A greedy algorithm approach

- Take the best item
- Then take next best
- Continue until cannot fit any more items (e.g. because of weight)

- Still have to decide what is “best”
  - Most valuable? Lightest? Best value/weight ratio? Something else?

A greedy approach to the knapsack problem

```
class Item(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = float(v)
        self.weight = float(w)
    def getName(self):
        return self.name
    def getValue(self):
        return self.value
    def getWeight(self):
        return self.weight
    def __str__(self):
        result = '<' + self.name + ', ' + str(self.value)\
                 + ', ’ + str(self.weight) + '>'
        return result
```

A greedy approach to the knapsack problem

```
def buildItems():
    names = ['clock', 'painting’, 'radio',
             'vase', 'book’ 'computer']
    vals = [175,90,20,50,10,200]
    weights = [10,9,4,2,1,20]
    Items = []
    for i in range(len(vals)):
        Items.append(Item(names[i],
                          vals[i], weights[i]))
    return Items
```

Being greedy

```
def greedy(Items, maxWeight, keyFcn):
    assert type(Items) == list and maxWeight >= 0
    ItemsCopy = sorted(Items, key=keyFcn, reverse = True)
    result = []
    totalVal = 0.0
    totalWeight = 0.0
    i = 0
    while totalWeight < maxWeight and i < len(Items):
        if (totalWeight + ItemsCopy[i].getWeight()) <= maxWeight:
            result.append((ItemsCopy[i]))
            totalWeight += ItemsCopy[i].getWeight()
            totalVal += ItemsCopy[i].getValue()
        i += 1
    return (result, totalVal)
```

Remember a function is an object

- `ItemsCopy = sorted(Items, key=keyFcn, reverse = True)`
- Using `keyFcn` parameter lets us generalize one procedure to use different measures of goodness
- Just requires that `keyFcn` defines an ordering on the list of elements
- Then use this to create an ordered list

- Use `sorted` to create a copy of the list

So let’s get greedy

```
def value(item):
    return item.getValue()

def weightInverse(item):
    return 1.0/item.getWeight()

def density(item):
    return item.getValue()/item.getWeight()

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

So lets get greedy

```
def testGreedys(maxWeight = 20):
    Items = buildItems()
    print('Items to choose from:')
    for item in Items:
        print '  ', item
    print 'Use greedy by value to fill a 
knapsack of size', maxWeight
    testGreedy(Items, maxWeight, value)
    print 'Use greedy by weight to fill a 
knapsack of size', maxWeight
    testGreedy(Items, maxWeight, weightInverse)
    print 'Use greedy by density to fill a 
knapsack of size', maxWeight
    testGreedy(Items, maxWeight, density)
```

And if we are greedy?

```
>>> testGreedys()
…
Use greedy by value to fill a knapsack of size 20
Total value of items taken = 200.0
   <computer, 200.0, 20.0>
Use greedy by weight to fill a knapsack of size 20
Total value of items taken = 170.0
   <book, 10.0, 1.0>
   <vase, 50.0, 2.0>
   <radio, 20.0, 4.0>
   <painting, 90.0, 9.0>
Use greedy by density to fill a knapsack of size 20
Total value of items taken = 255.0
   <vase, 50.0, 2.0>
   <clock, 175.0, 10.0>
   <book, 10.0, 1.0>
   <radio, 20.0, 4.0>
```

No guarantee that any greedy algorithm will find the optimal solution

Efficiency of the greedy approach

- Two factors to consider
  - Complexity of sorted
  - Number of times through the while loop

- Latter is bounded by number of items in list (hence linear)
- But sorted is O(n log n)
- So overall algorithm is O(n log n), where n is length of list of items