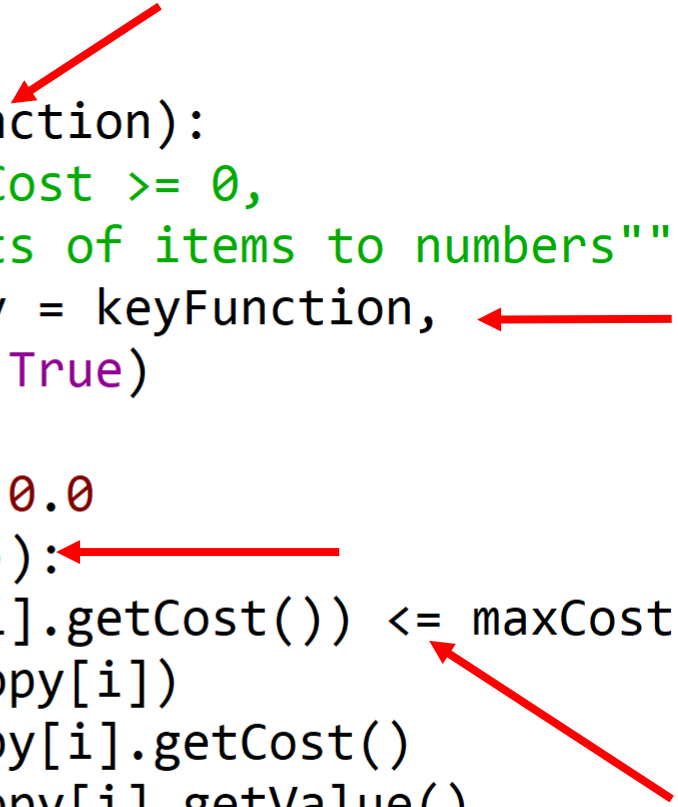
```
class Food(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = v
        self.calories = w
    def getValue(self):
        return self.value
    def getCost(self):
        return self.calories
    def density(self):
        return self.getValue()/self.getCost()
    def __str__(self):
        return self.name + ': <' + str(self.value)\
               + ', ' + str(self.calories) + '>'
```

Build Menu of Foods

```
def buildMenu(names, values, calories):
    """names, values, calories lists of same length.
       name a list of strings
       values and calories lists of numbers
       returns list of Foods"""
    menu = []
    for i in range(len(values)):
        menu.append(Food(names[i], values[i],
                         calories[i]))
    return menu
```

Implementation of Flexible Greedy

```
def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,
         keyFunction maps elements of items to numbers"""
    itemsCopy = sorted(items, key = keyFunction,
                       reverse = True)
    result = []
    totalValue, totalCost = 0.0, 0.0
    for i in range(len(itemsCopy)):
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:
            result.append(itemsCopy[i])
            totalCost += itemsCopy[i].getCost()
            totalValue += itemsCopy[i].getValue()
    return (result, totalValue)
```



Why use sorted rather than sort?

