Problem Session 1

Problem 1-1. Asymptotic behavior of functions

For each of the following sets of five functions, order them so that if f_a appears before f_b in your sequence, then $f_a = O(f_b)$. If $f_a = O(f_b)$ and $f_b = O(f_a)$ (meaning f_a and f_b could appear in either order), indicate this by enclosing f_a and f_b in a set with curly braces. For example, if the functions are:

$$f_1 = n, \qquad f_2 = \sqrt{n}, \qquad f_3 = n + \sqrt{n},$$

the correct answers are $(f_2, \{f_1, f_3\})$ or $(f_2, \{f_3, f_1\})$.

a)	**d)**
$f_1 = (\log n)^{2019}$	$f_1 = 2^n$
$f_2 = n^2 \log(n^{2019})$	$f_2 = n^3$
$f_3 = n^3$	$f_3 = \binom{n}{n/2}$
$f_4 = 2.019^n$	$f_4 = n!$
$f_5 = n \log n$	$f_5 = \binom{n}{3}$

Solution:

Recitation 01

a. $(f_1, f_5, f_2, f_3, f_4)$. This order follows directly from the claim in R01 that $(\log n)^a = o(n^b)$ for all positive constants a and b, as well as standard logarithm and exponentiation manipulations.

b. $(\{f_2, f_5\}, f_3, f_1, f_4)$. This order follows from the definition of the binomial coefficient and Stirling's approximation. The trickiest one is $f_3 = \Theta(2^n/\sqrt{n})$ (by repeated use of Stirling), which grows slower than f_1.

Problem 1-2. Given a data structure `D` supporting the four first/last sequence operations: `D.insert_first(x)`, `D.delete_first()`, `D.insert_last(x)`, `D.delete_last()`, each in $O(1)$ time, describe algorithms to implement the following higher-level operations in terms of the lower-level operations. Recall that `delete` operations return the deleted item.

(a) `swap_ends(D)`: Swap the first and last items in the sequence in `D` in $O(1)$ time.

Solution: Swapping the first and last items in the list can be performed by simply deleting both ends in $O(1)$ time, and then inserting them back in the opposite order, also in $O(1)$ time. This algorithm is correct by the definitions of these operations.

```
def swap_ends(D):
    x_first = D.delete_first()
    x_last  = D.delete_last()
    D.insert_first(x_last)
    D.insert_last(x_first)
```

(b) `shift_left(D, k)`: Move the first k items in order to the end of the sequence n `D` in $O(k)$ time. (After, the k^{th} item should be last and the $(k+1)^{\text{st}}$ item should be first.)

base case

Solution: To implement `shift_left(D, 1)`, delete the first item and insert it into the last position in $O(1)$ time. The list maintains the relative ordering of all items in the sequence, except has moved the first item behind all the others, so `shift_left(D, 1)` is correct. Then to implement `shift_left(D, k)`, move the first item to the last position as above, and then recursively call `shift_left(D, k - 1)` until reaching base case `shift_left(D, 1)`. By induction, if `shift_left(D, k - 1)` is correct, moving the first item to the last position restores correctness. `shift_left(D, k)` runs in $O(k)$ time because it makes $O(k)$ recursive calls until reaching the base case, doing constant work per call.

```
def shift_left(D, k):
    if (k < 1) or (k > len(D) - 1):
        return
    x = D.delete_first()
    D.insert_last(x)
    shift_left(D, k - 1)
```

before the start of each iteration/recursion, the loop invariant holds !!!

Problem 1-3. Double-Ended Sequence Operations

A dynamic array can implement a Sequence interface supporting worst-case $O(1)$-time indexing as well as insertion and removal of items at the back of the array in amortized constant time. However, insertion and deletion at the front of a dynamic array are not efficient as every entry must be shifted to maintain the sequential order of entries, taking linear time.

On the other hand, a linked-list data structure can be made to support insertion and removal operations at both ends in worst-case $O(1)$ time (see PS1), but at the expense of linear-time indexing.

Show that we can have the best of both worlds: design a data structure to store a sequence of items that supports **worst-case** $O(1)$-time index lookup, as well as **amortized** $O(1)$-time insertion and removal at both ends. Your data structure should use $O(n)$ space to store n items.

