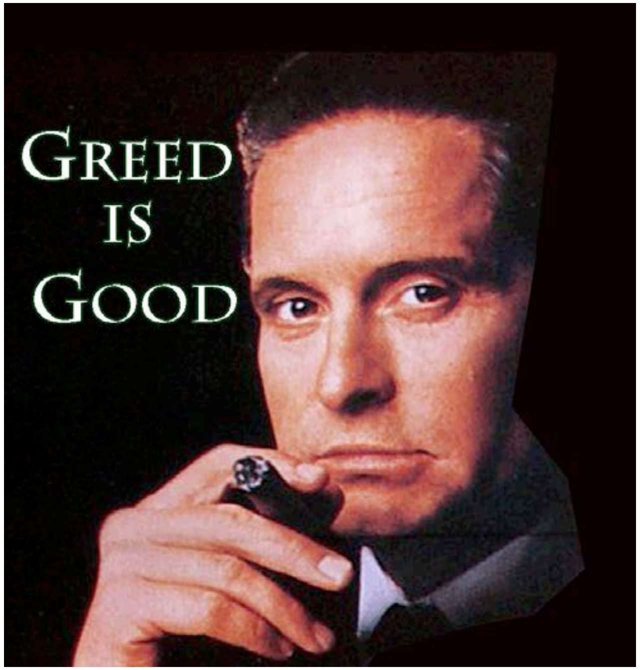
Optimization Problems, Lecture 2, Segment 1

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

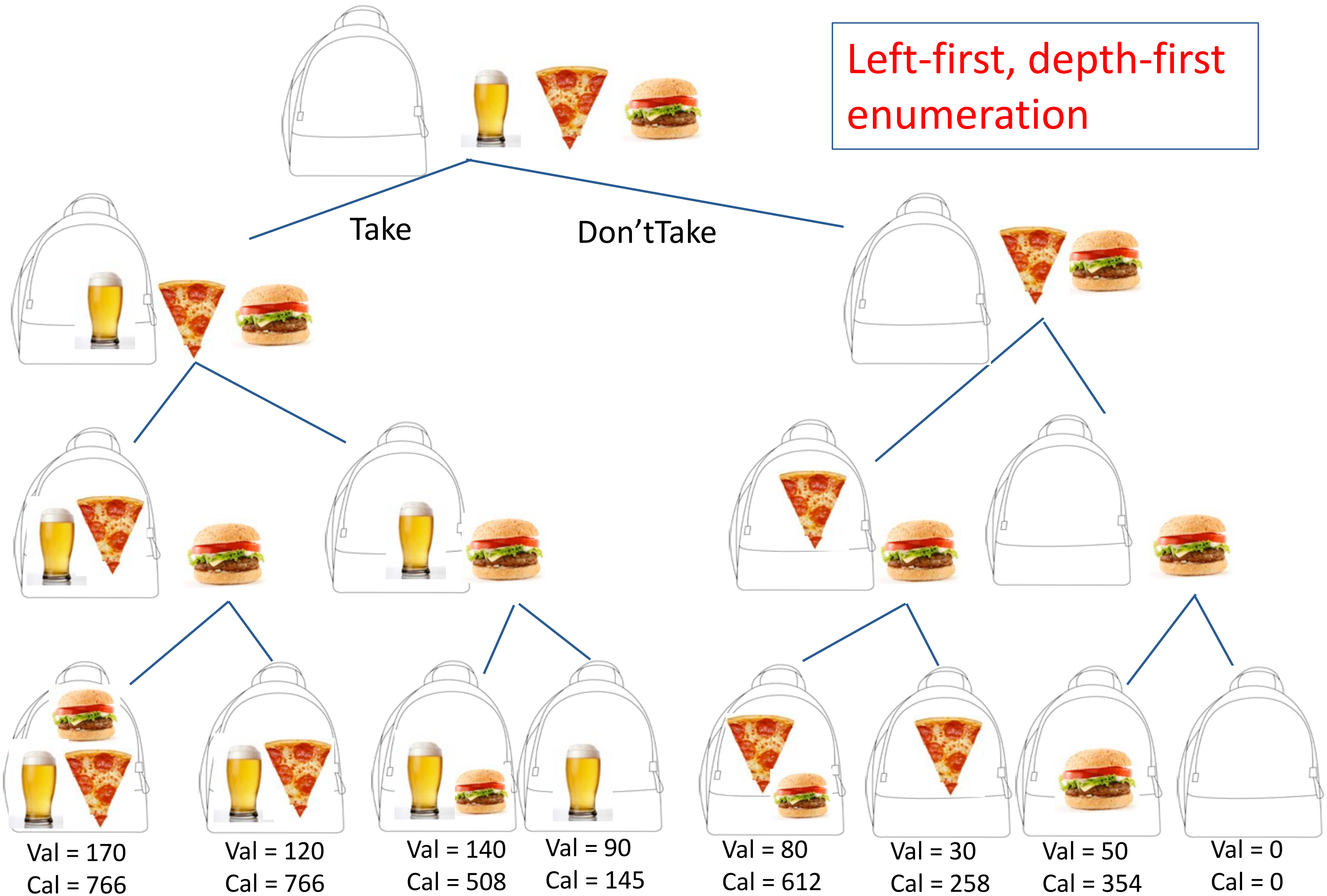
- 1. Enumerate all possible combinations of items.
- 2. Remove all of the combinations whose total units exceeds the allowed weight.
- 3. From the remaining combinations choose any one whose value is the largest.

