REMINDER

• Assignment 08 is due Friday, November 22th at 10 AM
  – Please come to office hours early!!! 😊

• Check Piazza about polls for Final Exam Q&A session:
  – Vote for your preferred weekend(s) here.
REVIEW

- Search & Sorting Algorithms
- Dictionaries
- Classes
  - `__init__`
  - `__repr__`
  - `__eq__`
  - User defined class methods
SEARCHING ALGORITHM: LINEAR SEARCH

• Comparing each element of a list (or data structure) one by one with the target, from one end to the other end.

• Best case run time: \( O(1) \)
• Worst case run time: \( O(n) \)
SEARCHING ALGORITHM: BINARY SEARCH

- Searching through a sorted list (or data structure).
- Start from the middle of the original section, and then decide go the next section (bigger/smaller) if middle value isn't the target.

- Best case run time: $O(1)$
- Worst case run time: $O(\log n)$
SORTING ALGORITHMS

• **Selection Sort**
  – Finds the smallest value from an index to the end of the list and swaps them.
  – *Worst run time:* $O(n^2)$

• **Insertion Sort**
  – Divides the list into *sorted* and *unsorted*
  – Place the next value of *unsorted* into its 'rightful' place in the *sorted* section.
  – *Worst run time:* $O(n^2)$

• **Merge Sort**
  – Divide and Conquer
  – Sort two halves of the lists and then re-combine them together
  – *Worst run time:* $O(n \log n)$
Sometimes a list could be very close to its sorted version, such that it looks like a rotation of a sorted list. For example, [4, 6, 1, 3] is a rotation of a sorted list of [1, 3, 4, 6], when every element is shifted by 2 to the right. We'll call this type of sorted list, r-Sorted\(^1\) list.

Write a function, `restore_sorted_list`, that consumes a r-Sorted list and mutates it to a sorted list.

\[
L = [4, 6, 1, 3] \\
\text{restore_sorted_list}(L) \Rightarrow \text{None} \\
L \Rightarrow [1, 3, 4, 6]
\]

Question credited to GeeksForGeeks.com

1. Ina was never one good with naming ...
DICTIONARIES

\[ d = \{\text{key1}: \text{value1}, \text{key2}: \text{value2}, \ldots\} \]

- Each element has a key (a way to look up info) and a value associated with the key

- Unordered collection (with each element being a key-value pair)
  \{\text{keyX}: \text{valueX}, \text{keyY}: \text{valueY}\} == \{\text{keyY}: \text{valueY}, \text{keyX}: \text{valueX}\} => \text{True}

- Like a REAL dictionary (a real dictionary is a word-definition pair; word = key, definition = value)
USEFUL DICTIONARIES FUNCTIONS

- `d[k]` → Get the value of `k`
- `d[k] = v` → Set key-value pair where key = `k` and value = `v`
- `d.keys()` → Creates a view of all the keys in `d`
- `d.values()` → Creates a view of all the values in `d`
- `d.pop(k)` → Removes key-value pair of `k` from `d` and returns the value associated with `k`
- `k in d` → returns `True` if `k` is a key in `d`

Side note: Think about the worst run time for each of these functions.
Write a function `list_multiples` that consumes a string `s` and returns a list in *alphabetical order* containing every character in `s` that appears more than once. Use dictionaries.

Examples:

```python
list_multiples("abcd") => []
list_multiples("bacaba") => ["a", "b"]
list_multiples("gtddyucaadsa") => ["a", "d"]
```
Write a function `xor` that consumes two dictionaries (`d1` and `d2`) and returns a dictionary.  
The returned dictionary will contain all the keys that appear in exactly one of `d1` or `d2` (but not both).  
The value associated with each key will be the same as the one found in the original dictionary.
EXAMPLES

d1 = {1:'a', 2:'b', 3:'c', 4:'d'}
d2 = {5:'e', 6:'f', 7:'g', 8:'h'}

xor(d1,d2) => {1:'a', 2:'b', 3:'c', 4:'d',
               5:'e', 6:'f', 7:'g', 8:'h'}

