CSC148: Week 10
http://www.cdf.utoronto.ca/~csc148h/summer/

Sophia Huynh
Summer 2018
Announcements

● Assignment 1 remarks due Monday
● Midterm 2 next week in EX100
  ○ Class after the midterm @ 7:30!
● Past midterms + Practice posted
  ○ Solutions for practice on Monday
● Exercise 6 due tomorrow night
● Exercise 7 and Lab 7 are up
Office Hours

- Friday: 12 - 6PM (BA5287)
- Monday: 2 - 6PM (BA5287)
- Tuesday: 11 - 1 PM (BA5287)
- Wednesday: 5 - 7 PM (BA3201)
- Thursday: 1 - 5 PM (BA3219)
Outline

● Binary Trees
  ○ Jacqueline will take over things for a bit during a break
● Assignment 2
● Midterm
**LinkedLists**
Points to 1 element.

**Trees**
Points to $n$ elements.
This week: Binary Trees

Points to 2 elements (at most).
This week: Binary Trees

Points to 2 elements (at most).
def __init__(self,
    value,
    next_ = None):
    self.value = value
    self.next_ = next_
**LinkedLists**

```python
def __init__(self, value, next_=None):
    self.value = value
    self.next_ = next_
```

**Trees**

```python
def __init__(self, value, children=None):
    self.value = value
    self.children = children[:][::] if children else []
```
def __init__(self, value,
    left = None,
    right = None):

    self.value = value
    self.left = left
    self.right = right
def __init__(self, value,
    left = None,
    right = None):
    self.value = value
    self.left = left
    self.right = right
BinaryTree

def __init__(self, value,
    left = None,
    right = None):

    self.value = value
    self.left = left
    self.right = right

Multiple children/subtrees (like with Trees)
get_values()

- **Function** that takes in a BinaryTree or None
- Returns a list of the values in **pre-order**
  - Root first, then children.
def get_values(lnk):
    if lnk is None:
        return []
    return [lnk.value] +
    get_values(lnk.next_)

Base case!
def get_values(t):
    if t is None:
        return []
get_values() for BinaryTrees

def get_values(t):
    if t is None:
        return []

    left_vals = get_values(t.left)
    right_vals = get_values(t.right)

Recursive calls!
Example
Example

[3, 1, 7, 0, 4]

```
[3, 1, 7, 0, 4]
```

```
8
```

```
3
```

```
1
```

```
7
```

```
0
```

```
4
```

```
6
```

```
9
```

```
2
```

```
5
```
Example

[3, 1, 7, 0, 4] 8 [6, 9, 2, 5]
Example

[8, 3, 1, 7, 0, 4, 6, 9, 2, 5]

[3, 1, 7, 0, 4]

[6, 9, 2, 5]

[3, 1, 7, 0, 4]

[6, 9, 2, 5]
Example

\[ [8, 3, 1, 7, 0, 4, 6, 9, 2, 5] \]

\[ [3, 1, 7, 0, 4] \]  \[ 8 \]  \[ [6, 9, 2, 5] \]

```
[3, 1, 7, 0, 4]

3
  1
    7
    0
  4

6
  9
    2
    5
```
Example

[8, 3, 1, 7, 0, 4, 6, 9, 2, 5]

[3, 1, 7, 0, 4]

[6, 9, 2, 5]
Example

\[ [8, 3, 1, 7, 0, 4, 6, 9, 2, 5] \]

\[ [3, 1, 7, 0, 4] \]

\[ [6, 9, 2, 5] \]
get_values() for BinaryTrees

def get_values(t):
    if t is None:
        return []
    left_vals = get_values(t.left)
    right_vals = get_values(t.right)
    return [t.value] + left_vals + right_vals

Put them together in one list.
get_values() for BinaryTrees

def get_values(t):
    if t is None:
        return []
    return [t.value] +
    get_values(t.left) +
    get_values(t.right)
count_occurrences

- Function that takes in a BinaryTree (or None) and a value
- Return how many times that value occurs in the BinaryTree
count_occurrences

def count_occurrences(t, value):
    if t is None:
        return 0

Base case!
Example: t

```
4
 /  \
3   4
 / \
1   4
   / \
  7   4
   / \
  4   4
 /  \
9   4
```
count_occurrences(t.left, 4) returns?