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

Left-first, depth-first enumeration



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
- So, if there are *n* items the number of nodes is
  - $\sum_{i=0}^{i=n} 2^i$
  - I.e., $O(2^{i+1})$
- 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)

```
def maxVal(toConsider, avail):
"""Assumes toConsider a list of items, avail a weight
Returns a tuples of the total value of a solution to the 0/1 knapsack
problem and the items of that solution"""
if toConsider == [] or avail == 0:
    result = (0, ())
elif toConsider[0].getCost() > avail:
    # Explore right branch only
    result = maxVal(toConsider[1:], avail)
else:
    nextItem = toConsider[0]
    # Explore left branch
    withVal, withToTake = maxVal(toConsider[1:], avail - nextItem.getCost())
    withVal += nextItem.getValue()
    # Explore right branch
    withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
    # Explore better branch
    if withVal > withoutVal:
        result = (withVal, withToTake + (nextItem,))
    else:
        result = (withoutVal, withoutToTake)
return result
```

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

Optimization Problems, Lecture 2, Segment 2

John Guttag

MIT Department of Electrical Engineering and Computer Science

Search Tree Algorithm

- Gave us a better answer than any of the greedy solutions
- Finished quickly
- But 2^8 is not a large number
- Let's look at what happens when we have a more extensive menu to choose from

Code to Try Larger Examples

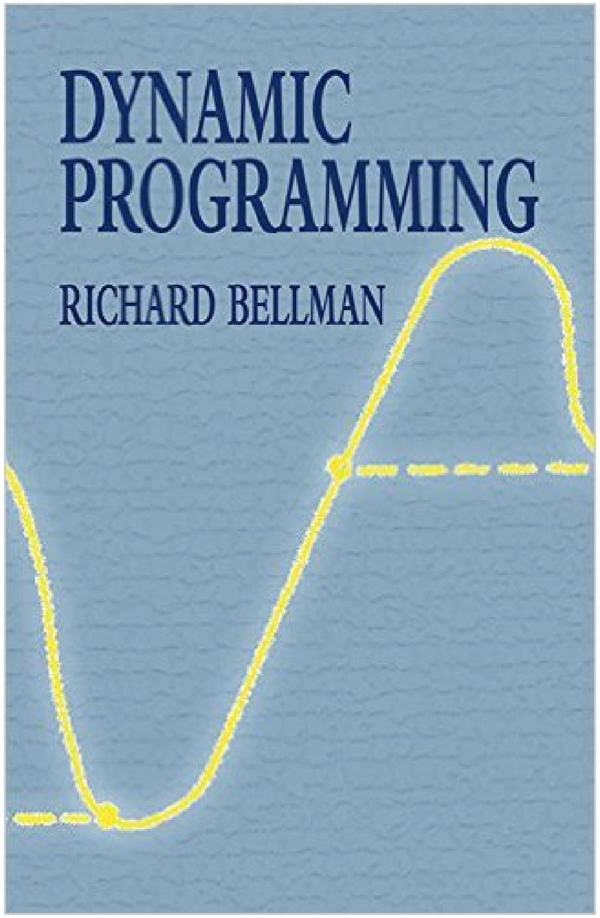
```
import random

def buildLargeMenu(numItems, maxVal, maxCost):
    items = []
    for i in range(numItems):
        items.append(Food(str(i),
                          random.randint(1, maxVal),
                          random.randint(1, maxCost)))
    return items


for numItems in (5, 10, 15, 20, 25, 30, 35, 40, 45):
    items = buildLargeMenu(numItems, 90, 250)
    testMaxVal(items, 750, False)
```

