Special Topics

CS 42

December 6, 2011

Use-It-Or-Lose-It

The Packing Problem

```
>>> 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):
```

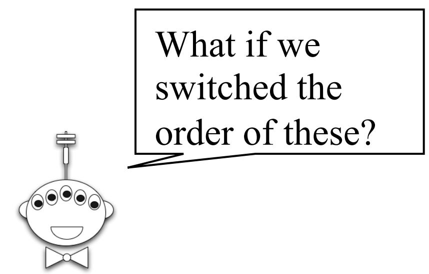
Try *THIS* without recursion!?

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
```



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:])
```

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…

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

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

```
def knapsack(capacity, WVList):
```

Worksheet and Demo

The Knapsack Problem…

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

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

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

Worksheet and Demo

The Knapsack Problem…

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

```
>>> 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]
```

Worksheet and Demo

The Knapsack Problem…

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

```
>>> 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:
```

Worksheet and Demo

The Knapsack Problem…

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

```
>>> 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:
```

Worksheet and Demo

The Knapsack Problem...

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

```
>>> 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:])
```

Worksheet and Demo

The Knapsack Problem…

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

```
>>> 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:])
```

Worksheet and Demo

The Knapsack Problem...

Item	Weight	Value
Spam	2	100
Tofu	3	112
Chocolate	4	125

Knapsack Capacity: 5? 6? 7?

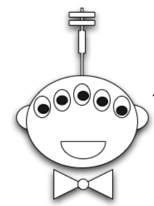
```
>>> knapsack(7, [ [2, 100], [3, 112], [4, 125] ])
237
```

```
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!

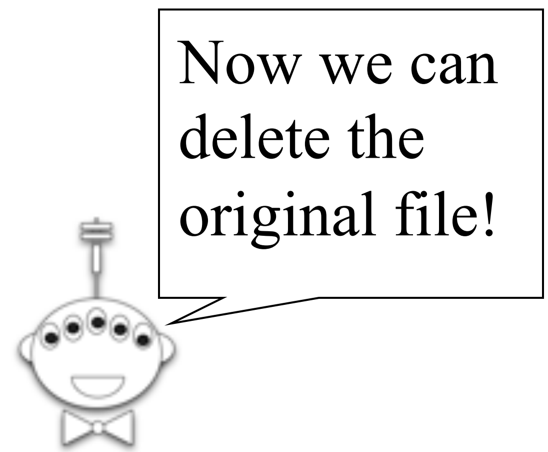
The order in which the subsets are presented is unimportant but within each subset, the order should be consistent with the input set.

Data Compression

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

compression algorithm (e.g. zip) →

**B6^9)=\n%
%spam!
=&&penguin/
?’ ,/+**



TEXT FILE
zzyzva.txt

58,254 bytes

TEXT FILE
zzyzva.txt.Z

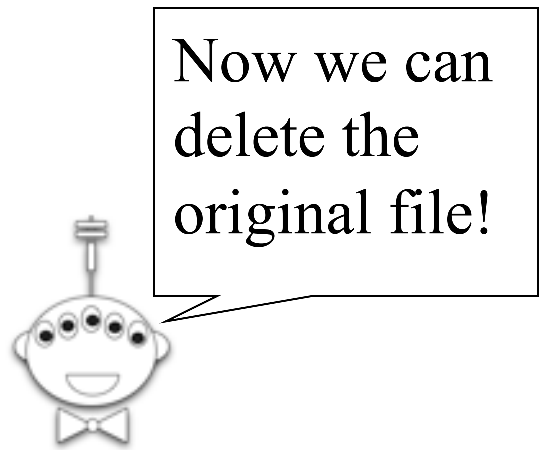
23,124 bytes

Data Compression

B6^9)=\n%
%spam!
=&&penguin/
?’ ,/+

TEXT FILE
zzyzva.txt.Z

23,124 bytes

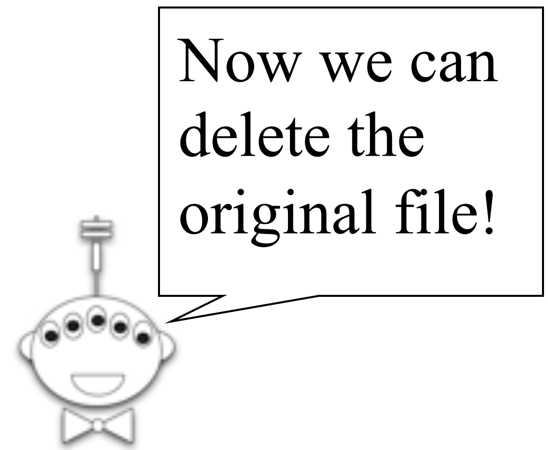


Data Compression

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

DEcompression algorithm (e.g. unzip)

B6^9)=\n%
%spam!
=&&penguin/
?’ ,/+



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…

TEXT FILE

Letter	ord(Letter)	Binary
T	84	01010100
h	104	01101000
e	101	01100101
z	122	01111010

- " "- 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

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

Variable Length Encodings

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

TEXT FILE

Letter	frequency	Binary code
z	0.25	0
y	0.10	1
x	0.09	00
a	0.08	01
…		
r	0.02	10100111100
p	0.01	10100111101

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

Cute idea, but what's the problem here?

The Prefix Property

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

TEXT FILE

Letter	frequency	Binary code
z	0.25	00
y	0.10	01
x	0.09	10
a	0.08	111
r	0.02	1100

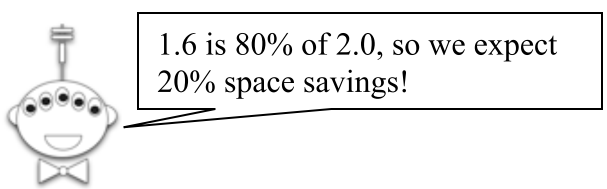
101110100001100 = 10 111 01 00 00 1100

Consider the Language "Spamish" which has only four letters in its alphabet…

Letter	freq	Fixed Length	Variable Length
s	0.6	00	0
p	0.2	01	10
a	0.1	10	110
m	0.1	11	111

Expected average number of bits per symbol = 2

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



The Variable Length Coding Problem…

Letter	Frequency
a_1	freq(a_1)
a_2	freq(a_2)
a_3	freq(a_3)
…	
a_n	freq(a_n)

These frequencies are from the specific file that we're planning to compress!!

Objective: Find a binary prefix code that minimizes…

The Variable Length Coding Problem…

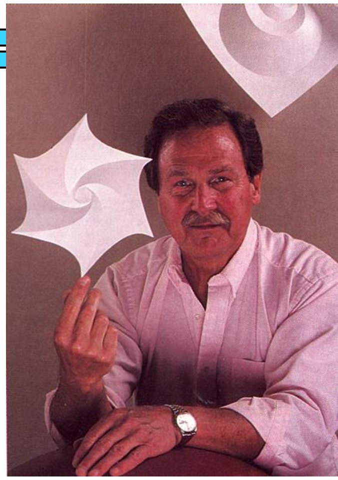
Letter	Frequency
a_1	$freq(a_1)$
a_2	$freq(a_2)$
a_3	$freq(a_3)$
…	
a_n	$freq(a_n)$

These frequencies are from the specific file that we're planning to compress!!

Objective: Find a binary prefix code that minimizes…

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

The David Huffman Story!



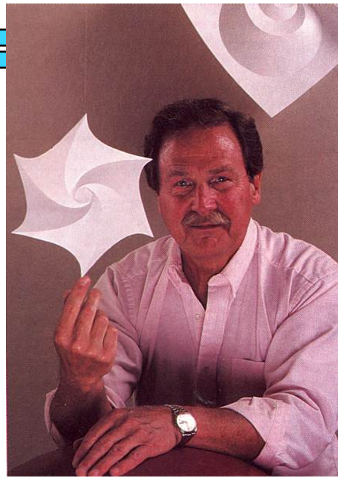
map smppam
ssampamsmam
…

TEXT FILE

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

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
ssampamsmam
…

TEXT FILE

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

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!

Letter	Frequency
h	0.40
a	0.20
r	0.15
v	0.15
e	0.06
y	0.04

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!

Letter	Frequency
h	0.40
a	0.20
r	0.15
v	0.15
e	0.06
y	0.04

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

How do I get these!?

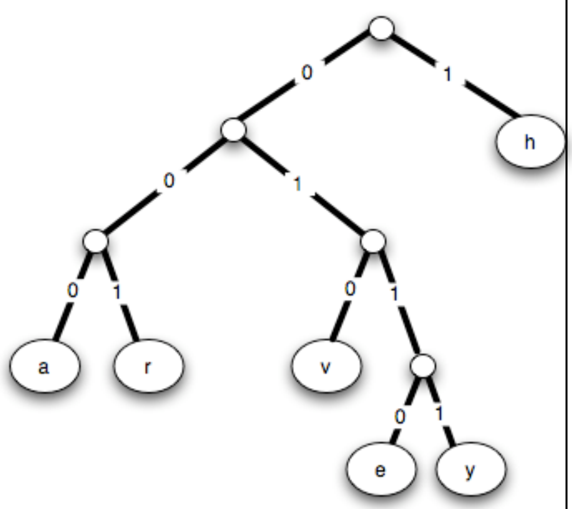
OBJECTIVE: Convert this into a tree…

Building the Huffman Tree!

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

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

(((a, r) , (v, (e, y))) , h)



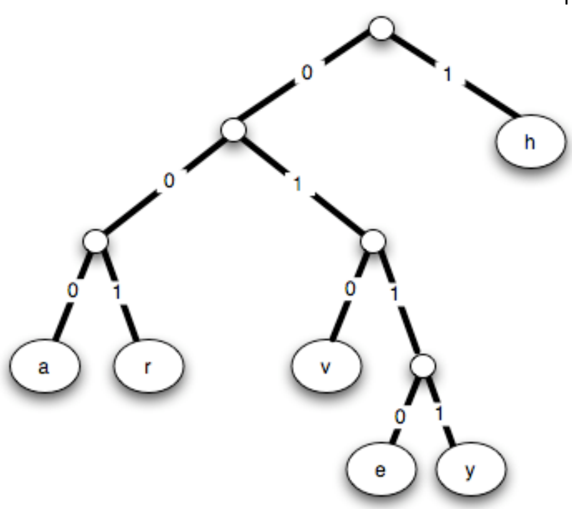

