

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 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://sicp-s1.mit.edu/fall19>
- Post privately on Piazza rather than emailing staff
- Course uses Python 3
 - 3.6 or 3.7 fine

Course Policies

■ Collaboration

- Okay: Helping others debug, discussing general attack on problem
- NOT okay:
 - Looking at code (from others in class or previous years)
 - **Allowing others to see your code**
 - Side-by-side coding
- Provide names of all “collaborators”
- We run code similarity program on all psets

■ Extensions on problem sets

- Will consider requests that come with support from S³
- Late days, 3 to use at your discretion

Grading: Problem Sets and Finger Exercises

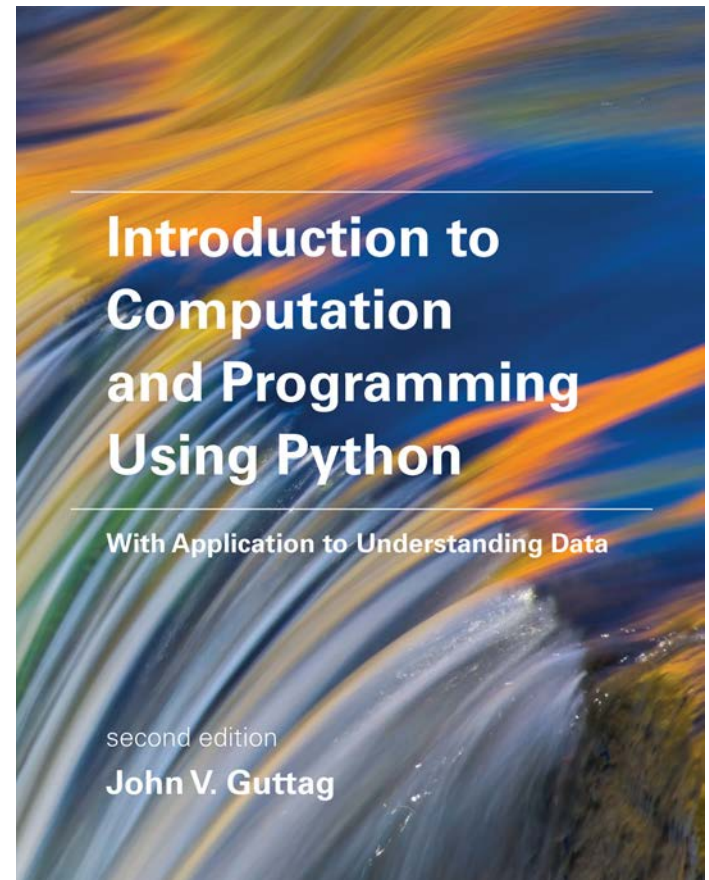
- Problem sets
 - 35% of final grade
 - Notice that due dates are not on consistent days of week
- Finger exercises on MITx
 - Mandatory exercises 10% of final grade
 - One per week
 - No extensions on due dates

Quizzes and Exams

- Microquizzes during scheduled lecture hours
 - At end of some lectures (see calendar)
 - Must have computer with wireless connection
 - 3 thirty minute quizzes (worth 30%, best 2 out of 3)
 - 1 forty-five minute quiz (worth 25%)
 - No makeups except under extraordinary circumstance
- Quizzes will cover material from lectures, problems sets, and assigned readings

Relevant Reading

- Section 12.1
- Section 5.4 (lambda functions)



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 quizzes
- Lectures will be faster paced
- Less about learning to program
- More about dipping your toe into computational modeling



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?

