

# Knapsack Problems and Dynamic Programming

(download slides and .py files from Stellar to follow along)

---

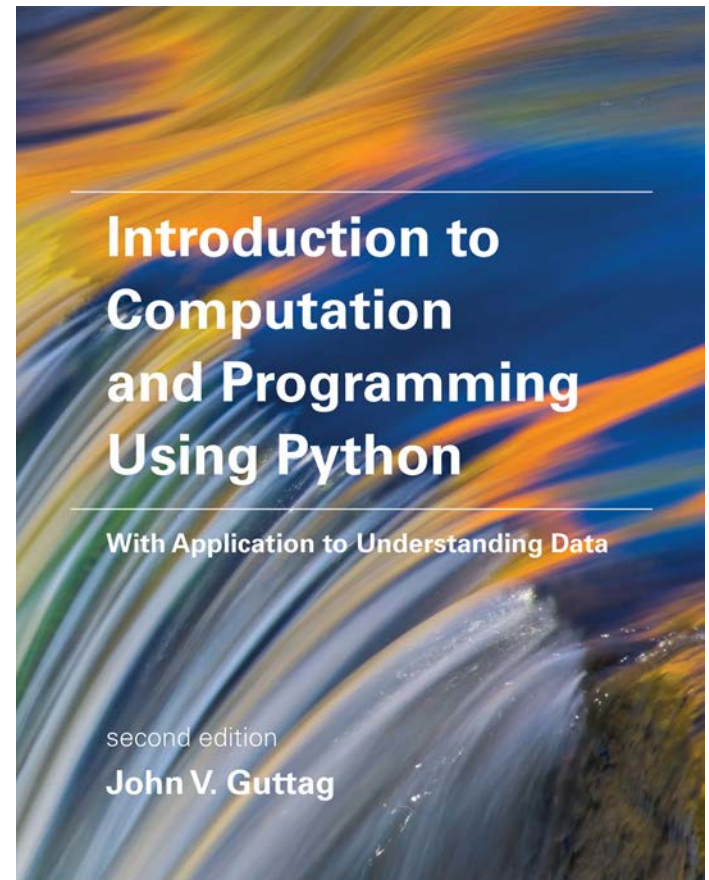
John Guttag

MIT Department of Electrical Engineering and  
Computer Science

# Relevant Reading

---

- Chapter 13
- Section 5.3.2 (List comprehension)



# Search Tree Implementation, revisited

---

- 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
- Once tree generated, chose a node with the highest value that meets constraints

# 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 the 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

---

```
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)
    #Choose better branch
    if withVal > withoutVal:
        result = (withVal, withToTake + (nextItem,))
    else:
        result = (withoutVal, withoutToTake)
return result
```

Don't explore paths  
that exceed  
constraint

Local variable  
result records  
best solution so far

# Some Things to Note

---

- Don't actually build a search tree
- Generate one path through the tree at a time
  - Path encoded in recursive call stack
- Keep track of best path so far
  - In local variable `result`

# Try It 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

Use greedy by value on 8 items

Total value of items taken = 284

burger: <100, 354>

pizza: <95, 258>

wine: <89, 123>

Use greedy by density on 8 items

Total value of items taken = 318

wine: <89, 123>

beer: <90, 154>

cola: <79, 150>

apple: <50, 95>

donut: <10, 195>

Use greedy by cost on 8 items

Total value of items taken = 318

apple: <50, 95>

wine: <89, 123>

cola: <79, 150>

beer: <90, 154>

| donut: <10, 195>

Optimal value = 353

cola: <79, 150>

pizza: <95, 258>

beer: <90, 154>

wine: <89, 123>



# Search Tree Worked Great


---

- Gave us a better answer than any of the greedies
- 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

---

```
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
```



```
import random  
random.seed(1)
```



```
numCalories = 750  
for numItems in (8, 16, 32, 64, 128, 256, 512, 1024):  
    items = buildLargeMenu(numItems, 100, 300)  
    print('Test on', len(items), 'items')  
    greedyVal = testGreedy(items, numCalories, False)  
    print('Best greedy solution =', greedyVal)  
    optVal = testMaxVal(items, numCalories, False)  
    print('Optimal solution =', optVal, '\n')
```