A) 0  B) 1  C) 2  D) 3
Answer:

C) 2
count_occurrences(t.left, 4) returns?

A) 0  B) 1  C) 2  D) 3
count_occurrences(t.right, 4) returns?

A) 0  B) 1  C) 2  D) 3
Answer:
D) 3
count_occurrences(t.right, 4) returns?

A) 0  B) 1  C) 2  D) 3
count_occurrences(t, 4)
count_occurrences(t, 4)
count_occurrences(t, 4)
count_occurrences(t, 4)

Want: 6
2 + 3 + 1
def count_occurrences(t, value):
    if t is None:
        return 0
    count_left =
        count_occurrences(t.left, value)
\textbf{count\_occurrences}

\begin{verbatim}
count_left = count_occurrences(t.left, value)
count_right = count_occurrences(t.right, value)
\end{verbatim}

Recursive call on right
count_right = count_occurrences(t.right, value)

count_current = 1 if self.value == value else 0
count_occurrences

count_current = 1 if
    self.value == value else 0

return count_current +
    count_left +
    count_right
contains

- Given a BinaryTree (or None) and a value
- Return whether the BinaryTree contains that value or not
contains(t, 2)
contains(t, 2) is True
def contains(t, value):

- Base case
- Recursive step
  - What are we making recursive calls on?
  - What do we expect back for some example?
  - What do we do with the results of the recursive calls?
def contains(t, value):
    if t is None:
        return False
def contains(t, value):
    if t is None:
        return False
    if t.value == value:
        return True
def contains(t, value):
    if t is None:
        return False
    if t.value == value:
        return True

Return True if one of the subtrees contain value.
def contains(t, value):
    if t is None:
        return False
    if t.value == value:
        return True
    return contains(t.left, value) or contains(t.right, value)
def contains(t, value):
    if t is None:
        return False
    if t.value == value:
        return True
    return contains(t.left, value) or contains(t.right, value)
get_height(t)

- Return the height of our BinaryTree

get_height(BinaryTree(3)) == 1
get_height(None) == 0
def get_height(t)

● Return the height of our BinaryTree
● Base case?
● Recursive step?
  ○ What recursive calls do you make?
  ○ What do you do with the results of those recursive calls?
def get_height(t)

if t is None:
    return 0

Base case!
def get_height(t)
if t is None:
    return 0

Recursive calls to left and right subtrees.
Expected: 2
Expected: 2

6  
9  2  5

Expected: 1
Want: 3
Include our node as 1 more level!

Expected: 2

6
9
2

5

Expected: 1
def get_height(t):
    if t is None:
        return 0

    Take the max of the heights of the subtrees and add 1.
```python
def get_height(t):
    if t is None:
        return 0
    return max(get_height(t.left),
               get_height(t.right))
    + 1
```
Break

Or we'll have a break after the next example.
Undecided.
Arithmetic Trees

\[
\begin{align*}
\text{Root: } & + \\
\text{Left child: } & - \\
\text{Right child: } & * \\
\text{Left child of left: } & 3 \\
\text{Left child of right: } & 1 \\
\text{Right child of left: } & 7 \\
\text{Right child of right: } & 1 \\
\text{Root of right: } & + \\
\text{Left child of right: } & 4 \\
\text{Right child of right: } & 2
\end{align*}
\]
Arithmetic Trees

Tells us what to do with the left and right subtrees
Arithmetic Trees

Internal nodes are always operators

Leaves are always numbers
Arithmetic Trees

Left - Right
3 - 1 == 2
Arithmetic Trees
Arithmetic Trees

7 + 1

7 + 1

+  

7 1

*  

/  

4 2
Arithmetic Trees

7 + 1

* 

4 / 2

+ 

7

1

4

2
Arithmetic Trees

(7 + 1) * (4 / 2)

7 + 1

7 + 1

7

4 / 2

4

2
get_result(t)

- Takes in an arithmetic BinaryTree
- Return the results of evaluating it.
  - If it's a leaf: Just return its value
    - Leaf: both t.left and both t.right are None
  - Use if-statements to handle operators
def get_result(t)
if t.left is None and t.right is None:
    return t.value