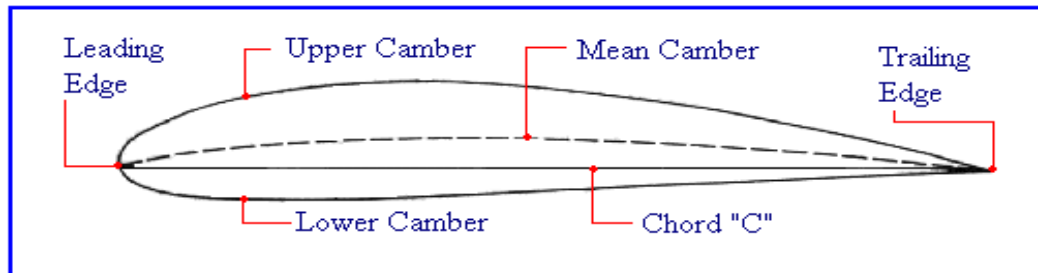
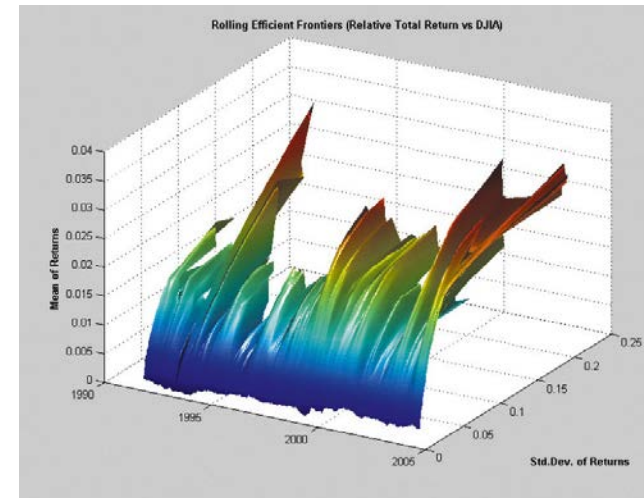
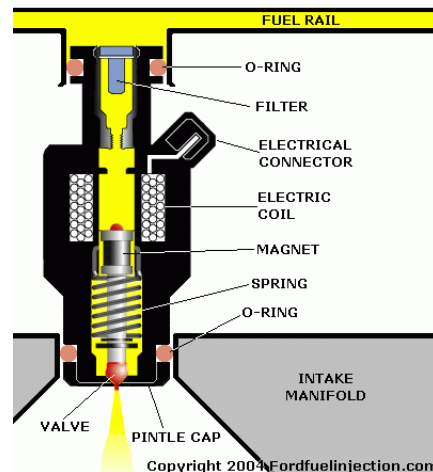
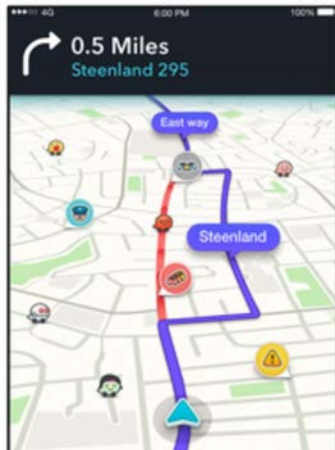
- An **objective function** that is to be maximized or minimized, e.g.,
 - Minimize money spent traveling from Boston to NYC



- 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



Pos	Player	Points	Salary
QB	Patrick Mahomes @ JAX (9)	22.7	\$ 7200
RB	Leonard Fournette vs KC (31)	19.7	\$ 6100
RB	Dalvin Cook vs ATL (28)	18.8	\$ 6000
WR	Curtis Samuel vs LAR (22)	12.2	\$ 4200
WR	Tyler Lockett vs CIN (10)	15.6	\$ 6000
WR	Marvin Jones @ ARI (13)	13.8	\$ 4800
TE	Zach Ertz vs WAS (4)	16.9	\$ 6100
Flex	Kerryon Johnson @ ARI (27)	18.1	\$ 5800
DEF	Baltimore Ravens @ MIA (24)	10.4	\$ 3800
Total		148.2	\$50000

Objective function: Maximize total “score” for roster

Constraints:

Total cost \leq \$50k

1 QB, 2 RB, 3 WR, 1 TE, 1 Flex, 1 Team Defense

A Not-So-Much-Fun Knapsack Problem



0/1 Knapsack Problem, Formalized

- Each item is represented by a pair, $\langle \text{value}, \text{weight} \rangle$
- The knapsack can accommodate items with a total weight of no more than w
- A vector, I , 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

