CS1010S Programming Methodology

Lecture 8
Implementing Data Structures

14 Oct 2020

Python you should know

Python Statements :

- `def`
- `return`
- `lambda`
- `if, elif, else`
- `for, while, break, continue`
- `import`

Data abstraction primitives:

- `tuple`
- `list`

Today's Agenda

- The Game of Nim
  - More Wishful Thinking
  - Understanding Python code
  - Simple data structures
- Designing Data Structures
- Multiple Representations

The Game of Nim

- Two players
- Game board consists of piles of coins
- Players take turns removing any number of coins from a single pile
- Player who takes the last coin wins

Let’s Play!!

How to Write This Game?

1. Keep track of the game state
2. Specify game rules
3. Figure out strategy
4. Glue them all together

Let’s start with a simple game with two piles

Start with Game State

What do we need to keep track of?

Number of coins in each pile!

Game State

Wishful thinking:

Assume we have:

```
def make_game_state(n, m):
        ...
```

where `n` and `m` are the number of coins in each pile.

What Else Do We Need?

```
def size_of_pile(game_state, p):
    ...
```

where p is the number of the pile

```
def remove_coins_from_pile(game_state, n, p):
    ...
```

where p is the number of the pile and n is the number of coins to remove from pile p.

Let’s start with the game

```
def play(game_state, player):
    display_game_state(game_state)
    if is_game_over(game_state):
        announce_winner(player)
    elif player == "human":
        play(human_move(game_state), "computer")
    elif player == "computer":
        play(computer_move(game_state), "human")
```

What happens if we evaluate:

```
play(make_game_state(5, 8), "mickey-mouse")
```

Take Care of Error Condition

```
def play(game_state, player):
    display_game_state(game_state)
    if is_game_over(game_state):
        announce_winner(player)
    elif player == "human":
        play(human_move(game_state), "computer")
    elif player == "computer":
        play(computer_move(game_state), "human")
    else:
        print("player wasn't human or computer:", player)
```

Displaying Game State

```
def display_game_state(game_state):
    print("")
    print(" Pile 1: " + str(size_of_pile(game_state,1)))
    print(" Pile 2: " + str(size_of_pile(game_state,2)))
    print("")
```

Game Over

Checking for game over:

```
def is_game_over(game_state):
    return total_size(game_state) == 0

def total_size(game_state):
    return size_of_pile(game_state, 1) + size_of_pile(game_state, 2)
```

Announcing winner/loser:

```
def announce_winner(player):
    if player == "human":
        print("You lose. Better luck next time.")
    else:
        print("You win. Congratulations.")
```

Getting Human Player’s Move

```
def human_move(game_state):
    p = input("Which pile will you remove from?")
    n = input("How many coins do you want to remove?")
    return remove_coins_from_pile(game_state, int(n), int(p))
```

Artificial Intelligence

```python
def computer_move(game_state):
    pile = 1 if size_of_pile(game_state, 1) > 0 else 2
    print("Computer removes 1 coin from pile "+ str(pile))
    return remove_coins_from_pile(game_state, 1, pile)
```

Is this a good strategy?

Game State

```
def make_game_state(n, m):
    return (10 * n) + m

def size_of_pile(game_state, pile_number):
    if pile_number == 1:
        return game_state // 10
    else:
        return game_state % 10

def remove_coins_from_pile(game_state, num_coins, pile_number):
    if pile_number == 1:
        return game_state – 10 * num_coins
    else:
        return game_state - num_coins
```

What is the limitation of this representation?

Another Implementation

```
def make_game_state(n, m):
  return (n, m)

def size_of_pile(game_state, p):
  return game_state[p-1]

def remove_coins_from_pile(game_state, num_coins, pile_number):
  if pile_number == 1:
    return make_game_state(size_of_pile(game_state,1) - num_coins,
                           size_of_pile(game_state,2))
  else:
    return make_game_state(size_of_pile(game_state,1),
                           size_of_pile(game_state,2) - num_coins)
```

Improving Nim

Lets modify our Nim game by allowing “undo”

- Only Human player can undo, not Computer
- Removes effect of the most recent move
  - i.e. undo most recent computer and human move
  - Human’s turn again after undo
- Human enters “0” to indicate undo

Key Insight

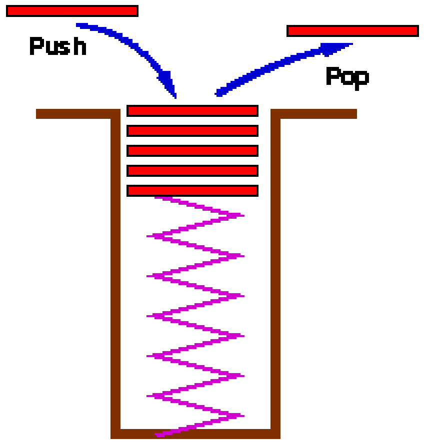
We need a data structure to remember the history of game states

History of game states

- Before each human move, add the current game state to the history.
- When undoing,
  - Remove most recent game state from history
  - Make this the current game state

Data structure: Stack

- A stack is a data structure with the LIFO property.
  - Last In, First Out
  - Items are removed in the reverse order in which they were added.