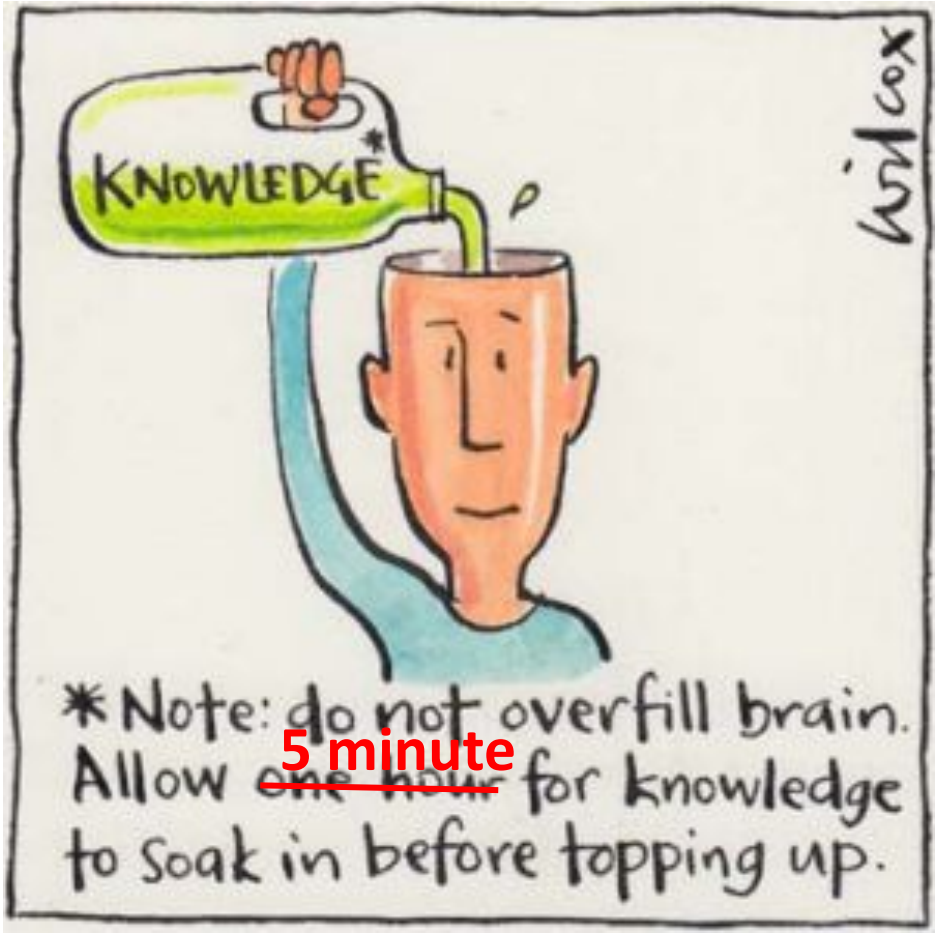
How does complexity grow relative to len(items)?

Algorithmic Efficiency

O(n)
O(n log n)
O(n**2)

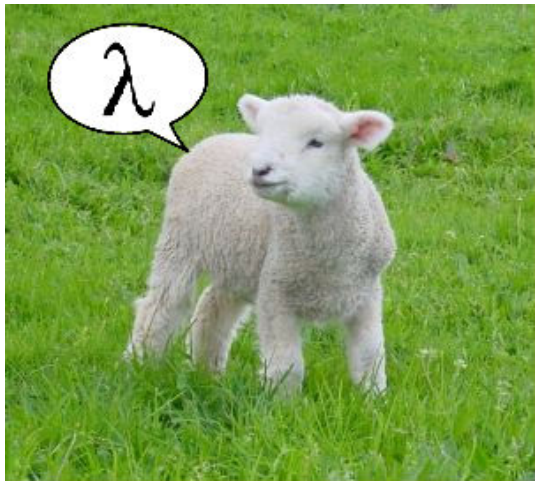
```
def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,
         keyFunction maps elements of items to numbers"""
    itemsCopy = sorted(items, key = keyFunction,
                       reverse = True)
    result = []
    totalValue, totalCost = 0.0, 0.0
    for i in range(len(itemsCopy)):
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:
            result.append(itemsCopy[i])
            totalCost += itemsCopy[i].getCost()
            totalValue += itemsCopy[i].getValue()
    return (result, totalValue)
```

