# CSC 120 FINAL EXAM REVIEW GUIDE

**Exam 3PM:** May 10, 2022 3:30-5:30

**Exam 4PM:** May 6, 2022 3:30-5:30

**Review Session:** May 4, 2022 5:30-6:50

Gittings 129B

## TOPIC LIST:

1. Python Review
2. References
3. Linked Lists
4. Classes and Objects
5. Recursion
6. Trees
7. Debugging
8. Big O

## TABLE OF CONTENTS:



## WHERE TO STUDY:

- Slide-decks
- Previous study guides
- Previous exams

## QUESTIONS?

- Ask During Feedback Meetings
- Ask on Discord
- Office Hours

# REVIEW PROBLEMS:

Note: these are not necessarily the questions that will be on your test. These questions were written collectively by the TAs (who haven't seen the test yet) and resemble what we think you need practice with for the test. Do not use this as your only resource for studying.

That being said, feel free to jump questions, and do the ones you think will help you the most. Ask questions on Discord, and attend a review session for additional clarification.

1. What is the difference between the == operator and the `is` comparator?

2. Give an example of a situation where using an array of elements would be better than using a linked list.

3. Give an example of a situation where using a set of elements would be better than using an array, linked list, dictionary, etc.

4. Given an array of tuples (each tuple containing two integers), make a function called `max_product(array)` which finds the maximum product of each tuple, and returns **only** the tuple pair that generates that product.

```
max_product( [(1, 5), (7, 6), (5, 2), (10, 3), (4, 9)] ) =>
(7, 6)
```

**5.** Draw a picture of all the variables as they stand after the code completes

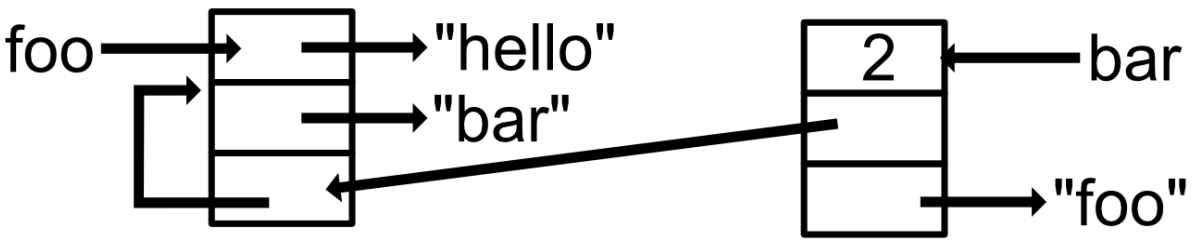
(Draw a reference diagram)

```
x = 5
y = x
y += 10

retval = [0, 1, 1]
for i in range(5):
      retval.append(retval[-1] + retval[-2])

retval[0] = y
retval[3] = x
retval[5] = [x, y]
retval[7] = retval[3]
retval[2] = retval[5]
retval[5] = retval[5] + [6]

retval = retval[-6:6]
```

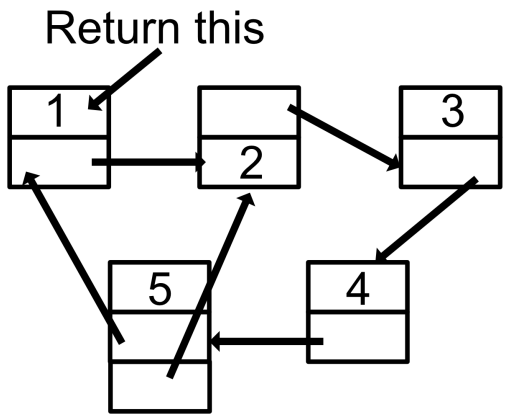
**6.** Given the following code, draw out the reference diagram

```
x = []
y = 10
for i in range(1, 5):
    x.append(y % i)
x.append([x, y])
x.append(x[0] + x[1])
z = (y * x[-3]) % len(x)
x[z] = x
```

**7.** Write a snippet of code that creates the following reference diagram. The diagram does not contain any linked lists.



8. Create the function `get_shape()` which returns the following data structure. The diagram does not contain any linked lists.