# Random Test Data

---

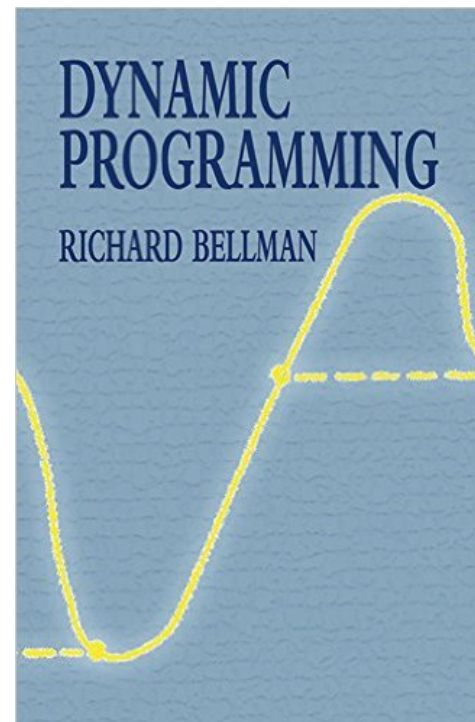
- When you want to see how things scale, you need large test sets
- You want to generate them, not type them
- Generating them randomly
  - Helps to avoid bias
  - Can generate different test data each run
    - Both good and bad
    - Used `random.seed` to ensure same result each time

**Run it**

# Is It Hopeless?

---

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



# Returning to an Example from 6.0001

---

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

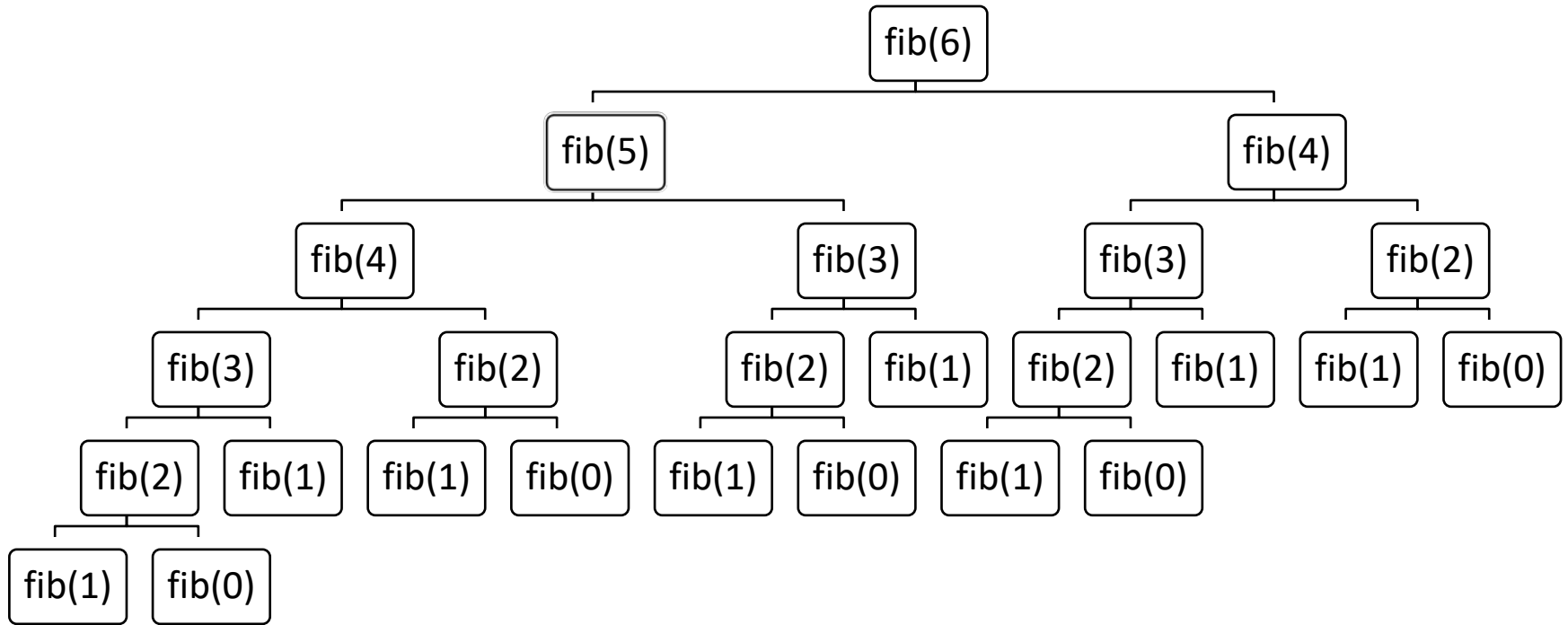
```
for i in range(0, 1001, 5):  
    print('fib(' + str(i) + ') =', fib(i))
```

How far do think we'll get before our patience runs out?