```
def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree
    for the given frequency dictionary. """
    while len(frequencies) >= 2:



    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!

(((a, r) , (v, (e, y))) , h)



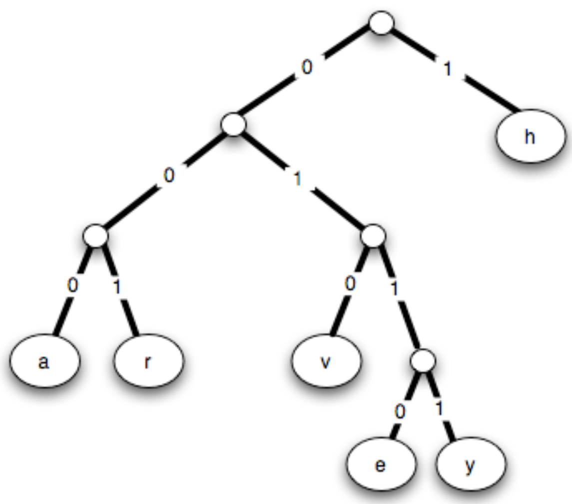

```
def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree
    for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]




    return ???
```

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!

(((a, r) , (v, (e, y))) , **h**)



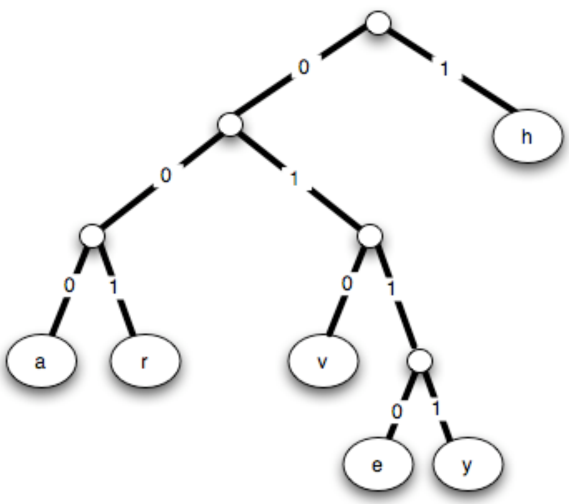

```
def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree
    for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        low2 = minfrequency(frequencies)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



    return ???
```

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!