9. Make a function called `partition_linked_list(head, target)` which takes the head of a linked list of integers, and an integer as target. Return a new linked list in which every value less than the target comes before every value greater than or equal to the target. The order of the numbers in each partition of the list does not matter. The target integer may not be an integer within the linked list.

10. Write a function `my_func(head)` which takes a head to a linked list. If the length of the list is odd, remove the middle value. If the length of the list is even, remove the second-to-last value. Return the head of the list.

11. Create the function `ll_to_rev_arr(head)` which takes the head of a linked list and returns the linked list contents as a reversed array.

    Example:
    input: 1 -> 2 -> 3 -> 4
    output: [4, 3, 2, 1]

12. Create the function `is_sorted(head)` where head is the head of a linked list and returns `True` if the values are in ascending order (duplicates are allowed). The function should return `False` if the items are not sorted.

**13.** Create a class that implements a doubly linked list (each node has a pointer to the next node, as well as a pointer to the previous node). Then implement the following functions:

- `add(head, value)`: adds a node to the linked list, doesn't return anything
- `delete(head, value)`: deletes a node with value from the linked list, return -1 if node not found, 0 otherwise
- `find(head, value)`: finds a node with value from the linked list and returns true if it exists in the linked list, false if it does not.

Assume that head points to the head node of the linked list

**14.** Write a `PlateStack` class which allows us to keep track of a stack of plates. It should have the `add(self, size)` method which allows us to add a plate of a specified size to the top of the stack. It should also have a `remove(self)` method which removes the topmost item from the stack and returns that plate's size. Your implementation should use an array where you add/remove plates from the beginning of the array.

**15.** Rewrite `PlateStack`, but using a Linked List. You should add/remove plates from the end of the list.

**16.** You work for a movie theater and the computer running the ticketing software was stolen. The next big hit movie *Snakes VS Coffee* is coming out tomorrow and you need a solution for getting people their tickets. As a result, you are tasked with creating a new system for adding people to a waiting list and giving tickets. Create a `TicketSoftware` class which holds a queue of people's names. Use the `add(self, name)` method to add someone to the end of the queue. Use `get_next(self)` to get the next person's name from the queue.

**17.** What are the 2 required elements for any recursive function?

**18.** Given two integers, create a function `count_occur(num1, num2)` which counts the number of occurrences of num2 in num1.

Given: count_occur(2222, 22), return 2

count_occur(0, 1), return 0

count_occur(8492378378910378, 378), return 3

**19.** Write a recursive function `extract_multiples_of(root, val)` which takes in the root of a tree(doesn't have to be a BST) and a value and returns all the values that are a multiple of that value. This must be returned as an array the order of which doesn't really matter. Helper functions are not allowed.

Example: Suppose my tree has the values: 2, 3, 4, 5, 8, 9, 81, 27, 25, 63, 56, 77

I give in the val = 9

Output: [9, 81,27,63]

**20.** Write a recursive function `product_digits(val)` that takes in a number and returns the product of all the digits in that number.

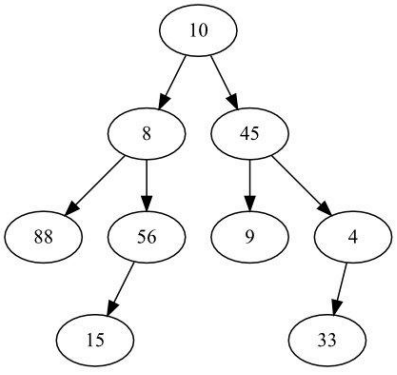
Given: product_digits(657893), return 45360

**21.** Write a recursive function which takes an array (python list) and prints each element on a separate line in reverse order.

**22.** How must a data structure be formatted in order to be a tree? What restrictions do you have to add to make it a binary tree, or a BST?