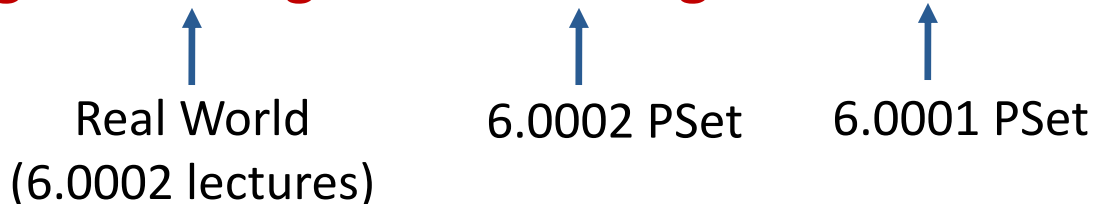
$$\sum_{i=0}^{n-1} V[i] * I[i].value$$

subject to the constraint that

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

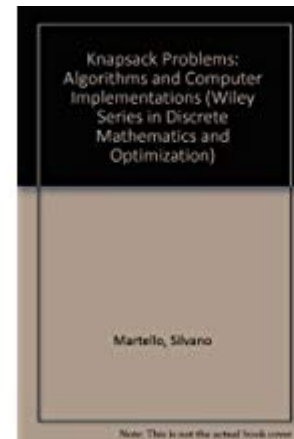
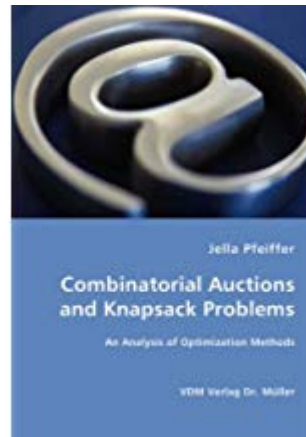
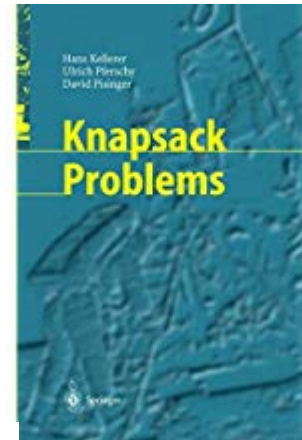
Going from an informal understanding of a problem to a rigorous problem statement is an important skill to develop.

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



Many Closely Related Problems

- Multiple knapsack problem
- Integer knapsack problem
- Multiple constraint knapsack problem
- Bin-packing problem
- ...



Complementary Knapsack Problem

minimizes

Find a V that ~~maximizes~~

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

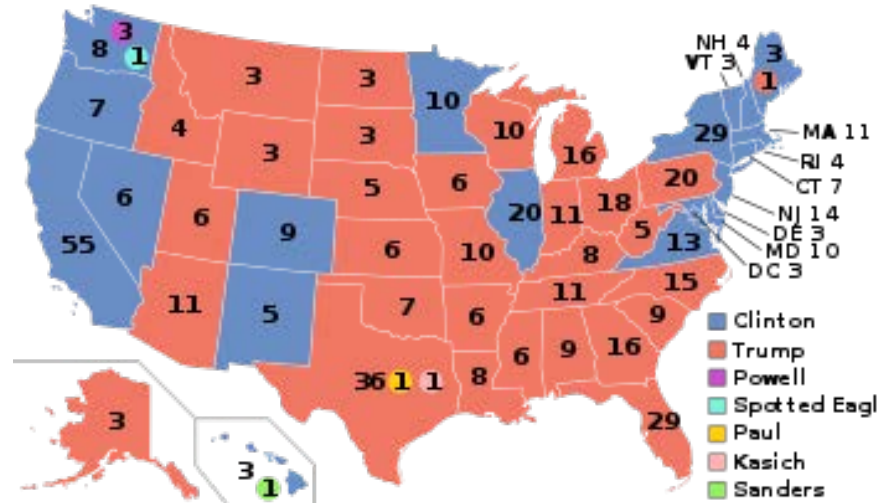
subject to the constraint that

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

Why might this be interesting?

How Close Was a Presidential Election?

