Introduction and Optimization Problems

Jianjia Wang

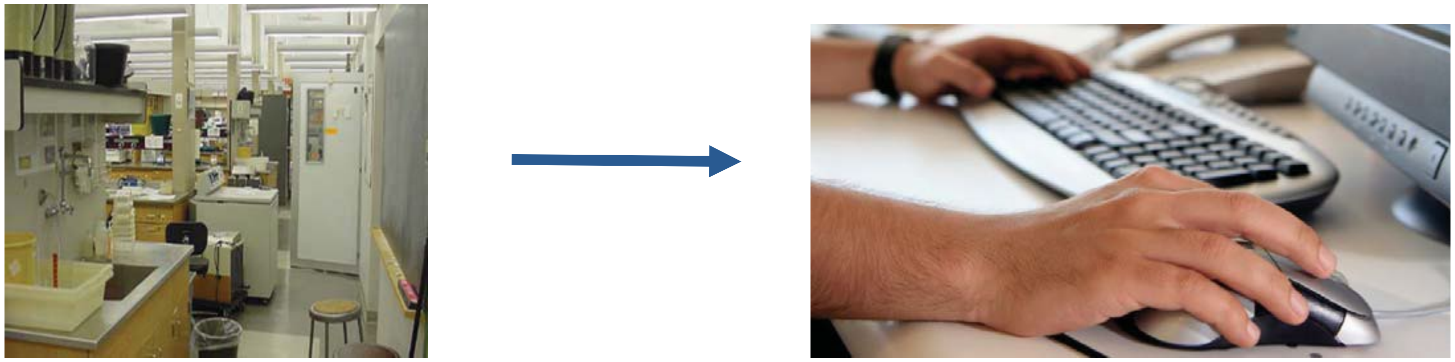
SHU School of Computer Engineering and Science

Relevant Reading for Today's Lecture

- Section 12.1
- Section 5.4 (lambda functions)
- Chapter 13

Computational Models

- Using computation to help understand the world in which we live
- Experimental devices that help us to understand something that has happened or to predict the future



- *Optimization models*
- Statistical models
- Simulation models

What Is an Optimization Model?

- An objective function that is to be maximized or minimized, e.g.,
  - Minimize time spent traveling from New York to Boston
- A set of constraints (possibly empty) that must be honored, e.g.,
  - Cannot spend more than $100
  - Must be in Boston before 5:00PM

Knapsack Problems

Knapsack Problem

- You have limited strength, so there is a maximum weight knapsack that you can carry
- You would like to take more stuff than you can carry
- How do you choose which stuff to take and which to leave behind?
- Two variants
  - 0/1 knapsack problem
  - Continuous or fractional knapsack problem

My Least-favorite Knapsack Problem

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 *f[i]* is taken If *V[i] = 0*, item *f[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

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

Brute Force Algorithm

- 1 Enumerate all possible combinations of items   That is to say, generate all subsets of the set of items   This is called the power set
- 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

Often Not Practical

- How big is power set?
- Recall
  - A vector, *V*, of length *n*, is used to indicate whether or not items are taken If *V[i] = 1*, item *f[i]* is taken If *V[i] = 0*, item *f[i]* is not taken
- How many possible different values can *V* have?
  - As many different binary numbers as can be represented in *n* bits
- For example, if there are 100 items to choose from, the power set is of size?
  - 1,267,650,600,228,229,401,496,703,205,376

Are We Just Being Stupid?

- Alas, no
- 0/1 knapsack problem is inherently exponential
- But don't despair

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 750 calories
- Choosing what to eat is a knapsack problem

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)
```

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)
```