Wishful thinking again

Assume we have the following:

`make_stack()` : returns a new, empty stack

`push(s, item)` : adds item to stack s

`pop(s)` : removes the most recently added item from stack s, and returns it

`is_empty(s)` : returns `True` if s is empty, `False` otherwise

Stack operations

```
>>> s = make-stack()
>>> pop(s)
None
```

empty stack, nothing to pop

```
>>> push(s, 5)
>>> push(s, 3)
>>> pop(s)
3
>>> pop(s)
5
is_empty(s)
True
```

Implement a stack
as homework

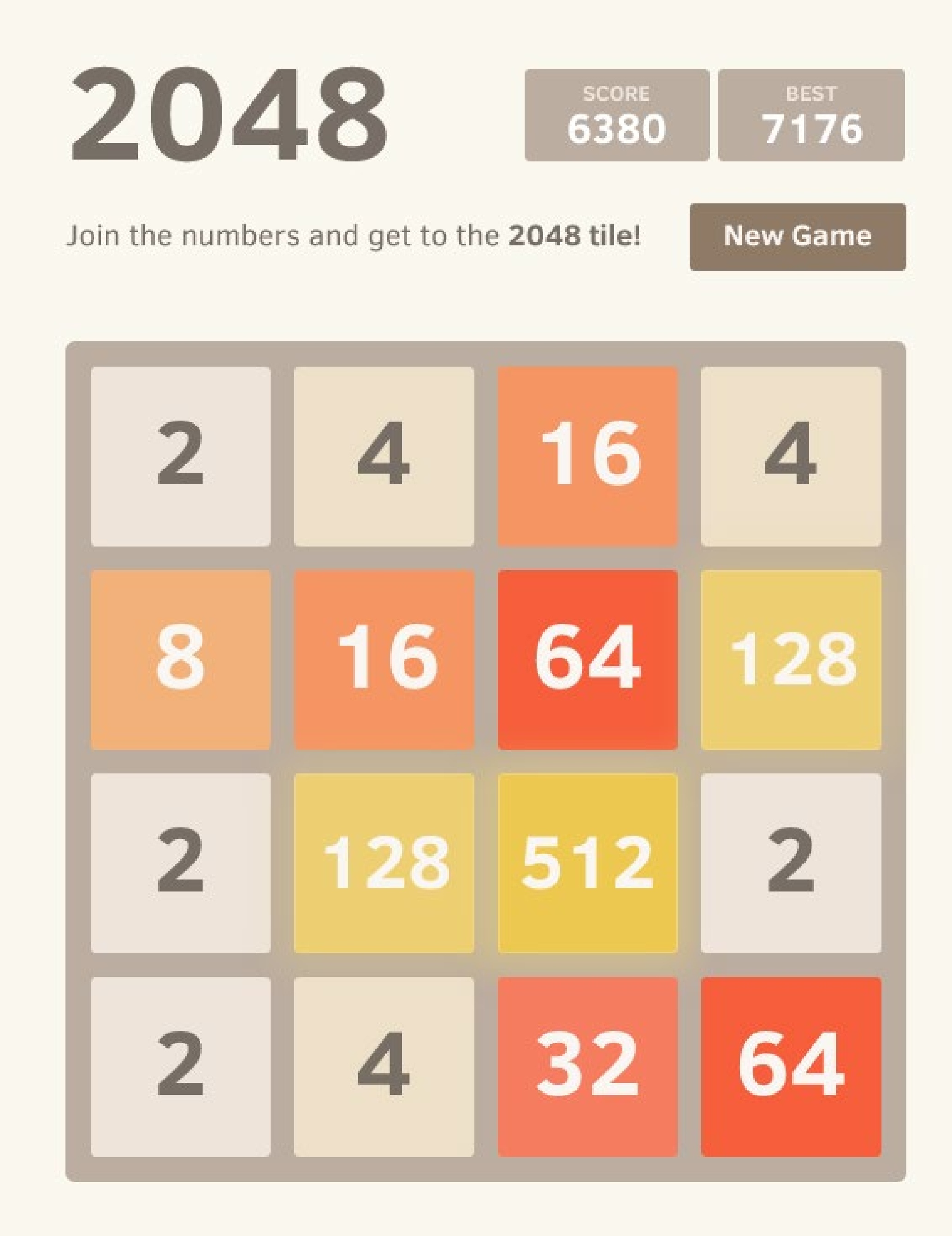
Changes to Nim

```
game_stack = make_stack()

def human_move(game_state):
    p = prompt("Which pile will you remove from?")
    n = prompt("How many coins do you want to remove?")
    if int(p) == 0:
        return handle_undo(game_state)
    else:
        push(game_stack, game_state)
        return remove_coins_from_pile(game_state, int(n), int(p))
```

Changes to Nim

```
def handle_undo(game_state):
    if is_empty(game_stack):
        print("No more previous moves!")
        return human_move(game_state)
    old_state = pop(game_stack)
    display_game_state(old_state)
    return human_move(old_state)
```


2048
SCORE
6380
BEST
7176
Join the numbers and get to the 2048 tile!
New Game
2 4 16 4
8 16 64 128
2 128 512 2
2 4 32 64

Data Structures: Design Principles

When designing a data structure, need to spell out:

- Specification
- Implementation

Specification (contract)

- What does it do?
- Allows others to use it.