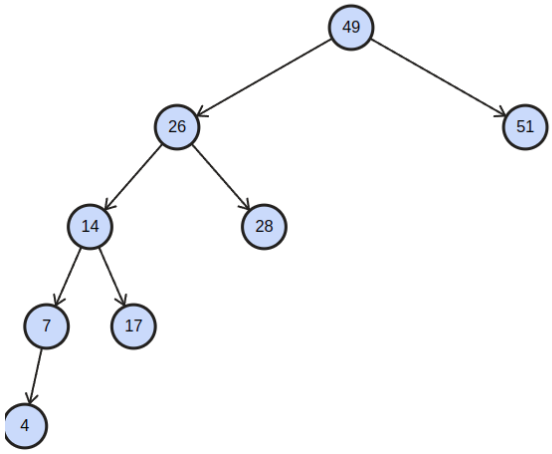
**23.** Given the following Binary Tree (not Binary Search Tree!), write out the following traversals:

- In-order traversal
- Pre-order traversal
- Post-order traversal
- Breadth-first traversal

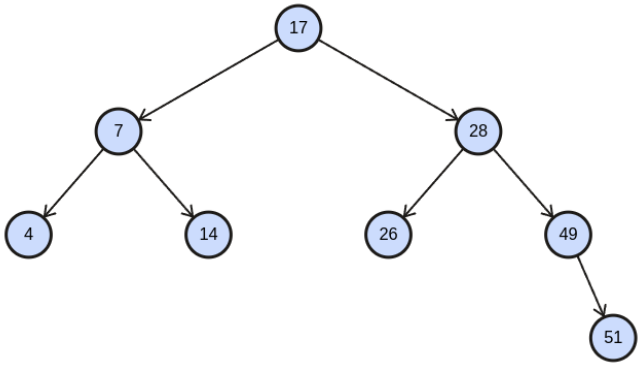


**24.** Write a function called `arr_to_balanced_bst(arr, start=0, end=-1)` which takes an array of integers and returns the root of a balanced binary search tree from its values. A balanced BST is a BST in which the middle most value of the array is the root. Everything less than the middle will be in the left child of the root, and everything greater the middle will be in the right child of the root (you may assume there will be no duplicate values). This pattern recursively continues through the children node, leaving you with a BST that has the minimum possible height.

Example of Unbalanced BST:




Example of Balanced BST (using the same values):



**25.** What is the name of the error produced in this code:

```
arr = [10, 22, 36, 40, 19, 72]
for index in range(len(arr)):
        arr[index] = arr[index + 1]
```

**26.** The following function intends to print out every other value in a Linked List by iterating through it. However, there is a bug. What is the bug and what inputs will cause the program to crash? How do you fix it?

```
def print_every_other(head):
    curr = head
        while curr is not None:
            print(curr.val)
            curr = curr.next.next
```

**27.**

Identify all the bugs in the following class

```
class MyClass
    def __init__():
        self.x = 10

    def compute_product(y):
        self.prod = x * self.y
        return self.prod
```

**28.** Describe what Big-O is and why we use it.

**29.** Give the Big O cost for the following functions:

a)
```
def func1(arr):
    total = 0
    for elem in arr:
        for i in range(len(arr)):
            total += elem + arr[i]
    return total
```

b)
```
def func2(arr):
    total = 0
    for elem in arr:
        for i in range(100):
            total += elem + i
     return total
```

c)
```
def func3(arr1, arr2):
   both = []
   for i in range(len(arr1)):
       both.append(arr1[i])

   for elem in arr2:
       both.append(elem)
   return both
```

d)
```
def func4(arr):
    if arr == []:
        return 0
    return arr[1:] + arr[0]
```

e)
```
def func5(n):
    total = 0
    while n > 0:
        total += n % 10
        n = n/10
    return total
```

30. Is appending to a Python list always O(1)? Why or why not?

31. What is the runtime for searching a BST for a given value? What if the tree was not a BST?

32. List the following Big-O notations from fastest to slowest:
    - O(logn)
    - O(n^50)
    - O(n)
    - O(2^n)
    - O(1)
    - O(n^2)

33. What are the Big-O complexities of `PlateStack.add(self, size)` and `PlateStack.remove(self)` from questions 14 and 15? Is there a way to improve their runtimes?

34. What are the steps in writing a program? What happens in each of these steps?

35. What is an abstract data type? List some examples

36. Create the function `reverse_arr(items)` which takes an array and returns a new array with the items reversed. You must use a stack in your implementation.

37. **Challenge question:** Do question 21 without using negative indexes (or by calculating the negative index).

38. **Challenge question:** Explain why using the Python “in” keyword on a set is O(1).