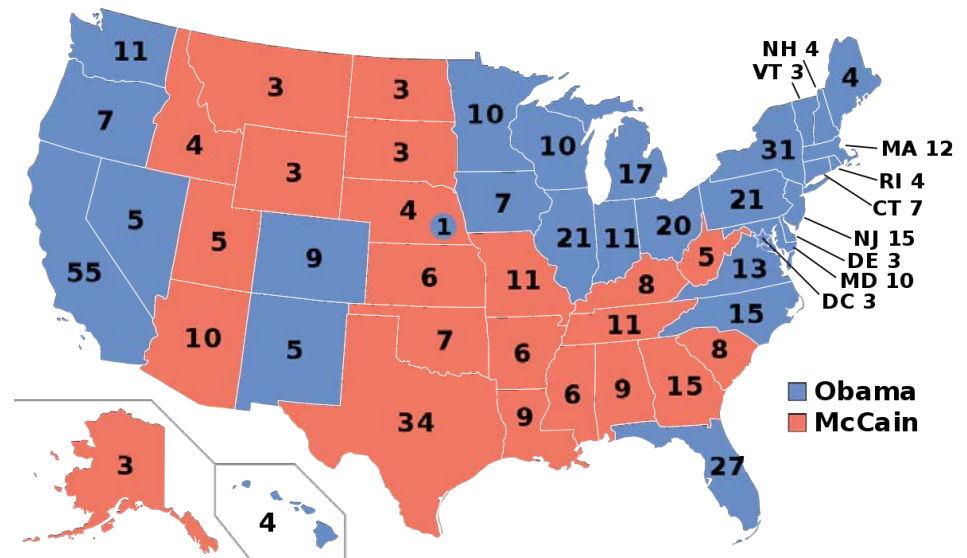
- 2016
 - Popular vote
 - Trump: 62,984,828
 - Clinton: 66,853,514
 - Electoral college
 - Trump: 304
 - Clinton: 227



Could outcome have been changed if, prior to election, some Clinton supporters had moved to a different state?

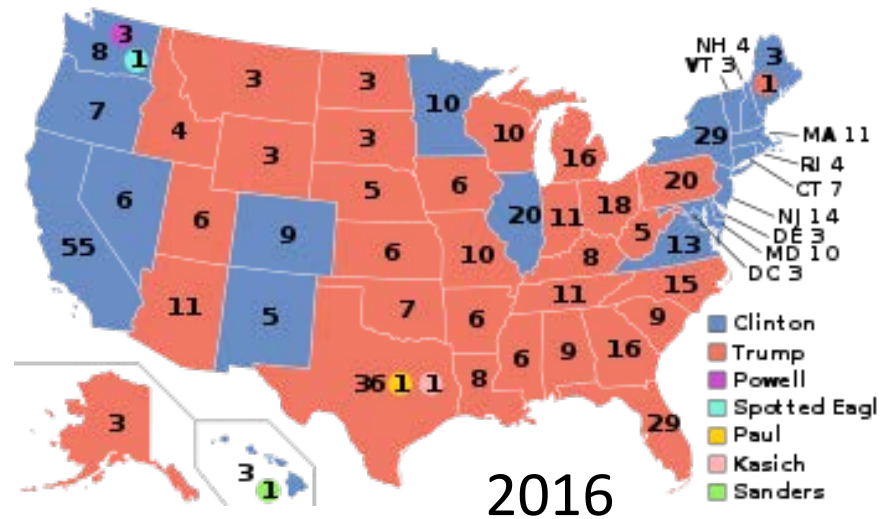
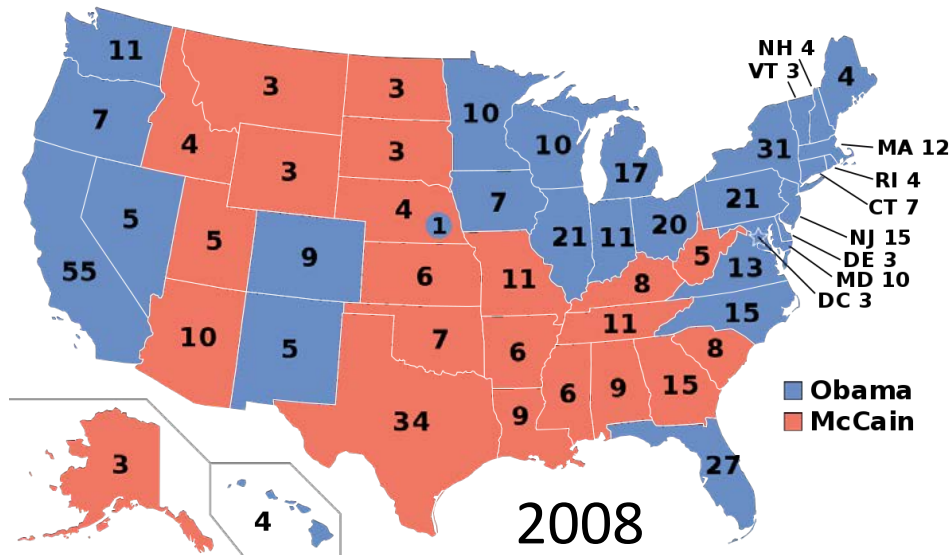
How Close Was a Presidential Election?