Is It Hopeless?

- In theory, yes
- In practice, no!
- Dynamic programming to the rescue



Dynamic Programming?

Sometimes a name is just a name

“The 1950s were not good years for mathematical research… I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics... What title, what name, could I choose? ... It's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

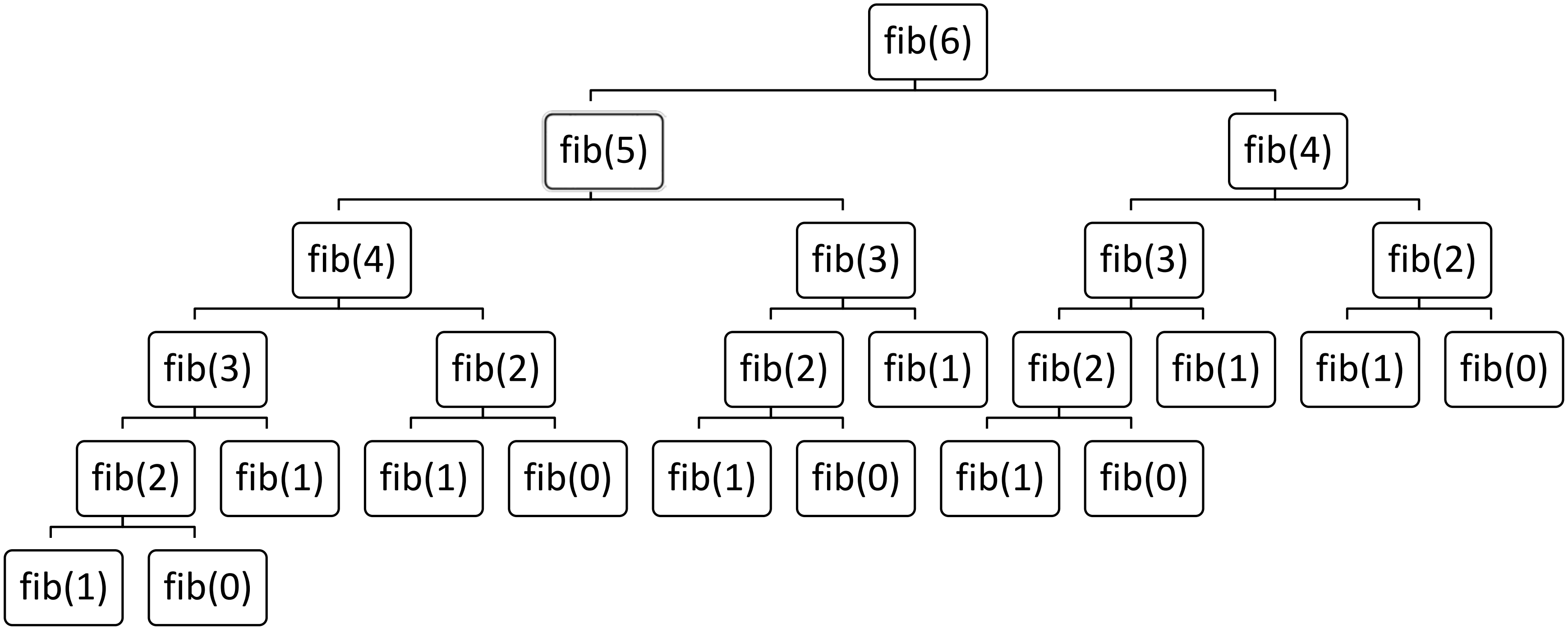
-- Richard Bellman

Recursive Implementation of Fibonnaci

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

```
fib(120) = 8,670,007,398,507,948,658,051,921
```

Call Tree for Recursive Fibonnaci(6) = 13



Clearly a Bad Idea to Repeat Work

- Trade a time for space
- Create a table to record what we've done
  - Before computing fib(x), check if value of fib(x) already stored in the table
    - If so, look it up
    - If not, compute it and then add it to table
  - Called memoization

Using a Memo to Compute Fibonnaci

```
def fastFib(n, memo = {}):