39. **Challenge question:** What is the following Code doing?

```
class thing:
    def __init__(x1, x2):
        if not x2:
            x1.x2 = 0
            x1.prev = None
        else:
            x1.x2 = x2
            x1.prev = thing(x2-1)

    def __str__(prev):
        return str(prev.x2) + " " + ["",str(prev.prev)][not not
prev.prev]
```

40. **Challenge question:** How long is Russ' beard?

# Solutions

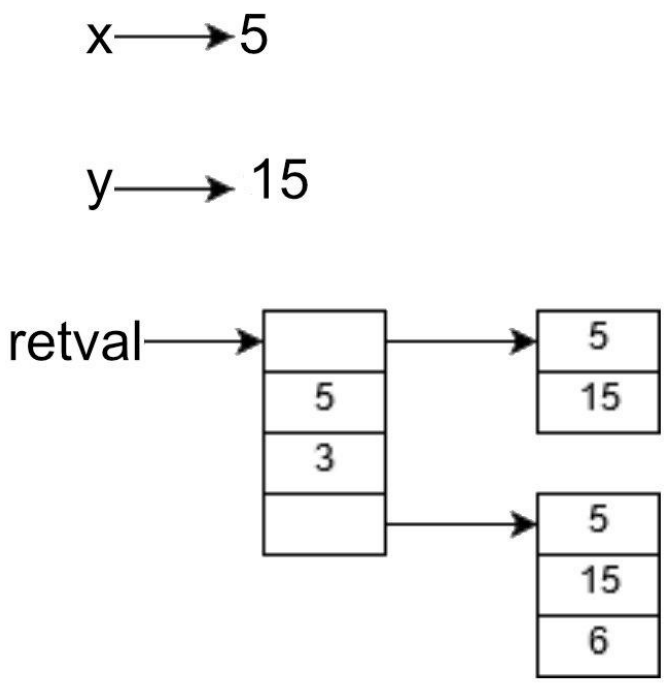
1. The == equals compares the values of the items passed on both sides. The `is` keyword compares the location of the values stored in memory.

2. An array is more useful whenever we need to repeatedly access elements in the middle of the list, or if we want to access elements by their index into the list. One example of this would be a 2D grid of elements for a map. We want to be able to access parts of the map using their coordinates.

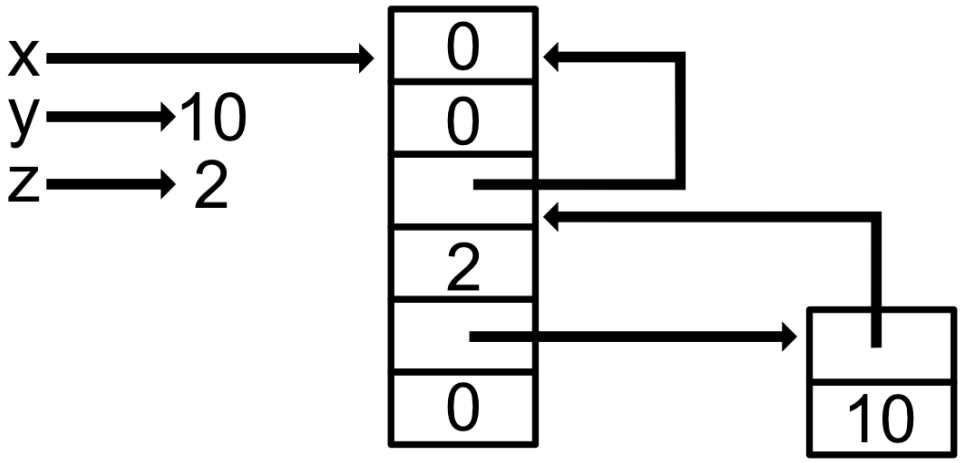
3. Sets are extremely useful for determining if something is in the set, and adding things to the set. One good example of this would be keeping track of locations we have visited if we are traversing through a data structure. For example, we could be traversing a maze. We can store each location we have been to in a set, and every time we find a new location we can check if it is in the set. If it is in the set, then ignore it, otherwise we can traverse through it and add it to the set of visited locations.