(((a, r) , (v, (e, y))) , h)



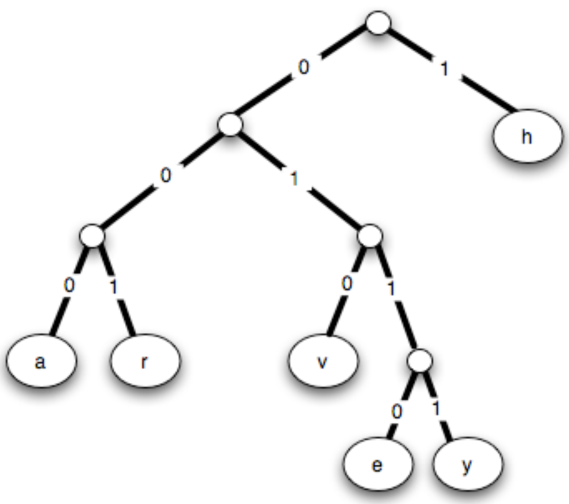

```
def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree
    for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        low2 = minfrequency(frequencies)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

        frequencies[(low1, low2)] =
                 freq1 + freq2

    return ???
```

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!**

(((a, r) , (v, (e, y))) , h)

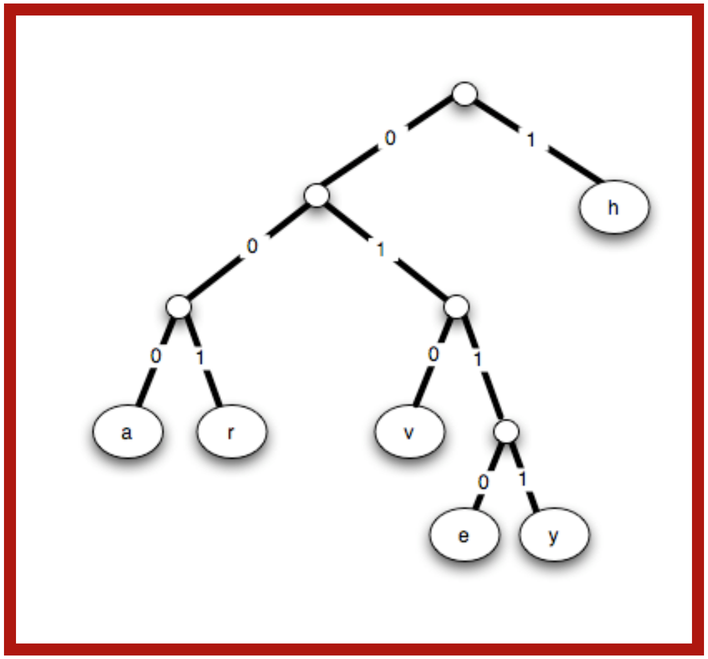



```
def build_huffman_tree(frequencies):
    """Returns a tuple representing the Huffman tree
    for the given frequency dictionary. """
    while len(frequencies) >= 2:
        low1 = minfrequency(frequencies)
        freq1 = frequencies[low1]
        del frequencies[low1]
        low2 = minfrequency(frequencies)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

        frequencies[(low1, low2)] =
                freq1 + freq2

    return frequencies.keys()[0]
```

The Huffman Encoder

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

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



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

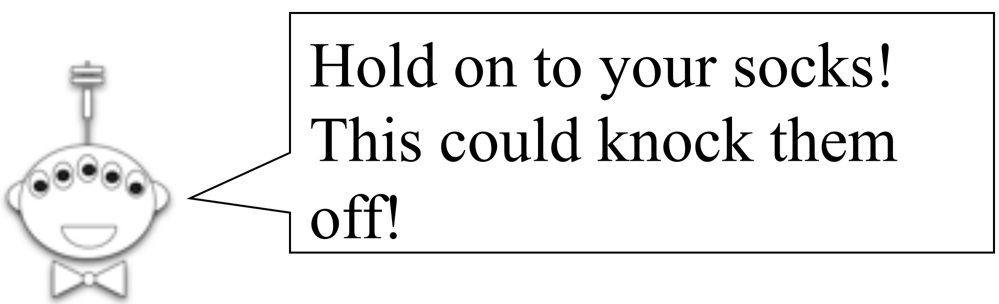
Arithmetic Coding

2^{-1}	2^{-2}	2^{-3}	2^{-4}		
1	0	0	0	=	1/2
1	1	0	0	=	3/4
0	1	0	1	=	5/16

Arithmetic Coding

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

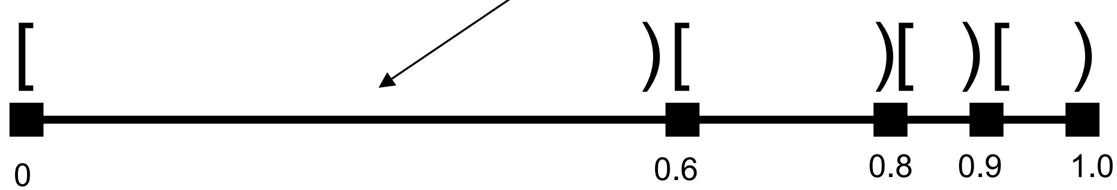
Encode: “sps”



