Use-It-Or-Lose-It
The Packing Problem

```python
>>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False
>>> subset(8, [2, 3, 5, 7, 10, 42])
True

def subset(target, L):

    Try THIS without recursion!?"
The Packing Problem

>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
    if target == 0: return True
The Packing Problem

>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
    if target == 0: return True
    elif L == []: return False

What if we switched the order of these?
The Packing Problem

```python
>>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
```
The Packing Problem

>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        useIt = subset(target - L[0], L[1:])
        loseIt = subset(target, L[1:])
The Packing Problem

>> subset(12, [2, 3, 4, 7, 10, 42])
True
>>> subset(8, [2, 3, 4, 7, 10, 42])
False

def subset(target, L):
    if target == 0: return True
    elif L == []: return False
    elif L[0] > target: return subset(target, L[1:])
    else:
        useIt = subset(target - L[0], L[1:])
        loseIt = subset(target, L[1:])
        return useIt or loseIt
The Knapsack Problem...

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ]
237

def knapsack(capacity, WVList):
```
The Knapsack Problem…

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
>>> knapsack(7, [[2, 100], [3, 112], [4, 125]])
237

def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0  # 2 bases cases!
```
The Knapsack Problem...

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity:  5?  6?  7?

```python
def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0  # 2 bases cases!
    else:
        firstItem = WVList[0]
        firstItemWeight = firstItem[0]
        firstItemValue = firstItem[1]```

Worksheet and Demo
The Knapsack Problem...

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
>>> knapsack(7, [[2, 100], [3, 112], [4, 125]])
237

def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0  # 2 bases cases!
    else:
        firstItem = WVList[0]
        firstItemWeight = firstItem[0]
        firstItemValue = firstItem[1]
        if firstItemWeight > capacity:
            return 0
        else:
            return max(firstItemValue + knapsack(capacity - firstItemWeight, WVList[1:]), knapsack(capacity, WVList[1:]))
```

Worksheet and Demo
The Knapsack Problem...

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0  # 2 bases cases!
    else:
        firstItem = WVList[0]
        firstItemWeight = firstItem[0]
        firstItemValue = firstItem[1]
        if firstItemWeight > capacity:  # must lose it!
            return knapsack(capacity, WVList[1:])
        else:
            return firstItemValue + knapsack(capacity - firstItemWeight, WVList[1:])
```

>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ])  
237
The Knapsack Problem...

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ])
237
```

```python
def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0 # 2 bases cases!
    else:
        firstItem = WVList[0]
        firstItemWeight = firstItem[0]
        firstItemValue = firstItem[1]
        if firstItemWeight > capacity:  # must lose it!
            return knapsack(capacity, WVList[1:])
        else:
            useIt = firstItemValue +
            knapsack(capacity-firstItemWeight, WVList[1:])
```

Worksheet and Demo
The Knapsack Problem…

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0 # 2 bases cases!
    else:
        firstItem = WVList[0]
        firstItemWeight = firstItem[0]
        firstItemValue = firstItem[1]
        if firstItemWeight > capacity:  # must lose it!
            return knapsack(capacity, WVList[1:])
        else:
            useIt = firstItemValue +
                knapsack(capacity-firstItemWeight, WVList[1:])
            loseIt = knapsack(capacity, WVList[1:])
    return max(useIt, loseIt)
```

```python
>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ])
237
```

Worksheet and Demo
The Knapsack Problem...

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spam</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>Tofu</td>
<td>3</td>
<td>112</td>
</tr>
<tr>
<td>Chocolate</td>
<td>4</td>
<td>125</td>
</tr>
</tbody>
</table>