Five Minute Break



Using greedy

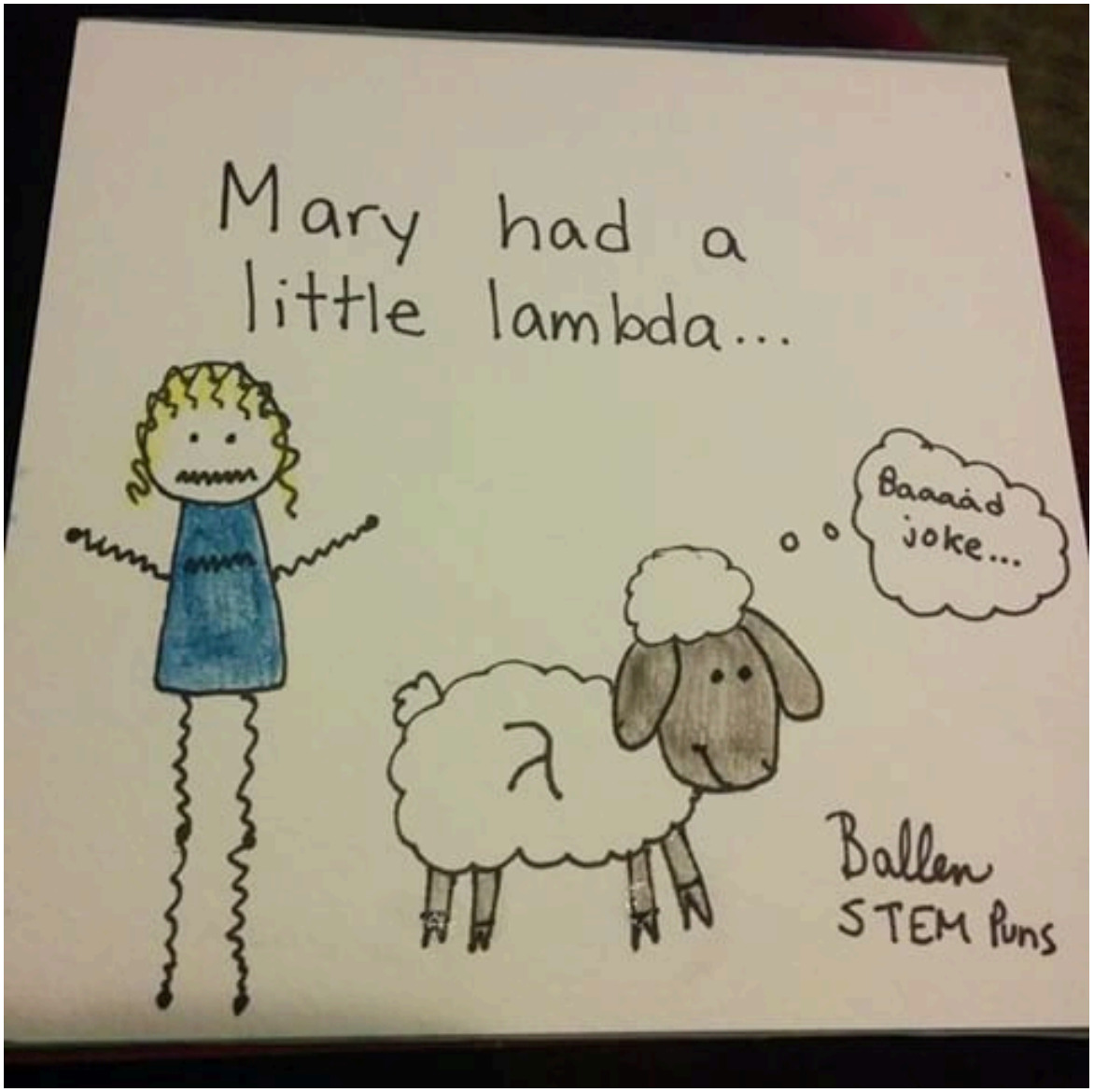
```
def testGreedy(items, constraint, metric):
    metrics = {'value': Food.getValue,'density':Food.density,
               'cost': lambda x: 1/Food.getCost(x)}
    try:
        taken, val = greedy(items, constraint, metrics[metric])
    except:
        print('Unknown metric', metric)
        return
    print('Total value of items taken =', val)
    for item in taken:
        print('   ', item)
```



lambda

- lambda used to create anonymous functions
  - `lambda` <id_1, id_2, … id_n>: <expression>
  - Returns a function of n arguments
- Can be very handy, as here
- Possible to write amazing complicated lambda expressions
- Don't—use `def` instead

In Honor of Prof. Grimson



Testing Different Definitions of "Best"

```
def testGreedys(foods, maxUnits):
    metric = input('Chose a metric (cost, value, or density): ')
    print('Use greedy by', metric, 'to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits, metric)

names = ['wine', 'beer', 'pizza', 'burger', 'fries',
         'cola', 'apple', 'donut', 'cake']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
foods = buildMenu(names, values, calories)
testGreedys(foods, 1000)
```