Base case: If t is a leaf, return its value.
Arithmetic Trees

get_result(t.left) = 8

get_result(t.right) = 2

7 + 1 * 2 / 4 = 8
def get_result(t)
if t.left is None and
t.right is None:
    return t.value

left_result = get_result(t.left)
right_result = get_result(t.right)
def get_result(t):

left_result = get_result(t.left)
right_result = get_result(t.right)

if t.value == '+':
    return left_result +
    right_result
def get_result(t):

left_result = get_result(t.left)
right_result = get_result(t.right)

if t.value == '-':
    return left_result - right_result
def get_result(t):

    left_result = get_result(t.left)
    right_result = get_result(t.right)

    if t.value == '*':
        return left_result * right_result
def get_result(t):
left_result = get_result(t.left)
right_result = get_result(t.right)

if t.value == '/':
    return left_result / right_result
Classification Tree

- Internal nodes: boolean functions
  - Takes a value and returns True/False
- If True is returned, go to the left subtree
- If False is returned, go to the right subtree
- Return the value of the leaf
Classification Tree

- True
  - < 65
    - False
      - > 300
        - Senior
      - < 20
        - True
          - < 65
          - < 20
            - < 3
              - Baby
            - < 12
              - Child
          - => 300
            - Immortal
            - Senior
        - Adult
      - => 300
        - Senior
Creating a node

def is_less_than_65(value):
    return value < 65

Make the function that we want.
def is_less_than_65(value):
    return value < 65

root = BinaryTree(is_less_than_65,  # Pass the function name as the value for our BinaryTree.
                  left = ...,  # ...
                  right = ...  # ...
)
Creating a node

child = BinaryTree("Child")
Creating a node

```
child = BinaryTree("Child")

teenager = BinaryTree("Teenager")
```
def is_less_than_12(value):
    return value > 12

some_node = BinaryTree(is_less_than_12,
                        child, teenager)

child =
    BinaryTree("Child")

teenager =
    BinaryTree("Teenager")

Creating a node
Calling on the function

```python
some_node = BinaryTree(is_less_than_12, child, teenager)
```

To call the function:

```python
some_node.value(14)
```
Calling on the function

To call the function:

```python
some_node.value(14)
```

```python
some_node.value == is_less_than_12
```
Calling on the function

To call the function:

```
is_less_than_12(14)
some_node.value ==
is_greater_than_12
```
get\_classification(t, to\_classify)
get_classification(t, to_classify)

value: 30

< 65

< 20

< 3 Baby

< 12 Child

> 300 Immortal Senior

< 12 Teenager

Adult
get_classification(t, to_classify)

value: 30
get_classification(t, to_classify)

value: 30

< 3

< 12

< 20

< 65

> 300

Adult

Immortal

Senior

Baby

Child

Teenager

True

False
get_classification(t, to_classify)

value: 30

< 65
    True
    < 20
        < 3
            Baby
        < 12
            Child
    Adult
    False
    > 300
        Immortal
        Senior
get\_classification(t, to\_classify)

value: 30

Adult
get_classification(t, to_classify)

- Write a function that returns the classification of t_classify using t
  - True -> Go to the left subtree
  - False -> Go to the right subtree
get_classification(t, to_classify)

if t.left is None and t.right is None:
    return t.value

Base case: t is a leaf
get_classification(t, to_classify)

if t.left is None and
    t.right is None:
    return t.value

if t.value(to_classify):
    Test to_classify on t.value
get_classification(t, to_classify)

if t.left is None and t.right is None:
    return t.value

if t.value(to_classify):
    return get_classification(t.left, to_classify)
get_classification(t, to_classify)

if t.left is None and
t.right is None:
    return t.value

if t.value(to_classify):
    return get_classification(t.left, to_classify)

return get_classification(t.right, to_classify)
Pre-order Traversals

● Same as Trees:
  ○ Root first
  ○ Then children (left -> right)
Pre-order Traversals

1

1

2

3

4 7 8 9

5

6

10
Pre-order Traversals

1, 2
Pre-order Traversals

1, 2, 4
Pre-order Traversals

1, 2, 4, 7
Pre-order Traversals

1, 2, 4, 7, 8
Pre-order Traversals

1, 2, 4, 7, 8, 5
Pre-order Traversals

1, 2, 4, 7, 8, 5, 9
Pre-order Traversals

1, 2, 4, 7, 8, 5, 9, 3
Pre-order Traversals

1, 2, 4, 7, 8, 5, 9, 3, 6
Pre-order Traversals

1, 2, 4, 7, 8, 5, 9, 3, 6, 10
Pre-order Traversals

1, 2, 4, 7, 8, 5, 9, 3, 6, 10
Post-order Traversals

● Same as Trees:
  ○ Children first (left -> right)
  ○ Root after
Post-order Traversals