Implementation

- How is it realized?
- Users do not need to know this.
- Choice of implementation.

Nim: Game state

piles, coins in each pile

size, remove-coin

Multiple representations possible

Specification

- Conceptual description of data structure.
  - Overview of data structure.
  - State assumptions, contracts, guarantees.
  - Give examples.

Specification

- Operations:
  - Constructors
  - Selectors (Accessors)
  - Predicates
  - Printers

Example: Lists

- Specs:
  - A list is a collection of objects, in a given order.
    - e.g. [], [3, 4, 1]

Example: Lists

Specs:

- Constructors: `list(), [ ]`
- Selectors: `[ ]`
- Predicates: `type, in`
- Printer: `print`

Multiset: Specs

A multiset (or bag or mset)

- is a modified set that allows **duplicate elements**
- count of each element is called the **multiplicity**
- arrangement of elements does not matter

Example:

- $\{a, b, b, a\}$, $\{b, a, b, a\}$ are the same
- both elements a and b have multiplicity of 2

Multisets: Specs

Constructors:

```
make_empty_mset, adjoin_mset,
union_mset, intersection_mset
```

Selectors:

Predicates:

```
multiplicity_of, is_empty_mset
```

Printers:

```
print_set
```

Multisets: Contract

For any multiset S, and any object x

```
>>> multiplicitiy_of(x, adjoin_mset(x, S)) > 0
True
```

Adjoining an element to an mset produces an mset with “one more” element

MSets: Contract

```
>>> multiplicitiy_of(x, union_set(S, T))
```

is equal to

```
>>> multiplicitiy_of(x, S) + multiplicitiy_of(x, T)
```

The elements of $(S \cup T)$ are the elements that are in S or in T

```
>>> multiplicitiy_of(x, empty_set)
False
```

No object is an element of the empty set.

etc...

Implementation

Choose a representation

- Usually there are choices, e.g. lists, trees
- Different choices affect time/space complexity.
- There may be certain constraints on the representation. These should explicitly stated.
  - e.g. in rational number package, denom $\neq 0$

Implementation

- Implement constructors, selectors, predicates, printers, using your selected representation.
- Make sure you satisfy all contracts stated in specification!

MSets: Implementation #1

- Representation: unordered list
  - Empty set represented by empty list.
  - Set represented by a list of the objects.

MSets: Implementation #1

Constructors:

```
def make_mset():
    '''returns a new, empty set'''
    return []
```

Predicates:

```
def is_empty_mset(s):
    return not s
```

MSets: Implementation #1

Predicates:

```
def multiplicitiy_of(x, s):  # linear search
    count = 0
    for ele in s:
        if ele == x:
            count += 1
    return count
```

Time complexity:

$O(n)$, n is size of set

MSets: Implementation #1

Constructors:

```
def adjoin_set(x, s):
    s.append(x)
```

Time complexity: $O(1)$

MSets: Implementation #1

Constructors:

```
def intersection_of(s1, s2):  # complete matching
    result = []
    for ele in s1:  # O(n)
        if ele not in result:  # O(n)
            n = min(multiplicity_of(ele, s1),
                    multiplicity_of(ele, s2))
            result.extend(n * [ele])
    return result
```

Time complexity: $O(n^2)$, n is size of set

MSets: Implementation #2

- Representation: ordered list
  - Empty set represented by empty list.
  - But now objects are sorted.

WHY WOULD WE WANT TO DO THIS?

MSets: Implementation #2

Note: specs does not require this, but we can impose additional constraints in implementation.

But this is only possible if the objects are comparable, i.e. concept of "greater_than" or "less_than".

e.g. numbers: <

e.g. strings, symbols: lexicographical order (alphabetical)

Not colors: red, blue, green

MSets: Implementation #2

Constructors:

```
def make_mset():
        return []  #as before
```

Predicates:

```
def is_empty_set(s):
    return not s #as before
```

MSets: Implementation #2

```
def adjoin_mset(x, s):
    # binary search
    low, high = 0, len(mset) - 1
    ...
    # found at mid, or not found
    s.insert(mid, x)
```

Time complexity: $O(n)$, n is size of set

MSets: Implementation #2

Predicates:

```
def multiplicity_of(x, s):
    low, high = 0, len(s) – 1  # binary search
    while low <= high:
        ...
        else:  # element found
            low, high = mid, mid # linear search left & right
            ...
            return high – low - 1
    return 0  # not found
```

