Problem Set 1

All parts are due Friday, September 13 at 6PM.

Name: Lydia Yu

Collaborators: Isaac Redlon, Terryn Brunelle

Problem 1-1.

(a) $\lim_{n\to\infty} \frac{n\log(n)}{(\log(n))^{2019}} = \infty$
$\lim_{n\to\infty} \frac{n^2\log(n^{2019})}{n\log(n)} = \infty$
$\lim_{n\to\infty} \frac{n^3}{n^2\log(n^{2019})} = \infty$
$\lim_{n\to\infty} \frac{2.019^n}{n^3} = \infty$
f_1, f_5, f_2, f_3, f_4

(b) $\lim_{n\to\infty} \frac{\log((\log(n)^{6006})}{\log(\log n)} = 6006, \lim_{n\to\infty} \frac{\log(\log n)}{\log((\log(n)^{6006})} = \frac{1}{6006}$
$\lim_{n\to\infty} \frac{\log(6006^{n^{6006}})}{\log(\log n)} = \infty$
$\lim_{n\to\infty} \frac{(\log n)^{\log(n^{6006})}}{\log(6006^{n^{6006}})} = \infty$
$\lim_{n\to\infty} \frac{(\log n)^{\log(n^{6006})}}{n^{6006}\log n} = \infty$
$\{f_2, f_1\}, f_4, f_3, f_5$

(c) $\lim_{n\to\infty} \frac{3^{4n}}{10^n} = \infty$
$\lim_{n\to\infty} \frac{4^{n^3}}{3^{4n}} = \infty$
$\lim_{n\to\infty} \frac{3^{2^n}}{4^{n^3}} = \infty$
$\lim_{n\to\infty} \frac{2^{2^{n+2}}}{3^{2^n}} = \infty$
f_5, f_1, f_4, f_2, f_3

(d) $\lim_{n\to\infty} \frac{\binom{n}{3}}{n^3} = \frac{1}{6}, \lim_{n\to\infty} \frac{n^3}{\binom{n}{3}} = 6$
$\lim_{n\to\infty} \frac{\binom{n}{n/2}}{n^3} = \infty$
$\lim_{n\to\infty} \frac{2^n}{\binom{n}{n/2}} = \infty$
$\lim_{n\to\infty} \frac{n!}{2^n} = \infty$
$\{f_2, f_5\}, f_3, f_1, f_4$

Problem 1-2.

(a) Just as there is a head node that is fixed and points to the first element in the linked list, you could add a tail node that is also fixed in memory so that anything you want to add to the end of the list points to the tail node, and the previous last element of the list would point to this new inserted node. Without this tail node, you would have to go through the entire list to find the last node in the sequence, which would take O(n) time for a list with n elements. The running time of this modified linked list is O(1) because you just find the node that points to the tail (which is fixed, so you always know where it is and therefore you always know where the last node is) and insert the new node so that the old node points to the new node and the new node points to the tail.

(b) Everything except for `delete_last(x)` is already in O(1) time (given that we use the modified linked list from part a). In order for `delete_last(x)` to work, we need to know what the element that comes before the current last element is so that we can make that one the new last element, but right now each node only points to the next element in the list and not the previous. Thus, we should store an additional pointer that points to the previous node in the sequence for each node so that the last item points to what would become the new last item if it were deleted. The running time would now be O(1) because now that each node has pointers in both directions, you know which node is the last one since it points to the tail so you can find it in constant time, and you know which node would become the new last node because the last node points back to it; there is no need to iterate through multiple elements of the lsit. The running time is worst case because every operation will always be completed in O(1) time.

(c) The `insert_first(x)` and `delete_first(x)` normally have to shift every element in the array up or down in order to make space for or replace the empty space of the first element, which would take O(n) time for an array of n elements. To make these operations O(1) time, we need some way to add/delete the first element without shifting the entire array. Just as the dynamic array adds extra space to the end that satisfies the fill ratio whenever the array is full, we could modify it so that it also adds space at the beginning whenever it is full. Thus, when the array is filled, a new empty array would be created so that there are empty slots are the beginning and the end of the array. The fill ratio would be satisfied such that the number of empty slots at the beginning is the same as the number of empty slots at the end (for example, if the ratio is 1/2, then 1/4 of the array would be empty space at the beginning and 1/4 would be empty space at the end). The data from the old array is copied into the newer, larger one, which has O(n) running time because you would first count from the empty spaces at the beginning until you have the required number of starting empty slots, and then you would add in all the old data. However, resizing the array doesn't happen as frequently as the individual operations do (it happens only, on average, once every n times), so the O(n) run time that is required for resizing is spread over an n number of

individual operations that each take O(1) time (so a total of n*O(1)) On average, the running time would be O(1), and this is therefore an example of amortized running times.