- 2008
 - Popular vote
 - Obama: 69,498,516
 - McCain: 59,948,323
 - Electoral college
 - Obama: 365
 - McCain: 173



Could outcome have been changed if, prior to election, some McCain supporters had moved to a different state?

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

Objective function to minimize: Number of votes moved from Winner to Loser

Constraint: Loser has at least the number of electoral votes needed to win

More Formally

minimizes

Find a V that ~~maximizes~~

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

$V[i] = 1$ for states won by Winner
 $I[i].value$ is the total number of additional votes needed to flip state to Loser

subject to the constraint that

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

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

w the number of additional 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

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

<https://stackoverflow.com/questions/7949705/variation-on-knapsack-minimum-total-value-exceeding-w/7950524#7950524>

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.

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 Often 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

				
McDonald's	BURGER KING	PIZZA HUT	KFC	HARVESTERS
Nuggets	Double Whopper & cheese	BBQ Americano (14 inch)	Crispy Twister	Harvester & cheese
	963 calories	2,848 calories	510 calories	970 calories
	Whopper & cheese	Double Pepperoni (13 inch)	Fillet Burger	Scampi &
	721 calories	2,392 calories	441 calories	950 calories
	Chicken Royale	Margherita (13 inch)	Popcorn chicken (kids portion)	BBQ Chick
	606 calories	2,256 calories	260 calories	950 calories
	Regular fries	Double Pepperoni (9 inch)	Regular fries	Mixed grill
	300 calories	1,200 calories	260 calories	910 calories
	Hamburger	Margherita (9 inch)	Original recipe one chicken piece	Salmon fil
	275 calories	1,128 calories	258 calories	380 calories

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

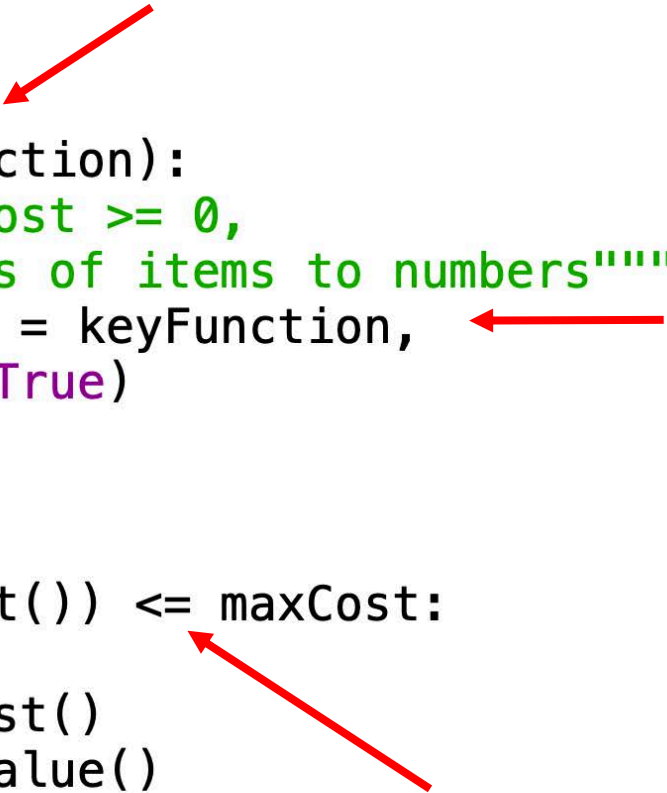
Class Food

```
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  
    for it in itemsCopy:  
        if (totalCost + it.getCost()) <= maxCost:  
            result.append(it)  
            totalCost += it.getCost()  
            totalValue += it.getValue()  
    return (result, totalValue)
```

Why use sorted rather than sort?

How does complexity grow relative to len(items)?

