CS 1110 Final Exam Solutions, May 2022

1. [8 points] Implement the following function.

```
def send_help(s):
    """
    s is a string of one or more non-empty sequences of dashes and dots
    separated by single exclamation points. Each sequence represents a
    letter in Morse Code (A= '.-', B= '-...', O= '---', S= '...')

    Using this exclamation point encoding, the sign for needing help
    (usually SOS: ...---... ) would be !...!---!...!

    We want to send help even if the SOS signal is 'hidden' among
    other sequences in s.

    Returns True if the sequences for SOS appear in that order even
    if other sequences are present. Otherwise return False.

    Examples:
    send_help('!...!---!...!')              returns True
    send_help('!.!...!.-!---!.!...!')       returns True
    send_help('!...!...!---!')              returns False (wrong order)
    send_help('!....!...!---!')             returns False ('....' is H)
    send_help('!...!...!.---!')             returns False ('.---' is J)
    """
```

Notice that one’s code also needs to work correctly on patterns like ’!—!...!—!...!’ (”OSOS”) and ’!...!—!—!...!’ (”SOOS”)

```
    if s.count("!...!") < 2 or s.count("!---!") < 1:
        return False
    i = s.find("!...!")
    i = s.find("!---!",i+1) #+1 is not specific but works
    if i == -1:
        return False
    i = s.find("!...!")
    if i == -1:
        return False
    return True
```

Alternate while-loop solution by student Annabelle A. (with slight edits)

```
    letters = s.split('!')
    sos = ['...', '---', '...']
    i = 0
    k = 0
    while i < len(letters) and k < len(sos):
        if letters[i] == sos[k]:
            k += 1
        i += 1
    return k == 3
```

Alternate loop solution by student Kimberly R.C. (with slight edits):

```
    letters = s.split('!')
    signal = ['...', '---', '...']

    for x in letters: # suggest not using "l" (ell) for variables, since looks like "1" (nu
        if x == signal[0]:
            signal.pop(0)
            if signal == []:
                return True
    return False
```

Alternate loop-based solution by Patrick C. (with slight edits)

```
    signal = ['!...!', '!---!', '!...!']
    start_i = 0
    for target in signal:
        i = s.find(target, start_i)
        if i == -1:
            return False
        start_i = i + len(target) -1
    return True
```

Alternate track-the-s-indices solution by Josh D. (with slight edits):

```
    letters = s.split('!')

    s_locations = []
    for i in range(len(letters)):
        if letters[i] == '...':
            s_locations.append(i)
    if len(s_locations) < 2:
        return False

    for j in range(s_locations[0]+1, s_locations[-1]):
        # searching between 1st and last s
        if letters[j] == '---':
            return True
    return False
```

Alternate solution, inspired by solution by Yixi C., that looks for an ”O” that is after the first ”S” and that has an ”S” after it.

```
oh_locations = [s.find('!---!')]
last_oh_location = oh_locations[0]
while -1 not in oh_locations and last_oh_location < len(s):
    oh_index = s.find('!---!', last_oh_location+1)
    oh_locations.append(oh_index)
    last_oh_location = oh_index
sos = False
for oh in oh_locations:
    first_s= s.find('!...!')
    if first_s != -1 and first_s < oh and s.find('!...!', oh + 1) > oh:
        # found an "O" that is after the first "S" and has an "S" afterwards
        sos = True
return sos
```

2. The function `encrypt` below should transform an input string (`plaintext`) into an *encrypted* string (`ciphertext`). Sadly, **there are multiple bugs in the code below. Read the specifications carefully; then, identify and fix the bugs.**

   The encryption uses the *Fibonacci sequence*. The Fibonacci sequence is defined recursively in the docstring of `fib()`, but the implementation uses a while loop. The first few numbers in the sequence are `1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...`

   As shown in the following table, the n^{th} letter in the alphabet is mapped to the n^{th} Fibonacci number:

n	1	2	3	4	5	6	7	8	...	26
Letter	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**...**	**Z**
Fibonacci Number	1	1	2	3	5	8	13	21	...	121393

   For example, "A", the first letter in the alphabet, is mapped to the first number in the sequence (fib(1) = 1), "B", the second letter in the alphabet, is mapped to the second number in the sequence (`fib(2) = 1`), "C" is mapped to the third (`fib(3) = 2`), "D" is mapped to the fourth (`(fib)(4) = 3`), etc.