4. 
```
def max_product(array):
      max1 = array[0][0]
      max2 = array[0][1]
      max_prod = max1 * max2
      for tup in array:
                  temp1 = tup[0]
                  temp2 = tup[1]
                  prod = temp1 * temp2
                  if prod > max_prod:
                              max1 = temp1
                              max2 = temp2
                              max_prod = prod
      return(max1,max2)
```

**5.**




**6.**




**7.**
```
foo = ["hello", "bar", None]
bar = [2, None, "foo"]
foo[-1] = foo
bar[1] = foo  # You will lose points if you do bar[1] = foo[-1]!
```

8. This is just one way of solving this problem. There are many other ways - including ones that take less lines of code

```
def get_shape():
    # Create each list
    one = [1, None]
    two = [None, 2]
    three = [3, None]
    four = [4, None]
    five = [5, None, None]

    # Then chain them all together
    one[1] = two
    two[0] = three
    three[1] = four
    four[1] = five
    five[1] = one
    five[2] = two

    # We return the *WHOLE* list
    # You will lose points if you return one[0]!
    return one
```

9. 
```
def partition_linked_list(head, target):
      before = ListNode(None)
      after = ListNode(None)
      before_cur = before
      after_cur = after

      while head is not None:
            if head.val >= target:
                  after_cur = ListNode(head.val)
                  after_cur = after_cur.next
            else:
                  before_cur = ListNode(head.val)
                  before_cur = before_cur.next
```

```
        head = head.next

        # clean fronts of partitions
        before = before.next
        after = after.next

        # join the partitions together
        if before is None:
              return after  # there were no values before

        before_cur.next = after
        return before
```

**10.**
```
def my_func(head):
    # Handle length 0 and 1 case
    if head is None or head.next is None:
          return

    # Handle length 2 case
    if head.next.next is None:
          return head.next

    # Get length of list
    curr = head
    length = 0
    while curr is not None:
          length += 1
          curr = curr.next

    curr = head
    if length % 2 == 1:
          iter_range = (length // 2) -1  # one before middle of
list
    else:
          iter_range = length - 3  # third-to-last item

    for i in range(iter_range):
```

```
            curr = curr.next

        curr.next = curr.next.next
    return head
```

**11.**
```
def ll_to_rev_arr(head):
    if head is None:
        return []

    rest = ll_to_rev_arr(head.next)
    return rest + [head.val]
```

**12.**
```
def is_sorted(head):
    # The empty list is sorted
    if head is None:
        return True

    # A list with a single item is sorted
    if head.next is None:
        return True

    # A case where the items are not sorted
    if head.val > head.next.val:
        return False

    # Check the rest of the list
    return is_sorted(head.next)
```

**13.**
```
class Node:
    def __init__(self):
        self.next = None
        self.prev = None
        self.val = None
    def add(head, value):
        if (head is None):
            head = Node()
            head.val = value
        else:
```

```
            curr = head
            while (curr.next is not None):
                  curr = curr.next
            newNode = Node()
            newNode.prev = curr
            newNode.val = value
            curr.next = newNode
def remove(head, value):
      if (head is None):
            return -1
      if (head.val == value):
            head = head.next
            return 0
      curr = head
      prev = head
      while (curr is not None):
            if (curr.val == value):
                  prev.next = curr.next
                  if (curr.next is not None):
                        curr.next.prev = prev
            prev = curr
            curr = curr.next
def find(head, value):
      if (head is None):
            return False
      curr = head
      while (curr is not None):
            if (curr.val == value):
                  return True
            curr = curr.next
      return False
```

**14.**
```
class PlateStack:
        def __init__(self):
            self._stack = []

        def add(self, size):
            self._stack.insert(0, size)

        def remove(self):
            assert len(self._stack) > 0
            return self._stack.pop(0)
```

**15.**
```
class PlateStack:
        def __init__(self):
            self._head = None

        def add(self, size):
            if self._head is None:
                self._head = ListNode(size)
                return

            curr = self._head
            while curr.next is not None:
                curr = curr.next

            curr.next = ListNode(size)

        def remove(self):
            assert self._head is not None

            if self._head.next is None:
                retval = self._head.val
                self._head = None
                return retval

            curr = self._head
            while curr.next.next is not None:
                curr = curr.next

            retval = curr.next.val
```

```
            curr.next = None
            return retval
```

**16.**
```
class TicketSoftware:
    def __init__(self):
        # Use a list to store people
        # (Challenge question: are there other implementations
       # that could perform better?)
        self._queue = []

    def add(self, name):
        # Add people to the end of the list
        self._queue.append(name)

    def remove(self):
        # Remove people at the front of the list
        return self._queue.pop(0)
```

**17.** Every recursive function must have at least one base case, and any number of recursive cases. If the base case is missing, then a stack overflow error will likely occur. If the recursive cases are missing, then the function will be unable to break the problem down.