1 2 4 7 8 5 9 3 6 10

7
Post-order Traversals

7, 8
Post-order Traversals

7, 8, 4
Post-order Traversals

1, 2, 3, 4, 9, 7, 8, 10
Post-order Traversals

1, 2, 4, 5, 7, 8, 9, 6, 10

7, 8, 4, 9, 5
Post-order Traversals

7, 8, 4, 9, 5, 2
Post-order Traversals

7, 8, 4, 9, 5, 2, 10
Post-order Traversals

7, 8, 4, 9, 5, 2, 10, 6
Post-order Traversals

7, 8, 4, 9, 5, 2, 10, 6, 3
Post-order Traversals

7, 8, 4, 9, 5, 2, 10, 6, 3, 1
Post-order Traversals

7, 8, 4, 9, 5, 2, 10, 6, 3, 1
Level-order Traversals

- Same as Trees:
  - By level (top to bottom, left to right)
Level-order Traversals

1
2
4
7
8
9

3
6
5
10
Level-order Traversals

1, 2
Level-order Traversals

1, 2, 3
Level-order Traversals

1, 2, 3, 4
Level-order Traversals

1, 2, 3, 4, 5
Level-order Traversals

1, 2, 3, 4, 5, 6
Level-order Traversals

1, 2, 3, 4, 5, 6, 7
Level-order Traversals

1, 2, 3, 4, 5, 6, 7, 8
Level-order Traversals

1, 2, 3, 4, 5, 6, 7, 8, 9
Level-order Traversals

1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Level-order Traversals

1, 2, 3, 4, 5, 6, 7, 8, 9, 10
In-order Traversals

- Left subtree first
- Then root
- Then right subtree
In-order Traversals
In-order Traversals

7, 4
In-order Traversals

7, 4, 8
In-order Traversals

7, 4, 8, 2
In-order Traversals

7, 4, 8, 2, 9
In-order Traversals

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

7, 4, 8, 2, 9, 5
In-order Traversals

7, 4, 8, 2, 9, 5, 1
In-order Traversals

7, 4, 8, 2, 9, 5, 1, 3
In-order Traversals

7, 4, 8, 2, 9, 5, 1, 3, 10
In-order Traversals

7, 4, 8, 2, 9, 5, 1, 3, 10, 6
In-order Traversals

7, 4, 8, 2, 9, 5, 1, 3, 10, 6
Assignment 2

- Material goes up to last week (Trees)
- Continuation of Assignment 1
- Details on the handout
  - I'll just clarify some parts of it with examples
Sorcerers

- **set_decision_tree()**
  - This is just a setter
  - Takes in a SkillDecisionTree and sets it
  - When attack() is used, it uses that SkillDecisionTree to pick a skill.
SkillDecisionTree