   The concatenation of all of these numbers (separated by spaces) becomes the ciphertext. Thus, encrypting the plaintext "fade" should result in the ciphertext "8 1 3 5" because "F" maps to 8, "A" maps to 1, "D" maps to 3, and "E" maps to 5.

```python
def fib(n):
    """
    Returns the nth Fibonacci number.
    Precondition: n > 0

    fib(1) = 1
    fib(2) = 1
    for n > 2 :
       fib(n) = fib(n-1) + fib(n-2)
    """
    if n == 1 or n == 2:
        return 1

    prev_prev = 1
    prev = 1
    curr_fib = prev_prev + prev
    counter = 3

    while counter <= n:
        prev_prev = prev
        prev = curr_fib
        curr_fib = prev_prev + prev
        counter += 1

    return curr_fib
```

```python
def encrypt(plaintext):
    """
    Encrypts the plaintext using the following rule:
       each (capitalized) letter of the plaintext is mapped
       to the corresponding element of the Fibonacci sequence.

    The concatenation of all these numbers (separated by
       spaces) becomes the ciphertext.

    Returns the ciphertext.

    Precondition: plaintext contains only letters
    """
    alphabet = "!ABCDEFGHIJKLMNOPQRSTUVWXYZ" # start with '!'
                                          # so letters start at index 1
    ciphertext = ""
    plaintext_index = 0
    while plaintext_index < len(plaintext):
        curr_letter = plaintext[plaintext_index]
        alpha_index = alphabet.index(curr_letter)
        ciphertext += str(fib(alpha_index)) + " "

        plaintext_index += 1

    return ciphertext
```

(a) [5 points] Consider the following call to `encrypt` and the Python error it triggers.

```
>>> print(encrypt("fade"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "cipher.py", line 47, in encrypt
    alpha_index = alphabet.index(curr_letter)
ValueError: substring not found
```

Below, explain where (a single line number) and why this error is triggered. **And**, write below the correct version of the line.

`curr_letter` is lowercase, but there are no lowercase letters in `alphabet`. The correct version of line 47 is

```
alpha_index = alphabet.index(curr_letter.upper())
```

[Alternate solution] Could also modify line 46:

```
curr_letter = plaintext[plaintext_index].upper()
```

Aside: **No** changes were needed in setting up `alphabet` (line 41) or `plaintext_index` (line 44).

The presence of '!' at the beginning of `alphabet` is a "padding" technique we also saw in the cards/Blackjack labs. With the padding by some non-letter character (doesn't matter whether it is a '!' or a number or whatever), the index of 'A' in `alphabet` is 1, not 0, and this matches nicely with the fact that the first valid argument for `fib` is $n = 1$. Similarly, the index of 'C' in `alphabet` is 3, not 2, and its encoding is `fib(3)`, and so on.

`plaintext_index` in an index into `plaintext`, and so needs to start at 0 (otherwise, we would miss encrypting its first letter.)

We do not want to lowercase `alphabet` because doing so would break encryption for capitalized inputs, which are allowed by the precondition to `encrypt()`.

(b) [5 points] After the first bug (above) is fixed, the call

```
>>> print(encrypt("fade"))
```

should print the following string: `''8 1 3 5''`.

Instead, it prints: `''13 1 5 8''`.

Below, explain where (a single line number) and why this problem is triggered. (Hint: three of the Fibonacci numbers look wrong; where are those calculated for each letter?). **And**, write below the correct version of the line.

'f', 'd', and 'e' map to the Fibonacci number AFTER the one it should. Looking at the while loop in `fib`, we see that the boolean condition in the while-loop header is true for one too many iterations. Thus, the error is on line 19 and the correct version is `while counter < n:`

Subtracting 1 from `alpha_index` in line 47 isn't valid because it risks breaking the precondition on `fib()`.

(c) [5 points] After the two bugs above are fixed, consider the following call to `encrypt` and the Python error it triggers.

```
>>> print(encrypt("attack@6:30"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "cipher.py", line 47, in encrypt
    alpha_index = alphabet.index(curr_letter)
ValueError: substring not found
```

Is this a bug in the code? If so, identify the issue and provide the correct version of the line. If not, explain why.

This is not a bug in the code — the input violates the precondition to encrypt (i.e., the plaintext is supposed to contain only letters).