Time complexity: $O(\log n)$, n is size of set

Intersection

Set 1: {1   3   4   4   8}

Set 2: {1   4   4   4   6   8   9}

Result: {1}

→ so 1 in intersection, move both set1, set2 cursor forward

Set 1: {1   3   4   8}

Set 2: {1   4   4   6   8   9}

Result: {1}

→ 3 < 4, 3 not in intersection, forward set1 cursor only

Intersection

```
Set 1: {1   3   4   4   8}
Set 2: {1   4   4   4   6   8   9}
Result: {1   4}
```

→ so 4 in intersection, forward both set1 & set2 cursor

```
Set 1: {1   3   4   4   8}
Set 2: {1   4   4   4   6   8   9}
Result: {1   4   4}
```

→ so 4 in intersection, forward both set1 & set2 cursor

Intersection

```
Set 1: {1   3   4   4   8}
Set 2: {1   4   4   4   6   8   9}
Result: {1   4   4}
```

→ 8 > 4, 4 not in intersection, forward set2 cursor

```
Set 1: {1   3   4   8}
Set 2: {1   4   4   4   6   8   9}
Result: {1   4   4}
```

→ 8 > 6, 6 not in intersection, forward set2 cursor

Intersection

```
Set 1: {1   3   4   4   8}
Set 2: {1   4   4   4   6   8   9}
Result: {1   4   4   8}
```

→ so 4 in intersection, forward both set1 & set2 cursor

```
Set 1: {1   3   4   4   8}
Set 2: {1   4   4   4   6   8   9}
Result: {1   4   4   8}
```

→ set1 empty, return result

MSets: Implementation #2

```
def intersection_of(s1, s2):  # “merge” algorithm
    result = []
    i, j = 0, 0
    while i < len(s1) and j < len(s2):
        if s1[i] == s2[j]:
            result.append(s1[i])
            i += 1
            j += 1
        elif s1[i] < s2[j]:
            i += 1
        else:
            j += 1
    return result
```

Time complexity:
$O(n)$, faster than previous!

Comparing Implementations

Time Complexity	Unordered List	Ordered List
`adjoin_mset`	append $O(1)$	insert $O(n)$
`multiplicity_of`	linear search $O(n)$	binary search $O(\log n)$
`intersection_of`	complete match $O(n^2)$	merge algorithm $O(n)$

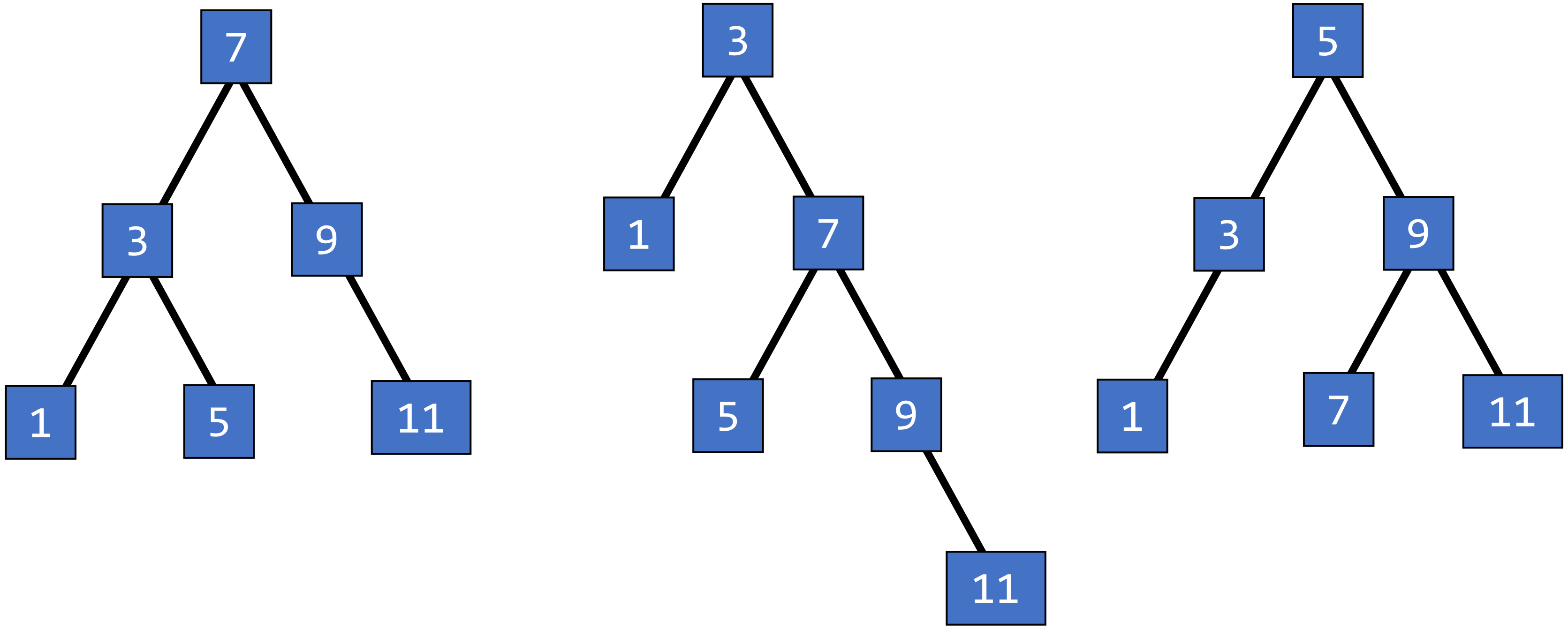
MSets: Implementation #3

- Representation: binary tree
  - Empty set represented by empty tree.
  - Objects are sorted.