Arithmetic Coding

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

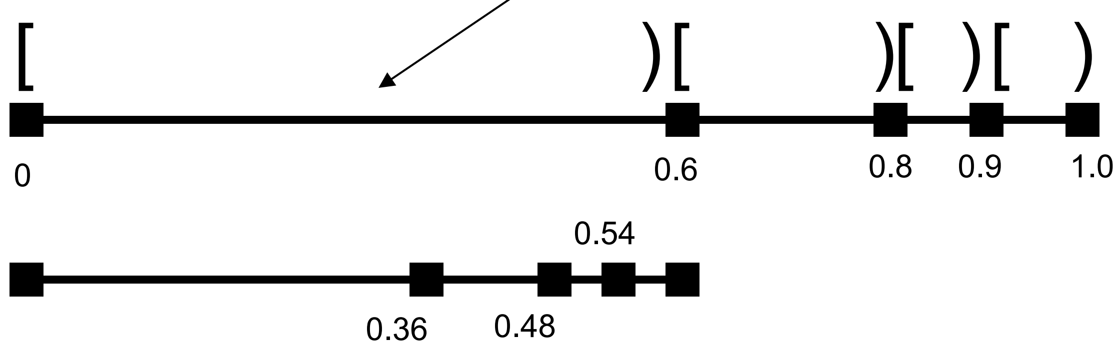
Encode: “sps”



Arithmetic Coding

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

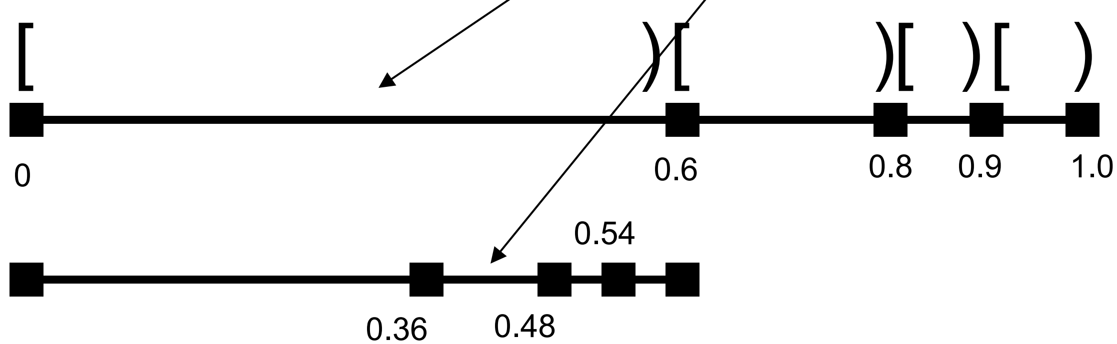
Encode: "sps"



Arithmetic Coding

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

Encode: “sps”

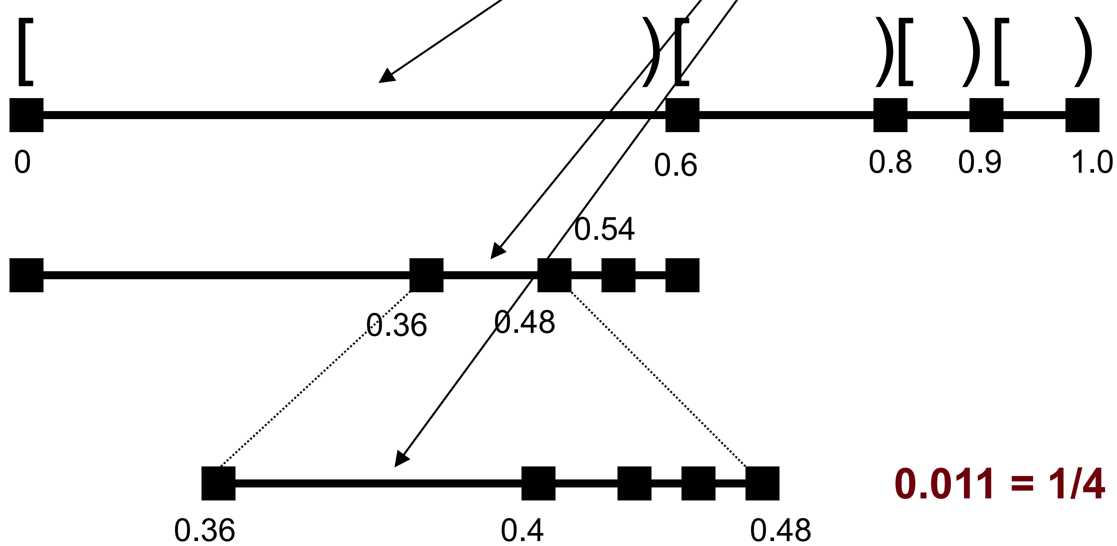


Arithmetic Coding

Letter	freq
s	0.6
p	0.2
a	0.1
m	0.1

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

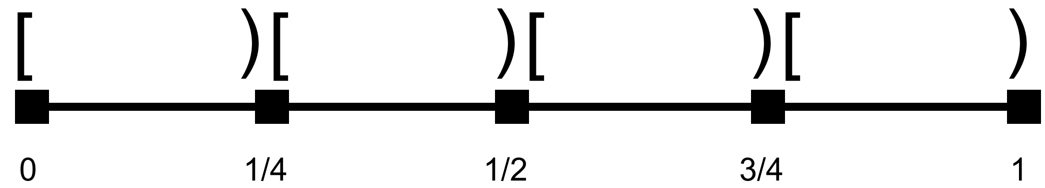
Encode: “sps”