3. [15 points] Implement the following function, making effective use of for-loops.

```
def extract_maxes(values):
    """
    values: a non-empty list of non-empty lists of ints >= 0
       (no need to enforce these preconditions)

    Modifies `values` as follows:
       Removes the largest value from each list of ints in `values`.
       If there is a tie (2-way or more), removes the last occurrence
          of the largest value in that list and ignores the earlier occurrences

    Return: a list containing the removed entries in their original order

    Ex: values = [ [4, 0, 12], [20], [4, 1, 2] ]
    Modified values = [ [4, 0], [], [1, 2] ]
    Returns [12, 20, 4]

    Ex: values = [ [12, 40, 16, 8], [30, 21, 30, 24] ]
    Modified values = [ [12, 16, 8], [30, 21, 24] ]
    Returns [40, 30]
    """
    # You may NOT use the built-in max() function.

    # BEGIN REMOVE
    removed = []
    for int_list in values:
        max_val = 0  # begin with smallest, positive value
        max_loc = -1 # 0 is not associated with an actual value in list
        for i in range(len(int_list)):
            if int_list[i] >= max_val: #  > (strictly greater than) is wrong.
                max_val = int_list[i]
                max_loc = i
        removed.append(max_val)
        int_list.pop(max_loc) # could also use del
    return removed
```

Some approaches that don't work:

- It doesn't work to sort a sublist of `values` to make it easy to find its max. The problem is that the original order of the sublist would need to be restored (it would be *bad* to change the order of the sublists, because the specification does not state that that could happen.)
- One cannot just use the list `remove()` method. The problem is that it removes the *first* occurrence of something, not the last.
- The following *does* work to remove an item at index `ind` from sublist `values[i]`:

```
row = values[i]
values[i] = row[:ind] + row[ind+1:]
```

However, the following will only change local variable `row`, *not* the input `values`, as was asked for by the problem:

```
row = values[i]
row = row[:ind] + row[ind+1:]
```

4. [15 points] In this question, we will model the spread of fake news through a social network. This network is composed of `Person` objects with the following attributes:

   - `name [str]` - unique non-empty name of this person
   - `following [list of Persons]` - people this person follows
   - `fact_checker [boolean]` - `True` if this person always fact checks their news before sharing. Otherwise (if the person shares news without doing their own research), `False`.

   Assume no two people share the same name, and no person can reach themselves by tracing through the following network.

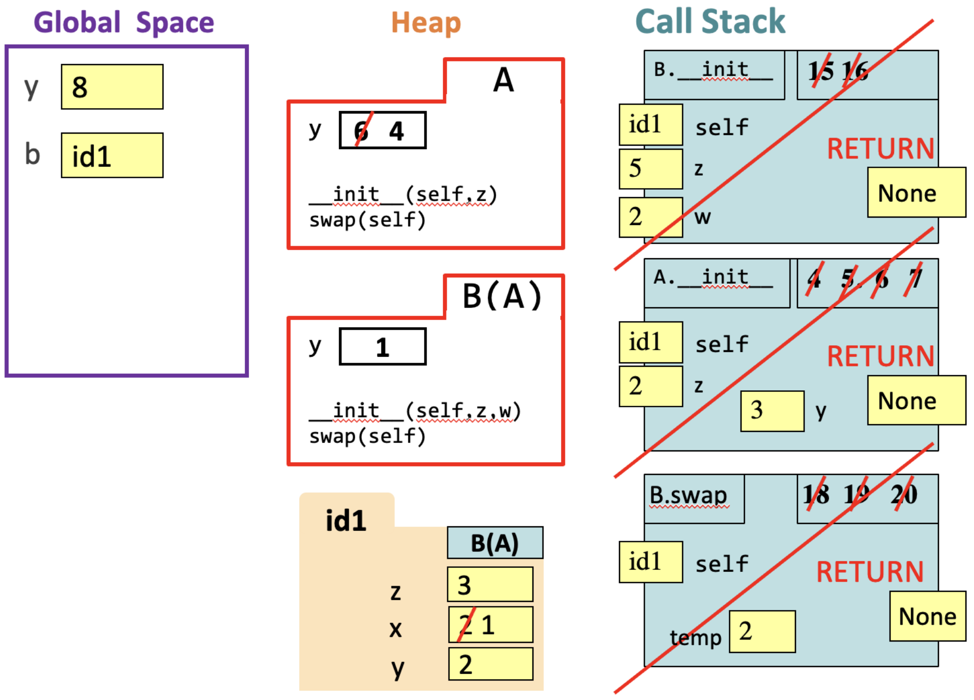
   Implement the function below, making effective use of recursion.

```
def find_spreaders(root):
    """ Returns a (possibly empty) list of names of persons in root's network
        (including root) who don't fact-check news. (They spread fake news.)

        Example:
           a = Person("Alice", [], True)
           b = Person("Bob", [], False)
           c = Person("Caitlin", [a, b], False)

           find_spreaders(a)   returns []
           find_spreaders(b)   returns ["Bob"]
           find_spreaders(c)   returns ["Caitlin", "Bob"]

    Precondition (no need to assert): root is a Person """

    # Note: the original question said to return a "list of persons" instead of
    # a "list of the names of persons". Credit was given for returning a
    # list of persons OR a list of person's names

    spreaders = []
    if not root.fact_checker:
        spreaders.append(root.name)
    for f in root.following: # if root.following is [] the loop is never entered
        more = find_spreaders(f)
        for m in more:  # can replace for-loop with spreaders += more
            spreaders.append(m)
    return spreaders

    # Alternate solution (treats root.following == [] as a special case)

    if root.following == []:
        if root.fact_checker:
            return []
        else:
```

```
        return [root.name]

spreaders = []
if not root.fact_checker:
    spreaders.append(root.name)
for f in root.following:
    spreaders += find_spreaders(f)
return spreaders
```