MSets: Implementation #3

- Each node stores 1 object.
- Left subtree contains objects smaller than this.
- Right subtree contains objects greater than this.

MSets: Implementation #3



Three trees representing the set {1, 3, 5, 7, 9, 11}

MSets: Implementation #3

Tree operators:

```
def make_tree(entry, left, right):
      return [entry, left, right]

def entry(tree):
      return tree[0]

def left_branch(tree):
        return tree[1]

def right_branch(tree):
        return tree[2]
```

MSets: Implementation #3

- Each node in the tree contains
  - The element
  - The count

```
def make_mset():
    '''returns a new, empty set'''
    return []
```

MSets: Implementation #3

```python
def adjoin_mset(x, s):     # binary search
    if is_empty_set(s):
        s.extend(make_tree([x, 1], [], []))
    elif x == entry(s)[0]:
        entry(s)[1] += 1  # O(1) update
    elif x < entry(s)[0]:
        adjoin_mset(x, left_branch(s))
    else:
        adjoin_mset(x, right_branch(s))
```

Time complexity: $O(\log n)$

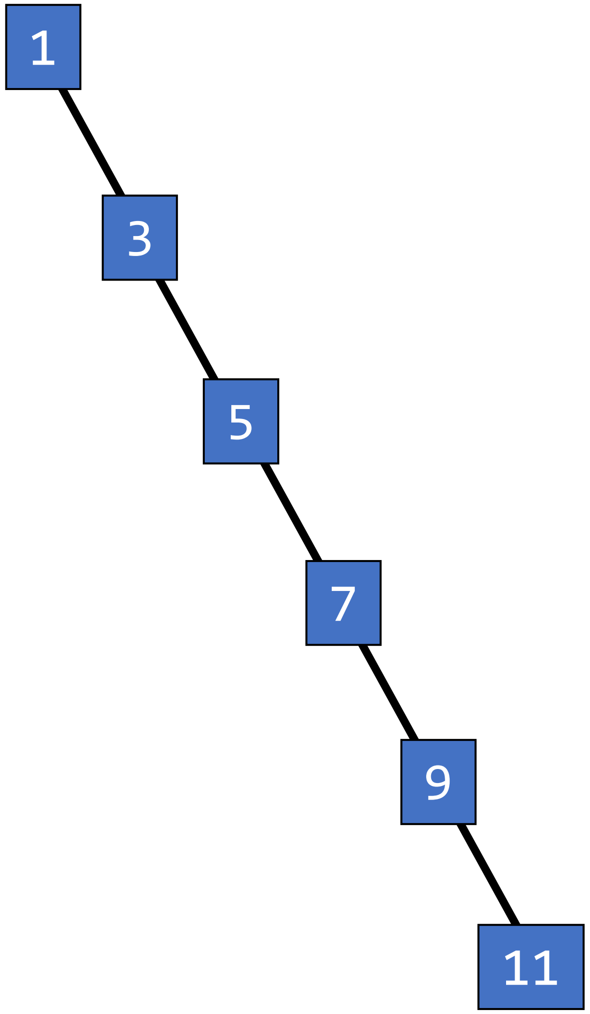
MSets: Implementation #3

```
def multiplicity_of(x, s):  # binary search
    if is_empty_set(s):
        return 0
    elif x == entry(s)[0]:
        return entry(s)[1]
    elif x < entry(s)[1]:
        return multiplicity_of(x, left_branch(s))
    else:
        return multiplicity_of(x, right_branch(s))
```

Time complexity: $O(\log n)$

Balancing trees

- Operation is $O(\log n)$ assuming that tree is balanced.
- But they can become unbalanced after several operations.
  - Unbalanced trees break the log n complexity.
- One solution: define a function to restore balance. Call it every so often.



Question of the Day

- How do we convert an unbalanced binary tree into a balanced tree?
- Write a function `balance_tree` that will take a binary tree and return a balanced tree (or as balanced as you can make it)

MSets: Implementation #3

```
def intersection_of(s1, s2):
    # traversing a BST left-mid-right will give the
    # elements in order

    # homework: Implement like ordered list
```

Time complexity:

$O(i + j)$ *where i and j are number of unique items in the sets*

Comparing Implementations

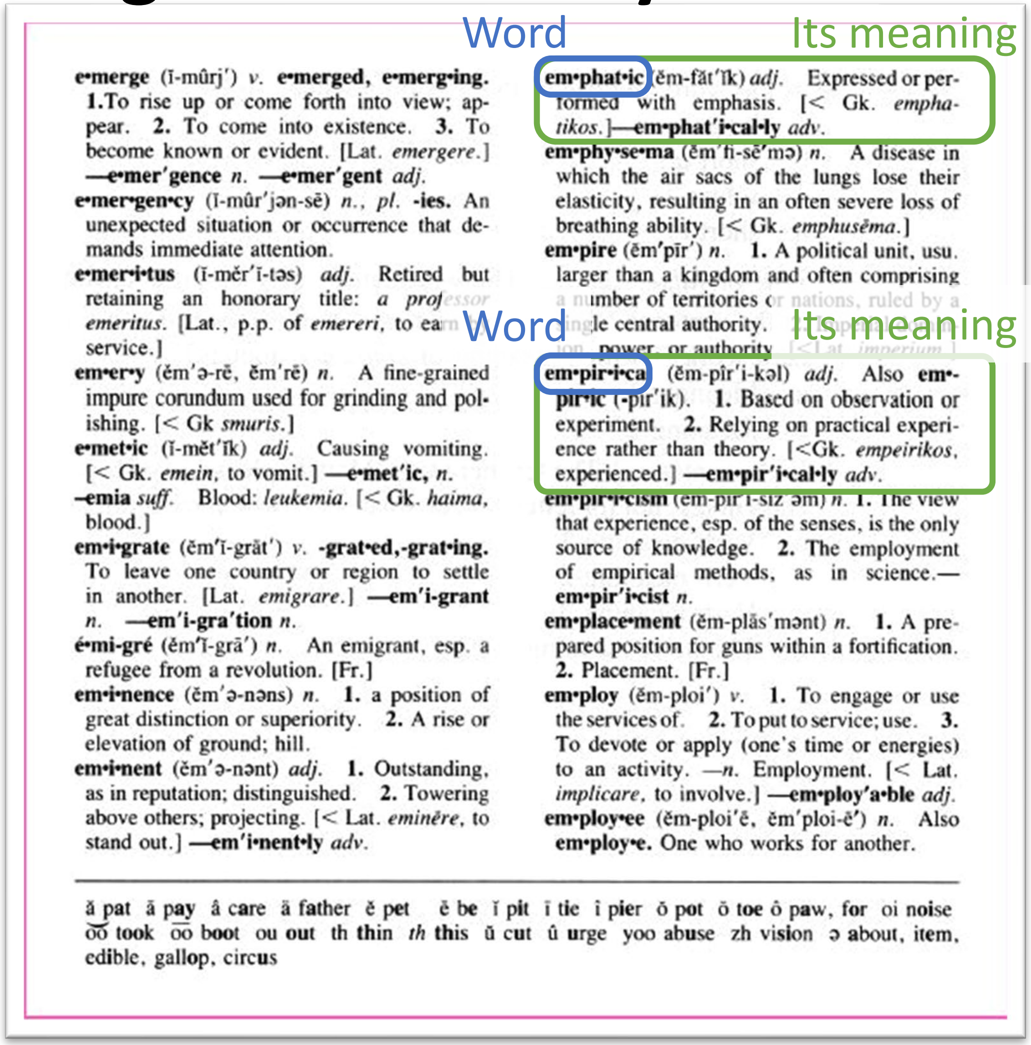
Time Complexity	Unordered List	Ordered List	Binary Search Tree
`adjoin_mset`	append $O(1)$	insert $O(n)$	binary search $O(\log n)$
`multiplicity_of`	linear search $O(n)$	binary search $O(\log n)$	binary search $O(\log n)$
`intersection_of`	complete match $O(n^2)$	merge $O(n + m)$	merge $O(i + j)$

Python Dictionary

- Often convenient to have a data structure that allow retrieval by keyword, i.e. put + get
- Table of key-value pairs. Commonly called Associative arrays
- Python dictionaries use the curly braces { }