Algorithmic Efficiency

$O(n)$

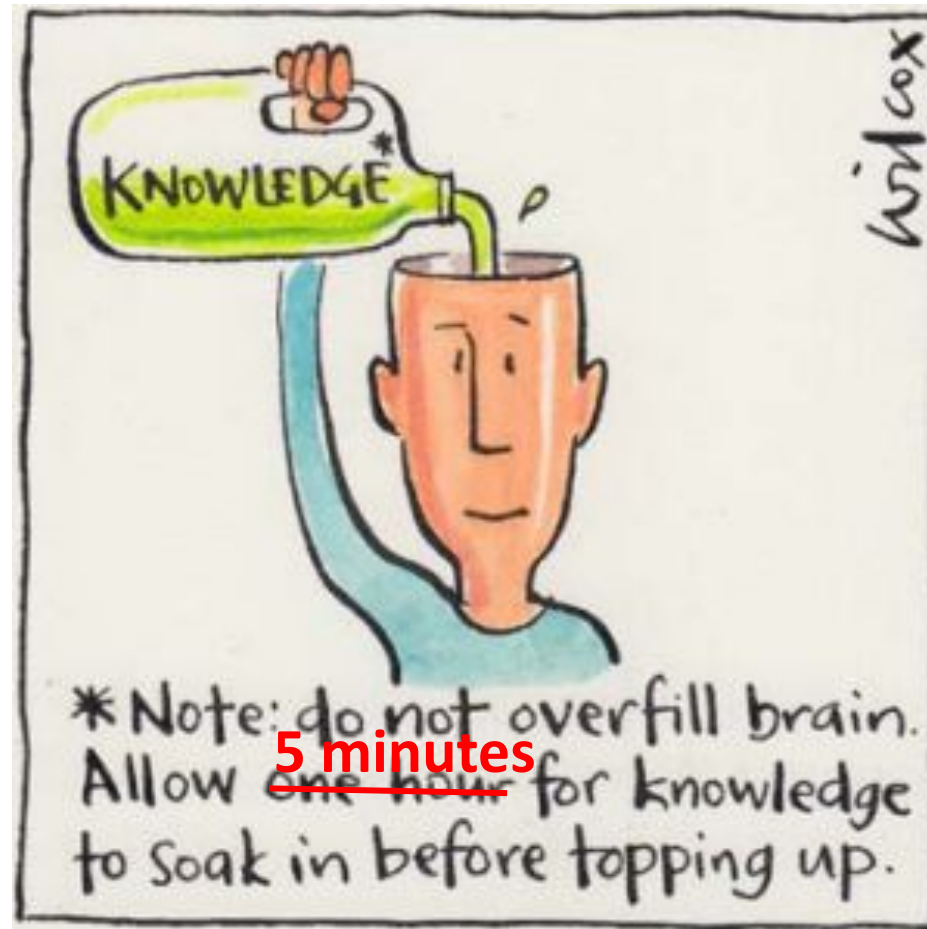
$O(n \log n)$

$O(n^2)$

```
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)

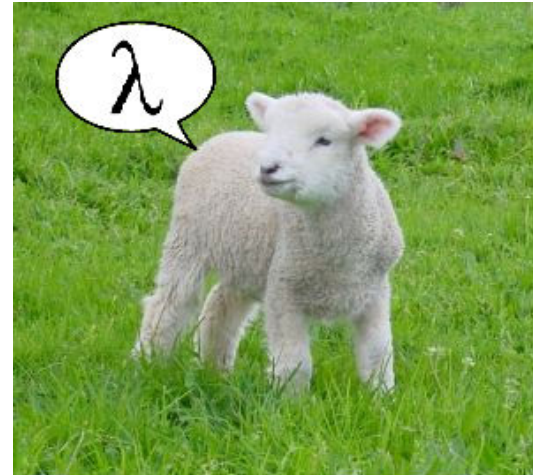
    result = []
    totalValue, totalCost = 0.0, 0.0
    for i in range(len(itemsCopy)): ←
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:
            result.append(itemsCopy[i])
            totalCost += itemsCopy[i].getCost()
            totalValue += itemsCopy[i].getValue()
    return (result, 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 <id1, id2, ... idn>: <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

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)
```


Testing Different Definitions of “Best”

```
def testGreedyys(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)  
testGreedyys(foods, 750)
```