Question 3

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 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)
```

?

lambda

- lambda used to create anonymous functions
    - `lambda` <id_1, id_2, . id_n>: <expression>
    - Returns a function of n arguments
- Can be very handy, as here
- Possible to write amazing complicated lambda expressions
- Don't- u se `def` instead

Using greedy

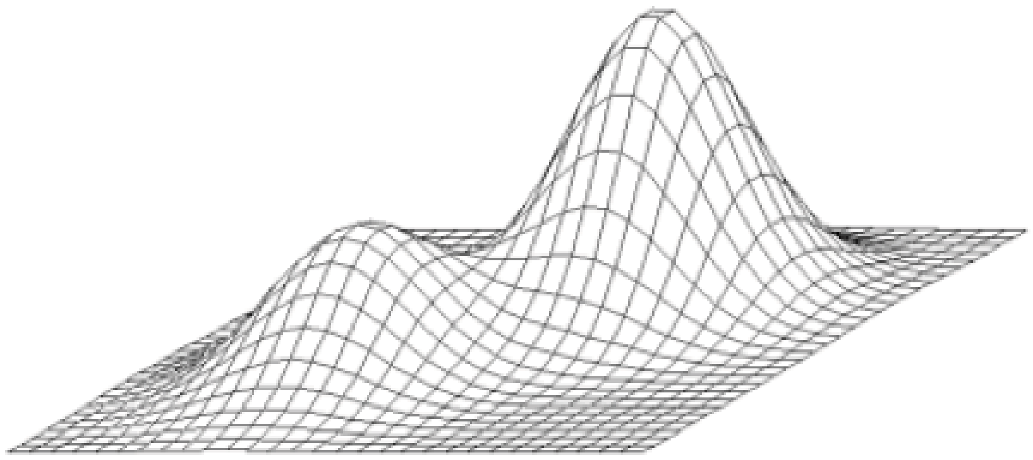
```
def testGreedys(foods, maxUnits):
    print('Use greedy by value to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('\nUse greedy by cost to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits,
               lambda x: 1/Food.getCost(x))
    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)
testGreedys(foods, 750)