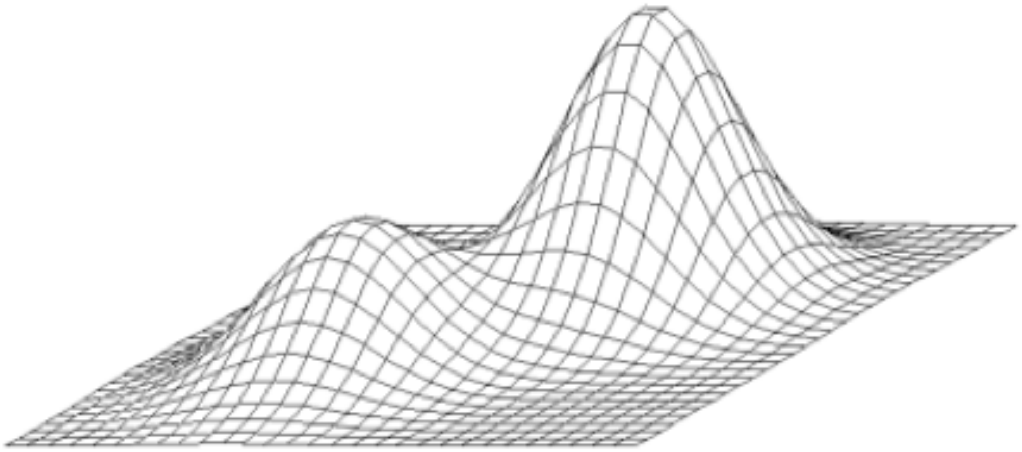
Running the Tests

```
names = ['wine', 'beer', 'pizza', 'burger', 'fries',
         'cola', 'apple', 'donut', 'cake']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
foods = buildMenu(names, values, calories)
testGreedys(foods, 750)
```

Run code

Why Different Answers?

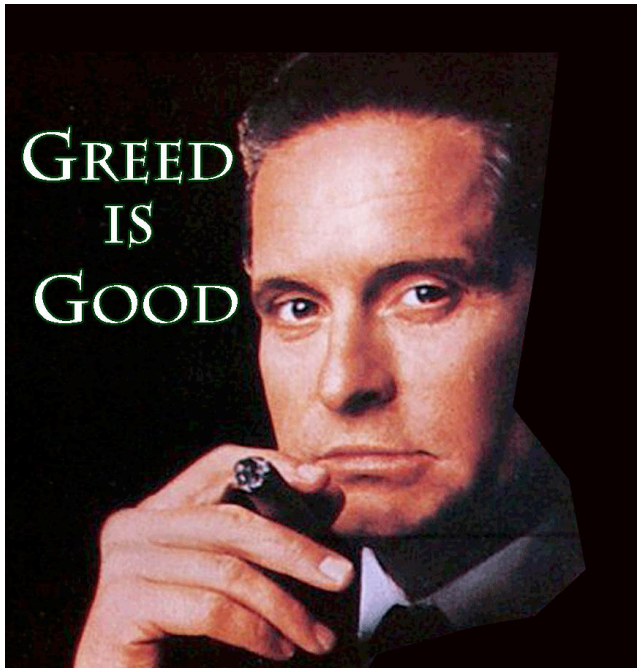
- Sequence of locally "optimal" choices don't always yield a globally optimal solution



- Is greedy by density always a winner?
  - Try testGreedys(foods, 1000)

The Pros Greedy

- Easy to implement
- Computationally efficient



Main Street?
Wall Street?
Vassar Street?
The Art of the Deal?

The Con of Greedy

- Does not always yield the best solution
  - Don't even know how good the approximation is
- Suppose we want to find a truly optimal solution?

Brute Force Algorithm

- 1. Enumerate all possible combinations of items.
- 2. Remove all of the combinations whose total units exceeds the allowed weight.
- 3. From the remaining combinations choose any one whose value is the largest.