*Why is it taking so long?*

# Call Tree for Recursive Fibonacci(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  $\text{fib}(x)$ , check if value of  $\text{fib}(x)$  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 = None):  
    """Assumes n is an int >= 0, memo used only by  
        recursive calls  
        Returns Fibonacci of n"""  
    if memo == None:  
        memo = {}  
    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
```



# Let's Try It

---

How far do think we'll get before our patience runs out?

Let's push our luck: `for i in range(0, 5001, 100):`

# Tabular Approach to DP

---

- Memoization top-down
  - Start from problem to be solved, the biggest problem
  - Build memo as new sub-problems come up
- Tabular bottom-up
  - Solve all sub-problems starting with smallest problem

```
def fastFibTab(n):  
    """Assumes n is an int >= 0  
       Returns Fibonacci of n"""  
    tab = [1]*(n+1) #only first two values matter  
    for i in range(2, n + 1):  
        tab[i] = tab[i-1] + tab[i-2]  
    return tab[n]
```

Try it

# Tabularization vs. Memoization

---

- If the original problem requires all subproblems to be solved, tabular method usually better
  - Tabular method easier to implement
  - Tabulation usually faster. (Tabulation has no overhead for recursion and can use a pre-allocated fixed size list.)
- If only some of the subproblems needs to be solved to solve the original problem, memoization usually better
  - More efficient because subproblems are solved **lazily**, only perform the the computations that are needed

# When Does DP Help?

---

- **Optimal substructure**: a globally optimal solution can be found by combining optimal solutions to local subproblems
