1. Greedy Algorithms

Thursday, April 2, 2020 10:56 AM

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0002-introduction-to-computational-thinking-and-data-science-fall-2016/lecture-videos/lecture-1-introduction-and-optimization-problems/

Computational Modelling

Science is moving out of the wet lab into computation. Models are tools used to describe the present state or predict the future state.

3 Types of Models
Optimization Model
Simulation Model
Statistical Model

Optimization Model

Start with an objective function to be maximized or minimized within a certain constraint.

- Fastest way to get to Boston (travel time is the objective function) under the constraint of $100 budget

Knapsack Problem
Optimize the space in a knapsack if you want to steal something

- Continuous knapsack problem: you can take fractions of something.
  - This is easy and can be solved by a Greedy Algorithm (take as much of the most valuable item until it's out and move to the second most important item)
- 0 to 1 knapsack problem: much more complex because once you take one item it constrains your other options
  - Taking food items, and it can't be greater than 1500 calories. Taking an item of 1300 calories will mean you have no other options left.

0 to 1 Knapsack Set up

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

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

<u>Solutions</u>

1. Generating all possibilities and combination and finding the largest V value that obeys the constraint, but it's not practical.
   - Sadly this is the *best solution* to this problem, it gets exponentially harder to calculate with each additional items, as with most optimization problem

2. Greedy Algorithm
   - Judging the "best" option (by most valuable, or least expensive, or highest value/units ratio) and choosing it until the knapsack is full
   - i.e. Building a food menu that will fit under the calorie restriction of 750 calories

Implementation of the Greedy algorithm

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) + '>'
```

```
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
```

Defining Greedy

```
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)
```

#Using Greedy

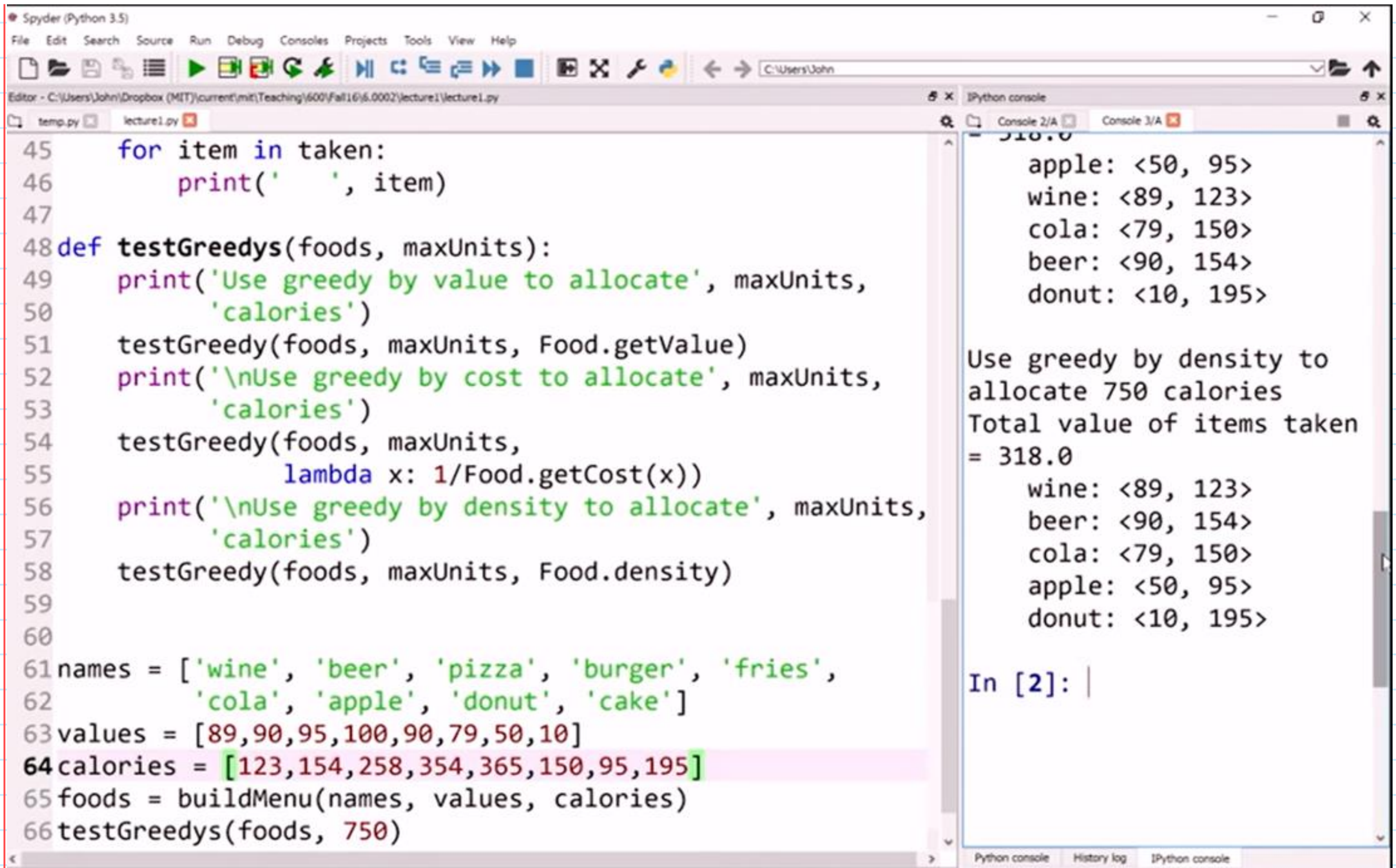
```
def testGreedys(maxUnits):
    print('Use greedy by value to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('\nUse greedy by cost to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits,
               lambda x: 1/Food.getCost(x))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

testGreedys(800)                       ?
```

<u>Lamda:</u> Lambda is used to create an anonymous function, anonymous in the sense that it has no name. So you start with the keyword lambda. You then give it a sequence of identifiers and then some expression

- What lambda does is it builds a function that evaluates that expression on those parameters and returns the result of evaluating the expression

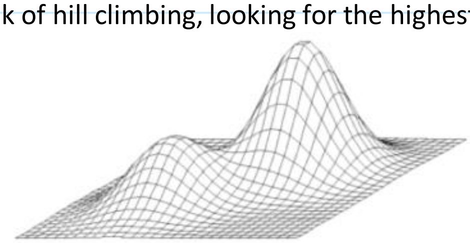
#Executing the Code



Note: The two solutions are the same set in a different order

Greedy Algorithm Limitations

The problem is that a greedy algorithm makes a sequence of local optimizations, chooses the locally optimal answer at every point, and that doesn't necessarily add up to a globally optimal answer.

- i.e. think of hill climbing, looking for the highest summit
    - 
    - A greedy algorithm says go up. If you can't go up anymore, stop. You'll reach a locally optimal decision, but you may not reach a globally optimal solution (because this mountain climbing greedy algorithm says never go backwards)

Other times, you have to redefine best (calories vs. density). Sometimes you can't define a "best" to reach an optimal solution