Knapsack Capacity: 5? 6? 7?

```python
>>> knapsack(7, [(2, 100), (3, 112), (4, 125)])
237
```

def knapsack(capacity, WVList):
    if capacity == 0 or WVList == []: return 0  # 2 bases cases!
    else:
        firstItem = WVList[0]
        firstItemWeight = firstItem[0]
        firstItemValue = firstItem[1]
        if firstItemWeight > capacity:  # must lose it!
            return knapsack(capacity, WVList[1:]
        else:
            useIt = firstItemValue +
            knapsack(capacity-firstItemWeight, WVList[1:])
            loseIt = knapsack(capacity, WVList[1:])
            return max(useIt, loseIt)
```

Worksheet and Demo
Power Set!

>>> powerset([1, 2])
[[], [2], [1], [1, 2]]

>>> powerset([1, 2, 3])
[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]

>>> powerset([1])

>>> powerset([])

This really demonstrates the power of functional programming!

The order in which the subsets are presented is unimportant but within each subset, the order should be consistent with the input set.
The zzyzva is known to be a xenophobic creature with a zealous personality...

Now we can delete the original file!

**TEXT FILE**
zzyzva.txt
58,254 bytes

**TEXT FILE**
zzyzva.txt.Z
23,124 bytes

Compression algorithm (e.g. zip)

B6^9)=%
%spam!
=&&penguin/
?’,/+
Data Compression

Now we can delete the original file!

TEXT FILE
zyzyzva.txt.Z

23,124 bytes
Data Compression

The zzyzva is known to be a xenophobic creature with a zealous personality...

DEcompression algorithm (e.g. unzip)

B6^9)="n% %spam! =&&penguin/ ?’,/+

Now we can delete the original file!

TEXT FILE
zzyzva.txt
58,254 bytes

TEXT FILE
zzyzva.txt.Z
23,124 bytes
Data Compression!

The zzyzva is known to be a xenophobic creature with a zealous personality…

But these statistics are on average, not for my essay on the zzyzva!

<table>
<thead>
<tr>
<th>Letter</th>
<th>ord(Letter)</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>84</td>
<td>01010100</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>01101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>01100101</td>
</tr>
<tr>
<td>z</td>
<td>122</td>
<td>01111010</td>
</tr>
</tbody>
</table>

- “ “ 1226754 19.04%
- E 655257 10.17%
- T 474521 7.37%
- A 425718 6.61%
- ... skipping a few ...
- J 5329 0.08%
- Q 4923 0.08%
- Z 3378 0.05%

English text letter frequencies
Variable Length Encodings

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>0.25</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0.10</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0.09</td>
<td>00</td>
</tr>
<tr>
<td>a</td>
<td>0.08</td>
<td>01</td>
</tr>
<tr>
<td>r</td>
<td>0.02</td>
<td>101001111100</td>
</tr>
<tr>
<td>p</td>
<td>0.01</td>
<td>101001111101</td>
</tr>
</tbody>
</table>

The zzyzva is known to be a xenophobic creature with a zealous personality…

Yes!! These frequencies are for *my* essay!!

Cute idea, but what’s the problem here?
The zzyzva is known to be a xenophobic creature with a zealous personality…

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
<th>Binary code</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>0.25</td>
<td>00</td>
</tr>
<tr>
<td>y</td>
<td>0.10</td>
<td>01</td>
</tr>
<tr>
<td>x</td>
<td>0.09</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>0.08</td>
<td>111</td>
</tr>
<tr>
<td>r</td>
<td>0.02</td>
<td>1100</td>
</tr>
</tbody>
</table>

101110100001100 = 10 111 01 00 00 1100
Consider the Language “Spamish” which has only four letters in its alphabet…

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
<th>Fixed Length</th>
<th>Variable Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
<td>10</td>
<td>110</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

Expected average number of bits per symbol = \( 0.6 \times 1 + 0.2 \times 2 + 0.1 \times 3 + 0.1 \times 3 = 1.6 \)

1.6 is 80% of 2.0, so we expect 20% space savings!
The Variable Length Coding Problem…

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$\text{freq}(a_1)$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$\text{freq}(a_2)$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$\text{freq}(a_3)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$a_n$</td>
<td>$\text{freq}(a_n)$</td>
</tr>
</tbody>
</table>

Objective: Find a binary prefix code that minimizes…

These frequencies are from the specific file that we’re planning to compress!!
The Variable Length Coding Problem…

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>freq($a_1$)</td>
</tr>
<tr>
<td>$a_2$</td>
<td>freq($a_2$)</td>
</tr>
<tr>
<td>$a_3$</td>
<td>freq($a_3$)</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>$a_n$</td>
<td>freq($a_n$)</td>
</tr>
</tbody>
</table>

Objective: Find a binary prefix code that minimizes…

freq($a_1$) $\times$ codelength($a_1$) +
freq($a_2$) $\times$ codelength($a_2$) + …
freq($a_n$) $\times$ codelength($a_n$)

These frequencies are from the specific file that we’re planning to compress!!
The David Huffman Story!

Huffman coding is one of the fundamental ideas that people in computer science and data communications are using all the time - Donald Knuth
The David Huffman Story!

map smppam
ssampamsman
...

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

TEXT FILE

ENCODING:
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter) - why is this a prefix code?
4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file
You Try It!

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>0.40</td>
</tr>
<tr>
<td>a</td>
<td>0.20</td>
</tr>
<tr>
<td>r</td>
<td>0.15</td>
</tr>
<tr>
<td>v</td>
<td>0.15</td>
</tr>
<tr>
<td>e</td>
<td>0.06</td>
</tr>
<tr>
<td>y</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Build the tree and write down the codes for each of the symbols

Then encode the string “haha” using this code
Dictionaries

>>> D.keys()
['Ran', 'Lucy', 'Justin']

>>> D
{'Ran': 'spam', 'Lucy': 'chocolate', 'Justin': 42}

>>> del D['Ran']

>>> D
{'Lucy': 'chocolate', 'Justin': 42}

>>> Cal = {2005 : "rooster",
2006 : "dog",
2007 : "pig",
2008 : "rat"}

>>> Cal[2008]
"rat"

>>> len(Cal)
4
Building the Huffman Tree!

<table>
<thead>
<tr>
<th>Letter</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>0.40</td>
</tr>
<tr>
<td>a</td>
<td>0.20</td>
</tr>
<tr>
<td>r</td>
<td>0.15</td>
</tr>
<tr>
<td>v</td>
<td>0.15</td>
</tr>
<tr>
<td>e</td>
<td>0.06</td>
</tr>
<tr>
<td>y</td>
<td>0.04</td>
</tr>
</tbody>
</table>

frequencies = { “h”: 0.40, “a”: 0.20, “r”: 0.15, “v”: 0.15, “e”: 0.06, “y”: 0.04 }

OBJECTIVE: Convert this into a tree...
Building the Huffman Tree!