Use a Search Tree to Do This

How Computer Scientists Draw Trees

Search Tree Implementation

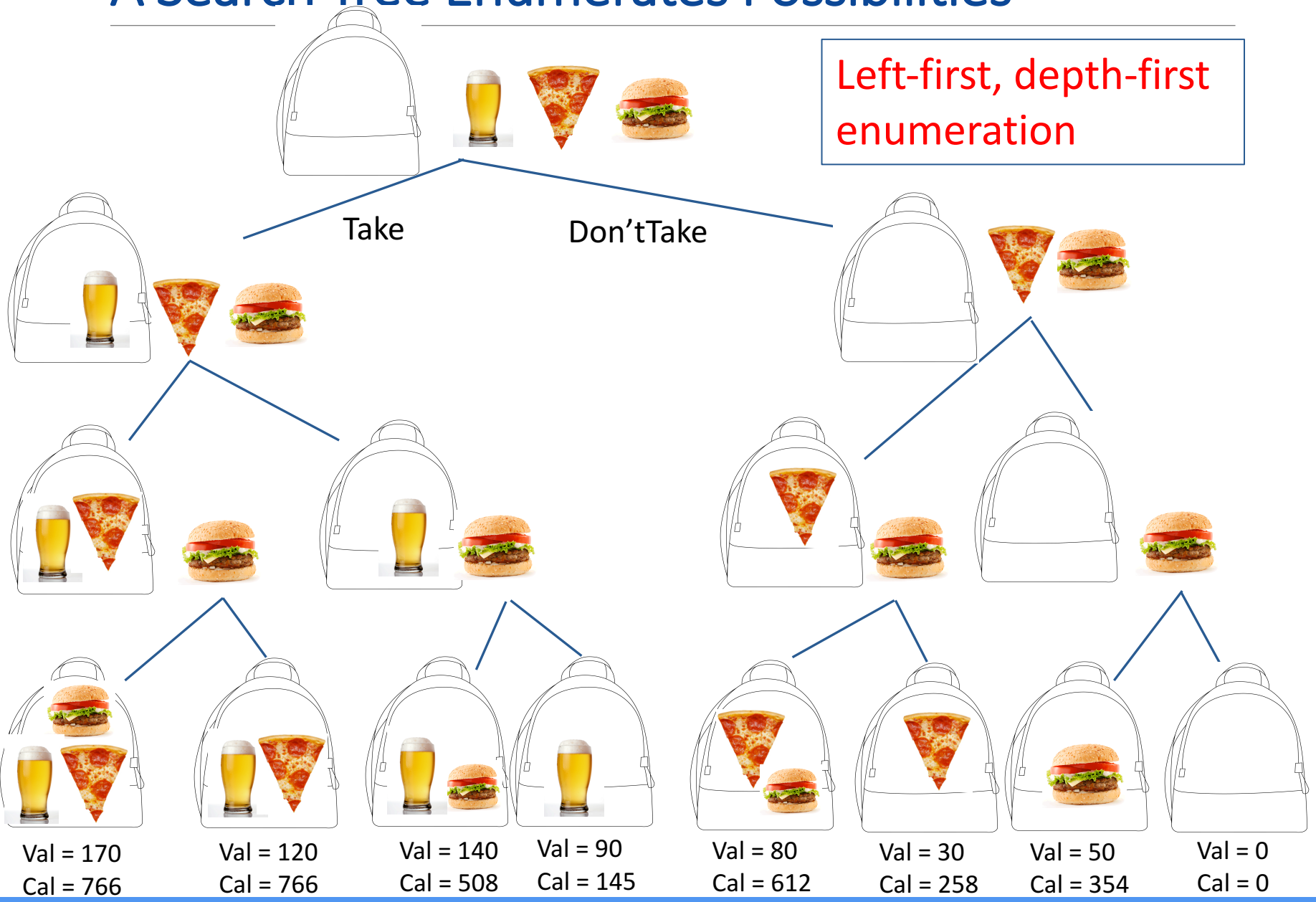
- The tree is built top down starting with the root
- The first element is selected from the still to be considered items
  - If there is room for that item in the knapsack, a node is constructed that reflects the consequence of choosing to take that item. By convention, we draw that as the left child
  - We also explore the consequences of not taking that item. This is the right child
- The process is then applied recursively to non-leaf children
- Once tree generated, chose a node with the highest value that meets constraints

Illustrative Example

- With calorie budget of 750 calories, chose an optimal set of foods from the menu

Food	beer	pizza	burger
Value	90	30	50
calories	154	258	354

A Search Tree Enumerates Possibilities



Computational Complexity

- Time based on number of nodes generated
- Number of levels is number of items to choose from
- Number of nodes at level *i* is 2^i
- So, if there are *n* items the number of nodes is
  - $\sum_{i=0}^{i=n} 2^i$
  - I.e., O(2^{n+1})
- An obvious optimization: don't explore parts of tree that violate constraint (e.g., too many calories)
  - Doesn't change complexity
- Does this mean that brute force is not a useful approach?

We'll Find Out on Wednesday