Dictionaries

English Dictionary



e·merge (ĭ-mûrj′) v. e·merged, e·merg·ing. 1. To rise up or come forth into view; appear. 2. To come into existence. 3. To become known or evident. [Lat. emergere.] —e·mer′gence n. —e·mer′gent adj.
e·mer·gen·cy (ĭ-mûr′jən-sē) n., pl. -cies. An unexpected situation or occurrence that demands immediate attention.
e·mer·i·tus (ĭ-mĕr′ĭ-təs) adj. Retired but retaining an honorary title: a professor emeritus. [Lat., p.p. of emereri, to earn service.]
em·er·y (ĕm′ə-rē, ĕm′rē) n. A fine-grained impure corundum used for grinding and polishing. [< Gk smuris.]
e·met·ic (ĭ-mĕt′ĭk) adj. Causing vomiting. [< Gk. emein, to vomit.] —e·met′ic, n.
-emia suff. Blood: leukemia. [< Gk. haima, blood.]
em·i·grate (ĕm′ĭ-grāt′) v. -grat·ed, -grat·ing. To leave one country or region to settle in another. [Lat. emigrare.] —em′i·grant n. —em′i·gra′tion n.
é·mi·gré (ĕm′ĭ-grā′) n. An emigrant, esp. a refugee from a revolution. [Fr.]
em·i·nence (ĕm′ə-nəns) n. 1. a position of great distinction or superiority. 2. A rise or elevation of ground; hill.
em·i·nent (ĕm′ə-nənt) adj. 1. Outstanding, as in reputation; distinguished. 2. Towering above others; projecting. [< Lat. eminēre, to stand out.] —em′i·nent·ly adv.
em·phat·ic (ĕm-făt′ĭk) adj. Expressed or performed with emphasis. [< Gk. emphatikos.] —em·phat′i·cal·ly adv.
em·phy·se·ma (ĕm′fĭ-sē′mə) n. A disease in which the air sacs of the lungs lose their elasticity, resulting in an often severe loss of breathing ability. [< Gk. emphusēma.]
em·pire (ĕm′pīr′) n. 1. A political unit, usu. larger than a kingdom and often comprising a number of territories or nations, ruled by a single central authority. [illegible] power or authority. [< Lat. imperium.]
em·pir·i·cal (ĕm-pĭr′ĭ-kəl) adj. Also em·pir·ic (-pĭr′ĭk). 1. Based on observation or experiment. 2. Relying on practical experience rather than theory. [<Gk. empeirikos, experienced.] —em·pir′i·cal·ly adv.
em·pir·i·cism (ĕm-pĭr′ĭ-sĭz′əm) n. 1. The view that experience, esp. of the senses, is the only source of knowledge. 2. The employment of empirical methods, as in science.—em·pir′i·cist n.
em·place·ment (ĕm-plās′mənt) n. 1. A prepared position for guns within a fortification. 2. Placement. [Fr.]
em·ploy (ĕm-ploi′) v. 1. To engage or use the services of. 2. To put to service; use. 3. To devote or apply (one's time or energies) to an activity. —n. Employment. [< Lat. implicare, to involve.] —em·ploy′a·ble adj.
em·ploy·ee (ĕm-ploi′ē, ĕm′ploi-ē′) n. Also em·ploy·e. One who works for another.

ă pat ā pay â care ä father ĕ pet ē be ĭ pit ī tie î pier ŏ pot ō toe ô paw, for oi noise o͝o took o͞o boot ou out th thin th this ŭ cut û urge yoo abuse zh vision ə about, item, edible, gallop, circus

Python Dictionary

Key	Value
'wind'	0
'desc'	'cloudy'
'temp'	[25.5, 29.0]
'rainfall'	{2:15, 15:7, 18:22}

Python Dictionary

```
{key1:value1, key2:value2, ...}

>>> {}  # empty dict
>>> weather = {'wind':0, 'desc':'cloudy',
               'temp':[25.5, 29.0],
               'rainfall': {2:15, 15:7, 18:22}}
```

is equivalent to

```
>>> weather = dict(wind=0, desc='cloudy',
                   temp=[25.5, 29.0],
                   rainfall= {2:15, 15:7, 18:22})
```

Python Dictionary

more dictionary constructors:

```
>>> dict([(1, 2), (2, 4), (3, 6)])
{1:2, 2:4, 3:6}

>>> weather = {{'wind':0, 'desc':'cloudy’,
               'temp':[25.5, 29.0]}

>>> weather['temp’]
[25.5, 29.0]
>>> weather['tomorrow']
KeyError: 'tomorrow'
```

Accessed using key

Python Dictionary

```
>>> 'wind' in weather
True
>>> 0 in weather
False
>>> weather['is_nice'] = True # adds an entry
>>> del weather['temp'] # delete an entry
>>> list(weather.keys())
['desc', 'is_nice', 'wind']
>>> list(weather.values())
['cloudy', True, 0]
```

Checks if key exists

Looping construct

```
>>> for key in weather:
        print(weather[key])
cloudy
True
0
>>> for key, value in weather.items():
         print(key, value)
desc cloudy
is_nice True
wind 0
>>> weather.clear() # delete all entries
```

Msets: Implementation #4

- Representation: dictionary
  - Keys will be the elements
  - Values will be the count

```
def make_mset():
    '''returns a new, empty set'''
    return {}
```

MSets: Implementation #4

```
def adjoin_mset(x, s):
    if x in s:
        s[x] += 1
    else:
        s[x] = 1

def multiplicity_of(x, s):
    return s.get(x, 0)
```

Time complexity: $O(1)$

```
def intersection_of(s1, s2):
  d = {}
  for x in s1:
    if x in s2:
      d[x] = min(s1[x], s2[x])
  return d
```

Time complexity: $O(i)$

Comparing Implementations

Time Complexity	Unordered List	Ordered List	Binary Search Tree	Dictionary
`adjoin_mset`	append $O(1)$	insert $O(n)$	binary search $O(\log n)$	dict access $O(1)$
`multiplicity_of`	linear search $O(n)$	binary search $O(\log n)$	binary search $O(\log n)$	dict access $O(1)$
`intersection_of`	full match $O(n^2)$	merge $O(n+m)$	merge $O(i+j)$	linear search $O(i)$

Multiple representations

- You have seen that for compound data, multiple representations are possible:
  - e.g. multisets as:
    1. Unordered list,
    2. Ordered list
    3. Binary search tree
    4. Dictionary

Multiple representations

- Each representation has its pros/cons:
  - Typically, some operations are more efficient, some are less efficient.
  - “Best” representation may depend on how the object is used.