3  Caster's SP > 20  Mage Attack

4  Target's HP < 30  Rogue Special

1  Caster's HP > 90  Rogue Attack

2  Target's SP > 40  Mage Special

5  Caster's HP > 50  Mage Attack

8  Rogue Attack

7  Rogue Special

6  Rogue Attack
SkillDecisionTree

Condition

Priority

Skill returned if False

5  Caster's HP > 50  Mage Attack

2  Target's SP > 40  Mage Special

1  Caster's HP > 90  Rogue Attack

4  Target's HP < 30  Rogue Special

6  Rogue Attack

3  Caster's SP > 20  Mage Attack

8  Rogue Attack

7  Rogue Special

3  Target's HP < 30  Rogue Special

6  Rogue Attack
Leaves still have functions for conditions, but it gets ignored.
SkillDecisionTree

5  Caster's HP > 50  Mage Attack

3  Caster's SP > 20  Mage Attack

2  Target's SP > 40  Mage Special

4  Target's HP < 30  Rogue Special

8  Rogue Attack

7  Rogue Special

6  Rogue Attack

1  Caster's HP > 90  Rogue Attack
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30

Caster's HP > 50
Mage Attack

3
Caster's SP > 20
Mage Attack

2
Target's SP > 40
Mage Special

4
Target's HP < 30
Rogue Special

1
Caster's HP > 90
Rogue Attack

5
Caster's HP > 50
Mage Attack

6
Rogue Attack

7
Rogue Special

8
Rogue Attack
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30

Caster's HP > 50
Mage Attack

3 Caster's SP > 20
Mage Attack

2 Target's SP > 40
Mage Special

4 Target's HP < 30
Rogue Special

6 Rogue Attack

1 Caster's HP > 90
Rogue Attack

7 Rogue Special

5 Caster's HP > 50
Mage Attack

8 Rogue Attack
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30
SkillDecisionTree

5 Caster's HP > 50 Mage Attack

3 Caster's SP > 20 Mage Attack

2 Target's SP > 40 Mage Special

4 Target's HP < 30 Rogue Special

6 Rogue Attack

1 Caster's HP > 90 Rogue Attack

7 Rogue Special

8 Rogue Attack

Caster HP: 100, SP: 40
Target HP: 50, SP: 30
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30
SkillDecisionTree

Caster HP: 100, SP: 40
Target HP: 50, SP: 30

3  Caster's SP > 20  Mage Attack

4  Target's HP < 30  Rogue Special

6  Rogue Attack

2  Target's SP > 40  Mage Special

5  Caster's HP > 50  Mage Attack

8  Rogue Attack

1  Caster's HP > 90  Rogue Attack

7  Rogue Special

Target's HP < 30

Caster's SP > 20
After Choosing a Skill

● We use that skill
  ○ Its effects on a BattleQueue/etc. Are also applied
  ○ Look at the use() method

● Only uses 15 SP!
RestrictedBattleQueue

- Modified version of the BattleQueue from A1
- add() is restricted based off some rules
- (When I say "copy" I don't mean use copy() -- copy()'s only for Minimax)
RestrictedBattleQueue

Rule 1: "The first time each character is added to the RestrictedBattleQueue, they're able to add"
**RestrictedBattleQueue**

We assume the first character in the RestrictedBattleQueue is the one adding.

<table>
<thead>
<tr>
<th>Character Order</th>
<th>Able to Add?</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**RestrictedBattleQueue**

We assume the first character in the RestrictedBattleQueue is the one adding.

<table>
<thead>
<tr>
<th>Character Order</th>
<th>Able to Add?</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

This means P1 is adding to the queue.
RestrictedBattleQueue

We assume the first character in the RestrictedBattleQueue is the one adding.

If we tried to call add(P2), we want them to be able to add.

<table>
<thead>
<tr>
<th>Character Order</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Able to Add?</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
RestrictedBattleQueue

Rule 1: "The first time each character is added to the RestrictedBattleQueue, they're able to add"

- This rule exists so that the 2nd rule doesn't cause an issue.
- Don't overthink it!
Rule 2: "Characters that are added to the RestrictedBattleQueue by a character other than themselves cannot add."
RestrictedBattleQueue

P1 is adding to the queue.

Character Order

Able to Add?

P1
Yes

P2
Yes
RestrictedBattleQueue

Character Order

Able to Add?

P1
Yes
P2
Yes
P2
No

add(P2)
RestrictedBattleQueue

Character Order

Able to Add?

P2
No
Yes

P1
Yes

P2

P2 is at the front.
P2 cannot add. add() calls do nothing.
Rule 3: "Characters that have 2 copies of themselves in the RestrictedBattleQueue already that can add cannot add."
RestrictedBattleQueue

Character Order

Able to Add?

P1
Yes

P2
Yes

P1 is adding
<table>
<thead>
<tr>
<th>Character Order</th>
<th>P1</th>
<th>P2</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Able to Add?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**add(P1)**

This one can add since there were < 2 P1s that could add.
add(P1) again
This one can't add since there were 2 P1s that could add already.
RestrictedBattleQueue

- Fixes the balance issue from A1
  - I.e. if a Rogue keeps using special attacks, all of those copies would persist in the BattleQueue!
Minimax

- This is the only place you're supposed to use `copy()`!
  - Don't use it anywhere else. It'll probably cause problems.
- Full rules in the handout; I'll skim over an example.
Minimax