```

Run code

Why Different Answers?

- Sequence of locally "optimal" choices don't always yield a globally optimal solution



- Is greedy by density always a winner?
  - Try testGreedys(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

Brute Force Algorithm

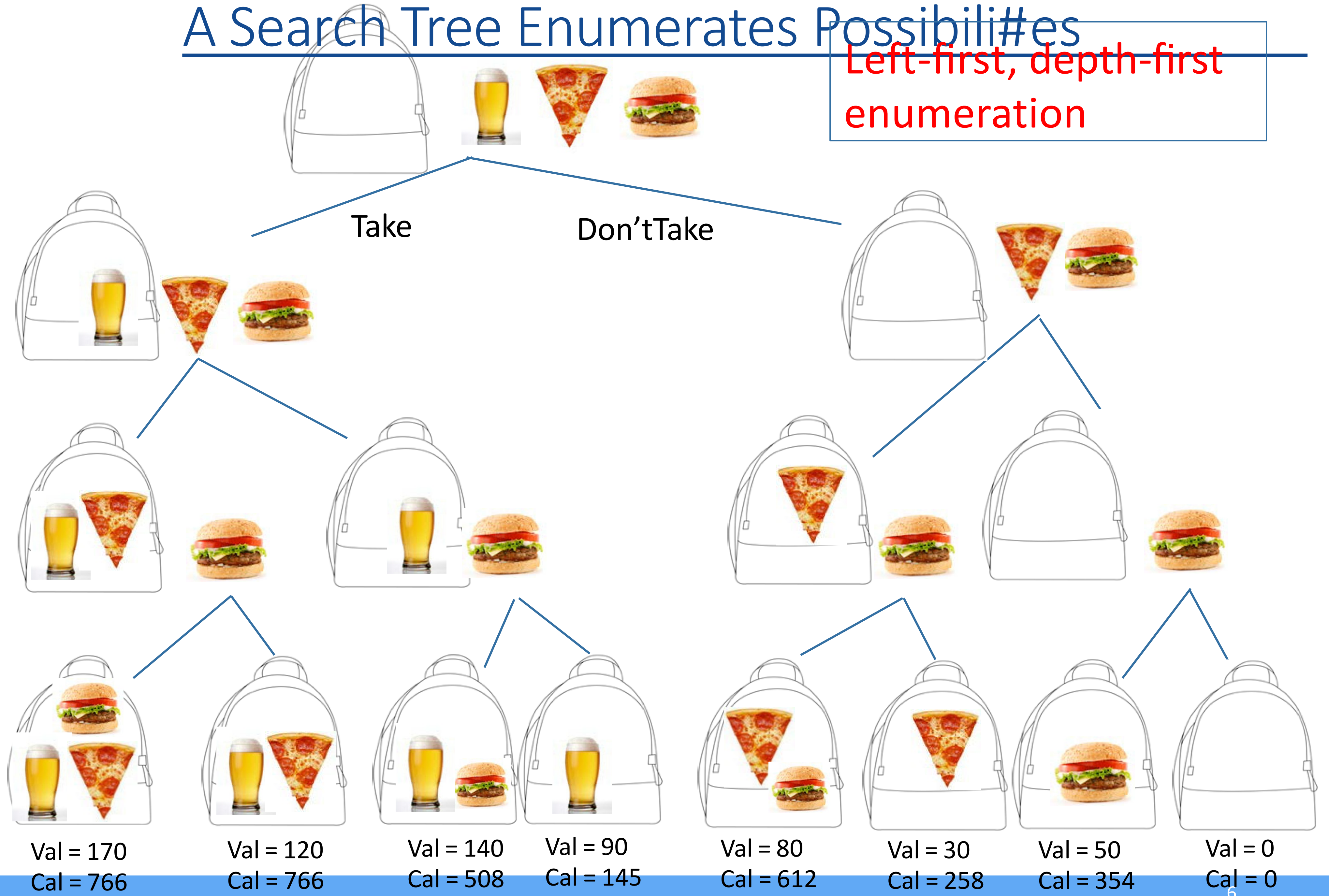
- 1. Enumerate all possible combinations of items.
- 2. Remove all of the combinations whose total units exceeds the allowed weight.
- 3. From the remaining combinations choose any one whose value is the largest.

Search Tree Implementation

- The tree is built top down staring 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 Possibili#es

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^{n+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)