0.011 = 1/4 +1/8 = 0.375

Arithmetic Coding

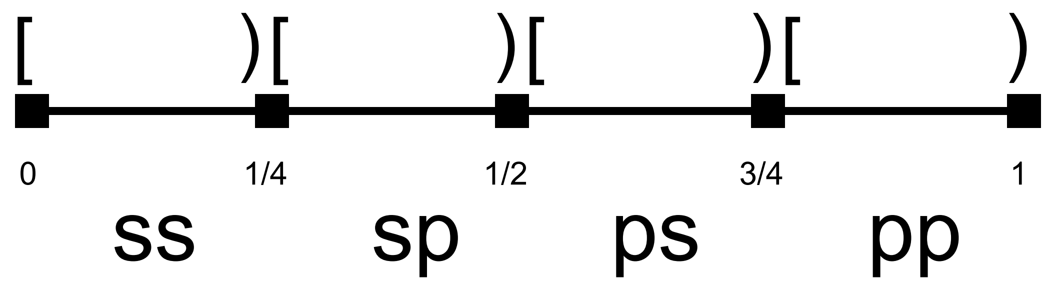
Letter	freq
s	0.5
p	0.5



Letter	freq
s	0.1
p	0.9

Arithmetic Coding

Letter	freq
s	0.5
p	0.5



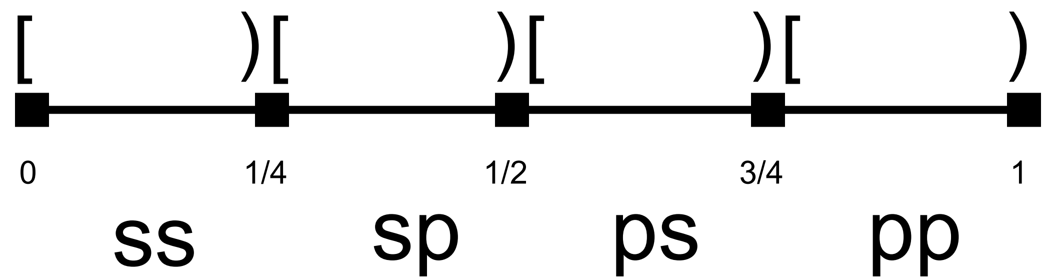
How many bits are used to encode “pp” ?

Letter	freq
s	0.1
p	0.9

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

Arithmetic Coding

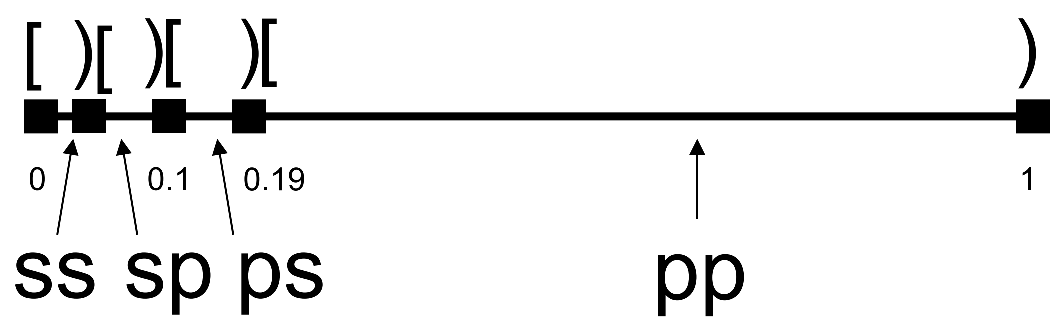
Letter	freq
s	0.5
p	0.5



How many bits are used to encode "pp" ?

Letter	freq
s	0.1
p	0.9

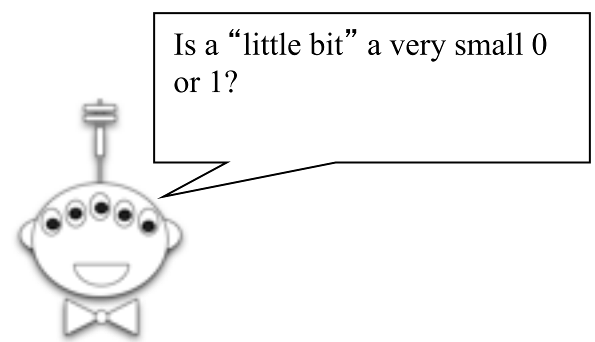
How many bits are used to encode "pp" in this case?



Prof. I. Lai's Lecture

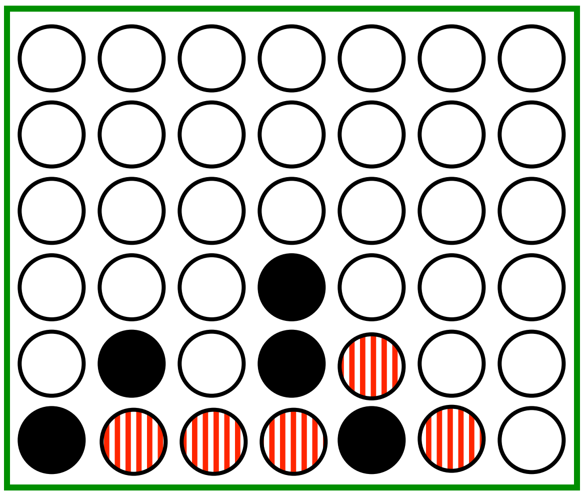
Prof. I. Lai
P.I.T.

"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…



```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |X| | | |
| |X| |X|O| | |
|X|O|O|O|X| |O|
---------------
 0 1 2 3 4 5 6
```

… but we can correct that!

Can I see a demo?

C4 **`Board`** class: methods

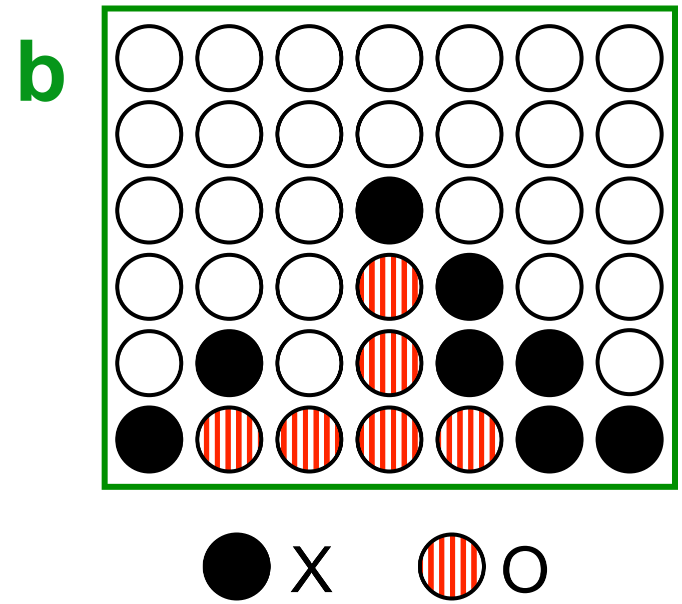
the "constructor"	`__init__( self, width, height )`
checks if allowed	`allowsMove( self, col )`
places a checker	`addMove( self, col, ox )`
removes a checker	`delMove( self, col )`
outputs a string	`__repr__( self )`
checks if any space is left	`isFull( self )`
checks if a player has won	`winsFor( self,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

Which of these will require the most thought?