  - For  $x > 1$ ,  $\text{fib}(x) = \text{fib}(x - 1) + \text{fib}(x - 2)$
- **Overlapping subproblems**: finding an optimal solution involves solving the same problem multiple times
  - Compute  $\text{fib}(x)$  or many times

# What About 0/1 Knapsack Problem?

---

- Do these conditions hold?



# Five Minute Break

---

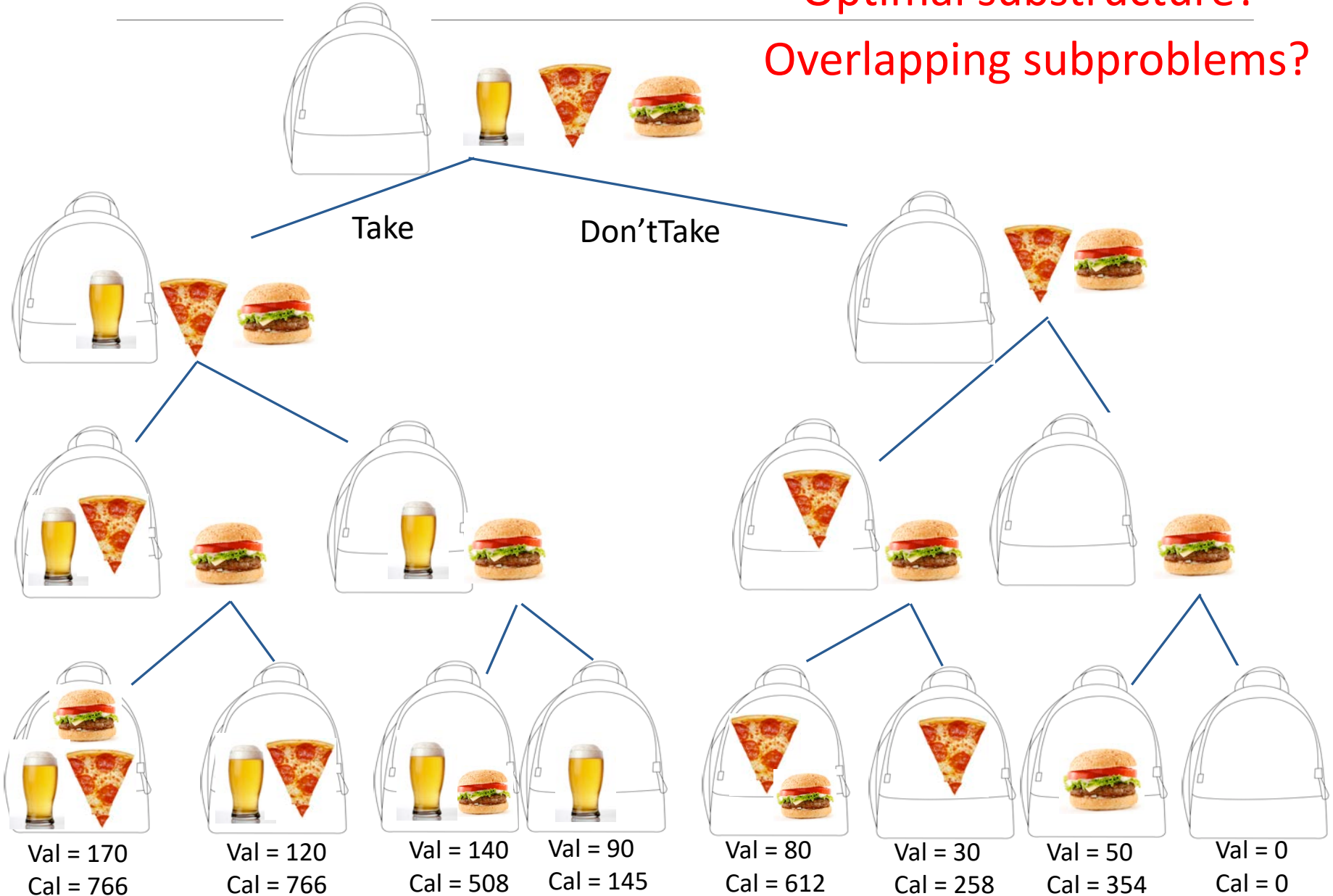
# Does the Knapsack Problem Exhibit

---

- **Optimal substructure**: a globally optimal solution can be found by combining optimal solutions to local subproblems
- **Overlapping subproblems**: finding an optimal solution involves solving the same problem multiple times

# Search Tree

Optimal substructure?  
Overlapping subproblems?



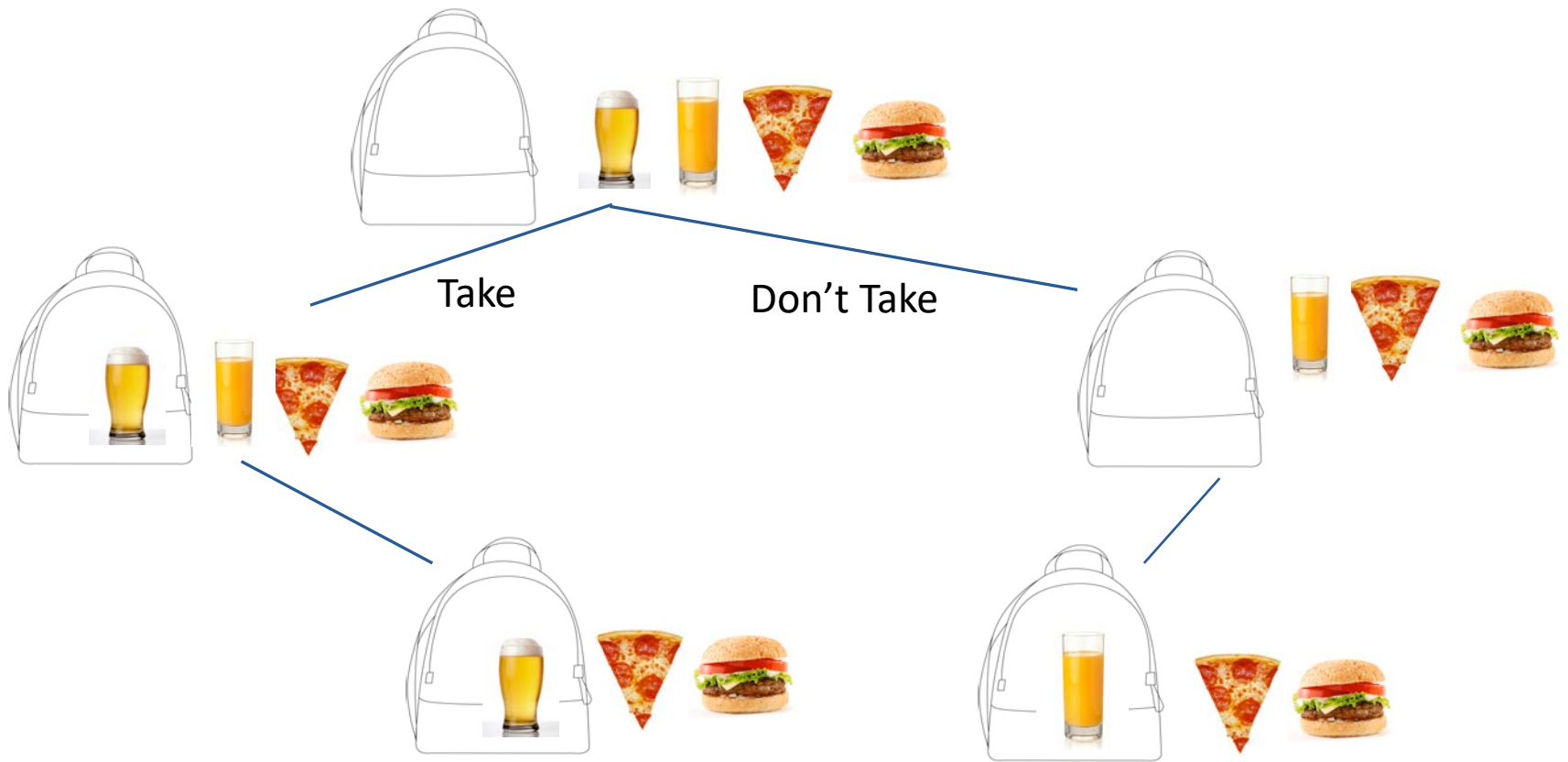


# A Different Menu

---

Food	beer	pizza	burger	juice
Value	90	30	50	85
calories	154	258	354	154

# A Subtree



# Problem Being 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!
- So, let's give DP a shot
  - Memoization or tabular?

# Modify maxVal to Use a Memo

---

- Add memo as a third argument
- 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)	$2^{**}\text{len(items)}$	Number of calls in DP
2	4	7
4	16	25
8	256	427
16	65,536	5,191
32	4,294,967,296	22,701

# Performance

len(items)	$2^{**} \text{len(items)}$	Number of calls in DP
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	340,282,366,920,938,463,463,374,607,431,768,211,456	83,319
256	115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457,584,007,913,129,639,936	176,614

# How Can This Be?

---

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

# A Miracle?

---

- No, but computational complexity can be subtle
- Algorithm falls into a complexity class called **pseudo-polynomial**