Solution: There are many possible solutions. One solution uses two-stacks to implement the deque, where care needs to be taken when popping from an empty stack, or pushing to a full stack. An alternative approach would be to store the queued items in the middle of an array rather than at the front, leaving a linear number of extra slots at both the beginning and end whenever rebuilding occurs, guaranteeing that linear time rebuilding only occurs once every $\Omega(n)$ operations.

For example, whenever reallocating space to store a sequence of n elements, copy them to the middle of a length $m = 3n$ array. To insert or remove an item to the beginning or end of the sequence, add or remove an element at the appropriate end in constant time. If no free slot is exists during an insertion, at least a linear number of insertions must have happened since the last rebuild, so we can afford to rebuild the array. If removing an item brings the ratio n/m of items to array size to below $1/6$, at least $m/6 = \Omega(n)$ removals must have occurred since the last rebuild, so we can again afford to rebuild the array.

A linear number of operations between expensive linear time rebuilds ensures that each dynamic operation takes at most amortized $O(1)$ time. To support array indexing in constant time, we maintain the index location i of the left-most item in the array and the number of items n stored in the array, both of which can be maintained in worst-case constant time per update. To access the j^{th} item stored in the queue sequence using zero-indexing, confirm that $i + j < n$ and return the item at index $i + j$ of the array container in worst-case constant time.

After the rebuild, there are m/3 items, then the number of items reduce to m/6,
then at least m/3 - m/6 = m/6 removals must have occurred !!! Just simple math

This is the essence of the amortized O(1) time

Problem 1-4. Jen & Berry's

Jen drives her ice cream truck to her local elementary school at recess. All the kids rush to line up in front of her truck. Jen is overwhelmed with the number of students (there are $2n$ of them), so she calls up her associate, Berry, to bring his ice cream truck to help her out. Berry soon arrives and parks at the other end of the line of students. He offers to sell to the last student in line, but the other students revolt in protest: "The last student was last! This is unfair!"

The students decide that the fairest way to remedy the situation would be to have the back half of the line (the n kids furthest from Jen) reverse their order and queue up at Berry's truck, so that the last kid in the original line becomes the last kid in Berry's line, with the $(n+1)^{\text{st}}$ kid in the original line becoming Berry's first customer.

more difficult than I have expected

(a) Given a linked list containing the names of the $2n$ kids, in order of the original line formed in front of Jen's truck (where the first node contains the name of the first kid in line), describe an $O(n)$-time algorithm to modify the linked list to reverse the order of the last half of the list. Your algorithm should not make any new linked list nodes or instantiate any new non-constant-sized data structures during its operation.

Solution: Reverse the order of the last half of the nodes in a list in three stages:

- find the n^{th} node a in the sequence (the end of Jen's line)
- for each node x from the $(n+1)^{\text{st}}$ node b to the $(2n)^{\text{th}}$ node c, change the next pointer of x to point to the node before it in the original sequence
- change the next pointer of a and b to point to c and nothing respectively

Finding the n^{th} node requires traversing next pointers $n-1$ times from the head of the list, which can be done in $O(n)$ time via a simple loop. We can compute n by halving the size of the list (which is guarenteed to be even).

To change the next pointers of the last half of the sequence, we can maintain pointers to the current node x and the node before it x_p, initially b and a respectively. Then, record the node x_n after x, relink x to point to the x_p, the node before x in $O(1)$ time. Then we can change the current node to x_n and the node before it to x, maintaining the desired properties for the next node to relink. Repeating n times, relinks all n nodes in the last half of the sequence in $O(n)$ time.

Lastly, by remembering nodes a, b, and c while the algorithm traverses the list, means that changing the exceptional next pointers at the front and back of the last half of the list takes $O(1)$, leading to an $O(n)$ time algorithm overall.

(b) Write a Python function `reorder_students(L)` that implements your algorithm.

Solution:

```
def reorder_students(L):
    n = len(L) // 2          # find the n-th node
    a = L.head
    for _ in range(n - 1):
        a = a.next
    b = a.next               # relink next pointers of last half
    x_p, x, x_p = a, b
    for _ in range(n):
        x_n = x.next
        x.next = x_p
        x_p, x = x, x_n
    c = x_p
    a.next = c               # relink front and back of last half
    b.next = None
    return
```

use "loop invariant", you can easily
find that after the loop terminates, x_p is the last node.
Just use "loop invariant" to write correct code !!!