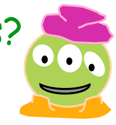
winsFor(self, ox)

```
b.winsFor( 'X' )
```

or 'O'



Thoughts?



Strategic thinking $\stackrel{?}{==}$ intelligence

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.

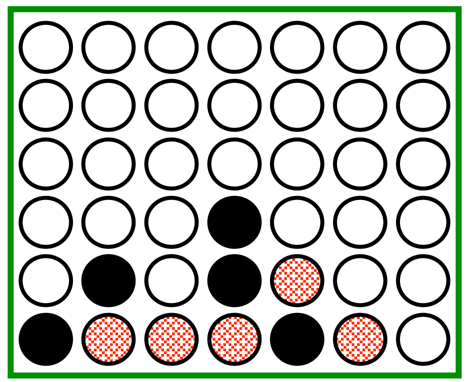
Alan Turing memorial
Manchester, England

Strategic thinking $\stackrel{?}{==}$ intelligence

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

computers

good at looking ahead in the game to find winning combinations of moves



humans

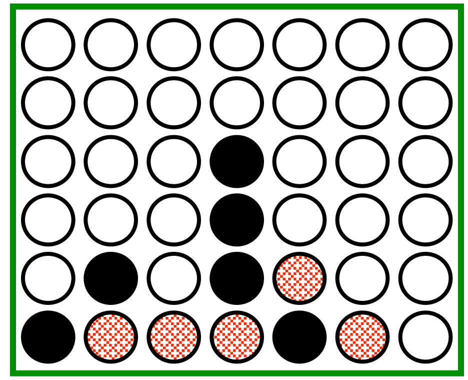
good at evaluating the strength of a board for a player

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

How humans play games…

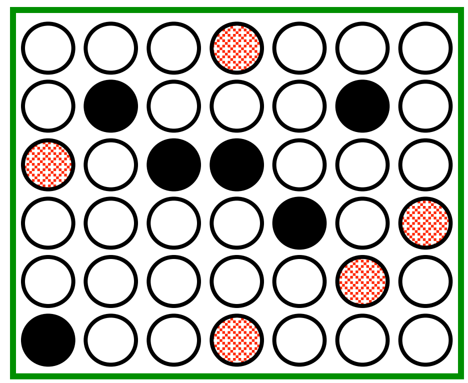
An experiment (by A. deGroot) was performed in which chess positions were shown to novice and expert players…

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

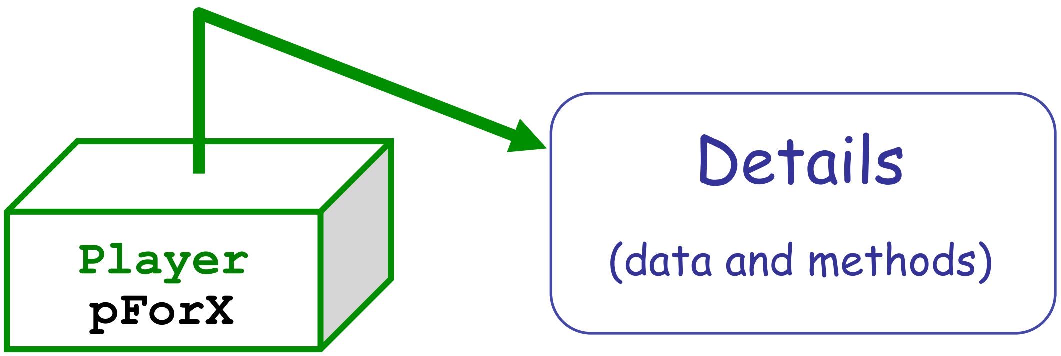


Random chess positions (not legal ones) were then shown to the two groups

- experts and novices did equally well (badly) at reconstructing them!



The **Player** class



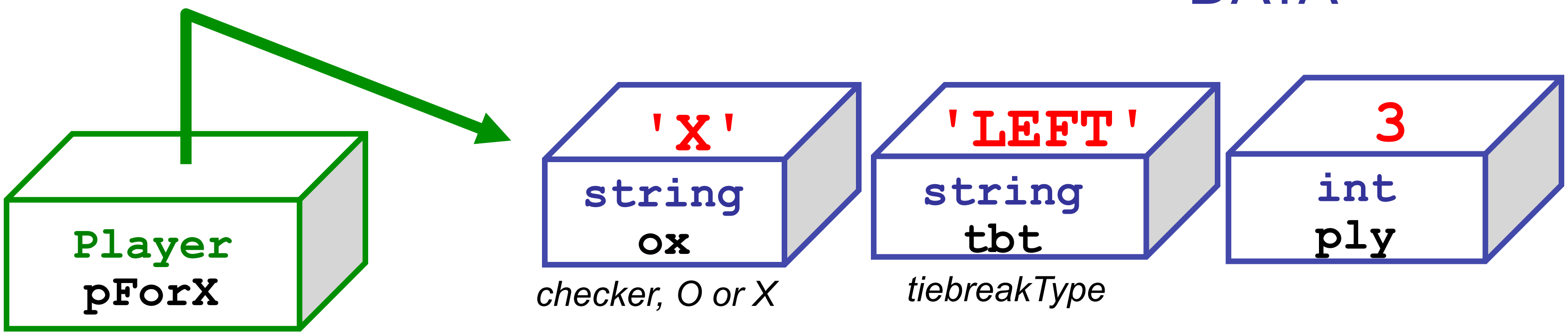
What data and methods are needed to construct and implement a Player object?

Let's see a demo!