```
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 solu<on 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

Code to Try Larger Examples

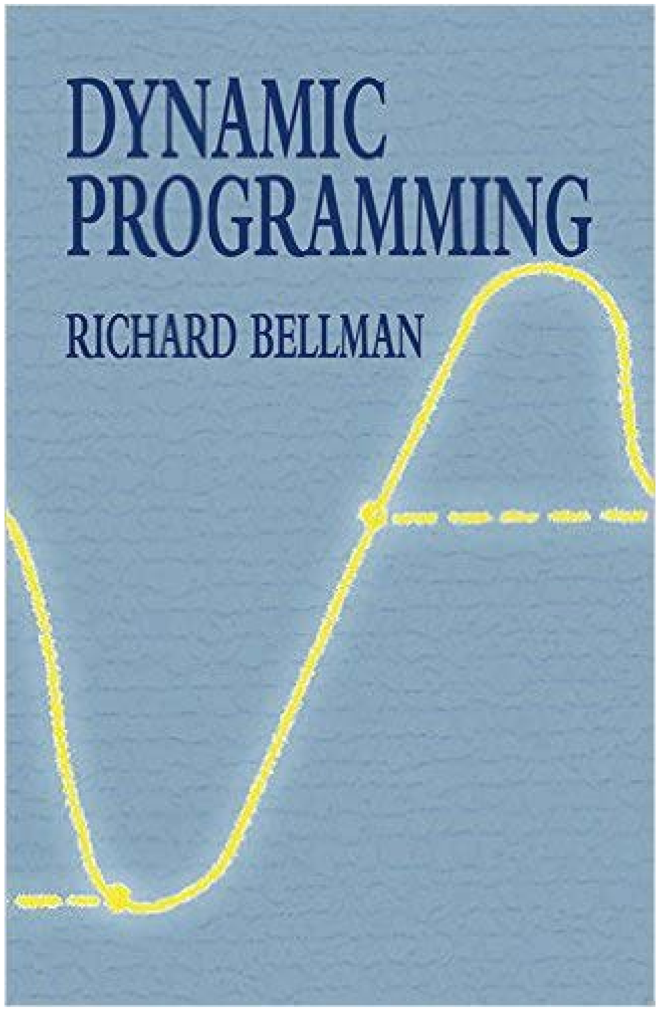
```
import random

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 (5,10,15,20,25,30,35,40,45,50,55,60):
    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.