- Our game is carried out in our BattleQueue
  - All the information we need is in it
- "Current player": The player returned by peek()
  - Defaults to P1 if there's no player (see BattleQueue code)
Minimax

- `get_state_score()` returns the highest guaranteed score for the current player
  - Takes in a BattleQueue!
get_state_score

- For a game that's over:

  Rogue: 50HP/10SP
  Mage: 0HP/10SP

  M -> R -> R
For a game that's over:

get_state_score

- Current player is the Mage.
- Loser's score: $-1 \times \text{Winner's HP}$

Rogue: 50HP/10SP
Mage: 0HP/10SP

M -> R -> R
For a game that's over:

- get_state_score

Current player is the Mage.
Loser's score:
-50

Rogue: 50HP/10SP
Mage: 0HP/10SP

M -> R -> R
get_state_score

- For a game that's over:
  - get_state_score

  Rogue: 50HP/10SP
  Mage: 0HP/10SP

  R -> M

  Current player is the Rogue.
  Winner's score:
  Their HP (50 in this case)
get_state_score

- For a game that's not over: Find out the highest score that the current player can guarantee.
  - Assume the other player also tries to maximize their own score.
Rogue: 40/6
Mage: 14/35
M -> R
Rogue: 40/6
Mage: 14/35
M -> R

Note: You don't need to implement this with a Tree. It's just easier to visualize this way.
Rogue: 40/6
Mage: 14/35
M -> R

Rogue: 30/6
Mage: 14/30
R -> M
Rogue: 40/6  
Mage: 14/35  
M -> R

Rogue: 30/6  
Mage: 14/30  
R -> M

Rogue: 30/3  
Mage: 7/30  
M -> R

Rogue: 10/6  
Mage: 14/5  
R -> R -> M

Rogue: 10/3  
Mage: 7/5  
R -> M
Rogue: 40/6
Mage: 14/35
M -> R

Rogue: 30/6
Mage: 14/30
R -> M

Rogue: 30/3
Mage: 7/30
M -> R

Rogue: 20/3
Mage: 7/25
R -> M

Rogue: 10/6
Mage: 14/5
R -> R -> M

Rogue: 10/3
Mage: 7/5
R -> M

Rogue: 0/3
Mage: 7/0
R -> R -> M
Rogue: 40/6
Mage: 14/35
M -> R

Rogue: 30/6
Mage: 14/30
R -> M

Rogue: 30/3
Mage: 7/30
M -> R

Rogue: 20/3
Mage: 7/25
R -> M

Rogue: 20/0
Mage: 0/25
M -> R

Rogue: 10/6
Mage: 14/5
R -> R -> M

Rogue: 10/3
Mage: 7/5
R -> M

Rogue: 10/0
Mage: 0/5
M -> R
Rogue: 40/6
Mage: 14/35
M -> R

Rogue: 30/6
Mage: 14/30
R -> M

Rogue: 30/3
Mage: 7/30
M -> R

Rogue: 20/3
Mage: 7/25
R -> M

Rogue: 20/0
Mage: 0/25
M -> R

Rogue: 0/3
Mage: 7/5
R -> R -> M

Rogue: 10/0
Mage: 0/5
M -> R

Rogue: 10/3
Mage: 7/5
R -> M

Rogue: 10/6
Mage: 14/5
R -> R -> M

M
-20

R
20

M
-20
We return 7!
Minimax

- Remember: Both players try to maximize their score!
  - Assume they'll never pick a suboptimal move

- Iterative minimax: Rough idea on how to implement it is given
  - Do the recursive one first!
Midterm 2

● 25 marks
● 3 questions:
  ○ Recursion with nested lists
  ○ Trees
  ○ Binary trees
● All of them can be solved recursively
Midterm 2

● Each question has 2 parts
  ○ Part 1: Not code based
  ○ Part 2: Programming question

● Anything after midterm 1 is fair game
  ○ Go through the labs, recursion practice, midterm practice, etc.
Midterm 2: Mark Breakdown

- Recursion: /7
- Trees: /10
- Binary Trees: /8

Advice: 50 minute midterm, 25 marks in total.
- 1 mark = 2 minutes of work
- Manage your time well!
Homework

- Exercise 7 (due next Thursday @ 11PM)
- Midterm next week @ EX100
  - Class afterwards @ 7:30PM
- Exercise 6 due tomorrow @ 11PM
Past Midterms