`Player`

Picture of a **`Player`** object

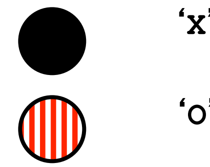
DATA




```
__init__(self, ox, tbt, ply)

__repr__(self)

oppCh(self)

scoreBoard(self, b)

scoresFor(self, b)

tiebreakMove(self, scores)

nextMove(self, b)
```

METHODS

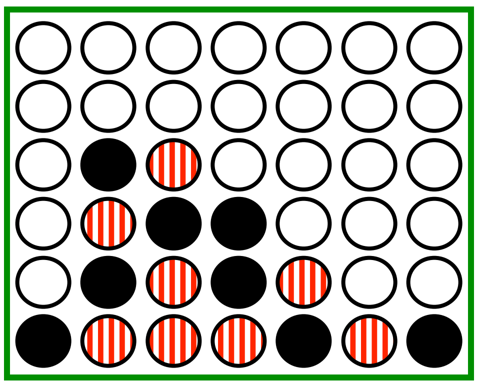
`scoreBoard`



Assigns a *score* to any board, **`b`**

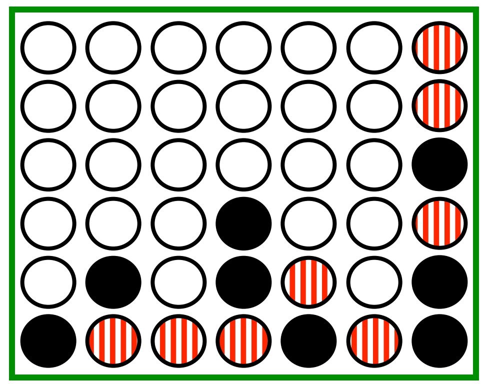
A simple system: →

100.0	**50.0**	**0.0**
for a win	**for anything else**	**for a loss**



Score for

Score for



Score for

Score for

Looking further ahead …

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

1 ply:
X's move

```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| |O|X| | | | |
|O|X|X|X| |O|O|
---------------
 0 1 2 3 4 5 6
```