5. [20 points] Execute the script below; draw the global space, the call frames, and the heap space (including both class folders and object folders). We drew the first class folder for you. For method call frames, give the method name as ⟨class name⟩.⟨method name⟩(), since we need to know which class's method is being called. Don't forget to draw the call frames for `__init__`.

```
1  class A:
2      y = 6

3      def __init__(self, z):
4          self.x = 2
5          y = self.y + z
6          A.y = y + 1
7          self.swap()

8      def swap(self):
9          temp = self.x
10         self.x = self.y
11         self.y = temp

12 class B(A):
13     y = 1

14     def __init__(self, z, w):
15         self.z = y - z
16         super().__init__(w)

17     def swap(self):
18         temp = self.x
19         self.x = self.y
20         self.y = temp

21 y = 8
22 b = B(5,2)
```



6. [10 points] This question simulates managing schools (groups) of fish in fish tanks. Here is the docstring for a new class `School`.

```
class School:
    """Instance attributes:
         fish_type [non-empty str]: type of the fish (e.g., "angelfish")
         count [int >= 0 ]: number of fish in this school
    """
```

And here is the docstring for a new class `FishTank`.

```
class FishTank:
    """Instance attributes:
          my_fish [School]: the school that lives in this fish tank
             a fish tank can contain only one school at a time
          CAPACITY [int >= 0]: max number of fish that can fit in the tank
             Once initialized, this value should not change.
    """
```

Implement the `migrate_fish` method of class `FishTank` so that it meets its specification.

```
    def migrate(self, tank2, n):
        """Move n fish from self to tank2.
           (counts of both schools should change accordingly)

        Preconditions (no need to assert):
         - the two tanks have the same fish_type
         - tank2 is a FishTank
         - n is a positive int

        Move n fish, but also only as many fish as the donor school can offer
        (based on its count) and also only as many as the receiving tank can accept
        (based on the count of its current school and capacity of the tank).
        """

        room_in_tank2 = tank2.CAPACITY - tank2.my_fish.count
        if room_in_tank2 < n:
            n = room_in_tank2 # migrate only the amount we have space for
        if self.my_fish.count < n:
            n = self.my_fish.count # migrate only the num fish we have to give

        # Alternate to the code block above:
        # min(n, self.my_fish.count, tank2.CAPACITY - tank2.my_fish.count)

        self.my_fish.count -= n
        tank2.my_fish.count += n
```

Alternate while-loop-based solution:

```
while self.my_fish.count > 0 and tank2.my_fish.count < tank2.CAPACITY and n > 0:
    self.my_fish.count -=1
    tank2.my_fish.count += 1
    n -= 1
```

