Exam One Review
Overview
API

- Provides basic structure for software components
  - Return values including data types
  - Name of each method or function
  - Required Parameters and associated data types for each
  - Description of what each component does
    - Not pictured here
Test __main__

- The functional scout for how your program is performing
  - Ensures each component operates correctly

```python
# Test program for Cell.
#
# To run this code and test your class, type "python Cell.py" at the
# command prompt. Your output should be exactly like what is shown
# on the assignment page.
#
if __name__ == "__main__":
    import Config
    # The config object stores defaults for sizes, sounds, and images
    config = Config.Config()

    # Fake a dungeon that is 3 x 3
    WIDTH = 3
    HEIGHT = 3
    StdDraw.setCanvasSize(WIDTH * config.cellPixels(), HEIGHT * config.cellPixels())
    StdDraw.setXscale(0, WIDTH * config.cellPixels())
    StdDraw.setYscale(0, HEIGHT * config.cellPixels())

    # Draw a wall at various locations
    wall = Cell(config.wallImage())
    wall.draw(config, 0, 0)
    wall.draw(config, 1, 1)

    # Wall with a different image
    wall2 = Cell(config.passageImage())
    wall2.draw(config, 2, 2)
```
Commenting Format

• Header comment
• Comment each method and function
  • Purpose/Intended Behavior
  • Input parameters
    • Name
    • Data type expected
    • Intended Behavior
• Return values
  • Name
  • Data type expected
  • Intended Behavior
• Guide the reader through any hacks you had to use

```python
from game import Agent
from game import Actions
from game import Directions
import random
from util import manhattanDistance
import util

def getFood(self):
    """
    Returns a Grid of boolean food indicator variables.
    
    Grids can be accessed via list notation, so to check if there is food at (x,y), just call
    if currentFood[x][y] -- True: ...
    """
    currentFood = state.getFood()
    if currentFood[x][y] == True:
        return self.data.food
```
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the gameState
    startPosition = gameState.get PacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)
    fringe = util.Queue()
    travel_log = []
    directioner = []
    overCost = 0
    begin = problem.getStartState()

    # Only one data type this time...
    fringe.push((begin, directioner))

    while not (fringe.isEmpty()):
        # Create a pair of values to expand the fringe into
        node, actions = fringe.pop()

        # If we haven't logged this location yet...
        if not (node in travel_log):
            # Put it in the log...
            travel_log.append(node)

            # Check it against our treasure map...
            if problem.isGoalState(node):
                return actions

            # Mark down where we can go from here...
            children = problem.getSuccessors(node)

            # For every place in this list of places we can go...
            for child in children:
                # Break it down into usable data...
                xy, direction, cost = child

                # Make a list of where each of these places are relative to us...
                newActions = actions + [direction]

                # We're using bfs, give orders to the crew!
                if isinstance(fringe, util.Stack) or isinstance(fringe, util.Queue):
                    fringe.push((xy, newActions))
Dynamic Lists

Arrays
List Operations

- **Insertion**
  - Append() – Adds item to the end of the list
    - `listName.append(item)`
  - Insert() – Adds item at the **given index**
    - `listName.insert(index, item)`

- **Removal**
  - Del() – Facilitates deletion of multiple elements
    - `Del listName[2 : 5]` (Deletes elements in position 2 through 5)
  - Pop() – Deletes at the given position
    - `listName.pop(2)` (Deletes the third element, index #2)

- **Sort**
  - sort() – Sorts the list in **increasing order**
    - `listName.sort()`
  - Sorted() – Behaves similar to sort(), grants flexibility at higher level usage
    - "**Returns a sorted list**"
    - `sortedList = Sorted(listName)`

- **Searching**
  - Index() – Finds an element, returns its index **starting at 0**
    - `listName.index(element)`
  - Count() – returns how many times an item occurs in a list
    - `listName.count(item)`
Performance
Scientific Method

- **Observe**
  - Find out what’s relevant

- **Hypothesize**
  - Structure that information into a model

- **Predict**
  - Use that model to look into the future

- **Verify**
  - Do that until you’re sure you’re right, or you can actually see into the future
    - With like, magic or some shit

- **Validate**
  - If it’s not working, find something that will.