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:
X 's move

```
| | | | | | | |
| | | | | | | |
| | | | |X| | |
| | | | |O|O| |
| |X|X| |X|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!

0-ply

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

1-ply

2-ply

`scoresFor( self, b )` returns a LIST of scores, one for each column you can choose to move next…

`scoresFor`'s idea

(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!

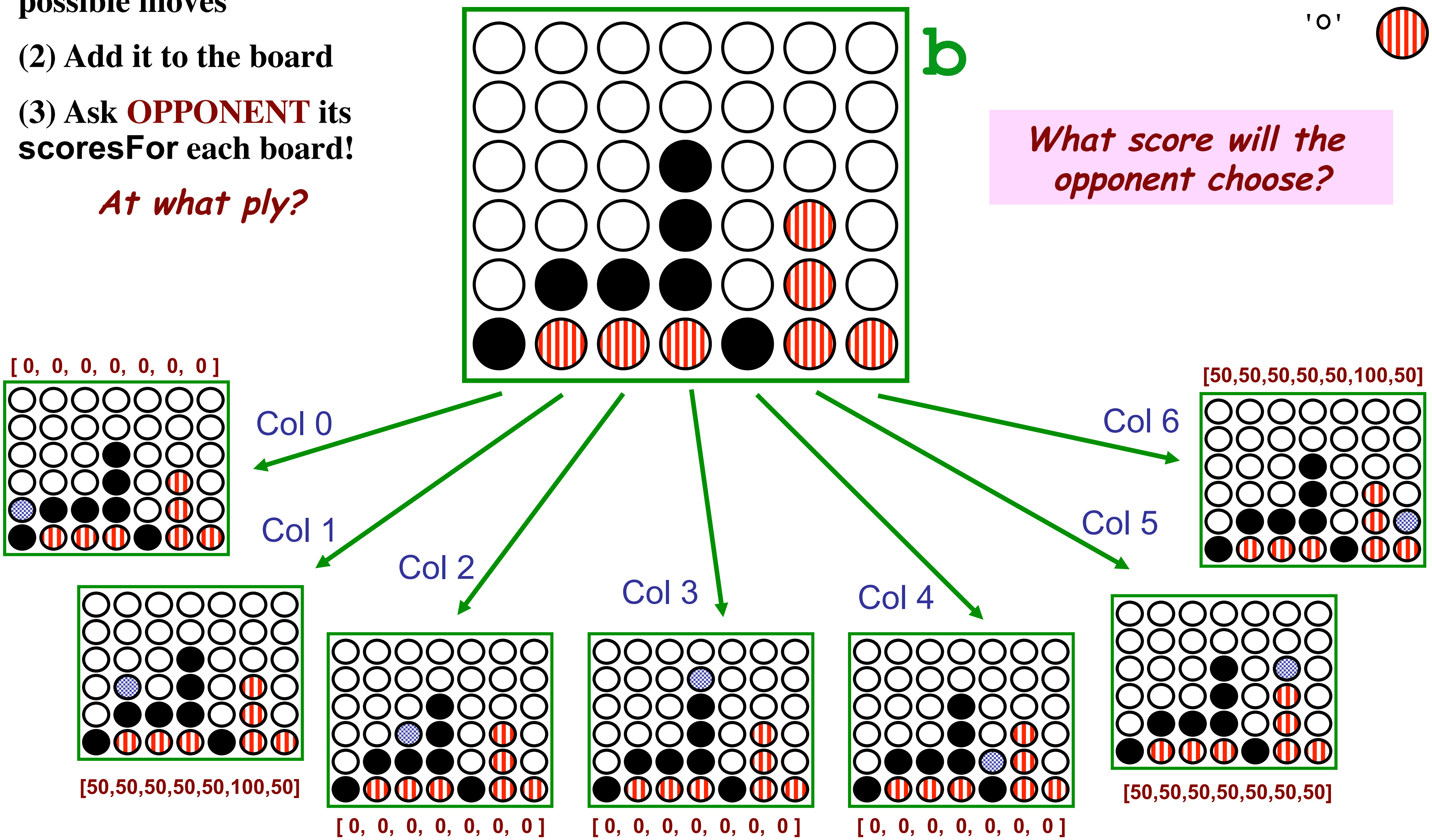
At what ply?

(self) 'X'

new 'X'

'O'

What score will the opponent choose?



these are all of the opponent's calls to `scoresFor` !

scoresFor's idea

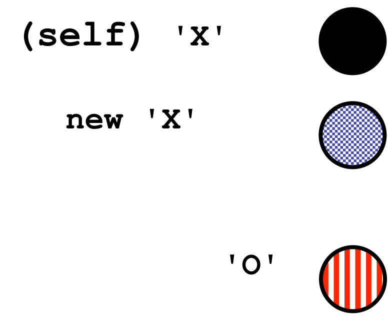
(0) Suppose you're playing at 3 ply...

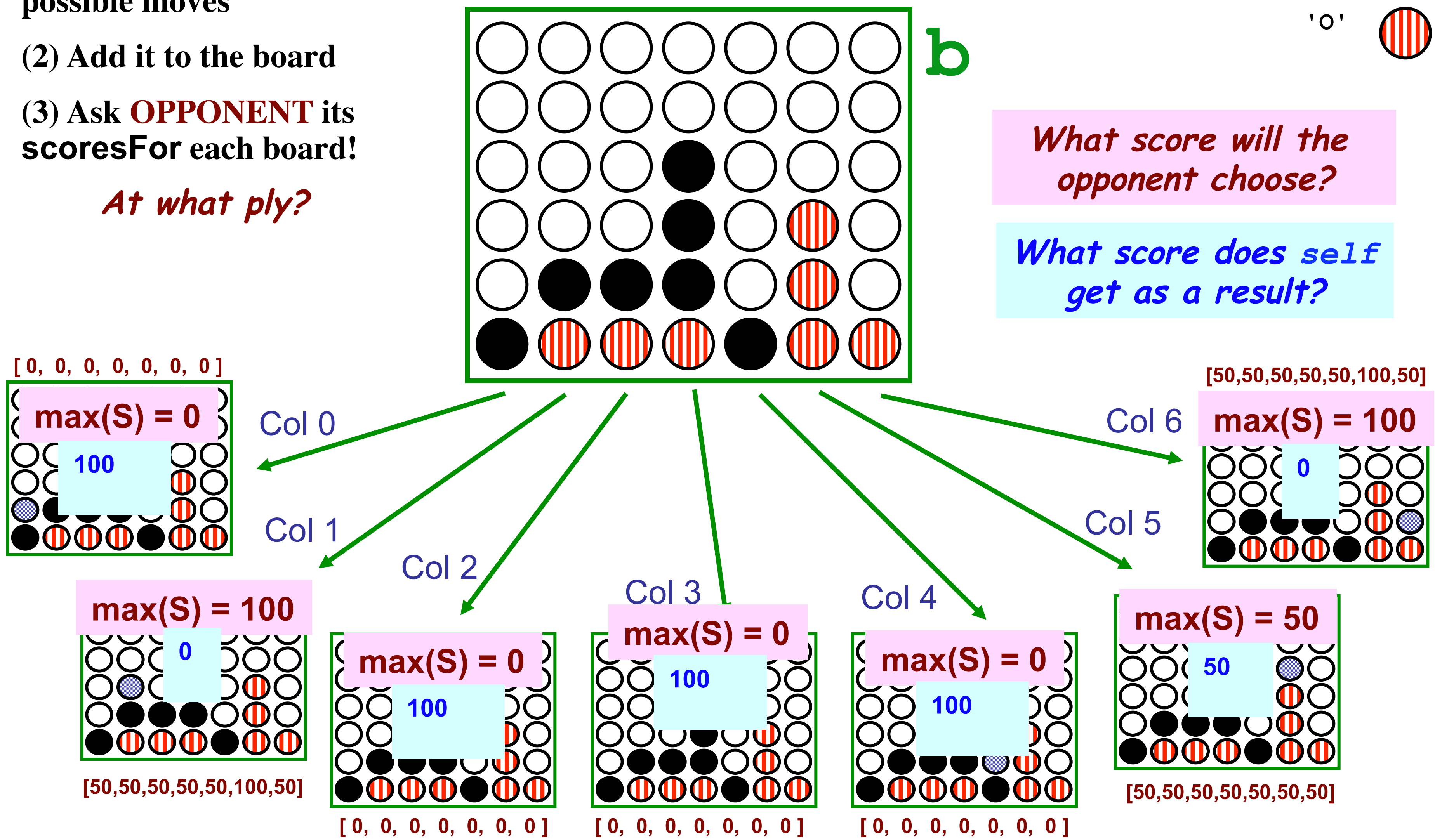
(1) For each of your possible moves

(2) Add it to the board

(3) Ask **OPPONENT** its scoresFor each board!

At what ply?





these are all of the opponent's calls to `scoresFor` !

Example 1-ply and 2-ply lookahead scores

```
|O| | | | | | |
|X| | | |O| |X|
|O| | | |X|O|X|
|X| | | |O|O|X|
|X| |X| |X|O|O|
|X| |O|O|O|X|X|
---------------
 0 1 2 3 4 5 6
```

It is O's move. What scores does a 1-ply lookahead for O assign to each move?

col 0	col 1	col 2	col 3	col 4	col 5	col 6

Which change at 2-ply?

```
| | | | | | |O|
| | | | | | |O|
| | | | | | |X|
|X| |X|O| | |O|
|X|O|O|X| |X|X|
|X|O|O|O| |O|X|
---------------
 0 1 2 3 4 5 6
```

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

col 0	col 1	col 2	col 3	col 4	col 5	col 6

Which change at 3-ply?

Computer Chess

Computers cut their teeth playing chess…

Ranking

first paper: 1950

beginner

500

early programs ~ 1960's

100's of moves/sec

amateur

1200

MacHack (1100) ~ 1967 MIT

10,000's of moves/sec

1,000,000's moves/sec

world ranked

2000

Slate (2070) ~ 1970's Northwestern

Deep Thought ~ 1989 Carnegie Mellon

Deep Blue ~ 1996 IBM

world champion

2800

Deep Blue rematch ~ 1997 IBM

3,500,000 moves/sec

Deep Fritz: 2002

X3D Fritz: 2003

Hydra: 2006

200,000,000 moves/sec