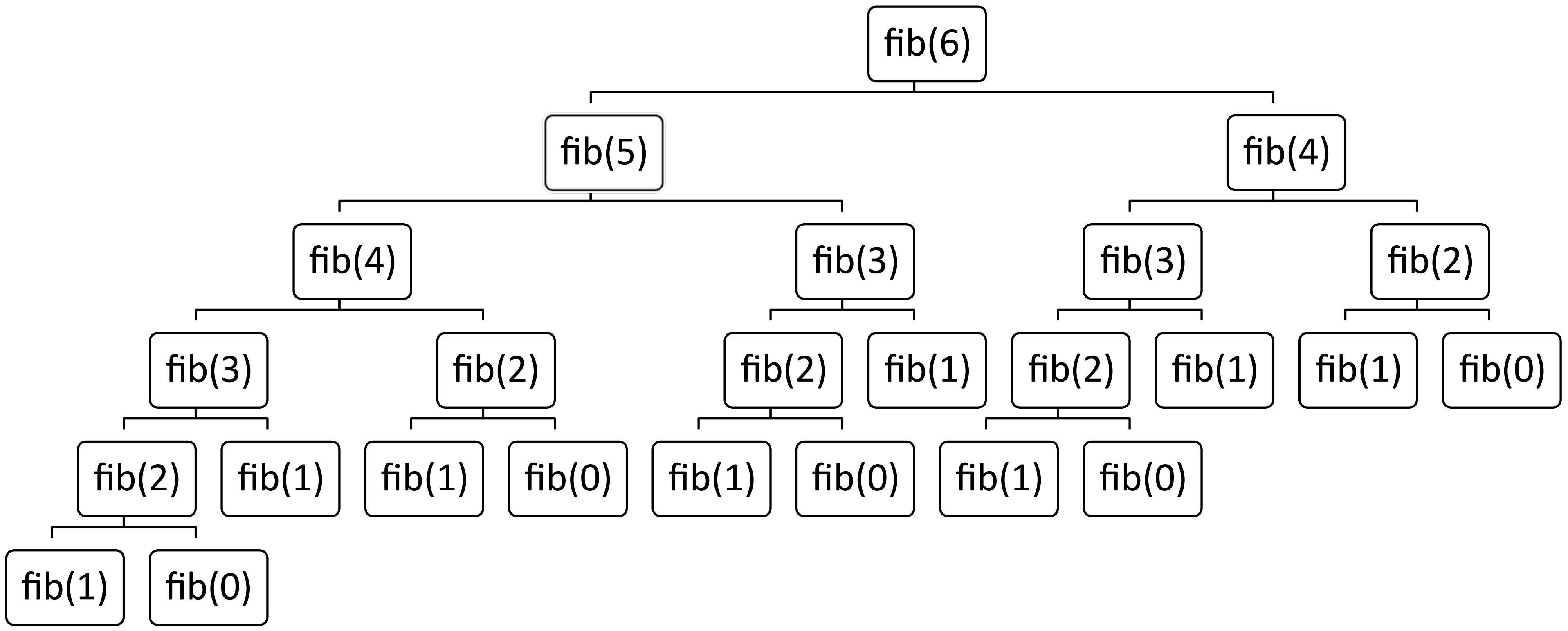
-- Richard Bellman

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?

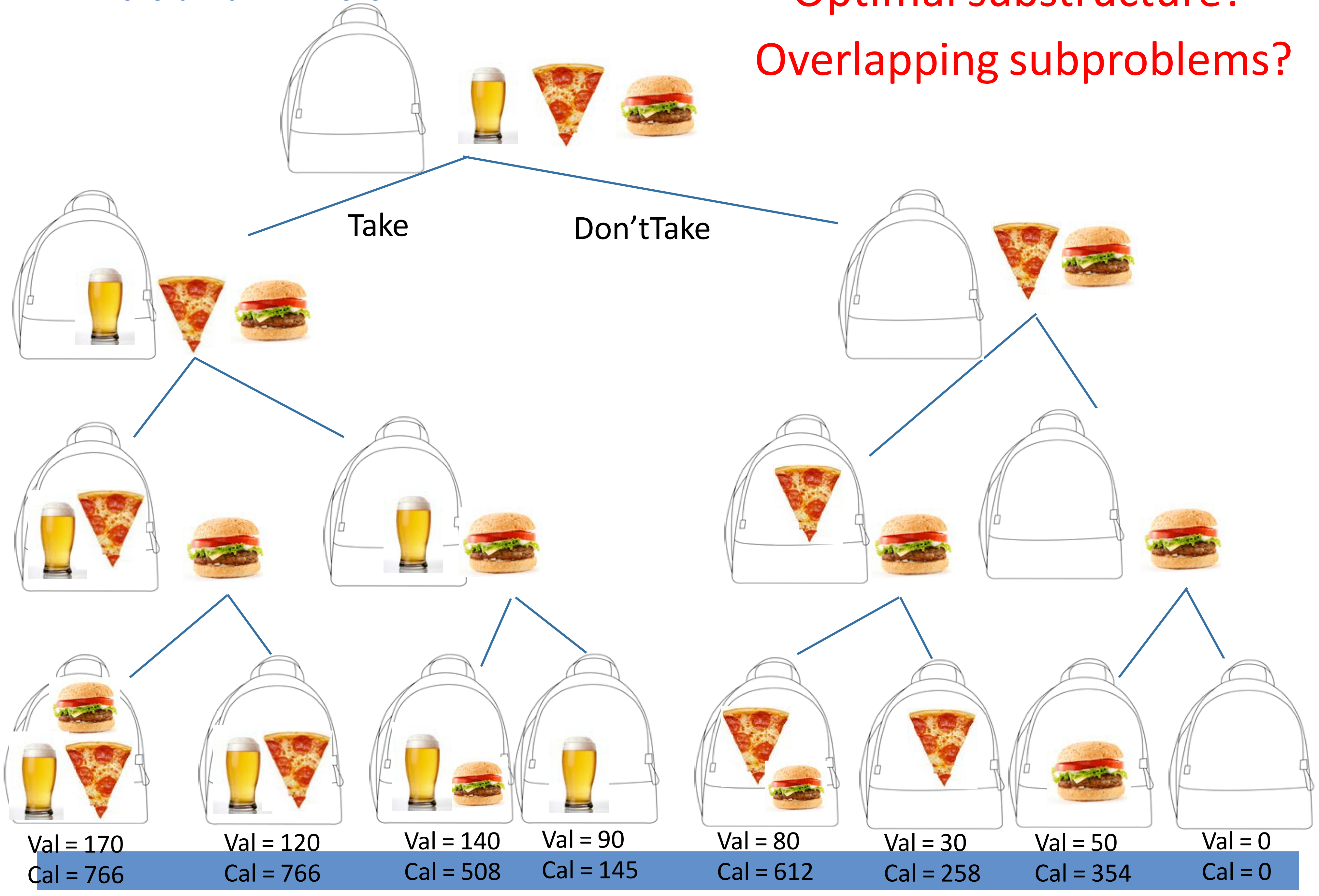
- Do these conditions hold?

Ques<ons 2 and 3

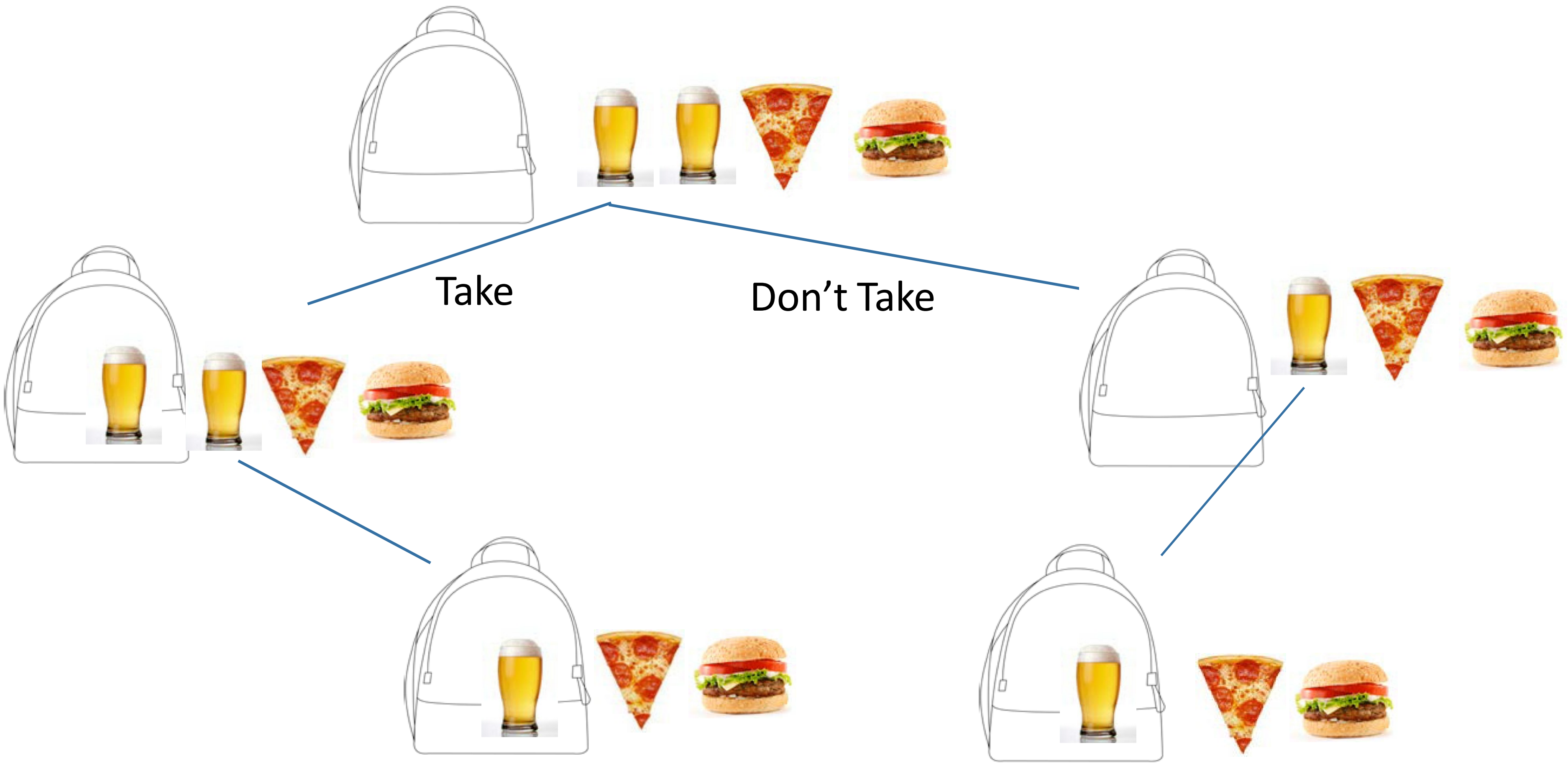
Search Tree

Optimal substructure?

Overlapping subproblems?



A Different Menu