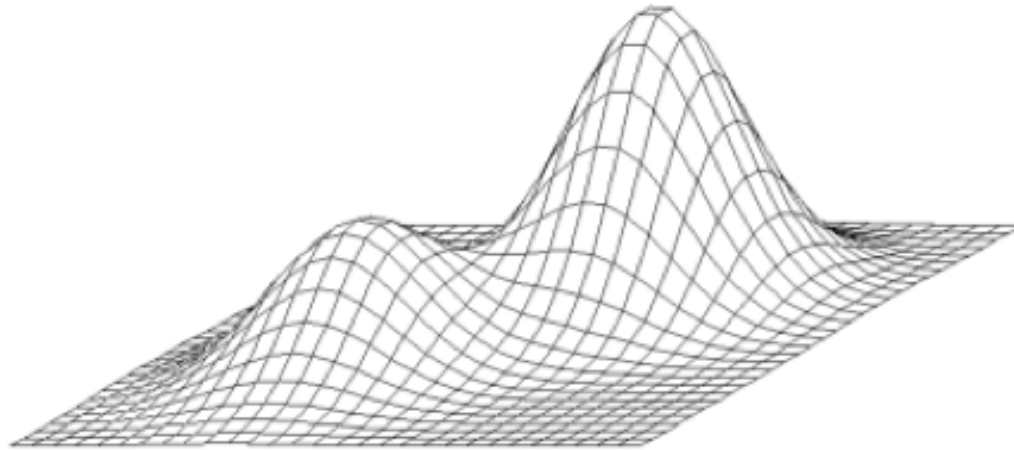
Running the Tests

```
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)  
testGreedy(foods, 750)
```

[Run code](#)

Why Different Answers?

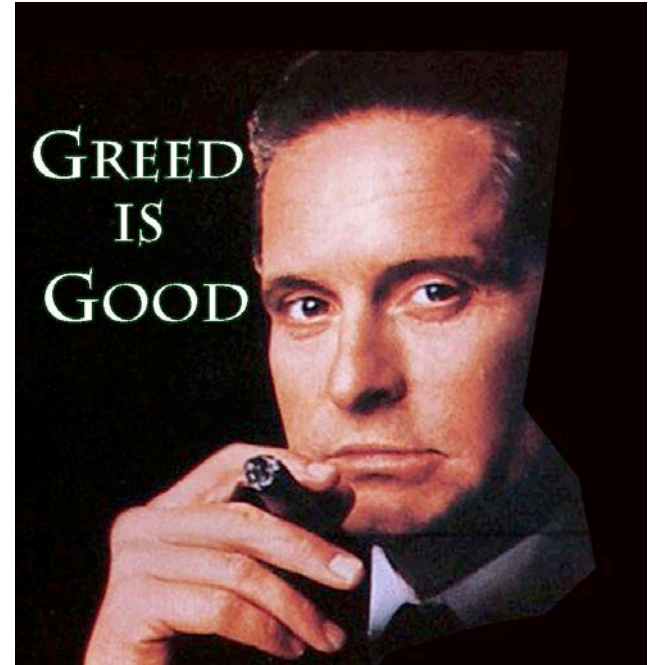
- Sequence of locally “optimal” choices don’t always yield a globally optimal solution



