Introduction and Optimization Problems

John Guttag

MIT Department of Electrical Engineering and Computer Science
6.0002 Prerequisites

- Experience writing object-oriented programs in Python
  - Preferably in Python 3.5
- Familiarity with concepts of computational complexity
- Familiarity with some simple algorithms
- 6.0001 sufficient

Question 1
Some Administrative Things

- **Problem sets**
  - Programming problems designed to
    - Improve your programming skills
    - Help you learn the conceptual material

- **Finger exercises**
  - Very small programming problems designed to help you learn a single programming concept

- **Reading assignments in textbook**
  - Another take on and more details about material covered by lectures and problem sets

- **Exam**: based on above
How Does It Compare to 6.0001?

- Programming assignments a bit easier
  - Focus more on the problem to be solved than on programming
- Lecture content more abstract
- Lectures will be a bit faster paced
- Less about learning to program, more about dipping your toe into data science
Honoring Your Programming Skills

- 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
- Statistical models
- Simulation models
Relevant Reading for Today’s Lecture

- Section 12.1
- Section 5.4 (lambda functions)
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
- Statistical models
- Simulation models
What Is an Optimization Model?

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

- A set of constraints (possibly empty) that must be honored, e.g.,
  - Cannot spend more than $100
  - Must be in Boston before 5:00PM

Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use.
Knapsack Problems
Knapsack Problem

- 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
  - 0/1 knapsack problem
  - Continuous or fractional knapsack problem
My Least-favorite Knapsack Problem

Images © sources unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use.
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 \(l[i]\) is taken. If \(V[i] = 0\), item \(l[i]\) is not taken
0/1 Knapsack Problem, Formlized

Find a \( V \) that maximizes

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

subject to the constraint that

\[
\sum_{i=0}^{n-1} V[i] \cdot I[i].weight \leq w
\]
Brute Force Algorithm

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.

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?
- 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?
  - 1,267,650,600,228,229,401,496,703,205,376

**Question 2**
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**

<table>
<thead>
<tr>
<th>Food</th>
<th>wine</th>
<th>beer</th>
<th>pizza</th>
<th>burger</th>
<th>fries</th>
<th>coke</th>
<th>apple</th>
<th>donut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>89</td>
<td>90</td>
<td>30</td>
<td>50</td>
<td>90</td>
<td>79</td>
<td>90</td>
<td>10</td>
</tr>
<tr>
<td>calories</td>
<td>123</td>
<td>154</td>
<td>258</td>
<td>354</td>
<td>365</td>
<td>150</td>
<td>95</td>
<td>195</td>
</tr>
</tbody>
</table>

- Let’s look at a program that we can use to decide what to order
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) + '>'
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
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)
def greedy(items, maxCost, keyFunction):
    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)
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:
        print('   ', item)
def testGreedys(maxUnits):
    print('Use greedy by value to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('
Use greedy by cost to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
    print('
Use greedy by density to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, Food.density)

testGreedys(800)
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
Using greedy

def testGreedys(foods, maxUnits):
    print('Use greedy by value to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('
Use greedy by cost to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
    print('
Use greedy by density to allocate', maxUnits, 'calories')
    testGreedy(foods, maxUnits, Food.density)

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)
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 and Cons of Greedy

- Easy to implement
- Computationally efficient

- But does not always yield the best solution
  - Don’t even know how good the approximation is

- In the next lecture we’ll look at finding truly optimal solutions