- **Experiments must be reproducible**
  - People have to be able to follow along

- **Hypotheses must be falsifiable**
  - If it’s not possible for you to be proven wrong, that doesn’t mean you’re right
**Metrics**

**Time**
- How long does your code run for?
- Tends to be integrated into your development software

```python
import time
t1 = time.time()
# Put the code you want to time here
t2 = time.time()
print(t2-t1)
```

**Space**
- Do you have the resources for it?
  - Physical memory management always factors into development at some point.

- 8 GB = 8.6 billion places to store a byte (byte = 256 possibilities)
- Python float, 64-bits = 8 bytes
- 8 GB / 8 bytes = over 1 million floats!
Empirical Analysis

Empirical
Verifiable by observation or experience rather than theory or pure logic

Analysis
Detailed examination of the elements of something
Doubling Hypothesis

Quick and easy way to find the order of growth of your program

- Order of Growth -
Mathematically, how does your input space relate to your runtime?

<table>
<thead>
<tr>
<th>Constant from ratio</th>
<th>Hypothesis</th>
<th>Order of growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$T = a \times N$</td>
<td>linear, $O(N)$</td>
</tr>
<tr>
<td>4</td>
<td>$T = a \times N^2$</td>
<td>quadratic, $O(N^2)$</td>
</tr>
<tr>
<td>8</td>
<td>$T = a \times N^3$</td>
<td>cubic, $O(N^3)$</td>
</tr>
<tr>
<td>16</td>
<td>$T = a \times N^4$</td>
<td>$O(N^4)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T(N)$</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>0.16</td>
<td>-</td>
</tr>
<tr>
<td>800</td>
<td>0.63</td>
<td>3.94</td>
</tr>
<tr>
<td>1600</td>
<td>4.33</td>
<td>6.87</td>
</tr>
<tr>
<td>3200</td>
<td>33.69</td>
<td>7.78</td>
</tr>
<tr>
<td>6400</td>
<td>263.82</td>
<td>7.83</td>
</tr>
</tbody>
</table>
Prediction

Instrumental in finding bottlenecks by determining what your program should be doing.

Estimating Constant, Making Predictions

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>0.16</td>
<td>-</td>
</tr>
<tr>
<td>800</td>
<td>0.63</td>
<td>3.94</td>
</tr>
<tr>
<td>1600</td>
<td>4.33</td>
<td>6.87</td>
</tr>
<tr>
<td>3200</td>
<td>33.69</td>
<td>7.78</td>
</tr>
<tr>
<td>6400</td>
<td>263.82</td>
<td>7.83</td>
</tr>
</tbody>
</table>

Keith's Desktop data

\[ T = a N^3 \]

\[ 263.82 = a (6400)^3 \]

\[ a = 1.01 \times 10^{-09} \]

Prediction:

How long for desktop to solve a 100,000 integer problem?

\[ 1.01 \times 10^{-09} (100000)^3 = 1006393 \text{ secs} = 280 \text{ hours} \]

Keith's Laptop data

\[ T = a N^3 \]

\[ 47311.43 = a (6400)^3 \]

\[ a=1.80 \times 10^{-07} \]

Prediction:

How long for laptop to solve a 100,000 integer problem?

\[ 1.80 \times 10^{-07} (100000)^3 = 1.80 \times 10^{08} \text{ secs} = 50133 \text{ hours} \]
Nested Loops and Growth

- **Nested loops**
  - A good clue to order of growth
  - But each loop must execute "on the order of" $N$ times
  - If loop not a linear function of $N$, loop doesn't cause order to grow

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            count += 1
```

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, 10000):
            count += 1
```

<table>
<thead>
<tr>
<th>$N^3$</th>
<th>$N^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>$T(N)$</td>
</tr>
<tr>
<td>5000</td>
<td>6.85</td>
</tr>
<tr>
<td>10000</td>
<td>53.48</td>
</tr>
<tr>
<td>20000</td>
<td>425.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$N$</th>
<th>$T(N)$</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>13.40</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>53.20</td>
<td>3.97</td>
</tr>
<tr>
<td>20000</td>
<td>212.49</td>
<td>3.99</td>
</tr>
</tbody>
</table>

$425.97 = a \times (20000^3)$
$a = 1.06 \times 10^6$

$212.49 = a \times (20000^2)$
$a = 5.31 \times 10^7$
Linked Lists
Sequential vs Linked

Sequential
- Fixed size
  - Allows for inefficient dynamics
- Simple
- A populated chunk of memory

Linked
- Nodes that hold data, and point to another node
  - Does not require nodes to be neighbors in memory
- Dynamic
  - Expand by updating node relationships
- Flexible
- Comparatively challenging
Anatomy of a Linked List

• Each node contains two parts
  • An item of any kind. Literally any kind of data can be in here.