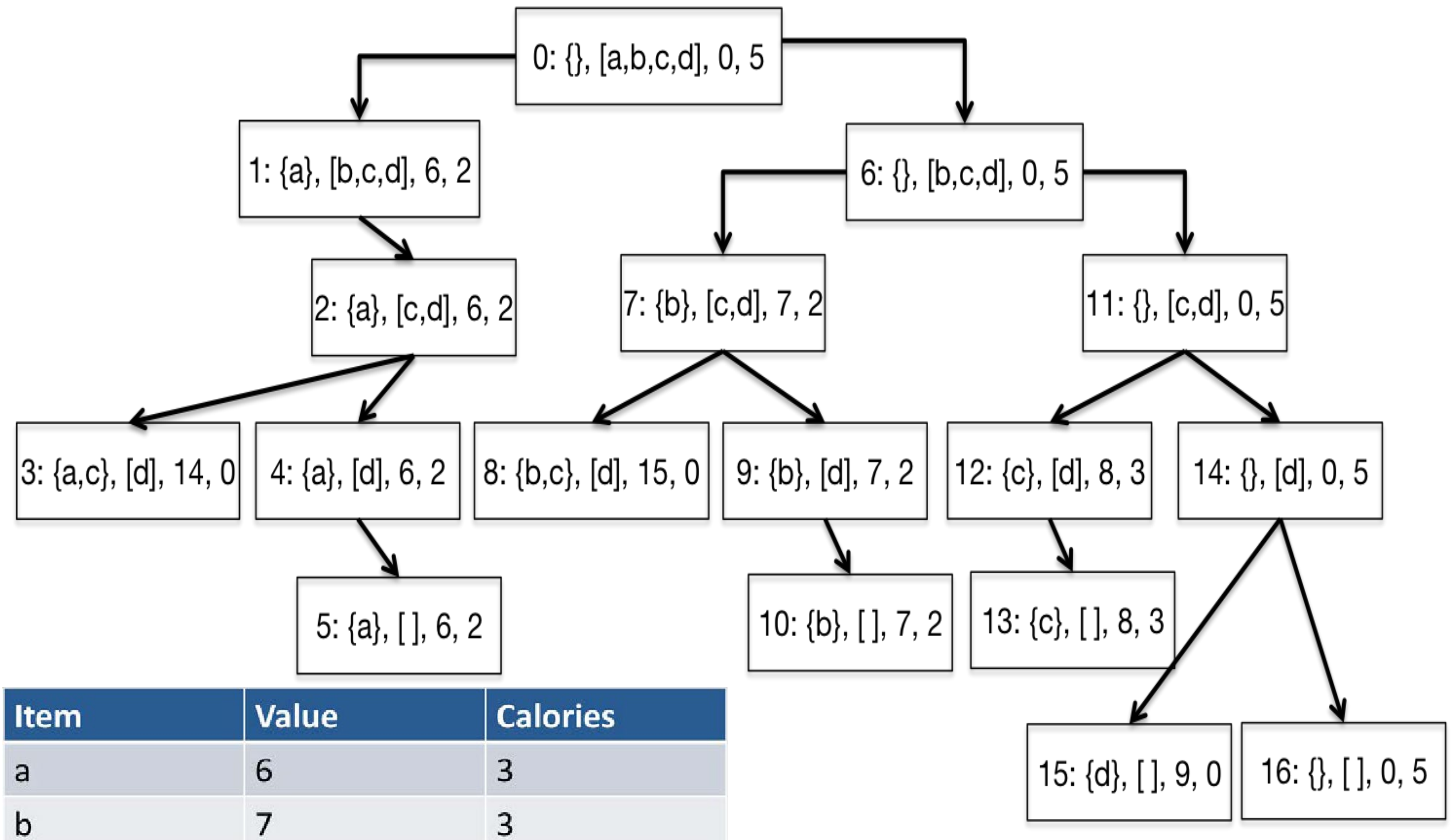


Need Not Have Copies of Items

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

Search Tree

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

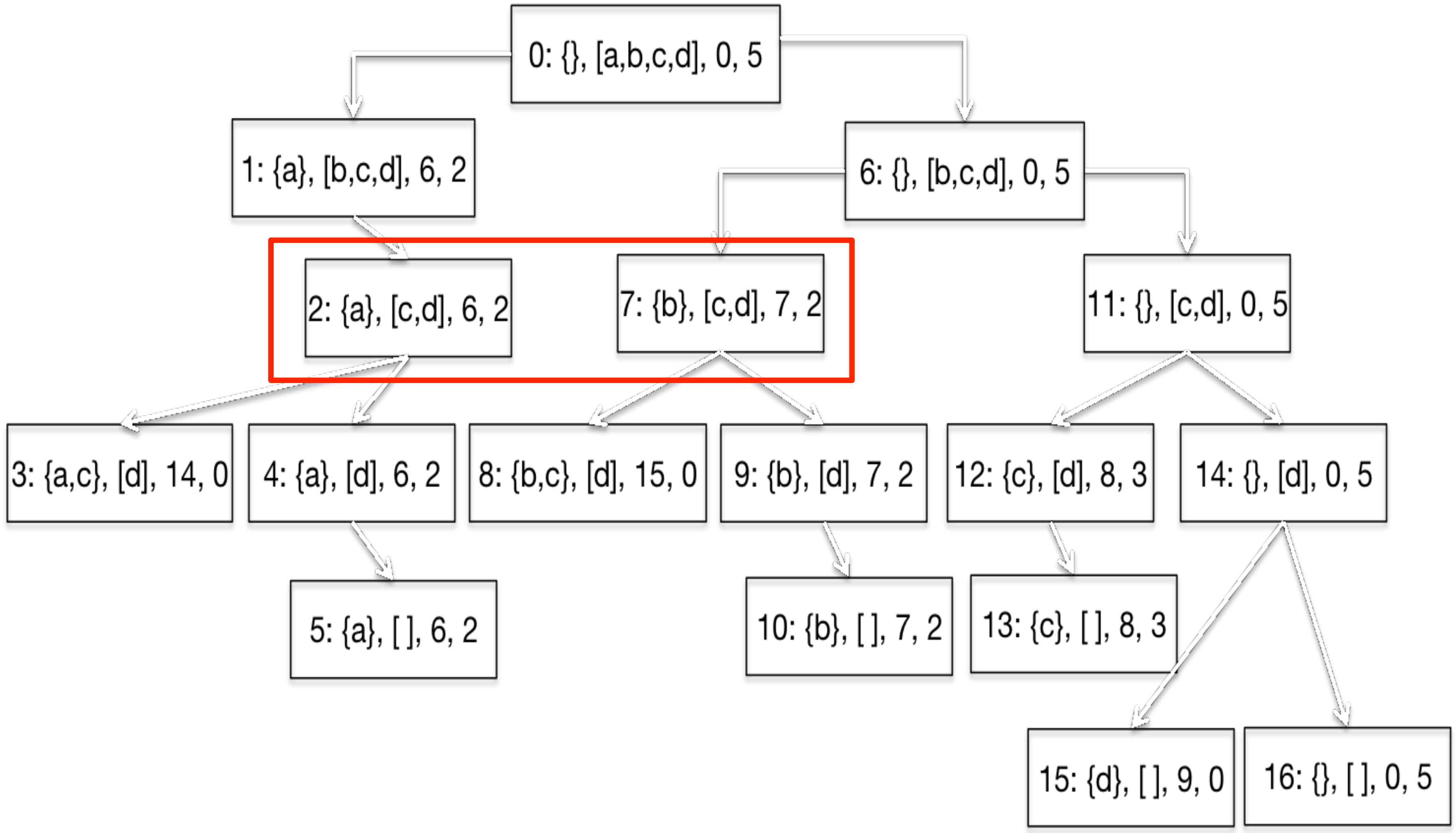


Item	Value	Calories
a	6	3
b	7	3
c	8	2
d	9	5

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 mafer!

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 leS to be considered, available weight)
  - Items leS 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	Absolutely huge	703,802

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 <me of `fastMaxVal` is governed by number of dis<nct 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

- 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

The “Roll-over” Optimization Problem

Score = ((60 – (a+b+c+d+e))*F + a*ps1 + b*ps2 + c*ps3 + d*ps4 + e*ps5

Objec<ve:

Given values for F, ps1, ps2, ps3, ps4, ps5

Find values for a, b, c, d, e that maximize score

Constraints:

a, b, c, d, e are each 10 or 0

$a + b + c + d + e \geq 20$