Problem 1-3.

(a)
```
x = 0
while x < k+1:
    temp = delete_first()
    insert_last(temp)
    x = k+1
```

This algorithm deletes the first element in the sequence and stores the returned value in a variable, then inserts that variable at the end of the sequence. This is repeated for the first k elements. Since `delete_first()` and `insert_last(temp)` each take O(1) time and are repeated k times, the overall running time for this algorithm is O(k).

Induction on the number of items to be moved:
Inductive hypothesis: the proposed algorithm moves the first k items in order to the end of the sequence.
Base case: the algorithm deletes the first item and stores it into a variable, then inserts that at the end of the sequence.
We can assume that this holds true for any k items in the list that have to be moved. We know that this also works for k+1 items because we've already moved k of them successfully, so we just need to move one more using the same process.

(b)
```
first = delete_first()
last = delete_last()
insert_last(first)
insert_first(last)
```

This algorithm stores the first and last elements of the sequence each in their own variable after deleting them. It then inserts the values stores in these variables in their new respective locations. These four operations only take O(1) time each, and there are no loops required, so the overall running time of the algorithm is O(1). Given any list of length greater than 1, the algorithm is always able to use the above four functions to temporarily store the first and the last items, and then insert them into their new locations.

Problem 1-4.

(a) A possible algorithm is one that first finds the midpoint node (the node that would represent the last kid in the first half of the original line) by iterating through the first n nodes in the original linked list, given that there are 2n kids total. We then store this middle node into a variable called `middle`. Now that we know where the end of the first line is, we know what the reverse order of the second line of kids is because `middle` points to what will become the last kid in the second line. We want to reverse the directions that the original pointers for the nodes in the second line are pointing in so that in the end, what was originally directly after `middle` is now the last element in the linked list, and `middle` is now pointing to the original last element of the list. This is done by iterating through all of the second-half nodes and setting their pointers to the opposite node (except for the first node right after `middle`, which will become the last node in the list and will not point to anything else) until you've reached the end of the original list and cannot iterate through any more nodes because they don't exist. Now, we reverse the pointer directions of the nodes that were not taken care of in the loop do to null-checking (to make sure that we don't end up trying to get a next node that doesn't actually exist in the loop) and set `middle` to point to the original last node. We will need three temporary variables to store the chunk of the list that we are currently looking at as we iterate through the list because otherwise, when we reverse the direction of the pointer, we lose track of which node to move on to next. Having a third stored node directs us to the next node to shift our frame of reference to. The running time of this algorithm is O(n): finding `middle` takes O(n) time because you must iterate through n nodes out of a total of 2n to get to the middle. Setting temporary variables are each O(1). Each step within the while loop also takes O(1) time, but these are all repeated n times for the n kids in the second half of the line for a total of O(k). The last two steps each take O(1) time. Therefore, overall, the running time is O(k), since we have 2*O(k) which is the same as O(k) asymptotically.

We can always find the middle node of the list because we always have access to the size of the list, and we just divide the size by 2 to get the midpoint. Now, we induct on the length of the second half of the list whose node pointers must be reversed:
Inductive hypothesis: the algorithm takes the second half of a linked list of length 2n (second half of length n) and reverses the directions of its pointers, and then points `middle` to the original last node.
Base case: if the second half of the list only has one node, there is no need to reverse any pointers, and the middle node is already pointing at the correct node. Inductive step: we can assume that this hypothesis holds true for any second half of a linked list with length n. We know that this would also be true if another node were added and the second half of the list has length n+1 because we just follow the same steps of setting the pointer of the new node to point towards the old last node, and then having `middle` point to this new node.

(b) Submit your implementation to `alg.mit.edu`.