    """Assumes n is an int >= 0, memo used only by
       recursive calls
      Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fastFib(n-1, memo) +\
                 fastFib(n-2, memo)
        memo[n] = result
        return result
```

When Does It Work?

- Optimal substructure: a globally optimal solution can be found by combining optimal solutions to local subproblems
  - For x > 1, fib(x) = fib(x - 1) + fib(x – 2)

- Overlapping subproblems: finding an optimal solution involves solving the same problem multiple times
  - Compute fib(x) or many times

What About 0/1 Knapsack Problem?

- Do these conditions hold?

Optimization Problems, Lecture 2, Segment 3

John Guttag

MIT Department of Electrical Engineering and Computer Science

Dynamic Programming

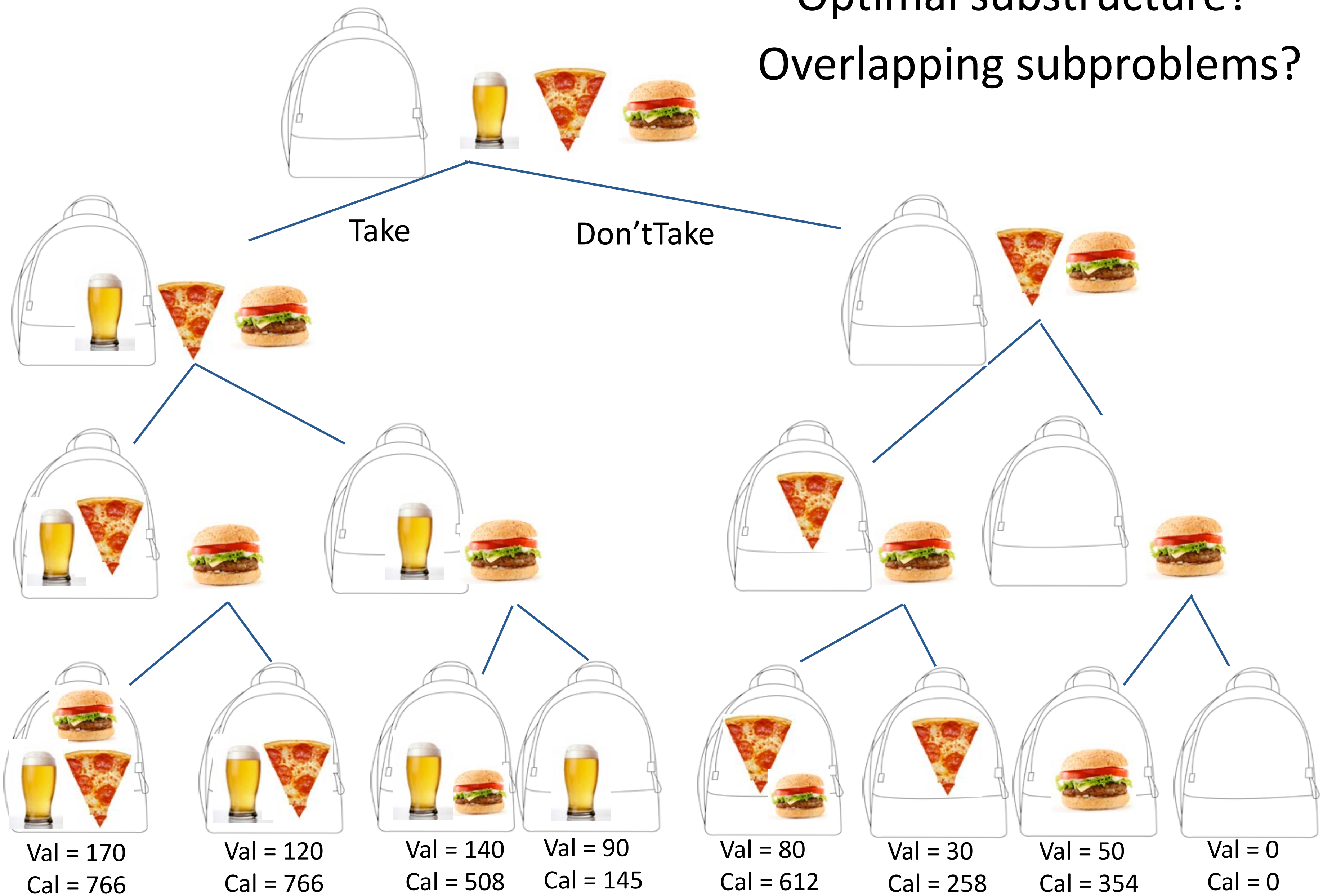
- Optimal substructure: a globally optimal solution can be found by combining optimal solutions to local subproblems
  - For x > 1, fib(x) = fib(x - 1) + fib(x – 2)