**18.**
```
def count_occur(num1, num2):
    if (num1 <= num2):
        if num1 == num2:
            return 1
        return 0
    else:
        num_digits = len(str(num2))
        if (num1 % (10 ** num_digits) == num2):
           return 1 + count_occur(num1 // (10 ** num_digits),
           num2)
        return count_occur(num1 // 10, num2)
```

**19.** One possible solution:

```
def extract_less_than(root, val):
    if root == None:
        return []
    if root.val % val == 0:
        return [root.val] + extract_less_than(root.left, val) +
               extract_less_than(root.right, val)
    return extract_less_than(root.left,val) +
           extract_less_than(root.right,val)
```

**20.** One possible solution:

```
def productDigits(val):
      if val == 0:
          return 1
      return int(val % 10) * productDigits(int(val/10))
```

**21.**

```
def print_rev(arr):
      if len(arr) == 0:
          return

      print(arr[-1])
      print_rev(arr[:-1])
```

**22.** A tree is a data structure in which a node contains references to a number of children. A binary tree is a more restricted tree in which the nodes can only contain references to at most 2 children. A BST is a binary tree in which the values must be arranged so that all the values to the left of a given node are smaller than the given node and all the values to the right are larger.

**23.**

In-order traversal: 88, 8, 15, 56, 10, 9, 45, 33, 4
Pre-order traversal: 10, 8, 88, 56, 15, 45, 9, 4, 33

Post-order traversal: 88, 15, 56, 8, 9, 33, 4, 45, 10

Breadth-first traversal: 10, 8, 45, 88, 56, 9, 4, 15, 33

**24.**
```
def arr_to_balanced_bst(arr, start=0, end=-1):
            if (start - end) == 0:
               return None

            # this means that we have just begun searching, so we
            # need to sort
            if start == 0 and end == -1:
               arr.sort()
               end = len(arr)

            mid = (start + end) // 2

            root = TreeNode(arr[mid])

            root.left = arr_to_balanced_bst(arr, start=start,
            end=mid)

            root.right = arr_to_balanced_bst(arr, start=mid + 1,
            end=end)

            return root
```

**25.** `IndexError`

**26.** A NoneType error occurs when current.next.next advances the current pointer past the end of the Linked List. This bug occurs whenever the input is a Linked List with an odd number of nodes. To fix the error, the condition in the while loop should be modified to check that neither current nor current.next are None.

**27.**

1. colon missing after MyClass on the first line
2. __init__ missing self parameter
3. compute_product missing self parameter
4. compute_product()'s x should be self.x
5. compute_product()'s self.y should be y
6. compute_product()'s self.prod should be a temporary local variable rather than a class variable (should be prod instead of self.prod)

**28.** Big-O is a way of tracking the worst-case runtime of our code as our inputs get incredibly large (Big-O does not care about very small input cases). Big-O does this by defining a general (not precise) function that helps us determine the average runtime of our function. This can then help us understand if our code can efficiently scale as our inputs get very very large. We can then use this Big-O value and compare it against other algorithms and compare which is better.

**29.** a) O(n^2)
b) O(n)
c) O(n+m) //since we are given two different arrays
d) O(n^2)
e) O(logn)

**30.** Appending to a Python list is amortized O(1) (written O(1)*). This means that in nearly all cases, it is O(1), but in a very small number of times it is O(n). The reasoning behind this is that underlying every Python list is an array of fixed space in memory, let's say of length n. Most of the time, there will be empty space in the array, so the element can simply be appended within the array, but in certain cases, the array will already be full of elements. When that happens, all of

the existing elements plus the new element must be copied into a brand new, larger array, which takes O(n) time. (For those who are interested: this means that appending n elements is O((n-1)(1) + (1)(n)) = O(2n-1), which still simplifies to O(n), but indicates that you're doing almost twice as much work to append n elements as would be expected if append was always O(1).)

