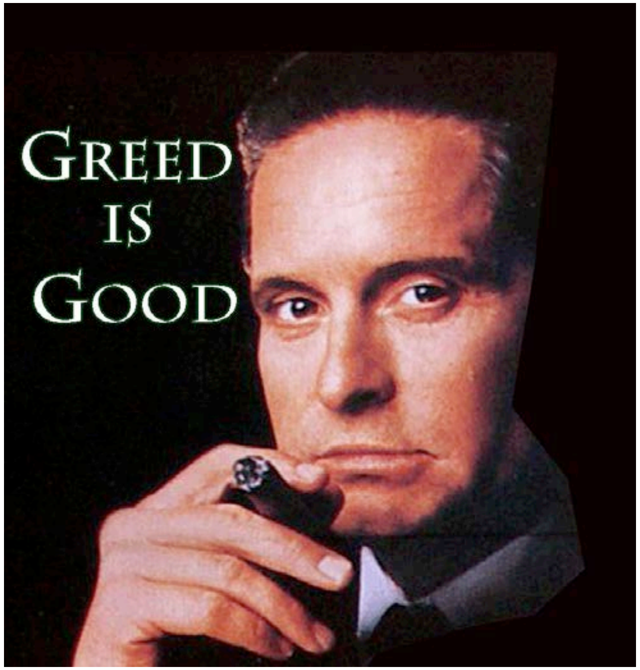
Optimization Problems, Lecture 2, Segment 1

John Guttag

MIT Department of Electrical Engineering and Computer Science

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
- On to finding optimal solutions

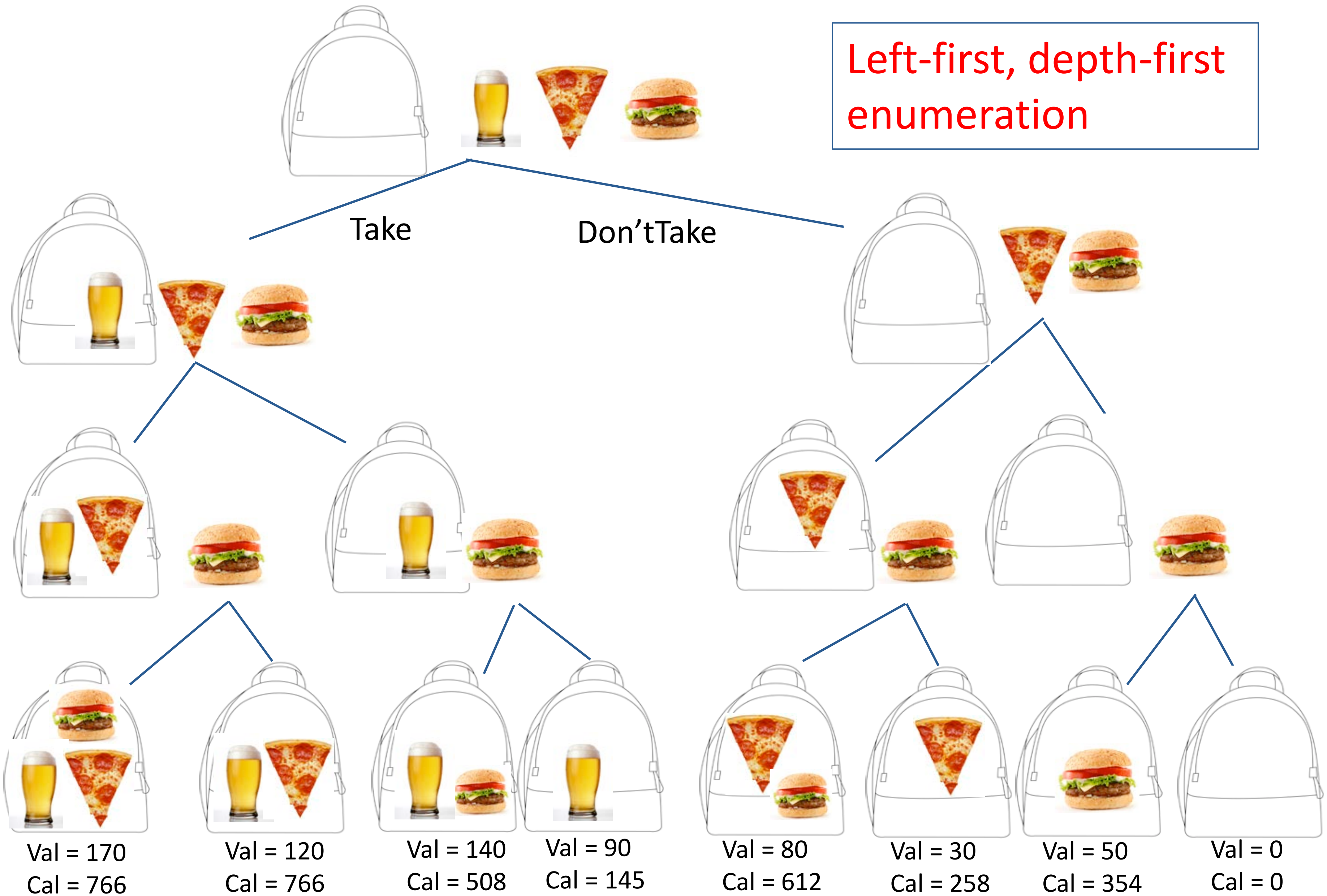
Brute Force Algorithm

- 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.

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
- Finally, chose a node with the highest value that meets constraints

A Search Tree Enumerates Possibilities



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 take or not
- So, if there are *n* items the number of nodes is
  - $\sum_{i=0}^{i=n} 2^i$
  - I.e., $O(2^{i+1})$ exponential
- An obvious optimization: don't explore parts of tree that violate constraint (e.g., too many calories)
  - Doesn't change complexity
- Does this mean that brute force is never useful?
  - Let's give it a try

Header for Decision Tree Implementation

```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,
               avail a weight
       Returns a tuple of the total value of a
           solution to 0/1 knapsack problem and
           the items of that solution"""
```

`toConsider`. Those items that nodes higher up in the tree (corresponding to earlier calls in the recursive call stack) have not yet considered

`avail`. The amount of space still available

Body of maxVal (without comments)

```
if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getUnits() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getUnits())
        withVal += nextItem.getValue()
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
    if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

Does not actually build search tree

Local variable `result` records best solution found so far

Try on Example from Lecture 1

- With calorie budget of 750 calories, chose an optimal set of foods from the 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

Search Tree Worked Great

- Gave us a better answer
- Finished quickly
- But 2^8 is not a large number
  - We should look at what happens when we have a more extensive menu to choose from