```
frequencies = { "h": 0.40, "a": 0.20, "r": 0.15, "v": 0.15, "e": 0.06, "y": 0.04 }

def build_huffman_tree(frequencies):
    
    Assume a function min_frequency(frequencies) that returns the character (key) with min frequency!

    while len(frequencies) >= 2:
        return ???

    # Build the Huffman tree...

    return ???
```

![Huffman Tree Diagram](image)
Building the Huffman Tree!

```
frequencies = { "h": 0.40, "a": 0.20, "r": 0.15, "v": 0.15, "e": 0.06, "y": 0.04 }

def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        return

Assume a function minfrequency(frequencies) that returns the character (key) with min frequency!

def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        return
```

```python
return ???
```
Building the Huffman Tree!

frequencies = { "h": 0.40, "a": 0.20, "r": 0.15, "v": 0.15, "e": 0.06, "y": 0.04 }

Assume a function minfrequency(frequencies) that returns the character (key) with min frequency!

def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        low2 = minfrequency(frequencies)
        freq2 = frequencies[low2]
        del frequencies[low2]

    return ???

return ???
Building the Huffman Tree!

```python
frequencies = { "h": 0.40, "a": 0.20, "r": 0.15, "v": 0.15, "e": 0.06, "y": 0.04 }

def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        low2 = minfrequency(frequencies)
        freq2 = frequencies[low2]
        del frequencies[low2]
        frequencies[(low1, low2)] = freq1 + freq2
    return ???
```

Assume a function `minfrequency(frequencies)` that returns the character (key) with min frequency!
Building the Huffman Tree!