- Overlapping subproblems: finding an optimal solution involves solving the same problem multiple times
  - Compute fib(x) or many times

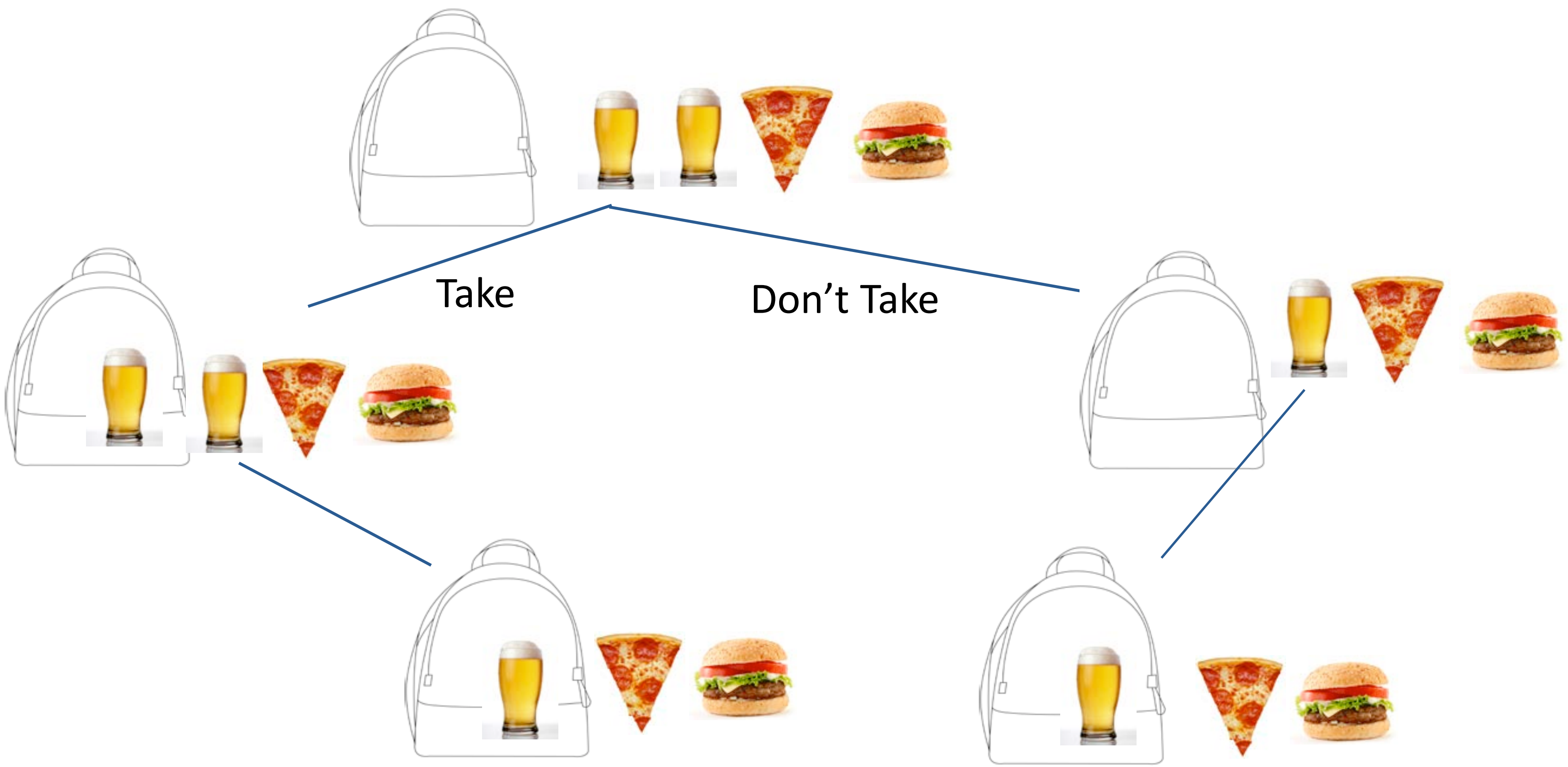
Search Tree

Optimal substructure?

Overlapping subproblems?



