

# Optimization Problems, Lecture 1, Segment 2

---

John Guttag

MIT Department of Electrical Engineering and  
Computer Science

# 0/1 Knapsack Inherently Exponential

---

**Give up**

**Approximate solution**

**Exact solution that is often fast**

# 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 800 calories
- Choosing what to eat is a knapsack problem



CC-BY Jaqeli

# 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

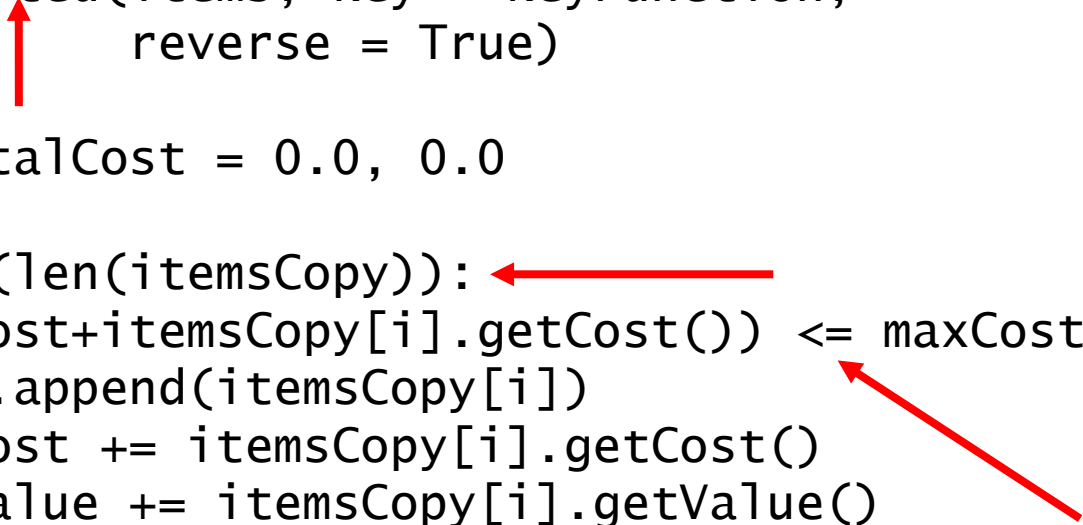
---

```
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)
```

Two red arrows are present. One arrow points vertically upwards from the bottom towards the variable 'itemsCopy' in the 'sorted' function call. The other arrow points diagonally upwards from the bottom right towards the variable 'maxCost' in the 'if' condition.



# Algorithmic Efficiency

---

```
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)
```

$$\begin{array}{r} n \log n \\ + \\ n \\ \hline n \log n \end{array} \quad \text{where } n = \text{len}(\text{items})$$

# Using greedy

---

```
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:
        print('    ', item)
```

# Using greedy

---

```
def testGreedy(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)  
  
testGreedy(800)
```

# lambda

---

- lambda used to create anonymous functions
  - $\lambda \text{ id}_1, \text{id}_2, \dots, \text{id}_n: \text{expression}$
  - Returns a function of  $n$  arguments

# lambda

---

- lambda used to create anonymous functions
  - `lambda <id1, id2, ... idn>: <expression>`
  - Returns a function of n arguments
- Possible to write amazing complicated lambda expressions
- **Don't**—use `def` instead

# Using greedy

---

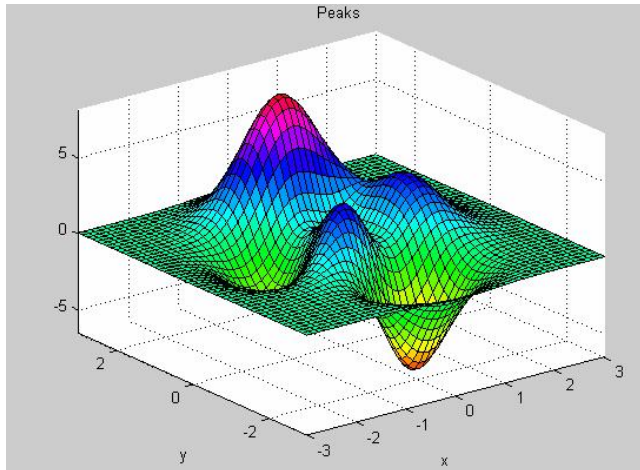
```
def testGreedy(foods, maxUnits):
    print('Use greedy by value to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('\nUse greedy by cost to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits,
                lambda x: 1/Food.getCost(x))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

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)
```

# Why Different Answers?

---

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

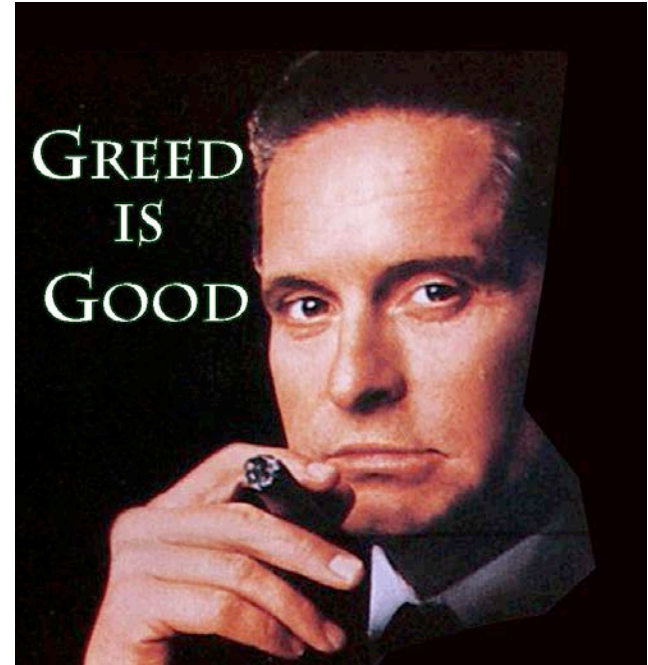


- Is greedy by density always a winner?
  - Try `testGreedy(foods, 1000)`

# 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