Problem Set 1

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

Name: Diego Escobedo

Collaborators: Noah Lee, Nick Baginski, John Rao

Problem 1-1.

(a) The correct ordering is $(f_1, f_5, f_2, f_3, f_4)$. We proved in recitation that $log(n)^a = O(n^b)$ for all positive constants a and b. Therefore, we can set a to 2019 and b to 1 and determine that f_1 is $O(n)$. For f_2, we can get rid of the 2019 by pulling it out and ignoring it, leaving $O(n^2 * log(n))$. The rest are self explanatory, and ordering them is trivial.

(b) The correct ordering is $(\{f_2, f_1\}, f_4, f_3, f_5)$. f_2 is already simplified. f_1 can be simplified to $6006log(log(n))$, and if we ignore the constant, it is the same as f_2. For f_4, we can pull out the n^{6006} and then the remaining log is simply a constant that we can ignore, meaning it is $O(n^{6006})$. f_3 is already simplified. The hard one in this case is figuring out where to place f_5. We can use the technique of taking limits that we learned in recitation to get:

$$\lim_{n\to\infty} \frac{(log(n))^{log(n)}}{f_x}$$

where f_x can be any of the functions we have already determined above. We can make our analysis even simpler by taking the log of the whole thing, leaving us with

$$\lim_{n\to\infty} log(\frac{(log(n))^{log(n)}}{f_x})$$

which can be further simplified to

$$\lim_{n\to\infty} log(n) * log(log(n)) - log(f_x)$$

From here, we can simply plug in all the different f_xs. If the limit goes to infinity, then f_x is $O(f_5)$, and if it goes to negative infinity, then f_5 is $O(f_x)$. By doing this, we can see that f_5 is the fastest growing.

(c) The correct ordering is $(f_5, f_1, f_4, f_2, f_3)$. Because all of the functions in this section are exponential functions, we have a pretty unique situation. Unlike polynomial functions, the constants in front of our ns do matter with this. That being said, it might be easier to convert all of the functions into ones of the same base. f_5 would thus become $2^{log_{10}(2)*n}$, or roughly $2^{0.3*n}$. f_2 becomes $2^{log_3(2)*4*n}$, or roughly $2^{2.52*n}$. f_4 is the only one with a polynomial function in its exponent, meaning it has to be above f_5 and f_1 because those are linear, but below the rest because they have exponential functions in the exponents. f_2 can be rewritten as $2^{.63*2^n}$, whereas f_2 can essentially be expressed as 2^{4*2^n}, meaning that f_2 grows the fastest.

(d) The correct ordering is $(\{f_5, f_2\}, f_3, f_1, f_4)$. f_5 can be simplified to be $n*(n-1)*(n-2)$ which if we remove the trailing terms is simply n^3, meaning it is on the same level as f_2. f_3 can be bounded by $O(\frac{\sqrt{n}}{n}*2^n)$, which is going to be less that 2^n. f_4 is obviously going to grow the fastest.

Problem 1-2.

(a) We could edit the implementation of linked lists to make sure that the head of the linked list has a slightly different structure than the rest of the nodes. This node, instead of having the two fields of node.item and node.next which every node has, would have: list.head.item, list.head.next, and list.head.tail, which is a pointer that points to the last element. We could go editing this field as we construct the linked list (since constructing the list happens in O(n) time anyway this tacks on an extra constant time operation to it that will save us time later... maybe this is amortizing?). Thus once we have finished constructing the list we have this structure set up that allows us to access the last element easily. To actually insert something at the end, we could simply do the following:

```
new_node = node(item = x, pointer = null) #instantiating new node
list.head.tail.next = new_node #changing the last element's pointer
list.head.tail = new_node #redefining tail as the new node
```

Further, some concerns might be raised about what happens when we insert a node at the start. In that case, all we need to do is store the old head's item and next in a temporary node (which is constant space), define the new head (including the old head's tail pointer), and have the new head's next point to the temp node (which is now #2 in the list).