A Different Menu

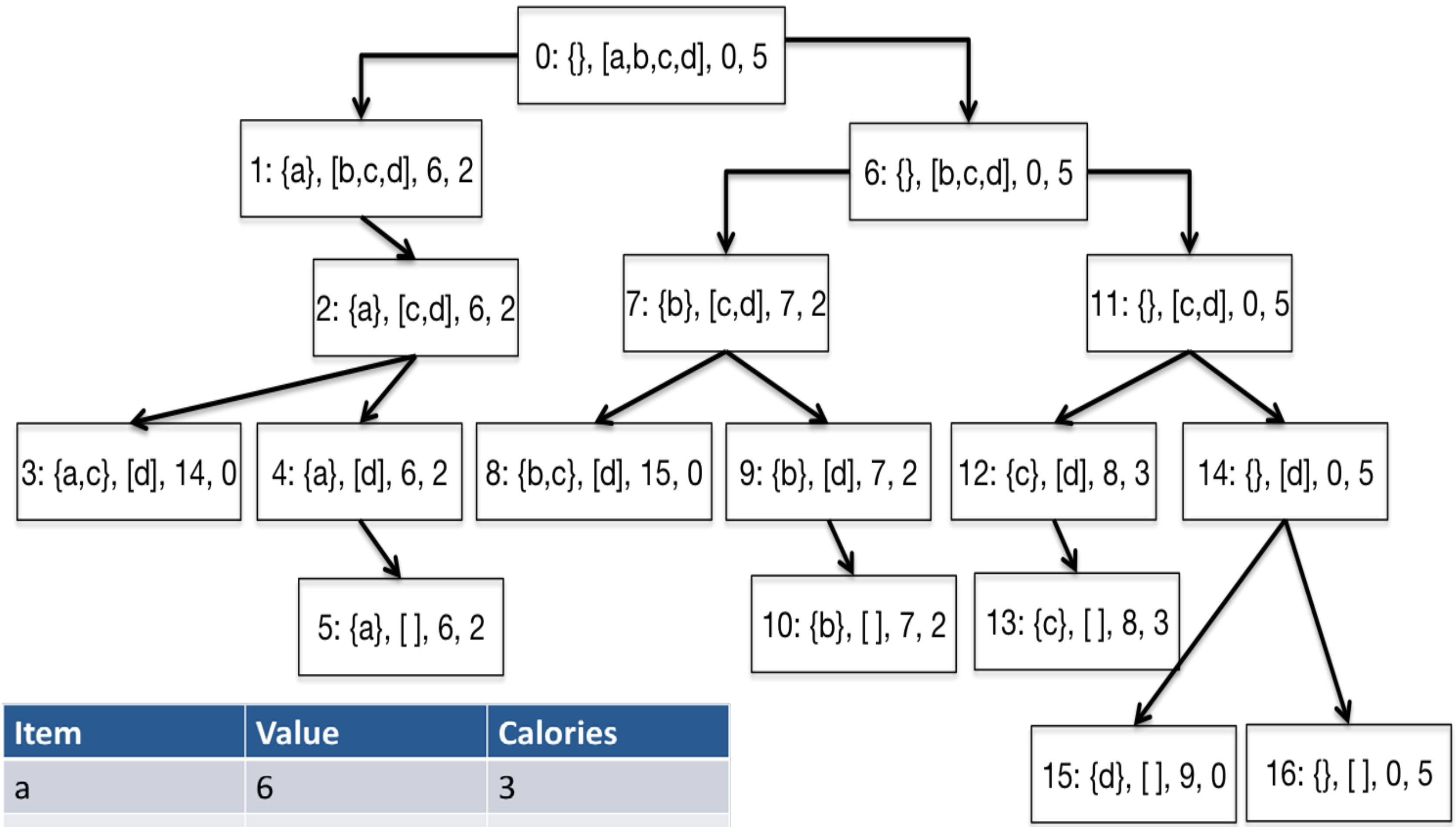


Need Not Have Copies of Items

Item	Value	Calories
a	6	3
b	7	3
c	8	2
d	9	5

Search Tree

- Each node = <taken, left, value, remaining calories>

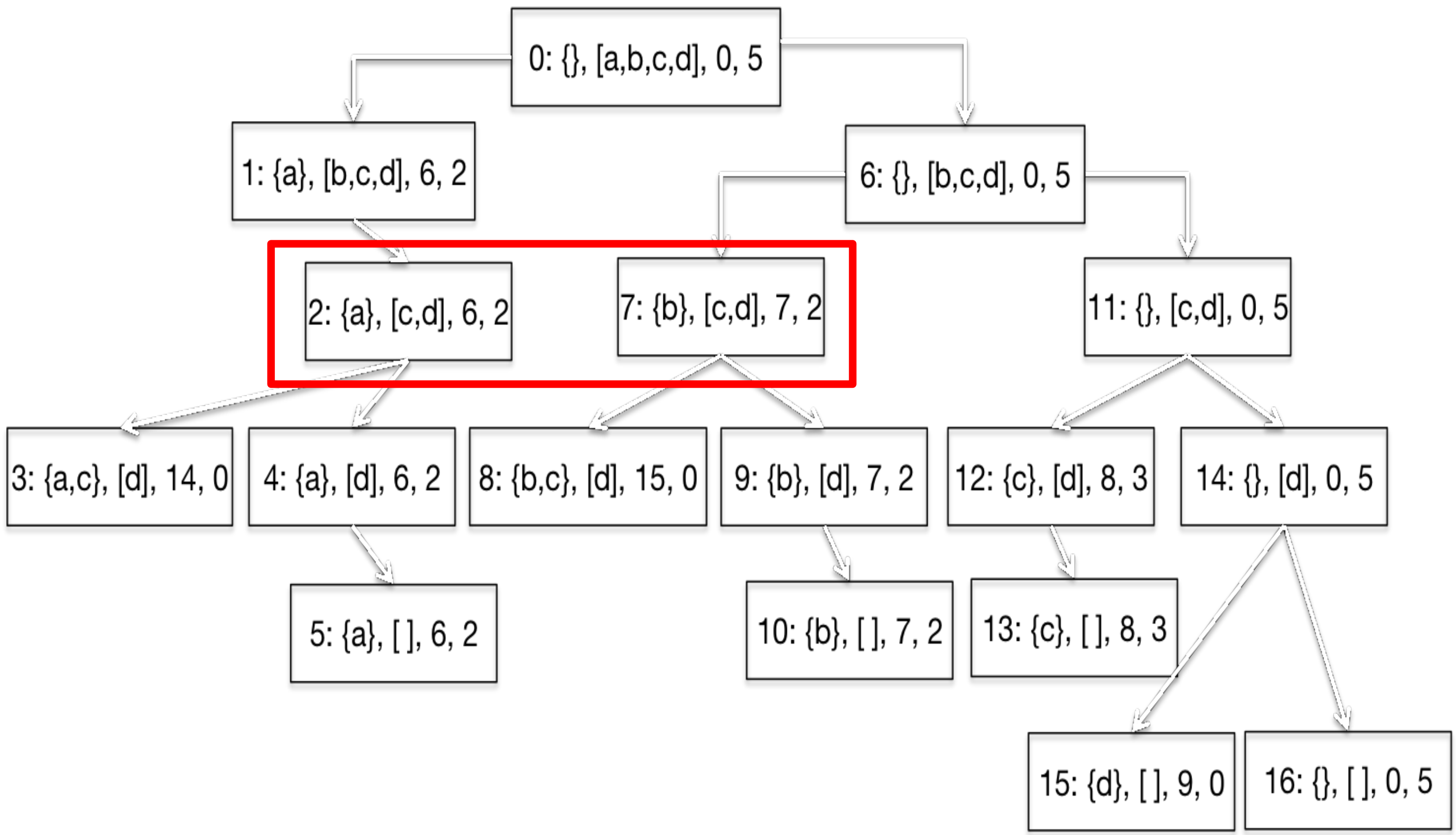


Item	Value	Calories
a	6	3
b	7	3
c	8	2
d	9	5

What Problem is Solved at Each Node?

- Given remaining weight, maximize value by choosing among remaining items
- Set of previously chosen items, or even value of that set, doesn’t matter!

Overlapping Subproblems



Modify maxVal to Use a Memo

- Add memo as a third argument
  - `def fastMaxVal(toConsider, avail, memo = {}):`
- Key of memo is a tuple
  - (items left to be considered, available weight)
  - Items left to be considered represented by `len(toConsider)`