```
frequencies = { "h": 0.40, 
    "a": 0.20 , "r" : 0.15, 
    "v": 0.15 , "e" : 0.06, 
    "y": 0.04 }

def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree 
for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        low2 = minfrequency(frequencies)
        freq2 = frequencies[low2]
        del frequencies[low2]
        frequencies[(low1, low2)] = freq1 + freq2
    return frequencies.keys()[0]
```

Assume a function `minfrequency(frequencies)` 
that returns the character (key) with min frequency!
The Huffman Encoder

Read input file into string S
Count letter frequencies in S
Build the Huffman tree
Find the Huffman code for each character
binary_sequence = ""
for each character c in S
  binary_sequence += Huffman code for c
Write output to file

frequencies = { "h": 0.40, "a": 0.20, "r": 0.15, "v": 0.15, "e": 0.06, "y": 0.04 }

h: 1 a: 000 r: 001
v: 010 e: 0110 y: 0111
The Huffman Decoder

Read compressed file into string E
E = Huffman table + weird symbols
Expand E to original text string S
Save S to file

6
2001
h: 1  a: 000  r: 001
v: 010  e: 0110  y: 0111
$a!*&spam^>\n):^)
pen*guin!*blah/~.\cs5!.-42
blahblahblah
Two Other Data Compression Schemes…

• Lempel-Ziv-Welsh
  banana banana banana banana

• Arithmetic Coding
## Arithmetic Coding

<table>
<thead>
<tr>
<th></th>
<th>$2^{-1}$</th>
<th>$2^{-2}$</th>
<th>$2^{-3}$</th>
<th>$2^{-4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1. $1, 0, 0, 0 = 1/2$
2. $1, 1, 0, 0 = 3/4$
3. $0, 1, 0, 1 = 5/16$
Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Encode: “sps”

Hold on to your socks!
This could knock them off!
## Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Encode: “sps”

![Diagram](attachment:arithmetic_coding_diagram.png)
Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Encode: “sps”
Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Encode: “sps”
Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

What’s the code?
How do we decode?
How are fractions encoded in binary?

Encode: “sps”

0.011 = 1/4 + 1/8 = 0.375
Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.5</td>
</tr>
<tr>
<td>p</td>
<td>0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.1</td>
</tr>
<tr>
<td>p</td>
<td>0.9</td>
</tr>
</tbody>
</table>
Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.5</td>
</tr>
<tr>
<td>p</td>
<td>0.5</td>
</tr>
</tbody>
</table>

How many bits are used to encode “pp”?

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.1</td>
</tr>
<tr>
<td>p</td>
<td>0.9</td>
</tr>
</tbody>
</table>

How many bits are used to encode “pp” in this case?
### Arithmetic Coding

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.5</td>
</tr>
<tr>
<td>p</td>
<td>0.5</td>
</tr>
</tbody>
</table>

How many bits are used to encode “pp”?

<table>
<thead>
<tr>
<th>Letter</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.1</td>
</tr>
<tr>
<td>p</td>
<td>0.9</td>
</tr>
</tbody>
</table>

How many bits are used to encode “pp” in this case?
“I've designed a new compression algorithm called ‘Lai Compression’ that can compress any file bigger than 1000 characters by at least a little bit.”
Aargh!

Python has no Connect-four datatype…

… but we can correct that!
C4 Board class: methods

- the “constructor”  `__init__( self, width, height )`
- checks if allowed  `allowsMove( self, col )`
- places a checker  `addMove( self, col, ox )`
- removes a checker  `delMove( self, col )`
- outputs a string  `__repr__( self )`
- checks if any space is left  `isFull( self )`
- checks if a player has won  `winsFor( self, ox )`
- play (person vs. person)!  `hostGame( self )`

Which of these will require the most thought?
winsFor( self, ox )

b.winsFor( 'X' )
or 'O'

Thoughts?

corner cases?
Two-player games have been a key focus of AI as long as computers have been around…