(b) This is a simple extension of what we did above. Now, our standard node has 3 fields instead of two: node.previous, node.item, and node.next. If we update these pointers as we are building the list (an operation that will always be O(n)), then it should take no extra time to create this structure. The head node would have 4 fields instead of 3 (with the extra field still being list.head.tail), but everything else would work the same, which means that insert first, delete first, and insert last would all be O(1). For delete last, the procedure is simple. We access the last node in constant time using list.head.tail, use constant space to copy the address of list.head.tail.previous onto a variable x, change list.head.tail to x, and finally access our new tail and change its next field to null (because it is now the last node). This allows us to complete all the each of the four first/last sequence operations in constant time. These operation running times are all worst case, since we are just tacking on constant time operation to the build() operation (which is O(n)), which means it takes no extra time. An amortized operation can, in the absolute worst case, still be O(n) (for example in dynamic arrays if you never end up using the extra space), whereas this will never be O(n).

(c) It is very easy to generalize the concept of amortizing extra space in the back to extra space in the front. In the front we simply look at the fill ratio and allocate Θ(n) extra space at the end to reduce the fill ratio back to 0.5. We can do the same thing for extra space in the back, where we start off with, for example, one extra space. If we use the insert first function, then we can simply allocate n extra spaces before the new first element. This is the exact same concept as amortizing for the front, so it is not a worst-case running time.

Problem 1-3.

(a) In pseudocode:

```
def shift_left(k)
    data = DataStructure()
    for i in range(k): # O(k) operation
        temp = data.delete_first() #O(1) operation
        data.insert_last(temp) #O(1) operation
```

This is correct because it will delete the first element, and append it to the back, k times. This ensures that the order is preserved. The running time is bounded from below by O(k) because we have to run the loop k times to make sure we can shift it completely, and is bounded by above by O(k) because each operation in the loop is constant time.

(b) In pseudocode:

```
def swap_ends():
    data = DataStructure()
    old_first = data.delete_first() #O(1) operation
    old_last = data.delete_last() #O(1) operation
    data.insert_first(old_last) #O(1) operation
    data.insert_last(old_first) #O(1) operation
```

This is correct because it will delete the first element, store it temporarily, then delete the last element, and store it in a temp. Then, it will add the old last element as the first element and vice versa. This ensures that the desired order is correct. We only use O(1) operations so the run-time is constant.

Problem 1-4.

(a) The solution to this problem is a little tricky, but it can be done in O(n) time. What we need to start off by doing is iterating through the list until we get to the nth element, which is our 'midpoint'. We need to keep this stored in a temp variable because we still need to edit its pointer to point to the last element once we are done. The idea for the rest of the points is that we are going to 'reverse' the pointers, which can be done by doing temp storage and a while loop to continue iterating through the list. We can instantiate a 'current' variable, a 'next' variable, and a 'temp' variable. The current will represent the node we are currently reversing, next will be the next node in the new ordering, and temp will hold the original next node. The first thing we do is change temp to the current node's original next node. Then we change the node.next pointer to next, which as we mentioned is the next one in the new sequence. Then we redefine next to be current, because the current node will be the next one once we iterate from the sequence again. Finally, we redefine current to be temp, which is the next node we will reverse. We have an exit condition where if current is ever None (which will happen in the last element, because for the last element current.next is None, and we set current = temp, which was already previously defined to be current.next), we will leave. Then, because the last next is going to be the last n+1th element in the new ordering, we set middle.next = next. Then, we return the list. As we can see, this all happened in a single iteration. We did not use any for loops with indices, we simply used the .next to redefine the node of interest and a while loop to go as far as we need. This means that the list of length n will take O(n) time to reverse. This algorithm will be correct because, as explained above, we are reversing the node.next pointers while making sure that there are no loops and that the middle node connects to the originally last node.

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