d3 = {5:'q', 6:'l', 7:'c', 8:'e'}

xor(d2,d3) => {}

d4 = {1:'a', 3:'f', 8:'u', 9:'t'}

xor(d1,d4) => {2:'b', 4:'d', 8:'u', 9:'t'}
CLASSES

- Python’s version of Racket structures
- Allows related information to be grouped together
- We’ll use `__init__`, `__repr__`, and `__eq__` with the class
- We'll also write our own class methods
**__init__** (initialize)

class name:
    def __init__(self, f1, f2, ...):
        self.field1 = f1
        self.field2 = f2
        ...

• Creates an object of this class:

    x = name(field1_val, field2_val, ...)

• Call the fields by:  x.field1

• Racket’s version:

    (define-struct name (field1_val field2_val ...))
    (name-field1 x)
If we try to print a class object, we’d get something like

```
<__main___.name instance at 0x12361c0>
```

We can print a more informative message using the `__repr__` command within the class definition

```
def __repr__(self):
    return "name: {0},{1},..."
    .format(self.field1,
            self.field2,...)
```

Think of `__repr__` as "represents"

Very similar to `__str__`
def __eq__(self, other):
    return isinstance(other, name) and \
    self.field1 == other.field1 and \
    self.field2 == other.field2 and \
    ...
    ...

• It will allow you to compare objects to see if they have same fields:

  x == y => True
class name:
    def __init__(self, f1, f2, ...):
    def __repr__(self):
    def __eq__(self, other):

def foo(self, ...):
    # Access field values: self.field1, ...
    # fn may update field values, use field values
    # for calculations, print information, or
    # return information
CLASS DEFINITION FOR STUDENT

The remaining questions will use the following class:

A **Student** is a class with fields **name**, **faculty**, **program**, **year**, and **courses**

- **name** is a non-empty string representing the student’s full name;
- **faculty** is a non-empty string representing the student’s faculty;
  - Full version: e.g. "Environment" rather than "Env"
- **program** is a non-empty string representing the person’s program (or major);
- **year** is a natural number representing the student’s academic year;
- **courses** is a list of strings representing the courses the student is taking in the current term;
EXAMPLES OF STUDENT OBJECTS:


- \( \text{Dan}_W = \text{Student}("Dan Wolczuk", "Mathematics", "Pure Mathematics", 1, ["MATH 148", "MATH 146", "CS 116"])) 

- \( \text{Logan}_S = \text{Student}("Logan Stanley", "Science", "Chemistry", 1, ["CHEM 120", "MATH 127", "PHYS 111"])) \)
QUESTION 4: ADD_COURSE

Write a class method `add_courses` in the Student class, which consumes a Student object, `self`, and a list of strings, `courses`. It adds the courses in `courses` to the student’s list of courses and prints a message indicating the number of courses the student is now taking.

Examples:

Paul_S.add_courses(["HLTH 230"]) will print "Paul Shen is currently taking 6 course(s)." and adds "HLTH 230" to Paul_S.courses

YQ_W.add_courses([]) will print "Y.Q. Wang is currently taking 3 course(s)." and nothing is added to YQ_W.courses
Write a function `organize_by_year` outside the class, which consumes a list of `Student` objects, `los`, and returns a dictionary where the keys will be natural numbers associating with the students’ years and its associated values is a list of names of the `Student` in the corresponding year.

Example:

```python
L = [Paul_S, Nicole_V, Dan_W, Logan_S]
organize_by_year(L)
=> {1:["Dan Wolczuk", "Logan Stanley"],
    2:["Paul Shen", "Y.Q. Wang"]}
```
Write a function `is_same_faculty` that consumes a non-empty list of students, `los`, and returns `True` if all the students belongs in the same faculty. Otherwise, the method returns `False`.

Example:

Mathies = [YQ_W, Dan_W]

`is_same_faculty(Mathies) => True`

`is_same_faculty([Nicole_V]) => True`

`is_same_faculty([Paul_S, Logan_S]) => False`