• A pointer to the next node
  • Linked lists are trains of data.

```python
class Node:
    def __init__(self, s):
        self.item = s
        self.next = None
```

Three Node objects hooked together to form a linked list

Special pointer value null (None in Python) terminates the list. We denote with a dot.
Building a linked list

```
first = Node()
first.item = "The"
second = Node()
second.item = "cat"
third = Node()
third.item = "sat"
first.next = second
second.next = third
```

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>&quot;cat&quot;</td>
</tr>
<tr>
<td>C1</td>
<td>C8</td>
</tr>
<tr>
<td>C2</td>
<td>-</td>
</tr>
<tr>
<td>C3</td>
<td>-</td>
</tr>
<tr>
<td>C4</td>
<td>&quot;The&quot;</td>
</tr>
<tr>
<td>C5</td>
<td>Co</td>
</tr>
<tr>
<td>C6</td>
<td>-</td>
</tr>
<tr>
<td>C7</td>
<td>-</td>
</tr>
<tr>
<td>C8</td>
<td>&quot;sat&quot;</td>
</tr>
<tr>
<td>C9</td>
<td>null</td>
</tr>
</tbody>
</table>
Traversing a linked list

Traversing a List

- Iterate over all elements in a linked list
  - Assume list is null terminated
  - Assume `first` instance variable points to start of list
  - Print all the strings in the list

```python
current = first
while current != None:
    print(current.item)
    current = current.next
```

```
"The" ——> "cat" ——> "sat" ——>
```

`first`
Operations on a linked list

Add – Inserts a node at the end of the list

Insert – Inserts a node at a specific point in the list

Remove – Removes a node at a specific point in the list
Stacks and Queues
Abstract Data Types vs Data Structures

ADTs
ADTs don’t specify implementation details
Just provides boundaries and tools

Data Structures
• Specific Implementations of ADTs.
  • Stacks can be implemented using either an array, or a linked list
The Stack

Stacks

Structure nodes as Last in, first out. (LIFO)

Operations

Push() – Inserts a node at the top of the stack

Pop() – Removes the node at the top of the stack
The Queue

Queue

Structure nodes as First in, first out. (LIFO)

Operations

Enqueue() – Inserts a node at the end of the queue

Dequeue() – Removes the node at the front of the queue
Inheritance
What’s the point?

The main purpose of inheritance is to make it easier for us to understand our code by structuring it in a way that reduces duplication.
Inheritance
Abstract Classes

- Abstract classes are **unimplementable ideas**
  - Very useful to use as the foundation for classes lower in the hierarchy
Generic Data Types

A *generic type* is a generic class or interface that is parameterized over types

If your class or method can work with more than one data type...

(Strings, Ints, nodes, structures you’ve made yourself, a tuple named doink that you haven’t renamed yet...)

It’s a generic.

If it *technically won’t break* when you run a second data type through it, it’s a generic. The goal is to re-use your code when you can
Multiple Inheritance
Multiple Inheritance

- Food
  - Honey
- Medicine
- Vehicle
- Asset
- Liability
  - Car
When does it break?

With python, it doesn’t really

Each language handles things differently
If your language doesn’t have multiple inheritance, just use an abstract class to pretend it does.
Recursion
Recursion

A method that calls itself is recursive

Solves problems by collapsing Must define a stopping point in on itself
Recursion

A method that calls itself is recursive

Solves problems by collapsing in on itself
Recursion

Hello Recursion

- **Goal:** Compute factorial $N! = 1 \times 2 \times 3 \ldots \times N$
  - Base case: $0! = 1$
  - Induction step:
    - Assume we know $(N - 1)!$
    - Use $(N - 1)!$ to find $N$

```python
import sys

def fact(N):
    if N == 0:
        return 1
    return fact(N - 1) * N

if __name__ == "__main__":
    N = int(sys.argv[1])
    print(str(N) + "! = " + str(fact(N)))
```
When to use

• If it is more intuitive to observe, use it.
• If it’s easier to debug, use it.
• If you will spend more resources doing things recursively, don’t.
• If you’re a time traveler with a craptop, don’t.
Stack Overflow occurs when you use more memory than your stack can manage.

Processes resolve following stack behavior.

Base case is resolved, and returned to the recursive call above.
Examples

Slide collection 07-08 on Moodle