```
def isPrime(x):  
    """Assumes x is an int > 2  
    Returns True if x is prime and false otherwise"""  
    for i in range(2, x):  
        if x%i == 0:  
            return False  
    return True
```

Linear in **value** of x

But for some range of values, we don't care about number of bits

But exponential in number bits used to represent x, i.e. in **length** of input— $2^{\log(x)}$



## And fastMaxVal?

---

- Running time of `fastMaxVal` is polynomial in 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

# More Distinct Combinations

---

```
def buildLargeMenu1(numItems, maxVal, maxCost):
    items = []
    for i in range(numItems):
        items.append(Food(str(i),
                           random.randint(1, maxVal),
                           random.random()*maxCost))

    return items

import random
random.seed(1)

numCalories = 750
for numItems in (8, 16, 32, 64, 128, 256, 512, 1024):
    items = buildLargeMenu1(numItems, 100, 300)
    print('Test on', len(items), 'items')
    optVal = testFastMaxVal(items, numCalories, False)
    print('Optimal solution =', optVal, '\n')
```

# Why is It Called Dynamic Programming?

---

“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, *Eye of the Hurricane: an Autobiography*

# Summary of Lectures 1-2 (so far)

---

- 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

# Since I Have a Few Minutes, more Python

---

- Conditional expressions
- List comprehension

# Conditional Expressions

---

`e1 if c else e2`

- 1) Evaluate `c`
- 2) If `c` is True, the value of the expression is `e1`
- 3) If `c` is False, the value of the expression is `e2`

`x/y if y != 0 else None`

# List Comprehension

---

```
L = [expression for item in L if conditional]
```

```
L = []  
for item in list:  
    if conditional:  
        L.append(expression)
```

# List Comprehension

---

```
L = [i*i for i in range(10)]
```

```
L = []  
for i in range(10):  
    L.append(i*i)
```



# List Comprehension

---

```
L = [i**2 for i in range(10) if i%2 != 0]
```

```
sentence = '6.0002 is the greatest'  
L = [word[0] for word in sentence.split(' ')]  
print(L)
```

```
D = {word[0]:sentence.count(word[0])\  
      for word in sentence.split(' ')}  
print(D)
```

# Today's Puzzler

---

```
def getSomething(n):  
    return [p for p in range(2, n+1)\  
            if 0 not in [p%d for d in range(2, p)]]  
  
print([x for x in range(101) if x not in\  
      ({i+1:getSomething(101)[i]\  
       for i in range(len(getSomething(101)))}).values()])
```