In 1945, Alan Turing predicted that computers would be better chess players than people in ~ 50 years…

*and thus would have achieved intelligence.*

Alan Turing memorial
Manchester, England
Two-player games have been a key focus of AI as long as computers have been around…

…humans and computers have different relative strengths in these games.
How humans play games...

An experiment (by A. deGroot) was performed in which chess positions were shown to novice and expert players...

- experts could reconstruct these perfectly
- novice players did far worse...

Random chess positions (not legal ones) were then shown to the two groups

- experts and novices did equally well (badly) at reconstructing them!
The **Player** class

What data and methods are needed to construct and implement a Player object?

Let's see a demo!
Player

__init__(self, ox, tbt, ply)
__repr__(self)
oppCh(self)
scoreBoard(self, b)
scoresFor(self, b)
tiebreakMove(self, scores)
extMove(self, b)
Assigns a score to any board, $b$

A simple system:

- $100.0$ for a win
- $50.0$ for anything else
- $0.0$ for a loss
Looking further ahead …

0 ply: *Zen* choice of move: *here and now*

1 ply: 1 ply: 1 ply:

X’s move

| | | | | | | |
| | | | | | | |
| | | | | | | |
|X| | | | | | |
|O|X|X|X|O|O| |

0 1 2 3 4 5 6

2 ply: 2 ply: 2 ply:

X’s move

| O | | | | | | |
| X | | | | | | |
|X|O|O|O|X| | |
|O|X|X|O|X|O| |

0 1 2 3 4 5 6

3 ply: 3 ply: 3 ply:

X’s move

| | | | | | | |
| | | | | | | |
| | | | |X| | |
|X|X| |X|O|O| |
|O|X|O| |O|X| |

0 1 2 3 4 5 6

(1) **Player** will win

(2) **Player** will avoid losing

(3) **Player** will set up a win by forcing the opponent to avoid losing
Looking *further* ahead…

`scoreBoard` looks ahead 0 moves

The "Zen" approach -- we are excellent at this!

If you look one move ahead, how many possibilities are there to consider?

<table>
<thead>
<tr>
<th>0-PLY</th>
<th>1-PLY</th>
<th>2-PLY</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scoresFor(self, b)</code> returns a LIST of scores, one for each column you can choose to move next…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(0) Suppose you're playing at 3 ply...

(1) For each of your possible moves

(2) Add it to the board

(3) Ask OPPONENT its scoresFor each board!

At what ply?

scoresFor's idea

What score will the opponent choose?

these are all of the opponent's calls to scoresFor!
(0) Suppose you're playing at 3 ply...

(1) For each of your possible moves

(2) Add it to the board

(3) Ask OPPONENT its scoresFor each board!

At what ply?

max(S) = 0

max(S) = 100

max(S) = 50

These are all of the opponent's calls to scoresFor!
It is O’s move. What scores does a 1-ply lookahead for O assign to each move?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>O</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Which change at 2-ply?

It is X’s move. What scores does a 2-ply lookahead for X assign to each move?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>O</td>
<td></td>
<td>O</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>O</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Which change at 3-ply?
Computer Chess

Computers cut their teeth playing chess...

- **Ranking**: 
  - **Beginner**: 500
  - **Amateur**: 1200
  - **World ranked**: 2000
  - **World champion**: 2800

**First paper**: 1950

**Early programs** ~ 1960’s

- **MacHack (1100)** ~ 1967 MIT
- **Slate (2070)** ~ 1970’s Northwestern
- **Deep Thought** ~ 1989 Carnegie Mellon
- **Deep Blue** ~ 1996 IBM
- **Deep Blue rematch** ~ 1997 IBM

**First Paper**

- **Deep Fritz**: 2002
- **X3D Fritz**: 2003
- **Hydra**: 2006

- **100’s of moves/sec**
- **10,000’s of moves/sec**
- **1,000,000’s moves/sec**
- **3,500,000 moves/sec**
- **200,000,000 moves/sec**