7. [20 points] In this question, we begin with a new `Message` class defined as follows:

```python
class Message:
    """Instance Attributes:
        author [non-empty str]: username of the author of the message.
           No two authors can have the same username.
        content [non-empty str]: content of this message
        likes [int >= 0]: number of likes the message has. Initially 0.
        dislikes [int >= 0]: number of dislikes the message has. Initially 0.
    """

    def __init__(self, a, c):
        """Creates a new Message with:
            author a, content c, 0 likes and dislikes

        Preconditions (no need to enforce):
            a: non-empty str
            c: non-empty str
        """
        self.author = a
        self.content = c
        self.likes = 0
        self.dislikes = 0

    def calculate_score(self):
        """
        Returns the score of this message
        """
        return self.likes - (self.dislikes//2)
```

A `Post` is a type of `Message`. `Posts` are different from messages in 3 ways:

1. They have titles.
2. Users can leave comments on a post.
3. Authors may pay to promote their post, which has the effect of boosting the post's score.

Implement the class `Post` below according to the provided specifications. Do not worry about enforcing preconditions.

```
class Post(Message):
    """Class attributes:
        promo_count [int]: number of posts currently promoted, initially 0
        PROMO_MAX: the max number of posts that can be promoted, set to 10

    Instance attributes:
    Includes those of Message. And also:
      title [non-empty str]: title of this post
      comments [list of Messages]: comments on this post. Initially empty.
      is_promoted [bool]: Whether the post is promoted. Initially False.
    """

    PROMO_MAX = 10
    promo_count = 0

    def __init__(self, a, c, t):
        """Creates a new Post with author a, title t, content c,
           0 likes and dislikes, empty coments, is_promoted set to False.

        Preconditions (no need to enforce):
            a: non-empty str
            t: non-empty str
            c: non-empty str
        """

        super().__init__(a, c)
        self.title = t
        self.comments = []
        self.is_promoted = False
```

A student pointed out that if the score is negative, a promoted post's score is twice as negative. This doesn't make much real-world sense, but the exam asked students to implement according to spec.

```
    def calculate_score(self):
        """ Returns the score [int] of this post

        If the post is promoted, the score is doubled
            Otherwise, the score is calculated the same as any other Message.
        """

        message_score = super().calculate_score()
        if self.is_promoted:
            return message_score * 2
        else:
            return message_score
```

```
def is_controversial(self):
    """ Return: True if this post is controversial; otherwise, False.

    A post is considered controversial if the number of comments is greater
    than the Post's score.
    """


    return len(self.comments) > self.calculate_score()

def promote(self):
    """
    Promotes this post (incrementing the number of promoted posts appropriately)

    Returns True if the promotion was successful, otherwise False

    A promotion cannot be performed if:
        1) the post is already promoted
        2) the post is controversial or
        3) The total number of promoted posts would be greater than PROMO_MAX
    """

    if self.is_promoted or self.is_controversial() or \
      Post.promo_count >= Post.PROMO_MAX:
        return False
    else:
        Post.promo_count += 1
        self.is_promoted = True
        return True
```

8. For each question, provide **only one answer**. If you provide 2, we will only grade the first.

   (a) [2 points] Which of the following statements about **while loops** is **false**?

   (A) While loops are well-suited to tasks where the exact number of iterations is unknown up front.

   (B) If you define a loop variable (like, `x`, a list element) at the start of a while loop, Python will update its value per iteration.

   (C) While loops introduce the possibility of being stuck in an infinite loop.

   (D) If a while-loop has header

   ```
   while not_raining:
   ```

   Python will check whether the variable `not_raining` is true before each iteration.

   (E) If you intend to remove a list element when iterating over a list, it's better to do this with a while loop than a for loop.

   Correct Answer: B

   (b) [2 points] Which of the following statements about **Binary Search** is **false**?

   (A) Binary Search is faster than Linear Search.

   (B) With each step of Binary Search, you can rule out half of the search space.

   (C) In Binary Search, doubling the size of the input list does *not* double the expected time of the search.

   (D) Binary Search works on any list, sorted or not.

   (E) Binary Search's complexity is on the order of $log_2 n$.

   Correct Answer: D

   (c) [2 points] What is the "work" that is measured when calculating the runtime complexity of Merge Sort? The work is measured by the number of...

   (A) calls to `merge`.

   (B) calls to `merge_sort`.

   (C) comparisons of individual elements in the list.

   (D) times the list is split in half.

   (E) elements in the list.

   Correct Answer: C