**31.** Searching through a BST takes O(log n) time. If the tree is not a BST, then it is O(n).

**32.** Fastest Slowest

O(1) O(logn) O(n) O(n^2) O(n^50) O(2^n)

**33.** The PlateStack class using the array (Question 14) inserts and removes items from the beginning of the array. Inserting at the front requires us to shift every element to the right one before we have the space to insert the item. Likewise, removing requires us to shift every element to the left. Both of these operations are O(n) as our list has no fixed size. Instead, it can grow towards infinity. Instead of inserting/removing from the front of the array, we should instead append elements to the end of the array and pop elements off the end as these are both O(1) operations.

The PlateStack class using the linked list (Question 15) requires us to traverse the entire list to get to the end every time. As our linked list can grow infinitely, this means we roughly traverse n many nodes before doing an O(1) operation/insertion. As a result, our operation is O(n) as the function takes longer the more nodes we have. To fix this, we should instead insert and remove from the head of the list which are both O(1) operations.

**34.** 1- Understand what tasks the program needs to perform

- Understand the requirements of the program, what the input and outputs should look like

2a - Figure out how to do these tasks

- Break down the program into major tasks, then break each of these tasks down into simple operations, and think about how to implement them
- Have a conceptual idea of *how* the program works before starting to code

2b - Write the code

- Write down the code!
- Create stub functions for each part of the program

3 - Make sure the program works properly

- Test your code
- Make sure the program achieves the intended goals from the start
- Use assertions to pinpoint locations where code does not work
- Creating/using testcases to test functionality

**35.** An abstract data type (ADT) is a way of describing specific operations that can be performed on a collection of data. This description does not use concrete data types such as Dictionaries, Arrays, etc. Instead, operations are described generically so it can be implemented in any data structure.

Examples of this are Stacks and Queues. Stacks define an ordered collection of data where items are added and removed from the same side (often called the "top"). This leads to Stacks being a Last-In-First-Out (LIFO) data structure as the last item to be added is the first to be removed. Queues are similar but say items are removed in the order they were added aka First-In-First-Out (FIFO). This is because the first item to go into the queue is the first one to come out (which is exactly the same as queues or lines in real life. The first person to wait is the first person to get helped).

**36.**
```
def reverse_arr(items):
    # We will only ever add and remove items from the end
    stack = []
    for obj in items:
        stack.append(obj)

    retval = []
    for i in range(len(stack)):
        obj = stack.pop()
        retval.append(obj)

    return retval
```

**37.**
```
def print_rev(arr):
    if len(arr) == 0:
        return

    print_rev(arr[1:])
    print(arr[0])
```

**38.** Underlying every set is another very important data structure called a **hash table**. In a hash table, items are stored in “buckets”, each with their own indexes (0, 1, 2…). When you want to insert an item into a hash table, the item is ran through a hash function (in a process known as **hashing**) to get what is known as a hash code. This hash function can be any mathematical function (such as y=x^2), where the idea is that for any value, the output from the hash function is unique enough that there will seldom be input items that produce the same hash code. Once we have an item’s hash code, we can put it into the bucket of the same number. Since for the same input, hash functions should always return the same output, checking if an item is “in” a set is as simple as hashing the value, and seeing if it’s in the corresponding bucket in the hash table, which is done in O(1)* time. (This is also why sets are unordered and unique, because two similar elements can produce wildly different hash codes, and if a key is already in its corresponding bucket, you aren’t going to insert it again!)

**39.** This is creating a linked list recursively! Copy and paste the code into your editor and then run print(thing(10)). Admittedly, I haven't found a huge use for recursive constructors, but it's pretty neat. The attribute names you're familiar with, are just convention and can be flexible - but this is why descriptive and appropriate variable names are important! (By the way, this is what Bennett and Kruse do in their free times, finding the "technically"'s of rules and exploiting them for humor)

**40.** "Sufficiently" - Russ anytime we try to ask