- First thing body of function does is check whether the optimal choice of items given the the available weight is already in the memo
- Last thing body of function does is update the memo

Performance

len(items)	2**len(items)	Number of calls
2	4	7
4	16	25
8	256	427
16	65,536	5,191
32	4,294,967,296	22,701
64	18,446,744,073,709,551,616	42,569
128	Big	83,319
256	Really Big	176,614
512	Ridiculously big	351,230
1024	Absurdly big	703,802

How Can This Be?

- Problem is exponential
- Have we overturned the laws of the universe?
- Is dynamic programming a miracle?

How Can This Be?

- Problem is exponential
- Have we overturned the laws of the universe?
- Is dynamic programming a miracle?
- No, but computational complexity can be subtle
- Running time of `fastMaxVal` is governed by number of distinct pairs, `<toConsider, avail>`
  - Number of possible values of `toConsider` bounded by `len(items)`
  - Possible values of `avail` a bit harder to characterize
    - Bounded by number of distinct sums of weights
  - Covered in more detail in assigned reading

Summary of Lectures 1-2

- Many problems of practical importance can be formulated as optimization problems
- Greedy algorithms often provide adequate (though not necessarily optimal) solutions
- Finding an optimal solution is usually exponentially hard
- But dynamic programming often yields good performance for a subclass of optimization problems—those with optimal substructure and overlapping subproblems
  - Solution always correct
  - Fast under the right circumstances