Typically in large software projects, multiple representations co-exist.

Why?

Many possible reasons

- Because large projects have long lifetime, and project requirements change over time.
- Because no single representation is suitable for every purpose.
- Because programmers work independently and develop their own representations for the same thing.

Multiple representations

Therefore, you must learn to manage different co-existing representations.

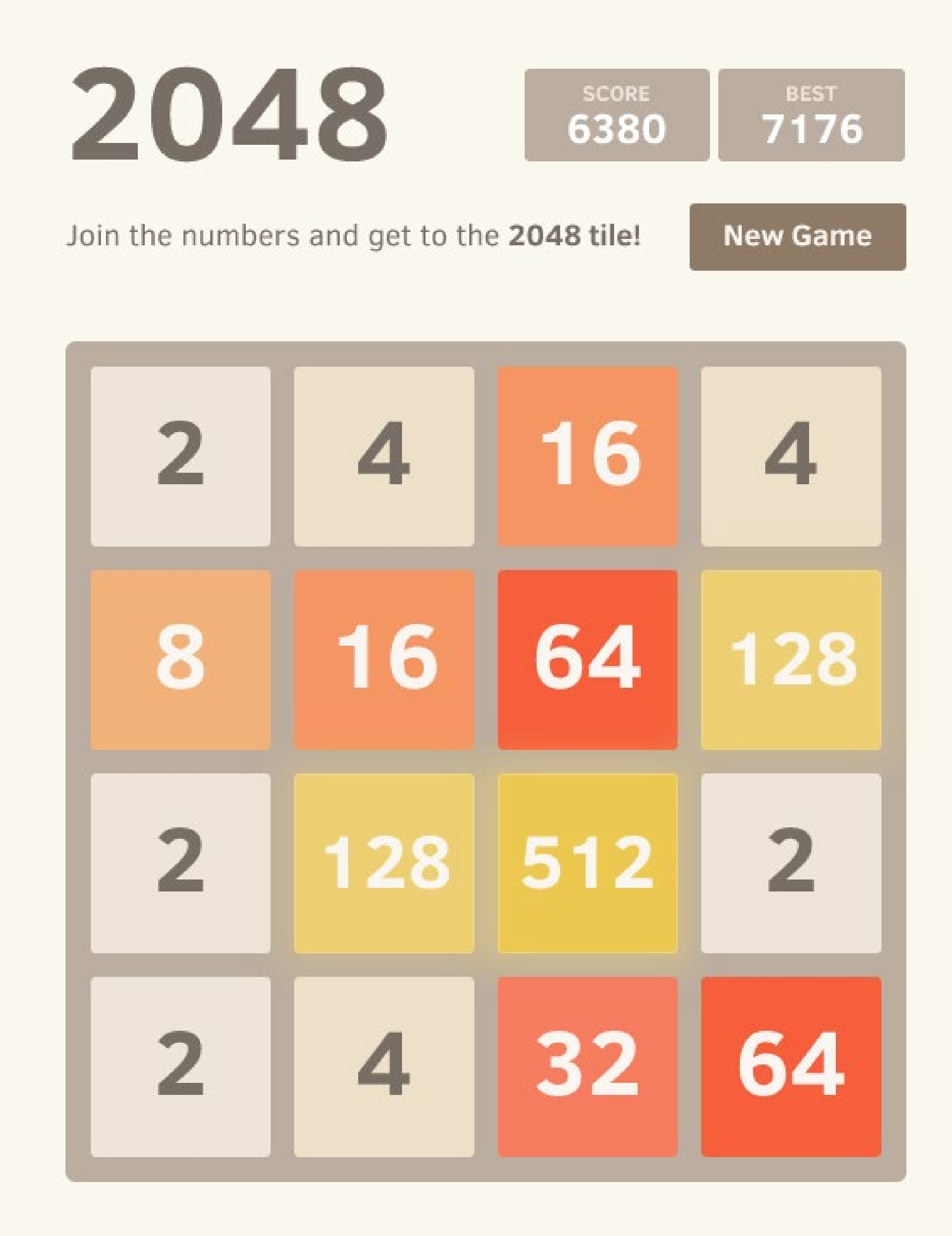
- What are the issues?
- What strategies are available?
- What are the pros/cons?

Summary

- Lots of wishful thinking (top-down)
- Design Principles
  - Specification
  - Implementation
- Abstraction Barriers allow for multiple implementations
- Choice of implementation affects performance!

If you have a lot of time on your hands….

- Play nim (dumb version)
- Re-write nim to allow for arbitrary number of piles of coins
- Write a smarter version of `computer-move`


2048
SCORE
6380
BEST
7176
Join the numbers and get to the 2048 tile!
New Game
2
4
16
4
8
16
64
128
2
128
512
2
2
4
32
64