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

The Pros Greedy

- Easy to implement
- Computationally efficient



Circle all that apply
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



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

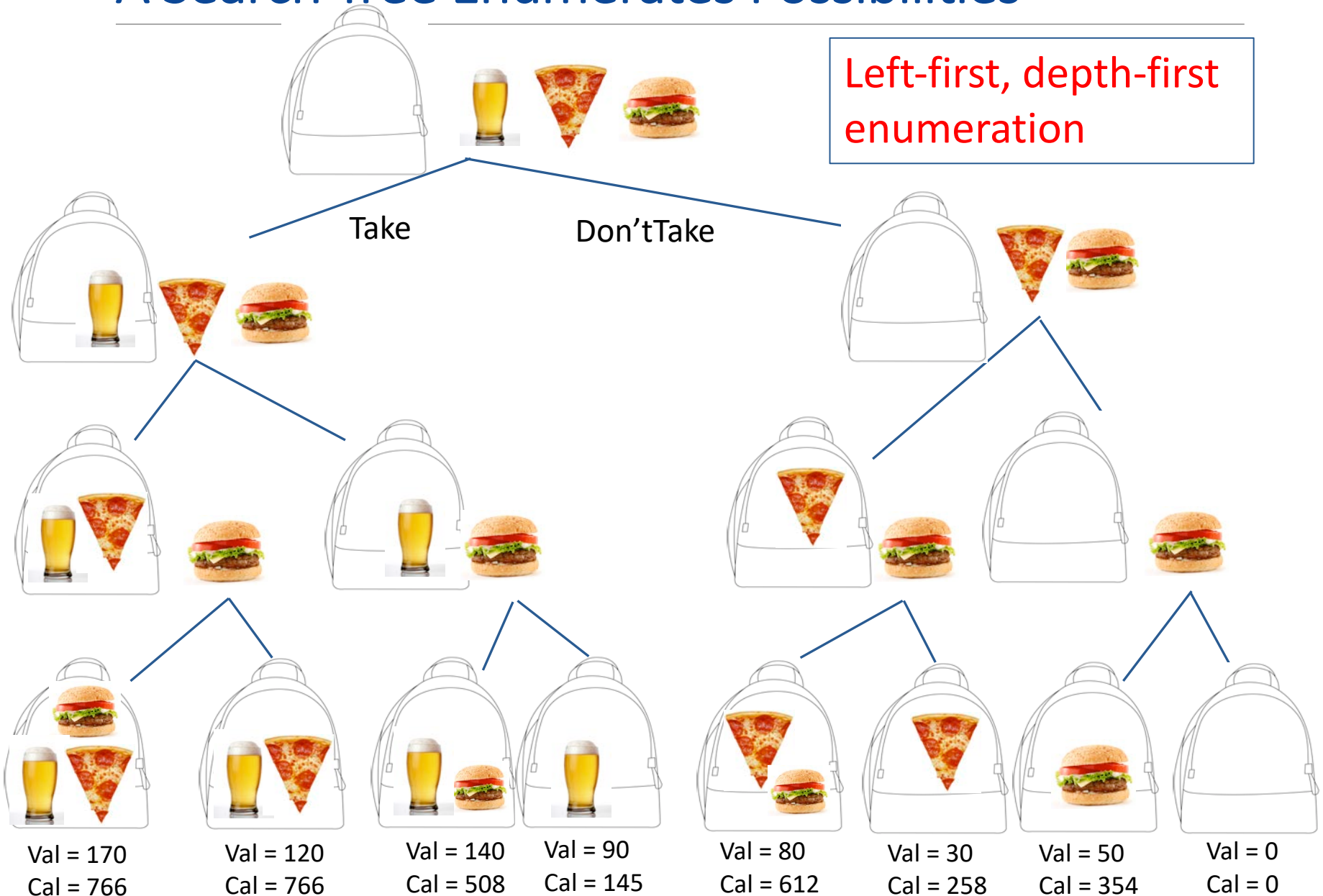
Illustrative Example

- 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

Left-first, depth-first enumeration



```

def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,
        avail a weight
        Returns a tuple of the total value of a solution
        to the 0/1 knapsack problem and the items of
        that solution"""
    if toConsider == [] or avail == 0:
        result = (0, ()) #0 value, nothing taken
    elif toConsider[0].getCost() > avail: #cannot afford current item
        #Explore right branch only
        result = maxVal(toConsider[1:], avail)
    else:

```

```

def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,
        avail a weight
        Returns a tuple of the total value of a solution
        to the 0/1 knapsack problem and the items of
        that solution"""
    if toConsider == [] or avail == 0:
        result = (0, ()) #0 value, nothing taken
    elif toConsider[0].getCost() > avail: #cannot afford current item
        #Explore right branch only
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getCost())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)

```



```

def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,
        avail a weight
        Returns a tuple of the total value of a solution
        to the 0/1 knapsack problem and the items of
        that solution"""
    if toConsider == [] or avail == 0:
        result = (0, ()) #0 value, nothing taken
    elif toConsider[0].getCost() > avail: #cannot afford current item
        #Explore right branch only
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake = maxVal(toConsider[1:],
                                    avail - nextItem.getCost())
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result

```

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

Ramification of This

```
timePerNode = 10**-9 #1 nano second
for numNodes in range(0, 101, 10):
    seconds = (2**(numNodes+1))*timePerNode
    years = seconds/(60*60*24*365)
    print('Time for', numNodes, 'nodes =', \
          round(years, 4), 'years')
```

Does this mean that we can't find an optimal solution to real problems?

We'll Find Out on Wednesday

